POLITECNICO DI TORINO

Master Degree Course in Computer Engineering

Master Degree Thesis

Privacy-Preserving Credentials for Self-Sovereign Identity with BBS+ Signatures

Supervisor
prof. Antonio Lioy

Candidate
Alessandro Guggino

Tutor
LINKS Foundation
Andrea Vesco, Ph.D.

Academic year 2021-2022

Summary

This document describes a new class of credentials focused on privacy, called privacy-preserving or anonymous credentials, based on BBS+ signatures. The main features needed to be such are data minimization and unlinkability, which is the ability to be non-correlatable when presented. BBS+ signatures offer these properties because they are multi-message signatures that allow a credential holder to select which attributes to disclose when generating an unlinkable zero-knowledge proof of knowledge of the credential and its signature. Furthermore, the designed and developed system has also the capabilities to bind credentials to their holders and to revoke credentials, all in a private manner. These privacy-preserving credentials can be perfectly integrated into the Self-Sovereign Identity model and the W3C standard of Verifiable Credentials, which enable humans, organizations, and things to have complete control of their identity.

Acknowledgements

I would like to thank my supervisor, professor Antonio Lioy, along with the LINKS Foundation researchers, Davide Margaria, Alberto Carelli, and especially Andrea Vesco, who has been an important mentor to me from my internship during my bachelor's degree. Also, thanks to Simone Dutto for his courtesy and help with some mathematical stuff.

I would like to dedicate this thesis to the closest people around me.
To my family.
To Ari.
To my friends, Davi, Dello, Giordi, Jack, Lele, Luci, Marti, Olek, and Ole.

Contents

Chapter 1

Introduction

"On the Internet, nobody knows you're a dog", says an iconic cartoon published by The New Yorker in 1993 [1]. The author of the cartoon, Peter Sneier, got the point of one of the Internet's biggest open issues. The Internet does not have an identity layer, a way to understand and verify who or what you are connecting to [2], because it was designed and developed to interconnect machines, for which a network-level identifier (i.e. the IP address) was enough, initially. But that does not add anything about the person, organization, or thing responsible for the machine that is communicating [3]. Also, the concept of trust was not undertaken, because the designers of the original architecture of the Internet knew and trusted each other [4]. Everything changed when the Internet opened up to the masses, gaining exponential success, directly proportional to the growth of its problems regarding security, privacy, and trust.

Digital interactions require trust between the parties involved. The introduction of digital identity can support gaining trust and thus limit cybercrime problems and attacks such as *shadow server*, *connection hijacking*, and *phishing*. These problems have one thing in common: their countermeasure. To mitigate them, strong authentication must be enforced. A digital identity solution must also focus on privacy: it is crucial to verify and authenticate the party with whom one is communicating, but it is also important to minimize data shared and not learn unnecessary details. In addition, the party should not be traceable when presenting its identity.

In this document, self-sovereign identity and verifiable credentials are introduced. In order to achieve **privacy-preserving credentials**, zero-knowledge proofs and BBS+ signatures have been integrated into verifiable credentials.

The topics are organized as follows:

- Chapter 1 is the current introduction.
- Chapter 2 covers digital identity and self-sovereign identity.
- Chapter 3 addresses zero-knowledge proofs.
- Chapter 4 is about the BBS+ signature scheme and its building blocks.

- Chapter 5 discusses the designed and developed anonymous credential system.
- Chapter 6 draws conclusions about this work.
- Appendix A contains the user's manual for installing and running the developed code.
- Appendix B contains the developer's manual to learn the technical details of the code.

Chapter 2

Digital Identity

2.1 What is a digital identity?

A digital identity is a set of claims, i.e., assertions of the truth of some attributes, about a digital subject, which is a person, an organization, or a thing existing in the digital ecosystem that is being described, made by another digital subject or by themself [2]. This set of claims may be stored in one or more digital credentials. Furthermore, an identity may collect attributes that can be categorized into three broad groups [5]:

- **Inherent attributes**. They are intrinsic to a subject and are not defined by relationships to external subjects. E.g., date of birth (for individuals), industry (for organizations).
- **Accumulated attributes**. They are received or developed and stacked over time, during which they may update. E.g., university degrees (for individuals), business records (for organizations), ownership history (for things).
- **Assigned attributes**. They are associated with the subject but are not related to its intrinsic nature. E.g., telephone number (for individuals), identifying number (for things).

In a digital identity ecosystem there are specific roles with defined functions [5]:

- **User**. It is the entity for which the system provides the identity, the subject.
- **Identity provider (IdP)**. It is the entity that has users' attributes, attests to their truthfulness, and gives the users identity. It may perform operations involving the users' identity on their behalf.
- **Relying party (RP)**. It is the entity that receives attestations from identity providers or users to verify. If the verification is successful, then it grants access to the users.

These parties interact with each other and form the foundation of the ecosystem in which digital identity is shaped.

A good digital identity system, capable of being widely used on an Internet-scale scenario, must comply with some fundamental laws [2]. The identity system must only disclose information identifying a user with the user's consent and must disclose the least amount of it. Moreover, the identifying information must be revealed only to parties with a necessary and justifiable need. The identity system must support both "omnidirectional" identifiers, which are public, invariant, and well-known and "unidirectional" identifiers, which are private and used just in a single relationship, to avoid correlation.

During an interaction that needs to identify the two parties involved, which do not have an already existing connection, there will be an *identity transaction* composed of three main aspects [5]:

- **Authorization**. The relying party determines the requirements for transaction eligibility and requests certain attributes.
- **Attributes**. The subject presents its proof of attributes in the response.
- **Authentication**. The relying party determines whether the attributes match the rules and policies defined, while, in addition, it verifies the authenticity of the attributes. If both checks are successful, the interaction can proceed.

An interaction for which the identity transaction described above is needed may require different *Levels of Assurance (LoA)*. A level of assurance [6] is a degree of confidence that parties have in the truthfulness of the identity being presented. To determine the assurance level required, three main elements must be considered:

- **Enrolment**. How the process for obtaining the identity is accomplished. This is related to the application and registration phase.
- **Management**. How the identity is managed and designed. This is about how many and which authentication factors are used:
  - knowledge-based, e.g., a password;
  - possession-based, e.g., a cryptographic key;
  - inherent, e.g., biometric data.

  In addition, it refers to specifications for issuance, delivery, activation, suspension, revocation, reactivation, renewal, and replacement.
- **Authentication**. How the authentication through the identity is performed. This is focused on the strength of the authentication mechanisms and the threats associated with them.

From these elements, three levels of assurance are extracted: low, substantial, and high. The digital identity is necessary to establish and reinforce the level of trust between participants in a relationship, even with a low level of assurance.

2.2 Identity models

Concerning trust, it is possible to abstract several Internet trust models which will be helpful to list the main identity models [7].

- **Sole Source**. A party issues identities and only trusts identities issued by itself, acting as a service provider (relying party) and as an identity provider. This single-party defines governance, privacy, and technical aspects for all participants. It may be used when the service provider has confidential information or valuable assets, for which a high level of assurance is needed. On the other hand, the service provider takes the management cost of the identity life cycle and the user is forced to create a whole new identity to access the service offered.

- **Peer-to-Peer Identity**. No central identity provider or governance agreement is present, each peer has the same powers, asserts its own identity, and decides who to trust. This model is very flexible and if the security requirements are low, it can grow very large.

- **Centralized Token Issuance, Distributed Enrolment**. A trusted central provider provides identity hardware tokens which are used by the subject to enroll in service providers. This can achieve strong authentication and provide a high level of assurance but it is expensive, complex and dependent upon a trusted third party.

- **Pairwise Agreement**. Two parties trust identities issued by one another. It may be highly customizable but time-consuming, complex, and expensive to negotiate.

- **Three-Party Model**. An identity provider issues identities to both the requester and service provider, which must agree to trust the same identity provider (trusted third party).

- **Federation**. It provides a set of contracts that allow service providers to recognize identities issued by one another, without having to negotiate individual agreements with every party. It can scale very high but it cannot be customized.

- **Four-Party Model**. The requester of the service and the service provider can use different identity providers. In this model, the service provider will ask to check the identity of the requester to its identity provider, which will then ask the requester's identity provider.

- **Individual Contract Wrappers**. The requester provides information to a service provider together with the terms for how that information can be used.

- **Open Trust Frameworks**. The specification is publicly available and describes a set of identity proofing, security, and privacy policies. It is authored by experts with the intent that compliance can be assessed. The identity provider may implement a Trust Framework, which may be verified by an assessor.

From the work cited above [7], it is possible to extract three main digital identity models that sum up the different architectures available nowadays [3]: the centralized identity, the federated identity, and the decentralized identity.

2.2.1 Centralized identity model

It is the first model, the original form of the internet identity, longly used and still in use today. In this model, the user registers an account with a service, establishing an identity only valid for the relationship. The service provider is the owner of the user's identity and all the data about the user belongs to the service provider, outside the user's control, even deleting the account. In addition, there are many problems including the user load of managing usernames and passwords, the differences in security and privacy policies in every service, and the presence of centralized databases of personal data. Because the identity is just for the relationship, which is good to enhance both security and privacy as long as usernames and passwords are never reused, none of the user's identity data is portable or reusable. Looking at trust models, it is possible to place it in the Sole Source model, where the identity provider and the relying party are the same party.

2.2.2 Federated identity model

The idea of the second model is to add an identity provider as a third party in the middle of the relation between the user and the service provider. In this way, it is possible to have a single account with the identity provider which can be used to log the user in and share basic identity data with any service that uses that identity provider. The federation is the set of all the services that use the same identity provider, while every service provider is the relying party. Because of this, it is possible to obtain the single sign-on (SSO) feature, allowing a user to log in with a single identity to any of several related services. It simplifies authentication by reducing usernames and passwords, and by improving the user's experience. The issues from the federated model come because the identity providers have to work with many service providers, and so they have to lower their security and privacy common policies. Due to that, these identity providers cannot be used to share valuable personal data in high-trust environments. Besides, the most used identity providers become some of the biggest targets for cybercrime. Since there is not one identity provider that works with every service, users need accounts with multiple identity providers, which are not portable like centralized identity accounts. Also, this model does not work well for organizations or things, because it is designed for individuals.

2.2.3 Decentralized identity model

The third, new model is designed starting from the paradigm introduced by Distributed Ledger Technologies (DLT), e.g., blockchain, which wants to remove the unnecessary third parties. Hence, this model is no longer based on accounts but it is based on a direct relationship between two parties acting as peers in a peer-to-peer private connection. The connection is shared and not owned by anyone, and it is decentralized because any peer is able to connect to any other, without any external help required. The distributed ledger is employed to store the public keys and to build a Decentralized Public Key Infrastructure (DPKI). The public keys are used to verify the signature on the digital identity credentials that peers exchange to provide proof of their identity. These credentials define and prove at least four important things:

- Who or what is the issuer;
- To whom or what it was issued;
- Whether it has been altered;
- Whether it has been revoked.

They can be issued and signed by any individual, organization, or thing and used anywhere they are trusted. Also, they may be self-issued and self-signed. With this model, the identity is as strong as the credentials it holds, so it may be employed in high-trust environments. It has the same flexibility and scalability as the Peer-to-Peer Identity, while it is close to the concepts introduced by the Individual Contract Wrappers and the Open Trust Frameworks. The decentralized identity is also called Self-Sovereign Identity (SSI) because the user is in control of their own identity, which is interoperable and portable.

2.3 Self-Sovereign Identity

Self-Sovereign Identity (SSI) is a solution built upon two major W3C (World Wide Web Consortium) specifications: Decentralized Identifiers (DIDs) [8] and Verifiable Credentials (VCs) [9]. Having two well-written and reviewed standards by W3C is the key to obtaining interoperability and making this technology mature. The SSI core concept is that the information regarding a user's identity must be controlled by the user. To achieve this, decentralized identifiers allow them to handle their cryptographic keys, while verifiable credentials introduce mechanisms to control the flow of information.

Both the DIDs and VCs definitions under the W3C are a standard, currently marked as a "Recommendation". This means that:
"It is a specification or set of guidelines or requirements that, after extensive consensus-building, has received the endorsement of W3C. It has been formally reviewed by

W3C Members, software developers, W3C Groups, and interested parties. W3C recommends the wide deployment of a Recommendation as a standard for the Web." [10]

2.3.1 Decentralized Identifiers (DIDs)

DIDs [8] are identifiers that refer to a subject determined by the controller of the DID, which may be the subject itself or someone else. DIDs are decoupled from centralized registries, identity providers, and certificate authorities. They are designed to be generated by their controller, which may be an individual, an organization, or a thing. Other parties may be involved to perform utility functions such as retrieving information from a DID, but the controller of a DID is able to prove control over it without requiring permission from anyone. Each controller (or subject) may have multiple DIDs to separate its identities and interactions so that the use of a DID is scoped to a specific context. Therefore, a DID may be "omnidirectional" or "unidirectional".

The DID is a URI (Uniform Resource Identifier) [11] that points to a DID document associated with a DID subject and stored on a verifiable data registry, such as a distributed ledger, a decentralized file system, or a distributed database. The DID document contains cryptographic material, such as public keys, and verification methods or services, which allow the DID controller to prove control of the DID and to enable trusted interactions. It may be encoded in JavaScript Object Notation (JSON) [12] or JSON-LD [13], which is a JSON-based format used to serialize Linked Data, a pattern for hyperlinking machine-readable data sets to each other, with the `@context` property. The DID methods are the mechanism to create, read (resolve), update, and deactivate (revoke) the DID and its associated DID document by interacting with their verifiable data registry.

Eventually, a DID is a text string composed of three parts divided by colons:

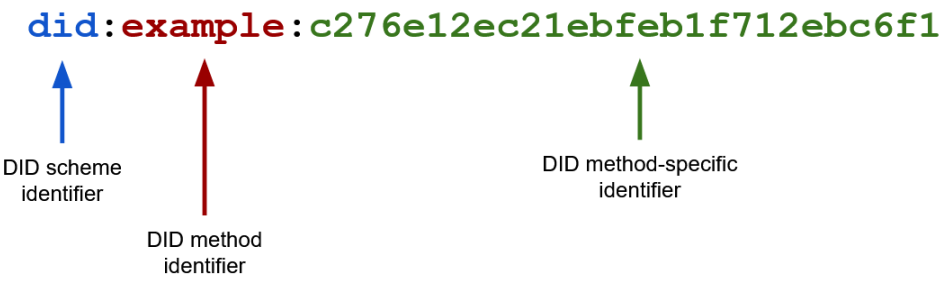


Figure 2.1. An example of a DID string.

- The DID URI scheme identifier;
- The DID method identifier;

- The DID method-specific identifier.

E.g., `did:example:c276e12ec21ebfeb1f712ebc6f1`.

While an example of a DID document associated with the above DID is the following JSON-LD document:

```
{
 "@context": [
  "https://www.w3.org/ns/did/v1",
  "https://w3id.org/security/suites/ed25519-2020/v1"
 ],
 "id": "did:example:c276e12ec21ebfeb1f712ebc6f1",
 "authentication": [{
  "id": "did:example:c276e12ec21ebfeb1f712ebc6f1#keys-1",
  "type": "Ed25519VerificationKey2020",
  "controller": "did:example:c276e12ec21ebfeb1f712ebc6f1",
  "publicKeyMultibase":
     "zH3C2AVvLMv6gmMNam3uVAjZpfkcJCwDwnZn6z3wXmqPV"
 }]
}
```

Listing 2.1. Example of a DID document in JSON-LD format from [8]

Where the `id` property denotes the DID of the DID subject, and the `authentication` object describes the verification method, a set of parameters useful to verify a proof. Inside of it, there are its identifier in `id`, the DID of the DID controller in `controller`, the type of the key in `type`, which defines the cryptographic algorithm where the key will be used, and then the actual public key in `publicKeyMultibase`.

2.3.2 Verifiable Credentials (VCs)

Verifiable Credential [9] is a data model that provides a way to express every sort of credential on the web in a cryptographically secure, privacy-respecting, and machine-verifiable manner. They allow any entity to say anything about any other entity, because of their extensibility. A verifiable credential can represent the same information that a physical credential represents but is encoded in JSON or JSON-LD format. The information consists of:

- Information related to identifying the subject of the credential;
- Information related to the issuing authority;
- Information related to the type of credential;
- Information related to specific attributes or properties being asserted by the issuing authority about the subject;

- Information related to constraints on the credential (e.g., expiration date, terms of use).

Verifiable credentials are tamper-evident and trustworthy because they include digital signatures done by issuers. Holders of verifiable credentials can generate Verifiable Presentations (VPs) to wrap the VCs and then share them with verifiers to prove they possess credentials with certain characteristics. They are called "verifiable" because any verifier is able to verify them, thanks to the digital signature attached. The verifiability of a credential does not imply the veracity of claims contained, which must come from a trusted issuer.

