Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

General-Purpose Zero-Knowledge Proofs for Verifiable Credentials

Master's Thesis

Damiano Mombelli

31.03.2025

Advisors: Prof. Dr. Kenneth Paterson, Dr. Martin Burkhart

Applied Cryptography Group
Institute of Information Security
Department of Computer Science, ETH Zurich

Cyber Defence Campus, armasuisse

Schweizerische Eidgenossenschaft
Confédération suisse
Confederazione Svizzera
Confederaziun svizra

armasuisse
Science and technology

Acknowledgements

This thesis has been a very challenging undertaking, and it would not have been possible without the help and support of many people. First, I want to thank my supervisor, Dr. Martin Burkhart, for his guidance throughout this thesis and always being available to give feedback and answer my questions. I am really grateful. I also want to thank the Cyber Defence Campus for providing a comfortable and focused workspace.

I want to express my gratitude to Prof. Dr. Kenny Paterson for overseeing directly my work and providing invaluable feedback throughout these six months.

Finally, I would like to thank my family, my parents and siblings, who have always been incredibly supportive and caring, love you. To my friends, thank you, in particular to Daniele and Andrea. They have also been working on their master's theses these past months and have accompanied me through some stressful times.

I also wish to acknowledge the Swiss e-ID dev team, in particular Jonas Niestroj, for providing early access to their source code.

This work was supported by the Cyber Defence Campus, Zürich.

Abstract

The Swiss Confederation has been tasked with developing the Swiss e-ID, an infrastructure for digital identities, following the principles of SSI, which enforce strong privacy requirements. These requirements are, however, hard to achieve with classical, standard methods such as ECDSA signatures. Recently, verifiable credentials based on zero-knowledge proofs have been proposed as an alternative to achieve higher privacy guarantees. In this thesis, we study the feasibility of implementing flexible, privacy-preserving verification logic for anonymous credentials using general-purpose zero-knowledge proofs. We provide an overview, comparison, and performance analysis of state-of-the-art zero-knowledge frameworks, and we design flexible credential verification logic using arithmetic circuits. We then implement a proof-of-concept framework for anonymous credentials based on zk-SNARKs and integrate it into the Swiss e-ID infrastructure. Our work highlights the flexibility of this approach, for example, we can seamlessly prove properties of values that were computed, or aggregated, from claims of multiple linked credentials. We also uncover issues and limitations of current zero-knowledge frameworks, especially regarding performance. For these, we indicate possible ways in which they could be addressed by future work. We show that this approach is practical with current technologies for reasonably complex statements, such as validating a credential, while future research is very likely to allow for much more complex verification logic.

Contents

Chapter 1

Introduction

1.1 Motivation

The Swiss federal government was tasked, in 2021, with developing the Swiss e-ID, a system for digital identities based on the principles of Self-Sovereign Identity (SSI), which enforce strong privacy requirements. Most important among these principles are the users' ownership of their digital identities, and their control over what information is revealed to which service provider. This approach is in stark contrast with current internet single sign-on architectures, where the privacy and control of the users over their identities is severely weakened by the reliance on central identity providers.

To meet these principles, the Swiss e-ID defines its own set of design principles and goals [38]:

1. Privacy by design, the system must, by default, protect the privacy of the users.
2. Minimisation, the holder (user) must be able to partially present a credential, only disclosing the necessary information. Additionally, a disclosure should also reveal the least amount of information needed for a given task. For example, if a service provider only asks for a minimum age, the the exact age of the user (i.e. their birthdate) should not be disclosed. This can be achieved using several techniques such as selective disclosures (hiding some claims of a credential), zero-knowledge or predicate proofs, and privacy preserving data structures such as cryptographic accumulators.
3. Decentralised storage, credentials should solely be store on a holder's device, this gives complete control to the user over every transmission of data.

However, at the time of writing, the current (beta version) implementation of the Swiss e-ID fails to satisfy some of these design principles, achieving

little to no privacy guarantees, due to the use of immutable, unique signatures and the necessity for holders to provide to verifiers (service providers) the full credential, albeit with non-disclosed claims being hidden, with each presentation. It additionally fails to provide full data-minimisation of selective disclosures due to not supporting zero-knowledge proofs, or any of the techniques mentioned above. Each disclosure is either all-or-nothing, for example, if we consider, again, the case where a verifier asks for a minimum age, the holder is forced to reveal their full birthdate. The Swiss e-ID team is aware of these problems (see discussion paper [38]), however, the government has decided to put interoperability with other digital identity infrastructures, in particular the EUDI, first and to pursue stronger privacy properties in a second phase.

There are some workarounds to these problems, such as batch issuance of one-time use credentials in order to circumvent the tracking made possible by the unique signatures of the credential, or using a series of boolean flags, for example stating that the age of the holder is above a certain threshold, to implement pseudo-zero-knowledge proofs for some limited use-cases. However, these solutions are impractical and necessitate more maintenance from the issuer's side. There have also been proposals using other technologies, such as BBS+ signatures, to achieve privacy and fully data-minimising selective disclosure, but these generally involve the use of less researched, less standardised cryptography.

In this thesis, we explore a different approach to solve these problems, based on recent works [7, 46, 81, 82] proposing general-purpose zero-knowledge proofs (in particular zk-SNARKs) as a basis for the implementation of anonymous verifiable credentials. zk-SNARKs allow for the execution of (almost) arbitrary logic and enable holders to prove in zero-knowledge statements about their credentials, such as the one in Listing 1.1. With this approach, the holder would send to the verifier a zero-knowledge proof showing that their credential satisfies (or does not satisfy) the given statement, while the verifier would check the validity of such a proof.

```
1 out = eid.is_valid AND eid.age >= 18
2         AND eid.locality == "Zuerich"
3         AND eid.nationality == "CH"
4         AND diploma.is_valid
5         AND (diploma.university == "ETHZ"
6         OR diploma.university == "EPFL")
7         AND LINKED(eid, diploma)
```

Listing 1.1: Example of a zero-knowledge statement

This solution would be perfectly zero-knowledge and very flexible, additionally credentials would never leave the holder's wallet. We can now also make use standardised but non-privacy-preserving primitives such as ECDSA signatures or status-list-based revocations without compromising

the holder's privacy. This is possible because we can prove properties about them in zero-knowledge without revealing information, such as an ECDSA signature or an index of a status list, which would uniquely identify a holder.

This thesis studies the feasibility of this approach, focusing, in particular, on the performance of the system, given the end-goal of deployment on mobile devices, and on the complexity of implementation. We explore the state-of-the-art of zero-knowledge frameworks, and we extend the infrastructure of the Swiss e-ID beta-version with zero-knowledge proofs. We design and implement flexible verification logic for anonymous credentials, and we integrate it in an end-to-end proof of concept of the system. Our work shows this approach to be feasible with current technologies for reasonably complex statements, such as the complete validation of an anonymous credential, for which we can generate proofs in 2.5 seconds. Additionally, we highlight the flexibility and expressiveness of our system, which is able, for example, to prove properties about values that were aggregated, or computed, from multiple claims across different, linked credentials. We also uncover some limitations, in particular in terms of performance, and we point to possible workarounds and optimisations.

1.2 Summary of Contributions

In this section we provide a summary of the main contributions of this thesis:

1. We conduct an overview of existing, state-of-the-art, general-purpose zero-knowledge proof frameworks. We highlight their main design features, cryptographic assumptions, theoretical performance, and suitability to our use-case. Additionally, we conduct benchmarks on a few select frameworks by measuring the execution time for generating and verifying proofs and their size.

2. We develop a set of primitives for expressing flexible zero-knowledge statements on verifiable credentials. These include verifying the integrity and validity of a credential, anonymous revocation, credential linking, and hardware binding. We show the importance of choosing zk-SNARK-friendly operations and point to fast implementations for them, in particular for hash functions and signature schemes based on ECDSA.

3. We develop an end-to-end proof-of-concept implementation of verifiable credentials based on general-purpose zero-knowledge proofs. For this purpose we and extend the beta version of the Swiss e-ID project to support zero-knowledge proofs. Our modifications are minimal and non-intrusive and can seamlessly work alongside SD-JWT/ECDSA credentials without replacing them. Additionally, we also create some

concrete examples of the possible statements that can be expressed with our system and provide for each of them a cost analysis.

1.3 Outline

In Chapter 2 we introduce some background knowledge related to our work, focusing in particular on the Swiss e-ID project and on zero-knowledge proofs.

The main contributions are then presented, divided as follows: in Chapter 3 we provide an overview and comparison of existing general-purpose zero-knowledge proof system,s and we provide a performance evaluation for some of them. In Chapter 4 we introduce the main basic components with which we can express flexible credential verification logic, and we provide for each of them an analysis of its cost and additional considerations for their usage. We then introduce our proof-of-concept implementation of zk-SNARK-based verifiable credentials in Chapter 5. We explain how we can integrate it, in a non-invasive way, into the Swiss e-ID infrastructure, how we implement the verification logic, and present its various components. Then, we describe the end-to-end workflow of our system starting from credential issuance and going through all the steps needed for then generating and verifying proofs about credentials. Finally, we provide a series of concrete examples of verification logic, and we show how expensive they are in terms of prover and verifier execution time. In Chapter 6 we cover some additional discussion points which didn't find a proper place elsewhere, and in Chapter 7 we discuss some related works. Finally, we present our conclusions and possible avenues for future work in Chapter 8.

Chapter 2

Background

In this chapter, we introduce the theoretical background of our work with a particular focus on the Swiss e-ID project and on zero-knowledge proofs. We first introduce the notation that we adopt over the course of this document (Section 2.1), and the Swiss e-ID project (Section 2.2). We provide definitions for zero-knowledge proofs (Section 2.3), proofs of knowledge (Section 2.4), and zk-SNARKs (Section 2.5). Then we introduce arithmetic circuits and their satisfiability problem (Section 2.6), and the rank-1 constraint system (Section 2.7). Finally, we briefly describe ECDSA signatures (Section 2.8).

2.1 Notation

We assume the following conventions:

- $e \leftarrow_{\$} U$ indicates sampling an element e uniformly at random from a set U.
- $\langle G \rangle$ indicates the cyclic subgroup of an elliptic curve C generated by the generator $G \in C$.
- $s \times P$ indicates the scalar multiplication of a point P on an elliptic curve with scalar s
- A sequence of length n is indexed from 0 to $n - 1$.
- `Monospaced font` is used to indicate code or elements relative to code.
- Over the course of this thesis we use the terms holder, prover, user, and subject interchangeably to indicate the entity who olds a set of credentials and generates verifiable presentations.

2.2 The Swiss e-ID Project

In this section we describe the Swiss e-ID project [38], focusing in particular on the privacy of its design. We describe its design principles (Section 2.2.1) and the possible choices of technologies ((Sections 2.2.2) and 2.2.3), finally we introduce its current, at the time of writing, implementation (Section 2.2.4).

2.2.1 Design Principles

The Swiss e-ID project is being implemented following on the principles of Self-Sovereign Identity (SSI) [1]. To meet these principles, the Swiss e-ID project defines a set of design goals (see Section 3 in [38]):

Privacy by design The system should protect by default the privacy of the holders, this applies w.r.t. both issuers and verifiers:

- The Swiss e-ID follows the SSI principles with three different roles: issuers, holders, and verifiers (see Figure 2.1). These roles communicate directly with each other:
  - Issuer issue signed credentials to holders.
  - Holders store the credentials that were issued to them inside their personal wallet applications. Additionally, they can create verifiable presentations of their credentials to authenticate themselves to a verifier.
  - Verifiers validate verifiable presentations of credentials.
- Issuer-holder privacy: it should NOT be possible for an issuer to know if and how a credential that it has issued is used.
- Verifier-holder privacy: it should NOT be possible for a verifier (or multiple colluding issuers and verifiers) to track a holder by observing the presentations of their credentials.

Data minimisation The system strives to minimise the amount of information that is exchanged between parties and is achieved with multiple strategies:

- The holder should have the ability to partially present a credential (selective disclosure).
- Where possible, information should be presented as a derivation (e.g. predicate or zero-knowledge proofs).
- Unnecessary data-flows are avoided via the SSI paradigm.

Decentralised storage Verifiable credentials are solely stored in a holder's wallet. As a consequence each verifiable presentation is under the direct control of, and needs to be authorized by the user.

Additional Requirements Principles A set of additional requirement were requested during public consultations for the development of the Swiss e-ID:

- Support for third party credentials (e.g. driving licences, membership cards, etc.).
- Support for hardware binding to bind a credential to a holder's device.
- Support for credential linking to achieve binding w.r.t. the holder when presenting multiple credentials.
- International compatibility, in particular with the European Union's Architecture Reference Framework [5].

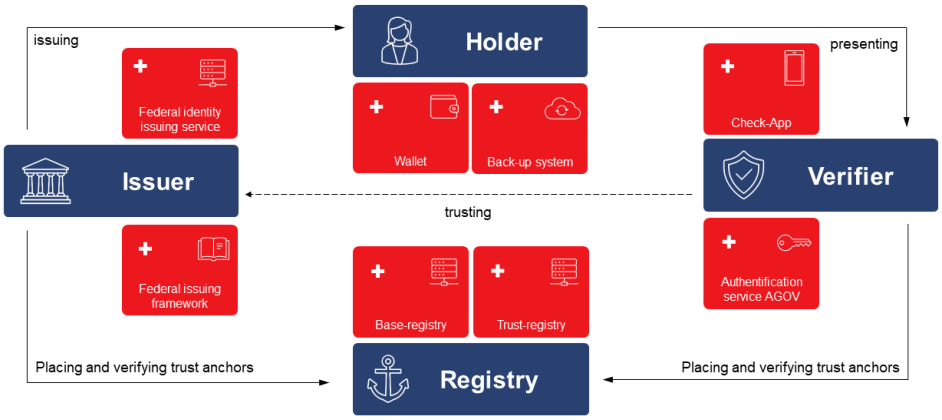

Figure 2.1: Swiss e-ID trust infrastructure[1]

The concrete implementation of the Swiss e-ID considers two possible technological scenarios (called Scenario A and Scenario B) which offer various trade-offs w.r.t. the target design principles.

2.2.2 Scenario A

With this scenario, the Swiss e-ID plans to implement a subset of the technologies proposed in the European Union's Architecture Reference Framework (ARF) for the European Digital Identity (EUDI) project. The ARF is based on well known and standardised technologies such as ECDSA and SHA256. Credentials are stored in an SD-JWT [91] format and the OID4VCI/OID4VP protocol is used to issue and present credentials.

[1] https://github.com/e-id-admin/open-source-community/assets/12694135/3fe1b206-7563-403a-ac92-be9e28d582e6

An SD-JWT file is a JSON Web Token implementing Selective Disclosures by including only the salted hash of each claim in the SD-JWT file. When disclosing a claim, the holder sends, together with the SD-JWT credential, a disclosure containing the value of the claim and the salt that was used to compute the corresponding hash.

While the use of standard cryptography is desirable, this scenario also raises some concerns, especially in regard to the privacy of the holders. In fact, ECDSA signatures are unique and immutable, and enable a verifier to trace a holder by simply tracking the signatures. Additionally, the salted hashes of all claims of a credential are always included in a verifiable presentation, and they can also be used to uniquely identify a holder. There are some possible solutions to these problems, such a batch issuance of one-time use credentials, but they tend to be impractical for large numbers of holders and presentations. Another downside of this scenario is the lack of support for zero-knowledge proofs, making selective disclosures only partially data-minimising.

In summary the main advantages of adopting this scenario are:

- Compatibility with the EU-EID.
- Use of common, well known, standardised technologies.
- Broad hardware support of the cryptography used.

While the main disadvantages are:

- No verifier-holder privacy which can lead to traceability of holders.
- Possible traceability also through revocation of credentials.
- No support for predicate or zero-knowledge proofs.
- Selective disclosures are only partially data-saving.

2.2.3 Scenario B

The goal of this scenario is to adopt a set of technologies that offer a higher level of privacy for the holders compared to Scenario A. In particular it aims at using JSON-LD-verifiable credentials [89] and BBS+ signatures [73] which are both commonly used protocols in SSI-based systems. While these protocols have seen less research and standardisation compared to their counterparts in scenario A, they meet important privacy property, especially in regard to unlinkability. In fact BBS+ has the ability of blinding, i.e. randomising, its signatures. This results in verifiable presentations of credentials that do not contain any constants and therefore cannot be linked to a specific holder. Selective disclosures are also supported, and, additionally, BBS+ is compatible with (general purpose) zero-knowledge proofs as shown by Yamamoto et al. [97, 98]

In summary the main advantages of adopting this scenario are:

- Unlinkability for holders.
- Selective disclosures are fully data-saving.
- Support for predicate and zero-knowledge proofs.
- Privacy preserving revocations may be possible (e.g. using cryptographic accumulators).

While the main disadvantages are:

- No interoperability with the EU-EID, at least initially.
- Use of less known, less researched cryptography with little to no hardware support.
- More complex integration.

2.2.4 Current Implementation

As of December 2024, the Swiss e-ID team opted to follow scenario A, at least for the implementation of the beta version of the system. According to the development team, this needs to be considered only as a "starting point for the introduction of government issued digital identities and a trust ecosystem. Continuous evolution is expected and will be necessary to ensure long-term service continuity, security and interoperability". In practice, we can expect the eventual switch to a set of privacy-preserving technologies, but as it stands at the time of writing, the implementation of the Swiss e-ID, in its first version, does not satisfy all of its envisioned design goals, in particular in terms of privacy and data-minimisation.

The format of the verifiable credentials is as specified by the SD-JWT-VC IETF standard [91] with some additional requirements [40]: compact serialisation must be supported, and the claims in Table 2.1 are always disclosed. Additionally, issuers, holders, and verifiers must support P-256 (secp256r1) as a key type with E256 JWT algorithm for signature generation and verification. Listing 2.1 shows an example of an SD-JWT credential, including its selective disclosures hashes.

```
{
    "_sd": [
        "-XOkB3ctxrW9B83zkDmKV9hEkGdJT4YMTULyiSekI5o",
        "0bgjSAZek2hv_RAnBsg_h8beYtBXzu7mspIUYH1Xu4M",
        "6-OGPwfAw8OJJJfGwD-IPTA5MHdfzOy8dBnoUVYI-xI",
        "ZUl9ASTM-MCkQ47YlgmaHPvBfEm33OpKWuigWQCeQZc",
        "bZddAPKrGBjvnpPI5zEZeQN3h5OFYdp7us1EnzGJmvk",
        "gih0888L6wri-_RWPJtp2E5colQUYJA2RReMIpAd_ts",
        "o4N2HerrK0e4O631HrrZIZOZyZ2tg0vjpjXf9L8F2EE",
        "tRV1AJAmPex0Ikjo81dtK1VUlbNUm7Gu0Be8NaJiiMM",
```

Claim	SD-JWT	Description
`_sd`	MUST	Array containing slated hashes of the claims, used for selective disclosure
`vct`	MUST	Specifies the type of credential (e.g. `identity_credential`)
`iss`	MUST	DID linked to the issuer, used to retrieve its public key
`iat`	SHOULD	Time of issuance of the credential
`nbf`	SHOULD	Time before which the credential is not valid
`exp`	SHOULD	Time after which the credential is not valid
`status`	SHOULD	Status of the credential
`sub`	MAY	Identifier of the subject of the verifiable credential
`cnf`	MAY	JWK for enforcing device/hardware binding

Table 2.1: SD-JWT claims that must be supported by the Swiss e-ID verifiable credential format [40]

```
            "vEP3p8ASztOhHgaSR2nolPpFZG1VhAYe3F5fSa4RBYY",
            "vTIDfLXPS9UhHkn402NsQ_ap0VDcoiKpp4YgxZgZRro",
            "ydGgzbR2I2ZQU5Tt_ogvGSNJBL7v9trUzuEWtqCWMUA"
        ],
        "_sd_alg": "sha-256",
        "vct": "https://credentials.example.com/
    identity_credential",
        "iss": "did:example:123456789",
        "nbf": 1740441600,
        "exp": 2055888000,
        "iat": 1741694987,
    }

```

Listing 2.1: Example of a Swiss e-ID SD-JWT credential

2.3 Zero-Knowledge Proofs

A zero-knowledge proof of a statement, first introduced by Goldwasser et al. [50], is an interactive protocol between a prover and a verifier satisfying the following properties:

1. **Completeness**: any true statement has a convincing proof of validity.
2. **Soundness**: no false statement has a convincing proof of validity.
3. **Zero-knowledge**: a proof of validity for a true statement leaks no information about the statement, other than the fact that it is true.

