Zero-Knowledge Proofs for Self-Sovereign Identities in Decentralized Services

Xavier Salleras

December 2022

TESI DOCTORAL UPF / 2022
Supervisor: Vanesa Daza
Department of Information and Communication Technologies

To my family.

Thanks

There are some roads long enough to make you feel that you are rolling them alone. But suddenly you realize that you would not have come that far without others' help. So I need to thank a few people, and this is why I feel that this page is the most important one in this thesis.

First, I need to thank Vanesa, for accepting me as a student, for her advice, and for her guidance during all these years. And for the feedback on my undergraduate thesis: it had more impact on me than you may think.

Then, for the love, for the patience, and for giving me the opportunity to study, I thank my parents, Maria and Joan. I also thank my brother, David, for being a reflection of whom I always have wanted to be. And I want to thank my sister, Montse, for taking care of me in my darkest moments. Plus, big thanks to Mia and Lasc, for being such a nice niece and nephew.

Somehow, this trip started way before I enrolled in the PhD. Probably I need to look back to the moment I moved to Barcelona to start my undergraduate degree. So I would like to thank the good friends I keep from that epoch, for what we learned back in time, and for the moments we keep spending nowadays. Thank you, Ariadna, Tula, and Víctor.

Discussing cryptography, having fun, sharing our frustrations, and eating ramen in summer at 34ºC is part of a PhD. And you need colleagues for that. That is why I want to thank the people I had the pleasure to meet in office 55.210: Alexandros, Arantxa, Conor, Federico F., Federico M., Pablo, Rasoul, Sergi, and Zaira.

Special thanks to Javier and Marta, for all the productive conversations and moments we spent during this last year. I learned a lot from you.

And finally, I need to thank one last person. For so many years together. For growing together and for being there. Both of us are closing a chapter, but many more are yet to come. Thank you, Claudia.

Abstract

Telecommunication systems such as mobile communications are evolving over time, and thanks to that, many digital services have arisen in the last decade. Usually, these services ask their users to provide sensitive information in order to identify them, and the service provider is the one managing all these data. From the point of view of the user, in most cases, there is no choice other than trusting the service provider, for what concerns the security of the service they provide, and the privacy of the data they request.

In that regard, this thesis studies and proposes a novel self-sovereign identity system based on Zero-Knowledge Proofs and Blockchain technologies, where the management of personal data of individuals using digital services is carried out by themselves, and their rights to use such services are verified in a decentralized manner. Furthermore, we implement a library that, even when Zero-Knowledge Proofs require high computing resources, allows us to compute these proofs in embedded systems. Like this, we demonstrate that deploying our self-sovereign identity system in a wide variety of devices is feasible.

Resum

Els sistemes de telecomunicacions tals com les comunicacions mòbils estan evolucionant amb el temps, i gràcies a això, molts serveis digitals han aparegut en l'última dècada. Normalment, aquests serveis demanen als seus usuaris facilitar dades sensibles amb l'objectiu d'identificar-los, i és el proveïdor del servei l'encarregat de gestionar-les. Des del punt de vista de l'usuari, en molts casos, no hi ha altra elecció que confiar en el proveïdor del servei, pel que fa a la seguretat del servei que proporciona, i la privacitat de les dades que demana.

En aquesta línia, aquesta tesi estudia i proposa un nou sistema d'identitat autogovernada basat en proves de coneixement zero i en tecnologies de cadena de blocs, on la gestió de les dades personals dels usuaris de serveis digitals és portada a terme per ells mateixos, i els seus drets a utilitzar tals serveis són verificats de forma descentralitzada. Addicionalment, implementem una llibreria que, tot i que les proves de coneixement zero requereixin recursos computacionals elevats, ens permet computar aquestes proves en sistemes encastats. D'aquesta manera, demostrem que desplegar el nostre sistema d'identitat autogovernada en una gran varietat de dispositius és viable.

Contents

Chapter 1

Introduction

Nowadays, there is a plethora of services that are provided and paid for online, like video streaming subscriptions, car or parking sharing, purchasing tickets for events, etc. Digital services usually issue tokens directly related to the identities of their users after registering on their platform, and the users need to authenticate using the same credentials each time they are willing to use the service. Likewise, when using in-person services like going to a concert, after paying for this service the user usually gets a ticket which proves that they have the right to use that service. In both scenarios, the main concerns are the centralization of the systems, and that they do not ensure customers' privacy. The involved Service Providers (SPs) are authorities that offer services and handle private data about users, and they need to be trusted in the fact that the acquired data will not be misused.

For such a reason, it became of paramount importance to have robust authentication protocols, able to ensure that users meet the requirements wanted by the SPs, but providing at the same time a private-by-design architecture. Having these needs in mind, Self-Sovereign Identity (SSI) systems [1] were introduced: systems allowing users to manage their identities in a fully transparent manner. Technically speaking, Zero-Knowledge Proofs (ZKPs) [2] are one of the main requirements for SSI systems to achieve their features. ZKPs are cryptographic primitives allowing a party to prove knowledge of some secret information, without leaking anything beyond the fact that such information is true. However, there are still some open problems to address, which we tackle in this thesis.

1.1 Open Problems

SSI systems [3] serve the purpose where a user can prove *ownership* of the right to use a specific service, while at the very same time, they are able to prove that their right is *valid*, i.e. the SP still accepts such a right, which has not expired or has not been revoked after being granted. While the first approach can be achieved by combining a digital signature signed by the SP and being verified into a ZKP, the privacy-preserving properties of this scheme make it difficult to revoke a right later on.

On the other hand, *unlinkability* is a requirement as well: the SP should not be able to link any user activity with other activities previously done. Otherwise, information about the user would be leaked, and user profiling techniques could be applied.

Moreover, another important feature is *decentralization* [4]: being able to acquire rights and use them by means of an environment that can be verified by all the involved parties, without having to rely on a central authority. In other words, integrating our solution into a Blockchain would lead to a decentralized and transparent protocol.

Finally, the user should have full control over the information about them that is shared and should be able to accept or deny each request for personal information. We define the different portions of personal information as provable *attributes*, which combined define a *license*, that can be seen as the right of someone to access a given service. We wrap up these properties as follows:

- **Proof of Ownership:** a user of a service is able to prove ownership of a license that allows them to use such a service, without leaking any information about them.
- **Unlinkability:** neither an eavesdropper adversary nor the SP can link any activity of their users with other activities done in the network.
- **Attribute Blinding:** the user is capable of deciding which information they want to leak to the SP, blinding the value and providing only the desired information.
- **Proof of Validity:** users can prove ownership of a valid license, that has been granted within a specific context (e.g. within a given Blockchain), that has not expired, and that has not been revoked.
- **Decentralized Nullification:** nullifying a license means that once used, it should not be spendable again. Decentralizing this process allows for a

public nullification of the license, so it cannot be used in other contexts as well. Even a malicious SP could not impersonate a user after receiving a valid proof, by resending it.

Designing the above features becomes a challenging task. Plus, there are several side-constraints to consider as well: thanks to 5G communications [5], which optimize the network for different devices and services, we see fast adoption of static internet-connected devices [6–8] like pollution sensors, traffic lights, surveillance cameras, etc. Moreover, other mobile Internet of Things (IoT) [9] devices, like autonomous cars, will be soon populating the cities. If we take into account all the computers, smartphones, smartwatches, etc., we observe how the density of devices is achieving high numbers. This can be translated into a growing amount of private information exchanged over the network, and in that regard, the need for SSI systems makes even more sense. For such a reason, there is a real need for cryptographic protocols able to guarantee users' privacy when using digital services, and it becomes important to design them in a feasible way to be deployed in devices with low computing resources, such as IoT devices.

1.2 Research Outcomes

In this thesis, we achieved the following research outcomes. First of all, we analyzed the state of the art on the cryptographic schemes used nowadays for authentication purposes, especially on IoT devices. This helped us to understand the insecurities that live in many digital systems, and the need for a common authentication framework designed with privacy in mind.

Considering the privacy properties of ZKPs and how some solutions use their features in specific privacy-preserving applications, we proved how they can be executed on a wide variety of devices, even on those with low computing resources. To do so, we implemented from scratch `ZPiE`, a ZKP library able to compute different kinds of ZKPs using computers, smartphones, and IoT devices. Upon success, we had the confirmation that protocols based on ZKPs can be executed on any kind of device if optimized properly.

Having the previous outcome reached, we designed an SSI system, `SANS`, whose properties are depicted in Table 1.1. `SANS`, which uses ZKPs as its backbone, focuses its application scenario in 5G environments, with scalability to any other digital service under some requirements. However, such a protocol lacks some privacy features that can be overcome with decentralization.

To overcome these privacy problems, we designed `FORT`, a novel protocol integrated into the Ethereum Blockchain, for achieving such features. It is an SSI system that relies on Non-Fungible Tokens (NFTs): assets uniquely identifiable that contain some specific information. To be precise, it provides an approach for proving that a given license, represented as an NFT, has been acquired into a given context, in this case, the Ethereum network. Plus, it integrates a way to blind the attributes that identify the user, so only the desired information is leaked. Nevertheless, we stated some problems that remained open: the transparent nature of the Ethereum Blockchain allows adversaries to apply profiling techniques on users of our solution. This leads to an *incomplete unlinkability* property. Plus, even when Ethereum is a decentralized network, the application of `FORT` followed a centralized approach. We tackled these problems, and we provide a fully-private-by-design and fully-decentralized protocol, that complies with all the properties above stated. In particular, we designed `Citadel`, a protocol that provides *complete unlinkability* thanks to the private nature of the Blockchain where it is integrated, Dusk Network. It also provides a *complete Proof of Validity*, where SPs can revoke licenses, and set expiration dates. Finally, the *decentralized nullification* of `Citadel` prevents misuses of the licenses, and a more efficient and optimized protocol, as we will detail in Chapter 7.

Table 1.1: Comparison of our Self-Sovereign solutions.

Property	**SANS**	**FORT**	**Citadel**
Proof of Ownership	✓	✓	✓
Attribute Blinding	✗	✓	✓
Unlinkability	Complete	Incomplete	Complete
Proof of Validity	✗	Incomplete	Complete
Nullification	Centralized	Centralized	Decentralized

1.3 Contributions

Now, we outline all the contributions presented in this thesis, and how they overcome the research outcomes. They can be summarized as follows:

- **LASER.** A lightweight and general solution based on a one-message protocol, which guarantees the integrity and validity of the authentication in Remote Keyless Entry (RKE) systems, protecting the communication against the well-known jamming-and-replay and relay attacks, without using complex cryptographic schemes. Moreover, we also adapt our protocol

for Passive RKE (PRKE) systems. We also provide a novel frequency-hopping-based approach that mitigates denial-of-service attacks. This research led to the following publication:

Vanesa Daza, Xavier Salleras. *LASER: Lightweight And SEcure Remote keyless entry protocol.* Proceedings of SECRYPT'19. 2019.

- **ZPiE.** A C library intended to create ZKP applications to be executed in embedded systems. Its main feature is portability: it can be compiled, executed, and used out-of-the-box in a wide variety of devices. Moreover, our proof-of-concept has been proven to work smoothly on different devices with limited resources, which can execute the ZKP authentication protocols introduced later in this thesis. This research led to the following publication:

  Xavier Salleras, Vanesa Daza. *ZPiE: Zero-knowledge Proofs in Embedded systems.* Special issue *Recent Advances in Security, Privacy, and Applied Cryptography* of the journal Mathematics (2021).

- **SANS.** A novel SSI scheme for the 5G network slicing. We grant users full control over their data: we introduce an approach to allow a user to prove his right to access a specific service without leaking any information about them. Our protocol is scalable and can be taken as a framework for improving related technologies in similar scenarios, like authentication in the 5G Radio Access Network (RAN) or other wireless networks and services. This research led to the following publication:

  Xavier Salleras, Vanesa Daza. *SANS: Self-sovereign Authentication for Network Slices.* Special issue *Trustworthy Networking for Beyond 5G Networks 2020* of the journal Security and Communication Networks.

- **FORT.** A decentralized system that allows customers to prove their right to use specific services (either online or in-person) without revealing sensitive information. To achieve decentralization we propose a solution where all the data is handled by a Blockchain. We describe and uniquely identify users' rights using NFTs, and possession of these rights is demonstrated

by using ZKPs. This research led to the following publication:

Xavier Salleras, Sergi Rovira, Vanesa Daza. *FORT: Right-Proving and Attribute-Blinding Self-Sovereign Authentication.* Special issue *Advances in Blockchain Technology* of the journal Mathematics (2022).

- **Citadel.** Even when ZKPs do not leak any information about the rights, the NFTs in `FORT` are stored as public values linked to known accounts, and thus, they can be traced. Here, we design a native privacy-preserving NFT model for the Dusk Network Blockchain, and on top of it, we deploy `Citadel`: our novel full-privacy-preserving SSI system, where the rights of the users are privately stored on the Dusk Network Blockchain, and users can prove their ownership in a fully private manner. This research led to the following manuscript:

  Xavier Salleras. *Citadel: Self-Sovereign Identities on Dusk Network.*

After the work done in `LASER`, where we study several security threats in IoT devices, a problem to solve was identified: how to execute complex cryptographic primitives in devices with low computing resources. As ZKPs are an essential tool to build SSI systems, we worked on `ZPiE`. Thanks to this library, building `SANS` was possible. Later, the decentralized approach introduced in `FORT`, which takes `SANS` as its backbone, was published. And finally, the last hanging flaws of `FORT` have been fixed with `Citadel`.

Furthermore, the following software has been developed out of the publications above introduced:

- The `ZPiE` library[1], supporting the following Zero-Knowledge schemes on top of the elliptic curves BN128 and BLS12-381:
  - zk-SNARKs for arithmetic circuits, in particular, the Groth'16 scheme. `ZPiE` also includes the following arithmetic circuits:
    - EdDSA signature verification over Baby JubJub elliptic curve and BN128.
    - MiMC-7 hash function (BN128 order).

[1] https://github.com/xevisalle/zpie

  - Bulletproofs, supporting both range proofs and aggregated range proofs.

- The CryptoolZ library[2], containing some cryptographic primitives coded in C, and intended to be used (but not limited to) in Zero-Knowledge applications. In particular, we implemented:
  - EdDSA signature algorithm over Baby JubJub elliptic curve and BN128.
  - MiMC-7 hash function (BN128 order).
- The `Citadel` Rust crate[3], which allows to deploy and use `Citadel`.

[2] https://github.com/xevisalle/cryptoolz
[3] https://github.com/dusk-network/citadel

Chapter 2

Preliminaries

In this section, we introduce the preliminaries of the solutions introduced in this thesis. We first introduce the notation used in this document, plus some mathematical background and several cryptographic primitives. Later, we introduce Blockchain technologies and their applications to IoT, and finally, we focus on the details and the specific use cases of smart contracts.

2.1 Notation

Throughout the document, we will refer to parties and number spaces using an uppercase calligraphic font, e.g. prover $\mathcal{P}$, or message space $\mathcal{M}$. We will use standard lowercase letters to refer to integers, and standard uppercase letters to refer to points, e.g. integer m, or the point P. The blackboard bold font will be used to represent algebraic structures, e.g. a finite field $\mathbb{F}$. We will use sans serif font for abstract elements of a protocol, e.g. a transaction tx.

2.2 On the Security of Cryptographic Schemes

In this section, we start by giving a definition of security [10, Chapter 2.2.1] for the sake of completeness. We later move to an overview of the main security assumption used throughout the thesis.

2.2.1 Security Definition

We consider an adversary $\mathcal{A}$ as a malicious player trying to attack a given cryptographic scheme, either by compromising the privacy of the data, its integrity, or denying the availability of the system. The chances of success of $\mathcal{A}$ in breaking a system is measured by its advantage $\mathsf{Adv}\mathcal{A}$. This is the difference between the adversary's probability of breaking the system and the probability that the system could be broken by trying at random.

We measure the security of a given scheme by its computational cost against the quantified security we can expect from such a scheme, where the latter is assessed by the security parameter $\lambda \in \mathbb{N}$. Using this setting, we say that a function $f : \mathbb{N} \to [0, 1]$ is *negligible* if

$$f(\lambda) = O\left(\frac{1}{p(\lambda)}\right) \tag{2.1}$$

for any polynomial p, and denote it using $\mathsf{negl}(\lambda)$. We consider that a scheme is secure if the advantage of an adversary is $\mathsf{negl}(\lambda)$. Furthermore, and for our purposes, we distinguish between the following adversaries:

- *Computational:* adversary running in probabilistic polynomial time (PPT).
- *Perfect:* adversary whose advantage is 0.
- *Eavesdropper:* passive listener on a channel.

2.2.2 The Discrete Logarithm Problem

We now define the Discrete Logarithm Problem (DLP), which we will directly use in our constructions in this thesis. Let $\mathbb{G}$ be a cyclic group of order N, where $\forall Q \in \mathbb{G}$ there exists a unique value $a \in \mathbb{Z}_N$ such that $aP = Q$. Here, a is the discrete logarithm of Q respecting P. Knowing the pair (P, Q), the DLP consists in finding a. Roughly speaking, if λ is chosen carefully, the DLP is considered to be hard to solve. We formalize it as follows:

Definition 1 (Discrete Logarithm Problem)**.** *We assume that the DLP related to a setting* $\mathsf{Gen}(1^\lambda)$ *holds iff, for any PPT adversary* $\mathcal{A}$*, the following probability is satisfied:*

$$\mathsf{Pr}\left[(\mathbb{G}, N, P) \leftarrow \mathsf{Gen}(1^\lambda); Q \leftarrow \mathbb{G}; a \leftarrow \mathcal{A}(\mathbb{G}, N, P, Q) : Q = aP\right] \leq \mathsf{negl}(\lambda)$$

2.3 Elliptic Curves

One of the main elements required for constructing the ZKP schemes that we will use in this work are elliptic curves. An elliptic curve is defined as follows.

Definition 2 (Elliptic curve)**.** *Let E be an algebraic curve defined by the projective solutions of the equation*

$$Y^2 = X^3 + aX + b,$$

for some $a, b \in \mathbb{F}_q$. If $4a^3 - 27b^2 \neq 0$, we call E an elliptic curve over $\mathbb{F}_q$*, and denote this by $E/\mathbb{F}_q$.*

We are particularly interested in the so-called pairing-friendly elliptic curves, and we describe them now. Let E be an elliptic curve over a finite field $\mathbb{F}_q$, where q is a prime number. We have the bilinear groups $(\mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T)$ of prime order p, and a pairing

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

being a bilinear map. As the map e is bilinear, the following relation is satisfied

$$e(aP, bQ) = e(P, Q)^{ab},$$

for any $P, Q \in E$.

We are particularly interested in two elliptic curves, needed later on to describe the transaction model of Dusk Network. They are the BLS12-381 [11] and the Jubjub [12] elliptic curves. Let p, q be two specific prime numbers of 255 and 381 bits, respectively. The curve BLS12-381 is defined over $\mathbb{F}_q$ by the equation

$$E : Y^2 = X^3 + 4,$$

and has different subgroups $\mathbb{G}_1, \mathbb{G}_2$ such that $\#\mathbb{G}_1 = \#\mathbb{G}_2 = p$. This curve is pairing-friendly, meaning that pairings can be efficiently computed. On the other hand, the Jubjub curve is defined by the equation

$$J : -X^2 + Y^2 = 1 + \left(-\frac{10240}{10241}\right) X^2 Y^2,$$

over $\mathbb{F}_p$ (it is important to recall that p is the order of a prime subgroup of the BLS12-381). We define a subgroup $\mathbb{J}$ whose order t is a 252-bit prime. Throughout the document, we will mainly use scalar values from the field $\mathbb{F}_t$, and elements from $\mathbb{J}$.

Throughout the document we will use other settings as well, in particular, the Barreto-Naehrig elliptic curve called BN128 [13], along with the Baby JubJub [14]. We will specify it when doing so.

2.4 Commitments and Hash Functions

A *commitment scheme* allows a party to commit to a secret value v, to be revealed at a later time. A commitment scheme works like a safe-deposit box, in the following sense. The party that wishes to make a commitment puts the value v inside the box, and lock it. They keep the key, but the box is kept in a public place. The commitment *hides* the value inside, until the owner decides to use the key and open it. At the same time, the commitment *binds* the value, ensuring that the owner cannot change it after committing. In particular, the commitment scheme that we will use in this thesis uses the specific *perfect hiding* and *computational binding* properties, which we define now.

Definition 3 (Perfect Hiding)**.** *A commitment scheme is perfectly hiding iff, for any computationally unbounded eavesdropper adversary $\mathcal{A}$,*

$$\mathsf{Pr}\left[\begin{array}{c} ck \leftarrow \mathsf{Setup}(1^\lambda); \\ m_0, m_1 \leftarrow \mathcal{A}(1^\lambda, ck); b \leftarrow \{0,1\}; \\ r \leftarrow \mathcal{R}; c \leftarrow \mathsf{Com}_{ck}(m_b; r); \tilde{b} \leftarrow \mathcal{A}(c) \end{array} : \tilde{b} = b \right] = \frac{1}{2}$$

Definition 4 (Computational Binding)**.** *A commitment scheme is computationally binding iff, for any PPT adversary $\mathcal{A}$,*

$$\mathsf{Pr}\left[\begin{array}{c} ck \leftarrow \mathsf{Setup}(1^\lambda); \\ m_0, m_1, r_0, r_1 \leftarrow \mathcal{A}(1^\lambda, ck) \end{array} : \begin{array}{c} m_0 \neq m_1; \\ \mathsf{Com}_{ck}(m_0; r_0) = \mathsf{Com}_{ck}(m_1; r_1) \end{array} \right] \leq \mathsf{negl}(\lambda)$$

We are particularly interested in Non-interactive Commitment Schemes, defined as follows:

Definition 5 (Non-interactive Commitment)**.** *A non-interactive commitment scheme consists of a tuple of algorithms (Setup, Commit, Open). The Setup algorithm $ck \leftarrow \mathsf{Setup}(1^\lambda)$ generates a public commitment key ck given the security parameter λ. Given the public commitment key ck, the commitment algorithm Commit defines a function $\mathsf{Com}_{ck} : \mathcal{M} \times \mathcal{R} \rightarrow \mathcal{C}$ for a message space $\mathcal{M}$, a randomness space $\mathcal{R}$ and a commitment space $\mathcal{C}$. Given a message $m \in \mathcal{M}$, the commitment algorithm samples $r \leftarrow \mathcal{R}$ uniformly at random and computes $\mathsf{Com}_{ck}(m; r) \in \mathcal{C}$. Given m, r and a commitment $c \in \mathcal{C}$, the Open algorithm $\mathsf{Open}_{ck}(m; r; c)$ outputs $1/0$ whether or not c is a valid commitment for the pair m, r. A non-interactive commitment scheme is perfectly hiding, and computationally binding under the DLP.*

In this work, we will use the Pedersen Commitment along with the Jubjub elliptic curve. Let $\mathbb{J}$ be a group of order t and set our message and randomness spaces $\mathcal{M}, \mathcal{R} = \mathbb{F}_t$ and our commitment space $\mathcal{C} = \mathbb{J}$. The Setup, Commit, and Open algorithms for Pedersen commitments are defined as follows:

- *Setup.* Sample and output the commitment key $ck = (G, G') \leftarrow \mathbb{J}$.
- *Commit.* On input a message $m \in \mathcal{M}$, sample randomness $r \leftarrow \mathbb{F}_t$ and output
$$c = \mathsf{Com}_{ck}(m; r) = mG + rG'.$$
- *Open.* Reveal m, r. With these values, anyone can recompute the commitment and check whether it matches the commitment previously provided.

The Pedersen Commitment scheme is perfectly hiding, and computationally binding under the DLP.

On the other hand, we will make use of hash functions, which we define as follows.

Definition 6 (Hash Functions)**.** *A* cryptographic hash function *is a function* $H : \{0,1\}^* \to \{0,1\}^\ell$ *that is collision-resistant, that is, it is hard to find* $x, x' \in \{0,1\}^*$ *such that* $x \neq x'$ *but* $H(x) = H(x')$*.*

Throughout the document, we will be using two specific hash functions: first, BLAKE2b [15], a lightweight and efficient hash function. Second, Poseidon [16], whose main feature is being SNARK-friendly: it is cheap in terms of computing resources when computed into a specific ZKP scheme called zk-SNARK.

2.5 Merkle Trees

Merkle trees [17] are data structures containing at every node the hash of its children nodes. Considering a k-ary tree of h levels, the single node at level 0 is called the *root* of the tree, and the k^h nodes at level h are called the *leaves*. Given a node placed in the level i, the k nodes in the level $i+1$ that are adjacent to it are called its *children*. Plus, a node is the other's *sibling* if they all are children of the same node.

The tree is partially updated every time a new value is written (or modified) into a leaf, always resulting in a new root of the tree. Furthermore, given a root r, it is easy to prove that a value x is in a leaf of a tree with root r. The proof works as follows:

- *Prove.* For $i = h, \ldots, 1$, let x_i be the node that is in level i and is in the unique path from x to the root. Let $y_{i,1}, \ldots, y_{i,k-1}$ be the $k-1$ siblings of x_i. Output

$$(x, (y_{1,1}, \ldots, y_{1,k-1}), \ldots, (y_{h,1}, \ldots, y_{h,k-1})).$$

- *Verify.* Parse input as $(x_h, (y_{1,1}, \ldots, y_{1,k-1}), \ldots, (y_{h,1}, \ldots, y_{h,k-1}))$, where x_h is the purported value and $y_{i,1}, \ldots, y_{i,k-1}$ are the purported siblings at level i. For $i = h-1, \ldots, 0$, compute[1]

$$x_i = H\left(x_{i+1}, y_{i+1,1}, \ldots, y_{i+1,k-1}\right).$$

  If x_0 equals the root r of the set we are proving membership of, the proof is verified.