The verifiable credentials ecosystem is composed of the so-called *trust triangle* [4], which involves three primary roles:

- **Issuers**. They are the source of credentials. They assert claims about one or more subjects, creating a VC from these claims, and transmitting the VC to a holder. In the general digital identity ecosystem, they may be defined as identity providers.
- **Holders**. They request credentials from issuers, hold them in their storage, and present them when requested by verifiers. Usually, they are the subject of the VC. In the general digital identity ecosystem, they are the users.
- **Verifiers**. They want trust assurance about the holder of a credential. They request the credentials they need and then follow their policy to verify the credential's authenticity and validity. In the general digital identity ecosystem, they are the relying parties.

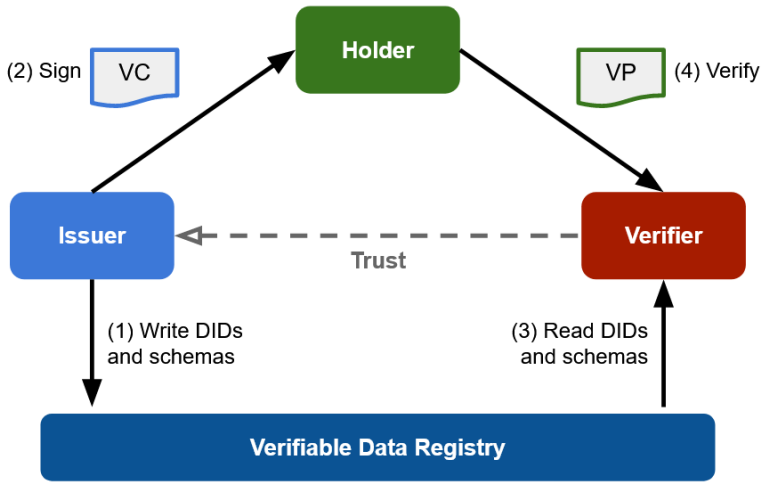


Figure 2.2. The verifiable credentials trust triangle from [4].

A credential can be used to establish trust only if the verifier has some degree of trust in the issuer and its policies, processes, and technologies. In the verifiable credentials trust triangle, a verifiable data registry is included to store public keys and credentials schemas. Firstly, the issuer writes a DID together with its public key to a verifiable data registry. Secondly, the issuer digitally signs a verifiable credential with its private key and issues it to a holder. This process does not involve any interaction with a verifiable data registry, so it is confidential between the issuer and the holder. Thirdly, a verifier requests a digital proof of one or more credentials from the holder, which generates and sends them. In the final, fourth step, the verifier gets the issuer's public key from the verifiable data registry and verifies the proof. Even if the self-sovereign identity technology stack makes use of DIDs and VCs, W3C-compliant verifiable credentials do not require decentralized identifiers. The issuer may also use conventional PKI-based digital certificates.

Examples

A simple example of a verifiable credential is the following JSON-LD document:

```
{
 "@context": [
  "https://www.w3.org/2018/credentials/v1",
  "https://w3id.org/security/suites/ed25519-2020/v1"
 ],
 "id": "https://example.edu/credentials/1872",
 "type": ["VerifiableCredential", "AlumniCredential"],
 "issuer": "did:example:c276e12ec21ebfeb1f712ebc6f1",
 "issuanceDate": "2022-06-01T19:23:24Z",
 "expirationDate": "2023-06-01T19:23:24Z",
 "credentialSubject": {
  "id": "did:example:ebfeb1f712ebc6f1c276e12ec21",
  "name": "John Doe",
  "alumniOf": {
   "id": "did:example:c276e12ec21ebfeb1f712ebc6f1",
   "name": "Example University",
   }
 },
 "proof": {
  "type": "Ed25519Signature2020",
  "created": "2022-06-01T19:23:25Z",
  "proofPurpose": "assertionMethod",
  "verificationMethod":
     "did:example:c276e12ec21ebfeb1f712ebc6f1#keys-1",
  "proofValue": "zeEdUoM7m9cY8ZyTpey83yBKeBcmcvbyrEQzJ19rD2[...]"
 }
}
```

Listing 2.2. Example of a Verifiable Credential in JSON-LD format from [9]

Where `@context` establishes the special terms that will be used in the JSON-LD document, such as `issuer` and `alumniOf`, using the static data available at the listed URIs. The `id` field specifies the unambiguous identifier for the credential as URI, `type` declares what data to expect in the credential through interpretation of the `@context` property, `issuer` defines the entity that issued the credential as URI. The remaining metadata are `issuanceDate` and `expirationDate`, which specify when the credential was issued and when the credential will expire. Then there is the `credentialSubject` object, which contains the claims about the subject of the credential: `id`, `name`, and `alumniOf`. Attached at the end there is the digital proof that makes the credential tamper-evident. The `proof` object contains its `type`, which specifies the cryptographic signature suite used to generate the signature, the signature creation date under `created`, the purpose of the proof under `proofPurpose`, the `verificationMethod` to identify the public key to verify the signature, and finally the digital signature value under `proofValue`.

Whilst a simple example of a verifiable presentation of the previous verifiable credential is the following JSON-LD document:

```
{
 "@context": [
  "https://www.w3.org/2018/credentials/v1"
 ],
 "type": "VerifiablePresentation",
 "verifiableCredential": [{
      ...
 }],
 "proof": {
  "type": "RsaSignature2018",
  "created": "2022-06-24T21:19:10Z",
  "proofPurpose": "authentication",
  "verificationMethod":
     "did:example:ebfeb1f712ebc6f1c276e12ec21#keys-1",
  "challenge": "1f44d55f-f161-4938-a659-f8026467f126",
  "domain": "4jt78h47fh47",
  "proofValue": "eyJhbGciOiJSUzI1NiIsImI2NCI6ZmFsc2UsIm[...]"
 }
}
```

Listing 2.3. Example of a Verifiable Presentation in JSON-LD format from [9]

Where the verifiable credential to present is in `verifiableCredential` and the digital signature of the presentation in `proof` contains the fields `challenge` and `domain` to protect against replay attacks.

Authorization

Verifiable credentials can be used to implement Role-Based Access Controls (RBACs) and Attribute-Based Access Controls (ABACs). These authorization layers can rely

on subject identification provided by verifiable credentials as a means of authorizing subjects to access resources.

In the Role-Based Access Control [14], access permissions are associated with roles and users are made members of appropriate roles within the system. Every user who holds the same role has the same set of permissions. This approach to access control is heavy to manage because capabilities must be associated directly with users or their roles or groups. Also, roles are often insufficient in the expression of complex access control policies. An alternative is the Attribute-Based Access Control [15], which grants or denies access requests by evaluating arbitrary attributes or various combinations of a set of attributes. The ABAC methodology allows for a more granular and flexible access control policy, two features that fit verifiable credentials very well.

2.3.3 Design goals

The design goals for decentralized identifiers and verifiable credentials, which are the pillars of self-sovereign identity, are collected in the DIDs specification [8] and summarized as follows:

- **Decentralization**. Remove the requirement for centralized authorities and single points of failure in identifiers management.
- **Control**. Give entities the power to control their identifiers without depending on external authorities.
- **Privacy**. Enable entities to handle the privacy of their information.
- **Security**. Enable sufficient security to rely on DID documents and verifiable credentials for the required level of assurance.
- **Proof-based**. Enable DID controllers and VCs issuers and holders to generate cryptographic proofs.
- **Discoverability**. Let entities discover DIDs for other entities.
- **Interoperability**. Use interoperable standards to scale globally.
- **Portability**. Be independent of any system.
- **Simplicity**. Make technology easy to understand, develop, and deploy.
- **Extensibility**. Enable the possibility to extend the data models without compromising interoperability, portability, and simplicity.

2.4 Privacy-related issues

2.4.1 Privacy

The approach of self-sovereign identity and DIDs and VCs standards described so far may be harmful to the user's privacy. The holder of a credential has to share the complete credential with the verifier, which may learn much more information than necessary.

First of all, it is important to use mechanisms that protect the data while storing and transporting VCs, such as TLS [16] to protect data while in transit as well as encryption or data access control to protect data while at rest. Also because verifiable credentials often contains personally identifiable information (PII) stored in the `credentialSubject` field.

The best practice for preventing privacy violations is to limit the disclosure of information to the minimum necessary. This practice follows the principle of data minimization [17], which can be expanded into three types:

- **Content minimization**. The amount of shared data should be strict to the minimum necessary in order to successfully accomplish the task.
- **Temporal minimization**. The data should be stored for the least amount of time necessary to execute the task.
- **Scope minimization**. The data should only be used for the strict purpose of the active task.

Besides, another important privacy enhancement related to data minimization is selective disclosure. Selective disclosure [17] is the ability of a holder to granularly decide what information to share. There are different solutions for selective disclosure of verifiable credentials [18]:

- **Just in time issuance**. Contact the issuer at request time to get a specific assertion. The infrastructure burden is proportional to the number of identities issued by the issuer, which must be highly available to handle it.
- **Trusted witness**. Use a trusted witness between the holder and the verifier. The witness receives the information and presents an assertion with only the information required by the verifier. This model requires a highly available and highly trusted third party.
- **Cryptographic solutions**. Use a cryptographic algorithm to disclose a subset of information to a larger assertion. The most common approach is using *Zero-Knowledge Proofs*, which will be explained in the next chapter of this document and will be employed to achieve privacy-preserving credentials.

After the activation of data minimization policies and selective disclosure, progressive trust enhancement can be applied. Progressive trust [17] is the procedure for gradually increasing the amount of data revealed as the communication proceeds and the trust is built. Hence, it is an escalation of data minimization and selective disclosure, proportional to the trust increase.

2.4.2 Unlinkability

Linkability is the ability to link data from multiple interactions to a single user. The interactions involve DIDs and verifiable credentials, and the user is the holder of these credentials. Linkability may also be called correlation and it can be performed by a verifier, by issuers and verifiers colluding, or by a third party observing the network. It is a way to collect data about a holder without their consent.

In certain SSI systems, unlinkability is a desired property. Unlinkability [19] ensures that a user may use multiple times resources or services without others being able to link these uses together. From an attacker's perspective, two actions made by users with the same anonymity set are unlinkable for him if the probability that these two actions are made by the same user is sufficiently close to $\frac{1}{N}$ with N number of users within the system [20]. Data minimization and selective disclosure may reduce correlation, by sharing only the information required to complete a transaction. A way, which can be accomplished through data minimization directly from the issuer or selective disclosure done by the holder, is to make each interaction unique by removing unique identifiers from credentials. Subjects of VCs are identified using the `credentialSubject.id` field, which creates a great risk of correlation, even greater when the identifiers are long-lived or used in multiple credentials. Similarly, the credential identifier in `id` allows malicious parties to correlate the holder. Another way to avoid linkability through identifiers is to use pairwise DIDs that are unique to each relationship and issue single-use verifiable credentials. With this solution, a holder could still be correlated by means of information in other attributes.

Furthermore, a verifiable credential holder may be linked by means of the signature value in the `proof` field. The properties in this field create a risk of correlation when the same values are used across more than one session or domain, and the value does not change. To solve this problem, it is possible to generate a random proof from a signature, to avoid signature-based correlation. This property will be further explored later in this document.

Finally, an unlinkability-related issue is that which concerns the revocation of verifiable credentials. Since credentials are claims made by an authority, the issuer should be able to revoke them. A revoked credential notifies verifiers that the credential should not be accepted. Classical revocation methods require verifiers to check the unique identifier of the credential against a public revocation list, which means, again, linkability. Regarding this topic, there will be a dedicated section in this document.

2.4.3 Binding

If an adversary gets an unprotected verifiable credential when in transit or at rest, then they will be able to reuse it in a *replay attack* and gain authorization to access a resource. A replay attack [21] is an attack in which the attacker is able to replay previously captured messages (in this case, credentials) to masquerade as one of the parties involved. Thereby, the attacker may produce an unauthorized effect or gain unauthorized access.

To achieve replay resistance, the VCs standard includes a holder identifier in the `credentialSubject.id` field. Then, having the holder DID within the credential, it is possible to bind them together and verify the binding with a challenge-response protocol through the verifiable presentation and its `proof` object. A challenge-response protocol [21] is an authentication protocol where the verifier sends the user a challenge (random value or nonce) that the user combines with a secret by applying a private key operation to the challenge to generate a response that is sent to the verifier. The verifier can verify the response by performing a public key operation on the response and establishing that the user possesses and controls the secret. The drawback is that the presence of unique identifiers inside a verifiable credential that may be exchanged multiple times with different verifiers leads to linkability, as discussed in the previous section.

The final solution for binding a credential to its holder is private holder binding. Holder binding is the process of showing that a credential was issued to a particular holder, while private holder binding allows holders to prove that credentials were issued to them without revealing anything else. Thus, the latter does not create a correlating factor for the holder that needs to be revealed during the presentation. This solution is implemented with a mechanism called *Linked Blinded Secret*, which will be explained in its own section.

Chapter 3

Zero-Knowledge Proofs

3.1 What is a ZKP?

A Zero-Knowledge Proof (ZKP) [22] is a cryptographic primitive which allows a prover to convince a verifier that a statement is true, without revealing any other information. It is a theorem-proving procedure to communicate a probabilistic proof. Having an *n-bit* long statement, a verifier may be erroneously convinced of its correctness with a very low probability, like $\frac{1}{2^n}$, and rightfully convinced of its correctness with a very high probability, like $1 - \frac{1}{2^n}$. To verify the correctness of a statement, the verifier of the proof must interactively ask questions and receive answers from the prover. Hence, these proofs are called "interactive". Moreover, it is possible to prove a theorem without communicating additional knowledge, because adding interaction to the process decreases the amount of knowledge that must be provided to prove a theorem. To summarize, a zero-knowledge proof must satisfy three properties:

- **Completeness**. If the statement is true, the honest prover must be able to convince the verifier.
- **Soundness**. If the statement is false, the malicious prover must not be able to convince the verifier that the statement is true, except for a negligible probability.
- **Zero-knowledge**. The verifier must not learn any information from the proof, except that the statement is true.

Furthermore, there are two types of statements that may be proved in zero-knowledge:

- Statements about the truthfulness of facts, e.g., "a specific graph has a three coloring";
- Statements about personal knowledge, e.g., "I know a three coloring for this graph".

The proof on the second type of statement is called ***proof of knowledge*** [23], which proves that the prover knows the secret information about the statement. This secret information is called *witness*.

3.1.1 Formal definitions

In this section, the properties of a proof system such as completeness, soundness, and zero-knowledge, are defined rigorously and mathematically. These definitions are expressed in the paper by Goldwasser, Micali, and Rackoff [22] and simplified in [24].

A model of computation is needed to formally express what an interaction between two computing machines is. In cryptography, it is often used the Turing machine.

Interactive Turing Machine

An interactive Turing machine (ITM) [22] is a multi-tape Turing machine. The tapes are a read-only input tape, a read-only random tape, a read-and-write work tape, a write-only output tape, a pair of communication tapes (one read-only and one write-only), and a read-and-write switch tape. In addition, it is associated with a single bit, called identity. The ITM is active if the content of its switch tape is equal to its identity, otherwise, the ITM is idle. While in idle mode, the machine's state, the heads on tapes, and the contents of tapes are not modified. The content of the input tape is called input, the content of the random tape is called random input, the content of the output tape is called output, and the contents of the communication tapes are called message sent and message received.

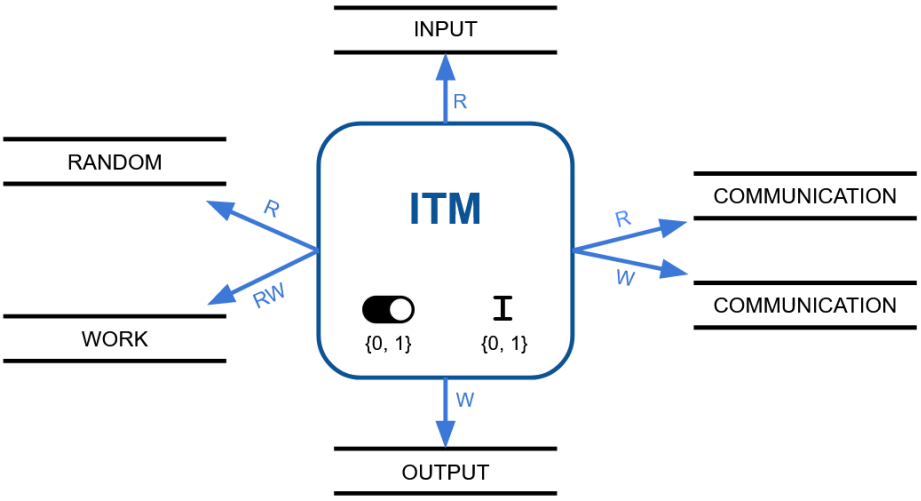


Figure 3.1. An interactive Turing machine from [22].

Now, there are two ITMs: a machine M_1 which is the prover P, and a machine M_2 which is the verifier V. The machines need to understand a language $\mathcal{L}$ to properly interact, thus the statement that the prover P wants to prove must be encoded in a particular language $\mathcal{L}$. A language $\mathcal{L}$ is a set of strings over some finite alphabet with some rules. During the setup phase of a communication protocol, the interacting parties agree upon a specific language, if it was not defined yet.