More formally, a couple of polynomial time algorithms (P, V) is a zero-knowledge proof for some language $\mathcal{L}$ and security parameter $\lambda \in \mathbb{N}$, if the following conditions are satisfied:

1. **Completeness**: for every $x \in \mathcal{L}$

$$\Pr\left[\ V(\pi) = 1 \ \middle|\ \pi \leftarrow P(x)\ \right] = 1$$

2. **Soundness**: for every $x \notin \mathcal{L}$ and (dishonest) prover P^*

$$\Pr\left[\ V(\pi) = 1 \ \middle|\ \pi \leftarrow P^*(x)\ \right] \leq negl(\lambda)$$

3. **Zero-knowledge**: there exists a polynomial time simulator S such that for all polynomial time distinguishers D and for all $x \in \mathcal{L}$

$$\Pr\left[D(\pi) = 1 \middle| \begin{array}{c} x \leftarrow D() \\ \pi \leftarrow P(x) \end{array}\right] = \Pr\left[\ D(\pi) = 1 \ \middle|\ \begin{array}{c} trap \leftarrow S(1^\lambda, \mathcal{L}) \\ x \leftarrow D() \\ \pi \leftarrow S(x, trap) \end{array}\right]$$

2.4 Proofs of Knowledge

A proof of knowledge is an interactive protocol in which a prover convinces a verifier of possessing knowledge of "something", called witness, satisfying the following properties:

- **Completeness**: a prover with knowledge of a witness succeeds in convincing a verifier of such knowledge.
- **Proof of knowledge**: no prover without knowledge of the witness can succeed in convincing a verifier.

More formally, given some language $\mathcal{L}$ in NP, witness set W, and relation $R := \{(x, w) \ : \ x \in \mathcal{L},\ w \in W(x)\}$, a proof of knowledge for relation R and security parameter $\lambda \in \mathbb{N}$ is a couple of polynomial time algorithms (P, S) satisfying the following:

1. **Completeness**: for every $(x, w) \in R$

$$\Pr\left[\ V(x, \pi) = 1 \ \middle|\ \pi \leftarrow P(x, w)\ \right] = 1$$

2. **Proof of knowledge (soundness)**: for every (dishonest) prover P^* there exists a polynomial time extractor E such that for every auxiliary input $z \in \{0,1\}^{\text{poly}(\lambda)}$

$$\Pr\left[\ \begin{array}{c} V(x, \pi) = 1 \\ (x, w) \notin R \end{array} \ \middle|\ \begin{array}{c} (x, \pi) \leftarrow P^*(z) \\ (x, w) \leftarrow E(z) \end{array}\ \right] \leq negl(\lambda)$$

Additionally, a proof of knowledge can also be zero-knowledge if there exists a polynomial time simulator S such that for all polynomial time distinguishers D, for all $x \in \mathcal{L}$ and for all every input $z \in \{0,1\}^{\text{poly}(\lambda)}$

$$\Pr\left[\begin{array}{c|c} D(\pi) = 1 & x \leftarrow D(z) \\ (x,w) \in R & \pi \leftarrow P(x,w) \end{array}\right] = \Pr\left[\begin{array}{c|c} D(\pi) = 1 & trap \leftarrow S(1^\lambda, R) \\ (x,w) \in R & (x,w) \leftarrow D() \\ & \pi \leftarrow S(z,x,trap) \end{array}\right]$$

2.5 zk-SNARKs

A Zero-knowledge Succinct Non-interactive Argument of Knowledge (zk-SNARK) is a non-interactive zero-knowledge proof system supporting "general computations", i.e. they can prove (in zero-knowledge) the correctness of any polynomial time computation. They generally operate in the Common Reference String Model (CRS), which requires a trusted setup in order to be secure (indicated by the protocol G below). The term 'Succinct' indicates, in particular, that both the proof size, and the time it takes to verify it are small.

More formally, a triple of polynomial time algorithms (G, P, V) is a zk-SNARK for language $\mathcal{L}$ in NP, witness set W, relation $R := \{(x,w) : x \in \mathcal{L}, w \in W(x)\}$ and security parameter λ is where

- $G(1^\lambda, L) \to (\sigma, \tau)$ outputs a proving/verifying key-pair (σ, τ) which is treated as a public parameter
- $P(\sigma, x, w) \to \pi$ outputs a non-interactive proof π of statement $x \in \mathcal{L}$ and witness $w \in W(x)$
- $V(\tau, x, \pi) \to \{0,1\}$ outputs the validity of proof π w.r.t. statement x

if the following conditions are satisfied:

1. **Completeness**: for every $(x,w) \in R$

$$\Pr\left[V(\tau,x,\pi) = 1 \;\middle|\; \begin{array}{c} (\sigma,\tau) \leftarrow G(1^\lambda, L) \\ \pi \leftarrow P(\sigma,x,w) \end{array}\right] = 1$$

2. **Proof of Knowledge** (soundness): for every (dishonest) prover P^* there exists a polynomial time extractor E such that for every auxiliary input $z \in \{0,1\}^{\text{poly}(\lambda)}$

$$\Pr\left[\begin{array}{c|c} V(\tau,x,\pi) = 1 & (\sigma,\tau) \leftarrow G(1^\lambda, L) \\ (x,w) \notin R & (x,\pi) \leftarrow P^*(z,\sigma,\tau) \\ & (x,w) \leftarrow E(z,\sigma,\tau) \end{array}\right] \leq negl(\lambda)$$

3. **Zero-knowledge**: there exists a polynomial time simulator S such that for all polynomial time distinguishers D and every auxiliary input

$z \in \{0,1\}^{\text{poly}(\lambda)}$

$$\Pr\left[\begin{array}{c|c} D(\pi) = 1 \\ (x,w) \in R \end{array} \middle| \begin{array}{c} (\sigma,\tau) \leftarrow G(1^\lambda, L) \\ (x,w) \leftarrow D(z,\sigma,\tau) \\ \pi \leftarrow P(\sigma,x,w) \end{array}\right] =$$
$$\Pr\left[\begin{array}{c|c} D(\pi) = 1 \\ (x,w) \in R \end{array} \middle| \begin{array}{c} (\sigma,\tau,trap) \leftarrow S(1^\lambda, L) \\ (x,w) \leftarrow D(z,\sigma,\tau) \\ \pi \leftarrow S(z,\sigma,x,trap) \end{array}\right]$$

4. **Succinctness**: there exists a universal, independent of R polynomial p such that:
   - the key generator G runs in time $p(\lambda + |L|)$
   - the prover P runs in time $p(\lambda + |L|)$
   - the verifier V runs in time $p(\lambda + |x|)$
   - an honestly generated proof π has size $p(\lambda + |x|)$

2.6 Arithmetic Circuits and Circuit Satisfiability

Arithmetic circuits are an NP language where each instance of an arithmetic circuit is an object composed of gates and wires and is defined over a finite field. Gates can either be inputs to the circuits or additions and multiplications over the finite field. Wires, instead, carry values between gates.

Formally, an arithmetic circuit C over a finite field $\mathbb{F}_p$ is an acyclic directed graph where every node with *indegree* zero is an input gate (which for our use-case can either be public or private) and can either represent a variable or a field element in $\mathbb{F}_p$. All other gates are either modular additions or modular multiplications. Edges in an arithmetic circuits can be labelled with field elements, which constrain the values these wires are allowed to carry.

The *circuit satisfiability problem* is defined as the problem of finding an assignment $s = [s_0, \ldots, s_{n-1}]$ to the input gates of a circuit C such that all the constraints in the C (enforced by labelling its wires) are satisfied, such an assignment s is called a *valid solution* of the circuit C w.r.t. its constraints. It is possible to prove knowledge of a valid solution of a circuit in zero-knowledge using zk-SNARKs, this is the basis for many of the proof systems that we are going to encounter over the course of this document.

Figure 2.2 shows a very simple example of circuit satisfiability were we constrain the output of the arithmetic circuit expressing the operation $a \cdot b$ to be equal to 33 by labelling the corresponding wire. A possible valid solution for this circuit is $\{a = 11, b = 3\}$, and a (zero-knowledge) proof-of-knowledge

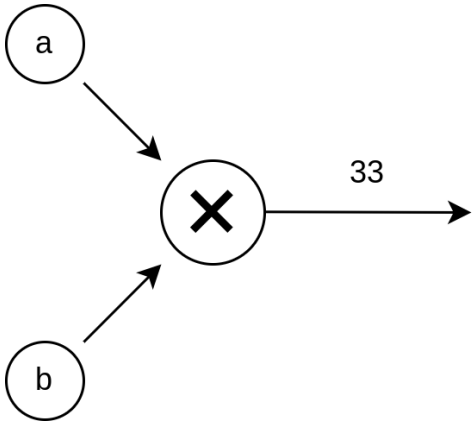

Figure 2.2: Example of a constrained arithmetic circuit

of a valid solution for the circuit would convince a verifier that the prover knows the factorisation of the number 33.

2.7 Rank 1 Constraints and the Complexity of Circuits

A rank 1 Constraint Systems (R1CS) is a sequence of equations

$$A_i \times B_i - C_i = 0$$

where A, B, C are three linear combinations of the form

$$\begin{aligned} &a_{i,0}w_0 + \cdots + a_{i,n-1}w_{n-1}, \\ &b_{i,0}w_0 + \cdots + b_{i,n-1}w_{n-1}, \\ &c_{i,0}w_0 + \cdots + c_{i,n-1}w_{n-1} \end{aligned}$$

respectively, where $[a_{i,0}, \ldots, a_{i,n-1}]$, $[b_{i,0}, \ldots, b_{i,n-1}]$, and $[c_{i,0}, \ldots, c_{i,n-1}]$ are vectors of constants and $[w_0, \ldots, w_{n-1}]$ is a solution for the constraint system. A solution w is valid if and only if it satisfies all equations in the constraint system. We say that a constraint $A_i \times B_i - C_i = 0$ is linear when at least one between A_i and B_i only has the constant vector, and is non-linear otherwise

R1CS can be used to express an arithmetic circuit. Finding a solution for an arithmetic circuit represented using R1CS is equivalent to finding the solution of the corresponding constraint system.

For example, the R1CS representation of the circuit in Figure 2.2 consists of a single non-linear constraint

$$[1 \cdot a] \times [1 \cdot b] - 33 = 0.$$

A possible satisfying assignment for this R1CS instance is $\{a = 11, b = 3\}$.

Of course, the bigger (more complex) a circuit is, and in turn the resulting R1CS instance, the more expensive it is to generate and verify a proof for it. In the context of zero-knowledge proofs of circuit satisfiability, the complexity of an arithmetic circuit is defined as the number of non-linear constraints in its R1CS representation, which is more-or-less equivalent to the number of multiplication gates in the circuit. This is the main metric that we employ to describe the cost of a statement.

2.8 Elliptic Curve Digital Signature Algorithm

Given an elliptic curve C defined over a finite field of prime order p and a generator $G \in C$ of a subgroup of C with prime order n, an Elliptic Curve Digital Signature Algorithm (ECDSA) scheme consists of a triplet of efficient algorithms $(Gen, Sign, Verify)$ where:

- $Gen() \rightarrow (sk, pk)$ generates a secret/public key-pair, where $sk = d$ for $d \leftarrow_{\$} \mathbb{Z}_n$ and $pk := Q = d \times G$.
- $Sign(sk, m) \rightarrow (r, s) = \sigma$ computes the signature (r, s) of the message m w.r.t. the secret key sk.
- $Verify(pk, \sigma, m) \rightarrow \{0, 1\}$ checks whether σ is a valid signature of message m w.r.t. the public key pk.

The signing and verification algorithms are defined by RFC 6979 [80] and are shown in algorithms 1 and 2 respectively. The choice of the elliptic curve C is critical for the security of the resulting signature scheme. In particular one should use a standardised, safe, curves such as the ones defined by the NIST standard for elliptic curve cryptographic [29].

Algorithm 1 $Sign(sk = d \in \mathbb{Z}_n, m)$

1: # Let H be a cryptographic hash function (e.g. SHA256)
2: $h \leftarrow \mathtt{H}(m) \mod n$
3: $k \leftarrow_{\$} \mathbb{Z}_n^+$
4: $(x, y) \leftarrow k \times G$
5: $r \leftarrow x \mod n$
6: **if** $r = 0$ **then**
7: **goto** 3
8: **end if**
9: $s \leftarrow k^{-1}(h + rd) \mod n$
10: **if** $s = 0$ **then**
11: **goto** 3
12: **end if**
13: **return** (r, s)

Algorithm 2 $Verify(pk = Q \in \langle G \rangle, \sigma = (r, s), m)$

1: # Let O be the neutral element of G w.r.t. point addition
2: **assert** $(Q \neq O) \wedge (Q \in C) \wedge (n \times Q = O)$
3: **assert** $(r \in \mathbb{Z}_n^+) \wedge (s \in \mathbb{Z}_n^+)$
4: # Let H be a cryptographic hash function (e.g. SHA256)
5: $h \leftarrow \mathtt{H}(m) \mod n$
6: $k \leftarrow_{\$} \mathbb{Z}_n$
7: $u_1 \leftarrow hs^{-1} \mod n$
8: $u_2 \leftarrow rs^{-1} \mod n$
9: $(x, y) \leftarrow u_1 \times G + u_2 \times Q$
10: **if** $(x, y) = O$ **then**
11: **return** `false`
12: **end if**
13: **if** $r \neq x \mod n$ **then**
14: **return** `false`
15: **end if**
16: **return** `true`

Chapter 3

Overview and Comparison of General-Purpose Zero-Knowledge Proof Systems

In this chapter, we provide an overview and analysis of existing general-purpose zero-knowledge proof systems and frameworks. First, we introduce the relevant literature on the topic, focusing on research milestones and noteworthy proof systems (Section 3.1), then, we present existing implementations, frameworks, and libraries of general-purpose zero-knowledge proof systems and highlight their main features and pros and cons (Sections 3.2 and 3.3). Finally, we compare the performance of various implementations of proof system by measuring their prover and verifier time and size of the generated proofs (Section 3.4).

Generally, we can categorise zero-knowledge proof frameworks into two different groups: backends and frontends. Backends (Section 3.2) are written in some general-purpose programming language and directly implement a proof system. They are used for generating and validate proofs and take as inputs a statement, represented using some constraint system, such as R1CS or AIR, a set of publicly known values called public input, and either a witness for the statement (when generating a proof), or the proof itself (when validating it). The witness is equivalent to a private input for the statement, and is only known to the prover. Frontends (Section 3.3), instead, are implementations of some domain-specific language (DSL) which allow developer to express statements in a more intuitive way compared to writing them as a set of constraints. These DSLs can either be similar to high-level languages such as Python, Java or Rust, or more low-level such as constraint-based DSLs, which rely on a lower level of abstraction (e.g. arithmetic circuits) to express statements. A frontend also implements a compiler to translate a statement described using its DSL into a set of constraints that can be pro-

vided as input to a backend.

3.1 Literature Overview

In this section we provide an overview of existing research and literature on the topic zero-knowledge proofs. We start from their inception (Section 3.1.1) and we then review the most important research milestones which lead to the development of zk-SNARKs (Section 3.1.2). From there, we follow their evolution into first universal zk-SNARKs (Section 3.1.3), and then zk-STARK (Section 3.1.4). Finally, we summarise all major proof systems and their main features (Section 3.1.5).

3.1.1 Introducing zero-knowledge proofs

Zero-knowledge proofs, first introduced by the seminal work of Goldwasser et al. [50], are a cryptographic primitive which allows an untrusted prover to convince a verifier that some statement is true without revealing any additional information about the statement (other than the fact that it is true). This is achieved via a series of interactions between the prover and the verifier. In our context, a statement is usually associated with a public input, known by both the prover and verifier, and a witness (or private input) known only by the prover. A very common example of a zero-knowledge proof is demonstrating knowledge of a private key associated with a particular public key without revealing any information relating to said private key. For this purpose, one can use, for example, Schnorr's protocol. In this case, the private key is the witness of the proof, while the public key and any additional public value are its public inputs. Interestingly, Schnorr's protocol is not only a zero-knowledge proof, but it is also a proof of knowledge, i.e. the protocol not only convinces the verifier of the correctness of the statement, but also that the prover has, indeed, knowledge of the value of the private key.

Blum et al. [20] later introduced non-interactive zero-knowledge proofs in the common reference string (CRS) model, which, contrary to traditional zero-knowledge proofs, allow for the verification of a claim without the need for the prover to interact with the verifier, achieved by replacing the interactions with a CRS. In general, most proof systems can be also made non-interactive using the Fiat-Shamir heuristic [45]. The scope of zero-knowledge proofs was then broadened to include more generic statements thanks to the work of Golderich et al. [10, 49], which showed that, under the assumption of a secure probabilistic encryption scheme, all languages in NP have zero-knowledge proofs. Until this point, proofs, such as Schnorr's protocol or range proofs [25], relied on specific mathematical tricks and could only be applied to their intended use-case. Using more expressive

languages, such as arithmetic circuits, enables us to prove more complex and almost arbitrary statements without the need to rely on use-case-specific strategies, with the only restriction being the general inability of using input-dependent control flow.

3.1.2 From loose succinctness to zk-SNARKs

When it comes to the performance of a proof system, the amount of information exchanged between the prover and the verifier is an important parameter to consider. The first steps towards minimising the amount of exchanged information, i.e. towards succinctness, were taken with the seminal work by Kilian [67]. Pairing-based proof systems were then introduced by Groth, Ostrovsky and Sahai [55, 56, 57, 58] and managed to produce the first linear size proofs (w.r.t. the number of non-linear constraints) using standard assumptions. (Pairing-based) Zero-knowledge succinct non-interactive arguments of knowledge (zk-SNARKs) were then introduced by Groth [54] and Chiesa et al. [19]. There exist many types of zk-SNARKs, and they differ from each other on different factors such as cryptographic assumptions, prover and verifier complexity, proof size, need for a trusted setup, and whether they are (plausibly) post-quantum. Generally, we can group them into those which operate under the CRS model (for circuit satisfiability) and need a trusted setup for each circuit, those which still rely on the CRS model but only need a trusted setup for all circuits up to some upper bound circuit size (called universal proof systems), and those which do not need a trusted setup (called transparent proof systems) and instead rely on publicly available randomness or other tools such as commitment schemes.

Initially, the size of the CRS and the prover's computation both scaled quadratically with the circuit size, which significantly negatively impacted the performance of the proof systems. Lipmaa [70] reduced the size of the CRS to be quasi-linear in the circuit size. However, the prover's computation still scaled quadratically. It was the work of Gennaro et al. [48] which, using quadratic arithmetic programs (QAPs) and quadratic span programs (QSPs), finally managed to reduce the CRS size to scale linearly and the prover's computation to scale quasi-linearly in the circuit size. This optimisation led to the development of many (almost) practical zk-SNARKs based on the knowledge of exponent (KoE) assumption, such as Pinocchio [77], Pantry [24], and Geppetto [33]. Lipmaa [71] then suggested more efficient quadratic span programs using error-correcting codes, while Groth et al. [37] refined QSPs into square span programs (SSPs) that produce proofs consisting of only 4 group elements. Following these innovations, Groth [53] introduced a new pairing-based proof system (called Groth16), which is, to this day, still one of the most widely used and fastest zk-SNARKs deployed in real-world applications.