We can prove membership in a set of size k^h by sending kh values, so we can state that the communication complexity for proving the membership is $O(kh)$. If the hash function is collision-resistant, the proof is sound.

2.6 Digital Signatures

Digital signatures are one of the most important pieces needed to build our solutions. In particular, we are interested in the Schnorr signature scheme, which we describe now. Let $G, G' \leftarrow \mathbb{J}$. Then, we have the following algorithms:

- *Setup.* Sample a secret key $\mathsf{sk} \leftarrow \mathbb{F}_t$ and compute a public key $\mathsf{pk} = \mathsf{sk}G$.
- *Sign.* To sign a message m using sk, sample $r \leftarrow \mathbb{F}_t$ and compute $R = rG$. Compute the challenge $c = H(m, R)$, and set

$$u = r - c\mathsf{sk}.$$

  Set the signature $\mathsf{sig} = (R, u)$.

- *Verify.* To verify a signature $\mathsf{sig} = (R, u)$ of a message m using pk, we first compute $c = H(m, R)$ and check whether the following equality holds:

$$R \stackrel{?}{=} uG + c\mathsf{pk},$$

  If it equals, accept the signature, reject otherwise.

[1] Additionaly, the prover also has to send $\lceil \log_2 k \rceil$ bits for each level, specifying the position of x_i with respect to its siblings, so that the verifier knows in which order to arrange the inputs of the hash.

This scheme is existentially unforgeable under chosen-message attacks under the DLP, in the random oracle model [18, Section 12.5.1].

We are also interested in a double-key version of this signature scheme, that will be used to delegate some computations later in the transaction model of Dusk Network. We have the following algorithms:

- *Setup.* Sample a secret key $\mathsf{sk} \leftarrow \mathbb{F}_t$ and compute a public keypair $(\mathsf{pk}, \mathsf{pk}') = (\mathsf{sk}G, \mathsf{sk}G')$.
- *Sign.* To sign a message m using sk, sample $r \leftarrow \mathbb{F}_t$ and compute $(R, R') = (rG, rG')$. Compute the challenge $c = H(m, R, R')$, and also

$$u = r - c\mathsf{sk}.$$

  Finally, set the signature $\mathsf{sig} = (R, R', u)$.
- *Verify.* To verify a signature $\mathsf{sig} = (R, R', u)$ of a message m using $(\mathsf{pk}, \mathsf{pk}')$, we first compute $c = H(m, R, R')$ and check whether the following equalities hold:

$$R \stackrel{?}{=} uG + c\mathsf{pk},$$
$$R' \stackrel{?}{=} uG' + c\mathsf{pk}'.$$

  If they are equal, accept the signature, reject otherwise.

Throughout the document, we will also use a variant of the Schnorr signature, called the Edwards-curve Digital Signature Algorithm (EdDSA) [14].

2.7 Zero-Knowledge Proofs

A Zero-Knowledge Proof (ZKP) [2] is a cryptographic primitive allowing a prover $\mathcal{P}$ to convince a verifier $\mathcal{V}$ that a public statement is true, without leaking any secret information.

Given a statement u, and a witness w being some secret information only known by $\mathcal{P}$, $\mathcal{P}$ wants to convince $\mathcal{V}$ that they know w. Both u and w are related by a set of operations defined by a *circuit*, a graph composed of different wires and gates, which leads to a set of equations involving the inputs and the outputs of these gates. Each of these equations is called a *constraint*. $\mathcal{P}$ can execute a proving algorithm using u as the set of public inputs, and w as the private inputs. This execution outputs a set of elements, which we call the proof π. $\mathcal{P}$ sends π to $\mathcal{V}$, who will use a verifying algorithm to verify that u is true, for a given w only known by $\mathcal{P}$. In essence, ZKPs must satisfy 3 properties:

- **Completeness:** If the statement is true, $\mathcal{P}$ must be able to convince $\mathcal{V}$.
- **Soundness:** If the statement is false, $\mathcal{P}$ must not be able to convince $\mathcal{V}$ that the statement is true.
- **Zero-knowledge:** $\mathcal{V}$ must not learn any information from the proof beyond the fact that the statement is true.

First ZKP schemes used to achieve the aforesaid properties by exchanging several messages between $\mathcal{P}$ and $\mathcal{V}$. However, non-interactive ZKPs [19] arose, providing an extra feature, where $\mathcal{P}$ could prove statements to $\mathcal{V}$ by sending them a single message, instead of several interactions.

Even so, computing and verifying ZKPs used to require high computing resources, and this made them impractical in real applications. More recently, Zero-Knowledge Succinct and Non-interactive ARguments of Knowledge (zk-SNARKs) [20] appeared: ZKPs that can be computed and verified in a more efficient way, compared to previous solutions, making them suitable for real applications, like privacy-preserving cryptocurrencies [21]. As shown in Table 2.1, one of the first efficient schemes was a zk-SNARK introduced in BSCTV'13 [22]. Such a scheme was later improved by the zk-SNARK introduced in Groth'16 [23], which is still one of the most efficient zk-SNARK schemes, especially when it comes to the verification algorithm. One of the main drawbacks of this kind of construction is the need for a trusted party that performs a *trusted setup*: a phase where some public parameters are generated, which will be used to generate and verify proofs. In this regard, Sonic [24] is a zk-SNARK that introduces a scheme where the setup can be updated for different circuits without the need to repeat the trusted setup generation. Another efficient zk-SNARK is Libra [25], which prover is guaranteed to outperform, even when the verifier does not have constant complexity. Another recent and widely used work on this topic was done by the authors in [26]. They introduce PlonK, a scheme with the same advantages that Sonic has, but improving the speed of the prover. Finally, recent research such as [27] shows a way to design a more efficient prover, while not compromising the verifier.

On the other hand, and beyond zk-SNARKs, we can find other ZKP schemes like Bulletproofs [28], which have the main advantage of not requiring a trusted setup. They are especially useful when the prover needs to compute a *range proof* [30], instead of an arithmetic circuit. Moreover, other concerns like post-quantum security have also arisen in the Zero-Knowledge field, and in this regard, we have a scheme supposed to be post-quantum secure, called zk-STARKs

Table 2.1: Comparison of different ZKP constructions, in regards to their asymptotic efficiency, where n is the number of gates of the circuit and d its depth. (TS = trusted setup, PQS = post-quantum secure, $|\pi|$ = proof size, $\odot$ = per-statement, $\triangle$ = updatable)

Scheme	TS	PQS	Prove	Verify	$\|\pi\|$	Assumption
BSCTV'13 [22]	$\odot$	no	$O(n \log n)$	$O(1)$	$O(1)$	q-PKE
Groth'16 [23]	$\odot$	no	$O(n \log n)$	$O(1)$	$O(1)$	q-PKE
Sonic [24]	$\triangle$	no	$O(n \log n)$	$O(1)$	$O(1)$	AGM
Libra [25]	$\triangle$	no	$O(n)$	$O(d \log n)$	$O(d \log n)$	q-SBDH, q-PKE
Bulletproofs [28]	no	no	$O(n)$	$O(n)$	$O(\log n)$	DLP
zk-STARKs [29]	no	yes	$O(n \log^2 n)$	$O(\log^2 n)$	$O(\log^2 n)$	CRHF

(Zero-Knowledge Succinct Transparent ARgument of Knowledge) [29]. Post-quantum security is not a property of zk-SNARKs or Bulletproofs.

Finally, the soundness property of each of the schemes described relies on different security assumptions [31]. Most of the zk-SNARK constructions use a strong assumption called q-Power Knowledge of Exponent (q-PKE), which is not the best solution. On the other hand, Bulletproofs or zk-STARKs use better assumptions: the DLP and Collision Resistant Hash Functions (CRHF), respectively.

To develop ZKP applications, libraries like the one provided in this thesis are required. One of the main libraries to accomplish this purpose is **libsnark**[2], a C++ library for constructing zk-SNARKs, which was used for some time by Zcash [21], based on the specific zk-SNARK construction introduced in [22], but supporting [23] as well, among others. Even when this library provides excellent benchmarks, one of the main drawbacks of this library, as the authors state, is not being well-optimized for ARM architectures.

Another library with similar benchmarks is **bellman**[3], implemented in Rust and meant for constructing zk-SNARKs, developed and currently used by Zcash.

Moreover, when it comes to developing DApps for the Ethereum blockchain, we previously stated that a verifier coded in Solidity is required. **ZoKrates**[4] is a python toolbox for zk-SNARKs intended to generate Solidity verifiers, to be deployed into the Ethereum Blockchain. Furthermore, a similar approach is **snarkjs**[5], a JavaScript library for constructing zk-SNARKs. It includes a

[2]https://github.com/scipr-lab/libsnark
[3]https://github.com/zkcrypto/bellman/
[4]https://github.com/Zokrates/ZoKrates
[5]https://github.com/iden3/snarkjs

clear API for generating trusted setups using a fairly secure MPC protocol, for generating proofs, and for verifying them. Plus, it also provides an easy way to export the verifier in Solidity, to deploy it into the Ethereum Blockchain.

2.7.1 zk-SNARKs

In this section, we do a high-level overview of how to construct a zk-SNARK (based on the construction introduced in [23]), which is also the scheme used in our proof-of-concept ZPiE.

zk-SNARKs [22] are the most used ZKPs, because they are short and succinct: the proofs can be verified in a few milliseconds. However, they require a trusted setup where some public parameters are generated. These parameters, called the Common Reference String (CRS) are used by $\mathcal{P}$ and $\mathcal{V}$ to generate and verify proofs. To generate the CRS, a secret randomness τ is used, and such a randomness should be destroyed afterward. If an attacker gets τ, the soundness property of the scheme breaks: the attacker would be able to compute false proofs that anyone could verify as if they were correct. As such, the CRS is commonly computed using a secure Multi-Party Computation (MPC) protocol [32], where τ can only be leaked if all the participants are malicious. The computing complexity of generating a setup, computing proofs, and verifying them, depends on the number of operations that we do in the circuit, which is also the number of gates n.

Regarding the security of zk-SNARKs, it mainly relies on the security of elliptic curves. Breaking the security of the elliptic curve used by a specific construction would lead to being able to generate false proofs and thus, breaking the soundness property of the scheme. Among the most used curves in ZKPs we have a Barreto-Naehrig curve [33] called BN128, which security level in practice is estimated to be 110-bits [34]. Another common curve in this scenario is BLS12-381 [21], which has around 128-bits of security, with the drawback of heavier group operations. More recent research is introduced in [35], where a new curve called BW6-761 is introduced. As stated by its authors, the verification of proofs is at least five times faster than other state-of-the-art curves.

The zk-SNARK construction we will work on requires a pairing-friendly elliptic curve E over a finite field $\mathbb{F}_q$, where q is a prime number, with the bilinear groups $(\mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T)$ of prime order p and a pairing $e : \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_T$ being a bilinear map. We need the following generators: an element g being a generator for $\mathbb{G}_1$, an element h being a generator for $\mathbb{G}_2$, and $e(g, h)$ being a generator for $\mathbb{G}_T$. In order to represent group elements, we use the multiplicative notation: we write $[a]_1$ for g^a, $[b]_2$ for h^b, and $[c]_T$ for $e(g, h)^c$.

Arithmetic circuit. A circuit is a directed acyclic graph composed of different wires and gates, which lead to a set of equations relating the inputs and the outputs of these gates. The inputs and the output of this circuit, as well as the operations defined in the gates, are elements over a prime field $\mathbb{F}_p$, where p is the order of E.

Rank 1 Constraint System. A Rank 1 Constraint System (R1CS) is a system of equations that checks the correctness of all the operations in our circuit, by grouping them in *constraints*. Each constraint is composed of a multiplicative gate with its two inputs and its output. Having a statement u and a witness w, a R1CS is defined as a set of vectors (a, b, c) describing our circuit, whose solution is a vector $s = (u, w)$ such that the following equation is satisfied:

$$< a, s > \cdot < b, s > - < c, s >= 0 \tag{2.2}$$

Quadratic Arithmetic Program. A Quadratic Arithmetic Program (QAP) is a polynomial representation of the R1CS, which bundles all its constraints into one. It is a tuple of the polynomials $(A, B, C, Z(x))$, where $Z(x)$ divides $A_i(x) \cdot B_i(x) - C_i(x)$ without remainder. They satisfy the following equation:

$$\sum_{i=0}^{m} s_i A_i(x)) \cdot \sum_{i=0}^{m} s_i B_i(x) - \sum_{i=0}^{m} s_i C_i(x) = H(x)Z(x) \tag{2.3}$$

Let $\mathcal{R}$ be a relation composed of the elliptic curve E over $\mathbb{F}_q$, the pairing e and a QAP $(A, B, C, Z(x))$ representing a circuit. The Groth'16 construction is divided into three algorithms (further details can be found in Appendix A), as depicted in Figure 2.1:

- $\mathsf{pk}, \mathsf{vk} \leftarrow \mathsf{Setup}(\mathcal{R})$: given the relation $\mathcal{R}$, the first step of the protocol generates a common reference string (CRS) $\sigma = ([\sigma_1]_1, [\sigma_2]_2)$. From the CRS, some elements will be extracted into what we call the proving key pk, sent to the prover $\mathcal{P}$ to generate proofs. Moreover, other required elements will be taken into the verifying key vk, and sent to the verifier $\mathcal{V}$ to verify the proofs generated by $\mathcal{P}$. In order to generate the CRS (i.e. pk and vk), a random set of values τ is used. This τ, also known as *trapdoor*, should be destroyed after performing the setup, as any party having τ would be able to generate false proofs. To solve this last drawback, the setup must be generated by a trusted party (i.e. a set of entities performing a secure MPC protocol). In scenarios such as cryptocurrencies using zk-SNARKs

(i.e. Zcash), an untrusty setup could lead to malicious parties using τ to create false transactions, thus leading to losses of money.

- $\pi \leftarrow \mathsf{Prove}(\mathcal{R}, \mathsf{pk}, u, w)$: a proof $\pi = ([\pi_A]_1, [\pi_B]_2, [\pi_C]_1)$ is generated by the prover, by multiplying u and w by some polynomials provided in σ. The prover also needs to compute the coefficients h of $H(x)$, which can be achieved in $O(n \log n)$ using Fast Fourier Transform (FFT) techniques, as explained in detail in Appendix B. Then, h is multiplied by a polynomial provided in σ. The number of multi-exponentiations required to compute π are (note that multi-exponentiations in $\mathbb{G}_2$ are more expensive than multi-exponentiations in $\mathbb{G}_1$):
  - to compute $[\pi_A]_1$: $|u| + |w|$ multi-exponentiations in $\mathbb{G}_1$.
  - to compute $[\pi_B]_2$: $|u| + |w|$ multi-exponentiations in $\mathbb{G}_2$.
  - to compute $[\pi_C]_1$: $|u| + 2 \cdot |w| + |h|$ multi-exponentiations in $\mathbb{G}_1$.
- $0/1 \leftarrow \mathsf{Verify}(\mathcal{R}, \mathsf{vk}, u, \pi)$: the verifier accepts the proof (1) if an equation composed of three pairings [36] holds. Otherwise, the proof is rejected (0). Moreover, modifying a single bit of the proof leads to a proof that cannot be verified.

As such, the workflow of these algorithms in real applications would work as follows: we first need to compute the setup by means of a trusted party (step 1 in Figure 2.1). Later, this party shall share the required values with each of the involved parties, the prover, and the verifier (step 2 in Figure 2.1). As shown in steps 3 and 4, the prover computes the proof π and sends it to the verifier, who verifies it (step 5 in Figure 2.1). Steps 1 and 2 are performed only once, and later, the prover can compute as many proofs as they want using the same values computed during the setup.

2.7.2 Bulletproofs

Bulletproofs [28] are short non-interactive zero-knowledge arguments of knowledge that require no trusted setup. This means that the prover $\mathcal{P}$ sends a single message to the verifier $\mathcal{V}$, and this is enough to prove knowledge of the secret information. There is no need to rely on any prior information generated by a trusted party.

Bulletproofs were designed to enable efficient confidential transactions in cryptocurrencies, but they have found many other applications, such as shortening proofs of solvency or enabling confidential smart contracts [37]. The main

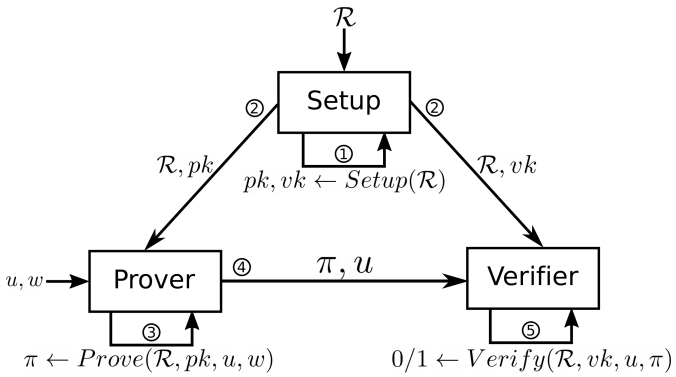


Figure 2.1: Zero-Knowledge Proof System.

technical feature of Bulletproofs is to prove that a committed value lies within a certain interval. For example, in the context of Blockchains, it is very useful to have an efficient protocol to prove that a secret value lies in the interval $[0, 2^n - 1]$ for some large value of $n \in \mathbb{Z}_{\geq 0}$. In the cryptographic community, this feature is called a *range proof*. Range proofs allow us to prove that a secret value (previously committed to) lies within a certain range. They do not leak any information about the secret value but the fact that it lies within the desired range.

Let $\mathbb{G}$ be a cyclic group of prime order p and let $\mathbb{Z}_p$ be the ring of integers modulo p. An *inner-product argument* lets $\mathcal{P}$ convince $\mathcal{V}$ that they know two vectors (bold font denotes a vector) $\mathbf{a}, \mathbf{b} \in \mathbb{Z}_p^n$ such that

$$C = \mathbf{g}^{\mathbf{a}}\mathbf{h}^{\mathbf{b}} \quad \text{and} \quad c = \langle \mathbf{a}, \mathbf{b} \rangle,$$

where $\mathbf{g}, \mathbf{h} \in \mathbb{G}^n$ are independent generators, $c \in \mathbb{Z}_p$, and $C \in \mathbb{G}$. Now, let $c \in \mathbb{Z}_p$ and let $C \in \mathbb{G}$ be a Pedersen Commitment to c using randomness r. An inner-product range proof allows P to convince V that $c \in [0, 2^n - 1]$ by proving the relation

$$\{(g, h \in \mathbb{G}, C, n \ ; \ c, r \in \mathbb{Z}_p) \ : \ C = h^r g^c \wedge c \in [0, 2^n - 1]\}.$$

Now consider the case where P needs to provide multiple range proofs at the same time. The idea of aggregated range proofs is to build a system that can provide a proof for multiple secret values and its efficiency is better than doing one proof for each of the secrets. Since the inner-product range proofs provided

by Bulletproofs have logarithmic size, it is possible to build efficient aggregated logarithmic range proofs. That is, it is possible to efficiently prove the relation

$$\{(g, h \in \mathbb{G}, \mathbf{C} \in \mathbb{G}^m \ ; \ \mathbf{c}, \mathbf{r} \in \mathbb{Z}_p^m) \ : \ C_j = h^{r_j} g^{c_j} \wedge c_j \in [0, 2^n - 1] \ \forall j \in [1, m]\},$$

where m corresponds to the number of proofs. Bulletproofs can be computed in $O(n)$, and verified in linear time as well. The communication complexity (the size of the proofs) is $O(\log n)$.

2.8 Blockchain

A Blockchain [38] is a unique and immutable data structure called ledger, and shared by a set of nodes. Cryptocurrencies like Bitcoin [39] use such technology, and populate the ledger with transactions exchanging money between parties. These transactions are cryptographically validated by the nodes of the network, to be sure that each user spends what belongs to them. This process is a consensus agreed on by all the users of the network (e.g. Proof of Work [40], Proof of Stake [41], etc.).

Beyond the feature of exchanging money, Blockchains like Ethereum [42] grant the possibility of executing decentralized applications (DApps) on-chain. DApps are possible thanks to smart contracts [43], programs that can be executed on-chain thanks to the Ethereum Virtual Machine (EVM) [44]. Such contracts and the EVM allow, for instance, to execute some action (like issuing a payment) upon fulfilling some conditions.

Furthermore, other Blockchains like Dusk Network [45] also provide virtual machines to execute smart contracts. In this particular case, Dusk has the `Rusk` virtual machine, which like the EVM can execute smart contracts, but with the difference that all the transactions handled by `Rusk` are private by default, thanks to ZKPs.

From a more technical perspective, it is worth mentioning that, in order to prevent saturation of the network, users are required to pay *gas* in order to execute transactions. This is the amount of Dusk coins per amount of bytes needed to execute a transaction. Depending on how busy the Dusk Network is, the price of the gas increases or decreases. Like this, performing a Denial-of-Service (DoS) attack becomes so expensive that is infeasible [46].

2.9 Smart Contracts

One of the most useful Blockchains in regards to our scenario is Ethereum [42]. It is a network whose purpose is not to be a currency for making payments but a way to run distributed applications (DApps). DApps are possible thanks to smart contracts [43], pieces of code executed on the Ethereum Virtual Machine (EVM) [44]. Such contracts and the EVM allow users, for instance, to be paid upon fulfilling some conditions. That is, for instance, distributed exchanges: applications where users buy or sell their cryptocurrencies to other users.

In order to execute transactions, Ethereum requires *gas*. This is the amount of Ether (Ethereum's coin) per amount of bytes needed to run a transaction. Depending on how busy the Ethereum network is, the price of gas increases or decreases. This can make using Ethereum very expensive. To overcome such a problem, Zero-Knowledge-Rollups (zk-Rollups) have been proposed recently [47]. They basically group several transactions into a single transaction of the main Ethereum Blockchain. Whereas the Ethereum network is called *Layer 1*, the zk-Rollup is commonly called application of *Layer 2*. zk-Rollups are possible thanks to ZKPs. zk-Rollups are introduced in Section 2.10.

Similarly, as Ethereum does, Dusk Network [45] is a Layer 1 Blockchain that provides a virtual machine called `Rusk` which enables the deployment and execution of smart contracts. However, they introduce the *Confidential Security Contract Standard (XSC)*, which ensures the preservation of transactional confidentiality while simultaneously guaranteeing compliance through the use of ZKPs. This opens the door to a wide variety of use cases where privacy is a must, but accountability is required at the very same time.

2.10 zk-Rollups

zk-Rollups [47], as depicted in Figure 2.2, create batches of several transactions in a Layer 2 scenario, and publish the whole batch into a single Layer 1 transaction. This saves a lot of gas that would be consumed if each transaction was executed directly on the main Blockchain. To do so, we have two actors, the *transactors* willing to create a rollup transaction, and the *relayers* computing the required operations to make the rollup work. In that regard, transactors send transactions to the relayers containing information about the sender, the receiver, the amount of tokens to be sent, etc. Such transactions also include a signature of the transaction. As stated previously, ZKPs require an elliptic curve, as proofs are sets of elements on such curves. For instance, the

BN128 is the currently used curve for zk-SNARKs in Ethereum. The signature scheme used is EdDSA, which also requires an additional elliptic curve where parameters are compatible with the zk-SNARKs elliptic curve (BN128). In this scenario, the Baby JubJub elliptic curve is used, for its compatibility with the parameters of BN128.

Once the relayer has received a bunch of transactions, they compute a Merkle tree of the previous accounts' state and the new state. Later, they compute a zk-SNARK which verifies all the signatures, and posts on the Blockchain a transaction containing this batch: the rollup transactions, the previous and new root states, and the zk-SNARK. This transaction is verified by a smart contract previously deployed on the Blockchain.

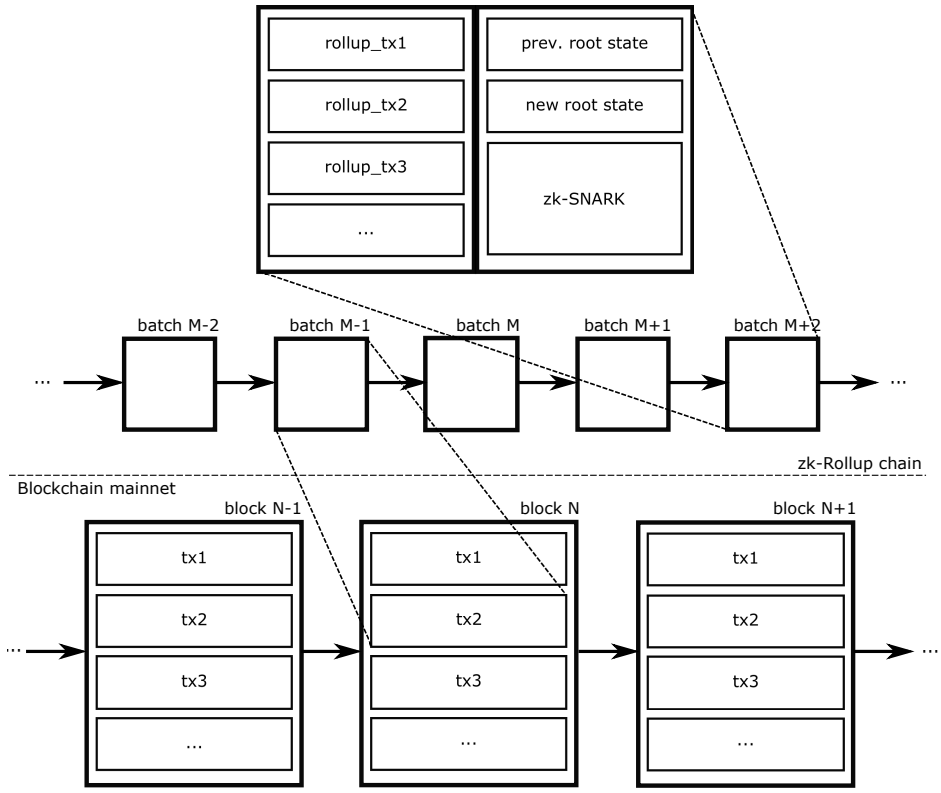


Figure 2.2: zk-Rollups overview.