(x, w) is a proof statement, where x is a public value and w (the witness) is private and only known by the prover P. If $(x, w) \in \mathcal{L}$, the proof is correct, while if $(x, w) \notin \mathcal{L}$, the proof is incorrect.

Interactive Proof System

A pair of interactive Turing machines (P, V) is an interactive proof system for a language $\mathcal{L}$ if the prover P has infinite power, the verifier V is polynomial-time, and the following conditions hold:

- **Completeness**. For all (x, w) in language $\mathcal{L}$, the probability that a prover P, which knows x and w, can convince an honest verifier V, which knows x, is significant (non-negligible).

$$\forall (x, w) \in \mathcal{L} : Probability[\langle P(x, w), V(x)\rangle = 1] \geq 1 - negligible$$

- **Soundness (Unforgeability)**. For all (x, w) not in language $\mathcal{L}$, the probability that a cheating prover P', which knows only x, can convince an honest verifier V, which knows x, is negligible.

$$\forall (x, w) \notin \mathcal{L} : Probability[\langle P'(x), V(x)\rangle = 1] \leq negligible$$

An algorithm runs in *polynomial time* if to compute a solution it performs a number of steps bounded by a polynomial function of n, $p(n)$, where n is the length of the algorithm's input [25].

Proof of knowledge protocols must satisfy an additional property:

- **Soundness (Knowledge extractability)**. For all (x, w) in language $\mathcal{L}$, there exists a polynomial-time algorithm E (the *extractor*) such that the witness w can be extracted from interactions between prover P and verifier V.

$$\forall (x, w) \in \mathcal{L}, \exists E :$$
$$Probability[E\langle P(x, w), V(x)\rangle = w] \geq 1 - negligible$$

The prover P, which knows the witness w, must have used it in a hidden way during the protocol execution to successfully convince the verifier. Then, the extractor E attests to the witness usage in the protocol execution. The extractor E is a Turing machine that computes the witness from the observation of the communications between the prover P and the verifier V. Hence, since the extractor has

extracted the witness only known by the prover, then the witness must have been used during the protocol execution.

In any case, eavesdropping on the communications does not leak any information about the witness w, because the protocol satisfies the zero-knowledge property:

- **Zero-knowledge**. For all (x, w) in language $\mathcal{L}$, there exists for all verifiers a simulator S such that no polynomial-time distinguisher D can distinguish an execution of the simulated protocol from the execution of a real interaction between the prover P and the verifier V.

$$\forall (x,w) \in \mathcal{L}, \forall V \; \exists S : Probability[D\langle P(x,w), V(x)\rangle = 1] \\ - Probability[D\langle S(x)\rangle = 1] \leq negligible$$

The distinguisher D is a decision-making algorithm that compares the interaction of the prover P and verifier V with the execution of a simulator S and formalizes the concept of indistinguishability. In the real interaction, the prover P makes use of its witness w, while in the simulated one, the simulator S has only the public value x. If the simulator produces a transcript of the interaction equal to the real one, then the two executions are indistinguishable. Since the simulator does not know the witness, its simulated transcript does not leak any information about it. Therefore, if the real and the simulated transcript are indistinguishable, then the interaction between the prover P and the verifier V does not leak any information about the witness w and thus it is zero-knowledge.

3.1.2 Real-world examples

The Strange Cave

The following example is described in [26]. There are two parties, Peggy the prover P and Victor the verifier V. Peggy found a cave that contains a door that opens up only by using a secret word, which Peggy knows. The cave is shaped like a circle, with the entrance on one side and the door blocking the opposite side. Victor does not believe that Peggy possesses the secret, so Peggy will prove that she knows the secret word without telling to Victor. Peggy goes into a random branch of the cave (left or right, called A and B), without Victor knowing which branch she chose. Victor, which is at the entrance of the cave to not hear nor see the secret, tells Peggy a random branch to come out of. If Peggy knows the secret word, then she can do whatever Victor wants every time, using the door if necessary. Otherwise, if Peggy does not know the secret word, she has a 50% chance of initially fooling Victor. By repeating this protocol many times, Peggy's chance of cheating on Victor decreases until it becomes negligible. The probability of Peggy fooling Victor is $\frac{1}{2^n}$, where n is the number of rounds of this interactive protocol.

In the end, Victor is convinced that Peggy can open the door because she knows the secret, but no additional information flowed to him during the execution of the protocol.

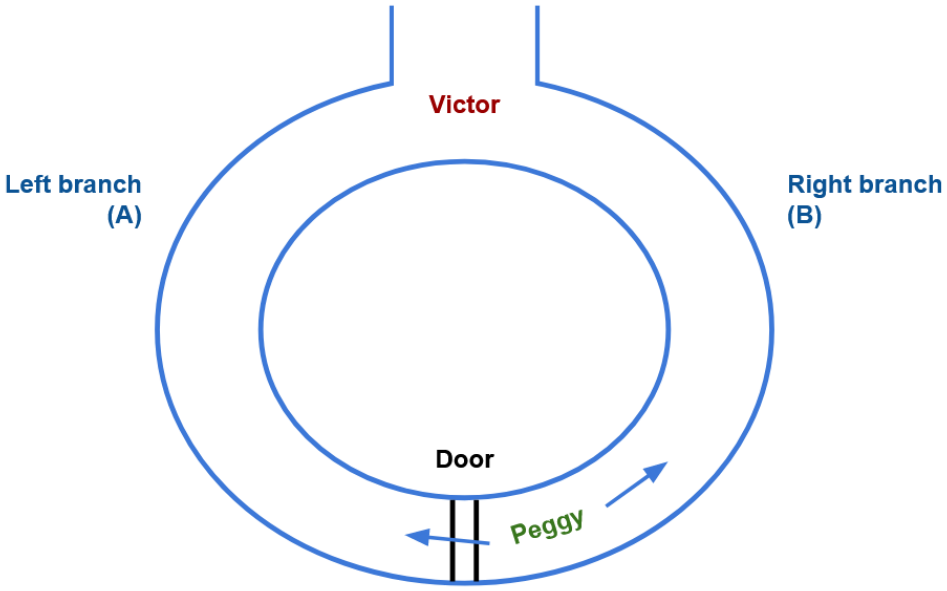


Figure 3.2. The strange cave from [26].

Graph Three-Colorability

A graph is a structure composed of a set of objects that may be linked to each other. More formally, a graph is a pair $G = (V, E)$, where V is a set of vertices (the objects) and E is a set of edges (the links to two objects).

Graph three-colorability (G3C) is a special case of *graph coloring*, which is a way of coloring the vertices of a graph so that no two adjacent vertices have the same color. Hence, a graph $G(V, E)$ is three-colorable [27] if there exists a mapping $\phi : V \to \{1,2,3\}$ (called proper coloring) such that each $(u, v) \in E$ satisfies $\phi(u) \neq \phi(v)$. It is an interesting problem because, for some graphs, it can be hard to find a solution, or even determine if a solution exists. The decision problem of whether a given graph supports a solution with three colors is in the complexity class *NP-complete*.

In this scenario, Peggy the prover P wants to convince Victor the verifier V that she knows a three-coloring for a certain graph, without revealing such coloring. The prover will prove it by performing a protocol of $|E^2|$ steps (where E is the set of edges and $|E|$ is the number of edges in the set), each of which involves the following actions [27]:

- Peggy permutes the three colors at random and hides the coloring from Victor.
- Victor chooses an edge of the graph at random.
- Peggy reveals the colors of the two nodes for which the selected edge is incident.

- Victor confirms that the two colors are valid because are different.

If the graph is three-colorable and Peggy knows a certain coloring, then Victor will never select an edge between two vertices that are not colored correctly. Otherwise, if the graph is not three-colorable or Peggy is cheating, there is a $\frac{1}{|E|}$ chance on each step that Peggy's attempt will be discovered. The probability becomes negligible after $|E^2|$ steps. Also, Victor gains no additional knowledge from the execution of the protocol, except that the graph is three-colorable, thus the system is zero-knowledge.

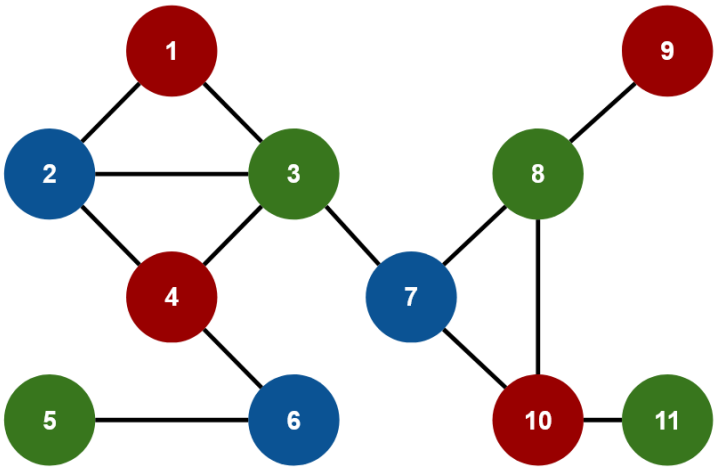


Figure 3.3. A three-colorable graph.

Since any problem in the class NP can be translated into an instance of the G3C problem, which is in the class NP-complete, this example proves that efficient zero-knowledge proofs exist for every statement with language in NP.

3.2 Commitment schemes

It is necessary to define which action is performed when the verb "to hide" is used during the execution of a protocol to get a zero-knowledge proof for NP languages. This hiding function is obtained through the use of *commitments*.

Commitment schemes [28] are used to enable a party to commit itself to a value while keeping it secret. If a commitment is *opened*, the hidden value is revealed, and it is guaranteed that the revealed value is the one that was committed during the committing phase. The opening phase is called this way because commitment schemes may be seen as the equivalent of non-transparent sealed envelopes.
To sum up, a commitment scheme is a two-party protocol composed of two phases:

- **Commitment phase**. A party called A has a secret s that wants to commit in order to send the commitment value c to another party B. The value c is the output of $c = \texttt{Commit}(s, r)$, where r is a random value.
- **Opening phase**. A sends to B, which has already received the commitment c, the former secret s and the randomness r so B can check $b = \texttt{Open}(c, s, r)$, where b is true if and only if $c = \texttt{Commit}(s, r)$.

A secure commitment scheme must have two properties:

- **Hiding property**. It is hard to compute any information about the secret s given the commitment c. This property, also called *secrecy requirement*, must be satisfied even if the receiver B tries to cheat.
- **Binding property**. For a given commitment c, it is hard to compute a different pair of secret and randomness (s', r') whose commitment is c. This property, also called *unambiguity requirement*, must be satisfied even if the sender A tries to cheat.

By introducing a commitment scheme into the previous example about graph three-colorability, the prover generates commitments to each vertex coloring. Then, these commitments are sent to the verifier. When the verifier chooses an edge, the prover reveals (opens) the committed values corresponding to the two vertices linked by the chosen edge.

3.2.1 Pedersen commitment scheme

The Pedersen commitment scheme [29] is a discrete logarithm problem-based scheme. It is defined in a group $\mathbb{G}_q$ of prime order q, where the discrete logarithm problem is infeasible. In this document, the discrete logarithm problem will reoccur and it will be explored further, as well as the Pedersen commitment scheme. For now, let's simply define that, given an exponentiation $c = a^b$, it must be hard to find the logarithm $b = \log_a(c)$.

The Pedersen commitment is computed as $c = \texttt{Commit}(m, r) = g^m h^r$, where m is the message, r is the randomness, and g, h are two random public generators of the group. The first term g^m commits the message, while the second term h^r works as a blinding factor to protect and randomize the commitment.

Moreover, the Pedersen commitment is *homomorphic* for the addition. This means that for two messages m_1, m_2 and two randomness r_1, r_2, we have:

$$\begin{aligned}
\texttt{Commit}(m_1, r_1) \cdot \texttt{Commit}(m_2, r_2) &= (g^{m_1} h^{r_1})(g^{m_2} h^{r_2}) \\
&= (g^{m_1} g^{m_2})(h^{r_1} h^{r_2}) \\
&= g^{m_1+m_2} h^{r_1+r_2} \\
&= \texttt{Commit}(m_1 + m_2, r_1 + r_2)
\end{aligned}$$

This property is useful because it allows commitments not only to be used to hide values but also to perform operations on them, while hidden. Pedersen commitments are not fully homomorphic, though, as they are homomorphic only for the addition and not for the multiplication.

3.3 Proof of Knowledge protocols

As previously described, proof of knowledge guarantees that a prover knows a secret about a statement. Additionally, if it is a zero-knowledge proof of knowledge, it does not reveal any information about the secret. This proof may be used to prove the knowledge of a discrete logarithm, such as a Pedersen commitment. Thereby, a prover P can prove that it knows the secret message m (or s) behind the commitment c, without revealing the secret message, which may also be called the witness w.

3.3.1 Schnorr identification protocol

The Schnorr identification protocol [30] performs a proof of knowledge of a discrete logarithm. As noted in the name, its purpose was to identify a party, which means that the party has to prove knowledge of the secret key corresponding to a certain public key.

The protocol (simplified in [24]) is defined for a cyclic group $\mathbb{G}_q$ of order q with public generator g. The prover P interacts with the verifier V to prove knowledge of $w = \log_g(x)$, where w is the witness and x is the public value, as follows:

1. The prover P commits a randomness $r \in \{1, ..., q-1\}$, generating the commitment $c = g^r$, which is sent to the verifier V.
2. The verifier V picks a random challenge $e \in \{1, ..., q-1\}$ and sends it to the prover P.
3. The prover P replies to the challenge with the response $y = r + ew$.
4. The verifier V accepts the response, convinced by the proof, if and only if $c = \frac{g^y}{x^e}$.

This is a valid proof of knowledge because there is an extractor E that performs the following steps:

1. It gets the prover to output the commitment $c = g^r$.
2. It chooses a randomness e_1 and inputs it to the prover, which outputs $y_1 = r + e_1 w$.

3. It rewinds the prover to the state of step 1, generates a different randomness e_2, and inputs it to the prover, which outputs $y_2 = r + e_2 w$.

4. It outputs the differences of the challenge-response pairs $\frac{y_1 - y_2}{e_1 - e_2}$.

Then, the output of the extractor E is the witness w, because $\frac{y_1 - y_2}{e_1 - e_2} = \frac{(r + e_1 w) - (r + e_2 w)}{e_1 - e_2} = w\frac{(e_1 - e_2)}{(e_1 - e_2)} = w$. The extraction fails if $e_1 = e_2$, which may happen with probability $\frac{1}{q}$. This is why their difference is specified. Since the extractor extracts the witness, the Schnorr identification protocol satisfies the special soundness requirement for proof of knowledge protocols.

Whilst, the Schnorr identification protocol does not fully satisfy the zero-knowledge property, because it works in the *honest verifier zero-knowledge (HVZK) model*. This means that, if a dishonest verifier V' does not choose a challenge e randomly, then it can learn something about the witness w. Under this simplification, the simulator S does not need to predict in advance an arbitrary behavior of the verifier. Instead, the simulator S can simulate a random choice of the challenge e. Thus, the simulator S acts as follows:

1. It chooses the prover's response y and the verifier's challenge e at random.

2. It computes the prover's commitment as $c = \frac{g^y}{x^e}$.

3. It simulates the prover's response y.

At the end of this simulation, it holds that $cx^e = \frac{g^y}{x^e} x^e = g^y$.

Protocols such as this one, which have three moves for sending and receiving a commitment, a challenge, and a response, are called *sigma protocols* (or $\Sigma - protocols$).

The Schnorr identification protocol is used as a building block later in this document.

3.4 Non-Interactive ZKPs

The interactive protocols are useful, but they only work if the verifier is online and willing to interact with the prover. It is possible to transform interactive ZKPs into non-interactive zero-knowledge (NIZK) proofs, which allow having only a single message sent by a prover to a verifier. The interaction between a prover and a verifier is simulated by the prover, thus direct communication with the verifier is unnecessary and proof generation can be done offline.

3.4.1 Fiat-Shamir heuristic

Non-interactive zero-knowledge proofs can be obtained using the Fiat-Shamir heuristic, also appointed as Fiat-Shamir transformation [31]. According to this technique, it is possible to replace the verifier's random challenge with the output value of a cryptographic hash function, whose inputs are the public values exchanged in the preceding steps.

By applying the Fiat-Shamir transformation to the Schnorr identification protocol, the public values to encode and concatenate before being entered as input to the cryptographic hash function $H()$ are the generator g, the public value x, and the commitment c.