Groth16 is a zk-SNARK for boolean and arithmetic circuit satisfiability which operates over asymmetric parings. The choice of using asymmetric pairing is dictated by their higher efficiency, but theoretically, we can instantiate Groth16 over any type of pairing. The proofs generated by Groth16 are extremely small, consisting of only three group elements. Consequently, the proof verification is fast and consists of a single pairing product equation using only three pairings. In order to achieve better prover time, Groth16 employs less conservative (from a security perspective) optimisations compared to the aforementioned proof systems, and it thus relies on stronger cryptographic assumptions. Namely, it is argued to be secure in the generic group model (GGM), which suffers from some of the same problems as the random oracle model (ROM).

3.1.3 Towards universal proof systems

The need of a per-circuit trusted setup hindered the deployment of zk-SNARK in real-world applications, both in terms of how cumbersome it is to introduce new circuits and in terms of the size of the proving/verifying keys-pair generated by the trusted setup, which can reach sizes of several MBs for higher counts of non-linear constraints. To circumvent at least the first problem, lots of research focused on universal proof systems.

The first steps towards universality were made in 2012 by Ben-Sasson et al. [12], who showed that random-access machine (RAM) computations could be efficiently reduced to a circuit satisfiability problem. Thanks to this discovery, together with improvement from Gennaro et al. [48], a series of RAM-based zk-SNARKs were developed over the years, where some of the more prominent are TinyRAM [16], vnTinyRAM [17], Buffet [93] and vRAM [100]. These proof system work by defining a program in RAM assembly, either directly or by compiling down a higher level language such as C, and proving its correct execution by running it over a virtual CPU, i.e. a circuit whose satisfiability encodes the correct execution of that program. One of the main advantages of using a virtual CPU is that the proof system is now universal since we only need a single circuit modelling the CPU for proving correct execution of any program written in its assembly language.

Truly universal zk-SNARKs rely instead on updatable universal reference strings (first introduced by Groth et al. [59]), which can be used for all circuits up to some upper bound circuit size. Examples of universal proof systems are Libra [96], Plonk [47] and Marlin [31]. In particular, Libra is the first universal general-purpose proof system to have optimal prover time (i.e. linear in the circuit size) and succinct proof size and verification time.

3.1.4 Discarding the CRS model

Another solution to the problems caused by the need for a CRS is to remove the reliance of a proof system on it altogether. A transparent proof system relies instead on publicly available randomness or other tools, such as commitment schemes, to fix the circuit to be proved/verified. In 2017, Ames et al. [2] developed Ligero, one of the first transparent proof systems. The Ligero protocol is a zero-knowledge probabilistically-checkable proof (PCP) based on interleaved Reed-Solomon codes, which was obtained by applying an optimised version of the general transformation of Ishai et al. [65] to a variant of the protocol for secure multiparty computation of Ishai et al. [36]. In a parallel and independent work, Ben-Sasson et al. [15] constructed another post-quantum transparent proof system called zk-STARK (zero-knowledge scalable transparent arguments of knowledge) in the IOP model relying on techniques such as Fast Reed-Solomon IOP of Proximity (FRI) [13]. From these works, many other PCP-based proof systems leveraging collision resistant hash functions were developed, such as Aurora [11], Virgo [99] and Fractal [30]. Since these proof systems only rely on collision resistant hash functions, they are also (plausibly) post-quantum.

In 2018, Bünz et al. [27] introduced Bulletproofs, the first transparent general-purpose zero-knowledge proof systems where the prover's computation scales linearly with the size of the circuit. It is built on techniques developed by Bootle et al. [22] and relies on the elliptic curve discrete logarithm assumption. However, Bulletproofs is not an actual zk-SNARK due to the fact its verification time is not succinct, but scales linearly instead. Spartan [86] is the first transparent zk-SNARK where the prover complexity scales linearly and the verifier complexity sub-linearly with the size of the circuit. To achieve this, Spartan introduces a new public-coin succinct interactive argument of knowledge. This argument makes a black-box usage of a (sparse) polynomial commitment scheme in order to fix the statement that is being proved/verified. For this reason, its soundness holds under the same assumptions needed by the polynomial commitment scheme. For instance, if we use a polynomial commitment scheme that relies on the (EC) DLog assumption, then the resulting Spartan protocol will also be secure under the same assumption. Conversely, if we instead use a post-quantum polynomial commitment scheme, we obtain a post-quantum version of Spartan.

Regarding (plausibly) post-quantum proof systems, another possible approach is to use lattices. However, historically, lattice-based proof systems have always performed worse than those based on hash functions, especially when it came to the size of the generated proof. Labrador [18] is the first lattice-based general-purpose proof system to close the proof-size gap with CRHF-based proof systems and even improve over them.

3.1.5 Summary of proof systems

Table 3.1 summarises all the proof systems introduced in this section and highlights some of their features. In particular, we indicate the type of protocol they use, the type of trusted setup they need, their cryptographic assumption, the complexity of the prover and verifier execution time and the proof size. For protocols, SNARK stands for Succinct Non-interactive ARguments of Knowledge, SNARG stands for Succinct Non-interactive ARguments, ShNARK stands for Short Non-interactive ARguments of Knowledge, and STARK stands for Scalable Transparent ARguments of Knowledge. For cryptographic assumptions, KoE stands for Knowledge of Exponent, GGM stands for Generic Group Model, AGM stands for Algebraic Group Model, (EC) DLog stands for Discrete Logarithm assumption over Elliptic Curves, CRHF stands for Collision Resistant Hash Function and M-SIS stands for Module Short Integer Solution problem. Regarding the complexity, n indicates the number of non-linear constraints, d indicates the depth of the circuit, g indicates the width of th circuit and $m := d \log g$.

3.2 Backends

In this section we introduce various implementations of existing proof systems and analyses their features. In particular, we are interested in:

- Implemented proof system(s), which determines proof and verifier time and proof size.
- Supported constraint systems, which influence the compatibility with frontends.
- The field on which the constraint system is defined, which influences what signature schemes are natively supported by the proof system. In particular, for proof systems operating on elliptic curves, the constraint system is defined over $\mathbb{F}_q$, where q is the order of the elliptic curve. For proof systems operating over finite fields, the constraint system is defined over the same finite field as the proof system's.

Libsnark libsnark (available under the `MIT` license at [84]) provides C++ implementations of many general-purpose proof systems, including two zk-SNARKs for the R1CS constraint system, one based on vnTinyRAM and the other being an implementation of Groth16. Additionally, it provides gadget libraries for constructing (complex) R1CS statements by combining instances of modular gadget statements. The curves supported by libsnark are `bn128`, `alt_bn128` and the `edwards` curve.

Snarkjs snarkjs (available under the `GLP-3.0` license at [62]) is a JavaScript implementation of Plonk, Fflonk and Groth16 specifically designed for

Proof system	Protocol	Setup	Assumptions	Prover	Proof size	Verifier	Year
Pinocchio	zk-SNARK	Per-circuit	KoE	$\mathcal{O}(n \log n)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$	2013
Pantry	zk-SNARK	Per-circuit	KoE	$\mathcal{O}(n \log n)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$	2013
TinyRam	zk-SNARK	Universal	KoE	$\mathcal{O}(n \log n)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$	2013
vnTinyRam	zk-SNARK	Universal	KoE	$\mathcal{O}(n \log n)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$	2014
Geppetto	zk-SNARK	Per-circuit	KoE	$\mathcal{O}(n \log n)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$	2015
Buffet	zk-SNARK	Universal	KoE	$\mathcal{O}(n \log n)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$	2015
Groth16	zk-SNARK	Per-circuit	GGM	$\mathcal{O}(n \log n)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$	2016
Ligero	zk-SNARK	None	CRHF	$\mathcal{O}(n \log n)$	$\mathcal{O}(\sqrt{n})$	$\mathcal{O}(n)$	2017
zk-STARK	zk-STARK	None	CRHF	$\mathcal{O}(n \log^2 n)$	$\mathcal{O}(\log^2 n)$	$\mathcal{O}(\log^2 n)$	2018
Bulletproofs	zk-ShNARK	None	(EC) DLog	$\mathcal{O}(n)$	$\mathcal{O}(\log n)$	$\mathcal{O}(n)$	2018
vRam	zk-SNARG	Universal	KoE	$\mathcal{O}(n \log n)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$	2018
Libra	zk-SNARG	Universal	KoE	$\mathcal{O}(n)$	$\mathcal{O}(d \log n)$	$\mathcal{O}(d \log n)$	2019
Plonk	zk-SNARK	Universal	KoE	$\mathcal{O}(n \log n)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$	2019
Aurora	zk-SNARK	None	CRHF	$\mathcal{O}(n \log n)$	$\mathcal{O}(\log^2 n)$	$\mathcal{O}(n)$	2019
Fractal	zk-SNARK	None	CRHF	$\mathcal{O}(n \log n)$	$\mathcal{O}(\log^2 n)$	$\mathcal{O}(\log^2 n)$	2019
Virgo	zk-SNARK	None	CRHF	$\mathcal{O}(n \log n)$	$\mathcal{O}(d \log n)$	$\mathcal{O}(d \log n)$	2020
Marlin	zk-SNARK	Universal	KoE, AGM	$\mathcal{O}(n \log n)$	$\mathcal{O}(d \log n)$	$\mathcal{O}(n)$	2020
Spartan	zk-SNARK	None	(EC) Dlog	$\mathcal{O}(n)$	$\mathcal{O}(\sqrt{n})$	$\mathcal{O}(\sqrt{n})$	2020
Labrador	zk-SNARK	None	M-SIS	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(\log n)$	2022

Table 3.1: Comparison of existing general-purpose zero-knowledge proof systems.

Circom and supporting R1CS-based statements. Additionally, it also provides tools for performing the trusted setup needed by the proof systems using the Powers of Tau protocol [23], visualising the constraint system generated by Circom, and for computing the witness for the circuit. The curves supported by snarkjs support are `alt_bn128` and `bls12-381`.

RapidSnark rapidsnark (available under the `GLP-3.0` license at [61]) is a C++ implementation of Groth16 specifically designed for Circom and supporting R1CS-based statements. However, it does not provide tools for performing a trusted setup, thus requiring external tools such as, for example, snarkjs. rapidsnark only supports the `alt_bn128` curve.

Bulletproofs bulletproofs (available under the `MIT` license at [35]) is a Rust implementation of the Bulletproofs proof system operating on Ristretto [60] over Curve25519. It supports statements expressed using R1CS, however, this feature is, at the moment of writing, still under development and described as unstable. The only curve supported by bulletproofs is Curve25519.

LibSTARK libSTARK (available as open-source software at [14]) is a C++ implementation of zk-STARK. It supports statements expressed as algebraic intermediate representations (AIRs) or as permuted algebraic intermediate representations (PAIRs). libstark operates on the binary field $\mathbb{F}_{2^{64}}$.

Spartan spartan (available under the `MIT` license at [85]) is a rust implementation of the Spartan proof systems. It operates on ristretto255, a prime-order group abstraction atop Curve25519 (a high-speed elliptic curve). There also exists a (slower) version of it, which operates on secq256k1 [78], an elliptic curve whose order is the same as the prime of the finite field of secp256k1 (the Bitcoin curve), which means that this version of spartan natively supports secp256k1 signatures. spartan accepts statements expressed using R1CS.

Libiop libiop (available under the `MIT` license at [83]) is a C++ library providing implementations for Ligero, Aurora and Fractal. All three proof systems support R1CS statements over any smooth prime field and binary extension field.

LegoSNARK LegoSNARK [28] (available under the `Apache-2.0` license at [88]) implements a hybrid, modular approach for the construction of a "global" zero-knowledge proof system based on commit-and-prove zk-SNARKs (CP-SNARKs). It provides a toolbox of specialized and optimised *gadget SNARKs* for proving a variety of relations together with a framework which allows the linking of said gadgets in a lightweight manner to combine them into more complex proofs. The motivation behind this approach is that general-purpose SNARKs are less efficient than specialized ones when it comes to specific operations, since they may not be able to exploit problem-specific nuances to get more efficient solutions. Moreover,

computations tend to be heterogeneous, containing, for example, both arithmetic and boolean components. The modular approach to zk-SNARK design provided by LegoSNARK allows developers to use the most suited SNARK for each subroutine of a proof and then link them together, gaining the efficiency of specialized SNARKs and the flexibility of general-purpose SNARKs. Moreover, the modular approach also allows for a reduction in the complexity of the design of a proof system since, instead of needing to handle arbitrary computations, one can focus on more specific problems. Finally, LegoSNARK also provides a compiler capable of adding CP capabilities to a broad class of existing zk-SNARKs, enabling interoperability between them. This means that LegoSNARK can operate on any elliptic curve and finite field for which there exists a proof system, similarly, it can potentially work with any constraint systems. However, this approach entails also some downsides. In particular, this approach introduces the need for the prover to commit to each input before being able to prove the statement, this can introduce additional computational overhead and complexity in the system.

3.3 Frontends

In this section we introduce various available frontends for expressing zero knowledge statements and analyse their features. In particular, we are interested in ease of use, type of DSL (high or low-level), flexibility and which target constraint systems they implement.

Circom Circom [9] defines a (low-level) domain-specific programming-language (also called constrained-based language or hardware language) for designing arithmetic circuits and a compiler which translates these circuits to a R1CS representation. Circom can be complemented with zk-SNARK implementations (in particular, with snarkjs and rapidsnark) to generate and verify proofs from a set of rank-1 constraints.

Circom, enables developers to express statements by way of an arithmetic circuits. Together with the compiled R1CS instance, Circom alsog generates a program that can efficiently compute a valid assignment to all the wires in the circuit for a given input. The output files of Circom can be directly fed to either snarkjs or rapidsnark, two libraries by the same developers of Circom to automate the generation and the verification of zk-SNARK proofs.

One of the main features of the Circom programming languages is that it is modular, i.e. it supports the definition of (smaller) parametrised circuits (called templates) which can be instantiated and combined into larger, more complex circuits. Additionally, there are numerous publicly available libraries offering hundreds of pre-built circuits, and Circom itself already

provides many useful primitives such as comparators, hash functions (e.g. Poseidon [52], SHA-256) and signature schemes [64]. Moreover, the Circom DSL being a low-level language, allows developers to apply advanced optimisations to the arithmetic circuits.

However, Circom presents some shortcomings too. These are, in particular, its relatively harder initial learning curve, the impossibility for control flow in to depend on the input, and the only supported datatype being integers modulo p, where $\mathbb{F}_p$ is the finite field on which the constraint system is defined. By default, the elliptic curves supported by Circom are `bn128`, `bls12377`, `bls12381`, `goldilocks`, `grumpkin`, `pallas`, `secq256k1` and `vesta`, however it is quite simple for developers to implement additional curves if needed.

Snarky Snarky [75] is an OCaml-like high-level DSL based on zk-SNARKs for verifiable computation developed by o1Labs[1]. In Snarky, a verifiable computation is defined as a "normal" computation with two additional augments:

1. Ability to pause the execution, ask its environment to provide it with a value, and then resume execution using that value.
2. Ability to assert that a constraint holds among some values, terminating with an exception if the constraint does not hold.

Snarky lets developers define zero-knowledge statements by writing a "normal" OCaml program, which is then translated to a set of R1CS constraints to be used by a zk-SNARK library for proof generation and verification.

The main downside of Snarky, which is common to almos all high-level DSLs, is that it is possible for the resulting circuits to be suboptimal due to the compiler missing some optimisations that a developer could implement when defining statements in a low-level language.

xJsnark xJsnark [68] is a programming framework for verifiable computation based on zk-SNARKs introducing user and compiler friendly features. These features allow developers to write programs in a Java-like language and subsequently enable the backend to extract additional information from the code which can be used to perform optimisations at compile-time and thus converting the user-provided program into a compact and optimised circuit. In particular, xJsnark promises to overcome both the shortcomings of low- and high-level DSLs, i.e. offering ease of use to developers through a high-level DSL while still implementing various circuit optimisation techniques to minimise circuit size to be competitive with low-level DSLs. xJsnark is developed as a Java extension on top of Jetbrains MPS V 3.3.5.

[1] https://www.o1labs.org/

One of the techniques used to achieve minimal circuits is the introduction of circuit-friendly building blocks for frequent operations such as memory accesses and integer arithmetic. Additionally, the compiler also makes more global arithmetic optimisations, and implements multi-variate polynomial minimisation techniques to minimise the generated circuit further. Finally, circuits are compiled into an R1CS representation.

RISC Zero RISC Zero [26] is a zero-knowledge general-purpose computing platform based on zk-STARK and the RISC-V microarchitecture. It can prove verifiable execution for programs written in arbitrary languages, as long as there exists a compiler for that language targeting RISC-V. At the time of writing SDK support exists for Rust, C, and C++.

RISC Zero takes code written in one of the supported languages (called guest program), and compiles it down to RISC-V assembly, which is then run inside a zero-knowledge virtual machine (zkVM) proving the correct execution of the program. The output of the zkVM is a receipt verifying the guest program's execution and the program's output(s). The receipt can take many forms:

1. A vector of zk-STARK proofs, called composite receipt.
2. A single zk-STARK proof, called succinct receipt.
3. A Groth16 receipt generated by verifying a succinct receipt using Groth16.
4. A fake receipt containing no proof.

RISC Zero's design makes it a very flexible and versatile system with the capability of potentially supporting any programming language and with the ability to be deployed on any machine capable of running VMs. However, reliance on a VM also presents some downsides, in particular, it is resource intensive (for local proof generation, the authors suggest a machine with at least 16 GB of RAM) and likely not suited for mobile devices.

Ligetron Ligetron [94] enables verifiable computation of programs written in C/C++ while being lightweight, space efficient, and, according to the developers, able to prove statements with billions of constraints on commodity hardware.

Instead of using R1CS constraints to represent the statement to prove, which do not retain any underlying semantic structure of the original program, Ligetron utilises WASM. In fact, WASM is sufficiently high-level to retain the semantic meaning of the original program while still being adequately low-level to connect to the prover/verifier. This additional information facilitates the compiler in optimising the circuit for space efficiency and proving time. Moreover, WASM also retains information about scoping and control structures. This intermediate representation is then expanded into a circuit

on the fly, at runtime, by the prover/verifier. The developers also provide a backend based on a modified version the Ligero proof system that is able to leverage the semantics of WASM. Since Ligetron uses WASM, it can also run efficiently on any major browser, including mobile ones, while being able to scale to very large circuit sizes.

However, Ligetron only implements a subset of WASM, moreover, only static control flow, i.e. control flow that does not depend on the witness, is supported. At the time of writing, there are no open source implementations of Ligetron.

Cairo Cairo [90] is a programming language designed for a turing complete zk-STARK-friendly virtual CPU. This CPU implements an efficient and practical von Neumann architecture that can be used with the zk-STARK proof system to enable verifiable computation. In practice, Cairo provides a layer of abstraction around zk-STARK that simplifies the way a computation is described. While it was designed for zk-STARK systems, it can also be adapted for other finite field-based systems such as zk-SNARKs.

Cairo uses RAP (Randomised AIR with Preprocessing) intermediate representation to model a virtual CPU, which is independent of the computation being proved/verified. This RAP describes the loop of fetching an instruction from memory, executing that instruction, and proceeding to the following instruction. The correct execution of a Cairo program is proved by running it on the virtual CPU whose computational integrity is ensured using zk-STARK.

While the CPU approach does present a computational overhead w.r.t. proof systems that directly compile statements into set of constraints, due to the need to decode the instructions and use memory operations, it also presents some advantages, mainly:

- Small, fixed size constraint set, which improves verification costs.
- There is a single RAP, that only needs to be implemented once (rather than per application), simplifying the process of auditing the proof system, i.e. when presenting a new application, the only thing that needs auditing is its code.