Chapter 3

LASER: Lightweight And SEcure Remote keyless entry protocol

The usage of RKE systems has been increasing over the years, being widely used to remotely lock and unlock cars, garage doors, sensors, doorbells, or alarms. The first RKE systems used a simple protocol, where a code was sent in plaintext to a receiver that had to execute a command, let us say, unlock a door. However, as sniffing and replaying the code was enough to be able to unlock such a door, a new scheme called *rolling codes* was developed, and it is still widely used nowadays. Such a scheme is supposed to be secure because the key fob computes and sends a new code each time it is used, and each code is accepted by the receiver just once. Nonetheless, it has been proved that rolling codes are vulnerable to different attacks, and authorities are starting to report[1] criminals taking profit of these vulnerabilities. This fact has led researchers to design new secure schemes [48] to protect these systems, but their complexity made manufacturers not implement them, so it would mean developing key fobs with some disadvantages, i.e. a higher price or a faster draining of the battery. This is due to the fact that many solutions proposed to use cryptographic schemes [49] which needed higher computing power than the available in the current fobs. Furthermore, the proposed protocols usually need

[1] https://www.west-midlands.police.uk/news/watch-police-release-footage-relay-crime

more than one message to exchange some private information or instruction commands. For example, some solutions [50] require to use a 4-way handshake before sending an instruction command, which increases the complexity of the protocol.

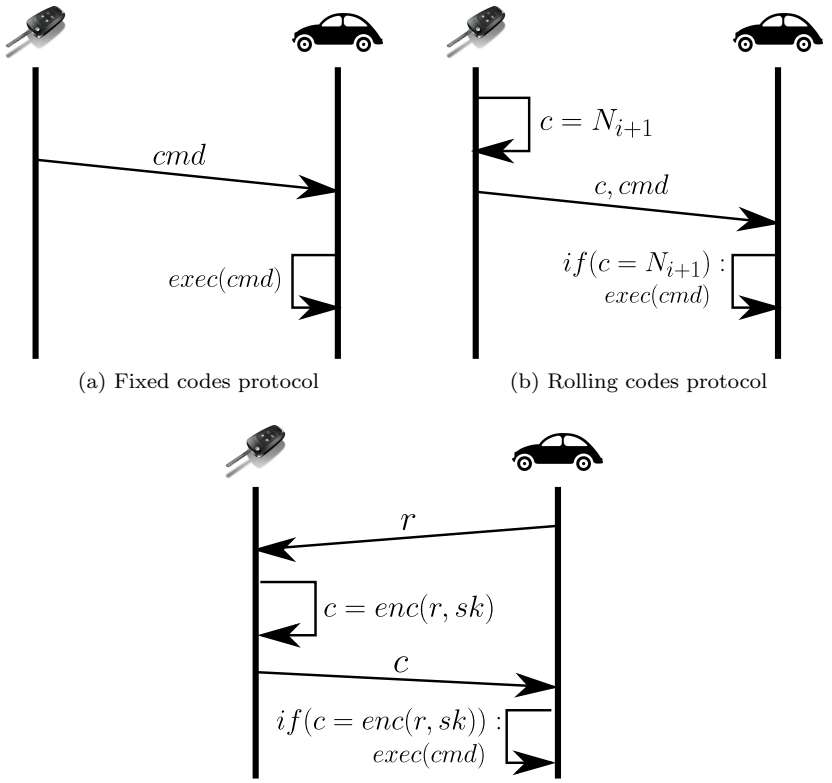
We provide a secure protocol to be implemented by manufacturers into both RKE and PRKE systems. Our scheme is robust against both jamming-and-replay attacks and relay attacks; furthermore, it mitigates the effectiveness of jamming-based denial-of-service attacks, thanks to the integration into the protocol of a frequency-hopping approach. Moreover, our solution is a one-message protocol for RKE systems and a two messages protocol for PRKE systems, where both approaches use a hash function proved to have low CPU resources consumption. As such, our solution is lightweight, scalable, and easy to implement. The purpose of this solution is to be applied to key fobs with the only requirement of having a real-time clock, synchronized periodically as detailed in our protocol. We also demonstrate how our solution can be implemented, and we achieve good results.

3.1 Remote Keyless Entry Systems

We call RKE to those systems which are composed of a fob $\mathcal{F}$ and a device $\mathcal{D}$. When a button on $\mathcal{F}$ is pressed, a radio frequency signal is sent to $\mathcal{D}$, including an instruction command that $\mathcal{D}$ will have to execute. These systems are commonly used to lock or unlock cars and open their boots, to open a garage door, to control a temperature sensor, etc. The main protocols used by these systems can be divided as follows:

- *Fixed codes.* This is the simplest scheme. As depicted in Figure 3.1a, $\mathcal{F}$ sends a command *cmd* to $\mathcal{D}$, which is essentially a bit stream referring to an action that $\mathcal{D}$ will have to perform.

- *Rolling codes.* There is a wide variety of rolling codes algorithms, but all of them rely on the idea of sending different codes each time a button of $\mathcal{F}$ is pressed. In order to accomplish this purpose, both $\mathcal{F}$ and $\mathcal{D}$ have previously agreed on a secret key from which derives a sequence of codes $N_1, N_2, ..., N_p$. Then, as depicted in Figure 3.1b, each time a button on $\mathcal{F}$ is pressed, the next code c is computed and sent to $\mathcal{D}$, who checks if the received number is equal to a value c that previously it also computed. Apart from c, a command cmd is also sent, which is typically a sequence of bits that refers to an action $\mathcal{D}$ will have to do, i.e. unlock a car. Each



(a) Fixed codes protocol

(b) Rolling codes protocol

(c) Challenge-response protocol

Figure 3.1: Main RKE and PRKE protocols

value c can be used only once. In case $\mathcal{D}$ may have not received some of the codes sent by $\mathcal{F}$, it commonly checks up to the next 256 generated codes, and when a correct value c is received by $\mathcal{D}$, all the codes behind it cannot be used again. One of the most used rolling codes devices has been KeeLoq [51].

On the other hand, Passive Remote Keyless Entry (PRKE) systems [52] are a special type of RKE. PRKE systems do not require the user to manipulate $\mathcal{F}$. Instead, as soon as $\mathcal{D}$ receives an external input (i.e. if $\mathcal{D}$ is a door, someone pulling the handle), it automatically sends a request to $\mathcal{F}$, which replies with a confirmation. The most used protocol [53] for PRKE systems is the challenge-response protocol:

Challenge-response protocol for PRKE Both $\mathcal{D}$ and $\mathcal{F}$ perform the following 2-message handshake:

1. First, $\mathcal{D}$ computes a random value r (the challenge), and sends it to F.
2. $\mathcal{F}$ encrypts r using a pre-shared symmetric-key sk, and sends the encrypted value c to $\mathcal{D}$.
3. $\mathcal{D}$ decrypts c using the same key sk and verifies the identity of $\mathcal{F}$.

For example, if $\mathcal{D}$ is a car using a challenge-response protocol, when the user carrying $\mathcal{F}$ pulls the car handle, $\mathcal{D}$ sends a message with a challenge r, and as soon as $\mathcal{F}$ receives it, it replies with its answer. This is depicted in Figure 3.1c.

3.2 Attacks against RKE and PRKE

Jamming-and-replay attack. As depicted in Figure 3.2, these attacks [54] are performed using two transceiver devices. One of them is placed near to $\mathcal{D}$, hidden from the view of the victim $\mathcal{V}$, and jamming the frequency used by the system an adversary $\mathcal{A}$ is willing to hack. Then, the other one is close to $\mathcal{F}$, eavesdropping on the communications. When $\mathcal{V}$ presses the button of $\mathcal{F}$, the signal it sends is jammed by the jamming transceiver $\mathcal{J}$, and $\mathcal{V}$ is forced to use an alternative (i.e. a physical key). Meanwhile, $\mathcal{A}$ captures the message sent by $\mathcal{F}$, and as $\mathcal{D}$ never receives it, $\mathcal{A}$ will be able to replay it later. Finally, the jammer can be remotely deactivated by $\mathcal{A}$, as soon as they are sure that $\mathcal{V}$ will not try to use $\mathcal{F}$ again.

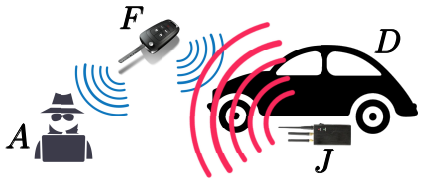


Figure 3.2: Jamming-and-replay attack

Relay attack. As it can be seen in Figure 3.3, this kind of attacks [55] are performed using two transceivers connected through an LTE network or similar. One of them is close to $\mathcal{D}$, and the other one to $\mathcal{F}$. Like this, they create a bridge between both endpoints. If the attacked system is a PRKE, when the adversary $\mathcal{A}_2$ pulls the car handle the challenge-response protocol is performed through the bridge created by both adversaries. Otherwise, if we are talking about an RKE system, we have to expect that the user may either accidentally press the button on $\mathcal{F}$, or leave it unattended (thus allowing the adversary $\mathcal{A}_1$ to press the button).

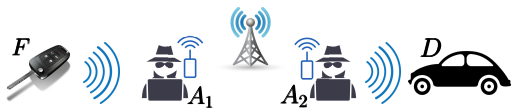


Figure 3.3: Relay attack

Denial-of-service (DoS) attack. This kind of attack [56] is also based on jamming the frequency used by the protocol, but in this case with the main goal of denying the service. It has a lower impact on the system security as it does not grant access to the system, but it bothers the user, who will require a physical key if they want to perform the action.

3.3 Related Work

Regarding the attacks against RKE systems, an important contribution on the topic has been recently done in [57]. They demonstrate as the jamming-and-relay attacks are nowadays still effective against a wide variety of modern cars,

by making use of two units of a radio frequency device called HackRF One[2], one for jamming and the other one for logging data and replaying later.

A particular RKE scheme based on rolling codes, and widely used by many manufacturers, is called *Hitag2* [58]. An important contribution related to this type of RKE has been done in [59], where a novel correlation-based attack is presented. This attack allows an adversary to recover the secret key used in *Hitag2* systems, just by eavesdropping on at least four of the codes sent by the fob. Thus, it allows the adversary to clone the fob. As stated in the paper, major manufacturers have sold systems with this vulnerability for over 20 years. As such, the need for new secure and easy-to-implement schemes becomes clear.

Implementation of the attacks. By making use of two radio frequency devices called Yardstick One[3] (YS1), a jamming-and-replay attack can be performed by using a python implementation[4] of this attack. This implementation makes use of a library called *rflib*, included in a software used by YS1 called RfCat[5]. That said, one antenna will be jamming while the other will be sniffing the code of the fob. The same implementation is useful for performing just the DoS attack. Moreover, taking this implementation as a starting point, implementing a relay attack is trivial.

Proposed solutions. Many secure schemes [48], [50] have been designed to increase the security of RKE and PRKE systems. The main problem they present is their complexity, so they use cryptographic schemes which are hard to implement into cheap key fobs. On the other hand, some schemes [60] have been proven to be both simple and effective against relay attacks. One of them, proposed in [61], demonstrates that a protocol calculating the time between message exchanges can determine if a relay attack is being performed against a PRKE or not. This is the main idea behind `LASER`, which also solves the replay vulnerability.

3.4 Solution Overview

In this section, we explain step-by-step our protocol, `LASER`, for both RKE and PRKE systems. We consider a fob $\mathcal{F}$ and a generic device $\mathcal{D}$, assuming it to be a car. First, both endpoints have to agree on a randomly generated secret key sk large enough to make a brute-force attack hard to accomplish (i.e. a 256-bits

[2] https://greatscottgadgets.com/hackrf/
[3] https://greatscottgadgets.com/yardstickone/
[4] https://github.com/exploitagency/rfcat-rolljam
[5] https://github.com/atlas0fd00m/rfcat

key). They also need to agree on a set of commands cmd, used for example to lock the car, unlock it, etc. $\mathcal{D}$ also has a car identification number (device_id) known by $\mathcal{F}$.

In both RKE and PRKE systems, both $\mathcal{F}$ and $\mathcal{D}$ will be required to compute a hash. The hash function used by both devices was required to be lightweight in order to optimize the timings and the resources consumption. For our implementation and analysis, we have chosen to use *Blake2*, a hash function proposed in [15], which guarantees a low power and computing resources consumption. Furthermore, it is proved to be as fast as *MD5*, but solving the security vulnerabilities *MD5* presents.

In particular, we are interested in using *Blake2s*, a version of *Blake2* optimized for 8-bit platforms, which are the kind of cheap processors commonly used for key fobs. Basing our solution on the usage of a hash function like *Blake2* instead of using some complex cryptographic scheme, we are decreasing the costs of implementing our solution, and also avoiding a fast draining of the battery.

Our solution performs a frequency-hopping protocol where the frequency channel used to transmit the messages changes each period of time p. This means that both $\mathcal{D}$ and $\mathcal{F}$ must agree on the same channel, and to achieve it they perform the following protocol.

Frequency-hopping for LASER The frequency-hopping for a specific endpoint, which has a number of available frequency channels N_c, is performed as follows:

1. Each period of time p (both $\mathcal{F}$ and $\mathcal{D}$ have previously agreed on this value) both parties get the current date-time d in a timestamp form and calculate the hash $h = Hash(\mathsf{sk}, d)$.

2. It calculates the channel ch, which is the modulo N_c of the integer representation of h: $ch \equiv int(h) \pmod{N_c}$.

The next subsections explain the specific details for both RKE and PRKE systems.

3.5 LASER for RKE

In this subsection, we first explain all the steps of the RKE protocol in detail, and later the main approach used to prevent each kind of attack.

3.5.1 Protocol Decription

In this scheme, $\mathcal{D}$ is required to be always listening to a specific channel, so it will be continuously performing the frequency-hopping protocol previously introduced. However, $\mathcal{F}$ will perform it just before starting the `LASER` protocol. When the owner of $\mathcal{D}$ wants to execute a command cmd by pressing a button on $\mathcal{F}$, $\mathcal{F}$ calculates ch by first calculating h, but rounding the timestamp to the previous multiple of p. Then, the next protocol is performed (as depicted in Figure 3.4):

LASER for RKE Both $\mathcal{D}$ and $\mathcal{F}$ follow the next protocol:

1. $\mathcal{F}$ takes the current timestamp t_{start}, and computes $h = Hash(\mathsf{sk}, t_{start})$.
2. $\mathcal{F}$ sends h over ch along with the real timestamp t_{start} and the command cmd.
3. As soon as $\mathcal{D}$ receives the message sent in the last step, it gets the current timestamp t_{end}, and checks if the difference between t_{start} and t_{end} is lower than or equal to a threshold γ, previously estimated.
4. If the above condition is true, and h is correct, $\mathcal{D}$ executes cmd.

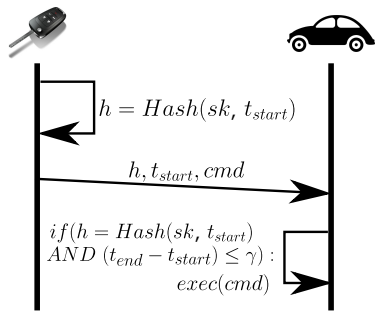


Figure 3.4: LASER for RKE

Accurate time synchronization between $\mathcal{F}$ and $\mathcal{D}$ is crucial, as $\mathcal{F}$ has to send an exact timestamp. To overcome this drawback, we propose the usage of the same approach we introduced in our protocol: if $\mathcal{F}$ sends a timestamp t_{start} that does not verifies $(t_{end} - t_{start}) \leq \gamma$, $\mathcal{D}$ replies with a message h_{sync}, t_{sync}, where t_{sync} is the correct timestamp and $h_{sync} = Hash(\mathsf{sk}, t_{sync})$. $\mathcal{F}$ updates

its real-time clock after verifying h_{sync}. The purpose of sending also a hash here is to prevent an adversary from being able to send messages to $\mathcal{F}$ to modify its current time.

3.5.2 Security Analysis

Preventing jamming-and-replay in RKE. To prevent jamming-and-replay, our solution sends a unique hashed value h of a string. Such string results from concatenating a secret key sk and the current timestamp t_{start} at the moment the protocol is initiated. Like this, each hash will be unique in time and will be accepted by the receiver just at that moment. Plus, the fact of concatenating a secret key makes it impossible for an adversary $\mathcal{A}$ to generate a new hash.

Preventing relay attack in RKE. We first need to estimate the threshold γ, which is the maximum amount of time a message should take going from $\mathcal{F}$ to $\mathcal{D}$. In this scenario, if a message took an amount of time $(t_{end} - t_{start})$ higher than γ, we could say that $\mathcal{F}$ is placed further from $\mathcal{D}$ than what it should be and that the protocol is performed by means of a relay attack, using an LTE network or similar.

Preventing DoS in RKE. Both endpoints have a range of frequency channels N_c available to perform the frequency-hopping protocol, and the aim is to agree on a channel ch without an adversary being able of knowing it. The purpose is to change the transmitting channel each short period of time p (let us say, 10 seconds), which should be defined by the manufacturer considering the best performance of the device. By doing this, an adversary willing to perform a DoS attack against us will have to jam a wide range of frequencies at the same time. It can be done by means of several jamming devices, which is an expensive investment[6].

3.6 LASER for PRKE

In this subsection, we first explain all the steps of the PRKE protocol in detail, and later we introduce the main approach used to prevent each kind of attack.

3.6.1 Protocol Decription

In this scheme, it will be $\mathcal{F}$ who is continuously performing the frequency-hopping protocol. When the owner of $\mathcal{D}$ wants to unlock it by pulling the

[6] https://www.jammer-store.com/hpj16-all-frequencies-jammer.html

handle, $\mathcal{D}$ calculates ch by first calculating h, but rounding the timestamp to the previous multiple of p. Then, the next protocol is performed (as depicted in Figure 3.5):

LASER for PRKE Both $\mathcal{D}$ and $\mathcal{F}$ follow the next protocol:

1. $\mathcal{D}$ sends over ch a synchronization message to $\mathcal{F}$ including the device_id. At the moment it sends the message, it also starts to calculate a message exchanging time t_e.
2. $\mathcal{F}$ computes and sends $h = Hash(\mathsf{sk}, t_p)$ to $\mathcal{D}$.
3. As soon as $\mathcal{D}$ receives the message sent in the last step, it stops the counter of t_e. Like this, now $\mathcal{D}$ knows a value t_e which is the time between $\mathcal{D}$ sending a message and receiving a response. If the received value h is correct and t_e is lower than or equal to a threshold γ, $\mathcal{D}$ executes the desired action.

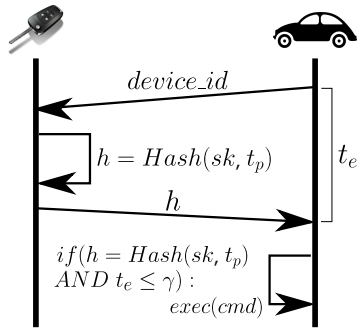


Figure 3.5: LASER for PRKE

In PRKE, if $\mathcal{D}$ does not receive a response after sending the first message of the protocol, it can be that t_p on $\mathcal{F}$ is incorrect. In this case, $\mathcal{D}$ must send h_{sync}, t_{sync} using all the other frequencies, to be able to reach the one used by $\mathcal{F}$, and make it update its current time.

3.6.2 Security Analysis

Preventing jamming-and-replay in PRKE. To prevent jamming-and-replay, in PRKE we also send a unique hashed value h of a string. Although we also

compute h concatenating sk and a timestamp, in this case, the latter is slightly different. For PRKE the prevention against relay attacks is based on another approach we explain in the next paragraph, and this is the reason why we can use the timestamp t_p calculated during the frequency hopping protocol as the value concatenated to sk. Like this, each h can be used only during a short period of time p, thus preventing jamming-and-replay.

Preventing relay attacks in PRKE. The value t_e is the time it takes a message to go from $\mathcal{D}$ to $\mathcal{F}$, plus a response message to go back to $\mathcal{D}$. By placing $\mathcal{F}$ next to $\mathcal{D}$ and pulling the handle of the car, we can calculate an estimated value γ, which is the threshold the protocol should never surpass. If a message took an amount of time t_e higher than γ, we could say that $\mathcal{F}$ is placed further from $\mathcal{D}$ than what it should be, and that the protocol is performed by means of a relay attack. As in this case is $\mathcal{D}$ who calculates t_e, $\mathcal{F}$ will not be required to calculate the current timestamp t_{start}, thus the protocol will be less time and power consuming for it.

Preventing DoS in PRKE. For PRKE systems, the prevention against DoS attacks works essentially like in RKE systems.

3.7 Estimating the Threshold

In this section, we estimate the threshold γ using our proof-of-concept[7], and then we use it to analyze the robustness of both systems against relay attacks. For each system RKE and PRKE we have tried to execute a command one thousand times. The success rate has been 100% in both cases, meaning that the command has been always executed. By logging the timestamps into a dataset, we have found out that the time it takes for a message to go from one endpoint to the other one is never higher than $t_{max} = 136\ ms$ for the RKE solution, as shown in Table 3.1. For PRKE systems, where the calculated time is how much it takes $\mathcal{D}$ to receive $\mathcal{F}$'s reply, the maximum time it took has been $t_{max} = 175\ ms$.

At this point, we could think about the possibility of choosing the maximum value as the threshold. However, it could be dangerous if a relay attack is performed: for the RKE system, if the message takes the minimum time $t_{min} = 55\ ms$ to go to the adversary $\mathcal{A}_1$, and the second adversary $\mathcal{A}_2$ gets to send the relayed message in the same amount of time, it would take 110 ms. Assuming that the adversaries will not be able to exchange the relayed message

[7] https://github.com/xevisalle/laser

Table 3.1: Information extracted from timestamps of RKE and PRKE systems, expressed in milliseconds.

System	t_{max}	t_{min}	t_{avg}	t_{Q3}
RKE	136	55	71	79
PRKE	175	113	157	164

through an LTE network or similar in less than $t_{max} - 110 = 26\ ms$ is a weak premise. To solve this, we could take the average amount of time, but then we are compromising the usability of the system, so most of the time the user will have to press the button more than once, as shown in Figure 3.6. We can overcome this problem by calculating the third quartile of the dataset, which is higher than the average in both RKE and PRKE systems. We can see in Figure 3.6 that now the effectivity is higher as well. As every time we press the button in the fob we are sending around 6 messages, the probability of failing when trying to execute a command is almost negligible, so the success rate for each message is almost 75%.

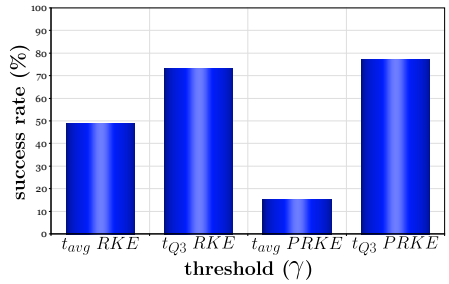


Figure 3.6: Success rate when trying to execute a command in both RKE and PRKE systems considering different thresholds.

3.8 Robustness against Relay Attacks

Let us have an RKE relay attack scenario as depicted in Figure 3.7. If the minimum time it can ever take for the user's hardware to send a message from $\mathcal{F}$ to $\mathcal{D}$ is t_{min}, we can be sure that $t_{FA_1} = t_{min}$ is the minimum value that can be achieved. As such, our scheme is secure as far as the adversaries are not able

to achieve the following statement:

$$\begin{aligned} t_{FA_1} + t_{A_1A_2} + t_{A_2D} &\leq \gamma \\ (t_{A_1A_2} + t_{A_2D}) &\leq \gamma - t_{FA_1} \\ (t_{A_1A_2} + t_{A_2D}) &\leq \gamma - t_{min} \end{aligned} \tag{3.1}$$

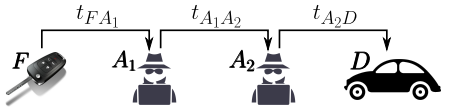


Figure 3.7: RKE relay attack scenario.

On the other hand, we have a PRKE relay attack scenario as depicted in Figure 3.8. If the minimum time it can ever take for the user's hardware to send a message from $\mathcal{D}$ to $\mathcal{F}$ and send the answer back to $\mathcal{D}$ is t_{min}, we can be sure that $(t_{DA_2} + t_{FA_1}) = t_{min}$ is the minimum value that can be achieved. As such, our scheme is secure as far as the adversaries are not able to achieve the following statement:

$$\begin{aligned} t_{DA_2} + t_{A_2A_1} + t_{A_1F} + t_{FA_1} + t_{A_1A_2} + t_{A_2D} &\leq \gamma \\ (t_{A_2A_1} + t_{A_1F} + t_{A_1A_2} + t_{A_2D}) &\leq \gamma - (t_{DA_2} + t_{FA_1}) \\ (t_{A_2A_1} + t_{A_1F} + t_{A_1A_2} + t_{A_2D}) &\leq \gamma - t_{min} \end{aligned} \tag{3.2}$$

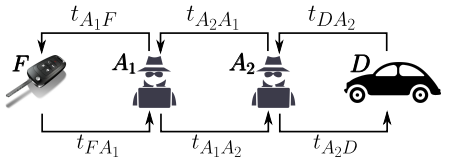


Figure 3.8: PRKE relay attack scenario.