The prover P, which knows g, w, and x, is modeled as follows:

Prover $P(g, w, x = g^w)$:

1. It picks a random value r.
2. It generates the commitment $c = g^r$.
3. It calculates the digest $e = H(g||x||c)$, where $H()$ is a cryptographic hash function and $||$ means concatenation.
4. It calculates $y = r + ew$
5. It outputs $\pi = (c, e, y)$

While the verifier V, which knows g, x and receives the proof π from the prover P, is modeled as follows:

Verifier $V(g, x = g^w, \pi = (c, e, y))$:

1. It calculates the digest $e = H(g||x||c)$, where $H()$ is a cryptographic hash function and $||$ means concatenation.
2. It checks if $g^y = cx^e$.

The Fiat-Shamir heuristic is secure against chosen message attacks in the *random oracle model (ROM)* [32]. A random oracle [33] responds to every query with a uniformly random response. If identical queries are repeated, it responds the same way each time. One-time functions, such as cryptographic hash functions, can be modeled as random oracles. If a system is proven secure when every hash function is replaced by a random oracle, it means that the system needed it to satisfy strong randomness assumptions about the hash function's output. Thereby, the system is secure under the random oracle model, which is a weaker model than the standard one. It is not known whether random oracles exist. In the negative case, the Fiat-Shamir heuristic is proven to be insecure [34].

3.4.2 ZKPs in digital signatures

Schnorr noted [30] that also a message m can be entered as an additional input to the cryptographic hash function $H()$. Therefore, he obtained a proof of knowledge of a witness w and also a commitment to a message m that is cryptographically linked to the NIZK proof. If the proof is correct, then only a party that knows the witness could have committed that message. This is what defines a digital signature, where the witness w is the private key, $x = g^w$ is the public key, and m is the signed message. Hence, digital signatures are non-interactive zero-knowledge proofs. In fact, the Schnorr signature scheme is obtained by employing the Fiat-Shamir transformation to the Schnorr identification protocol.

3.4.3 General-purpose ZKPs

An extension of the work done by Fiat and Shamir is the *common reference string (CRS)* [35]. The common reference string is a value shared between the prover and the verifier to build non-interactive zero-knowledge proofs, and thus without requiring a challenge from the verifier. The common reference string may be generated by a trusted third party or through a secure *multi-party computation (MPC)* [36].

ZKP systems can be used to provide privacy in many fields and applications. Since there are multiple different general-purpose non-interactive zero-knowledge proof schemes, the main characteristics to distinguish and identify them are the following [37]:

- **Succinctness**. It is about the size of the proofs and the amount of time needed to verify them. If a system is succinct, then the proofs it produces are short (in the order of hundreds of bytes) and fast (in the order of milliseconds) to verify.
- **Transparency**. It is about the initial setup to agree on a set of parameters and common values, which are part of the common reference string. There are two types of setup:
  - *Trusted*. It means that the party that created the CRS has access to secrets that allow to forge proofs. Thereby, the party must be trusted. As mentioned before, the multi-party computation can decrease the risk because many participants compute these parameters, and if a single participant is honest and deletes its keys after the so-called "ceremony", then no one can forge proofs.
  - *Transparent*. It means that no trusted third party is needed to create the parameters of the system.
- **Quantum-resistance**. It is about the cryptographic primitives used to build the system, and whether they are resistant to quantum computers or not.

Nowadays, the most popular general-purpose ZKPs are zk-SNARKs, zk-STARKs, and Bulletproofs.

zk-SNARKs

The most used among the three listed above are zk-SNARKs (Zero-Knowledge Succinct and Non-interactive ARguments of Knowledge) [38], which include many different schemes. They are non-interactive schemes with zero knowledge and succinctness. Furthermore, they require a trusted setup. It is important to notice that they are not proofs of knowledge but arguments of knowledge. The difference between proofs and arguments is about their soundness [39]. A proof system has *statistical soundness*, where even a computationally unbounded prover cannot convince a verifier of a false statement, except for a negligible probability. On the other hand, an argument system has *computational soundness*, where only a polynomial-time prover cannot convince a verifier of a false statement. The computing complexity for proving (and verifying) a proof about a specific statement depends on the number of operations N needed to generate (or verify) the proof. The cryptographic security assumptions behind zk-SNARKs are strong: discrete logarithm problem and secure bilinear pairing (which will be explained in the next chapter). Having these assumptions, zk-SNARKs are not post-quantum secure.

zk-STARKs

The ZKP scheme zk-STARK (Zero-Knowledge Succinct and Transparent ARgument of Knowledge) [40] is transparent, thus it does not require a trusted setup. Although its succinctness, the zk-STARK proof size is much higher than the zk-SNARKs one. The unique characteristic of this scheme is that it is supposed to be post-quantum secure. The quantum-resistance property is obtained through the use of collision-resistant hash functions (CRHF) as security assumptions.

Bulletproofs

Bulletproofs [41] is a succinct non-interactive zero-knowledge proof protocol without a trusted setup. This scheme is well suited for efficient range proofs on committed values. A range proof allows a party to prove that a certain value behind a commitment lies in a given range. Beyond range proofs, Bulletproofs provides short zero-knowledge proofs for general-purpose statements. Since it relies on the discrete logarithm assumption, it is not quantum-resistant.

Here is a summary table[1] of the three general-purpose ZKP systems:

	zk-SNARKs	*zk-STARKs*	*Bulletproofs*
Prover complexity	$\mathcal{O}(N \log N)$	$\mathcal{O}(N poly \log N))$	$\mathcal{O}(N \log N)$
Verifier complexity	$\sim \mathcal{O}(1)$	$\mathcal{O}(poly \log N)$	$\mathcal{O}(N)$
Proof size	200 B	45-135 kB	1.5-2.5 kB
Trusted setup	Yes	No	No
Post-quantum	No	Yes	No

Table 3.1. Comparison of general-purpose ZKPs.

[1]Table from https://github.com/matter-labs/awesome-zero-knowledge-proofs.

Chapter 4

BBS+ Signatures

4.1 Building blocks

This section introduces necessary algebra concepts for defining and comprehending BBS+ signatures.

4.1.1 Algebra fundamentals

First of all, it is helpful to briefly review two algebraic structures that underlie cryptographic operations: groups and fields [42].

Groups

A group $\mathbb{G}$ is a set of elements with an operation $\circ$ which combines two elements of $\mathbb{G}$. It has the following properties:

- **Closure**. The group operation $\circ$ is *closed*, thus for all $a, b \in \mathbb{G}$, it holds that $a \circ b = c \in \mathbb{G}$.
- **Associativity**. The group operation $\circ$ is *associative*, thus for all $a, b, c \in \mathbb{G}$, it holds that $a \circ (b \circ c) = (a \circ b) \circ c$.
- **Identity**. For each $a \in \mathbb{G}$ there exists an element $1 \in \mathbb{G}$, called the *neutral* element or *identity* element, such that $a \circ 1 = 1 \circ a = a$.
- **Inverse**. For each $a \in \mathbb{G}$ there exists an element $a^{-1} \in \mathbb{G}$, called the *inverse* of a, such that $a \circ a^{-1} = a^{-1} \circ a = 1$.

The group operation $\circ$ denotes multiplication in multiplicative groups and addition in additive groups.

A group $\mathbb{G}$ is *abelian* if it satisfies the four properties above, plus the following additional property:

- **Commutativity**. The group operation $\circ$ is *commutative*, thus for all $a, b \in \mathbb{G}$, it holds that $a \circ b = b \circ a$.

A group $\mathbb{G}$ is *finite* if it has a finite number of elements. The cardinality or order of the group $\mathbb{G}$ is denoted as $|\mathbb{G}|$.

The order $ord(a)$ of an element $a \in \mathbb{G}$ is the smallest positive integer k such that $a^k = a \circ a \circ ... \circ a \ (k \ times) = 1$.

A group $\mathbb{G}$ is *cyclic* if contains an element a with maximum order $ord(a) = |\mathbb{G}|$. Elements with maximum order are called *generators*, because every element $b \in \mathbb{G}$ is a power of this element for some i, $a^i = b$. This means that the generator generates the entire group.

A subgroup $\mathbb{S}$ of a group $\mathbb{G}$ is a subset of the elements of the group ($\mathbb{S} \subseteq \mathbb{G}$) and it is itself a group.

Groups allow to abstract many different structures, such as $\mathbb{R}$, $\mathbb{C}$, $\mathbb{Z}$, to work with them in a general way.

Fields

A field $\mathbb{F}$ is a set of at least two elements, with two operations: $\oplus$ (addition) and $\odot$ (multiplication). It satisfies the following axioms:

- $\mathbb{F}$ forms an abelian group (where identity = 1) under the operation $\oplus$.
- $\mathbb{F}^* = \mathbb{F} \backslash \{0\} = \{a \in \mathbb{F}, a \neq 0\}$ forms an abelian group (where identity = 0) under the operation $\odot$.

Moreover, a field $\mathbb{F}$ has the following property:

- **Distributivity**. For all $a, b, c \in \mathbb{F}$, it holds that $(a \oplus b) \odot c = (a \odot c) \oplus (b \odot c)$.

A field $\mathbb{F}$ is *finite* if it has a finite number of elements. It may be referred to as *Galois field (GF)*.

Fields are needed to have all four basic arithmetic operations (i.e., addition, subtraction, multiplication, division) in a single structure, which contains an additive and a multiplicative group.

4.1.2 Discrete Logarithm problem

The hard problem at the basis of the algorithms that will be introduced is the discrete logarithm problem (DLP), which is the underlying one-way function of the Diffie-Hellman Key Exchange protocol [42].

Given a finite cyclic group $\mathbb{G}$ with the group operation $\circ$ and cardinality n, a generator $g \in \mathbb{G}$ and an element $y \in \mathbb{G}$, the discrete logarithm problem is finding the integer x, where $1 \leq x \leq n$, such that $y = g \circ g \circ ... \circ g \ (x \ times) = g^x$.

Usually, it is defined over $\mathbb{Z}_p^*$ of order $p-1$, where p is a prime, and the problem is to determine the integer $1 \leq x \leq p-1$ such that $g^x \equiv y \bmod p$.

The assumption is that it is infeasible to find x such that $y = g^x$. It is important to note that there are cyclic groups in which the discrete logarithm problem is not difficult and thus it is not a one-way function.

4.1.3 Elliptic Curves

An elliptic curve E is a curve on a plane with points (x, y), where x and y are the coordinates. It is described by its equation $y^2 = x^3 + ax + b$, called *Weierstrass equation*, where a, b are defined such that $4a^3 + 27b^2 \neq 0$. Hence, an elliptic curve is a set of points that are solutions to its equation. In addition to them, there exists an imaginary point called *point at infinity 0* [42].

Usually, elliptic curves are considered over a finite field $\mathbb{F}_q$, where all arithmetic is performed modulo a prime q. They are noted as $E(\mathbb{F}_q)$. Formally, an elliptic curve $E(\mathbb{F}_q)$ can be defined as follows:

$$\{(x, y) \in (\mathbb{F}_q)^2 \ : y^2 \equiv x^3 + ax^2 + b \ (mod \ q), \\ 4a^3 + 27b^2 \not\equiv 0 \ (mod \ q)\} \cup \{0\}$$

Elliptic curves are abelian groups, where the elements of the group are the points of the elliptic curve. Elliptic curves satisfy the same properties of abelian groups:

- **Closure**. The addition of two points, elements of the elliptic curve, generates a point that belongs to that curve.
- **Associativity and commutativity**. The addition is both associative and commutative. Given three aligned points P, Q, R, then $P + Q + R = P + (Q + R) = Q + (P + R) = R + (P + Q) = ... = 0$.
- **Identity**. The identity element is the point at infinity 0.
- **Inverse**. The inverse of a point is the one symmetric on the x-axis.

The group operation is the addition $+$, but it is possible to define another operation, the *scalar multiplication*, which is $kP = P + P + ... + P \ (k \ times)$, where k is an integer and P is a point of the elliptic curve.

The order (or cardinality) of an elliptic curve group, noted as $|E|$, is the number of points in the elliptic curve.

Elliptic Curve Discrete Logarithm problem

Elliptic curves have a similar problem to the discrete logarithm problem: the elliptic curve discrete logarithm problem (ECDLP). It is the problem of finding the number k, where $1 \leq k \leq |E|$, given an elliptic curve E with a generator point $P \in E$ and another point $Q \in E$, such that $Q = kP$.

In fact, translating an operation on elliptic curves means performing multiplication instead of exponentiation and addition instead of multiplication.

4.1.4 Bilinear Maps

Let $\mathbb{G}_1$, $\mathbb{G}_2$, and $\mathbb{G}_T$ groups of prime order p. A map $e : \mathbb{G}_1 \times \mathbb{G}_2 \rightarrow \mathbb{G}_T$ must satisfy the following properties [43]:

- **Bilinearity**. For all $g_1 \in \mathbb{G}_1$, $g_2 \in \mathbb{G}_2$ and $x, y \in \mathbb{Z}_p$, it holds that $e(g_1^x, g_2^y) = e(g_1, g_2)^{xy}$.
- **Non-degeneracy**. For all generators $g_1 \in \mathbb{G}_1$ and $g_2 \in \mathbb{G}_2$, it holds that $e(g_1, g_2) \neq 1$ and it generates an element in $\mathbb{G}_T$.
- **Efficiency**. There exists an efficient algorithm that outputs the bilinear group $(p, \mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, e, g_1, g_2)$ and an efficient algorithm to compute $e(a, b)$ for any $a \in \mathbb{G}_1$, $b \in \mathbb{G}_2$.

There are three types of bilinear maps, also called *pairings* [44]:

- **Type-1**. It has a symmetric setting, where $\mathbb{G}_1 = \mathbb{G}_2$.
- **Type-2**. It has an asymmetric setting, where $\mathbb{G}_1 \neq \mathbb{G}_2$, with an efficiently computable isomorphism $\psi : \mathbb{G}_2 \rightarrow \mathbb{G}_1$.
- **Type-3**. It has an asymmetric setting, where $\mathbb{G}_1 \neq \mathbb{G}_2$ and there is no efficiently computable isomorphism ψ.

Note that an *isomorphism* is a function to compare similarities between two groups. If there exists an isomorphism between two groups, then they are isomorphic, which means that they have the same properties and need not be distinguished.

While type-1 pairings can be used to reduce a hard problem in a group to an easier problem in another group, type-3 pairings are used to build many cryptographic systems.

Usually, $\mathbb{G}_1, \mathbb{G}_2$ are cyclic subgroups of elliptic curves over a finite field, while the target group $\mathbb{G}_T$ is a multiplicative subgroup over a large field.

4.1.5 q-Strong Diffie-Hellman problem

The q-Strong Diffie-Hellman problem (qSDH) has two versions based on different pairing types.

The first version, called "Eurocrypt version", is defined in type-1 and type-2 pairings [45]:

> Given a $q+2$-tuple $(g_1, g_2, g_2^x, g_2^{x^2}, ..., g_2^{x^q}) \in \mathbb{G}_1 \times \mathbb{G}_2^{q+1}$ with $g_1 = \psi(g_2)$, the q-SDH problem is to output a pair $(c, g_1^{1/(x+c)}) \in \mathbb{Z}_p^* \times \mathbb{G}_1$.

The second version, called "Journal of Cryptography (JOC) version", is defined in type-3 pairings [46]:

> Given a $q+3$-tuple $(g_1, g_1^x, g_1^{x^2}, ..., g_1^{x^q}, g_2, g_2^x) \in \mathbb{G}_1^{q+1} \times \mathbb{G}_2^2$, the q-SDH problem is to output a pair $(c, g_1^{1/(x+c)}) \in \mathbb{Z}_p \backslash \{-x\} \times \mathbb{G}_1$.

The assumption is that it is infeasible to find a random $x \in \mathbb{Z}_p$ and output $(c, g_1^{1/(x+c)})$, where $c \in \mathbb{Z}_p$ and $g_1^{1/(x+c)} \in \mathbb{G}_1$.

4.2 BLS12-381 Elliptic Curve

The BBS+ signature is a pairing-based digital signature implemented over the BLS12-381 elliptic curve. In order to build a pairing-based cryptographic system over elliptic curves, it is necessary to design these elliptic curves in such a way that the pairing is secure. To achieve this, the discrete logarithm problems in the elliptic curve group $E(\mathbb{F}_q)$ and in the pairing target group $\mathbb{F}_q^*$ must both be computationally infeasible [47]. A *pairing-friendly* elliptic curve must have a small *embedding degree* (which will be explained later) to allow efficient pairings, but not too small, to be safe against attacks that can efficiently resolve the discrete logarithm problem, such as the MOV attack [48]. The embedding degree k must be in the range $6 < k < 100$. In addition, the elliptic curve must have a large prime-order subgroup.

The pairing-friendly elliptic curve BLS12-381 is part of the curves family BLS from Barreto, Lynn, and Scott [49]. Specifically, BLS12-381 is designed by Bowe [50] and a good primer about it is written in [51].

4.2.1 Parameters

The curve equation is $y^2 = x^3 + 4$, thus the values from the general elliptic curve equation are $a = 0$ and $b = 4$. Since the curve belongs to the BLS family, its parameters are set using a single parameter x that is selected to give the curve desired properties for implementation.