A significant downside to Cairo is that, there are no easy way for specifying which of the inputs of a computation are private. In fact Cairo appears to be a tool more suited for ensuring computational integrity rather than for proving zero-knowledge statements.

Zilch Zilch [92] is a zero-knowledge framework for interactive zero-knowledge proofs based on zk-STARK and zk-SNARKs. It enables verifiable computation for arbitrary programs written in the ZeroJava programming language, which is a subset of Java designed explicitly for zero-knowledge arguments.

Zilch implements a compiler which takes ZeroJava programs and translates them into zMIPS assembly. zMIPS is Zilch's extension to the MIPS ISA, and it is specifically designed to support zero-knowledge proofs. The assembly code is passed to the backend, together with input and witness, which translates it first into an arithmetic circuit and then into an arithmetic intermediate representation (AIR) describing the circuit. These constraints are then used by the prover/verifier to generate/validate a proof a computational integrity. For proof generation and verification, Zilch uses the zk-STARK proof system.

Some of the downsides of Zilch are:

- The extremely high cost of calling methods from external classes, which triple the proving time w.r.t. when inlining the function being called. This makes modular designs detrimental to the performance.
- Limiting the declaration of variables to the beginning of a method and nowhere else after that.
- Poor documentation.

3.4 Evaluation of Existing Frameworks

In this section we present an evaluation and comparison of the performance of some of the backends that we presented in Section 3.2. We decide to only focus on proof systems supporting R1CS since we are planning to use Circom as our frontend of choice. Due to time constraints, we can only run measurements on a small sample of proof systems, we try to select them, in such a way as to cover as many approaches to zk-SNARKs as possible, Table 3.2 shows our choice of proof systems. We choose rapidsnark and snarkjs since they were developed appositely for Circom and implement two very popular proof systems in Groth16 and Plonk, this way we also have both a universal proof system and one which requires a per-circuit trusted setup. For transparent systems, we opted for spartan, both the original version and the one based on secq256k1, and libiop to represent both DLog- and CRHF-based transparent proof systems.

In order to compare proof systems to each other, we measure prover time, verifier time and proof size scaling over the number of non-linear constraints in the circuits. For this purpose, we used synthetic circuits generated using Circom, i.e. circuits that do not hold any particular semantic meaning but are simply a series of non-linear constraints chained together. This approach makes it easier to precisely control the number of non-linear constraints for each measurement but has the downside of generating very structured circuits, which could slightly skew the results in favor of proof systems that can take advantage of such structure. However, from our experiments, we con-

Proof system	Transparent	Universal	Post-quantum
rapidsnark (Groth16)	No	No	No
snarkjs (Groth16)	No	Yes	No
snarkjs (PLONK)	No	Yes	No
Spartan (curve25519)	Yes	Yes	No
Spartan (secq256k1)	Yes	Yes	No
libiop (Ligero)	Yes	Yes	Yes
libiop (Aurora)	Yes	Yes	Yes

Table 3.2: Choice of backends

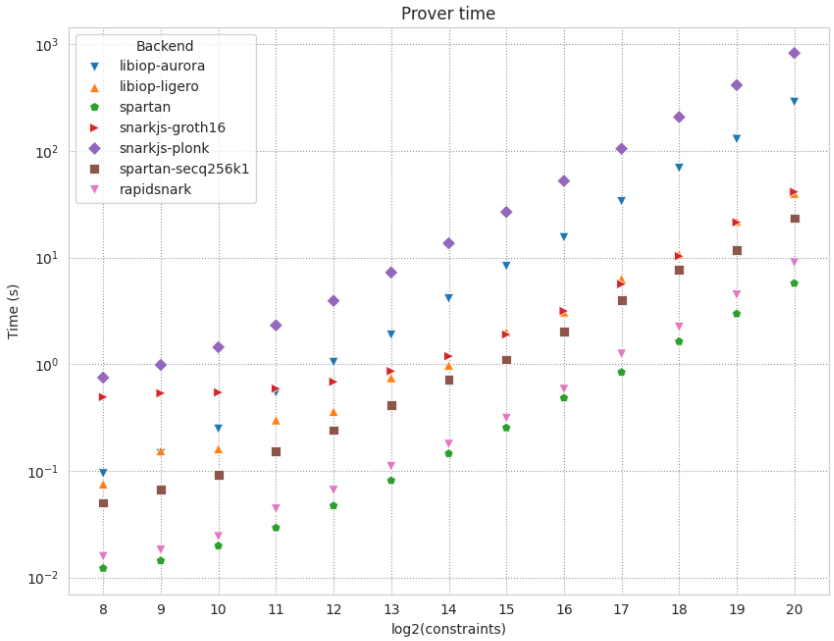

Figure 3.1: Prover execution time scaling over the number of non-linear constraints

clude that this phenomenon, if at all present, has a negligible impact on the measurements. The measurements were made on a zBook from HP with an i7-1255U and 32 GB of memory. Figures 3.1, 3.2 and 3.3 show the results for the measurements of prover time, verifier time and proof size, respectively. Each data-point in the graph corresponds to a single measurement.

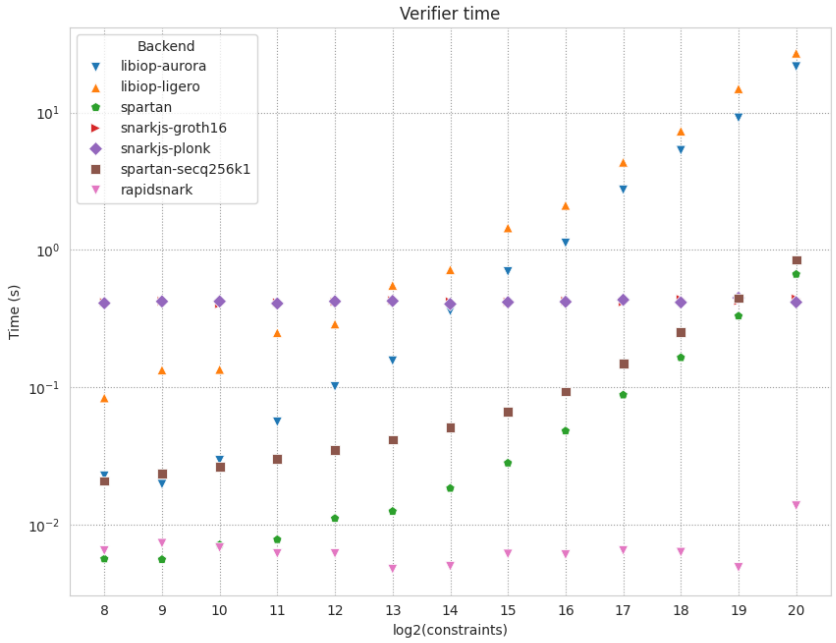

Figure 3.2: Verifier execution time scaling over the number of non-linear constraints

3.4.1 Prover execution time

Figure 3.1 shows the results of the measurements of the prover's execution time scaling over the number of non-linear constraints (see Appendix A for numerical results). On the x-axis we have the logarithm base 2 of the number of non-linear constraints in the circuit, and on the y-axis we have the execution time in seconds, also represented with a logarithmic scale. We notice that RAPIDSNARK and the base version of SPARTAN are by far the fastest, with an execution time, for 2^8 non-linear constraints of 0.0157s and 0.0122s respectively and of 8.95s and 5.72s respectively for 2^{20} non-linear constraints. The version of SPARTAN based on the secq256k1 curve (SPARTAN-SECQ256K1) is a distant third, with its prover time being around 4x slower than the other two at 0.0501s for 2^8 non-linear constraints and 23.3s for 2^{20} non-linear constraints. LIGERO and the JavaScript implementation of GROTH16 provided by SNARKJS also have similar performances to SPARTAN-SECQ256K1, albeit slightly worse. Finally, AURORA and PLONK have the worst performances, with PLONK being last by a large margin with an execution time of 0.746s for 2^8 non-linear constraints and 824s for 2^{20} non-linear constraints. It is important to stress that we should minimise the prover's execution time as much as possible because, when deployed, this will be running on a user's

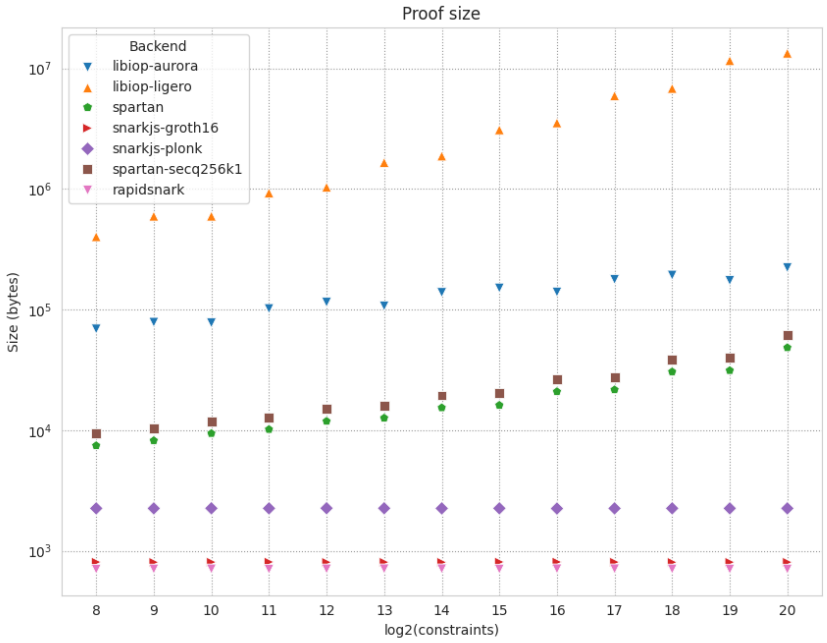

Figure 3.3: Proof size scaling over the number of non-linear constraints

mobile device, which has a limited computational power.

3.4.2 Verifier execution time

Figure 3.2 shows the results of the measurements of the verifier's execution time scaling over the number of non-linear constraints (see Appendix A for numerical results). On the x-axis we have the logarithm base 2 of the number of non-linear constraints in the circuit, and on the y-axis we have the execution time in seconds, also represented with a logarithmic scale. First, we observe that, indeed, both implementations of Groth16 (rapidsnark and snarkjs) and also Plonk have a constant verification time, ignoring the measurement noise resulting from the very low running time, of around 0.007s, 0.4s and 0.4s respectively, with the C++ version of Groth16 being, of course, faster than the JavaScript one. Please note that the data-points for `snarkjs-groth16` are barely visible in the plot because they are covered by `snarkjs-plonk`. Both versions of Spartan also have reasonably low execution time, which ranges between $\sim$ 0.01s for 2^8 non-linear constraints and $\sim$ 0.8s for 2^{20} non-linear constraints. Meanwhile, Ligero and Aurora are the slowest ones, especially when the number of non-linear constraints goes over 2^{15} (which is very common, as we are going to see in Section 4, for our

use-case). In general, we can assume that the verifier would have access to more computational resources than the prover. For this reason, minimising verification time is less of a priority for us since it is not the bottleneck of our implementation, in particular, when considering that for both Groth16 and Spartan the verification time is always below one second.

3.4.3 Generated proof size

Figure 3.3 shows the results of the measurements of the size of the generated proofs by scaling over the number of non-linear constraints (see Appendix A for numerical results). On the x-axis we have the logarithm base 2 of the number of non-linear constraints in the circuity while on the y-axis we have the size of the proof in bytes represented with a logarithmic scale. We still observer that both implementations of Groth16 (rapidsnark and snarkjs) and also Plonk have the lowest proof size being $\sim$ 700B and $\sim$ 2250B respectively, this time with no difference between the C++ and JavaScript versions of Groth16. Again, both versions of Spartan are middle-of-the-pack with proof sizes ranging from 7 456B for 2^8 non-linear constraints and 61 557B for 2^{20} non-linear constraints. Meanwhile, Ligero and Aurora produce the largest proofs, with Ligero going up to a proof of $\sim$ 13.3MB for 2^{20} non-linear constraints. In general, we should aim for backends that produce reasonably small proofs to avoid exchanging several MBs of data between prover and verifier, especially considering, again, that the goal is deployment on mobile devices.

3.4.4 Discussing the measurements

By considering all three plots, it seems clear then that the final choice of backend for our implementation is going to be either rapidsnark or spartan for minimising the prover time, or spartan-secq256k1 for achieving still a reasonable prover time, circa 4x slower than base spartan and rapidsnark but faster than any other proof system, with the added benefit of having access to a more secure, natively supported signature scheme in secp256k1. In general, we can expect execution time on smartphone to be 2-3x worse than what we measured here, and we must also consider this fact when choosing a framework.

An interesting note are also the differences between Ligero and Aurora. We can notice, in fact, that the first tries to prioritise a faster prover time at the expense of producing proofs that are larger and slower to verify while the latter tries to minimise proof size and verification time at the cost of a slower prover time.

Chapter 4

Building Blocks for Arithmetic Circuits

In this chapter we introduce some of the most commonly used circuit such as comparators (Section 4.1), hash functions (Sections 4.2 and 4.3), dynamic array lookups (Section4.4), and signature schemes (Sections 4.5 and 4.6). A particular focus is placed on their cost, in number of non-linear constraints, and any additional details one must consider when using them. Table 4.1 summarises the costs of the various building blocks.

Operation	**# Non-linear constraints**
`IsZero`	1
`IsEqual`	2
`LessThan(b)`	`b`
`GreaterThan(b)`	`b`
`Poseidon`	213 - 609
`SHA256`	29 380
`Array Lookup(N)`	`4N`
`Secp256k1Fast`	$\sim$ 4 000
`Secp256k1Full`	$\sim$ 18 000
`NISTP256Fast`	$\sim$ 200 000

Table 4.1: Overview of basic circuits and their costs, `b` indicates the maximum bit-size of the circuit's inputs while `N` indicates the size of the input array.

4.1 Comparators

With comparators we refer to a set of circuits designed to determine the relation between to integers, these are `IsZero`, `IsEqual`, `LessThan` and `GreaterThan`. In general, these circuits are cheap costing at most a couple of hundreds of constraints, table 4.2 summarises their costs in more detail.

IsZero This circuit tests whether the input signal `in` is zero, and outputs 1 if that is the case and 0 otherwise. This may seem straightforward at first, but it hides some subtleties. The implementation first uses a non-constraining intermediate signal `inv` to compute the inverse of the input, if it exists, if not, `inv` is assigned the value 0. The output of the circuit is `-in * inv + 1` which has value 1 when `in` is zero and 1 in all other cases. Snippet B.1 shows the Circom code for the circuit.

IsEqual Testing for equality of two signal inputs `in0` and `in1` is implemented by computing `in0 - in1` and checking whether the resulting value is 0 by using an instance of the `IsZero` circuit, whose output is used as the final output of `IsEqual`.

LessThan, GreaterThan It is not possible to directly express less-than or greater-than operations using modular arithmetic. Instead, the implementation converts the two input signals into a bit-representation of the values they carry and then performs a bit-wise comparison to determine which of the two values is bigger/smaller. This results in the cost of the circuit being proportional to the bit-size of the inputs with one non-linear constraint per bit. The two circuits `LessThan(b)` and `GreaterThan(b)` are parametrised where `b` indicates the bit-size of the inputs.

Circuit	# Non-linear constraints
`IsZero`	1
`IsEqual`	2
`LessThan(b)`	`b`
`GreaterThan(b)`	`b`

Table 4.2: Cost of comparators, `b` indicates the maximum bit-size of the circuit's inputs

4.2 Poseidon Hash

The Poseidon hash function is a circuit friendly hash function developed by Grassi et al. [52]. With 'circuit friendly primitives' we mean algorithms that use only (or mostly) modular arithmetic operations. The Poseidon hash

function consists of a series of permutation rounds based on the Hades design strategy [51]. Each round requires the use of a set of constants which are uniquely defined by the concrete instantiation of Poseidon. The instantiation is determined by the choice security parameters, the finite field on which the hash function operates, and the size of the input (in number of field elements). These constants must be precomputed and passed as input to the circuit, this means that, for each Poseidon instantiation, in particular for each input size that we want to support, we need to precompute a set of constants. Since the size of each set of constants grows quite quickly with the size of the input, we decided to support a maximum input size of 16 field elements. In case this input size is not enough, we simply nest multiple instances of the circuit implementing the Poseidon hash. Table 4.3 shows the cost in non-linear constraints of the Poseidon hash function for some of the supported input sizes. Please, note that the round constants are specific for a certain finite field, this means that they need to be recomputed if one wants to use different proof system implemented over a different finite field.

Regarding the security of Poseidon, in particular in comparison to SHA256, we point to Section 6.2. In a nutshell, both Poseidon and SHA256 guarantee 128 bits of security, however, we must be aware that Poseidon has received less scrutiny and testing than SHA256, so it is possible that future work will show it to be insecure.

Input size	**# Non-linear constraints**
1	213
2	240
4	297
6	354
8	402
10	459
12	504
14	537
16	609

Table 4.3: Cost of the Poseidon hash function in number of non-linear constraints w.r.t. the input size in number of 256-bit integers

4.3 SHA256

The SHA256 hash function is not friendly to arithmetic circuits due to its heavy reliance on boolean operations such as bitwise `and`-s, `xor`-s and shifts. For example, computing SHA256 for a 256-bit input results in a circuit size of 29 380 non-linear constraints, almost 138 times larger than computing the Poseidon hash for an input of the same size. This nicely shows how critical it is to choose, when possible, circuit friendly primitives when building circuits in order to minimise their cost.

4.4 Dynamic Array Lookup

Circom has no support for dynamic array lookup. To work around this limitation, we create a circuit which, given as input signals an array `arr` of size N and a target index `ind`, performs the following actions:

1. Multiplies each element of `arr` by 1, if the element's index in the array is equal to `ind`, and by 0 otherwise, and stores the results in a different array `arr'`.
2. Computes the sum of all elements of `arr'` yielding the value contained in the target entry of `arr`.

This results in a circuit size of $4N$ non-linear constraints, snippet B.2 shows its Circom implementation. Note that we are accessing an element of an array in zero-knowledge, this means that if the target index is a private input, the verifier doesn't learn which element of the array was accessed, even if the array itself is publicly known.

4.5 secp256k1 Signature Verification

The Bitcoin curve, also called secp256k1, is an elliptic curve used in Bitcoin's public key cryptography and is standardised by the *Standards for Efficient Cryptography (SEC)* [69]. Due to our choice of backend (see Section 5.1), the finite field of the constraint system is the same as finite field of secp256k1. As a consequence of this choice, our proof system natively supports secp256k1 signatures, i.e. all modular operations needed to perform elliptic-curve-arithmetic are performed "for free" by the backend's proof system. In fact, an arithmetic circuit implementing secp256k1 ECDSA signature verification using Algorithm 2, with natively supported modular operations for elliptic curve arithmetic, has a relatively small size of $\sim$ 18 000 non-linear constraints.

Note that we are verifying a signature in zero-knowledge, this means that, for a public key known by the verifier, we can prove to be in possession

of a valid signature (w.r.t. the known public key) of some message without disclosing neither the value of the signature, nor of the plaintext message, to the verifier. This is important because it means that we are able to achieve unlinkability using a non-privacy-preserving signature scheme, and without the need for BBS+, because we are able to verifier signatures without revealing any information, such as the value of the signature, which would uniquely identify a holder.

We can further optimise this circuit by performing as many expensive operations as possible, such as elliptic-curve scalar multiplications, outside the circuit. In practice, we move the computation of $u_1 \times G$ to be computed off-circuit. For this purpose we define two new points

$$T := r^{-1} \times R \text{ and } U := -\left(r^{-1}m \times G\right)$$

where r, R, m and G are defined as in algorithms 1 and 2. T and U can be computed off-circuit and be passed to it as public inputs. More precisely, the verifier has public knowledge of T, U, r and R, while Q and s remain secret. Then, the prover can demonstrate that they were able to compute a valid signature for m by proving that knowledge of a witness (s, Q) such that

$$s \times T + U = Q$$

where Q is the public key.