If we take as an example the results we got, the adversaries trying to hack `LASER` should achieve the next statements to succeed, where $\gamma = t_{Q3} = 79\ ms$ and $t_{min} = 55\ ms$ for RKE:

$$(t_{A_1A_2} + t_{A_2D}) \leq 24\ ms \tag{3.3}$$

And $\gamma = t_{Q3} = 164\ ms$ and $t_{min} = 113\ ms$ for PRKE:

$$(t_{A_2A_1} + t_{A_1F} + t_{A_1A_2} + t_{A_2D}) \leq 51\ ms \tag{3.4}$$

For what concerns the bridge between $\mathcal{A}_1$ and $\mathcal{A}_2$, ideally it could be done through an LTE network or similar. Knowing that the average uplink latency in LTE networks is 10.5 ms [62], we could assume two adversaries getting lower values for $t_{A_1A_2}$ and $t_{A_2A_1}$. Even so, assuming that a relay attack can be successful against `LASER` is a strong premise.

Chapter 4

ZPiE: Zero-knowledge Proofs in Embedded systems

Beyond anonymous cryptocurrencies, ZKPs are also used in *smart contracts* to grant more privacy to its users: for instance, issuing a payment after using some service, while keeping the identity of the user secret. Protocols based on such approaches would be desirable in IoT scenarios, where many services are used, and most of them (such as medical applications) collect sensitive data. However, ZKP schemes require high computational resources, especially when the proof is generated. Even when current ZKP implementations do a great job in terms of efficiency, they are far from being portable, especially for embedded systems, as most of them focus on web applications, with the usage of programming languages like JavaScript, WebAssembly, or Python. Some solutions [63] try to distribute the operations to be done to generate the proof between trusted servers. However, such approaches require a trusted environment and a persistent and fast connection with the server. Optimal solutions would be novel implementations focused on devices with low resources.

We introduce `ZPiE`, a portable C library for generating and verifying ZKPs. As an initial proof-of-concept, `ZPiE` implements the specific ZKP construction called zk-SNARK. Our library provides a clear API to create proofs, and to verify them. Upon doing several tests on different devices, we have proved that, unlike other state-of-the-art solutions, `ZPiE` can be executed in `x86`, `x86_64`,

`aarch64` and `arm32` CPU architectures out-of-the-box. In addition, after performing several experiments using a `x86_64` CPU (compatible with other state-of-the-art libraries) and comparing the results with other solutions, we have proved that `ZPiE` has a similar performance. Furthermore, our solution can be easily integrated with existing implementations to be used in smart contracts.

In this chapter, we provide all details regarding `ZPiE`. We start with a comprehensive explanation of how it was designed, to later move to the technical details. We later provide the techniques used to improve the efficiency of the code. Finally, we provide an explanation of how to use the library.

4.1 Related Work

To develop ZKP applications, libraries like the one provided in this thesis are required. One of the main libraries to accomplish this purpose is **libsnark**[1], a C++ library for constructing zk-SNARKs, which was used for some time by Zcash [21], based on the specific zk-SNARK construction introduced in [22], but supporting [23] as well, among others. Even when this library provides excellent benchmarks, one of the main drawbacks of this library, as the authors state, is not being well-optimized for ARM architectures.

Another library with similar benchmarks is **bellman**[2], implemented in Rust and meant for constructing zk-SNARKs, developed and currently used by Zcash.

Moreover, when it comes to developing DApps for the Ethereum Blockchain, we previously stated that a verifier coded in Solidity is required. **ZoKrates**[3] is a python toolbox for zk-SNARKs intended to generate Solidity verifiers, to be deployed into the Ethereum Blockchain. Furthermore, a similar approach is **snarkjs**[4], a JavaScript library for constructing zk-SNARKs. It includes a clear API for generating trusted setups using a fairly secure MPC protocol, for generating proofs, and for verifying them. Plus, it also provides an easy way to export the verifier in Solidity, to deploy it into the Ethereum Blockchain.

[1] https://github.com/scipr-lab/libsnark
[2] https://github.com/zkcrypto/bellman/
[3] https://github.com/Zokrates/ZoKrates
[4] https://github.com/iden3/snarkjs

4.2 Solution Overview

`ZPiE`[5] has been designed to provide a clear interface for developing ZKP applications. As such, it provides an API to design circuits, perform the setup phase, generate proofs, and verify them. Furthermore, our solution can easily integrate the verifier in Solidity applications (i.e. using ZoKrates or snarkjs), in order to deploy smart contracts into the Ethereum network. `ZPiE` can be used in devices with very low resources like a Raspberry Pi Zero, and in a wide variety of CPU architectures: `x86`, `x86_64`, `aarch64`, and `arm32`.

Our solution has been designed with portability and scalability in mind. For such a reason, we decided to use C to code it, which is still widely used in embedded systems, and allows us full control over the hardware resources. For dealing with big numbers we use GNU GMP[6], which is one of the fastest approaches to do operations with large numbers in C. For the group operations over elliptic curves, and pairings, we rely on the MCL library[7], a well-optimized set of functions that offer us support for all the operations we need to perform. Both GMP and MCL offer support for `x86`, `x86_64`, `aarch64`, and `arm32` CPU architectures. Moreover, all the code has been designed to split the workload into threads, to increase the performance especially when the prover is executed in a multicore CPU. Regarding the circuit design, both the circuit parser and the circuit' code developed by the user are coded in pure C, so they are both compiled altogether along with all the other code for maximum performance.

4.3 Efficiency

The first step to being able to use our proof system is to generate the CRS through the setup algorithm. Using Groth'16, the setup has a complexity of $O(n)$, so we need to compute a number of elements that depends only on the size of the circuit. Regarding the verifier, where the complexity time is constant ($O(1)$), they only need to compute 3 pairings and verify that an equation holds, which is not expensive in terms of power consumption. Plus, the operations which require more power consumption are done by the prover when computing the proof. That is computing the h coefficients, and doing the multi-exponentiations in $\mathbb{G}_1$ and $\mathbb{G}_2$.

[5] https://github.com/xevisalle/zpie
[6] https://gmplib.org/
[7] https://github.com/herumi/mcl

4.3.1 Computing h coefficients

As stated in Appendix B, we rely on an FFT function to compute the coefficients in $O(n \log n)$. Our FFT function has been designed to be as efficient as possible. The size of the domain used for the FFTs in $N_e = 2^l$, where l is a big integer, so $|h| = N_e$. In total, the prover has to perform 3 inverse FFTs over a domain S, 3 FFTs over a shifted domain T, and one last inverse FFT over T. As depicted in Figure 4.1, this set of operations is, for any number of constraints, the same small percentage of all the operations performed to generate the proof.

4.3.2 Multi-exponentiations

Our solution uses three different multi-exponentiation approaches: the naive multi-exponentiation (which is a serial approach), the multi-exponentiation function provided MCL, and a Bos-Coster multi-exponentiation algorithm (further details provided in Appendix C). While the former achieves the worst results, MCL gets better marks. However, our Bos-Coster implementation achieves the best results. We split the Bos-Coster operations into chunks to increase the performance when using multithreading. As depicted in Figure 4.1, the multi-exponentiations represent a huge percentage of the operations to be done. In addition, the heap sorting, a step required after each Bos-Coster execution, increases exponentially in terms of global percentage.

4.4 Applications

Some of the use cases of ZKPs involve proving to another party that we know the preimage of a hashed value, or that we have a valid signature for a secret message. We implemented both approaches using `ZPiE`, and provide an API to easily use them.

To use our library, we need to write a circuit that will be parsed later. To do so, Listing 4.1 shows an example of how to compute the MiMC [64] hash of a preimage $x_i n$. As can be seen, we set a public output h, the hash of the secret preimage $x_i n$ (in such a process some randomness k is used as well). In other words, this circuit will compute a proof that once verified, the verifier will be sure that who computed the proof knows $x_i n$. As can be seen in Listing 4.2, `ZPiE` can compute the setup, the proof, and verify it by simply executing each algorithm as shown in the snippet.

In addition, and as shown in Listing 4.3, `ZPiE` can easily verify an EdDSA [65] signature by calling the function `verify_eddsa(...)`, providing the required

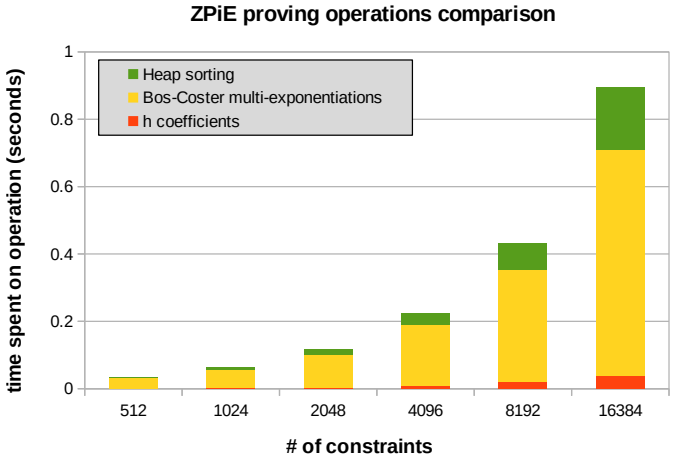


Figure 4.1: CPUs proving operations workload comparison of ZPiE executed in a i7-11370H CPU in single-thread mode. The used zk-SNARK is Groth'16, and the elliptic curve BN128.

parameters as shown in the snippet. In this scenario, the prover is proving to the verifier that they know a valid signature for some secret message. In that regard, the code can be modified depending on the use case, to select which values are public or secret.

```
#include "../circuits/mimc.c"

// main function called by the circuit parser
void circuit()
{
    element h, x_in, k;

    // we init the public output h, and private inputs x_in and k
    init_public(&h);
    init(&x_in);
    init(&k);

    // we manually set preimage and randomness values
    input(&x_in, "1234");
    input(&k, "112233445566");

    // compute a MiMC hash
    mimc7(&h, &x_in, &k);
```

```
}
```

Listing 4.1: Circuit example for computing the MiMC hash of a preimage.

```
#include "../src/zpie.h"

int main()
{
    // we perform the setup (../data/provingkey.params and
    // ../data/verifyingkey.params)
    init_setup();
    perform_setup();

    // we generate a proof (../data/proof.params)
    init_prover();
    generate_proof();

    // we verify the proof (../data/proof.params)
    init_verifier();
    if (verify_proof()) printf("Proof verified.\n");
    else printf("Proof cannot be verified.\n");
}
```

Listing 4.2: Program execution example

```
#include "../circuits/eddsa.c"

// main function called by the circuit parser
void circuit()
{
    element out[4];
    // we init the public output
    for (int i = 0; i < 4; ++i)
    {
        init_public(&out[i]);
    }

    // we provide some example values
    char *B1 = "52996192406415512816348655835182970302
    82874472190772894086521144482721001553";
    char *B2 = "16950150798460657717958625567821834550
    30166316162470778722281593618263896820 3";
    char *R1 = "12629481114452250573734381948187634057
    00457487429548371463214326190311895864";
    char *R2 = "12533500305127747239777484416561675628
    19556206595920173944684166862354088 3587";
    char *A1 = "21629779320182474195265732521833299809
    98244455230514252940923630110499778 6342";
```

```
    char *A2 = "90118124453810306641426220662183318451
    40881847034934166630871421746105699091";
    char *msg = "1234";
    char *signature = "26745918808888623786883838327785
    44719712589720536086195711614716571270945520 7";

    // we verify the signature
    verify_eddsa(out, B1, B2, R1, R2, A1, A2, msg, signature);
}
```

Listing 4.3: Circuit example for verifying an EdDSA signature

4.5 Experiments and Results

In this section, we perform several experiments to prove the efficiency of our solution. In order to do so, we implemented a circuit composed of 16384 constraints as shown in Listing 4.4.

```
// main function called by the circuit parser
void circuit()
{
    element out;
    // we init the public output
    init_public(&out);

    int mulsize = 16384;
    element arr[mulsize];

    // we init an array of secret elements
    init_array(arr, mulsize);

    // we manually set a value
    input(&arr[0], "12345678");

    // do x multiplications
    for (int i = 1; i < mulsize; i++)
    {
        mul(&arr[i], &arr[1], &arr[i-1]);
    }

    mul(&out, &arr[0], &arr[mulsize-1]);
}
```

Listing 4.4: Circuit example for computing 16384 constraints

As stated previously, zk-SNARKs are composed of three algorithms. The setup is performed only once, so the performance of this algorithm is not of big

importance in practice. However, using `ZPiE`, the setup of a circuit of 16384 constraints can be performed in 17 seconds using a laptop CPU in single-thread. Regarding the proof verification, our zk-SNARK construction has a succinct verifier which verifies any proof in just 0.0011 seconds.

Moreover, we performed several experiments to prove the performance of `ZPiE` when computing proofs. First, we compared our solution with a well-optimized library intended to generate proofs in desktop applications, libsnark. As depicted in Figure 4.2, `ZPiE` achieves great results, similar to the ones achieved by libsnark, both in single or multi-thread modes.

Then, as depicted in Figure 4.3, we executed our solution with different constraint amounts in two different processors, either in single and multi-thread modes. As can be seen, the laptop processor i7-11370H (`x86_64`) achieves excellent results either in single or multi-thread modes. Regarding the mobile processor Snapdragon 845 (`aarch64`), the results are still excellent in multi-thread. In single-thread mode, the difference is bigger, yet the proofs can still be executed in a fairly small amount of time.

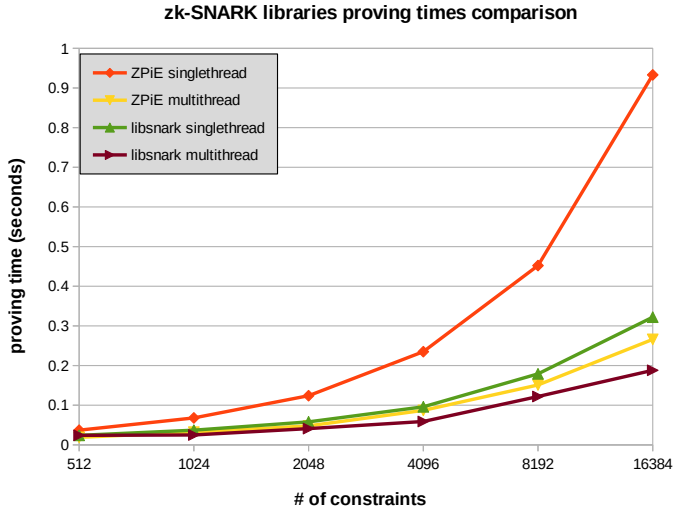


Figure 4.2: ZPiE and libsnark proving times comparison using different constraint amounts, in single-thread and multi-thread modes. All the tests are executed using a i7-11370H CPU. The used zk-SNARK is Groth'16, and the elliptic curve BN128.

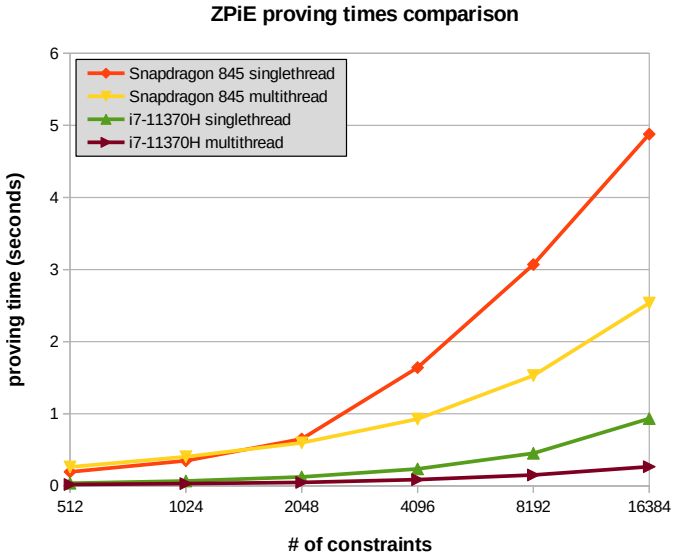


Figure 4.3: CPUs proving times comparison of ZPiE using different constraint amounts, in single-thread and multi-thread modes. The used zk-SNARK is Groth'16, and the elliptic curve BN128.

Furthermore, we successfully computed proofs using a Raspberry Pi Zero W, which CPU architecture is ARM6l (`arm32`) and has a very low clock frequency (700 MHz). As depicted in Figure 4.4, the results are much higher than using mobile or desktop processors, yet the proofs can still be executed. As such, ZKP applications could be executed in embedded devices, at least when speed is not of paramount importance. For instance, protocols such as `SANS`, which needs around 5000 constraints, could be executed in less than a minute using a Raspberry Pi Zero W. This allows IoT devices to use privacy-preserving protocols based on zk-SNARKs.

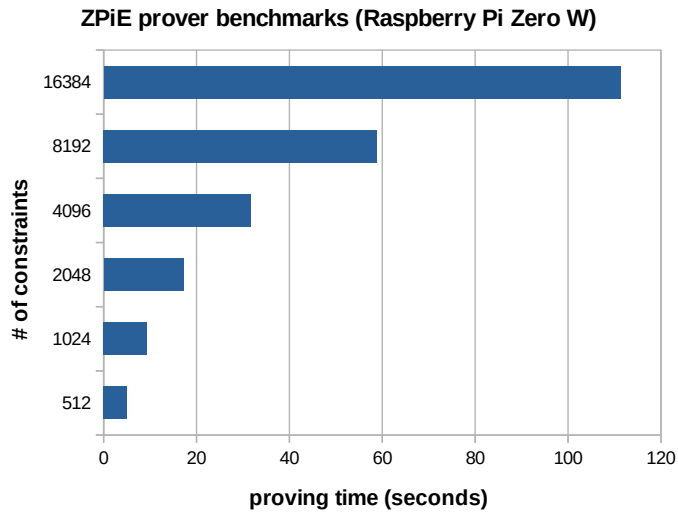


Figure 4.4: CPUs proving times of ZPiE executed in a Raspberry Pi Zero W. The used zk-SNARK is Groth'16, and the elliptic curve BN128.

Chapter 5

SANS: Self-sovereign Authentication for Network Slices

5G communications enhanced the way how mobile devices are connected to cellular networks. They not solely improved the 4G Radio Access Network (RAN), but also introduced a new paradigm where devices with different specifications are routed through different physical and logical networks, called *network slices*. This opened new business models, for instance, creating network slices for specific services offered by third parties. Like this, a Slice Operator (SO) ruling a network slice may want to offer a service to users meeting some requirements (*e.g.*, users enrolled in a governmental program, users who have paid for using such a service, etc.). Among the growing density of IoT devices using 5G communications, we can find examples of devices sharing sensitive data over the network: medical devices exchanging private information or autonomous cars sharing their location with a network slice. Needless to say, this data should not be traced by any SO or eavesdropper. In such a scenario, traditional authentication schemes leak all this data to the SO. As such, SSI [4] becomes an important feature to implement: systems where users can control, access, and transparently consent to their identities, preventing entities from tracking and gathering their personal data. Likewise, the main idea behind SSI systems is to provide a unique mechanism for users to authenticate into different services, providing only the required information, information that shall be non-traceable.

We introduce SANS, a novel self-sovereign authentication approach where a user demonstrates his right to access a service, without leaking any information about them. Our approach is an underlying protocol to be integrated into existing SSI systems, avoiding any user activity being linked with any other activity done in the past or the future. Moreover, it also prevents the SO or an attacker from impersonating them from tracking users' activity. Our protocol grants the user with these main features:

- **Proof of Ownership:** the user can prove that they meet the requirements needed for using a specific service.
- **Unlinkability:** the SO has no way to relate any user activity with another activity done in the network.

We use ZKPs to achieve the aforesaid key features, allowing a user to prove their right to access a specific service, requested by an SO, without leaking any information about them.

5.1 Mobile Communications

5G is the fifth generation of mobile communications [5], which achieves faster speeds than LTE networks and more reliable service. The 5G network is split into different network slices, which are independent networks dedicated and optimized for specific services. This new architecture is built employing Software-Defined Networking (SDN) and Network Functions Virtualization (NFV), along with the physical infrastructure. All these changes lead to higher performance: higher speeds, lower delays, and much less network latency. As depicted in Figure 5.1, different kinds of User Equipment (UE) are part of different slices, depending on their specifications or the services they are willing to use. In a nutshell, the main network slices are:

- **eMBB slice:** The enhanced Mobile Broadband (eMBB) slice is meant for services that require high bandwidth, like Internet browsing, high-definition video streaming, virtual reality, etc.
- **mMTC slice:** The massive Machine Type Communications (mMTC) slice aims to group a high density of devices, which do not have other essential requirements like low latency or high bandwidth. Examples of this are IoT devices, specifically in the context of smart cities.

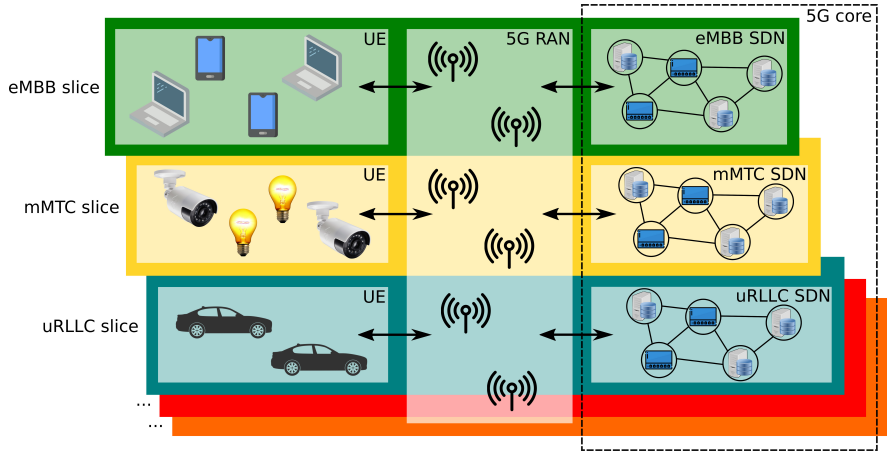


Figure 5.1: General 5G architecture overview.

- **uRLLC slice:** The ultra-Reliable and Low-Latency Communications (uRLLC) slice aims to provide very low network latency, a crucial requirement for services like autonomous driving or remote management.

As depicted in Figure 5.1, users connect their UE to the small 5G cells of the 5G RAN, which forward the connections to the 5G core network, split into different software-defined networks (*i.e.*, eMBB, mMTC, uRLLC...).

Furthermore, access to the 5G core network is allowed not solely from the new 5G RAN, but also from other networks like the 4G RAN or optical fiber connections, depending on the requirements of the service. As such, we understand 5G as a heterogeneous network (HetNet), a network interconnecting devices with different specifications and protocols, where a common and trustworthy authentication scheme would be a desirable feature.

5.2 Related Work

SSI systems have gained a lot of interest in the last few years. The author in [1] envisioned an SSI system where users can control, consent, and widely use their identities among different services, along with other properties. These properties were redefined in [4] by the Sovrin Foundation[1]. They introduced

[1] https://sovrin.org/

the guidelines on how SSI systems can be implemented along with Blockchain technologies, providing a distributed architecture of trust without central authorities managing users' data. In this regard, SSI authentication schemes like the one proposed in [66] make use of Blockchain technologies for deploying a decentralized and private authentication system.

A good review of the state-of-the-art regarding this topic is done in [3]. As they state, ZKPs allow a user to prove ownership of an identity, *i.e.*, proving knowledge of a secret key related to a public key stored in a Blockchain.

As stated previously, the core of network slicing relies on an SDN-based architecture. In this regard, interesting research is addressed in [67], where a novel authentication scheme preventing multiple types of SDN authentication attacks is introduced. This makes even more sense in the context of a medical cloud sharing sensitive information, a fact that has led to schemes [68] guaranteeing a secure authentication in this scenario.

A more specific use case related to our approach is introduced in [69]. They state some of the benefits of SSI for IoT devices, like the fact that the identities of the owners of different devices are stored locally in the devices, rather than on a centralized entity (*i.e.*, the SO in our scenario). As explained by the authors, SSI provides a layered authentication system separating application authentication from channel authentication, where the former handles the trust requirements. This grants a more reliable end-to-end security, where secure communication is established among different protocols.

Among the aforesaid studies regarding SSI, to the best of our knowledge, there are no solutions applied to 5G network slices. In this regard, we propose a solution to integrate SSI into network slices in the next section.

5.3 Protocol Description

We start with a high-level description of `SANS`, and later move to a more detailed one: a user willing to join a network slice to use its service may be required to meet some requirements, like having paid a subscription fee. As such, the user is a prover $\mathcal{P}$ willing to prove to a verifier $\mathcal{V}$, the SO, that they have paid such an amount (the statement). Our protocol accomplishes this purpose. To do so, an important requirement of our protocol is being able to prove knowledge of contracts signed using a given secret key: $\mathcal{P}$ must convince $\mathcal{V}$ that they know a contract and its signature, which is verified using a public key. The contract can be a secret value, and still, $\mathcal{V}$ must be convinced. In order to be efficient, the used signing algorithms have to be ZKP-friendly, and this means

that its operations can be reduced to a low number of constraints. For instance, the Edwards-curve Digital Signature Algorithm (EdDSA) [65] is a fast signing algorithm widely used with zk-SNARKs. Moreover, signature algorithms in zk-SNARKs must be combined with efficient hashing functions as well. One of the most efficient zk-SNARK-friendly hashes to the date is Poseidon [16], which needs 8 times fewer constraints for its circuit than the widely used Pedersen hash.

Our authentication scheme is divided into two protocols, depicted altogether in Figure 5.2. The first one is the *service registration protocol*, to be performed for each issued payment.

SANS SERVICE REGISTRATION PROTOCOL This protocol allows the user to register to use a given service. Its steps are as follows:

1. $\mathcal{P}$ provides $\mathcal{V}$ some requested information req (*e.g.*, a statement from the bank stating that a payment has been issued).

2. After verifying req, $\mathcal{V}$ generates a unique byte-array token identifying the user, and sends it to them along with a timestamp t_{exp} representing the contract expiration date. Moreover, $\mathcal{V}$ provides a signature $S = \mathsf{sign}_{\mathsf{sk_{SO}}}(\mathsf{token}, t_{exp})$ and its public key $\mathsf{pk_{SO}}$.