The design goals for BLS12-381 are the following:

- The BLS parameter x must have a low *Hamming weight*, which means that it is an integer with a low number of bits set to 1. This is important for the efficiency of the *Miller loop*, the algorithm used in implementations to calculate pairings.
- The field modulus q must be a prime number and have 383 bits or fewer. It is 381-bits long due to the selected x. This means that the number of bits needed to represent coordinates on the curve is 381, which is good for the efficiency of arithmetic operations on it.
- The order of the subgroups r must be a prime number and have 255 bits or fewer, for the same efficiency reason as above.
- The curve must target the 128-bits security level.

The BLS parameter is $x = -2^{63}-2^{62}-2^{60}-2^{57}-2^{48}-2^{16} =$ `-0xd201000000010000`, which gives both a large q and a low Hamming weight. The field modulus is $q = \frac{1}{3}(x-1)^2(x^4-x^2+1)+x$ and the subgroup order is $r = x^4 - x^2 + 1$. Their values are calculated by substituting the x in their formulas with the value of the BLS parameter.

4.2.2 Field extensions

The field $\mathbb{F}_q$ is composed of field elements that are integers modulo q, from 0 to $q-1$. The field $\mathbb{F}_{q^2}$ is the quadratic extension of $\mathbb{F}_q$ and its field elements are first-degree polynomials, e.g., $a_0 + a_1 x$. A coefficient x^2 will appear in multiplications, so the polynomials must be reduced following a rule. The rule must be a degree k polynomial, in this example $k = 2$, and it must be irreducible. If there exists an irreducible k-degree polynomial in $\mathbb{F}_q$, then it is possible to extend the field to $\mathbb{F}_{q^k}$ and represent its elements as degree $k-1$ polynomials, e.g, $a_0+a_1x+...+a_{k-1}x^{k-1}$.

Large extension fields such as $\mathbb{F}_{q^{12}}$, which is used by BLS12-381, can be implemented as towers of smaller extensions.

4.2.3 Curves, subgroups and twists

BLS12-381 comprises two curves that are defined over different fields. The first curve $E(\mathbb{F}_q)$ is defined over the finite field $\mathbb{F}_q$ with equation $y^2 = x^3 + 4$. The second curve $E'(\mathbb{F}_{q^2})$ is defined over an extension of $\mathbb{F}_q$ to $\mathbb{F}_{q^2}$ with equation $y^2 = x^3 + 4(1+i)$. This curve is much bigger ($|E'| \sim q^2$) than the first one ($|E| \sim q$) because the curve equation has more solutions when the domain is extended to complex numbers.

The presence of two curves is needed to have two distinct groups to define a pairing. The simple curve $E(\mathbb{F}_q)$ has only a single large subgroup of order r, which is $\mathbb{G}_1$. Therefore, the field over which E is defined is extended to find a curve

$E'(\mathbb{F}_{q^k})$ with more subgroups of order r, from which one is selected to be $\mathbb{G}_2$. The amount k needed to extend the base field to find the second group is called the *embedding degree* of the curve, which in BLS12-381 is $k = 12$.

Doing arithmetic operations in $\mathbb{F}_{q^{12}}$ is inefficient, but it is possible to transform the curve $E'(\mathbb{F}_{q^{12}})$ into a curve defined over a lower degree field using a *twist*. BLS12-381 uses a "sextic twist", which reduces the degree of the extension field by a factor of six. This means that $\mathbb{G}_2$ can be defined over $\mathbb{F}_{q^2}$ instead of $\mathbb{F}_{q^{12}}$.

4.2.4 Embedding degree

The embedding degree k is the smallest positive integer such that r divides $(q^k - 1)$ and it gives the smallest field extension $\mathbb{F}_{q^k}$ that satisfies the following conditions:

- $\mathbb{F}_{q^k}$ must contain more than one subgroup of order r, to construct $\mathbb{G}_2$.
- $\mathbb{F}_{q^k}$ must contain all the rth *roots of unity*, to construct $\mathbb{G}_T$. Roots of unity form a subgroup in $\mathbb{F}_{q^{12}}$ of order r, which is the group $\mathbb{G}_T$.

The embedding degree is a tradeoff between security and efficiency: a higher embedding degree makes the discrete logarithm problem harder to solve in $\mathbb{G}_T$, but it increases the number of arithmetic operations to perform and so it decreases efficiency. Currently, the embedding degrees that are being used are $k = 12$, $k = 24$, $k = 48$ [52].

4.2.5 Security level

In elliptic curve cryptography, security refers to making the discrete logarithm problem hard enough. For pairing-friendly curves, the DLP must be hard in all three groups $\mathbb{G}_1$, $\mathbb{G}_2$, $\mathbb{G}_T$.

The security level is a measure of the strength achieved by a cryptographic primitive, an elliptic curve in this case. It is expressed in bits and n-bit security means that the attacker must perform 2^n operations to be successful.

BLS12-381 was intended to offer a 128-bit security level but a report [53] showed that the actual security level is between 117 and 120 bits. Nevertheless, the security decrease is minimal and it does not raise concerns, thus the security level of the curve is adequate.

4.3 BBS+ Signature Scheme

The BBS signature scheme is a digital signature scheme that provides short group signatures. Group signatures [54] provide anonymity for signers in a group with

many members. A member of the group can sign messages but the resulting signature cannot reveal information about the signer because it is verified with the group's public key.

Firstly, the BBS signature scheme was defined by Boneh, Boyen, and Shacham in 2004 [55]. Then, it was modified to achieve a signature scheme with efficient protocols, similar to the one developed by Camenisch and Lysyanskaya [56] but not based on RSA [57]. It was described to build an anonymous authentication system by Au, Susilo, and Mu in 2006 [58], and it was referred to as *BBS+ signature* since it was a modification to the original scheme. Later, the new scheme was used in a direct anonymous attestation system for trusted platform modules by Camenisch, Drijvers, and Lehmann in 2016 [43]. The proof of knowledge of a BBS+ signature has been modified by them in order to secure it in type-3 pairings and under the JOC version of the q-SDH assumption. This allows proofs to be defined over $\mathbb{G}_1$ instead of $\mathbb{G}_T$, which means more efficiency.

The BBS+ signature scheme provides data integrity and verifiable authenticity. It is employed in this work as a digital signature for digital credentials, such as verifiable credentials, because it has unique features to increase the privacy level of self-sovereign identity solutions. Since BBS+ allows signers to sign multiple messages while producing a single signature, each attribute of a credential is handled as a message to sign. In particular, the three key properties of BBS+ signatures are the following [59]:

- **Selective disclosure**. A signer can sign multiple messages and produce a single constant-size signature. A prover that possesses the messages and the signature can generate a proof whereby they can select which messages to disclose, while no information about the undisclosed messages is revealed. The proof maintains the integrity and authenticity of the disclosed messages as they are signed by the signer.

- **Unlinkable proofs**. The proofs are zero-knowledge proofs of knowledge of the signature. This means that a verifier that receives a proof cannot determine which signature was used to generate the proof. Therefore, two proofs generated from the same signature are unlinkable.

- **Proof of possession**. The proofs prove to a verifier that the prover was in possession of a signature, together with the signed messages. The scheme supports binding a presentation header to the generated proof, which may be a cryptographic nonce, to provide freshness to the proof and avoid replay attacks.

This scheme specifies that the signatures and the proofs are defined in $\mathbb{G}_1$, to make the operations involved in both algorithms more efficient. On the other hand, public keys are defined in $\mathbb{G}_2$.

Each signed message m_1, which is an integer after being output from a cryptographic hash function, is paired with a specific generator H_1, which is an elliptic

curve point in $\mathbb{G}_1$. In addition to message generators, the scheme uses two additional generators: H_s and H_d. The generator H_s signs the blinding value s of the signature, while H_d signs the signature's domain, which binds the signature (and/or the generated proof) to a specific context and may protect additional information.

The BBS+ signature scheme is going through the process of standardization and it has been divided into two parts: a core draft [59], which has been submitted to IETF, and an extension draft [60], which is still in an early stage. The core draft provides a specification for the following operations: key generation, sign, signature verification, proof generation, and proof verification. Whilst the extension draft specifies other operations needed to perform *blind signatures*: Pedersen commitment, zero-knowledge proof of knowledge of committed messages generation together with its verification, blind sign, and unblind signature.

All the major algorithms will be described in the following sections as a mix of the reference paper by Camenisch et al. [43] and the two drafts. The operations are performed on elliptic curves, specifically over the BLS12-381 curve, so there will be additions and multiplications. Even if not specified, membership checks in verification algorithms must be done to ensure that a public key or points in signatures or proofs are elements of a prime-order subgroup.

4.3.1 Core

Key generation

The private key or secret key (`sk`) is chosen randomly as a sample in the range $0 < \texttt{sk} < r$, while the public key (`pk`) is calculated by multiplying P_2, a public generator point in $\mathbb{G}_2$, with the secret key. Lastly, the generators are taken as random points in $\mathbb{G}_1$. Note that L is the number of messages to sign.

KeyGen ()

1 : $\mathsf{sk} \xleftarrow{\$} \mathbb{Z}_r$
2 : $\mathsf{pk} = W \leftarrow P_2 \cdot \mathsf{sk}$
3 : $(H_s, H_d, H_1, ..., H_L) \xleftarrow{\$} \mathbb{G}_1$
4 : **return** $\mathsf{sk}, \mathsf{pk}, (H_s, H_d, H_1, ..., H_L)$

Sign

The sign operation computes a signature from a secret key over a vector of messages and/or a header included in the domain value. Note that P_1 is a public generator point in $\mathbb{G}_1$ and the fraction $\frac{1}{\texttt{sk} + e}$ is the modular inverse $(\texttt{sk} + e)^{-1}\ mod\ r$. This fraction is the link to the q-SDH assumption explained above. Moreover, each input of the cryptographic hash function `hash` is in bytes, while the output is an integer.

$Sign\ (\mathsf{sk}, \mathsf{pk}, (H_s, H_d, H_1, ..., H_L), (m_1, ..., m_L), \mathsf{header})$

1 : $\mathsf{domain} \leftarrow \mathtt{hash}(\mathsf{pk} \| L \| H_s \| H_d \| H_1 \| ... \| H_L \| \mathsf{header})$

2 : $e, s \leftarrow\!\$\ \mathbb{Z}_r$

3 : $B \leftarrow P_1 + H_s \cdot s + H_d \cdot \mathsf{domain} + \sum_{i=1}^{L} H_i \cdot \mathtt{hash}(m_i)$

4 : $A \leftarrow B \cdot \frac{1}{\mathsf{sk} + e}$

5 : **return** $\sigma = (A, e, s)$

Signature verification

The verify operation checks that a signature is valid against a public key. To accomplish this, pairings e are used. Note that for a valid signature, the pairings multiplication must be equal to 1, which is the identity in $\mathbb{G}_T$ ($\mathbb{F}_{q^{12}}$).

$Verify\ (\mathsf{pk}, \mathsf{signature}, (H_s, H_d, H_1, ..., H_L), (m_1, ..., m_L), \mathsf{header})$

1 : $(A, e, s) = \mathsf{signature}$

2 : $W = \mathsf{pk}$

3 : $\mathsf{domain} \leftarrow \mathtt{hash}(\mathsf{pk} \| L \| H_s \| H_d \| H_1 \| ... \| H_L \| \mathsf{header})$

4 : $B \leftarrow P_1 + H_s \cdot s + H_d \cdot \mathsf{domain} + \sum_{i=1}^{L} H_i \cdot \mathtt{hash}(m_i)$

5 : **if** $e(A, W + P_2 \cdot e) \cdot e(B, -P_2) = 1$ **then**

6 : **return** true

7 : **else**

8 : **return** false

9 : **fi**

Proof generation

The proof generation computes a zero-knowledge proof of knowledge of a signature and the signed messages, with the possibility of selectively disclosing messages from the set of signed messages. To specify the disclosed messages, their indexes from the original set are supplied as a list $(i_1, ..., i_R)$, where R means revealed. Whilst the list of the undisclosed messages is referred to as $(i_1, ..., i_U)$, where U means unrevealed. In addition, a presentation header (`ph`) may be included to provide freshness to the generated proof. This proof is also called *signature proof of knowledge (SPK)*. Note that an operation $(-a)$ is a modular operation $-a \ mod\ r$. The value c is the digest needed by the Fiat-Shamir transformation to make this proof non-interactive. However, interaction is introduced if a presentation header is supplied.

ProofGen $(\mathsf{pk}, \mathsf{signature}, (H_s, H_d, H_1, ..., H_L), (m_1, ..., m_L), \mathsf{header}, (i_1, ..., i_R), \mathsf{ph})$

1 : $(i_1, ..., i_U) = (1, ..., L) \backslash (i_1, ..., i_R)$
2 : $(A, e, s) = \mathsf{signature}$
3 : $\mathsf{domain} \leftarrow \mathtt{hash}(\mathsf{pk} \| L \| H_s \| H_d \| H_1 \| ... \| H_L \| \mathsf{header})$
4 : $r_1, r_2, \widetilde{e}, \widetilde{r}_2, \widetilde{r}_3, \widetilde{s} \leftarrow\$ \mathbb{Z}_r$
5 : $\widetilde{m}_{j1}, ..., \widetilde{m}_{jU} \leftarrow\$ \mathbb{Z}_r$
6 : $B \leftarrow P_1 + H_s \cdot s + H_d \cdot \mathsf{domain} + \sum_{i=1}^{L} H_i \cdot \mathtt{hash}(m_i)$
7 : $r_3 \leftarrow r_1^{-1} \; mod \; r$
8 : $A' \leftarrow A \cdot r_1$
9 : $\bar{A} \leftarrow A' \cdot (-e) + B \cdot r_1$
10 : $D \leftarrow B \cdot r_1 + H_s \cdot r_2$
11 : $s' \leftarrow r_2 \cdot r_3 + s \; mod \; r$
12 : $C_1 \leftarrow A' \cdot \widetilde{e} + H_s \cdot \widetilde{r}_2$
13 : $C_2 \leftarrow D \cdot (-\widetilde{r}_3) + H_s \cdot \widetilde{s} + \sum_{j=1}^{U} H_j \cdot \mathtt{hash}(\widetilde{m}_j)$
14 : $c \leftarrow \mathtt{hash}(A' \| \bar{A} \| D \| C_1 \| C_2 \| R \| i_1 \| ... \| i_R \| m_{i1} \| ... \| m_{iR} \| \mathsf{domain} \| \mathsf{ph})$
15 : $\widehat{e} = c \cdot e + \widetilde{e} \; mod \; r$
16 : $\widehat{r}_2 = c \cdot r_2 + \widetilde{r}_2 \; mod \; r$
17 : $\widehat{r}_3 = c \cdot r_3 + \widetilde{r}_3 \; mod \; r$
18 : $\widehat{s} = c \cdot s' + \widetilde{s} \; mod \; r$
19 : **for** j **in** $(j1, ...jU)$
20 : $\quad \widehat{m}_j \leftarrow c \cdot \mathtt{hash}(m_j) + \widetilde{m}_j \; mod \; r$
21 : **endfor**
22 : **return** $\pi = (A', \bar{A}, D, c, \widehat{e}, \widehat{r}_2, \widehat{r}_3, \widehat{s}, (\widehat{m}_{j1}, ..., \widehat{m}_{jU}))$

Proof verification

The proof verify operation checks whether a proof is valid against a public key used to generate the original signature. Pairings e are employed here too. Note that the if-clause $A' = 1$ checks that the point A' is not equal to the identity point in $\mathbb{G}_1$.

$ProofVerify$ $(\mathsf{pk}, \mathsf{proof}, L, (H_s, H_d, H_1, ..., H_L), (m_1, ..., m_R), \mathsf{header}, (i_1, ..., i_R), \mathsf{ph})$

1 : $(A', \bar{A}, D, c, \widehat{e}, \widehat{r}_2, \widehat{r}_3, \widehat{s}, (\widehat{m}_{j1}, ..., \widehat{m}_{jU})) = \mathsf{proof}$
2 : $W = \mathsf{pk}$
3 : $\mathsf{domain} \leftarrow \mathtt{hash}(\mathsf{pk} \| L \| H_s \| H_d \| H_1 \| ... \| H_L \| \mathsf{header})$
4 : $C_1 \leftarrow (\bar{A} - D) \cdot c + A' \cdot \widehat{e} + H_s \cdot \widehat{r}_2$
5 : $T \leftarrow P_1 + H_d \cdot \mathsf{domain} + \sum_{i=1}^{R} H_i \cdot \mathtt{hash}(m_i)$
6 : $C_2 \leftarrow T \cdot c - D \cdot \widehat{r}_3 + H_s \cdot \widehat{s} + \sum_{j=1}^{U} H_j \cdot \widehat{m}_j$
7 : $c_v \leftarrow \mathtt{hash}(A' \| \bar{A} \| D \| C_1 \| C_2 \| R \| i_1 \| ... \| i_R \| m_{i1} \| ... \| m_{iR} \| \mathsf{domain} \| \mathsf{ph})$
8 : **if** $c \neq c_v$ **then**
9 : **return** false
10 : **fi**
11 : **if** $A' = 1$ **then**
12 : **return** false
13 : **fi**
14 : **if** $e(A', W) \cdot e(\bar{A}, -P_2) \neq 1$ **then**
15 : **return** false
16 : **fi**
17 : **return** true

4.3.2 Extension

Pedersen commitment

The commit operation performs a Pedersen commitment to blind messages that, when signed, are unknown to the signer. The algorithm returns a Pedersen commitment from a vector of messages and a generated blinding factor that will be used to unblind the blind signature. This operation is also called *Pre-BlindSign*.