This approach results in a much smaller circuit for signature verification with a size of only $\sim 4\,000$ non-linear constraints. It is important to note, however, that this approach, while hiding the value of the public key, makes publicly known the message being signed and part of its signature. As such, it should be used only in applications where this doesn't lead to a loss of privacy, such as, for instance, proof of the possession (or knowledge) of a private key.

4.6 Optimised NIST P-256 Signature Verification

One of the design requirements of the Swiss e-ID is support for hardware binding. Unfortunately, there is no hardware support for secp256k1, therefore, at least for the time being, we need to rely on NIST standard curves, in particular NIST P-256, for implementing hardware binding.

The first obstacle to implementing NIST P-256 signature verification in Circom, is the difference between the finite field of the constraint systems and the finite field over which NIST P-256 is defined. This means that we need to implement from scratch all modular arithmetic and curve operations (point

addition and scalar multiplication) needed for verifying an ECDSA signature. Moreover, we need to implement big-integers to avoid overflow when computing modular operations.

An additional challenge comes from the fact that Circom does not support input-based control flow, this makes implementing elliptic curve operations, i.e. point addition and point scalar multiplication, extremely convoluted and constraints-wise expensive, in particular, scalar multiplication (implemented with a "double and add" loop) alone needs 1.8 millions non-linear constraints (we use the NIST P-256 implementation by PSE [44] as a reference for the costs of the various operations). This means that verifying an ECDSA signature using algorithm 2, which consists of 2 scalar multiplications and one point addition, would theoretically need almost 3.7 millions constraints. In practice though, an actual implementation, such as the one by PSE [44], requires "only" 1.9 millions non-linear constraints. The reason for such a low number of constraints is that they precompute a large sample of multiples of G and use them to speed up the computation of $x \times G$. Still, this is far too expensive.

Fortunately, for the purposes of hardware binding, we don't care whether we disclose the plaintext or its signature, rather, we want to keep the public key of the holder secret in order to avoid linkability. Therefore, we can apply the same optimisation that we used with secp256k1 and move compute $u_1 \times G$ off-circuit. Still, the number of constraints is again too high with 1.8 millions non-linear constraints.

To further reduce the number of non-linear constraints, we apply to T the same trick that is used to accelerate the computation of the multiples of G, that is, we precompute a bunch of multiples of T and we pass them to the circuit as public inputs. More precisely, for a fixed stride t we compute

$$(2^{t \cdot i} \cdot j) \times T,\ i \in \{0, \ldots, 31\},\ j \in \{0, \ldots, 255\}.$$

According to the implementation by PSE, a stride of $t = 8$ is the most efficient. This final optimisation lowers the number of non-linear constraints to 212 093, this is almost a 9x improvement over the previous version and makes it feasible to verify ECDSA signatures in zero-knowledge in a reasonable time, although one should still prioritise using secp256k1 signatures where possible and limit the use of NIST-P256 to cases where it is strictly necessary such as with hardware binding.

Chapter 5

Implementing Zero-Knowledge Proof-Based Verifiable Credentials

In this chapter we provide an overview of our proof-of-concept implementation of a general-purpose zero-knowledge proof-based verifiable credential system. We introduce the zero-knowledge frameworks that we use in our implementation (Section 5.1). We then extended the format of the Swiss e-ID verifiable credential with all the additions and design considerations necessary to support zero-knowledge proofs (Section 5.2). Afterwards, we implement the most important functionalities, such as verifying the integrity and validity of a credential, preventing cheating from the part of the holder, credential linking, and hardware binding (Sections 5.4, 5.3, 5.5, and 5.6). Finally, we present the architecture of our proof-of-concept implementation and how to use it (Sections 5.7 and 5.8), and we introduce some concrete example of its possible applications (Section 5.9).

The codebase of our implementation can be found here, see the `README.md` file for instructions on how to run the code locally. The folder `scripts` contains samples end-to-end demonstrations of the workflow of the system, including credential issuance, circuit compilation, input and witness computation, and proof generation and verification. The folder `circuits` contains various basic circuit-component that can be used to express more complex statements, while the `proofs` folder contains actual concrete examples of statements that can be proved/verified with our system. The codebase is written in a mix of Java (for the issuer), Rust (for the SPARTAN proof system), and Python (for helper scripts). Additionally, statements are expressed in the CIRCOM language.

5.1 Framework Selection

In this section we present the frameworks, both frontend and backend, that we use for our proof-of-concept implementation additionally providing the motivations for our choices.

Circom The main reasons for choosing to use Circom as the frontend [63] are its widespread use, and the existence of many, publicly available, circuit implementations of useful primitives such as NIST P-256, SHA256, regular expression, and parsers. Additionally, it provides high flexibility in the choice of the target finite field of the constraint system which makes it potentially compatible with any backend supporting R1CS constraints. Finally, even though Circom is a low-level language, and thus has a potentially harder initial learning curve, it allows expressing statements in a modular fashion, by simply combining already existing building blocks, process which poses very little challenge, even for somebody who's not familiar with the language.

Spartan (secq256k1) Spartan is our backend of choice mainly due to three reasons: it has a linear-time prover, it is transparent, and it produces relatively small proofs. We prefer to go with a transparent proof system due to the unreasonably large size that CRSs can have when dealing with large circuits (up to tens of megabytes in size). More specifically, we opted for the secq256k1-based version of Spartan [78] because, even though it is slower than the original Spartan implementation, it is the only proof system whose natively supported signature scheme (secp256k1) sees real-world usage and has been shown to be secure by its widespread adoption and lack of vulnerabilities, so far. We deem the added security to be worth the loss in performance, which is roughly 4x worse for prover time. Additionally, Spartan can be easily converted into a post-quantum proof system by changing the current polynomial commitment scheme based on the EC Dlog assumption into a post-quantum polynomial commitment scheme such as Ligero's polynomial commitment [32], at the cost of the prover time scaling quasilinearly in the number of non-linear constraints, which is a nice feature to have when trying to future-proof our system.

5.2 Credentials for Zero-Knowledge Proofs

In this section we present and discuss all the additions and extensions to the format of the Swiss e-ID verifiable credentials necessary to integrate them in a zk-SNARK-based ecosystem. We first describe how we encode the claims to be compatible with arithmetic circuits (Section 5.2.1), then, we explain how we sign credentials in a way which allows to efficiently verify their integrity (Section 5.2.2). Finally, we introduce a set of helper claims that we add to the verifiable credentials ((Section 5.2.3)), and discuss how

to, and why we need to, define a precise schema for each credential type (Section 5.2.4).

5.2.1 Encoding claims

Since the only data-type that Circom supports are integers in a finite field, we must encode all claims into field elements. In particular, for our implementation, we are dealing with 256-bit integers over the finite field of secp256k1. In the following paragraphs we describe our strategies for encoding some of the data-types that can appear in a credential.

Integers No encoding is necessary as long as they have positive values. In case of negative integers, a common strategy is to sum them with a fixed value s such that the result is positive. If we choose to do so, the value of s must be either made public by the issuer or be added to the credential as an additional claim. Of course, one must also pay attention that the original value of the claim doesn't exceed the order of the finite field, if that were to happen, one can use big-integers to avoid overflowing.

Dates Dates can be easily encoded using UNIX timestamps. The only problem arises with dates before 1970-01-01 which have negative-valued timestamps. Again, we can solve this problem summing them with a fixed value d such that the result is positive. In our implementation, we chose to add, by default, a value equivalent in seconds to 100 years to all timestamps. This should be enough to avoid negative values for most use-cases. Of course the offset can be easily modified if need be.

Strings To encode strings, we first represent them as an array of bytes using `utf-8` encoding. Since we are working with 256-bit integers, we divide the resulting array into chunks of 32 bytes each, and we interpret each chunk as a 256-bit array. The choice of endianness makes no difference as long as it is consistent among all parties. The final encoding is an array of field elements representing the original string in a 1-to-1 mapping. To hide the length of the string, and to support strings with dynamic, although upper bounded, sizes, we pad the array up to some fixed, pre-defined size.

Lists and Objects Encoding of Json lists and object is straightforward and involves appling recursively the previous rules. For simplicity of implementation we choose to not support lists and objects and instead we only deal flattened SD-JWTs.

5.2.2 Signing the credential

When thinking about possible solutions for signing a zk-SNARK-compatible SD-JWT credentials, one must take into consideration that we need to be able to interact with the signature using an arithmetic circuit. In particular, we

need to be able to efficiently reconstruct the plaintext message starting from the credential's claims in order to prove integrity of the credential. While it is possible to parse an SD-JWT using an arithmetic circuit [79], this is also extremely expensive (more than 1.2 millions non-linear constraints for a 1 KB Json file) and impractical (recall, we cannot use input-dependent control flow). Moreover, such a solution creates the need to include in the credential the encoded version of each of its claims, which more than doubles its size, while, ideally, we would like to modify the credential as little as possible as to make integrating zk-SNARKs into already existing implementations relatively simple.

A more practical solution is to introduce a new claim in the verifiable credential, which aggregates all other (encoded) claims and have the issuer sign it (in addition to also signing the SD-JWT). This approach makes reconstructing the plaintext message within a circuit simpler, since we remove the need for parsing an SD-JWT by replacing it with an object with a well-known, fixed, circuit-friendly structure. Moreover, this approach does not bloat the resulting SD-JWT since the encoded claims can re-computed on-the-fly by the prover and passed as private inputs to the circuit.

Another important consideration is the choice of cryptographic primitives. For instance, while SHA256 is a very sturdy, battle-tested hash function, it is also extremely unfriendly to arithmetic circuits (see Section 4.3) and thus prohibitively expensive to use. A better choice would be to use a circuit friendly hash function such as the Poseidon [52], however one must keep in mind that Poseidon has not received as much scrutiny as SHA256 so, even though it should supposedly offer 128 bits of security, it could be affected by some unknown vulnerability. Same goes for signatures, NIST-P256 would be the optimal choice, especially since it is also the signature scheme of choice of the Swiss e-ID project, but, again, it is too expensive (see Sections 4.5 and 4.6). Instead, we should use whatever signature scheme that is natively supported by the proof system, which, in this case, is secp256k1.

For our implementation, the new aggregator claim is a 16-ary Merkle-tree where each leaf l_i is defined as

$$l_i := H_{Pos}(H_{Pos}(cn_i), cv_i)$$

where H_{Pos} denotes the Poseidon hash function, cn_i is the name of the i-th claim (e.g. `"vct"`), and cv_i is the encoding of the value of the i-th claim. We choose to work with the hashes of the names of the claims in order to only deal with constant size values. The reason for using a 16-ary tree is dictated by the fact that, for the same number of claims, recomputing its root requires fewer constraints than when using a binary tree, in fact reconstructing a 256-claims 16-ary trees requires $\sim 10\,000$ non-linear constraints while reconstructing a 256-claims binary tree requires $\sim 20\,000$ non-linear constraints.

Finally, the signature of the zero-knowledge credential is defined as

$$(zk_r, zk_s) := ECDSA_{Sign}^{secp256k1}(sk, zk_{root})$$

where sk is the private key of the issuer and zk_{root} is the root of the Merkle-tree aggregating the claims of the credential being issued.

5.2.3 Adding helper claims

Given what we previously discussed, we need to add some helper claims to the final SD-JWT verifiable credential (show in table 5.1). In particular, we add the root of the Merkle-tree resulting from the aggregation of the claims and its signature. We also include the value that is added to all timestamps to avoid negative values (`date_offset`), and a per-subject unique string, which we use to link together different credentials issued to the same subject (`cred_bind`, see Section 5.5). Listing 5.1, shows a concrete example of a SD-JWT credential which includes our additions.

Claim	SD-JWT	Description
`zk_root`	MUST	The root of the Merkle-tree of the credential's claims
`zk_r`	MUST	First component of the secp256k1 signature of `zk_root`
`zk_s`	MUST	Second component of the secp256k1 signature of `zk_root`
`cred_bind`	MUST	Value used for linking credentials to each other, unique for each subject
`date_offset`	MUST	Offset added to all timestamps to avoid negative values

Table 5.1: Additional helper claims added to the SD-JWT format

```
{
    "_sd": [
        "-XOkB3ctxrW9B83zkDmKV9hEkGdJT4YMTULyiSekI5o",
        "0bgjSAZek2hv_RAnBsg_h8beYtBXzu7mspIUYH1Xu4M",
        "6-OGPwfAw8OJJJfGwD-IPTA5MHdfz0y8dBnoUVYI-xI",
        "ZUl9ASTM-MCkQ47YlgmaHPvBfEm33OpKWuigWQCeQZc",
        "bZddAPKrGBjvnpPI5zEZeQN3h50FYdp7us1EnzGJmvk",
        "gih0888L6wri-_RWPJtp2E5colQUYJA2RReMIpAd_ts",
        "o4N2HerrK0e4O631HrrZIZOZyZ2tg0vjpjXf9L8F2EE",
        "tRV1AJAmPex0Ikjo81dtK1VUlbNUm7Gu0Be8NaJiiMM",
```

```
        "vEP3p8ASztOhHgaSR2nolPpFZG1VhAYe3F5fSa4RBYY",
        "vTIDfLXPS9UhHkn402NsQ_ap0VDcoiKpp4YgxZgZRro",
        "ydGgzbR2I2ZQU5Tt_ogvGSNJBL7v9trUzuEWtqCWMUA"
    ],
    "_sd_alg": "sha-256",
    "vct": "https://credentials.example.com/identity_credential",
    "iss": "did:example:123456789",
    "nbf": 1740441600,
    "exp": 2055888000,
    "iat": 1741694987,
    "zk_root": "10313080...25007824",
    "zk_s": "37416030...84165969",
    "zk_r": "76342475...21227156",
    "cred_bind": "bf8332c5...59d768c2",
    "date_offset": 3200000000,
    }
```

Listing 5.1: Example of a zk-extended Swiss e-ID SD-JWT credential

5.2.4 Credential Schemas

For each type of credential that we wish to support, we need a fixed schema defining:

1. The type of each claim (e.g. number, string, date, etc.)
2. An absolute order among the claims.

This ensures both that all claims are encoded correctly, according to their types, and that the order of the leaves of the Merkle-tree is known to all, such that anybody can correctly reconstruct it. Credential schemas are defined in the `issuer_metadata.json`[1] file. An example of a credential schema is show in snippet 5.2

```
{
    "foo": {
        "value_type": "number"
    },
    "bar": {
        "value_type": "string"
    },
    "baz": {
        "value_type": "date"
    },
    "order": ["foo", "bar", "baz"]
}
```

Listing 5.2: Example of a credential schema

Additionally, all claims must have a value associated to them (i.e. all claims are mandatory), a default value must be defined for claims with no associated value. One must pay close attention when defining default values, in

[1] `src/issuer/src/main/resources/issuer_metadata.json`

particular one must understand what is its semantic meaning and whether that could potentially lead to statements that should evaluate to false being proved as true, or vice versa. For instance, the claim `nbf` defines a timestamp before which the credential is not valid, if we choose 0 to be its default value, than we could end up with credentials that are considered valid even before they were issued. This may or may not constitute a problem, depending on the application, still, one must consider whether 0 is the best choice of default value for this case, or whether something else would be better suited, for example the issuance time.

5.3 Verifying the Validity and Integrity of Credentials

In this section we show how we can implement the verification of the validity and integrity of a given credential using arithmetic circuits. This process consists of three steps:

1. Verifying the signature of the credential,
2. Checking that the credential has not expired,
3. Checking that the credential has not been revoked (optional).

Signature validation Verifying the signature of a credential consists of two basic operations. First we reconstruct the root of the Merkle-tree using its leaves. Then we verify the signature using the recomputed Merkle-tree root as the plaintext message. This process not only ensures that the signature in our possession was signed by the issuer, but it also proves the integrity of the leaves of the Merkle-tree, and, in turn, of the claims of the credential.

Checking expiration Checking whether a credential is expired is pretty simple and consists of a single `LessThan` circuit where the (supposed to be) smaller input is the credential expiration date, and the other input is the current time, both expressed as UNIX timestamps.

Checking the status of the credential To check the status of a credential we (dynamically) access corresponding entry of the corresponding status list (consisting of 131 072 entries), and check whether its value indicates that the credentials is valid. Each status list defines the size of its entries, where the only allowed sizes are, as per [72], 1, 2, 4, and 8. The Swiss e-ID uses a status list where each entry is 1 bit in size and representing two possible status:

- `0x00`: valid,
- `0x01`: invalid (revoked),

which results in a status list of size 2^{14} bytes. Note that not all issuers may implement a status list for the credentials they issue, if that's the case, then we simply omit this last check.

Code snippet B.5 shows a possible Circom implementation of the verification of the validity and integrity of a credential.

5.4 Preventing Prover Foul Play

While a prover may be in possession of a valid credential, nothing is preventing them to use forged values inside the proof instead of the actual claims. Indeed, we must consider very carefully what a statement actually proves. Are we checking that the prover possesses a valid credential which states that they are older than 18, or that they possess a valid credential and know a birthdate corresponding to somebody who's older than 18 at that time?

In this section we introduce a mechanism aimed at preventing foul-play from the prover's side by tying each value used inside the proof to the signature of the credential. In practice, any proof receives as part of its private inputs an array containing the leafs of the Merkle-tree whose root is signed by the issuer. Recall that each leaf l_i is defined as

$$l_i := H_{Pos}(H_{Pos}(cn_i), cv_i)$$

where H_{Pos} denotes the Poseidon hash function, cn_i and is the name of the i-th claim, and cv_i represent the encoding of the value of the i-th claim. The integrity of the leafs can be enforced by reconstructing the root of the Merkle-tree and verifying its signature (see Section 5.3). We can thus force the prover to use the correct claims by, tying the supposed value cv'_i of each claim used in the proof to the corresponding leaf l_i by recomputing

$$l'_i = H_{Pos}(H_{Pos}(cn_i), cv'_i)$$

inside the circuit and the testing that l'_i is equal to l_i. Code snippet B.3 shows the Circom implementation of this mechanism while snippet B.4 illustrates how we can apply it to a concrete use-case.

5.5 Credential Linking

Credential linking is the mechanism that ensures that all the credentials used in a verifiable presentation were issued to the same subject. A common strategy to implement credential linking, is to take a set of claims shared by a pair of credentials (e.g. first, last name, and birthdate) and prove that their values are the same. There are some problems with this approach, in particular, cases where two credentials share no claims. It could also happen, although rarely, that two different subjects share the same values for the set of claims used to enforce credential linking, this would enable

these two subjects to use each other's credentials interchangeably while still satisfying the constraints of credential linking.

To circumvent these two problems, and to make credential linking checks cheaper and uniform over any pair of credentials, we introduce a new claim (called `cred_bind`). This is a per-subject unique value (e.g. SHA256 of the subject's public key) contained in all credentials issued to that subject. Then, to check whether two or more credentials were issued to the same subject, we simply need to compare the `cred_bind` claims across all credentials using a `IsEqual` circuit.

This approach is similar to the link secret[2] introduced in Hyperledger Indy [34].

5.6 Hardware Binding

Hardware (or device) binding is a requirement of the Swiss e-ID. For this purpose, the holder generates a private-public key-pair using the secure element on their device, and communicates the resulting public key to the issuer, which will include it as a claim in the credential. These hardware modules implement only a small selection signature schemes [3, 4], in particular they mostly support only the NIST standard curves[29] P-256, P-384 and P-521. This means, that for the implementation of hardware binding, we cannot use a natively supported signature scheme, but we are forced instead to use a much more expensive one. For our implementation we choose to focus on P-256, we refer to Section 4.6 for how to implement it as an arithmetic circuit and for possible optimisations. Signature scheme aside, implementing hardware binding is quite straightforward. The verifier provides the holder with a challenge nonce. The holder proves possession of their private key asking the secure element on their device to produce a signature and then verifying the signature in zero-knowledge using the public key contained in their credential without leaking it.