After having registered into the service, the user can use the provided parameters to authenticate into the service each time they needs to use it, and thus, create a new session into the service. Moreover, in order to avoid replay attacks [70] (*i.e.*, an eavesdropper taking the proof and replying it to the SO), every proof must include the hash of the secret token concatenated to a variable public parameter c. Further details of such an approach are discussed in Section 5.4.

SANS SESSION AUTHENTICATION PROTOCOL This protocol is meant to be performed each time the user wishes to prove their right to use a service, following the next steps:

1. P computes a proof π out of the circuit depicted in Figure 5.3, whose inputs are:

   - c (**public input**)
   - token (**private input**)
   - S (**private input**)

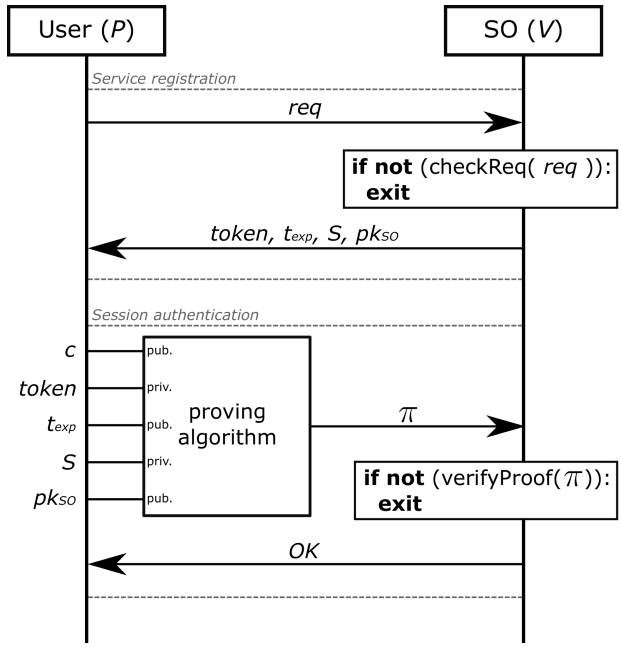


Figure 5.2: Overview of the service registration and the session authentication.

- $\mathsf{pk}_{\mathsf{SO}}$ (**public input**)
- t_{exp} (**public input**)

2. V verifies the proof π and grants the service.

As shown, we prove knowledge of a secret token concatenated to its expiration date t_{exp}, which is the preimage of a public hash. This is our contract, and we also prove that we know its secret signature (signed by $\mathcal{V}$) using the public key $\mathsf{pk}_{\mathsf{SO}}$. This outputs 0 if the signature is verified.

5.4 Security Analysis

In this section, we analyze the security of our solution. We also detail how to overcome some possible attacks.

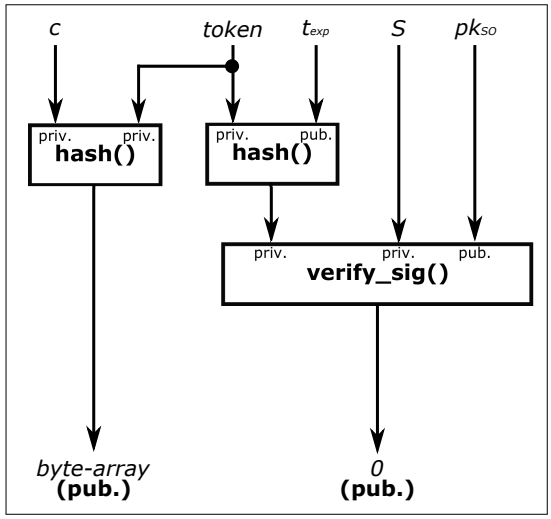


Figure 5.3: Circuit used by the session authentication protocol.

False proofs generation. The main drawback of some ZKP constructions like zk-SNARKs is the need for a trusted setup. In many scenarios, like in Zcash, an untrusty setup could lead to huge losses of money if a malicious party gets the trapdoor τ and starts to create false transactions. However, this is not a problem in our solution: a different setup can be generated by each SO. If the SO keeps and spreads the trapdoor τ, anyone knowing τ will be able to access the service by generating false proofs. As such, the protocol is secure as long as the setup is generated only by the SO and they destroy τ. Furthermore, as stated previously, the ZKP construction that best fits our solution at the moment of writing this is the Groth'16 zk-SNARK. As such, the security of `SANS` depends on a q-PKE assumption.

Elliptic curve attacks. The security of our solution also relies on the security of elliptic curves. One of the most used curves in ZKPs is a Barreto-Naehrig curve [13] called BN128, which security level in practice is estimated to be 110-bits [34]. This means that an attacker willing to break BN128 shall perform 2^{110} operations. Other curves like BLS12-381 [21] estimate around 128-bits of security, with the drawback of heavier group operations. Breaking the security of the used elliptic curve would lead to being able to generate false proofs.

Account sharing. Every computed proof is different since it is generated using random parameters, allowing the user to generate different proofs with the same inputs. As such, the user could generate multiple proofs for other users, which would access the service with a single subscription. To overcome this issue, a simple solution is integrated into our protocol: every proof must include the hash of the secret token concatenated to a variable public parameter c. Ideally, this parameter could be a timestamp with a specific accuracy, for instance, the date in format yyyy/mm/dd plus the time in format hh:mm without seconds. Plus, the signature verification will output 0 if correct, and thus, the hash as well. Like this, an SO receiving the same hash more than once could identify that those proofs have been computed using the same token. As such, if two users are trying to use the service at the very same time, the SO can relate and reject both connections.

5G RAN authentication. One of the main concerns about our solution is to provide a fully private authentication, where the SO cannot learn the identity of the user. In this scenario, we still have another party, the Internet Service Provider (ISP), who acts as an SDN controller providing the architecture and the workflows for optimal network slicing. As such, the ISP learns the identities of the users from the moment that the UE accesses the 5G RAN. To overcome this, we envision the usage of `SANS` when the UE is required to authenticate for accessing the 5G RAN. In other words, the UE would be proving his right to access the 5G RAN, for instance by proving that the user has paid the last month's bill to the ISP.

5.5 Efficiency Analysis

This section describes several efficiency considerations of `SANS`.

Computational complexity. The setup protocol depends only on the number of gates, so this protocol has a linear computing complexity $O(n)$. One of the most consuming operations done by the prover is to compute the coefficients of a polynomial $H(x)$, which can be computed more efficiently employing FFT techniques [71], leading to a computing complexity of $O(n \log n)$. Furthermore, multi-exponentiations must be taken into account as well, and can be improved using algorithms like Bos-Coster, as explained previously. The verifier has to do a constant computation of group exponentiations and an equation composed of three pairings.

Prover optimizations. There are different operations performed by the zk-SNARK prover which can be parallelized in order to improve its efficiency. This

means that CPU and GPU multiprocessing techniques can be applied to speed up the implementations. Even so, the usage of external computing resources as done in [72] can be taken into account. For instance, in the case of a prover being a smartwatch with low computing resources, the heaviest computations could be precomputed by the user's phone, whose computing power should be higher.

Circuit size. Our circuit contains a single EdDSA signature combined with two hashes (to the date of writing this, Poseidon seems the best option). The authors of *circomlib*[2] developed optimal EdDSA and Poseidon circuits, which leads our solution to a total size of 7565 constraints and affordable computing times as shown in the next subsection.

5.6 Implementation and Benchmarks

We implemented[3] our solution using *snarkjs*, a JavaScript and WASM framework for implementing zk-SNARK applications. The reason for choosing this option is its simplicity for implementing circuits and its portability in web environments. In this regard, we deployed our implementation in a web server, to be executed by different devices using different web browsers. Overall, the number of constraints of this implementation is 7565, and as depicted in the chart of Figure 5.4, our solution outperforms in high-performance CPUs (i7-8750H), either using Mozilla Firefox or Google Chrome. As such, our solution could be used in desktop applications with no problems with regard to performance.

On the other hand, the proving time increases notably in low-performance processors (Intel Atom x7-Z8750), achieving timings higher than 2 seconds both in Firefox and Chrome. An interesting fact is how Chrome performs slightly better than Firefox in its desktop version, which does not apply to mobile CPUs (Snapdragon 845). Regarding Snapdragon 845, even when it is a top mobile processor, we can see as the results are not as good as i7-8750H. However, the achieved results prove that our solution is feasible in performance, especially when portability is a priority. Moreover, the memory consumption has been in all tests between 150 and 200MB (not taking into account what is consumed by default by the browsers).

Furthermore, we also tested *libsnark*[4], a well-optimized C++ zk-SNARKs library achieving excellent benchmarks, but with the drawback of not being as

[2] https://github.com/iden3/circomlib/
[3] https://github.com/xevisalle/sans
[4] https://github.com/scipr-lab/libsnark

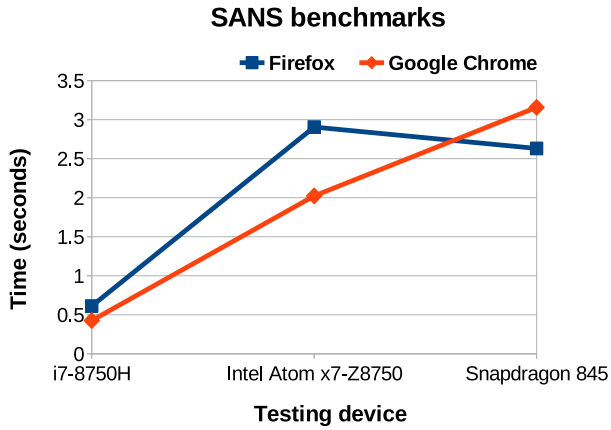


Figure 5.4: CPUs proving times comparison of SANS using a *snarkjs* implementation executed in multi-core mode for browsers. The used zk-SNARK is Groth'16, and the elliptic curve BN128. Intel Atom x7-Z8750 and i7-8750H run desktop browsers for Linux, Snapdragon 845 runs Firefox and Chrome for Android 10.

portable as other solutions like snarkjs. For instance, as the authors of libsnark state, the library is not well-optimized for ARM architectures (*e.g.* Snapdragon 845), and the BN128 curve is not supported in this architecture.

We implemented a circuit with the same amount of constraints that our solution has, and we executed the prover in multi-core mode using Groth'16 and the BN128 curve. The obtained results are shown in the chart of Figure 5.5. As can be seen, libsnark achieves much better results than snarkjs, so implementing `SANS` using this library would be even more feasible. Regarding memory consumption, libsnark performs better as well: around 20 MB in both tested devices. Furthermore, optimized libraries for mobiles and embedded systems would lead to additional performance improvement, so future work in this regard would be an exciting research topic.

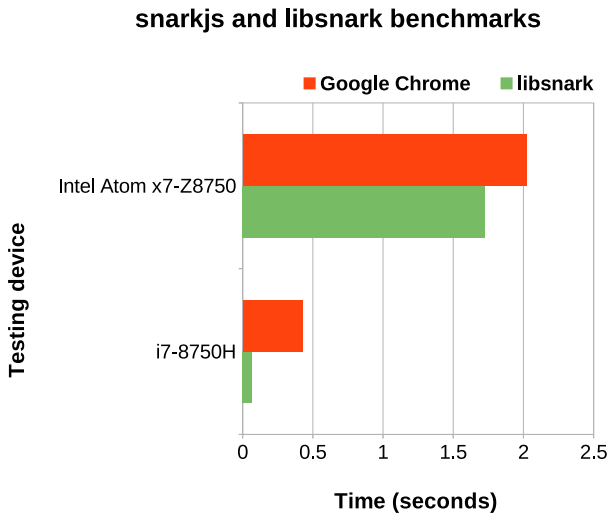


Figure 5.5: CPUs proving times comparison of *snarkjs* and *libsnark* 7565 constraint circuits executed in multi-core mode. The used zk-SNARK is Groth'16, and the elliptic curve BN128. Intel Atom x7-Z8750 and i7-8750H run a desktop Linux distribution.

Chapter 6

FORT: Right-proving and Attribute-blinding Self-sovereign Authentication

A high density of devices is also translated to more data shared over the network. A concerning fact is what happens with the data shared by users, especially when such data is sensitive or can simply be used to profile users with no permission. Even when the usage of Internet technologies is increasing very fast, some security and privacy concerns [73, 74] still need to be addressed. For instance, medical devices sharing sensitive information about patients through the network, GPS applications, or autonomous driving, are applications that collect a lot of data about us. Even if the company behind says no personal data is collected, we can only trust them, with no possibility of detecting misbehavior.

In this context, new digital services have appeared in the market, changing the way how users interact with them. Among many use cases, we can find car sharing, buying tickets for events, subscriptions to streaming services, etc. As centralization was a property of these applications that used to lead to control of the network by some individuals, Blockchains [39] started to change the way people interact with online services. The most common example, cryptocurrencies, has become a payment method without central authorities (i.e. banks) con-

trolling the stream of the issued transactions and all the collateral information. Moreover, beyond being a payment solution, Blockchains like Ethereum [42] offer a way to execute programs *on-chain*. Those programs, called *smart contracts*, allow issuing a payment to a specific party as soon as this party proves that they meet some requirements specified in the contract. This same approach is used in many DApps [75] nowadays, like paying a subscription to some service.

Decentralization implies that public data stored in the Blockchain can be accessed by anyone. This leads to some privacy concerns: as Blockchains publicly store all the network activity, user tracking or profiling becomes an issue to be addressed. In such regard, the problem gets worse when users of a Blockchain-based service need to interact with real-world services (i.e. proving to the staff of an event that you paid for the ticket), so if anyone learns your Blockchain identity, they learn all your history.

To solve the privacy concerns that arose on Blockchain applications, ZKPs started to be integrated within Blockchain projects like Zcash [21]. In the Zcash example, these primitives allow users to issue transactions without leaking their identity or the amount of money they are spending while proving that they are solvent.

In such a scenario, the concept of SSI appeared. In the previous chapter, we introduced `SANS`, allowing users to prove to Service Providers (SP) that they own a token that proves their right to use a specific service. Such a solution is suitable in many scenarios, but in some cases can have efficiency drawbacks since it relies on a ZKP construction called zk-SNARKs [23], which requires high computing power. This scheme is executable on Internet of Things (IoT) devices thanks to implementations like `ZPiE` [76], but taking a fair amount of time. This fact makes such a solution infeasible in use cases where IoT devices must prove several things in a short amount of time (i.e. willing to use a smartwatch to prove a right, having a door sensor with a cheap CPU verifying proofs, etc.). Besides, this solution is still centralized, which means that if the SP disappears, the user no longer owns the right.

In this chapter, we introduce `FORT`, a novel self-sovereign authentication protocol, combined with Blockchain technologies to provide a solution where users of a service acquire *rights*, which are a set of different provable *blinded attributes*. Such attributes are portions of personal information which have been blinded: they are invisible to the SP, and only the user can decide how much information about them has to be leaked. These attributes are represented by NFTs [77] on the Blockchain, which can be granted *on-chain* by entities providing services, the SPs, and verified *off-chain*. For instance, a car willing to access a smart city would have to prove its right to do so, that is having two

attributes: a certificate stating that the car has a low emissions level, and a fee payment receipt for entering the city. Once the right is represented in the Blockchain using an NFT, the car will be able to prove off-chain the possession of such a right, by using a ZKP. Such proof will state the possession of a valid NFT, without leaking the identifier of such NFT nor the identity of the car owner. Furthermore, our solution also skips third-party fees: for instance, in the scenario of buying tickets online for some event, in many cases, the ticket is issued by a third party that handles all the ticketing management, and who needs to be trusted. Furthermore, this party charges the users a service fee. Our solution relies on the Blockchain. Thus, the event organizer does not need to rely on third parties, and the user does not share his identity nor pay a service fee.

Our contribution relies on zk-SNARKs, but also on range proofs, another ZKP scheme where users prove that a value lies within a given range, without leaking such a value to other parties. In particular, we use the Bulletproofs [28] range proofs scheme. For that reason, our second contribution in this chapter is the implementation of a Bulletproofs module for `ZPiE`. Our implementation achieves excellent benchmarks, and using such a module, we implement our protocol and show its efficiency in IoT devices.

6.1 Related Work

SSI systems [1] have the premise of deploying protocols where users of different services can manage their identities in a secure, transparent, and private way. A general idea in this regard, and similar to our solution, was envisioned as a system where users can claim and prove possession of different rights associated with their identities, without compromising their privacy [4]. Furthermore, the combination of SSI systems with ZKPs has become a new research topic in the last few years [3]. In this regard, solutions like `SANS` [78] introduce a private authentication mechanism based on ZKPs. Using such tools, `SANS` allows users to prove their rights to access several services, without the Service Provider (SP) knowing the identity of the users, while guaranteeing that the users are allowed to use the service (e.g. the users have paid a subscription fee).

There are some differences between `SANS` and this work. In all cases, ownership of a given token can be proved (Proof of Ownership). Moreover, this work can prove that the token exists in the Blockchain (Proof of Validity). We also deploy attributes blinding, where our solution becomes completely self-sovereign: users can choose to reveal specific portions of their data in a transparent way.

Regarding privacy in online transactions, other research papers like [79] explore an interesting way to provide a privacy-preserving authentication protocol by means of Physical Unclonable Functions (PUFs), providing a solid and efficient protocol.

Besides, ongoing research regarding how zk-SNARKs can contribute to scalability in Blockchains is done in [80], where research on distributed proofs generation making use of recursive zk-SNARKs is done.

On the other hand, combining IoT devices and NFTs is not an unexplored research area. Recent research [81] introduced a solution to manage IoT devices securely. They associate NFTs stored in Blockchains with IoT devices, to grant them a unique and indivisible identity.

Finally, to the best of our knowledge, there are no other solutions that provide a private-by-design and self-sovereign system to authenticate users, providing at the very same time a decentralized architecture, just like `FORT` does.

6.2 Solution Overview

Our solution is meant to be used in scenarios where users need to prove their right to use a service, for instance, accessing a house rented online. In such a scenario, we envision the usage of a certificate installed in the user's smartphone (or smartwatch, or any similar device) which can be validated by some sensor installed in the door of the house. Once validated, the SP might want a proof of meeting some requirements, linked with the previously validated certificate. We detail this workflow (as depicted in Figure 6.1) in this section, as follows:

1. **Read on-chain information:** the user acquires some attributes granted by third parties, which can be an SP to whom the user is buying a ticket or subscription, a governmental entity verifying your personal information, a bank providing a proof of solvency, etc. Such attributes are granted through an NFT stored in a Blockchain. The SP issues an NFT representing the user's attributes. The SP mints this NFT on-chain, and later transfers it to the user's address. Now, the user can read these attributes from the Blockchain.

2. **Compute proof (the certificate):** the user acts as a prover, and computes a ZKP from the information collected from the NFT, as detailed in the circuit of Figure 6.2, and installs this certificate in his device.

3. **Send proof (read certificate):** The user tries to use the service by showing the certificate to the SP, who reads it.

4. **Verify on-chain information:** the SP needs to partially read the Merkle tree of the Blockchain (as detailed in the next section) to be able to verify (in the next step) that the attributes the user wants to prove are really on-chain (the NFT).

5. **Verify proof (validate the certificate):** The SP verifies the ZKP, thus verifying the rights of the user.

After performing this protocol, the SP can ask the user for some information about the attributes, for instance, if they lay within a specific range. To do it, the user computes a bulletproof and sends it to SP, the verifier. Then, the SP verifies that the bulletproof sent by the user is correct, and knows for sure that the value is within a specific range.

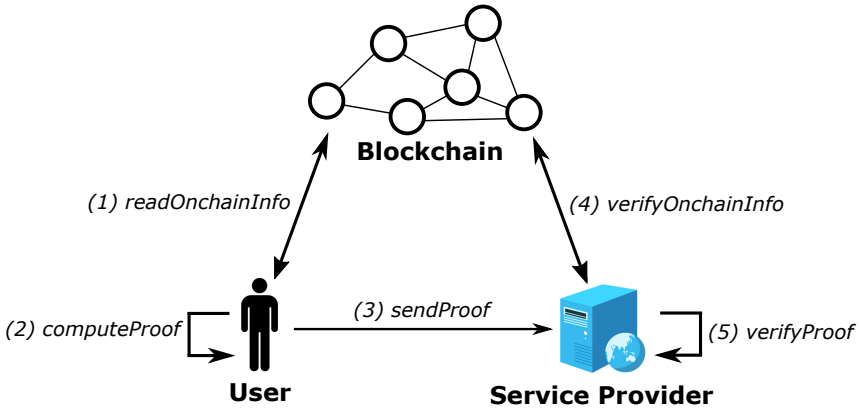


Figure 6.1: FORT protocol scenario overview.

Ideally, a desirable way to implement our protocol would be creating the NFTs directly using a Blockchain, and proving that we own them using the signature details involving the Blockchain transaction representing each NFT. However, this has some constraints regarding scalability and efficiency: first, the gas fees in the case of Ethereum can become very expensive, so using a transaction for a single right is far from optimal. Second, the elliptic curve used to sign Ethereum transactions, the *secp256k1*, is not pairing-friendly, so generating proofs proving ownership of the private key used to sign the transaction will not be efficient. To solve this problem, `FORT` relies on zk-Rollups for scalability, and on EdDSA for proofs off-chain.

6.3 Protocol Decription

To be able to prove rights to access the service, the first thing a user needs to do is to receive an NFT stating some attributes about them. To do so, the user needs to contact the SP and provide them a proof of meeting some requirements. Then, SP executes Algorithm 1: after validating the requirements, it issues an NFT representing the user's attributes. The SP mints this NFT on-chain, and later transfers it to the user's address.

Algorithm 1 Create NFT

Environment: vector of x attributes: $\mathsf{attributes}[x]$; user's address: $\mathsf{pk_{user}}$; user's conditions: cnd

$\mathsf{nft} \leftarrow \mathsf{create\_nft}(\mathsf{attributes}[x])$:
if $\mathsf{verify\_conditions}(\mathsf{cnd})$ **then**
    $\mathsf{nft.id} = \mathsf{rand}()$;
    $\mathsf{nft.attr} = \mathsf{attributes}[:]$;
    $\mathsf{nft.S} = \mathsf{sign}_{\mathsf{sk_{SP}}}(\mathsf{H}(\mathsf{nft.id}, \mathsf{nft.attr}, \mathsf{pk_{user}}))$;
    $\mathsf{mint\_nft}(\mathsf{nft})$;
    $\mathsf{transfer\_nft}(\mathsf{nft}, \mathsf{pk_{user}})$;
end

Upon receiving the NFT, the user is ready to anonymously prove possession of such an NFT, and thus, their rights. To do so, the user will follow the next protocol.

FORT PROTOCOL We have a user willing to use a service, and a Service Provider (SP) offering it. They perform the following steps:

1. (**user**) : Read the NFT transaction to be proved, which is published in the Blockchain, and the IDs of a set of NFT transactions in the batch $\mathsf{batch\_ids}$, where $|\mathsf{batch\_ids}| = 2^x$, and x is agreed by consensus. From this, using the leaf corresponding to the user's $\mathsf{nft.id}$, create a Merkle proof $\mathsf{merkle\_proof}$.

2. (**user**) : Send a proof π to the SP, computed using the circuit depicted in Figure 6.2, along with the $\mathsf{batch\_ids}$.

3. (**SP**) : Receive $(\pi, \mathsf{batch\_ids})$ from the user.

4. (**SP**) : Collect the ID of the transactions in the batch and compute the root of the Merkle tree root.

5. (**SP**) : Execute Algorithm 2, if it returns 1, grant the service, reject otherwise.

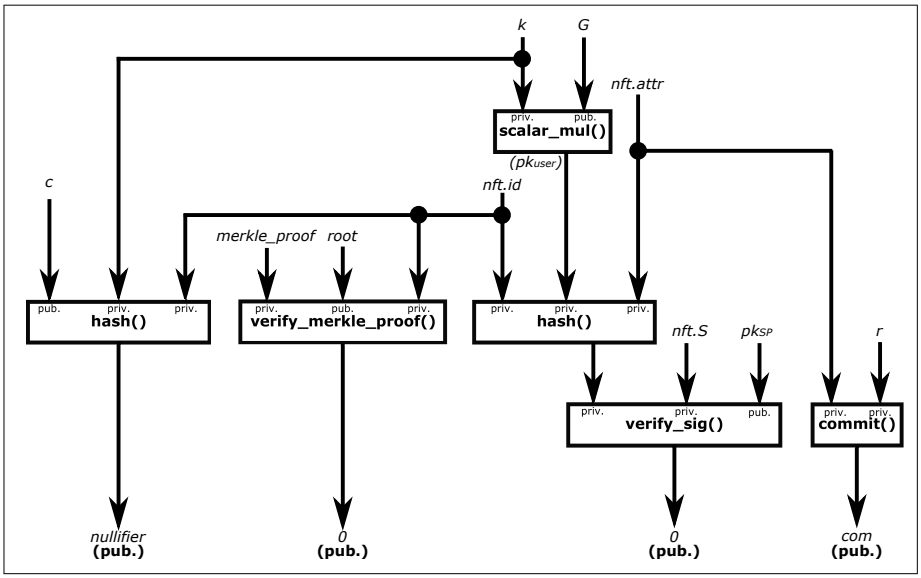


Figure 6.2: Circuit for our solution.

Algorithm 2 Verify right

Environment: Zero-knowledge proof π.

$1/0 \leftarrow \mathsf{verify\_right}(\pi)$:

if $(\mathsf{nullifier}, 0, 0) \leftarrow \mathsf{verify\_proof}(\pi)$ *and* $!(\mathsf{is\_seen}(\mathsf{nullifier}))$ **then**
  return 1;
else
  return 0;
end

As explained above, the proof used in the protocol uses the circuit depicted in Figure 6.2. As can be seen, the circuit includes a verification of the signature $\mathsf{nft.S}$, using the public key of the SP $\mathsf{pk_{SP}}$. The inputs of the signature were the attributes $\mathsf{nft.attr}$ along with the ID $\mathsf{nft.id}$ and the public key of the user $\mathsf{pk_{user}}$.