$Commit$ $((H_s, H_d, H_1, ..., H_L), (m_1, ..., m_U), (i_1, ..., i_U))$

1 : $s' \leftarrow\$ \mathbb{Z}_r$
2 : $\mathsf{commitment} \leftarrow H_s \cdot s' + \sum_{i=1}^{U} H_i \cdot \mathtt{hash}(m_i)$
3 : **return** $s', \mathsf{commitment}$

Commitment proof generation

The commitment proof generation computes a zero-knowledge proof of knowledge of committed messages. It may be included a nonce in the proof, which will provide freshness to the generated proof while making it interactive. The commitment proof is also called *blind messages proof* and is needed before performing a blind signature.

$CommitmentProofGen$ (commitment, s', $(H_s, H_d, H_1, ..., H_L), (m_1, ..., m_U), (i_1, ..., i_U)$, nonce)

1 : $\widetilde{s} \leftarrow\$ \mathbb{Z}_r$
2 : **for** i **in** $(1, ..., U)$
3 : $\quad \widetilde{r}_i \leftarrow\$ \mathbb{Z}_r$
4 : **endfor**
5 : $\widetilde{U} \leftarrow H_s \cdot \widetilde{s} + \sum_{i=1}^{U} H_i \cdot \widetilde{r}_i$
6 : $c \leftarrow \mathtt{hash}(\text{commitment} \| \widetilde{U} \| \text{nonce})$
7 : $\widehat{s} \leftarrow \widetilde{s} + c \cdot s' \ mod \ r$
8 : **for** i **in** $(1, ..., U)$
9 : $\quad \widehat{r}_i \leftarrow \widetilde{r}_i + c \cdot \mathtt{hash}(m_i) \ mod \ r$
10 : **endfor**
11 : **return** $\pi = (c, \widehat{s}, \widehat{r})$

Commitment proof verification

The commitment proof verify operation checks whether a proof of knowledge of committed messages is valid.

$CommitmentProofVerify$ (commitment, proof, $(H_s, H_d, H_1, ..., H_L), (i_1, ..., i_U)$, nonce)

1 : $(c, \widehat{s}, \widehat{r}) = \text{proof}$
2 : $\widehat{U} \leftarrow \text{commitment} \cdot (-c) + H_s \cdot \widehat{s} + \sum_{i=1}^{U} H_i \cdot \widehat{r}_i$
3 : $c_v \leftarrow \mathtt{hash}(\text{commitment} \| \widehat{U} \| \text{nonce})$
4 : **if** $c = c_v$ **then**
5 : $\quad$ **return** true
6 : **else**
7 : $\quad$ **return** false
8 : **fi**

Blind sign

The blind sign operation computes a blind signature from a secret key over a commitment, known messages, and/or a header included in the domain value. The signer should validate the commitment before signing it, verifying the received proof of knowledge of committed messages. Note that K is the number of known messages.

BlindSign $(\mathsf{sk}, \mathsf{pk}, (H_s, H_d, H_1, ..., H_L),$
$(m_1, ..., m_K), (i_1, ..., i_U), \mathsf{header}, \mathsf{commitment})$

1: $L = K + U$
2: $(i_1, ..., i_K) = (1, ..., L) \backslash (i_1, ..., i_U)$
3: $\mathsf{domain} \leftarrow \mathtt{hash}(\mathsf{pk} \| L \| H_s \| H_d \| H_1 \| ... \| H_L \| \mathsf{header})$
4: $e, s'' \leftarrow\!\!\$\, \mathbb{Z}_r$
5: $B \leftarrow \mathsf{commitment} + P_1 + H_s \cdot s'' + H_d \cdot \mathsf{domain} + \sum_{j=1}^{K} H_i \cdot \mathtt{hash}(m_i)$
6: $A \leftarrow B \cdot \frac{1}{\mathsf{sk} + e}$
7: **return** $\sigma = (A, e, s'')$

Unblind signature

The unblind signature operation unblinds the received blind signature with the blinding factor returned by the Pedersen commitment. The unblinded signature is a valid signature over the messages that were committed and the known messages. The resulting signature should be verified after the unblinding.

UnblindSignature $(\mathsf{blind\_signature}, s')$

1: $(A, e, s'') = \mathsf{blind\_signature}$
2: $s \leftarrow s' + s''\ mod\ r$
3: **return** $\sigma = (A, e, s)$

4.3.3 Scheme design and implementation

The BBS+ signature scheme is implemented over the BLS12-381 pairing-friendly elliptic curve. Since the scheme is based on pairings and elliptic curves, several classes and methods have been developed to deal with fields, field extensions, elliptic curve points, and pairings.

Elliptic curve points can be defined with different coordinate systems as follows [51]:

- **Affine coordinates**. The traditional point representation with two coordinates, (x, y).
- **Projective coordinates**. They introduce a third coordinate to represent a point as (x, y, z). A Projective point represents the Affine point $(\frac{x}{z}, \frac{y}{z})$.
- **Jacobian coordinates**. They introduce a third coordinate too. A Jacobian point (x, y, z) represents the Affine point $(\frac{x}{z^2}, \frac{y}{z^3})$.

In this implementation work, in addition to Affine coordinates, Jacobian coordinates are employed for optimization reasons, so that point addition does not require slow modular inversion.

To perform pairings, *Miller loop* and *final exponentiation* algorithms have been developed. The Miller loop [61] is a double-and-add algorithm employed to calculate a pairing, while the final exponentiation is used to map the result of the Miller loop to a unique element of $\mathbb{F}_{q^{12}}$. To optimize the verification operations that multiply two pairings, a multi-pairing algorithm can be developed. This algorithm allows multiplying the two results of Miller loops and performing only one final exponentiation. With such optimization that saves a final exponentiation call, a performance improvement by a factor of 1.7 is obtained. Therefore, the verification time is almost halved.

Where the cryptographic hash function `hash` is specified in the schema, it is implemented with the primitive from the SHA-3 family SHAKE256 [62]. SHAKE256 is an *extendable-output function (XOF)*, which is different from hash functions because it produces a variable-size output but it is possible to use it similarly in order to output fixed-size values.

Generators points $(H_s, H_d, H_1, ..., H_L)$ are chosen as random points in $\mathbb{G}_1$. To implement this, hashing to the elliptic curve is required and it is done by using the *simplified SWU algorithm* [63] applied to the BLS12-381 curve [64]. Hence, random points are obtained by hashing (mapping) random bytes to points in $\mathbb{G}_1$ (or $\mathbb{G}_2$, in other possible cases).

4.3.4 Results

The implementation of the BBS+ signature scheme was done in Python while following the current standardization draft [59]. The performance values are means of $n = 30$ iterations and are computed with one message (one credential attribute). They are high because the programming language Python is slower than other high-level languages since it manages the memory itself and the code is interpreted run-time instead of being compiled. Moreover, no optimized libraries that wrap C code to Python were employed. Note that the tests were run on a machine with a `Intel(R) Core(TM) i5-8250U CPU @ 1.60 GHz` and a `8 GB RAM`.

Performance table

Operation	*Time (in milliseconds)*
Key generation	151.81 ms
Sign	57.02 ms
Proof generation	353.62 ms
Proof verify	3478.74 ms

Table 4.1. Performance values of BBS+ scheme with BLS12-381 in Python.

Sizes table

Element	*Size (in bytes)*
Private key	32 B
Public key	96 B
Elliptic curve point	48 B
Signature	112 B
Proof	304 B + 32 B/msg

Table 4.2. Size values of BBS+ scheme with BLS12-381.

4.4 Comparison with CL Signatures

Before BBS+ signatures, Camenisch-Lysyanskaya signatures [56] were the reference digital signature for implementing a scheme with selective disclosure, blind signatures, and zero-knowledge proofs, which are all useful features for signing digital credentials. The CL signature scheme allows for the efficient implementation of two protocols:

- A protocol for computing a digital signature in a two-party computation way, which allows a signer to issue a signature without knowing all the messages being signed.
- A protocol for proving knowledge of a digital signature in a zero-knowledge way, which allows a signature holder to prove its possession of a signature and signed messages, without revealing them.

Both protocols can be developed with BBS+ signatures, where the first one is the blind signature operation and the second one is the zero-knowledge proof of knowledge of a signature. Moreover, in the BBS+ scheme, there is no need to use *range proofs*, which are required by the CL scheme.

The CL signature scheme is based on RSA, which is based on the *prime factorization problem*. Given two prime numbers p, q and the resulting $n = pq$, the prime

factorization problem is to find two numbers p', q' such that $n = p'q'$. Hence, RSA signatures and CL signatures are based on the assumption for which it is difficult to factorize n. Schemes based on this problem require very large prime numbers to be secure. This leads to the following two issues:

- **Largeness**. Keys need to be large to be secure and signatures become big too.
- **Slowness**. The key generation is slow because it is difficult to find large prime numbers. Due to largeness, signing is slower than other signature schemes.

On the other hand, BBS+ keys, signatures, and proofs are smaller and the BBS+ signature scheme is more performant than the CL signature scheme.

Here are two comparison tables[1] between CL signatures and BBS+ signatures. The performance values are means of $n = 30$ iterations and are computed with one message (one credential attribute). The BBS+ size values from the resource are different from the ones of the implementation related to this work. Anyway, they are similar enough to get a valid comparison with CL signatures.

Performance table

	CL signatures	*BBS+ signatures*
Key generation	8800 ms	2.69 ms
Sign	93 ms	1.43 ms
Proof generation	13 ms	5.61 ms
Proof verify	11 ms	4.61 ms

Table 4.3. Performance comparison between CL and BBS+ schemes.

Sizes table

	CL signatures	*BBS+ signatures*
Private key	256 B	48 B
Public key	771 B + 257 B/msg	293 B + 97 B/msg
Signature	672 B	193 B
Proof	696 B + 74 B/msg	312 B + 104 B/msg

Table 4.4. Size comparison between CL and BBS+ schemes.

[1] From https://docs.google.com/presentation/d/1JRzlzS3Y3NTm_NPzxlnud7xIDmGL_9aHHX55MMwvtlU.

Chapter 5

Privacy-Preserving Credentials

5.1 Anonymous Credentials

Prior to the introduction of self-sovereign identity, verifiable credentials, and their focus on privacy, a research topic that studies how to preserve or possibly enhance privacy in digital credentials was already present in the literature. Users have many relationships, which are made of interactions that generate data about users, with different parties. In these relationships, the parties may be identified through two extreme approaches. The first one requires users to show unique identifiers that can authenticate them, while the second one lets users be anonymous and protect their privacy.

A digital credentials solution where parties can authenticate themselves while preserving privacy is possible [65]. This type of credentials is called *anonymous credentials* or *zero-knowledge credentials*. In this work, these credentials are considered verifiable credentials, thus the ecosystem consists of the trust triangle: issuer, holder, and verifier. The goal is to remove the ability to identify and track holders (users) to even colluding issuers and verifiers. In addition, holders should be able to control their identities and credentials.

5.1.1 Design goals and choices

An anonymous credential system should achieve the following features regarding security, privacy, and usability [66]:

- **Unforgeability**. Holders cannot forge issuer-generated credentials.
- **Unlinkability**. One or more verifiers cannot link the activities of holders across them.
- **Minimal disclosure**. Holders can selectively reveal only the necessary information to establish relationships with verifiers.

- **Predicates**. Holders can prove they hold valid information on conditions requested by verifiers, without revealing them. E.g., given a holder's birthdate in the credential, it is presented only the information that the holder's age is greater than 18.
- **Private holder binding**. Holders can be bound to their credentials without creating a correlating factor that needs to be revealed upon presentations.
- **Privacy against issuers**. Issuers cannot learn whether holders have interacted with verifiers, or with a certain verifier.
- **Revocability**. Issuers can revoke previously-issued credentials without interactions with holders.
- **Offline verification**. Verifiers can verify whether a credential is authentic and not-revoked (valid) without any cooperation.
- **Efficiency**. All the operations involved in the issuance, management, presentation, and verification of anonymous credentials must be efficient.

The implemented anonymous credential system obtains unforgeability because BBS+ signatures and proofs are unforgeable according to security analysis.

Unlinkability is firstly achieved by blinding the issuer's signature, which is randomized when the signature proof of knowledge is generated by the holder. Hence, the static and unique value of the digital signature inside the credential is avoided. Secondly, along with minimal disclosure, the selective disclosure property of BBS+ signatures enables the non-disclosure of unnecessary or unique information that may lead to correlation. Moreover, for each relationship, the holders use one or more different keypairs (different secret keys and public keys associated with different DIDs).

Predicates proofs are not supported by the BBS+ signature scheme as it is and are not included in the implementation. At the moment, their function may be replaced by issuers adding an extra attribute in credentials, according to what verifiers would like to check. E.g., issuers may add an attribute `isOver18: {true, false}` together with an attribute `birthdate`, which may not be revealed. To develop predicates, other zero-knowledge proofs algorithms may be employed, such as zk-SNARKs or Bulletproofs.

Private holder binding is a fundamental feature for verifying the ownership of credentials. It is achieved with the Linked Blinded Secret mechanism, which will be explained in the next section. The mechanism adds some steps to the credentials issuance protocol and it is omitted in this section for simplicity reasons.

In this construction, issuers do not learn anything about holders' interactions if the information being shared with verifiers, with which they may collude, is as generic as possible.

The revocation of credentials by issuers, as well as the verification of whether credentials are authentic and valid by verifiers, can be done without interactions.

There are different approaches to privacy-preserving revocation that will be discussed in more detail in this chapter.

Regarding efficiency, the BBS+ signature scheme has short keys, signatures, and proof, whilst the execution times are in the order of units of milliseconds, thus are acceptable.

5.1.2 Protocols

The stages of the protocols of the anonymous credential system are described below. The interaction between an issuer and a holder, which corresponds to the credential issuance, is simplified because the private holder binding and the revocation mechanisms are omitted for now. All the steps present in these protocols may be automated.

Credential issuance

The credential issuance phase involves an issuer and a holder. It may start upon a request from the holder or directly from the issuer. The holder identification may be accomplished in a variety of ways: through the presentation of another credential from a trusted issuer, after an action by the holder, in person in the case of humans, or by a system administrator in the case of things. Then, the issuer creates a credential with claims about the holder, signs the credential, and sends it as a verifiable credential (VC) to the holder. The holder, once the credential is received, verifies the signature on it and finally stores both of them.

Issuer-Holder interaction

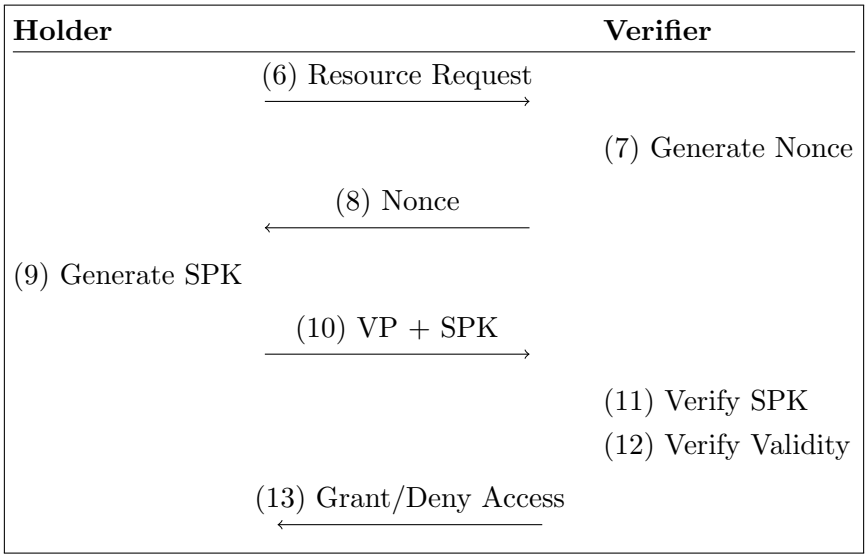
Issuer		**Holder**
(1) Issue VC		
(2) Sign VC		
	(3) VC + Signature $\longrightarrow$	
		(4) Verify Signature
		(5) Store VC

Credential presentation

The credential presentation phase involves a holder and a verifier. When the holder wants to access a certain resource offered by the verifier, it gets the authorization to do it by presenting a credential issued by a trusted issuer. The holder starts sending a request to the verifier, who responds with a cryptographic nonce. The nonce is used by the holder to generate a signature proof of knowledge (SPK),

which proves the knowledge of the issuer's signature on the credential and the signed attributes without revealing either of them. A truly anonymous credential should not disclose any attributes but, in some cases, it is necessary to disclose a small subset of them. Then, the holder sends the selectively disclosed credential as a verifiable presentation (VP) and the proof to the verifier. The verifier verifies the proof with the issuer's public key and checks whether the credential is valid or revoked before granting or denying access to the requested resource.