5.7 Architecture Overview

In this section we provide an overview of the various components of our proof-of-concept implementation.

Proof system We use Circom as frontend and spartan-secq256k1 as backend. In order to integrate these two components together, we need a middle layer which translates both the circuit (`.r1cs`) file and the witness (`.wtns`) file generated by Circom into a form compatible with spartan-secq256k1. The format of the file containing the R1CS representation of an arithmetic circuit and of the file containing the witness can be found in the `info` folder.

[2] https://hyperledger.github.io/anoncreds-spec/#term:link-secret

The code for translating the `.r1cs` file can be found it `src/circuit_reader` and implements a straightforward conversion into the appropriate Rust struct. There is however a tiny catch. In Circom, each wire has an ID associated to it, where ID 0 always represents the constant 1, IDs 1 to l are associated to the public inputs and outputs of the circuit (where #public inputs + #public outputs = l), and the remaining n IDs are associated to all private values of the circuit, including its private inputs. In synthesis the wire IDs are sorted as `[ONE | public | private]`. However, spartan-secq256k1 uses a different order for the wire IDs, i.e. `[private | ONE | public]`, which means that we must pay attention to remap them accordingly when translating the `.r1cs` file.

The code for translating the `.wtns` file can be found in `src/witness_reader`, and implements a simple one-to-one conversion into the corresponding Rust struct.

Issuer For the implementation of the issuer we build upon the beta version of the Swiss e-ID issuer [39], and extend it to support general-purpose zero-knowledge proofs. More specifically, we modify the Java class responsible for creating and signing SD-JWT credentials[3] such that it implements the additional helper claims described in Section 5.2.3. For this purpose we first encodes all the claims as showed in Section 5.2.1, and then we generate the corresponding Merkle-tree and the signature of its root, as per Section 5.2.2. This turns out to be quite a small modification, with around 20 lines of code being added to `SdJwtCredential.java`, while the bulk of the functionality is implemented in an additional Java class[4] with circa 300 lines of code.

Holder The holder is a mix of Rust and python code which can be found in `src/prover` and `src/python_util`. When proving a statement, the holder performs the following actions:

1. Generates the input of the statement.
2. Generates the `.wtns` file containing the corresponding witness.
3. Generates a proof for the statement with spartan-secq256k1 using the witness generated in the previous step.

In order to automatically generate the input file, and to facilitate addressing multiple credentials within a single statement, we associated to each statement a file called `proof_metadata.json`. This JSON file contains four lists, where each of their elements represents some information relating to one of the credentials addressed by the statement.

- `inputs`: each element is a list indicating which claims of the corresponding credential are passed as inputs to the statement.

[3] `src/issuer/src/main/java/ch/admin/bj/swiyu/issuer/oid4vci/service/SdJwtCredential.java`
[4] `src/issuer/src/main/java/ch/admin/bj/swiyu/issuer/oid4vci/zk/CredentialZK.java`

- `order`: each element is a list indicating the order of the claim in the corresponding credential.
- `type`: each element is a list indicating the type of each claim in the corresponding credential.
- `prefix`: each element is a string which is prepended to each of the inputs of the corresponding credential to avoid name-collisions.

A file called `auxiliary_inputs.json` provides additional (global) inputs to the statement such as the issuer's public key or the current timestamp.

Verifier The verifier is a simple SPARTAN-SECQ256K1 implementation which, given as inputs an `.r1cs` file expressing a statement and a corresponding SPARTAN proof with associated public inputs and outputs, verifies the validity of said proof.

5.8 End-to-End Workflow

In this section we describe the end-to-end workflow of our proof-of-concept implementation starting from credential issuance and going through all the steps necessary to then generate and verify a proof for a statement on a credential.

1. Credential issuance The first step in the workflow is to obtain a credential from the issuer. To this end we need to send a *credential offer request* to the issuer containing a *credential offer* such as the one in Listing 5.3.

```
{
    "offer_id": "1c2e0220-ac16-493a-9ab8-cdd7c4307309",
    "vct": "identity_credential",
    "offer_data": {
        "data": {
            "first_name": "Axel",
            "last_name": "Andersson",
            "email": "aa@example.com",
            "phone_number": "+41223334455",
            "nationality": "CH",
            "address": "Zollstrasse 62",
            "locality": "Zuerich",
            "canton": "Zuerich",
            "postcode": "8005",
            "country": "CH",
            "birthdate": "1998-01-04T00:00:00Z"
        },
        "cnf": {
            "x": "61a993e5...7a289b28",
            "y": "698b8954...31d915d5"
        },
        "cred_bind": "bf8332c5...59d768c2",
        "status_list": {
```

```
            "idx": 1124,
            "uri": "status_list_01"
        }
    },
    "nbf": "2025-02-25T00:00:00Z",
    "exp": "2035-02-24T00:00:00Z"
}
```

Listing 5.3: Example of a credential offer

The *credential offer* contains some metadata about the credential such as its schema (indicated by `"vct"`) and its not-before and expiration date (indicated by `"nbf"` and `"exp"`). It also contains the actual content of the credential defined by the `"offer_data"` field where `"data"` defines the claims attributed to the subject of the credential, while `"cnf"`, `"cred_bind"` and `"status_list"` are additions that we made to simplify our implementation, which contain the public key for hardware binding, the value for credential linking, and the identifiers for the credential's entry in its status list respectively.

The issuer then returns the compact serialisation (as defined in section 7.2 of RFC 7517 [66]) corresponding signed SD-JWT credential with our zero-knowledge extensions as described over the course of this section.

2. Expressing and compiling statements Statements are expressed using the Circom language. Generally, the process of writing a statement consists of combining "simpler" circuits together, Appendix B includes some examples of building-blocks-circuits that can be used for this purpose, into a final, more complex circuit. In Circom, the complete, final statement is defined with the *main component* expression and is indicated as

$$\texttt{component main \{public [pi}_0\texttt{, ..., pi}_{n-1}\texttt{]\} = statement();}$$

where the {`public [...]`} clause indicates the set public inputs of the statement, i.e. inputs known by the verifier, while all other inputs are, by default private, i.e. unknown to the verifier. All outputs in Circom are public.

Once a statement is defined, it can be compiled using Circom into its R1CS representation. We can achieve this using the following command,

```
circom statement.circom --r1cs --c --prime secq256k1 --O2
```

where:

- `statement.circom` is the file containing the *main component* expressing the statement,
- `--r1cs` indicates the target intermediate representation/constraint system (in this case R1CS).

- `--c` indicates the choice of language for the witness generator program (can be either C or WASM).
- `--prime` indicates the elliptic curve used by the proof system (in this case secq256k1).
- `--O2` indicates the optimisation strategy used by the compiler, it can either be `O0`, `O1`, or `O2`. `O2` is the highest and is able to eliminate all linear constraints in most cases.

This command has two outputs, one is an `.r1cs` file containing the R1CS representation of the statement while the other is (in our case) a C program for efficiently generating the witness for the circuit, i.e. an assignment to all the wires in the circuits satisfying all constraints.

3. Generating proof input and witness Given a credential and a statement, we need now to generate the witness for the statement w.r.t. the credential. As mentioned before, a witness is an assignment to all the wires of a circuit such that all of its constraints are satisfied. It includes all public and private inputs, all outputs, and all other intermediate values. First we need to create a (valid) input for the circuit, i.e. an assignment to all inputs, both public and private of the circuit. For this purpose we provide a Python script[5] which generates a `input.json` file containing the corresponding input.

Then, we can use the `input.json` file and the witness generator program created by Circom to compute the corresponding witness, contained in a `.wtns` file. Note that is is possible to create an invalid input for the circuit, i.e. an input for which there exists no wire assignment satisfying all the constraints, if this happens the witness generations fails and no witness file is generated.

4. Generating and verifying proofs We can now generate a proof of the output of the statement, w.r.t. a credential, given the corresponding `.r1cs` and `.wtns` files, using spartan-secq256k1. Then we can verify said proof, again using spartan-secq256k1, by using the same `.r1cs` file, and providing the public inputs and outputs of the circuit to the verifier. Note that the proof only ensures that the output of the statement is correct w.r.t. the input, to actually know whether the credential satisfies the statement we need to interpret the output in the context of the statement.

5.9 Concrete Examples

In this section we take a closer look at some concrete examples that we developed for our proof-of-concept implementation, highlighting the more

[5] `src/python_util/gen_input.py`

Statement	#constraints	#inputs	Prover time	Verifier time
BV	24 857	1 073	2 504	196
BV + age check	25 467	1 076	2 554	199
Credential linking (simple)	50 077	1 302	5 110	226
Credential linking (advanced)	117 176	1 548	12 968	495
Hardware binding	238 228	99 387	20 868	809

Table 5.2: Summary of the concrete examples, BV $\triangleq$ basic validation, prover time and verifier time expressed in milliseconds

prominent use-cases that we envision for our system such as basic credential validation, credential linking with cross-credential constraints, and hardware binding. We go into more detail on their design and evaluate them both in terms of complexity (in the number of non-linear constraints) and of execution time needed to generate and verify a proof, table 5.2 summarises the results.

For our examples we defined three different credential schemas, modelling an identity document, a university diploma and a reference letter respectively. These schemas can be found the aforementioned `issuer_metadata.json` file. We also instantiate a status list[6] with entry-size of 2 bits representing three possible status:

- `0x00`: valid,
- `0x01`: invalid (revoked),
- `0x02`: suspended,

and which can be fully represented with 1024 256-bit integers.

We use a single issuer for issuing all three credential types to simplify the process of writing the statements, please keep in mind that this would not be the case in a real-world scenario, where there will likely be multiple issuers in play.

5.9.1 Credential Validation and Additional Checks

The most basic, and also the most common, statement that one can prove about a credential is to check its validity and integrity. Section 5.3 already explains the necessary operations needed to prove such a statement, i.e.

[6] `status_list/status_list_01`

verifying the signature of the credential and making sure that the credential has not expired and has not been revoked. We implemented this statement[7] for the setting described above which results in a circuit size of 24 857 non-linear constraints. We are able to generate a proof for it in $\sim$ 2.5 seconds while the verification takes $\sim$ 0.2 seconds.

The bulk of the constraint-cost lies in the signature verification ($\sim$ 20 000 non-linear constraints) and in the dynamic access of the revocation list (4000 non-linear constraints). Adding any other operations to the statement has, in most cases, a negligible impact on the cost of the resulting circuit. In fact we also wrote a second "basic" statement[8], which, in addition to checking the validity of a credential, also checks that the age of the older is above 18. The resulting circuit is 25 467 non-linear constraints is size, only 610 more, and has the same prover and verifier execution time.

5.9.2 Credential linking and cross-credential constraints

We created two examples to illustrate the use and possible applications of credential linking. The first[9] one is a simple application of credential linking with only two credentials. These are a digital identity document and a university diploma, and are issued to the same subject. We check the validity for both credentials and compare their `cred_bind` claims to verify that they were issued to the same holder. We also perform some additional checks on the claims of the two credentials, Listing 5.4 displays in detail the statement that we want to prove.

```
1 out = eid.is_valid AND eid.age >= 18
2         AND eid.locality == "Zuerich"
3         AND eid.nationality == "CH"
4         AND diploma.is_valid
5         AND (diploma.university == "ETHZ"
6         OR diploma.university == "EPFL")
7         AND LINKED(eid, diploma)
```

Listing 5.4: Example of simple credential linking

This statement results in a circuit with 50 077 non-linear constraints, we are able to generate a proof for it in $\sim$ 5.1 seconds and verify it in $\sim$ 0.3 seconds. Again, the bulk of the cost of the circuit is the verifying the signatures of the two credentials which accounts for around 40 000 non-linear constraints.

The second example[10] is more involved and includes 5 different credentials across three credential schemas. These are an identity document, a university diploma and three "reference letters", which contain information on a

[7] `proofs/basic_verification/proof.circom`
[8] `proofs/basic_verification_age_check/proof.circom`
[9] `proofs/basic_credential_binding/proof.circom`
[10] `proofs/advanced_credential_binding/proof.circom`

person's previous jobs including profession and the time period of the employment (indicated by start-date and end-date). Again, we first check the validity of all credentials and also that the were issued to the same subject. Then we perform some additional operations, and, in particular, we check that the total years of experience in either "Security Engineering" or "Security Analyst" across all reference letters amount to at least five years. Listing 5.5 displays in more detail the statement that we want to prove.

```
// rfi is the i-th reference letter
rfs = [rf1, rf2, rf3]

out = eid.is_valid AND eid.age >= 18
        AND diploma.is_valid
        AND diploma.degree == "Computer Science MSc"
        AND (diploma.university == "ETHZ"
        OR diploma.university == "EPFL")
        AND rf1.is_valid
        AND rf2.is_valid
        AND rf3.is_valid
        SUM([(rf.end_date - rf.start_date) FOR rf IN rfs
            WITH rf.profession IN
            {"Security Engineering", "Security Analyst"}]) >= 5
        AND LINKED(eid, diploma, rf1, rf2, rf3)
```

Listing 5.5: Advanced example of credential linking

The resulting circuit consists of 117 176 non-linear constraints, the proof generation time is 13 seconds and the verification time is 0.5 seconds. This example serves to illustrate an additional feature of our approach, i.e. we can easily perform operations on multiple claims belonging from different credentials that are linked together.

5.9.3 Hardware binding

We also implemented an example[11] to illustrate a concrete use of hardware binding. We use the circuit described in Section 4.6 to implement NIST P-256 signature verification. For that, we need to precompute a bunch of scalar multiplications of the point T and pass them as public inputs to the circuit, this results in 98 304 additional inputs. We check hardware binding together with the basic credential validation, the resulting circuit is 238 228 non-linear constraints in size. For this circuit, the proof generation takes $\sim$ 21 seconds while the proof verification takes $\sim$ 0.8 seconds.

5.9.4 Malicious statements

Our last concrete example is aimed at highlighting the possible dangers related to blindly trusting statements. In fact, it is possible to exploit the

[11] `proofs/hardware_binding/proof.circom`

circuits written with CIRCOM to leak information about the credential. Our example[12] adopts a quite direct approach and simply dumps all the claims of the credential directly into the output of the statement. Of course, it's possible to use more subtle approaches such as hiding information about the claims inside the output of the statement in a way which is harder to detect when looking at the code.

A possible solution to this problem could be sending statements expressed human-readable way to the holder which would than perform an on-the-fly translation of that statement into a circuit and then compile it. However, compiling circuits is time-consuming, making this approach impractical. Of course a verifier could itself translate and compile a human-readable statement beforehand and use verifiable execution to prove that the resulting `.r1cs` file corresponds to that statement, but we would be introducing a lot of complexity and additional steps into the system.

A more realistic solution would be to either establish a set of trusted entities which would audit each circuit guaranteeing that it is not malicious, or to define a collection of standardised open source proofs. A holder would then only accept to generate proofs for circuits which either have been certified by these entities, or are included in the trusted collection.

[12] `proofs/malicious/proof.circom`

Chapter 6

Discussions

In this chapter we cover a set of additional consideration and discussion points about our verifiable credential system, in particular about its performance (Section 6.1) and its security (Section 6.2). Additionally, we briefly go over verifier designated proofs and how we could implement them (Section 6.3).

6.1 On the Performance of the System

As we have seen in the previous chapter, the most expensive operation in our system is the generation of a proof, with execution times ranging from $\sim$ 2 seconds for simpler statements, to tens of seconds when dealing with more complex operations like hardware binding. We consider prover times below 5 seconds to be acceptable for our use-case, and we find higher values are to be impractical. Realistically, we think that the vast majority of zero-knowledge statements that a holder would need to prove are going to be very similar to the ones shown in Section 5.9.1, which would already be feasible to prove with our current system.

Another important performance consideration, which could potentially be significant when working with smartphones, is on the size of the `.r1cs` and `.wtns` files, which can get quite large for higher counts of non-linear constraints. For example, in the hardware binding example mentioned in Section 5.9.3 ($\sim$ 220 000 non-linear constraints) the `.r1cs` file has a size of 50 MB while the `.wtns` of 11 MB. Of course, larger file sizes translate to higher prover times, given that it takes longer to load them, but another problem could also be storage space, since the `.r1cs` files are likely going to be stored directly on the holders' devices for faster access.

Therefore, when striving for better performance, in addition to minimising the size of the circuit, we should also try to keep the number of inputs and

outputs of the circuit small, which are the only aspect of the witness size that we can directly control. Goals which can sometime clash with each other, considering that one of the most common strategies for smaller circuits is to move expensive operations to be computed off-circuit, which can in turn lead to a higher number of inputs, as it is the case with hardware binding and NIST P-256 (see Section 4.6).

6.2 On the Security of the System

The theoretical security of our system relies both on the security of the zk-SNARK framework that we use and on the security of the cryptographic primitives that we employ. Regarding zk-SNARK frameworks, it's important to stress that all available implementations (except very few such as rapidsnark and snarkjs) are purely experimental and were developed for research purposes. In particular, they have never received any security review or audit, it's very likely that they contain multiple bugs and security flaws, and should NOT be used in any real-world application. Therefore, an actual implementation of our system would need to either carefully test the frameworks that it uses or build its own implementation of a proof system with a particular focus on security and performance.

Regarding the cryptography used in our proof-of-concept, standard primitives, such as SHA256 and NIST curves, are still too expensive. Therefore, we opted for alternative cheaper solutions, namely Poseidon hashes and secp256k1. While both have been (for the moment) shown to be secure, guaranteeing 128-bits of security, they also have received less scrutiny from cryptanalysts and less field-testing, moreover, NIST suggests the use of secp256k1 only for blockchain-related applications [29] with no mentions about its use in other applications. In practice, our choice of primitives should provide a sufficient level of security for the time being, however, one must be aware that it is possible that vulnerabilities for these primitives could be eventually discovered. Therefore, one should strive to optimise, and move towards the adoption of well known, secure cryptographic standards.

Additionally, we should also consider the possibility of side-channel attacks. We did not have the opportunity to deeply study potential vulnerabilities of our system related to this kind of threats, still, we can mention that due to Circom being limited to static control flow, generating a proof for a fixed statement is done in constant time independently of the input, which excludes one type of side-channel, namely timing attacks.

6.3 On Designated Verifier Proofs

With designated verifier proofs we refer to those proofs that are specifically constructed for, and can convince a single verifier and nobody else. It is, in practice, a mechanism to prevent the retransmission of proofs, and information contained within, to third parties. While impersonation can be prevented by including verifier-provided nonces, or current timestamps in the proof, a verifier can still leak the information revealed by the proof, using the proof itself as evidence of its validity. We can implement designated verifier proofs by modifying the output of the statement to be true when either the original statement is true, or the prover knows the private key of the designated verifier. This modified statement would still convince the designated verifier of the output of the original statement, since it (theoretically) knows that it's private key has not been leaked, while any other verifier will not be convinced because the designated verifier could create true statements for arbitrary inputs given that it knows its own private key. This modification can be easily implemented with Circom and would cost $\sim 4\,000$ non-linear constraints since we do not need to keep the verifier's public key secret, and we can therefore use the fast circuit implementation for secp256k1 signature verification.