The challenge c is hashed along with nft.id and the user's private key k. nft.id is also used to verify the merkle_proof.

Finally, and before the SP grants the service, the prover might have to create a bulletproof to reveal some information about the attributes. This is done using com, a Pedersen Commitment. This process is detailed in Algorithm 3.

Algorithm 3 Create Bulletproof

Environment: secret key k; vector of x attributes: attributes$[x]$; vector of x commitments: $C[x]$
$\pi_b[x] \leftarrow$ create_bulletproof$(C[x],$ attributes$[x])$:
for i *in* x **do**
  $\pi_b[i] \leftarrow$ bulletproof$(C[i],$ attributes$[i])$
end

6.4 Security Analysis

An important element to consider when analyzing the security of FORT is the ZKP scheme to use. The main drawback of some ZKP constructions like zk-SNARKs, when used in scenarios like cryptocurrencies, is the need for a trusted setup. An untrusty setup could lead to huge losses of money if a malicious party gets the seed used to compute it, so it could create false transactions. This is not a problem in our solution: a different setup can be generated by each SP, as the proofs are verified off-chain by a single entity, the SP, and they are the main interested in not leaking the secret seed.

Moreover, the soundness property of each scheme relies on different security assumptions, like some zk-SNARK schemes relying on the q-PKE assumption. Furthermore, the security of these schemes relies on the security of elliptic curves, where breaking the security of the selected curve would lead to being able to generate false proofs. One of the most used curves in ZKPs is the BN128, which security level in practice is estimated to be 110-bits [34]. Other curves like BLS12-381 [21] estimate around 128-bits of security, with the drawback of heavier group operations. More recent research is introduced in [82], where a new curve called BW6-761 is introduced. As stated by its authors, the verification of proofs is at least five times faster than other state-of-the-art curves.

Regarding the circuit we have designed, our solution grants several privacy and authentication features:

- **Proof of Ownership:** the circuit used in FORT verifies a signature $\mathsf{nft.S}$ of an input $\mathsf{nft.id}, \mathsf{nft.attr}, \mathsf{pk_{user}}$, using the public key of SP, $\mathsf{pk_{SP}}$. Also, $\mathsf{pk_{user}}$ is the output of the scalar multiplication kG, where k is the user's private key. This ensures that the user owns the NFT, as only they can compute the public key using the private key, while keeping both values private, so SP cannot learn the identity of the user.

- **Proof of Validity:** the user computes a Merkle tree out of a batch $\mathsf{batch\_ids}$, and provides into the circuit a Merkle proof as a private input, for a given $\mathsf{nft.id}$. This ensures that the NFT the user is proving ownership of has been transacted in the Blockchain. SP can compute the root of the Merkle tree $\mathsf{root'}$ itself, and check if it equals root.

- **Unlinkability:** as the values identifying the user are private inputs of the circuit, the SP has no way to link any user activity done in the network with other activities done by the same user.

- **Nullification:** the circuit computes the hash of $\mathsf{nft.id}$, the private key k, and a challenge c. The format of this value could change in different scenarios. Taking the example of proving ownership of a ticket for an event, ideally, c would be the date of such an event. If $\mathsf{is\_seen}(\mathsf{nullifier}, \mathsf{previous}[]) \stackrel{?}{=} 1$ holds, it means that someone already entered the event with the same NFT. This is true because neither $\mathsf{nft.id}$ nor k can change, so $\mathsf{nullifier}$ will always be the same for a given public input c. This prevents a user to use the same right multiple times, and to compute valid proofs for other users.

- **Attribute blinding:** the private information the user wants to share only when required, the attributes, are private inputs of the circuit. Such values are committed using a Pedersen Commitment (i.e. com, but as many as required can be included in the circuit), so the verifier learns these commitments, and the prover later uses a Bulletproof to prove knowledge of them, and to prove that they are within a specific range.

FORT, as introduced in this section, can also be seen as a framework to be modified to match the needs of every use case that our solution could be deployed to. This means, selecting the proper ZKP scheme to be used, recomputing the certificate each time instead of using Bulletproofs, selecting a different challenge c, etc.

6.5 Implementation and Benchmarks

In this section, we explain the capabilities and implementation details of the Bulletproofs module we developed and later explain how we implemented our specific solution using our module.

6.5.1 Bulletproofs Module

We implemented Bulletproofs as a module integrated into `ZPiE`. As explained previously, this library uses GMP and MCL as dependencies: GMP is a pure C library used to handle big numbers and operations involving them, and MCL is a library written in C++, which offers a C wrapper for using it in pure C projects, used to do elliptic curve operations. The library also supports the elliptic curves BN128 and BLS12-381, which are thus also supported by our implementation. We implemented an API that allows us to generate aggregated range proofs using the Bulletproofs scheme above referred, and to verify them. The instructions on how to compile and use the library can be found in the README of the repository. The code can be used as explained in Listing 1.

```
#include "../src/zpie.h"

int main()
{
    // init the bulletproofs module for 2 aggregated proofs of 64
    bits
    bulletproof_init(64, 2);

    // set some values to prove knowledge of and compute the
    bulletproof
    unsigned char *si[] = {"1234", "5678"};
    bulletproof_prove(si);

    // verify the bulletproof (../data/bulletproof.params)
    if(bulletproof_verify()) printf("Bulletproof verified.\n");
    else printf("Bulletproof cannot be verified.\n");
}
```

Listing 6.1: Bulletproof generation example. Generation of 2 aggregated proofs of 64 bits.

We benchmarked our implementation as depicted in Figure 6.3. Moreover, we improved the efficiency of our solution by using multi-threading in several parts of the prover and the verifier, where splitting the operations in different cores was possible. As we can see, we are benchmarking the time it takes by the

prover, either in single-core (SC) or multi-core (MC), to compute the proofs. We performed the experiments for several amounts of aggregated proofs of 64 bits, using a 4-cores CPU, and the BN128.

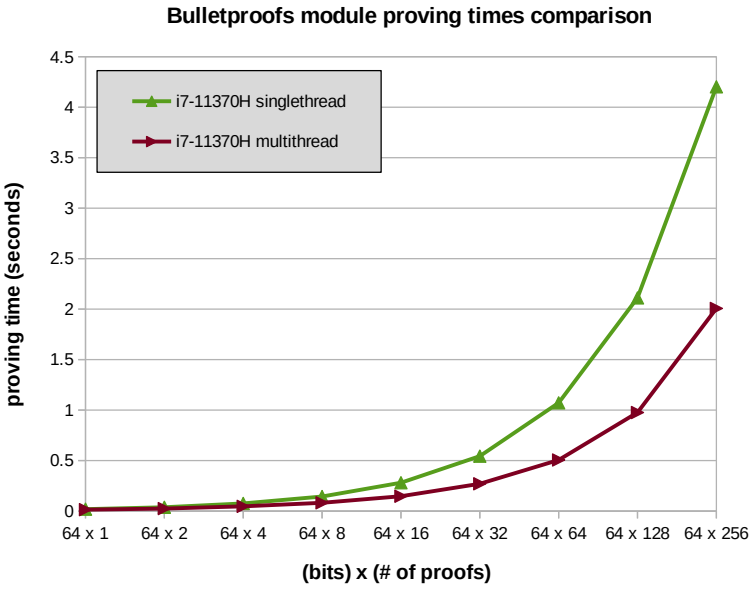


Figure 6.3: CPUs proving times of our solution.

6.5.2 Solution Deployment

In this subsection, we detail the deployment of the three main parts of our protocol, *generate rights*, *generate the certificate*, and *prove the attributes*.

Generate Rights

The first step to using our solution is to generate the rights that our users will need to prove. To do so, an SP needs to provide a service and sell its subscription using an NFT minted to a smart contract-based Blockchain. For testing purposes, we used an Ethereum testnet where we created test NFTs using a reference implementation of ERC-721 (the Ethereum NFT standard)[1].

[1] https://github.com/nibbstack/erc721

After deploying an NFT to the Blockchain, a user can buy it. Once done, they are ready to generate the certificate.

The computational costs for generating the NFTs are negligible, as no heavy cryptographic computations are involved in the process. Regarding the time it takes to be reflected on the Blockchain, it would depend on how crowded it is (typically it will take only a few minutes). On the other hand, one of the main concerns regarding this step when deploying it into the mainnet is the amount of gas required to execute the smart contract that mints the NFT. As discussed before, using zk-Rollups would be the best choice to reduce the cost when moving our solution to a production environment.

Generate the Certificate

As explained previously, the prover precomputes the certificate, which is a zk-SNARK, required to use a specific service. The SP will verify the certificate and will be sure of the prover's right to use the service. We used `circomlib`[2] to estimate the number of constraints of the circuit used in our solution and thus, its efficiency. To create our circuit, we rely on four main functions:

- scalar_mul(): the circuit needs to multiply a number k by a point on an elliptic curve G. To do this scalar multiplication using BN128, `circomlib` uses **776 constraints**.

- hash(): the circuit needs to perform 2 fixed hashes, plus a variable number of hashes to compute a Merkle tree. A fairly secure and efficient hash function is Poseidon [16], which only uses **210 constraints** in `circomlib`.

- verify_signature(): we use the state-of-the-art signature scheme EdDSA [65] over BN128 provided in `circomlib`, which uses **4018 constraints**.

- merkle_tree(): the circuit needs to compute a Merkle tree. Assuming that $|batch\_ids| = 256 = 2^8$, our solution will need to compute 8 Poseidon hashes. This sums up to **1680 constraints**.

In total, our circuit can be implemented using **6894 constraints**. We coded a proof-of-concept using `ZPiE`[3], and executed the code using a laptop, a smartphone and a Raspberry Pi Zero. To demonstrate the scalability of our solution,

[2] https://github.com/iden3/circomlib
[3] https://github.com/xevisalle/zpie

we also executed the circuit using `snarkjs`[4], a JavaScript implementation of zk-SNARKs which can be executed in web browsers. This is perfect for scalability in web applications, with the performance drawback it involves, compared with binaries executed directly in the kernel. Table 6.1 shows the results.

Table 6.1: Performance results of FORT in different devices using different implementations. All experiments use Groth'16 and BN128.

Device	Prover	Verifier
Raspberry Pi Zero W (`ZPiE`)	79.058 s	0.134 s
Snapdragon 732G (`ZPiE`)	0.830 s	0.005 s
i7-11370H (`ZPiE`)	0.157 s	0.000733 s
i7-11370H - Firefox (`snarkjs`)	0.694 s	0.022 s

As can be seen, either in high-end devices (a laptop CPU like i7-11370H) or in mobile CPUs (snapdragon 732G), the proofs used in our protocol can be computed in a fair small amount of time using `ZPiE`. On the other hand, the time increases a lot when talking about extremely low-end CPUs like the one used in the Raspberry Pi Zero. Nevertheless, computing the proof in about a minute taking into account the single-core 700MHz CPU that it uses (approx. 10$), is a good result. Furthermore, an advantage of `FORT` is that proofs can be precomputed much before being used. Plus, even in worst-case scenarios, protocols like the one introduced in [72] would allow those devices to rely on computations on other servers owned by the same user, using a secure channel.

Regarding the verification of these proofs, as we stated previously, the verifier is succinct: all the proofs can be verified in just a few milliseconds, with no relation to the size of the circuit. As can be seen, `ZPiE` outperforms here even in the Raspberry Pi, where it takes roughly 0.1 seconds to verify proofs.

Finally, we can see how either the prover and the verifier in `snarkjs` are much slower than `ZPiE` for the same CPU. However, such a result was expected and taking into account the trade-off between performance and scalability, still it is a great result.

Prove the Attributes

The SP, after verifying the certificate, might want to be sure that some of the attributes `nft.attr` meet some additional requirements (e.g. being within a

[4]https://github.com/iden3/snarkjs

given range). For such purpose, we compute a Bulletproof from the Pedersen Commitment described in the zk-SNARK circuit. We use the module introduced in the last section to achieve this outcome. In Listing 2 we show how to deploy our solution, where the prover proves knowledge of the Pedersen Commitment, and that the secret lies within the range $[0, 2^8 - 1]$.

```
#include "../src/zpie.h"

int main()
{
    // we init the bulletproofs module, for a bulletproof of 8 bits
    bulletproof_init(8, 1);

    // we get the context (G, H, V[], gammas[])
    context ctx;
    bulletproof_get_context(&ctx);

    // we state that we will provide the random gamma and we assign
     it
    // according to the one used in the certificate
    bulletproof_user_gammas(1);
    mclBnFr_setInt(&ctx.gammas[0], 1234); // r = 1234

    // we need to create a bulletproof for this commitment:
    // out4 = attr*G + r*H
    // we set the input attr = "250"
    unsigned char *si[] = {"250"};
    bulletproof_prove(si);

    // now P -> V: Bulletproof
    // V reads out4 from the certificate, and verifies the
    Bulletproof:
    if(bulletproof_verify()) printf("Bulletproof verified.\n");
    else printf("Bulletproof cannot be verified.\n");
}
```

Listing 6.2: Implementation of our solution.

The above code for proving knowledge of an 8-bit attribute takes only **0.3 seconds** on a Raspberry Pi Zero. This time increases as the size of the attributes does the same, but being a fair amount of time to be able to use our solution in IoT devices without problems. Executing the same approach using a zk-SNARK will require around 776 constraints, and the benchmark gives us **10.5 seconds**. As such, it is clear that Bulletproofs are a much better approach for this specific use case, where provers will be able to execute the protocol instantly using low-powered devices.

Chapter 7

Citadel: Self-Sovereign Identities on Dusk Network

In this chapter, we introduce two main contributions. First, we design a private NFT model to be integrated with Dusk Network. Using such a model, a user buying an NFT will receive a token that only they will be able to read. This approach has full integration with the Dusk Network Blockchain: the changes to the original protocol are minimal and have zero impact on their performance. Plus, our contribution is secure under the same assumptions taken for the original transaction model of Dusk, called Phoenix.

Second, we introduce `Citadel`: an SSI system fully integrated with Dusk that allows users to acquire licenses (a.k.a. rights), and prove their ownership using ZKPs. By means of our novel and private NFT model, the licenses are privately stored in the Blockchain, and thus, we solve the traceability problem that other solutions had. In particular, we provide a system with the following capabilities:

- **Proof of Ownership:** a user of a service is able to prove ownership of a license that allows them to use such a service.
- **Proof of Validity:** our solution introduces the possibility to revoke licenses. Users can prove ownership of a valid license, that has not been revoked.
- **Unlinkability:** the SP cannot link any activity of their users with other activities done in the network.

- **Decentralized Nullification:** our system solves the problem regarding the possibility of reusing the proofs, where a malicious SP could impersonate the user after receiving a valid proof: by means of an on-chain and decentralized nullification, like done in the standard stack of Dusk, proofs cannot be reused.
- **Attribute Blinding:** the user is capable of deciding which information they want to leak to the SP, blinding the value and providing only the desired information.

We describe a novel protocol for authenticating in several services but preserving at the very same time our privacy. Imagine we want to buy a ticket for a concert. Nowadays, what we would do in most cases is to buy the ticket using a web page that will get information about our browser, our credit card, our bank account, ourselves... Moreover, they will charge us a fee for the ticket management service, and probably they will charge a fee to the concert promotor as well. Furthermore, as soon as we show the ticket to the concert, they will be able to link our image to the information previously gathered.

Using `Citadel`, we can do much better. In particular, we want to buy a license (a.k.a. a right) to use some service (e.g. the ticket of the example above is a license, to be used in a concert, which is a service) without leaking any information about us. Furthermore, we want to use such a service as many times as permitted by the SP, without them being able to link our activity or learn our identity. Moreover, we want a decentralized system that does not rely on third parties to manage our identities and licenses.

To achieve the aforesaid features, we first rely on Dusk Network as the decentralized framework that our solution is based. Then, we need a way to privately share assets among users of the Dusk Network. For this reason, in Section 7.3 we design a novel and private NFT model for Dusk. Finally, we need a way to anonymously prove ownership of our acquired licenses (i.e. the NFTs), so we introduce our solution in Section 7.4.

Furthermore, our solution is fully integrated into the Dusk stack, where the deployment of the solution will have minimal impact on other parts already implemented. In this same regard, Dusk has some features allowing users to delegate heavy computing tasks to trusted parties, in a secure and private manner. Our solution has been designed taking all these features into account, and thus, heavy computing tasks of our protocol can be delegated as well. This fact is important, as allows for better scalability and faster integration of our protocol into a wider set of scenarios, like web environments, IoT devices with low computing power, etc.

After describing our solution in full detail, we analyze its security, and finally provide benchmarks using our proof-of-concept implementation[1], to demonstrate its deployment feasibility.

7.1 Related Work

Interesting SSI approaches have been introduced recently, like the one detailed in [83]. This paper states a way to deploy an SSI system that grants anonymity to its users at the network level. Other solutions like [84] introduce a Blockchain-based system for preserving the privacy of users when managing their vaccine certificates.

In most cases, state-of-the-art SSI systems use ZKPs as the backbone of their architecture: cryptographic primitives allowing users to prove knowledge of some information, without leaking anything about it. This is the case of `SANS` [78], where the authors introduce a private authentication protocol based on these primitives. Using `SANS`, users can prove their rights to access different services, without the SP knowing the identity of the users.

Often, they also rely on Blockchain technologies [39], in order to achieve decentralization and immutability, when it comes to buying and granting rights to users. For instance, projects like *iden3*[2] or *Jolocom*[3] build SSI systems where owners of Decentralized Identities (DIDs) are able to manage them in a private manner. At the time of writing this, both solutions rely on the Ethereum Blockchain.

On the other hand, `FORT` is an SSI system that relies on NFTs. What they do, is represent the right acquired by someone as an NFT stored on a Blockchain, and they can prove ownership of this right by means of a ZKP. However, even when it does a great job preserving the privacy of the users of different services, this solution still has some open problems to address: the NFTs, as implemented nowadays, are publicly stored on Blockchains like Ethereum. This means that, even when users can privately prove ownership of such rights, they can still be traced on-chain. As stated in the open problems section, being able to integrate their solution into Blockchains like Dusk Network would lead to enhanced privacy. Dusk Network is a Blockchain where all the transactions are private by default, and capable of executing smart contracts with built-in privacy features.

[1] The proof-of-concept implementation can be found in the following repository: https://github.com/dusk-network/citadel

[2] https://iden3.io

[3] https://jolocom.io

Being able to integrate a private-by-design NFT model into Dusk, would lead to the possibility of designing and deploying an SSI system on top of it, which would prevent on-chain traceability.

Furthermore, FORT presents another problem: the SPs need to be trusted, as the ZKPs sent to them could be reused by them, impersonating like this the users. As such, finding a way to ensure that a license that has been already used cannot be reused in other scenarios, would be a desirable feature.

7.2 Building Blocks: The Phoenix Transaction Model

In this subsection, we introduce the details about Phoenix[4], the transaction model used by Dusk Network.

7.2.1 Overview of Phoenix

Dusk Network is an open-source public Blockchain with a UTXO-based architecture that allows the execution of obfuscated transactions and confidential smart contracts. In Phoenix, UTXOs are called *notes*, and the network keeps track of all these notes by storing their hashes in the leaves of a *Merkle tree of notes*. In other words, when a transaction is validated, the network includes the hashes of the new notes to the leaves of this tree.

All transactions include a ZKP called tx_proof that proves that the transaction has been performed following the network rules. Essentially, what this ZKP does is the following: first, it nullifies a note that the user is willing to spend. Second, proves that the user knows the value of a new note to mint, that will be sent to the receiver. Finally, proves that the amount of Dusk coins nullified is equal to the amount of coins created.

Greater details about the parameters included in the transaction have been skipped here, for the sake of completeness. Nevertheless, the next subsections explain how the protocol manages the notes nullification and minting, by introducing first the different keys the users have to handle.

7.2.2 Protocol Keys

In Phoenix, we have different kinds of keys. First, we have the static keys that belong to each user of the network, and we introduce them here as follows.

[4] https://github.com/dusk-network/phoenix-core

Let $G, G' \leftarrow \mathbb{J}$ be two JubJub points acting as our generators. We denote by H^{Poseidon} and H^{BLAKE2b} the Poseidon and BLAKE2b hash functions, respectively. Each user computes the following keys:

- **Secret key:** $\mathsf{sk} = (a, b)$, where $a, b \leftarrow \mathbb{F}_t$.
- **Public key:** $\mathsf{pk} = (A, B)$, where $A = aG$ and $B = bG$.

As noticed, Phoenix uses two-element keys, which allows users of the network to delegate the process of scanning for the notes addressed to them.

On the other hand, each note is associated with a unique one-time keypair (an approach introduced in [85]), instead of using the static public key of the receiver, which hinders traceability.

The computation of these keys is based on the Diffie–Hellman key exchange protocol [86]. The note public key of a note sent to a receiver with public key pair $\mathsf{pk} = (A, B)$, and its associated note secret key, are computed as follows:

- **Note public key:** a sender willing to send money to a receiver whose public key $\mathsf{pk} = (A, B)$ is known by them in advance, must first compute a note public key npk following next steps.

  1. Sample r uniformly at random from $\mathbb{F}_t$.
  2. Compute a symmetric Diffie–Hellman key $\mathsf{k_{DH}} = rA$.
  3. Compute a one-time public key $\mathsf{npk} = H^{\mathsf{BLAKE2b}}(\mathsf{k_{DH}})G + B$.
  4. Compute $R = rG$.

  The sender of a note will attach to it the note public key npk and the partial Diffie–Hellman R used to create npk. Given a pair (npk, R), the receiver can identify whether the note was sent to them by recomputing $\tilde{\mathsf{k}}_{\mathsf{DH}} = aR$ (using their secret a), and checking the equation

  $$\mathsf{npk} \stackrel{?}{=} H^{\mathsf{BLAKE2b}}(\tilde{\mathsf{k}}_{\mathsf{DH}})G + B.$$

- **Note secret key:** the receiver can compute the note secret key $\mathsf{nsk} = H^{\mathsf{BLAKE2b}}(\mathsf{k_{DH}}) + b$, to be used when willing to spend that note. This key can only be computed by the receiver of the note since they are the only ones holding the whole secret key $\mathsf{sk} = (a, b)$, and sk cannot be recovered from public information. This is due to the DLP in $\mathbb{J}$.

7.2.3 Protocol Details

A note is defined as the following set of elements:

$$\mathsf{N} = \{\mathsf{type}, \mathsf{com}, \mathsf{pos}, \mathsf{nonce}, \mathsf{enc}, \mathsf{npk}, R\}.$$

where type indicates the type of the note, either transparent or obfuscated; com is a commitment to the value of the note; pos is the position of the note in the Merkle tree of notes; nonce is an initialization vector needed for the encryption scheme; enc is an encryption of the opening of com that can be decrypted using the receiver's view key; npk is the note's public key, whose associated private key nsk can only be computed by the receiver of the note; and R is a point in the Jubjub subgroup $\mathbb{J}$ that allows the receiver to compute nsk and also identify that they are the receiver of the transaction.

We describe Phoenix in the general scenario in which a sender wishes to send different amounts of money $\mathsf{v}_1, \ldots, \mathsf{v}_\mathsf{n}$ to different receivers with public keys $\mathsf{pk}_1, \ldots, \mathsf{pk}_\mathsf{n}$. We assume the sender owns a set of notes $\{\mathsf{N}_1^{\mathsf{old}}, \ldots, \mathsf{N}_\mathsf{m}^{\mathsf{old}}\}$ each with an associated amount w_i such that

$$\sum_{i=1}^{m} \mathsf{w}_\mathsf{i} \geq \sum_{i=1}^{n} \mathsf{v}_\mathsf{i},$$

i.e. the sender has enough funds.

When creating the transaction to transfer the funds, the sender will have to nullify the set of old notes being spent and mint a new set of notes $\{\mathsf{N}_1^{\mathsf{new}}, \ldots, \mathsf{N}_\mathsf{n}^{\mathsf{new}}\}$ with the corresponding values v_i, and assigned to the corresponding receivers. The most common case is $\mathsf{n} = 2$, where a sender generates a note $\mathsf{N}_1^{\mathsf{new}}$ with value v for a receiver, and a second note $\mathsf{N}_2^{\mathsf{new}}$ for themselves with value $\mathsf{change} = \mathsf{v} - \sum_{i=1}^{m} \mathsf{w}_\mathsf{i}$.

To mint a new note for a receiver whose static public key is pk, we first compute the note public key (npk, R) of the receiver as described in Section 7.2.2. Next, we need to set the type of the transaction: if the transaction is transparent, we set $\mathsf{type} = 0$, and if the transaction is obfuscated, we set $\mathsf{type} = 1$. We also set v to the amount of money of the new note N. Finally, we need to commit to v, and encrypt the opening as well. To do so, we first set a blinding factor for the commitment and a nonce for the encryption:

- If $\mathsf{type} = 0$, set $s = 0$ and $\mathsf{nonce} = 0$.
- If $\mathsf{type} = 1$, set $s \leftarrow \mathbb{F}_t$ and $\mathsf{nonce} \leftarrow \mathbb{F}_t$.

and compute the value commitment $\mathsf{com} = \mathsf{Com}_{ck}(v; s)$. Then, we encrypt the opening of com:

- If $\mathsf{type} = 0$, then set $\mathsf{enc} = \mathsf{v}$.
- If $\mathsf{type} = 1$, then $\mathsf{enc} = \mathsf{Enc}_{\mathsf{k_{DH}}}(\mathsf{v}||s; \mathsf{nonce})$.