Holder-Verifier interaction



5.1.3 Known weaknesses

In this anonymous credential system, there may be known weaknesses that can be mitigated, depending on the use cases. Some of these weaknesses are inherent in any system with such features [66].

- **Malicious issuer**. A malicious issuer may issue false credentials or revoke a credential without justifications. A list of trusted issuers should be maintained.
- **Malicious verifier**. A malicious verifier may ask for many unnecessary credential attributes. The final decision on whether or not to share some attributes, after the requests of verifiers, should be up to the holders.
- **Credentials sharing**. A holder may give a personal credential to another peer. In case a credential should not be sharable, the holder's secret used during the Linked Blinded Secret protocol to obtain the binding between a holder and a credential may limit this weakness. It may limit even more if the secret is stored inside a secure element, whence it could not be read to be shared.

- **Issuer leakage**. Anyone with access to an issuer's database can learn information about holders and their credentials. This weakness may be prevented by issuers storing as little information as possible and encrypting their databases.

- **Holder leakage**. Anyone with access to a holder's system, where the holder may be a human, an organization, or a thing, can learn information about the holder and its credentials. Encryption of credentials while at rest, as well as while they are in transport, may mitigate this problem. Moreover, cryptographic secret keys, which include also the secret of the Linked Blinded Secret, may be compromised if they are not stored in a tamper-resistant secure element.

- **Bad randomness usage**. The randomness used during the protocols may be from untrusted sources and feasible to brute force. In this system, the randomness must be infeasible to guess and generated from a secure and trusted source, such that signatures and proofs are harder to forge or break.

- **Issuer-Verifier collusion**. When an issuer and a verifier collude with each other, they may identify a holder if it shares with the verifier identifiable information that is obviously known by the issuer.

- **Verifier-Verifier collusion**. When verifiers collude with each other, they may correlate the activities and the relationships of a certain holder if it shares with them a static attribute that can be traced.

- **Linkable issuance**. An issuer that issues few credentials may lead to holders' linkability when colluding with a verifier. The issuers should be active and issue many credentials with information as generic as possible.

5.2 Linked Blinded Secret

The Linked Blinded Secret (LBS), also called *link secret* [56], [67], enables more trusted interactions with verifiable credentials because it gives verifiers the ability to check that a credential was issued to the holder presenting it.

The Linked Blinded Secret is a large random number, a *master secret*, which is then committed using a cryptographic commitment algorithm. The commitment allows the holder to prove it knows the secret without revealing it. The commitment is sent by the holder to the issuer that signs it together with the credential attributes, while the secret is never shared. It is called this way because of its features:

- **Linked**. The same master secret is inserted and signed in different credentials for two purposes: to link the holder to its credentials and to link the credentials to each other in case it is needed.

- **Blinded**. The master secret is opaqued and randomized by a blinding commitment algorithm at every credential issuance.
- **Secret**. The master secret is kept private and is never revealed by the holder. Only commitments and proofs of knowledge of the committed secret are shared.

As mentioned before, the Linked Blinded Secret has two purposes:

- **Proof of holder binding**. Since the holder is the only one who knows the master secret, no one else is able to generate a proof to present the credential. The secret is not shared, thus no linkable attributes such as DIDs are forwarded: private holder-credential binding is achieved.
- **Proof of credential linking**. The master secret is the same across all credentials owned by the same holder, which may combine more credentials in a single presentation. Each credential contains a different Linked Blinded Secret but the holder can prove that the secret behind those values is the same, without revealing it. This shows that all the credentials presented are linked to the same holder without producing a correlatable identifier.

The second purpose, proof of credential linking, is not supported by the system implemented in this work. This decision was made since it was not a required feature. Nevertheless, it may be added by implementing a proof of equality of discrete logarithms. Which is, in other words, an algorithm that proves that the secret behind different commitments is the same.

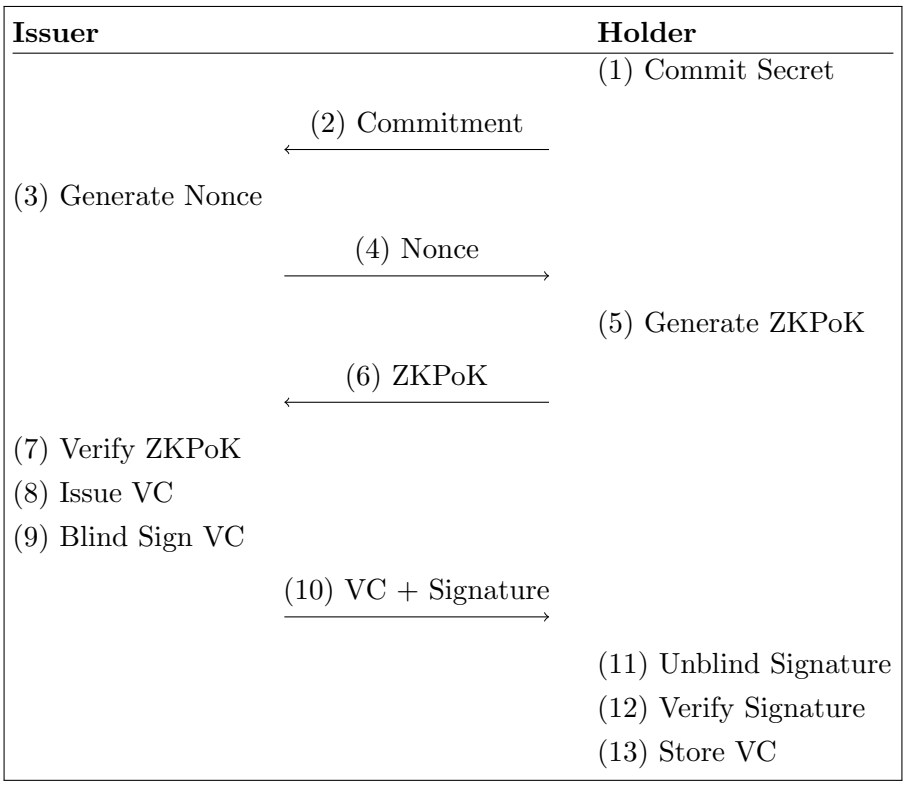
5.2.1 Protocol design and implementation

The private holder binding feature obtained with the Linked Blinded Secret adds some steps to the basic version of the credential issuance protocol. Whilst the credential presentation protocol remains the same for both the messages exchanged and the operations to be performed. The only difference is that the secret must be in the set of undisclosed messages when generating a signature proof of knowledge.

In this protocol, the holder has a master secret that must not be revealed. The holder computes a Pedersen commitment to the secret and then sends the commitment to the issuer. The issuer wants to verify whether the holder knows the secret behind the commitment, thus it generates and sends a nonce to the holder. The holder computes and sends a zero-knowledge proof of knowledge (ZKPoK) of committed messages with the received nonce, where the message is the secret. The issuer verifies the proof and, if it is successful, issues the verifiable credential (VC) with a blind signature. The issuer blind signs the credential because it knows all the attributes except the commitment, which is a "blind attribute". When the holder receives the verifiable credential, it unblinds the blind signature in order to obtain a valid signature over all the attributes plus the secret behind the commitment. In

fact, to generate a signature proof of knowledge, the secret must be known, not the commitment. Finally, the holder verifies the unblinded signature and stores the verifiable credential.

Issuer-Holder interaction



Due to this protocol, only the holder can generate the signature proof of knowledge to present the credential because it is the only one who knows the secret, which has been blinded and linked. The issuer knows the random commitment, while the verifier knows nothing. Moreover, only the holder knows the valid signature over the credential attributes and the secret because the issuer knows the blind signature (never reused), while the verifier knows the random signature proof. In this way, private holder binding and unlinkability are achieved.

5.3 Revocation

Anonymous credential systems bring authentication to access digital resources or services while preserving privacy. Issuers provide holders with credentials that certify their attributes and permissions. When using credentials to access a resource or a service, it is necessary to ensure their validity, in addition to checking the information they contain and their authenticity.

The revocation of a credential must be supported for many reasons [68]: the holder may have lost its permissions, the secrets underlying the credential (secret

key and master secret) may have been compromised, or the claims may have become outdated. Though, the privacy requirement makes the development of a revocation mechanism more difficult.

In order to design a revocation mechanism, the number of total users, the frequency of credential usage, and the revocation speed are parameters that must be taken into account.

5.3.1 Accumulator

In non-anonymous credential systems, where privacy is not a concern, a possible solution to revocation is a *revocation list*. It is a list of all the serial numbers of revoked credentials, similar to a *certificate revocation list (CRL)* [69], which can be queried both online and offline. This solution cannot work for anonymous credentials, because revealing a unique serial number would lead to linkability. The general idea may still be applied if holders prove that the credential's serial number is among the list of valid numbers, without revealing it.

Dynamic accumulators [70] accumulate all the valid serial numbers into a single value that is published by issuers. Moreover, they provide an algorithm that enables holders to prove that the serial numbers of their credentials are contained in the accumulated value. Whenever a credential is revoked, the accumulator value is updated by removing the credential's serial number and then the new value is published.

Accumulator-based schemes require that holders maintain the accumulator value updated to be able to generate the validity proofs. In addition to proving the possession of the presented credential, holders have to prove also its validity by referring to the accumulator.

5.3.2 Signature update

A different approach is to limit credentials lifetime by specifying a validity time frame as a date and periodically re-issue non-revoked and non-expired credentials [68]. The credentials are valid only for a specific period, called *epoch*, which may be days, hours, or even minutes, depending on the requirements.

Issuers and holders run the credential issuance protocol only once and then the issuer may update some credential attributes, including the validity time frame, and publish them. Holders can retrieve the new values and the new BBS+ signature, which only changed for the value A from the triplet (A, e, s). Therefore, holders recompute their credentials to make them valid again. Verifiers do not need to check any external revocation list, thus the presentation of credentials is efficient.

This solution introduces an additional burden on the issuers' infrastructure in terms of bandwidth, computational power, and availability. Though, the required work for issuers is comparable to other solutions because issuers may pre-compute the update offline and then periodically publish the new values.

Very short epochs, e.g. one hour, demand a lot of activity from the issuers because they have to provide updates for all non-revoked credentials. A fine granularity should be preferred to have more flexibility and speed of action. To handle it, holders could request updates from issuers only when needed, instead of issuers publishing updates.

The update signature algorithm implemented with the BBS+ signature scheme is performed by issuers that know their secret key, the signature to be updated, the generators, the old attribute to be replaced and its index within the array. Then, the new value A of the signature (A, e, s) is computed by removing the old attribute and adding the new one. Note that, in order to know where to put the valid time frame attribute, a credential schema must be agreed upon between issuers and verifiers. Eventually, the schema may be published on a verifiable data registry.

$UpdateSignature\ (\mathsf{sk}, \mathsf{signature}, (H_1, ..., H_L), m_{old}, m_{new}, index)$

1 : $(A, e, s) = \mathsf{signature}$

2 : $B \leftarrow A \cdot (\mathsf{sk} + e\ mod\ r)$

3 : $B \leftarrow B - (H_{index} \cdot \mathtt{hash}(m_{old})) + (H_{index} \cdot \mathtt{hash}(m_{\mathbf{new}}))$

4 : $A \leftarrow B \cdot \dfrac{1}{\mathsf{sk} + e}$

5 : **return** A

5.4 Use cases

There are several use cases for anonymous credentials. They have the same application scenarios as verifiable credentials, but with increased emphasis and additional focus on privacy. They are useful in digital contexts and relationships, where it is possible to automate some steps and improve usability, security, and privacy of services that require credentials. These credentials, as well as self-sovereign identity in general, can be applied to both human-related and thing-related systems.

The following is a list of different use cases for anonymous verifiable credentials in human-related scenarios:

- **Identity card**. Credentials can be used to verify the holders' identity by revealing only the information needed by verifiers. It may be enough to know that the credential was issued by a certain country.
  E.g. An individual wants to access a European-only online service. They can authenticate themselves by showing that their ID card is issued by a certain country, which is part of the European Union.

- **Employment certificate**. Credentials can be used to verify current and past employment, where the holders are the employees and the issuers are the employers.
  E.g. An individual wants to apply for a new job. They prove the skills they

acquired during their past jobs, without revealing their salary to prospective employers.

- **Educational certificate**. Credentials can be used to certify attending a course, passing an exam, or acquiring a degree.
  E.g. An individual wants to show that they have a master's degree in a certain subject without revealing their name and their grade.

- **Sign-up and Login**. Credentials can be used to quickly sign up for services. Then, service providers may issue credentials to users to enable them to a quick and password-less login.

While the following list contains use cases related to things and machines:

- **TPM attestation**. Anonymous credentials with BBS+ signatures [43] can be used during the direct anonymous attestation (DAA) protocol [71]. This protocol performs remote authentication of the trusted platform module (TPM) [72], which is a hardware module that securely creates and stores cryptographic keys and confirms that the operating system and firmware are what they are supposed to be.

- **Security token**. Anonymous verifiable credentials can be used as security tokens such as JWT-based [73] access tokens used in the OAuth 2.0 protocol [74], which is an authorization framework that enables a third-party application to obtain access to an HTTP service. They would provide unlinkability as well as replay protection.

- **IoT identity**. Credentials can be used to secure Internet of things (IoT) devices, such as automobiles, thermostats, sensors, medical devices, and door locks. They may be exchanged in several IoT scenarios: machine-to-machine communication, machine-to-person communication, and digital twin. Self-sovereign identity would be useful for identifying and authorizing devices, managing device updates, maintaining secure communications, and ensuring data privacy and integrity.

5.5 Future work

The work done on the project associated with this document resulted in the implementation of an anonymous credential system with holder binding and revocation.

The next steps to improve the work may introduce predicates proofs employing zk-SNARKs or Bulletproofs and proofs of credential linking using the Linked Blinded Secret. In addition, a second revocation mechanism with accumulators would be useful to have more flexibility depending on system requirements. Eventually, the code could be optimized by developing it in another programming language and then integrated into a platform or application to provide a full-stack self-sovereign identity with anonymous verifiable credentials.

Chapter 6

Conclusions

The growing concern about privacy and centralization in a cyber-world has led to the creation of self-sovereign identity, a decentralized user-centric digital identity solution. However, self-sovereign identity alone is only a paradigm, which must be supported by data formats, cryptographic algorithms, protocols, and standardizations.

Verifiable credentials play an important role. They are a mechanism for representing digital credentials in a secure, privacy-respecting, and machine-verifiable way. To make them truly privacy-friendly, zero-knowledge proofs need to be introduced.

The required proofs are implemented using the BBS+ signature scheme, which supports properties such as selective disclosure and unlinkable zero-knowledge proofs of knowledge. BBS+ signatures are crucial to the development of the work carried out in this document. They are the basis of the developed anonymous credential system that enables parties to have more trusted digital interactions with each other. Trust can be established while preserving privacy through data minimization and unlinkability. In addition to these features, the developed anonymous credential system has a focus on security: signatures on credentials cannot be forged, credentials can be revoked by issuers, and credentials are bound only to their actual holders. Particular effort was put into this last property, the holder binding, which requires the use of several cryptographic algorithms included in the BBS+ signature scheme and its extension.

The goal of this work was to create a system for issuing, presenting, and verifying credentials that would preserve the privacy of their holders. The developed anonymous credential system achieves this goal if it is used by the parties involved with caveats. The problem is that it could lead to excessive anonymity, even where there should not be as it might be useful to track malicious behaviors. However, even though this system was developed following the *privacy by design* principle, it is a flexible tool that can be shaped according to different requirements.

Appendix A

User's manual

Installation and execution

Install the code

The code developed during this work is written in `Python 3.10`, thus the user needs to install it on its machine. Once the user has the directory with the code, some steps must be done to install it with its dependencies and run it.

Firstly, the user has to open a terminal and has to change its working directory to the one with the code, by entering:

```
cd BBSPlus
```

Then, the user must install the external dependencies that are not included in the Python Standard Library. They can do it by using `pip`, the package installer for Python.

```
pip install pycryptodome
pip install base58
```

Alternatively, instead of installing the external libraries one by one, the user can type the following command to automatically install all the requirements:

```
pip install -r requirements.txt
```

Use the code

To integrate BBS+ signatures developed in this code into a novel or existing project it is necessary to have the `bbs` folder.

Then, it is possible to import the class with the BBS+ signatures-related functions, which is in the file `bbs.py`, by typing:

```
from bbs.bbs import BBS
```

After that, the user can call the functions they want to use.

bbs.py

It contains operations from the BBS+ signature scheme and its extension.

`class BBS`:
This class defines the BBS+ signature scheme and contains the functions it describes.