Chapter 7

Related Work

In this chapter we briefly mention other research work on the subject of verifiable credentials based on general-purpose zero-knowledge proofs.

zk-creds developed by Rosenberg et al. [81] is a framework for verifiable credentials based on general-purpose zero-knowledge proofs which removes the need for issuers to hold signing keys. Instead, credentials are issued to a so-called issuance list, i.e. a public bulletin board implemented using technologies such as transparent logs, Byzantine systems or blockchains and maintained by each issuer. This approach requires only to trust the issuers and removes the need to also trust their signing keys. ZK-CREDS implement the issuance list using a Merkle-forest based on Poseidon hashes. The reason for using Merkle-forests is to offer a tunable trade-off between prover and verifier time since shorter Merkle-trees have faster membership proofs. A credential is issued by adding a commitment of it as a leaf to one of the Merkle-trees, and it is revoked by removing the same leaf. When presenting a credential, a holder proves that:

1. They have a credential committed in one of the Merkle-trees of the Merkle-forest.
2. The same credential meets a set of access criteria.

The framework is built on top of GROTH16, therefore it can express arbitrary statements on a credential, while needing a trusted setup for each individual statement. Additionally, it can support arbitrary credential formats and can also be used to convert existing non-anonymous credentials into anonymous ones. Compared to our system, ZK-CREDS completely removes signature verifications, and thus the main bottleneck of generating proofs. This feature also makes it easier to develop a post-quantum verifiable credentials systems based on general-purpose zero-knowledge proofs because it removes the need to switch to, and implement a post-quantum signature scheme using arithmetic circuits. However, this approach also completely changes the

architecture of the system, making it harder to transition to from the current SSI-based architecture.

Heimdall developed by Babel and Sedlmeir [7] is a zk-SNARK-based implementation of verifiable credentials. Credentials are represented as a binary Merkle-tree where the leaves of the left half contain metadata about the credential such as revocation information, credential schema, or expiration date, and the leaves of the right half contain the claims of the credential encoded as 253-bit integers. Heimdall uses Groth16 as its proof system on the `alt_bn128` elliptic curve, therefore it relies on Poseidon hashes and EdDSA signatures on the Baby Jubjub elliptic curve [95] to create and sign the Merkle-tree. This approach is quite similar to our work in this thesis, however it uses a less secure signature scheme and is in general more rigid in the type of credentials in can represent (e.g. it has no direct support for dynamic size strings) making it harder to integrate with existing SD-JWT credentials schemas. The reliance on Goth16 also means that a it needs a trusted setup for each statement.

Uuna Saarela explored in her master's thesis [82] the use of zk-SNARKs to perform identity checks on EUID credentials. For this purpose she uses the ZK-SecreC [21] DSL in combination with the Mac'N'Cheese [8] proof system. ZK-SecreC is a strongly typed, functional DSL for describing zero-knowledge computations that are compiled into circuits. It is designed for applications with large proof sizes and input data with complex structures. It provides fine-grained control in defining where computations should be performed, either in- or off-circuit. The resulting framework directly parses the CBOR representation of EUID credentials by computing a bunch of SHA256 hashes and can perform arbitrary operations over their claims. This approach has the advantage of not needing any modification on the issuer size and on the format of the credentials, in contrast to our design, however the high amount of SHA256 operations results in extremely large circuits and high prover times. Additionally, the verification of the credential's signature is performed off-circuit, meaning that this approach only guarantees the zero-knowledge property for the claims of a credential and not for its signature, introducing the potential linkability of the holders via the tracking of the ECDSA signatures.

Frigo and Shelat [46] presented a strategy to perform extremely efficient ECDSA signature verifications for NIST standard curves and SHA256 computations by working over multiple finite fields simultaneously. In particular, for ECDSA signature, they use a quadratic field extension enabling fast zero-knowledge operations (such as number theoretic transforms) in fields where the would normally be slow, such as the ones of the NIST curves, due to them not having proper roots of unity. For SHA256 computations they follow the work of Diamond and Posen [41, 42] and make use of the

extension field GF(2^k) to speed-up boolean operations. Their proof system combines an optimised verifiable computation protocol based on sumcheck and the Ligero proof system. Using this approach they create a framework for verifiable credentials capable of efficiently parsing and verifying credentials in CBOR format with a prover time of around 1 second and a verifier time of around 0.5 seconds. Additionally, this system can be integrated seamlessly into existing verifiable credential system without the need to modify the issuer or the credentials format.

Paquin et al. [76] proposed an approach for implementing verifiable credentials based on zk-SNARKs which requires no changes on the issuer size and no re-issuance of credentials. They observed that, generally, most of the work done by proofs is always the same, and is dominated by the signature verification. Their design moves this expensive operation to a pre-processing phase which the prover executes only once for each credential. This pre-processing phase leverages the Groth16 proof system to generate of proof of a valid credential's signature which outputs the claims of the credential, and can be re-randomised with each presentation to avoid linkability of the holder. They then use strategies similar to LegoSNARK to combine this pre-processed proof with additional constraints on the claims of the credentials. They show that this approach is able to create proofs for credentials signed with either RSA or ECDSA in a matter of milliseconds at the cost of a one-time pre-processing phase which takes around 15 seconds.

Chapter 8

Conclusions

The Swiss confederation has been tasked with developing the Swiss e-ID, an infrastructure for digital identities, following the principles of SSI, which enforce strong privacy requirements. These requirements are, however, hard to achieve with classical, standard methods such as ECDSA signatures. Recently, verifiable credentials based on zero-knowledge proofs have been proposed as an alternative to achieve higher privacy guarantees. In this thesis we studied the feasibility of using general-purpose zero-knowledge proofs, in particular zk-SNARKs, for implementing flexible logic for verifiable credentials. We conducted an overview and comparison of existing, state-of-the-art, zero-knowledge proof frameworks and analysed their performance with a particular focus on the prover's execution time. We further provided a suite of basic primitives for expressing flexible credential verification logic, and we developed an end-to-end proof-of-concept implementation of zk-SNARK-based verifiable credentials by extending the functionalities of the beta version of the Swiss e-ID. Our additions are non-intrusive and can work alongside the existing protocols without interference.

The work has uncovered the many challenges related to our approach and highlighted the current bottlenecks in the system. We presented a set of possible solutions to some of these problems, in particular for optimising the cost of the statements. Finally, we presented the various components needed for the implementation of our system and we showed that this approach is already feasible with the existing technologies for simple statements such as validating a credential.

In conclusion, the research field on general-purpose zero-knowledge proofs is extremely active, with several new proof systems being developed each year. Therefore, we are positive that, even though at the time of writing we found our system to be only practical for proving very relatively simple logic, in the near feature new developments in the research will close the performance gap that is currently present in our implementation.

In Section 8.1 we detail several avenues for future work which could contribute to improving and expanding the current design of the system.

8.1 Future Work

During the course of this work we had to make several compromises, due to either time constraints or complexity, in terms of performance, scope of the project, design features, and security. These result in the following avenues for future work, divided by topic.

Performance

- Improve the performance of the system to achieve practical proof generation time even for complex statements (Section 5.9), this includes:
  - Studying more proof systems, in particular post-quantum systems and those compatible with efficient implementations of standard cryptographic primitives, such as, for example, the work from Frigo and Shelat [46] presented in Section 7.
  - Explore other strategies for more efficient implementations of expensive operations (Chapter 4), for example, look-up tables [6, 87] for efficiently computing SHA256 hashes.
- Work towards standardisation of, and hardware support for circuit-friendly primitives.

Security

- Deeper security analysis and testing of both the proof systems that are used and of the actual implementation.
- Additionally, consider moving towards post-quantum proof systems and signature schemes. For the first, lattice-based proof-system such as LaZer [74] appear to be quite promising. For the latter, one can study the feasibility of implementing post-quantum signature schemes, such as Dilithium [43], using arithmetic circuits, to our knowledge, at the time of writing, no such implementation exists.

Implementation

- Deeper integration of the implementation in the Swiss e-ID infrastructure (Section 5.7), in particular in regard to holder and verifier.
- Expand the scope of the implementation to include a more diverse set SD-JWTs, in particular introducing support for lists and objects and for more data types such as floating point numbers (Section 5.2.1).
- Study other possible architecture for verifiable credential systems, such as, for example, the one implemented in zk-creds (Chapter 7) were the

main bottleneck for zk-SNARKs, i.e. verification of ECDSA signatures, is removed.

- Consider other aspects of deployment, for example defining a trust infrastructure for preventing malicious proofs (Section 5.9.4).

Usability

- We noticed that, when expressing statements in Circom, we had a lot of repeating structures in the code and that instantiating components with several inputs is quite awkward and prone to errors (see Listing B.5). However, this type of code is also very structured, and well suited to be generated automatically. Therefore, it would be desirable to implement a simple DSL for describing verification logic for credentials, similar for example to what we used in this document (see Listing 1.1), that would be automatically compiled into Circom circuits
- Consider how to visualize the functionality implemented by a statement to the users.

Appendix A

Benchmark Results

Here we provide concrete values for the measurements of prover time, verifier time, and proof size that were displayed in Figures 3.1, 3.2, and 3.3 respectively.

log(#constraints)	rapidsnark	snarkjs (Groth16)	snarkjs (Plonk)	Spartan (curve25519)	spartan (secq256k1)	libiop (Ligero)	libiop (Aurora)
8	0.0157	0.491	0.746	0.0122	0.0501	0.0749	0.0943
9	0.0180	0.533	0.982	0.0143	0.0668	0.154	0.146
10	0.0241	0.542	1.44	0.0198	0.0924	0.160	0.247
11	0.0439	0.589	2.31	0.0292	0.151	0.298	0.542
12	0.0658	0.682	3.92	0.0440	0.240	0.358	1.04
13	0.110	0.858	7.22	0.0809	0.413	0.742	1.88
14	0.177	1.18	13.5	0.145	0.717	0.971	4.11
15	0.311	1.89	26.6	0.253	1.10	1.99	8.29
16	0.582	3.14	52.2	0.481	2.03	3.07	15.4
17	1.24	5.62	104	0.839	3.97	6.30	33.6
18	2.22	10.2	207	1.63	7.71	10.7	68.8
19	4.46	21.3	413	2.96	11.8	21.7	128
20	8.95	41.1	824	5.72	23.3	39.8	286

Table A.1: Results of the measurements of the prover's execution time in seconds

log(#constraints)	rapidsnark	snarkjs (Groth16)	snarkjs (Plonk)	Spartan (curve25519)	spartan (secq256k1)	libiop (Ligero)	libiop (Aurora)
8	0.00649	0.416	0.410	0.00566	0.0209	0.0839	0.0227
9	0.00734	0.425	0.422	0.00560	0.0236	0.133	0.0197
10	0.00682	0.405	0.408	0.00717	0.0264	0.135	0.0295
11	0.00619	0.414	0.423	0.00782	0.0304	0.250	0.0561
12	0.00619	0.420	0.425	0.0111	0.0352	0.289	0.101
13	0.00476	0.430	0.404	0.0125	0.0417	0.553	0.156
14	0.00500	0.420	0.417	0.0184	0.0515	0.718	0.359
15	0.00614	0.418	0.419	0.0282	0.0670	1.44	0.696
16	0.00606	0.425	0.434	0.0482	0.0933	2.11	1.12
17	0.00651	0.418	0.415	0.0882	0.150	4.34	2.74
18	0.00632	0.438	0.448	0.164	0.251	7.32	5.31
19	0.00489	0.426	0.416	0.330	0.445	14.8	9.14
20	0.0137	0.441	0.419	0.666	0.849	27.1	21.5

Table A.2: Results of the measurements of the verifier's execution time in seconds

log(#constraints)	rapidsnark	snarkjs (Groth16)	snarkjs (Plonk)	Spartan (curve25519)	spartan (secq256k1)	libiop (Ligero)	libiop (Aurora)
8	705	807	2248	7456	9369	401184	69088
9	707	804	2247	8208	10315	593600	78528
10	708	806	2252	9408	11835	593600	77632
11	705	808	2249	10160	12781	925792	101920
12	708	804	2254	11872	14957	1031904	115136
13	709	805	2252	12624	15903	1654336	107232
14	709	804	2248	15360	19391	1870912	138624
15	706	808	2250	16112	20337	3086080	150976
16	709	808	2242	20896	26449	3525376	139840
17	709	806	2248	21648	27395	5918208	177920
18	707	804	2250	30528	38755	6801920	193024
19	706	805	2248	31280	39701	11554816	174400
20	706	805	2248	48352	61557	13328384	222848

Table A.3: Results of the measurements of the generated proof size in bytes

Appendix B

Sample Circuits

Here we provide the sample Circom circuits that were referenced in Sections 4 and 5.

```
template IsZero() {
    signal input in;
    signal output out;
    signal inv;

    inv <-- in = 0 ? 1 / in : 0;
    out <== -in * inv + 1;
    in * out === 0;
}
```

Listing B.1: Circom code for 0 testing

```
template ListSelector(N) {
    signal input arr[N];
    signal input ind;
    signal output selected;

    var selectedElement[N];
    component eqs[N];

    for (var i = 0; i < N; i++) {
        var selector = IsEqual()([i, ind]);
        selectedElement[i] = selector * arr[i];
    }

    selected <== CalculateSUM(N)(selectedElement);
}
```

Listing B.2: Circom code for dynamic array lookup

```
template VerifyClaimMembership(n_claims, index) {
    signal input hashed_claims[n_claims];
    signal input claim_name;
    signal input claim_value;

    component ph = Poseidon(2);
    ph.inputs[0] <== claim_name;
    ph.inputs[1] <== claim_value;
    ph.out === hashed_claims[index];
}
```

Listing B.3: Circom code for enforcing prover's honesty

```
template VerifyClaimMembership(n_claims, index) {
    signal input hashed_claims[n_claims];
    signal input now_timestamp;

    // Result of the Poseidon hash of the string "exp"
    signal input exp_name;

    // Expiration date encoded as a UNIX timestamp
    signal input exp_value;

    signal output not_expired;

    // Enforce usage of the correct value of "exp"
    component ver_exp = VerifyClaimMembership(n_claims, 4);
    ver_exp.hashed_claims <== hashed_claims;
    ver_exp.claim_name <== exp_name;
    ver_exp.claim_value <== exp_value;

    // Operations with "exp"
    component exp_check = LessThan(128);
    exp_check.in[0] <== now_timestamp;
    exp_check.in[1] <== exp_value;
    not_expired <== exp_check.out;
}
```

Listing B.4: Example of enforcing prover's honesty w.r.t. some claim

```
template CredentialValidityCheck(n_claims, sl_length) {
    // ---------------- Public inputs ----------------
    signal input exp_name;

    signal input status_list[sl_length];
    signal input status_list_uri_name;
    signal input status_list_uri_value[15];

    signal input status_list_idx_name;

    signal input iss_pk_x;
    signal input iss_pk_y;


    signal input now_timestamp;
    signal input date_offset_name;
    signal input date_offset_value;

    // ---------------- Private inputs ----------------
    signal input zk_r;
    signal input zk_s;

    signal input exp_value;
    signal input status_list_idx_value;


    signal input hashed_claims[n_claims];

    // ---------------- Outputs ----------------
    signal output out;

    // Verify signature
    component ver = VerifyCredentialSignature(n_claims);
    ver.Qx <== iss_pk_x;
    ver.Qy <== iss_pk_y;
    ver.hashed_claims <== hashed_claims;
    ver.r <== zk_r;
    ver.s <== zk_s;
    signal valid_signature <== ver.valid;

    // Verify non-expiration
    component ver_exp = VerifyClaimMembership(n_claims, 4);
    ver_exp.hashed_claims <== hashed_claims;
    ver_exp.claim_name <== exp_name;
    ver_exp.claim_value <== exp_value;

    component ver_date_offset = VerifyClaimMembership(n_claims,
    10);
    ver_date_offset.hashed_claims <== hashed_claims;
    ver_date_offset.claim_name <== date_offset_name;
    ver_date_offset.claim_value <== date_offset_value;

    component exp_check = LessThan(128);
    exp_check.in[0] <== now_timestamp + date_offset_value;
```

```
    exp_check.in[1] <== exp_value;
    signal not_expired <== exp_check.out;

    // Verify non-revocation
    component ver_sl_uri = VerifyClaimMembershipString(n_claims,
    6);
    ver_sl_uri.hashed_claims <== hashed_claims;
    ver_sl_uri.claim_name <== status_list_uri_name;
    ver_sl_uri.claim_value <== status_list_uri_value;

    component ver_sl_idx = VerifyClaimMembership(n_claims, 5);
    ver_sl_idx.hashed_claims <== hashed_claims;
    ver_sl_idx.claim_name <== status_list_idx_name;
    ver_sl_idx.claim_value <== status_list_idx_value;

    component status_check = StatusListCheck(512, 256, 2);
    status_check.status_list <== status_list;
    status_check.status_list_idx <== status_list_idx_value;
    signal status <== status_check.status;

    component revoked_check = IsZero();
    revoked_check.in <== status;
    signal not_revoked <== revoked_check.out;

    component m_and = MultiAND(3);
    m_and.in[0] <== valid_signature;
    m_and.in[1] <== not_expired;
    m_and.in[2] <== not_revoked;

    out <== m_and.out;
}
```

Listing B.5: Circom code verifying integrity and validity of a credential

Bibliography

[1] Christopher Allen. *The Path to Self-Sovereign Identity*. https://www.lifewithalacrity.com/article/the-path-to-self-soverereign-identity/. 2016.

[2] Scott Ames et al. "Ligero: Lightweight Sublinear Arguments Without a Trusted Setup". In: *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. CCS '17. Dallas, Texas, USA: Association for Computing Machinery, 2017, pp. 2087–2104. ISBN: 9781450349468. DOI: 10.1145/3133956.3134104. URL: https://doi.org/10.1145/3133956.3134104.

[3] Android. *Hardware-backed Keystore*. https://source.android.com/docs/security/features/keystore.

[4] Apple. *SecureEnclave*. https://developer.apple.com/documentation/cryptokit/secureenclave.

[5] EUDI Architecture and Reference Framework. *The Common Union Toolbox for a Coordinated Approach Towards a European Digital Identity Framework*. https://digital-strategy.ec.europa.eu/en/library/european-digital-identity-wallet-architecture-and-reference-framework.

[6] Arasu Arun, Srinath Setty, and Justin Thaler. *Jolt: SNARKs for Virtual Machines via Lookups*. Cryptology ePrint Archive, Paper 2023/1217. 2023. URL: https://eprint.iacr.org/2023/1217.

[7] Matthias Babel and Johannes Sedlmeir. "Bringing data minimization to digital wallets at scale with general-purpose zero-knowledge proofs". In: *ArXiv* abs/2301.00823 (2023). URL: https://api.semanticscholar.org/CorpusID:255394041.

[8] Carsten Baum et al. *Mac'n'Cheese: Zero-Knowledge Proofs for Boolean and Arithmetic Circuits with Nested Disjunctions*. Cryptology ePrint Archive, Paper 2020/1410. 2020. URL: https://eprint.iacr.org/2020/1410.

[9] Marta Bellés-Muñoz et al. "Circom: A Circuit Description Language for Building Zero-Knowledge Applications". In: *IEEE Transactions on Dependable and Secure Computing* 20.6 (2023), pp. 4733–4751. DOI: 10.1109/TDSC.2022.3232813.

[10] Michael Ben-Or et al. "Everything Provable is Provable in Zero-Knowledge". In: *Annual International Cryptology Conference*. 1990. URL: https://api.semanticscholar.org/CorpusID:38392736.

[11] Eli Ben-Sasson et al. *Aurora: Transparent Succinct Arguments for R1CS*. Cryptology ePrint Archive, Paper 2018/828. 2018. URL: https://eprint.iacr.org/2018/828.

[12] Eli Ben-Sasson et al. *Fast Reductions from RAMs to Delegatable Succinct Constraint Satisfaction Problems*. Cryptology ePrint Archive, Paper 2012/071. 2012. URL: https://eprint.iacr.org/2012/071.