Now, we can set the new note to

$$\mathsf{N} = \{\mathsf{type}, \mathsf{com}, \mathsf{nonce}, \mathsf{enc}, \mathsf{npk}, R\}.$$

The next step is to compute a ZKP using the circuit depicted in Figure 7.1 to prove the following elements:

- **Membership**: the sender must prove that every $\mathsf{N} \in \{\mathsf{N}_\mathsf{i}^{\mathsf{old}}\}_{\mathsf{i}=1}^{\mathsf{m}}$ is included in the Merkle tree of notes. To do so, the sender provides a Merkle proof for $H^{\mathsf{Poseidon}}(\mathsf{N})$, and the circuit verifies the Merkle proof in `verify_merkle_proof()`. We observe that all these inputs are private and hence, the proof will not reveal which note is being spent, only that it belongs to the Merkle tree of notes.
- **Ownership**: the sender must prove that they hold the note secret key nsk of every note $\mathsf{N} \in \{\mathsf{N}_\mathsf{i}^{\mathsf{old}}\}_{\mathsf{i}=1}^{\mathsf{m}}$. Instead of including their private key as an input to the circuit and computing npk inside, the sender proves (using the `verify_signature()` box inside the circuit) that they can sign a message with that key. In this case, they use the double-key Schnorr signature scheme to sign the hash of the transaction.
- **Nullification**: the sender must prove that $\mathsf{nullifier} = H^{\mathsf{Poseidon}}(\mathsf{npk}'||\mathsf{pos})$. Note that the sender provides the nullification key $\mathsf{npk}' = \mathsf{nsk}G'$ as an input to the circuit and not the note secret key nsk. As we just explained, the double-key Schnorr signature guarantees that npk' is indeed $\mathsf{nsk}G'$. The result of the `hash()` box is the nullifier, which is a public output of the circuit that is later included as part of the transaction.
- **Balance integrity**: the `verify_balance()` box checks that

$$\sum_{i=1}^{m} \mathsf{w_i} - \sum_{i=1}^{n} \mathsf{v_i} - \mathsf{gas} = 0, \tag{7.1}$$

  where gas is the maximum amount of gas that the sender is willing to pay for the transaction.

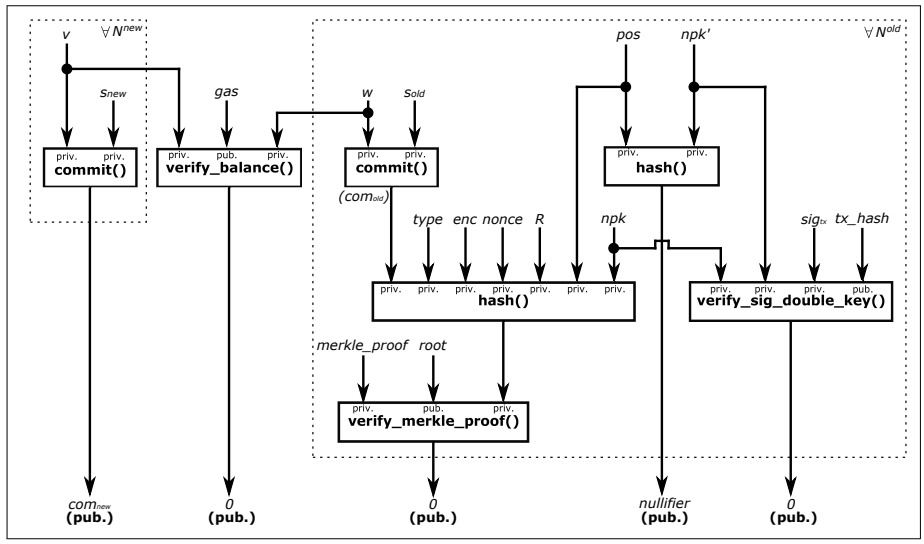


Figure 7.1: Arithmetic circuit for a Dusk transaction proof.

Observe that using the double-key Schnorr signature as proof that the user holds nsk, allows users to delegate the generation of the ZKP to a partially trusted third party, that is, a *proof helper*. This delegation would require the user to entrust the old and new values of the notes to the proof helper, but not their secret key.

Finally, the remaining checks not verified inside the circuit are performed by the network. For instance, checking that the nullifier included in the transaction matches the output of the circuit, or that the note has the right typeset.

7.3 A Private NFT Model for Dusk

As described in the previous section, coins in Dusk are represented as notes, and they can be either transparent notes (type $= 0$), or obfuscated notes (type $= 1$). Here we introduce two new types of notes: transparent NFT notes (type $= 2$) and obfuscated NFT notes (type $= 3$).

As stated previously, a user willing to spend Dusk notes needs to mint new notes while nullifying the old ones. Like this, a user can spend one note, and create a new note for the receiver, and another one with the change for them-

selves. When willing to mint a new NFT, a user will need to execute the minting contract where a Dusk note will be used to pay for the contract gas (and thus, it will be nullified), and a new note will be created to receive the change. Additionally, a new NFT note will be created. The creation of an NFT note does not need to be part of the ZKP circuit, as it is not involved in the balance to be verified. As such, it is enough to include the new NFT note in the transaction. A note representing an NFT contains the same data as other notes do, but in this case, what before was the note value in Dusk coins v, now is the $\mathsf{payload}_{\mathsf{NFT}}$ of the NFT. To mint a new note, we first compute the note public key npk and the value R of the receiver as described in Section 7.2.2, plus the symmetric key k_{DH}. Then, we set the parameters of each new NFT note N by executing a function

$$\mathsf{mint\_nft}(\mathsf{npk}, R, \mathsf{payload}_{\mathsf{NFT}}, \mathsf{k}_{\mathsf{DH}})$$

whose workflow is described in Algorithm 4.

As described previously, users willing to spend notes have to nullify them, a process that involves providing a ZKP whose circuit computes the hash of the note. In this process, the parameter type is set to private, as it is not relevant information for the protocol. **It is of paramount importance** to notice that after deploying this model, this parameter has to be public. Otherwise, an adversary could spend an NFT note pretending to be spending a regular note and would be able to create huge amounts of money out of the blue.

On the other hand, and as described in this section, the changes in the whole protocol are minimal. As such, deploying our model to the current system should be trivial.

7.4 Description of Citadel

Now, we are going the introduce all the details about `Citadel`. Then, we will detail its security analysis, and finally, we will perform some experiments in order to get benchmarks of the protocol.

7.4.1 Protocol Details

Let us have a user willing to pay a service provider SP for a license to use their service, and willing to anonymously prove ownership of this license afterward. First, the user will execute a payment in the Dusk Network addressed to the SP, including into the transaction the required information to receive the license. Upon receiving the payment, the SP will send back a license to the user, using

Algorithm 4 Minting algorithm for private NFTs.

Inputs:
(npk, R): the public note key of the receiver npk and the related value R.
$\mathsf{payload}_{\mathsf{NFT}}$: a value being the desired content of our NFT note N.
k: a symmetric encryption key.

Algorithm:

1. Set the type of note.
   - If the NFT note is transparent, set $\mathsf{type} = 2$.
   - If the NFT note is obfuscated, set $\mathsf{type} = 3$.
2. Set a nonce for the encryption.
   - If $\mathsf{type} = 2$, set $\mathsf{nonce} = 0$.
   - If $\mathsf{type} = 3$, set $\mathsf{nonce} \leftarrow \mathbb{F}_t$.
3. Encrypt the $\mathsf{payload}_{\mathsf{NFT}}$.
   - If $\mathsf{type} = 2$, then set $\mathsf{enc} = \mathsf{payload}_{\mathsf{NFT}}$.
   - If $\mathsf{type} = 3$, then $\mathsf{enc} = \mathsf{Enc}_{\mathsf{k}}(\mathsf{payload}_{\mathsf{NFT}}; \mathsf{nonce})$, where $\mathsf{Enc}()$ is a symmetric encryption function.
4. Set the new NFT note to

$$\mathsf{N} = \{\mathsf{type}, \mathsf{enc}, \mathsf{nonce}, R, \mathsf{npk}\}.$$

the same Blockchain. In order to use the license, the user will have to call a smart contract deployed in the Dusk Network, called the *license contract*. Essentially, the user will provide a proof that demonstrates that they own a valid license, the license contract will verify the proof, and will append a license nullifier to a Merkle tree of nullifiers. By means of a session cookie included in the same contract call, and addressed to the SP, the user will be able to request the service using an off-chain and secure channel. The protocol is described in full detail as follows.

CITADEL PROTOCOL The protocol has three main parties: a Service Provider SP offering a service, and publicly sharing its public key $\mathsf{pk_{SP}}$, a user willing to use a service provided by the SP, and the Dusk Network Blockchain. The protocol consists of eight main steps performed between the involved parties. These steps are depicted in Figure 7.2, and described as follows.

1. (**user**) send_note_license_req : Compute a note public key $(\mathsf{npk_{user}}, R_{\mathsf{user}})$ belonging to the user, using the user's own public key, and also an additional key $\mathsf{k_{user}} = H^{\mathsf{Poseidon}}(\mathsf{npk_{user}}, \mathsf{nsk_{user}})$, by computing first the user's $\mathsf{nsk_{user}}$. Then, send the required amount of Dusk coins to the SP, in order to pay for the service. Into the same transaction, send an NFT to the SP using the function $\mathsf{mint\_nft}(\mathsf{npk_{SP}}, R_{\mathsf{SP}}, \mathsf{payload_{NFT}}, \mathsf{k_{DH}})$, whose arguments are computed as follows:

   - $(\mathsf{npk_{SP}}, R_{\mathsf{SP}})$ is the SP's note public key, computed through his public key $\mathsf{pk_{SP}}$.
   - $\mathsf{payload_{NFT}} = (\mathsf{npk_{user}}, R_{\mathsf{user}}, \mathsf{k_{user}})$.
   - $\mathsf{k_{DH}}$ is computed using the SP's public key.

2. (**SP**) get_note_license_req : Continuously check the network for incoming license requests. Upon receiving the payment from a user, define a set of attributes *attr* representing the license, and compute a digital signature as follows:

$$\mathsf{sig_{lic}} = \mathsf{sign\_single\_key}_{\mathsf{sk_{SP}}}(\mathsf{npk_{user}}, \mathsf{attr})$$

3. (**SP**) send_note_license : Set the $\mathsf{payload_{NFT}} = \{\mathsf{sig_{lic}}, \mathsf{attr}\}$, and send the license to the user using the function $\mathsf{mint\_nft}(\mathsf{npk_{user}}, R_{\mathsf{user}}, \mathsf{payload_{NFT}}, \mathsf{k_{user}})$.

4. (**user**) get_note_license : Receive the note containing the license.

5. (**user**) call_nullify_license : When desiring to use the license, nullify it by executing a call to the license contract. The following steps are performed:
   - The user sets a session cookie $\mathsf{sc} = (\mathsf{s_0}, \mathsf{s_1}, \mathsf{s_2}) \leftarrow \mathbb{F}_t$.
   - The user creates a new NFT note where $\mathsf{payload_{NFT}} = \mathsf{sc}$, and the SP is the receiver.
   - The user issues the transaction that includes the NFT described in the previous step, by calling the license contract. In this case, the tx_proof is computed as done in the standard Phoenix model, but into the same circuit, the circuit depicted in Figure 7.3 is appended.
   - The network validators will execute the smart contract, which verifies the proof. Upon success, the NFT note will be forwarded, and the license nullifier $\mathsf{nullifier_{lic}}$ will be added to the Merkle tree of nullifiers.
6. (**SP**) get_note_session_cookie : Receive a note containing the session cookie sc.
7. (**user**) req_service : Request the service to the SP, establishing communication using a secure channel, and providing the tuple $(\mathsf{tx\_hash}, \mathsf{attr}, \mathsf{pk_{SP}}, c, \mathsf{sc})$.
8. (**SP**) grant_service : Grant or deny the service upon verification of the following steps:
   - Check whether or not the values $(\mathsf{attr}, \mathsf{pk_{SP}}, c)$ are correct.
   - Check whether or not the openings $((\mathsf{pk_{SP}}, \mathsf{s_0}), (\mathsf{attr}, \mathsf{s_1}), (c, \mathsf{s_2}))$ match the commitments $\mathsf{com}_0^{hash}, \mathsf{com}_1, \mathsf{com}_2$ found in the transaction tx_hash.

As can be seen, the fact that only the user knows the nsk required to compute $\mathsf{sig_{lic}}$ allows them to prove ownership of the license, by means of the double key Schnorr signature, and this license is verified by proving knowledge of a valid signature verified with the public key of the SP.

Moreover, we can appreciate in the circuit how the license is linked to the $\mathsf{npk_{user}}$, and the user also verifies a Merkle proof that proves membership of this note in the Dusk Network. This brings the revocation feature: if under some circumstances the SP no longer accepts some previously issued licenses, they can prove to the network that a given note contains a license issued by them, and under a consensus agreement, it will be removed. As such, after removal, the user will not be able to provide valid proofs for this license anymore. Plus, it can happen that a user receives a license, but the transaction is finally not

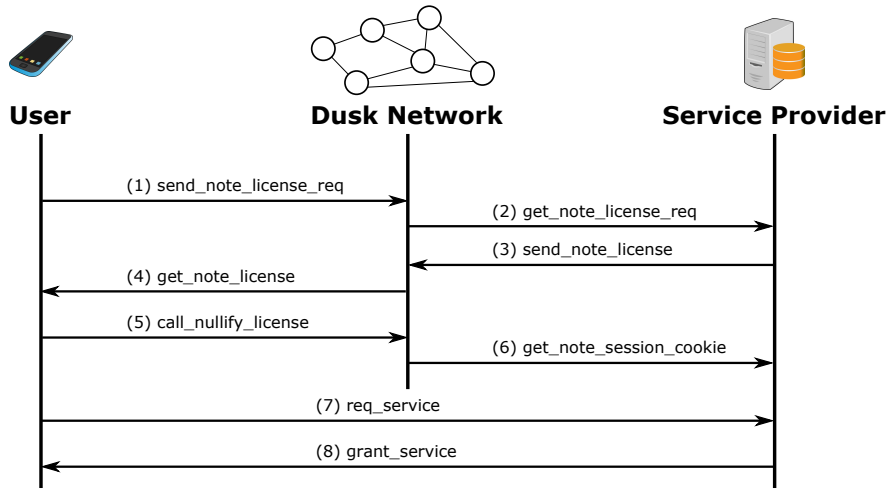


Figure 7.2: Overview of the protocol messages exchanged between the user, the Dusk Network, and the SP.

accepted in the Blockchain (e.g. the transaction proof was not correct, some regulation checks have failed, etc.). In this scenario, the received license will not be valid, and the user indeed will not be able to provide a correct Merkle proof.

Furthermore, the SP might request the user to nullify the license they are using (i.e. this is a single-use license, like entering a concert). This is done through the computation of $\mathsf{nullifier}_{\mathsf{lic}}$. The deployment of this part of the circuit has two different possibilities:

- If we set $c = 0$ (or directly remove this input from the circuit), the license will be able to be used only once.
- If the SP requests the user to set a custom value for c (e.g. the date of an event), the license will be able to be reused only under certain conditions.

It is also interesting to notice that the whole protocol has been designed with perfect integration into Dusk Network. As can be noticed, all the information needed to prove ownership of a license is stored on-chain. As such, a user setting up a new instance of the Dusk wallet will be able to retrieve all the licenses by simply knowing his static secret key, as would be done with the whole Dusk protocol. The same happens with the delegation of the received notes check,

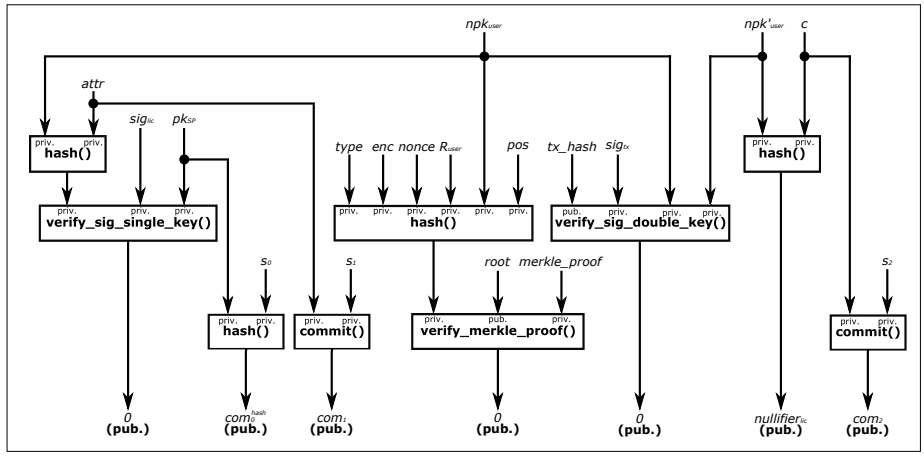


Figure 7.3: Arithmetic circuit for proving a license's ownership.

where the user delegates the process of checking which notes are addressed to them. In `Citadel`, the user can securely delegate the check of received licenses. Furthermore, proof delegation is also possible, as the user knowing `nsk` will use this key to sign a specific transaction, and this cannot be modified.

7.4.2 Security Analysis

We start the security analysis of `Citadel` by elaborating on the ZKP scheme to use. The need for a trusted setup is one of the main drawbacks of some ZKP constructions like PlonK, especially when used in scenarios like cryptocurrencies. An untrusty setup where an adversary gets the seed used to compute it would allow them to create false transactions, and this would lead to huge losses of money. In `Citadel`, an untrusted setup would lead to user impersonation, and being able to use others' licenses.

On the other hand, the soundness property of PlonK relies on the AGM, a weaker assumption than the q-PKE. Regarding the elliptic curve used in our scenario, the BLS12-381 [21] is estimated to have around 128-bits of security, which complies with the security standards.

We now put the spotlight on the security of the circuit we have designed, which grants the following features:

- **Proof of Ownership:** the circuit used in `Citadel` verifies a signature

$\mathsf{sig}_{\mathsf{lic}}$ of an input message $(\mathsf{npk}_{\mathsf{user}}, \mathsf{attr})$, using the public key of SP, $\mathsf{pk}_{\mathsf{SP}}$. Also, a double key signature $\mathsf{sig}_{\mathsf{stx}}$ of a transaction hash $\mathsf{tx\_hash}$ is verified in-circuit, referring to the transaction where the ZKP will be appended.

$\mathsf{sig}_{\mathsf{lic}}$ verification ensures that the license attributes are correct, and $\mathsf{sig}_{\mathsf{stx}}$ ensures that the user owns such a license, as only they can compute $\mathsf{npk}_{\mathsf{user}}$ using the note secret key and compute such a signature, while keeping all these values private, so SP cannot learn the identity of the user. An adversary would not be able to prove ownership as long as $\mathsf{nsk}_{\mathsf{user}}$ is not leaked to them. This is true under the DLP.

- **Proof of Validity:** the fact that $\mathsf{npk}_{\mathsf{user}}$ is part of the signature $\mathsf{sig}_{\mathsf{lic}}$, ensures that the license is assigned to a specific note of the Dusk Network, and thus, a specific user of this Blockchain. The circuit verifies a Merkle proof of the NFT note containing the license, which is included in the Merkle tree of notes. This ensures that the license the user is proving ownership of has been transacted in the Dusk Network, and is a valid license at the moment of issuing the transaction.

  An adversary willing to successfully prove ownership of a transferred license would have to craft a new pair ($\mathsf{npk}_{\mathsf{user}}$,attr) that verifies $\mathsf{sig}_{\mathsf{lic}}$. This is infeasible under the DLP. Furthermore, the crafted $\mathsf{npk}_{\mathsf{user}}$ would have to be a collision verifying the Merkle proof.

- **Unlinkability:** the user sends the one-time key pair $(\mathsf{npk}_{\mathsf{user}}, R_{\mathsf{user}})$ to the SP, instead of the public key pk. The fact that the information about the user learned by the SP is a set of one-time values ensures that the identity of the user sending these values cannot be linked to other activities done in the network. The key $\mathsf{npk}_{\mathsf{user}}$ is computed from the value $\mathsf{nsk}_{\mathsf{user}}$, which is kept secret and used only one time. As there are no other values involved in the process that identifies the user, they cannot be linked to the user's identity. This is true as long as the user does not reuse $\mathsf{nsk}_{\mathsf{user}}$. On the other hand, $\mathsf{npk}_{\mathsf{user}} = H^{\mathsf{BLAKE2b}}(rA)G + B$, where r is sampled at random and (A, B) is the user's public key. As both $H^{\mathsf{BLAKE2b}}(rA)G$ and B are only known by the user, there is no way an adversary can learn B, because $\mathsf{npk}_{\mathsf{user}}$ can be decomposed in many ways.

  From the point of view of the network, there is unlinkability as well: when issuing the transaction, no one is able to link the nullified license to the SP, as the $\mathsf{pk}_{\mathsf{SP}}$ is blinded by committing to this value using the $\mathsf{hash}()$ function and a random value s_0. An adversary would not be able to learn $\mathsf{pk}_{\mathsf{SP}}$ as long as the randomness involved in the hashing process is not leaked to

them. This is true assuming that the hashing function is collision-resistant. On the other hand, both attr and c could leak information about the service and the user. For this reason, we commit to these values (as they are scalars instead of points, we can use the Pedersen Commitment which requires fewer constraints than the hash function). An adversary would not be able to learn (attr, c) as long as the random values involved in the commitments are not leaked to them. This is true under the DLP, which holds for the Pedersen Commitment.

- **Decentralized Nullification:** the circuit computes the hash of $\mathsf{npk}_{\mathsf{user}}$ and a public challenge c, resulting in $\mathsf{nullifier}_{\mathsf{lic}}$. The format of the c value could change in different scenarios. Taking the example of proving ownership of a ticket for an event, ideally, c would be the date of such an event. If a function checking if a given nullifier has been previously seen, results in the equation $\mathsf{is\_seen}(\mathsf{nullifier}_{\mathsf{lic}}, \mathsf{previous\_nullifiers}[]) \stackrel{?}{=} 1$ holding, it means that someone already entered the event with the same license. As such, we ensure that a user cannot use the same license multiple times, nor compute valid proofs for other users. $\mathsf{npk}_{\mathsf{user}}$ is fixed in advance, as such, $\mathsf{nullifier}_{\mathsf{lic}}$ will always be the same for a given public input c, which needs to be validated by the SP.

- **Attribute Blinding:** As described previously, the user provides an opening for the commitment com_1 to the SP, thus leaking the attr value. An adversary would not be able to provide a valid opening as long as the randomness involved in the commitment of attr is not leaked to them. This is true under the DLP, which holds for the Pedersen Commitment.

  Depending on the use case, it could be desirable that the values involved in attr are kept totally or partially private. In this scenario, and as suggested in the `FORT` protocol, the user could instead provide an additional proof of knowledge, proving to the SP that they know the opening of com_1. As an example, a Bulletproof is a kind of ZKP allowing to prove knowledge of a value that lies within a certain range.

7.4.3 Benchmarks

We use `dusk-plonk` to code the circuit used in our solution. Like this, we get the number of constraints of its different elements and thus, the efficiency of computing (and verifying) proofs. Our circuit needs four main functions:

- hash(): we use the Poseidon hash function, which uses **977 contraints** when hashing 1 input. The amount of constraints increases depending on the number of inputs.
- commit(): the Pedersen Commitment requires **527 constraints**.
- verify_sig_single_key(): Dusk uses the Schnorr proof signature scheme over BLS12-381, which uses **3388 constraints** when using the single key version.
- verify_sig_double_key(): Dusk uses the Schnorr proof signature scheme over BLS12-381, which uses **6645 constraints** when using the double key version.
- verify_merkle_proof(): the circuit needs to verify a Merkle proof. Dusk uses Merkle trees of depth 17, as such, our solution will need to compute 17 Poseidon hashes. This sums up to **17807 constraints**.

We implemented our circuit using a total amount of **34861 constraints**. Here, we need to add the constraints needed to compute the default tx_proof, which are **31486 constraints** for nullifying one note. We benchmarked both the prover and verifier using an Apple Silicon M1 CPU. The prover takes **16.232 seconds** to compute the proof, and the verifier **0.007 seconds** to verify it.

Plus, it has to be taken into account that an advantage of `Citadel` is that licenses can be nullified much before being used. This means that long ZKP proving times will not have a big impact on the performance of the protocol even when using devices with low computing resources. Nonetheless, and as mentioned previously, computations can be delegated as done in the standard Phoenix model, so high-performing CPUs will compute the proofs even in less time.

Bibliography

[1] Christopher Allen. The path to self-sovereign identity. Accessed 2020-07-17. http://www.lifewithalacrity.com/2016/04/the-path-to-self-soverereign-identity.html. 1, 51, 63

[2] S Goldwasser, S Micali, and C Rackoff. The knowledge complexity of interactive proof-systems. In *Proceedings of the Seventeenth Annual ACM Symposium on Theory of Computing*, STOC '85, pages 291–304, New York, NY, USA, 1985. ACM. 1, 15

[3] Alexander Mühle, Andreas Grüner, Tatiana Gayvoronskaya, and Christoph Meinel. A survey on essential components of a self-sovereign identity. *Computer Science Review*, 30:80–86, Nov 2018. 2, 52, 63

[4] Sovrin Foundation. Sovrin: A Protocol and Token for Self-Sovereign Identity and Decentralized Trust. https://sovrin.org/wp-content/uploads/Sovrin-Protocol-and-Token-White-Paper.pdf, January 2018. 2, 49, 51, 63

[5] ETSI (3GPP). Procedures for the 5G System (5GS), v15.5.1, release 15, May 2019. 3, 50

[6] Francisco Ramos, Sergio Trilles, Andrés Muñoz, and Joaquín Huerta. Promoting pollution-free routes in smart cities using air quality sensor networks. *Sensors*, 18(8), 2018. 3

[7] Syed Misbahuddin, Junaid Ahmed Zubairi, Abdulrahman Saggaf, Jihad Basuni, Sulaiman A-Wadany, and Ahmed Al-Sofi. Iot based dynamic road traffic management for smart cities. In *2015 12th International Conference on High-capacity Optical Networks and Enabling/Emerging Technologies (HONET)*, pages 1–5, 2015. 3

[8] Fadi Al-Turjman and Joel Poncha Lemayian. Intelligence, security, and vehicular sensor networks in internet of things (iot)-enabled smart-cities: An overview. *Computers & Electrical Engineering*, 87:106776, 2020. 3

[9] Sakshi Painuly, Priya Kohli, Priya Matta, and Sachin Sharma. Advance applications and future challenges of 5g iot. In *2020 3rd International Conference on Intelligent Sustainable Systems (ICISS)*, pages 1381–1384, 2020. 3

[10] Javier Silva Velón. Zero-knowledge proofs and isogeny-based cryptosystems, 2020. 9

[11] Sean Bowe and Jack Grigg. Implementation of the BLS12-381 pairing-friendly elliptic curve construction. Available online: https://github.com/zkcrypto/bls12_381 (accessed on 1 June 2022). 11

[12] Sean Bowe, Eirik Ogilvie-Wigley, , and Jack Grigg. Implementation of the Jubjub elliptic curve group. Available online: https://github.com/zkcrypto/jubjub (accessed on 1 June 2022). 11

[13] Paulo S. L. M. Barreto and Michael Naehrig. Pairing-friendly elliptic curves of prime order. Cryptology ePrint Archive, Report 2005/133, 2005. https://eprint.iacr.org/2005/133. 11, 55

[14] Jordi Baylina and Marta Bellés. Eddsa for baby jubjub elliptic curve with mimc-7 hash. 11, 15

[15] J. P. Aumasson, Samuel Neves, Zooko Wilcox-O'Hearn, and Christian Winnerlein. BLAKE2: simpler, smaller, fast as MD5. https://blake2.net/, January 2013. 13, 31

[16] Lorenzo Grassi, Dmitry Khovratovich, Christian Rechberger, Arnab Roy, and Markus Schofnegger. Starkad and poseidon: New hash functions for zero knowledge proof systems. Cryptology ePrint Archive, Report 2019/458, 2019. https://eprint.iacr.org/2019/458. 13, 53, 72

[17] Ralph C Merkle. A digital signature based on a conventional encryption function. In *Conference on the theory and application of cryptographic techniques*, pages 369–378. Springer, 1987. 13

[18] Jonathan Katz and Yehuda Lindell. *Introduction to modern cryptography*. CRC press, 2020. 15

[19] Manuel Blum, Paul Feldman, and Silvio Micali. Non-interactive zero-knowledge and its applications. In *Proceedings of the Twentieth Annual ACM Symposium on Theory of Computing*, STOC '88, pages 103–112, New York, NY, USA, 1988. ACM. 16

[20] Eli Ben-Sasson, Alessandro Chiesa, Eran Tromer, and Madars Virza. Succinct non-interactive zero knowledge for a von neumann architecture. Cryptology ePrint Archive, Report 2013/879, 2013. https://eprint.iacr.org/2013/879. 16

[21] Daira Hopwood, Sean Bowe, Taylor Hornby, and Nathan Wilcox. Zcash Protocol Specification - Version 2019.0.2, 2019. https://github.com/zcash/zips/blob/master/protocol/protocol.pdf, Accessed on 28/09/2021. 16, 17, 18, 40, 55, 62, 68, 88

[22] Eli Ben-Sasson, Alessandro Chiesa, Eran Tromer, and Madars Virza. Succinct non-interactive zero knowledge for a von neumann architecture. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 781–796, San Diego, CA, August 2014. USENIX Association. 16, 17, 18, 40

[23] Jens Groth. On the size of pairing-based non-interactive arguments. In Marc Fischlin and Jean-Sébastien Coron, editors, *Advances in Cryptology – EUROCRYPT 2016*, pages 305–326, Berlin, Heidelberg, 2016. Springer Berlin Heidelberg. 16, 17, 18, 40, 62

[24] Mary Maller, Sean Bowe, Markulf Kohlweiss, and Sarah Meiklejohn. Sonic: Zero-knowledge snarks from linear-size universal and updatable structured reference strings. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, CCS '19, page 2111–2128, New York, NY, USA, 2019. Association for Computing Machinery. 16, 17

[25] Tiacheng Xie, Jiaheng Zhang, Yupeng Zhang, Charalampos Papamanthou, and Dawn Song. Libra: Succinct zero-knowledge proofs with optimal prover computation. In Alexandra Boldyreva and Daniele Micciancio, editors, *Advances in Cryptology – CRYPTO 2019*, pages 733–764, Cham, 2019. Springer International Publishing. 16, 17

[26] Ariel Gabizon, Zachary J. Williamson, and Oana Ciobotaru. Plonk: Permutations over lagrange-bases for oecumenical noninteractive arguments of knowledge. Cryptology ePrint Archive, Report 2019/953, 2019. https://ia.cr/2019/953, Accessed on 28/09/2021. 16

[27] Jonathan Lee, Srinath Setty, Justin Thaler, and Riad Wahby. Linear-time and post-quantum zero-knowledge snarks for r1cs. Cryptology ePrint Archive, Report 2021/030, 2021. https://ia.cr/2021/030, Accessed on 28/09/2021. 16

[28] Benedikt Bünz, Jonathan Bootle, Dan Boneh, Andrew Poelstra, Pieter Wuille, and Greg Maxwell. Bulletproofs: Short proofs for confidential transactions and more. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 315–334, 2018. 16, 17, 20, 63

[29] Eli Ben-Sasson, Iddo Bentov, Yinon Horesh, and Michael Riabzev. Scalable, transparent, and post-quantum secure computational integrity. Cryptology ePrint Archive, Report 2018/046, 2018. https://eprint.iacr.org/2018/046. 17

[30] Eduardo Morais, Tommy Koens, Cees Van Wijk, and Aleksei Koren. A survey on zero knowledge range proofs and applications. *SN Applied Sciences*, 1(8):1–17, 2019. 16

[31] Shafi Goldwasser and Yael Tauman Kalai. Cryptographic assumptions: A position paper. In Eyal Kushilevitz and Tal Malkin, editors, *Theory of Cryptography*, pages 505–522, Berlin, Heidelberg, 2016. Springer Berlin Heidelberg. 17

[32] Sean Bowe, Ariel Gabizon, and Ian Miers. Scalable Multi-party Computation for zk-SNARK Parameters in the Random Beacon Model. Cryptology ePrint Archive, Report 2017/1050, 2017. https://eprint.iacr.org/2017/1050, Accessed on 28/09/2021. 18

[33] Paulo S. L. M. Barreto and Michael Naehrig. Pairing-friendly elliptic curves of prime order. In Bart Preneel and Stafford Tavares, editors, *Selected Areas in Cryptography*, pages 319–331, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg. 18

[34] Alfred Menezes, Palash Sarkar, and Shashank Singh. Challenges with assessing the impact of nfs advances on the security of pairing-based cryptography. Cryptology ePrint Archive, Report 2016/1102, 2016. https://eprint.iacr.org/2016/1102. 18, 55, 68

[35] Youssef El Housni and Aurore Guillevic. Optimized and secure pairing-friendly elliptic curves suitable for one layer proof composition. In Stephan

Krenn, Haya Shulman, and Serge Vaudenay, editors, *Cryptology and Network Security*, pages 259–279, Cham, 2020. Springer International Publishing. 18

[36] Jean-Luc Beuchat, Jorge E. González-Díaz, Shigeo Mitsunari, Eiji Okamoto, Francisco Rodríguez-Henríquez, and Tadanori Teruya. High-speed software implementation of the optimal ate pairing over barreto–naehrig curves. In Marc Joye, Atsuko Miyaji, and Akira Otsuka, editors, *Pairing-Based Cryptography - Pairing 2010*, pages 21–39, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg. 20

[37] Benedikt Bünz, Shashank Agrawal, Mahdi Zamani, and Dan Boneh. Zether: Towards privacy in a smart contract world. Cryptology ePrint Archive, Report 2019/191, 2019. https://eprint.iacr.org/2019/191, Accessed on 28/09/2021. 20

[38] Stephan Leible, Steffen Schlager, Moritz Schubotz, and Bela Gipp. A review on blockchain technology and blockchain projects fostering open science. *Frontiers in Blockchain*, 2:16, 2019. 22

[39] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system, 2008. https://bitcoin.org/bitcoin.pdf, Accessed on 06/02/2022. 22, 61, 77

[40] Arthur Gervais, Ghassan O Karame, Karl Wüst, Vasileios Glykantzis, Hubert Ritzdorf, and Srdjan Capkun. On the security and performance of proof of work blockchains. In *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*, pages 3–16, 2016. 22

[41] Iddo Bentov, Charles Lee, Alex Mizrahi, and Meni Rosenfeld. Proof of activity: Extending bitcoin's proof of work via proof of stake [extended abstract] y. *ACM SIGMETRICS Performance Evaluation Review*, 42(3):34–37, 2014. 22

[42] Gavin Wood. Ethereum: A secure decentralised generalised transaction ledger (istanbul version), June 2021. 22, 23, 62

[43] Anastasia Mavridou and Aron Laszka. Designing secure ethereum smart contracts: A finite state machine based approach. In *International Conference on Financial Cryptography and Data Security*, pages 523–540. Springer, 2018. 22, 23

[44] Everett Hildenbrandt, Manasvi Saxena, Nishant Rodrigues, Xiaoran Zhu, Philip Daian, Dwight Guth, Brandon Moore, Daejun Park, Yi Zhang, Andrei Stefanescu, et al. Kevm: A complete formal semantics of the ethereum virtual machine. In *2018 IEEE 31st Computer Security Foundations Symposium (CSF)*, pages 204–217. IEEE, 2018. 22, 23

[45] Toghrul Maharramov, Dmitry Khovratovich, and Emanuele Francioni. The dusk network whitepaper, 2021. https://dusk.network/uploads/The_Dusk_Network_Whitepaper_v3_0_0.pdf, Accessed on 04/02/2022. 22, 23

[46] Ting Chen, Xiaoqi Li, Y. Wang, Jiachi Chen, Zihao Li, Xiapu Luo, Man Ho Allen Au, and Xiaosong Zhang. An adaptive gas cost mechanism for ethereum to defend against under-priced dos attacks. *ArXiv*, abs/1712.06438, 2017. 22

[47] Ethworks Reports. Zero-knowledge blockchain scalability, 2020. 23

[48] X. Lv and L. Xu. AES encryption algorithm keyless entry system. In *2012 2nd International Conference on Consumer Electronics, Communications and Networks (CECNet)*, pages 3090–3093, April 2012. 25, 30

[49] Xiao Ni, Weiren Shi, and Victor Foo Siang Fook. AES security protocol implementation for automobile remote keyless system. *IEEE 65th Vehicular Technology Conference - VTC2007-Spring*, April 2007. 25

[50] T. Glocker, T. Mantere, and M. Elmusrati. A protocol for a secure remote keyless entry system applicable in vehicles using symmetric-key cryptography. In *2017 8th International Conference on Information and Communication Systems (ICICS)*, pages 310–315, April 2017. 26, 30

[51] Microchip Technology Inc. *TB003 An Introduction to KEELOQ Code Hopping*, 1996. 28

[52] Joseph David King. *Passive remote keyless entry system*, 06 1998. US6236333B1. 28

[53] NXP Semiconductors N.V. NXP keyless entry/go solutions: Advancing keyless entry/go, April 2012. 28

[54] S. Kamkar. Drive it like you hacked it: New attacks and tools to wirelessly steal cars. DEFCON 23, 2015. 28

[55] Aurélien Francillon, Boris Danev, and Srdjan Capkun. Relay attacks on passive keyless entry and start systems in modern cars, 2010. https://eprint.iacr.org/2010/332. 29

[56] Neha Thakur and Aruna Sankaralingam. Introduction to jamming attacks and prevention techniques using honeypots in wireless networks. IRACST - International Journal of Computer Science and Information Technology and Security (IJCSITS), April 2013. 29

[57] Omar Adel Ibrahim, Ahmed Mohamed Hussain, Gabriele Oligeri, and R. Di Pietro. Key is in the air: Hacking remote keyless entry systems. In *Proc. of the International Workshop on Cyber Security for Intelligent Transportation Systems (CSITS2018)*, July 2018. 29

[58] Philips Semiconductors. HT2 DC20 S20 HITAG2 Transponders Revision 2.0, March 1998. 30

[59] Flavio D. Garcia and David Oswald. Lock It and Still Lose It-On the (In)Security of Automotive Remote Keyless Entry Systems. *25th USENIX Security Symposium (USENIX Security 16)*, 2016. 30

[60] H. Jeong and J. So. Channel correlation-based relay attack avoidance in vehicle keyless-entry systems. *Electronics Letters*, 54(6):395–397, 2018. 30

[61] A. Ranganathan and S. Capkun. Are we really close? verifying proximity in wireless systems. *IEEE Security Privacy*, pages 1–1, 2018. 30

[62] Zubair Amjad, Axel Sikora, Jean-Philippe Lauffenburger, and Benoit Hilt. Latency reduction in narrowband 4g lte networks. In *2018 15th International Symposium on Wireless Communication Systems (ISWCS)*, 2018. 38

[63] Howard Wu, Wenting Zheng, Alessandro Chiesa, Raluca Ada Popa, and Ion Stoica. DIZK: A distributed zero knowledge proof system. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 675–692, Baltimore, MD, August 2018. USENIX Association. 39

[64] Martin Albrecht, Lorenzo Grassi, Christian Rechberger, Arnab Roy, and Tyge Tiessen. Mimc: Efficient encryption and cryptographic hashing with minimal multiplicative complexity. In Jung Hee Cheon and Tsuyoshi Takagi, editors, *Advances in Cryptology – ASIACRYPT 2016*, pages 191–219, Berlin, Heidelberg, 2016. Springer Berlin Heidelberg. 42

[65] Daniel J. Bernstein, Niels Duif, Tanja Lange, Peter Schwabe, and Bo-Yin Yang. High-speed high-security signatures. Journal of Cryptographic Engineering 2, 2012. 42, 53, 72

[66] A. Othman and J. Callahan. The horcrux protocol: A method for decentralized biometric-based self-sovereign identity. In *2018 International Joint Conference on Neural Networks (IJCNN)*, pages 1–7, 2018. 52

[67] L. Fang, Y. Li, X. Yun, Z. Wen, S. Ji, W. Meng, Z. Cao, and M. Tanveer. Thp: A novel authentication scheme to prevent multiple attacks in sdn-based iot network. *IEEE Internet of Things Journal*, 7(7):5745–5759, 2020. 52

[68] Liming Fang, Changchun Yin, Lu Zhou, Yang Li, Chunhua Su, and Jinyue Xia. A physiological and behavioral feature authentication scheme for medical cloud based on fuzzy-rough core vector machine. *Information Sciences*, 507:143–160, 2020. 52

[69] Geovane Fedrecheski, Jan M. Rabaey, Laisa Caroline de Paula Costa, Pablo C. Calcina-Ccori, William Takeshi Pereira, and Marcelo Knörick Zuffo. Self-sovereign identity for iot environments: A perspective. *2020 Global Internet of Things Summit (GIoTS)*, pages 1–6, 2020. 52

[70] Vanesa Daza and Xavier Salleras. LASER: Lightweight And SEcure Remote keyless entry protocol (Extended version). *arXiv preprint arXiv:1905.05694*, 2019. https://arxiv.org/pdf/1905.05694.pdf. 53

[71] V. Migliore, M. M. Real, V. Lapotre, A. Tisserand, C. Fontaine, and G. Gogniat. Exploration of polynomial multiplication algorithms for homomorphic encryption schemes. In *2015 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*, pages 1–6, 2015. 56

[72] Howard Wu, Wenting Zheng, Alessandro Chiesa, Raluca Ada Popa, and Ion Stoica. Dizk: A distributed zero knowledge proof system. Cryptology ePrint Archive, Report 2018/691, 2018. https://eprint.iacr.org/2018/691. 57, 73

[73] Sidra Ijaz, Munam Ali Shah, Abid Khan, and Mansoor Ahmed. Smart cities: A survey on security concerns. *International Journal of Advanced Computer Science and Applications*, 7(2):612–625, 2016. 61

[74] Liesbet Van Zoonen. Privacy concerns in smart cities. *Government Information Quarterly*, 33(3):472–480, 2016. 61

[75] P. Shamili, B. Muruganantham, and B. Sriman. Understanding concepts of blockchain technology for building the dapps. In Subhransu Sekhar Dash, Swagatam Das, and Bijaya Ketan Panigrahi, editors, *Intelligent Computing and Applications*, pages 383–394, Singapore, 2021. Springer Singapore. 62

[76] Xavier Salleras and Vanesa Daza. Zpie: Zero-knowledge proofs in embedded systems. *Mathematics*, 9(20), 2021. 62

[77] Jacob Evans William Entriken, Dieter Shirley and Nastassia Sachs. Eip-721: Erc-721 non-fungible token standard, January 2018. 62

[78] Xavier Salleras and Vanesa Daza. Sans: Self-sovereign authentication for network slices. *Security and Communication Networks*, 2020, 2020. 63, 77

[79] Georgios Fragkos, Cyrus Minwalla, Jim Plusquellic, and Eirini Eleni Tsiropoulou. Artificially intelligent electronic money. *IEEE Consumer Electronics Magazine*, 10(4):81–89, 2021. 64

[80] Yuri Bespalov, Alberto Garoffolo, Lyudmila Kovalchuk, Hanna Nelasa, and Roman Oliynykov. Probability models of distributed proof generation for zk-snark-based blockchains. *Mathematics*, 9(23), 2021. 64

[81] Javier Arcenegui, Rosario Arjona, and Iluminada Baturone. Secure management of iot devices based on blockchain non-fungible tokens and physical unclonable functions. In *Applied Cryptography and Network Security Workshops*, pages 24–40. Springer International Publishing, 2020. 64

[82] Youssef El Housni and Aurore Guillevic. Optimized and secure pairing-friendly elliptic curves suitable for one layer proof composition. Cryptology ePrint Archive, Report 2020/351, 2020. https://eprint.iacr.org/2020/351. 68

[83] Quinten Stokkink, Dick H. J. Epema, and Johan Pouwelse. A truly self-sovereign identity system. *CoRR*, abs/2007.00415, 2020. 77

[84] Amal Abid, Saoussen Cheikhrouhou, Slim Kallel, and Mohamed Jmaiel. Novidchain: Blockchain-based privacy-preserving platform for covid-19 test/vaccine certificates. *Software: Practice and Experience*, 52(4):841–867, 2022. 77

[85] Nicolas Van Saberhagen. Cryptonote v 2.0. 2013. 79

[86] Whitfield Diffie and Martin Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, 22(6):644–654, 1976. 79

Appendix A

Groth'16

Let $\mathcal{R}$ be a relation composed of the elliptic curve E over $\mathbb{F}_q$, the pairing e and a QAP $(A, B, C, Z(x))$ representing a circuit. The Groth'16 construction is divided into three algorithms:

- $\sigma \leftarrow \mathit{Setup}(\mathcal{R})$: To perform the setup, we pick $\alpha, \beta, \gamma, \delta, x$ from $\mathbb{F}_r$ and define $\tau = (\alpha, \beta, \gamma, \delta, x)$. Later we compute $\sigma = ([\sigma_1]_1, [\sigma_2]_2)$:

$$\begin{aligned} vk' &= \{\frac{\beta A_i(x) + \alpha B_i(x) + C_i(x)}{\gamma}\}_{i=0}^{l} \\ pk' &= \{\frac{\beta A_i(x) + \alpha B_i(x) + C_i(x)}{\delta}\}_{i=l+1}^{m} \\ \sigma_1 &= (\alpha, \beta, \delta, \{x^i\}_{i=0}^{n-1}, vk', pk', \{\frac{x^i t(x)}{\delta}\}_{i=0}^{n-2}) \end{aligned} \tag{A.1}$$

$$\sigma_2 = (\beta, \gamma, \delta, \{x^i\}_{i=0}^{n-1}) \tag{A.2}$$

- $\pi \leftarrow \mathit{Prove}(\mathcal{R}, \sigma, s)$ The prover randomly picks r, c in $\mathbb{F}_p$ and computes $\pi = ([\pi_A]_1, [\pi_B]_2, [\pi_C]_1)$:

$$\begin{aligned}
\pi_A &= \alpha + \sum_{i=0}^{m} s_i A_i(x) + r\delta \\
\pi_B &= \beta + \sum_{i=0}^{m} s_i B_i(x) + c\delta \\
\pi'_C &= \frac{\sum_{i=l+1}^{m} s_i(\beta A_i(x) + \alpha B_i(x) + C_i(x)) + h(x)t(x)}{\delta} \\
\pi_C &= \pi'_C + \pi_A c + \pi_B r - rc\delta
\end{aligned} \tag{A.3}$$

- $0/1 \leftarrow Verify(\mathcal{R}, \sigma, u, \pi)$: the verifier accepts the proof iff the following equation holds:

$$\begin{aligned}
[p_1]_T &= [\pi_A]_1 \cdot [\pi_B]_2 \\
[p_2]_T &= [\alpha]_1 \cdot [\beta]_2 \\
[p_3]_T &= \sum_{i=0}^{l} s_i [\frac{\beta A_i(x) + \alpha B_i(x) + C_i(x)}{\gamma}]_1 \cdot [\gamma]_2 \\
[p_4]_T &= [C]_1 \cdot [\delta]_2 \\
[p_1]_T &= [p_2]_T + [p_3]_T + [p_4]_T
\end{aligned} \tag{A.4}$$

Appendix B

FFT Techniques

Our QAP is a 3-matrix set of size $N \times M$. Working on $\mathbb{F}_p$, where p is a prime number and the order of the used elliptic curve, we find a generator g for our field. Having this, we find two values k and an extended size for N, called $N_e = 2^l$ (where l is an integer 'large enough') such that $p = kN_e + 1$. As can be seen, N_e is a power of 2, as 2-addicity is a desirable property for more efficient FFT algorithms. With this, now we can find our N_eth primitive root of unity:

$$\omega \equiv g^k \mod p \tag{B.1}$$

This generates our domain:

$$S = \{1, \omega, ..., \omega^{N_e - 1}\} \tag{B.2}$$

Now we need to compute three polynomials $A(z) = \{A_0, ..., A_{M-1}\}, B(z) = \{B_0, ...,$
$B_{M-1}\}, C(z) = \{C_0, ..., C_{M-1}\}$, where z is our toxic waste x defined in the Groth setup:

$$
\begin{aligned}
A_i(z) &= \sum_{j=0}^{N_e-1} L_{i,j} \cdot \frac{Lag_j(z)}{z - S_j} \\
B_i(z) &= \sum_{j=0}^{N_e-1} R_{i,j} \cdot \frac{Lag_j(z)}{z - S(j)} \\
C_i(z) &= \sum_{j=0}^{N_e-1} O_{i,j} \cdot \frac{Lag_j(z)}{z - S(j)}
\end{aligned}
\tag{B.3}
$$

where

$$
\begin{aligned}
Lag_1(z) &= \frac{z^{N_e} - 1}{N_e} \\
Lag_{j+1}(z) &= \omega Lag_j(z)
\end{aligned}
\tag{B.4}
$$

As such, the setup has defined three polynomials $A(x), B(x), C(x)$. Now, using a solution to our circuit w the prover computes the following values:

$$
\begin{aligned}
A &= \sum_{i=0}^{M-1} A_i \cdot w_i \\
B &= \sum_{i=0}^{M-1} B_i \cdot w_i \\
C &= \sum_{i=0}^{M-1} C_i \cdot w_i
\end{aligned}
\tag{B.5}
$$

And we can check:

$$
A * B - C = 0 \tag{B.6}
$$

Now, the prover computes the evaluation of three polynomials:

$$
\begin{aligned}
A_j &= \sum_{i=0}^{M-1} L_{i,j} \cdot w_i \\
B_j &= \sum_{i=0}^{M-1} R_{i,j} \cdot w_i \\
C_j &= \sum_{i=0}^{M-1} O_{i,j} \cdot w_i
\end{aligned}
\tag{B.7}
$$

Now, the prover will use $A(z), B(z), C(z)$ to compute the coefficients h of $H(x)$:

$$
H(x) = \frac{A(x)B(x) - C(x)}{Z(x)} \tag{B.8}
$$

In order to do so, the prover selects a random σ and computes a shifted domain $T = \{\sigma, \sigma\omega, ..., \sigma\omega^{N_e-1}\}$. He also sets $Z = \sigma^{N_e} - 1$ and does what follows:

- Computes 3 IFFTs in our domain S:

$$
\begin{aligned}
A_S &= IFFT(A, S) \\
B_S &= IFFT(B, S) \\
C_S &= IFFT(C, S)
\end{aligned}
\tag{B.9}
$$

- Computes 3 FFTs in our domain T:

$$
\begin{aligned}
A_T &= FFT(A_S, T) \\
B_T &= FFT(B_S, T) \\
C_T &= FFT(C_S, T)
\end{aligned}
\tag{B.10}
$$

- Computes $H = \dfrac{A_T B_T - C_T}{Z}$ point by point.
- Computes the shifted coefficients $h_T = IFFT(H, T)$ and it finally gets $h = h_T/\sigma$ point by point.

Appendix C

Bos-Coster

Let the pairs $(s_1, P_1), (s_2, P_2), ..., (s_n, P_n)$ be the elements of the multi-exponentiations to perform. We sort the list from large to small s_i, by means of a well-optimised binary heap. Then, while the list is larger than 1:

- $(s_1, P_1) = (s_1 - s_2, P_1)$
- $(s_2, P_2) = (s_2, P_1 + P_2)$
- if $s_1 = 0$, we remove this pair
- We sort the list again

When only one pair remains, $s_1 P_1$ is our solution.