[13] Eli Ben-Sasson et al. "Fast Reed-Solomon Interactive Oracle Proofs of Proximity". In: *45th International Colloquium on Automata, Languages, and Programming (ICALP 2018)*. Ed. by Ioannis Chatzigiannakis et al. Vol. 107. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2018, 14:1–14:17. ISBN: 978-3-95977-076-7. DOI: 10.4230/LIPIcs.ICALP.2018.14. URL: https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.ICALP.2018.14.

[14] Eli Ben-Sasson et al. *libSTARK*. https://github.com/elibensasson/libSTARK?tab=License-1-ov-file.

[15] Eli Ben-Sasson et al. *Scalable, transparent, and post-quantum secure computational integrity*. Cryptology ePrint Archive, Paper 2018/046. 2018. URL: https://eprint.iacr.org/2018/046.

[16] Eli Ben-Sasson et al. *SNARKs for C: Verifying Program Executions Succinctly and in Zero Knowledge*. Cryptology ePrint Archive, Paper 2013/507. 2013. URL: https://eprint.iacr.org/2013/507.

[17] Eli Ben-Sasson et al. *Succinct Non-Interactive Zero Knowledge for a von Neumann Architecture*. Cryptology ePrint Archive, Paper 2013/879. 2013. URL: https://eprint.iacr.org/2013/879.

[18] Ward Beullens and Gregor Seiler. *LaBRADOR: Compact Proofs for R1CS from Module-SIS*. Cryptology ePrint Archive, Paper 2022/1341. 2022. URL: https://eprint.iacr.org/2022/1341.

[19] Nir Bitansky et al. *From Extractable Collision Resistance to Succinct Non-Interactive Arguments of Knowledge, and Back Again*. Cryptology ePrint Archive, Paper 2011/443. 2011. URL: https://eprint.iacr.org/2011/443.

[20] 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. Chicago, Illinois, USA: Association for Computing Machinery, 1988, pp. 103–112. ISBN: 0897912640. DOI: 10.1145/62212.62222. URL: https://doi.org/10.1145/62212.62222.

[21] Dan Bogdanov et al. "ZK-SecreC: a Domain-Specific Language for Zero-Knowledge Proofs". In: *2024 IEEE 37th Computer Security Foundations Symposium (CSF)*. 2024, pp. 372–387. DOI: 10.1109/CSF61375.2024.00010.

[22] Jonathan Bootle et al. *Efficient Zero-Knowledge Arguments for Arithmetic Circuits in the Discrete Log Setting*. Cryptology ePrint Archive, Paper 2016/263. 2016. URL: https://eprint.iacr.org/2016/263.

[23] Sean Bowe, Ariel Gabizon, and Ian Miers. *Scalable Multi-party Computation for zk-SNARK Parameters in the Random Beacon Model*. Cryptology ePrint Archive, Paper 2017/1050. 2017. URL: https://eprint.iacr.org/2017/1050.

[24] Benjamin Braun et al. *Verifying Computations with State (Extended Version)*. Cryptology ePrint Archive, Paper 2013/356. 2013. DOI: 10.1145/2517349.2522733. URL: https://eprint.iacr.org/2013/356.

[25] Ernest F. Brickell et al. "Gradual and Verifiable Release of a Secret". In: *A Conference on the Theory and Applications of Cryptographic Techniques on Advances in Cryptology*. CRYPTO '87. Berlin, Heidelberg: Springer-Verlag, 1987, pp. 156–166. ISBN: 3540187960.

[26] Jeremy Bruestle, Paul Gafni, and the RISC Zero Team. *RISC Zero zkVM: Scalable, Transparent Arguments of RISC-V Integrity*. https://dev.risczero.com/proof-system-in-detail.pdf. 2023.

[27] Benedikt Bünz et al. "Bulletproofs: Short Proofs for Confidential Transactions and More". In: *2018 IEEE Symposium on Security and Privacy (SP)*. 2018, pp. 315–334. DOI: 10.1109/SP.2018.00020.

[28] Matteo Campanelli, Dario Fiore, and Anaïs Querol. *LegoSNARK: Modular Design and Composition of Succinct Zero-Knowledge Proofs*. Cryptology ePrint Archive, Paper 2019/142. 2019. DOI: 10.1145/3319535.3339820. URL: https://eprint.iacr.org/2019/142.

[29] Lily Chen et al. *Recommendations for Discrete Logarithm-based Cryptography: Elliptic Curve Domain Parameters*. Tech. rep. NIST Special Publication (SP) 800-186. Gaithersburg, MD: National Institute of Standards and Technology, 2023. DOI: 10.6028/NIST.SP.800-186.

[30] Alessandro Chiesa, Dev Ojha, and Nicholas Spooner. *Fractal: Post-Quantum and Transparent Recursive Proofs from Holography*. Cryptology ePrint Archive, Paper 2019/1076. 2019. URL: https://eprint.iacr.org/2019/1076.

[31] Alessandro Chiesa et al. *Marlin: Preprocessing zkSNARKs with Universal and Updatable SRS*. Cryptology ePrint Archive, Paper 2019/1047. 2019. URL: https://eprint.iacr.org/2019/1047.

[32] conroi. *Ligero Polynomial Commitment*. https://github.com/conroi/lcpc.

[33] Craig Costello et al. *Geppetto: Versatile Verifiable Computation*. Cryptology ePrint Archive, Paper 2014/976. 2014. URL: https://eprint.iacr.org/2014/976.

[34] Stephen Curran et al. *AnonCreds Specification*. https://hyperledger.github.io/anoncreds-spec/.

[35] dalek-cryptography. *bulletproofs*. https://github.com/dalek-cryptography/bulletproofs?tab=readme-ov-file.

[36] Ivan Damgård and Yuval Ishai. "Scalable Secure Multiparty Computation". In: *Advances in Cryptology - CRYPTO 2006*. Ed. by Cynthia Dwork. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 501–520. ISBN: 978-3-540-37433-6.

[37] George Danezis et al. *Square Span Programs with Applications to Succinct NIZK Arguments*. Cryptology ePrint Archive, Paper 2014/718. 2014. URL: https://eprint.iacr.org/2014/718.

[38] Swiss eID Dev Team. *Discussion Paper: Initial technological basis for the Swiss trust infrastructure*. https://github.com/e-id-admin/open-source-community/blob/main/discussion-paper-tech-proposal/discussion-paper-tech-proposal.md.

[39] Swiss eID Dev Team. *eidch-issuer-agent-oid4vci*. https://github.com/swiyu-admin-ch/eidch-issuer-agent-oid4vci.

[40] Swiss eID Dev Team. *Swiss e-ID and trust infrastructure: Interoperability profile*. https://github.com/e-id-admin/open-source-community/blob/main/tech-roadmap/swiss-profile.md.

[41] Benjamin E. Diamond and Jim Posen. *Polylogarithmic Proofs for Multilinears over Binary Towers*. Cryptology ePrint Archive, Paper 2024/504. 2024. URL: https://eprint.iacr.org/2024/504.

[42] Benjamin E. Diamond and Jim Posen. *Succinct Arguments over Towers of Binary Fields*. Cryptology ePrint Archive, Paper 2023/1784. 2023. URL: https://eprint.iacr.org/2023/1784.

[43] Leo Ducas et al. *CRYSTALS – Dilithium: Digital Signatures from Module Lattices*. Cryptology ePrint Archive, Paper 2017/633. 2017. URL: https://eprint.iacr.org/2017/633.

[44] Privacy & Scaling Explorations. *circom-ecdsa-p256*. https://github.com/privacy-scaling-explorations/circom-ecdsa-p256.git. 2021.

[45] Amos Fiat and Adi Shamir. "How to Prove Yourself: Practical Solutions to Identification and Signature Problems". In: *Advances in Cryptology - CRYPTO '86, Santa Barbara, California, USA, 1986, Proceedings*. Vol. 263. Lecture Notes in Computer Science. Springer, 1986, pp. 186–194. DOI: 10.1007/3-540-47721-7_12.

[46] Matteo Frigo and Abhi Shelat. *Anonymous credentials from ECDSA*. Cryptology ePrint Archive, Paper 2024/2010. 2024. URL: https://eprint.iacr.org/2024/2010.

[47] Ariel Gabizon, Zachary J. Williamson, and Oana Ciobotaru. *PLONK: Permutations over Lagrange-bases for Oecumenical Noninteractive arguments of Knowledge*. Cryptology ePrint Archive, Paper 2019/953. 2019. URL: https://eprint.iacr.org/2019/953.

[48] Rosario Gennaro et al. *Quadratic Span Programs and Succinct NIZKs without PCPs*. Cryptology ePrint Archive, Paper 2012/215. 2012. URL: https://eprint.iacr.org/2012/215.

[49] Oded Goldreich, Silvio Micali, and Avi Wigderson. "Proofs that yield nothing but their validity or all languages in NP have zero-knowledge proof systems". In: *J. ACM* 38.3 (July 1991), pp. 690–728. ISSN: 0004-5411. DOI: 10.1145/116825.116852. URL: https://doi.org/10.1145/116825.116852.

[50] 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. Providence, Rhode Island, USA: Association for Computing Machinery, 1985, pp. 291–304. ISBN: 0897911512. DOI: 10.1145/22145.22178. URL: https://doi.org/10.1145/22145.22178.

[51] Lorenzo Grassi et al. *On a Generalization of Substitution-Permutation Networks: The HADES Design Strategy*. Cryptology ePrint Archive, Paper 2019/1107. 2019. URL: https://eprint.iacr.org/2019/1107.

[52] Lorenzo Grassi et al. *Poseidon: A New Hash Function for Zero-Knowledge Proof Systems*. Cryptology ePrint Archive, Paper 2019/458. 2019. URL: https://eprint.iacr.org/2019/458.

[53] Jens Groth. *On the Size of Pairing-based Non-interactive Arguments*. Cryptology ePrint Archive, Paper 2016/260. 2016. URL: https://eprint.iacr.org/2016/260.

[54] Jens Groth. "Short Pairing-Based Non-interactive Zero-Knowledge Arguments". In: *Advances in Cryptology - ASIACRYPT 2010*. Ed. by Masayuki Abe. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 321–340. ISBN: 978-3-642-17373-8.

[55] Jens Groth. "Simulation-sound NIZK proofs for a practical language and constant size group signatures". In: *Proceedings of the 12th International Conference on Theory and Application of Cryptology and Information Security*. ASIACRYPT'06. Shanghai, China: Springer-Verlag, 2006, pp. 444–459. ISBN: 3540494758. DOI: 10.1007/11935230_29. URL: https://doi.org/10.1007/11935230_29.

[56] Jens Groth, Rafail Ostrovsky, and Amit Sahai. "New Techniques for Noninteractive Zero-Knowledge". In: *J. ACM* 59.3 (June 2012). ISSN: 0004-5411. DOI: 10.1145/2220357.2220358. URL: https://doi.org/10.1145/2220357.2220358.

[57] Jens Groth, Rafail Ostrovsky, and Amit Sahai. "Non-interactive zaps and new techniques for NIZK". In: *Proceedings of the 26th Annual International Conference on Advances in Cryptology*. CRYPTO'06. Santa Barbara, California: Springer-Verlag, 2006, pp. 97–111. ISBN: 3540374329. DOI: 10.1007/11818175_6. URL: https://doi.org/10.1007/11818175_6.

[58] Jens Groth and Amit Sahai. *Efficient Non-interactive Proof Systems for Bilinear Groups*. Cryptology ePrint Archive, Paper 2007/155. 2007. DOI: 10.1137/080725386. URL: https://eprint.iacr.org/2007/155.

[59] Jens Groth et al. *Updatable and Universal Common Reference Strings with Applications to zk-SNARKs*. Cryptology ePrint Archive, Paper 2018/280. 2018. URL: https://eprint.iacr.org/2018/280.

[60] Mike Hamburg. *Ristretto*. https://ristretto.group/.

[61] Iden3. *rapidnark: C++ implementation of zk-SNARKs*. https://github.com/iden3/rapidsnark. 2020.

[62] Iden3. *snarkjs: JavaScript implementation of zk-SNARKs*. https://github.com/iden3/snarkjs. 2020.

[63] iden3. *circom*. https://github.com/iden3/circom.

[64] iden3. *circomlib*. https://github.com/iden3/circomlib.

[65] Yuval Ishai et al. "Zero-knowledge from secure multiparty computation". In: *Proceedings of the Thirty-Ninth Annual ACM Symposium on Theory of Computing*. STOC '07. San Diego, California, USA: Association for Computing Machinery, 2007, pp. 21–30. ISBN: 9781595936318. DOI: 10.1145/1250790.1250794. URL: https://doi.org/10.1145/1250790.1250794.

[66] Michael B. Jones, John Bradley, and Nat Sakimura. *JSON Web Signature (JWS)*. RFC 7515. May 2015. DOI: 10.17487/RFC7515. URL: https://www.rfc-editor.org/info/rfc7515.

[67] Joe Kilian. "A note on efficient zero-knowledge proofs and arguments (extended abstract)". In: *Proceedings of the Twenty-Fourth Annual ACM Symposium on Theory of Computing*. STOC '92. Victoria, British Columbia, Canada: Association for Computing Machinery, 1992, pp. 723–732. ISBN: 0897915119. DOI: 10.1145/129712.129782. URL: https://doi.org/10.1145/129712.129782.

[68] Ahned Kosba. *xJsnark*. https://github.com/akosba/xjsnark.

[69] BROWN D. R. L. "SEC 2 : Recommended elliptic curve domain parameters". In: *Standars for Efficient Cryptography* (2010). URL: https://cir.nii.ac.jp/crid/1572824501046106880.

[70] Helger Lipmaa. *Progression-Free Sets and Sublinear Pairing-Based Non-Interactive Zero-Knowledge Arguments*. Cryptology ePrint Archive, Paper 2011/009. 2011. URL: https://eprint.iacr.org/2011/009.

[71] Helger Lipmaa. *Succinct Non-Interactive Zero Knowledge Arguments from Span Programs and Linear Error-Correcting Codes*. Cryptology ePrint Archive, Paper 2013/121. 2013. URL: https://eprint.iacr.org/2013/121.

[72] Tobias Looker, Paul Bastian, and Christian Bormann. *Token Status List*. Internet-Draft draft-ietf-oauth-status-list-09. Work in Progress. Internet Engineering Task Force, Mar. 2025. 70 pp. URL: https://datatracker.ietf.org/doc/draft-ietf-oauth-status-list/09/.

[73] Tobias Looker et al. *The BBS Signature Scheme*. Internet-Draft draft-irtf-cfrg-bbs-signatures-08. Work in Progress. Internet Engineering Task Force, Mar. 2025. 119 pp. URL: https://datatracker.ietf.org/doc/draft-irtf-cfrg-bbs-signatures/08/.

[74] Vadim Lyubashevsky, Gregor Seiler, and Patrick Steuer. *The LaZer Library: Lattice-Based Zero Knowledge and Succinct Proofs for Quantum-Safe Privacy*. Cryptology ePrint Archive, Paper 2024/1846. 2024. URL: https://eprint.iacr.org/2024/1846.

[75] o1-labs. *snarky*. https://github.com/o1-labs/snarky.

[76] Christian Paquin, Guru-Vamsi Policharla, and Greg Zaverucha. *Crescent: Stronger Privacy for Existing Credentials*. Cryptology ePrint Archive, Paper 2024/2013. 2024. URL: https://eprint.iacr.org/2024/2013.

[77] Bryan Parno et al. *Pinocchio: Nearly Practical Verifiable Computation*. Cryptology ePrint Archive, Paper 2013/279. 2013. URL: https://eprint.iacr.org/2013/279.

[78] Personaelabs. *spartan-ecdsa*. https://github.com/personaelabs/spartan-ecdsa.

[79] Pluto. *web-prover-circuits*. https://github.com/pluto/web-prover-circuits.

[80] Thomas Pornin. *Deterministic Usage of the Digital Signature Algorithm (DSA) and Elliptic Curve Digital Signature Algorithm (ECDSA)*. RFC 6979. Aug. 2013. DOI: 10.17487/RFC6979. URL: https://www.rfc-editor.org/info/rfc6979.

[81] Michael Rosenberg et al. "zk-creds: Flexible Anonymous Credentials from zkSNARKs and Existing Identity Infrastructure". In: *2023 IEEE Symposium on Security and Privacy (SP)*. 2023, pp. 790–808. DOI: 10.1109/SP46215.2023.10179430.

[82] Uuna Saarela. "Practical Zero-Knowledge within the European Digital Identity Framework: Implementing Privacy-Preserving Identity Checks". MA thesis. Aalto Univesity School of Electrical Engineering, 2024.

[83] scipr-lab. *libiop*. https://github.com/scipr-lab/libiop.

[84] scipr-lab. *libsnark*. https://github.com/scipr-lab/libsnark.

[85] Srinath Setty. *spartan*. https://github.com/microsoft/Spartan.

[86] Srinath Setty. *Spartan: Efficient and general-purpose zkSNARKs without trusted setup*. Cryptology ePrint Archive, Paper 2019/550. 2019. URL: https://eprint.iacr.org/2019/550.

[87] Srinath Setty, Justin Thaler, and Riad Wahby. *Unlocking the lookup singularity with Lasso*. Cryptology ePrint Archive, Paper 2023/1216. 2023. URL: https://eprint.iacr.org/2023/1216.

[88] IMDEA Software. *LegoSNARK*. https://github.com/imdea-software/legosnark.

[89] Manu Sporny, Dave Longley, and David Chadwick Ivan Herman. *Verifiable Credentials Data Model v2.0*. https://www.w3.org/TR/vc-data-model-2.0/. 2025.

[90] Starkware. *Cairo*. https://github.com/starkware-libs/cairo.

[91] Oliver Terbu, Daniel Fett, and Brian Campbell. *SD-JWT-based Verifiable Credentials (SD-JWT VC)*. https://www.ietf.org/archive/id/draft-ietf-oauth-sd-jwt-vc-04.html.

[92] TrustworthyComputing. *Zilch*. https://github.com/TrustworthyComputing/Zilch.

[93] Riad S. Wahby et al. *Efficient RAM and control flow in verifiable outsourced computation*. Cryptology ePrint Archive, Paper 2014/674. 2014. DOI: 10.14722/ndss.2015.23097. URL: https://eprint.iacr.org/2014/674.

[94] Ruihan Wang, Carmit Hazay, and Muthuramakrishnan Venkitasubramaniam. "Ligetron: Lightweight Scalable End-to-End Zero-Knowledge Proofs Post-Quantum ZK-SNARKs on a Browser". In: *2024 IEEE Symposium on Security and Privacy (SP)*. 2024, pp. 1760–1776. DOI: 10.1109/SP54263.2024.00086.

[95] Barry WhiteHat, Marta Bellés, and Jordi Baylina. *ERC-2494: Baby Jubjub Elliptic Curve*. https://eips.ethereum.org/EIPS/eip-2494. 2020.

[96] Tiancheng Xie et al. *Libra: Succinct Zero-Knowledge Proofs with Optimal Prover Computation*. Cryptology ePrint Archive, Paper 2019/317. 2019. URL: https://eprint.iacr.org/2019/317.

[97] Dan Yamamoto, Lovesh Harchandani, and Oleg Nosov. *Composite proof system*. https://github.com/docknetwork/crypto.

[98] Dan Yamamoto, Yuji Suga, and Kazue Sako. "Formalising Linked-Data based Verifiable Credentials for Selective Disclosure". In: *2022 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)*. 2022, pp. 52–65. DOI: 10.1109/EuroSPW55150.2022.00013.

[99] Jiaheng Zhang et al. *Transparent Polynomial Delegation and Its Applications to Zero Knowledge Proof*. Cryptology ePrint Archive, Paper 2019/1482. 2019. URL: https://eprint.iacr.org/2019/1482.

[100] Yupeng Zhang et al. "vRAM: Faster Verifiable RAM with Program-Independent Preprocessing". In: *2018 IEEE Symposium on Security and Privacy (SP)*. 2018, pp. 908–925. DOI: 10.1109/SP.2018.00013.