`def keyGen(self, ikm)`:
Generate a secret key (private key).
Input: `ikm`, input key material, a stream of bytes
Output: Private key

`def skToPk(self, sk)`:
Generate a public key from a secret key and a generator in $\mathbb{G}_2$.
Input: `sk`, secret key (private key)
Output: Stream of bytes

`def keyValidate(self, PK)`:
Check if a public key is valid.
Input: `PK`, public key, a stream of bytes
Output: Boolean

`def createGenerators(self, dst, seed, length)`:
Create a set of generators in $\mathbb{G}_1$.
Input:

- `dst`, domain separation tag, a stream of bytes
- `seed`, a stream of bytes
- `length`, an integer

Output: List of jacobian points

`def sign(self, sk, PK, H_s, H_d, messages, H, header)`:
Compute a signature from a secret key and an array of messages plus an optional header.
Input:

- `sk`, secret key (private key)
- `PK`, public key, a stream of bytes
- `H_s`, a jacobian point
- `H_d`, a jacobian point
- `messages`, a list of bytes
- `H`, a list of jacobian points
- `header`, a stream of bytes

Output: `signature`, a stream of bytes

`def verify(self, PK, H_s, H_d, signature, messages, H, header)`:
Check that a signature is valid.
Input:

- `PK`, public key, a stream of bytes

- `H_s`, a jacobian point
- `H_d`, a jacobian point
- `signature`, a stream of bytes
- `messages`, a list of bytes
- `H`, a list of jacobian points
- `header`, a stream of bytes

Output: Boolean

`def spkGen(self, PK, H_s, H_d, signature, messages, H, revealed, header, ph)`:
Compute a zero-knowledge proof of knowledge of a signature, while optionally selectively disclosing messages from the signed set.
Input:

- `PK`, public key, a stream of bytes
- `H_s`, a jacobian point
- `H_d`, a jacobian point
- `signature`, a stream of bytes
- `messages`, a list of bytes
- `H`, a list of jacobian points
- `revealed`, a list of integers
- `header`, a stream of bytes
- `ph`, presentation header, a stream of bytes

Output: `spk`, a stream of bytes

`def spkVerify(self, PK, H_s, H_d, spk, messages, H, revealed, header, ph)`:
Check that a zero-knowledge proof of knowledge of a signature is valid. Only the disclosed messages are in the input.
Input:

- `PK`, public key, a stream of bytes
- `H_s`, a jacobian point
- `H_d`, a jacobian point
- `spk`, a stream of bytes
- `messages`, a list of bytes
- `H`, a list of jacobian points
- `revealed`, a list of integers
- `header`, a stream of bytes
- `ph`, presentation header, a stream of bytes

Output: Boolean

`def preBlindSign(self, H_s, messages, H, unrevealed)`:
Blind messages with Pedersen commitment for blind signatures.
Input:

- `H_s`, a jacobian point
- `messages`, a list of bytes

- `H`, a list of jacobian points
- `unrevealed`, a list of integers

Output: Tuple

- `s_prime`, an integer
- `commitment`, a jacobian point

`def blindMessagesProofGen(self, H_s, commitment, s_prime, messages, H, unrevealed, nonce)`:
Compute a zero-knowledge proof of committed messages to be verified before computing blind signatures.
Input:

- `H_s`, a jacobian point
- `commitment`, a jacobian point
- `s_prime`, an integer
- `messages`, a list of bytes
- `H`, a list of jacobian points
- `unrevealed`, a list of integers
- `nonce`, a stream of bytes

Output: `nizk`, a stream of bytes

`def blindMessagesProofVerify(self, H_s, commitment, H, unrevealed, nizk, nonce)`:
Check that a zero-knowledge proof of committed messages is valid.
Input:

- `H_s`, a jacobian point
- `commitment`, a jacobian point
- `H`, a list of jacobian points
- `unrevealed`, a list of integers
- `nizk`, a stream of bytes
- `nonce`, a stream of bytes

Output: Boolean

`def blindSign(self, sk, PK, H_s, H_d, commitment, messages, H, unrevealed)`:
Compute a blind signature from a secret key, a commitment, and an array of known messages.
Input:

- `sk`, secret key (private key)
- `PK`, public key, a stream of bytes
- `H_s`, a jacobian point
- `H_d`, a jacobian point
- `commitment`, a jacobian point
- `messages`, a list of bytes
- `H`, a list of jacobian points
- `unrevealed`, a list of integers

Output: `blind_signature`, a stream of bytes

`def unblindSign(self, blind_signature, s_prime)`:
Unblind a blind signature to make it valid.
Input:

- `blind_signature`, a stream of bytes
- `s_prime`, an integer

Output: `signature`, a stream of bytes

`def octetsToSignature(self, signature_octets)`:
Decode an octet string (bytes) into a signature.
Input: `signature_octets`, a stream of bytes
Output: Tuple

- `A`, a jacobian point
- `e`, an integer
- `s`, an integer

`def signatureToOctets(self, A, e, s)`:
Encode a signature into an octet string (bytes).
Input:

- `A`, a jacobian point
- `e`, an integer
- `s`, an integer

Output: Stream of bytes

`def octetsToSpk(self, proof_octets)`:
Decode an octet string (bytes) into an SPK.
Input: `proof_octets`, a stream of bytes
Output: Tuple

- `A_prime`, a jacobian point
- `A_bar`, a jacobian point
- `D`, a jacobian point
- `c`, an integer
- `e_cap`, an integer
- `r2_cap`, an integer
- `r3_cap`, an integer
- `s_cap`, an integer
- `m_cap`, a list of integers

`def spkToOctets(self, proof)`:
Encode an SPK into an octet string (bytes).
Input: `proof`, a tuple

- `A_prime`, a jacobian point
- `A_bar`, a jacobian point
- `D`, a jacobian point
- `c`, an integer

- `e_cap`, an integer
- `r2_cap`, an integer
- `r3_cap`, an integer
- `s_cap`, an integer
- `m_cap`, a list of integers

Output: Stream of bytes

`def octetsToNizk(self, proof_octets):`
Decode an octet string (bytes) into a NIZK proof.
Input: `proof_octets`, a stream of bytes
Output: Tuple

- `c`, an integer
- `s_cap`, an integer
- `r_cap`, a list of integers

`def nizkToOctets(self, nizk):`
Encode a NIZK proof into an octet string (bytes).
Input: `nizk`, a tuple

- `c`, an integer
- `s_cap`, an integer
- `r_cap`, a list of integers

Output: Stream of bytes

`def updateSignature(self, sk, signature, old_message, new_message, H, index):`
Update a signature with a new message. This function is not present in the IETF standardization.
Input:

- `sk`, secret key (private key)
- `signature`, a tuple (A, e, s)
- `old_message`, a stream of bytes
- `new_message`, a stream of bytes
- `H`, a list of jacobian points
- `index`, an integer

Output: Stream of bytes (A)

Run the tests

There are some tests written in Python to verify and collect the results obtained in terms of performance and size. In addition, there is a script to test the specific features offered by BBS+ signatures and the developed module.

To retrieve performance data about the operations that BBS+ signatures offer, the user must execute the following command:

```
python test_performance.py
```

Note that the data produced by this test are used to populate the table 4.1.

To retrieve data regarding the sizes of the elements produced by BBS+ signatures, such as keys, elliptic curve points, signatures, and proofs, the users must execute the following command:

```
python test_sizes.py
```

Note that the data produced by this test are used to populate the table 4.2.

The `main.py` file contains functions to try and test all the BBS+ signature scheme operations developed for the anonymous credential system. To inspect the data produced, a log file `main.log` is created. Type the following command to run the script:

```
python main.py -<option>
```

Where `<option>` is a number from the following options list:

1. Signature
2. Proof of knowledge of a signature
3. Proof of knowledge of committed messages
4. Blind signature
5. Linked Blinded Secret
6. Update signature

Run the demo

A demonstration of the usage of the anonymous credential system has been developed. It consists of three scripts that perform the three roles in the ecosystem, the verifiable credentials trust triangle: issuer, holder, and verifier. To inspect the data produced, a log file for each script (`issuer.log`, `holder.log`, `verifier.log`) is created. In Microsoft Windows, it is possible to run the demo by executing the following command:

```
python demo.py
```

Alternatively, the user can start the three scripts from three different terminals (or tabs), in this order:

```
python issuer.py
python verifier.py
python holder.py
```

Demo explanation

The issuer is initialized by generating its secret key and public key, together with the generators $(H_s, H_d, H_1, ..., H_L)$, where L is the number of attributes of the credentials it issues. Then, it publishes its public key and generators on a verifiable data registry. Moreover, the issuer maintains a list of issued credentials, which is regularly updated, and a list of credential statuses, to know whether a credential is valid or not. The policies about expiration and revocation of credentials are that the expiration date is after one year and the validity time frame is one minute, for demonstration purposes.

The verifier is initialized by retrieving the issuer's public key and generators from the verifiable data registry.

The holder is initialized by generating the master secret for the Linked Blinded Secret and retrieving the issuer's public key and generators.

The demo consists of three different situations between the issuer, the holder, and the verifier:

1. The holder requests a credential from the issuer by sending a commitment and the related zero-knowledge proof. The issuer issues the credential and blind signs it. The holder uses the credential by revealing some attributes with the signature proof of knowledge to access a resource offered by the verifier. If the credential is authentic and valid, the verifier grants access to the holder.

2. After a while, the holder re-uses the same credential to access a resource of the verifier. In this case, the credential is revoked because the validity time frame is elapsed, thus the verifier denies access to the holder.

3. The holder requests the issuer to update its signature over the credential with a new validity time frame. After a check, the issuer updates the signature and the credential. Then, the holder presents again the credential to the verifier and receives the authorization to access the requested resource.

Appendix B

Developer's manual

Modules

Multiple modules are used to provide functionalities and make the code work. Some of them have been developed, while others are included in the Python Standard Library or installed externally.

Python libraries

Among the modules in the Python Standard Library (version 3.10.6), the following are used:

- `secrets`. It generates cryptographically strong random numbers. It is used to generate randomness in BBS+ scheme functions and utility functions.
- `copy`. It is used to copy objects without creating a binding between the source and the destination.
- `json`. It exposes functions to handle JSON objects used in the demo.
- `time`. It provides time-related functions. It is used in the demo to make the holder sleep for demonstration purposes.
- `datetime`. It supplies functions to manipulate dates and times. It is used in the demo to define issuance dates, expiration dates, and validity time frames.
- `logging`. It implements a flexible event-logging system. It is used to build the logger in `logger.py`, which logs the main operations performed and their results during the demo.
- `threading`. It constructs a high-level threading interface. It is used in the demo on the issuer's side to deploy two different servers, one for credentials issuance and one for signatures update.

- `socket`. It provides a socket interface to make the issuer, the holder, and the verifier communicate in the demo.
- `typing`. It supports the definition of functions' input and output types.
- `os`, `sys`. They offer operating system-related and interpreter-related utility functions.

External libraries

External dependencies are as few as possible but the following two were necessary to achieve functionalities absent in the Python Standard Library:

- `pycryptodome`. It is a package of low-level cryptographic primitives. It is used to convert long integers to bytes and for SHAKE256.
- `base58`. It is used to encode bytes to strings and decode strings to bytes. Base58 is similar to base64 but it avoids both non-alphanumeric characters and letters which might look ambiguous when printed.

BBS

In the `bbs` folder there are all the modules needed to implement the BBS+ signature scheme based on the BLS12-381 elliptic curve. The `bbs.py` module, which contains operations from the BBS+ signature scheme and its extension, has already been explained in the user's manual.

Note that the implementations of the cryptographic libraries `ec.py`, `fields.py`, `pairing.py`, `key.py`, and `hashes.py` are taken and modified from Chia Network[1]. Whilst the implementation of the optimized simplified SWU map to BLS12-381 $\mathbb{G}_1$ `opt_swu_g1.py` is taken from Algorand[2].

bls12381.py

It defines the class that implements the BLS12-381 elliptic curve.

`class BLS12_381`:
This class defines BLS12-381, a pairing-friendly elliptic curve. It contains its parameters, which are returned by two functions.

`def parameters(self)`:
Return the parameters for the normal elliptic curve (for points in $\mathbb{G}_1$).

[1] Chia Network repository https://github.com/Chia-Network/bls-signatures.

[2] Algorand repository https://github.com/algorand/bls_sigs_ref.

Input: None
Output: Object with the normal elliptic curve values

`def parameters_twist(self):`
Return the parameters for the sextic twist elliptic curve (for points in $\mathbb{G}_2$).
Input: None
Output: Object with the sextic twist elliptic curve values

ec.py

It defines classes and methods for elliptic curve points, arithmetics, and twists.

`class AffinePoint`:
This class defines an affine point. It contains several functions, including the ones to perform arithmetic operations (add, subtract, multiply, negate, equal), conversions, copies, and checks.

`def __init__(self, x, y, infinity, ec)`:
Input:

- `x`, a field element (x coordinate)
- `y`, a field element (y coordinate)
- `infinity`, a boolean to specify whether the point is at infinity
- `ec`, an elliptic curve

Output: Affine point object

`def is_on_curve(self)`:
Check if the point is on the curve, so that $y^2 = x^3 + ax + b$.
Input: None
Output: Boolean

`def to_jacobian(self)`:
Convert the affine point to a jacobian point.
Input: None
Output: Jacobian point

`class JacobianPoint`:
This class defines a jacobian point. It contains several functions, including the ones to perform arithmetic operations (add, subtract, multiply, negate, equal), conversions, copies, and checks.

`def __init__(self, x, y, z, infinity, ec)`:
Input:

- `x`, a field element (x coordinate)
- `y`, a field element (y coordinate)
- `z`, a field element (z coordinate)
- `infinity`, a boolean to specify whether the point is at infinity
- `ec`, an elliptic curve

Output: Jacobian point object

`def is_on_curve(self)`:
Check if the point is on the curve, so that $y^2 = x^3 + ax + b$.
Input: None
Output: Boolean

`def subgroup_check_g1(self)`:
Check if the point is on the $\mathbb{G}_1$ subgroup.
Input: None
Output: Boolean

`def subgroup_check_g2(self)`:
Check if the point is on the $\mathbb{G}_2$ subgroup.
Input: None
Output: Boolean

`def to_affine(self)`:
Convert the jacobian point to an affine point.
Input: None
Output: Affine point

Then, there are generic functions called by internal functions of the two classes.

`def sign_Fq(element, ec)`:
Check the sign of a field ($\mathbb{F}_q$) element.
Input:

- `element`, a field element
- `ec`, an elliptic curve

Output: Boolean

`def sign_Fq2(element, ec)`:
Check the sign of a field ($\mathbb{F}_{q^2}$) element.
Input:

- `element`, a field element
- `ec`, an elliptic curve

Output: Boolean

`def point_to_bytes(point_j, ec, FE)`:
Convert a point to bytes.
Input:

- `point_j`, a jacobian point
- `ec`, an elliptic curve
- `FE`, a field type ($\mathbb{F}_q$ or $\mathbb{F}_{q^2}$)

Output: Bytes

`def bytes_to_point(buffer, ec, FE)`:
Convert bytes to a point.
Input:

- `buffer`, a stream of bytes
- `ec`, an elliptic curve
- `FE`, a field type ($\mathbb{F}_q$ or $\mathbb{F}_{q^2}$)

Output: Jacobian point

`def y_for_x(x, ec, FE)`:
Solve $y = \sqrt{x^3 + ax + b}$.
Input:

- `x`, a field element (x coordinate)
- `ec`, an elliptic curve
- `FE`, a field type ($\mathbb{F}_q$ or $\mathbb{F}_{q^2}$)

Output: Bytes

`def double_point(p1, ec, FE)`:
Perform an affine point doubling.
Input:

- `p1`, an affine point
- `ec`, an elliptic curve
- `FE`, a field type ($\mathbb{F}_q$ or $\mathbb{F}_{q^2}$)

Output: Affine point

`def add_points(p1, p2, ec, FE)`:
Perform an addition between two affine points.
Input:

- `p1`, an affine point
- `p2`, an affine point
- `ec`, an elliptic curve
- `FE`, a field type ($\mathbb{F}_q$ or $\mathbb{F}_{q^2}$)

Output: Affine point

`def double_point_jacobian(p1, ec, FE)`:
Perform a jacobian point doubling.
Input:

- `p1`, a jacobian point
- `ec`, an elliptic curve
- `FE`, a field type ($\mathbb{F}_q$ or $\mathbb{F}_{q^2}$)

Output: Jacobian point

`def add_points_jacobian(p1, p2, ec, FE)`:
Perform an addition between two jacobian points.
Input:

- `p1`, a jacobian point
- `p2`, a jacobian point
- `ec`, an elliptic curve
- `FE`, a field type ($\mathbb{F}_q$ or $\mathbb{F}_{q^2}$)

Output: Jacobian point

`def scalar_mult(c, p1, ec, FE)`:
Perform a scalar multiplication between an integer and an affine point.
Input:

- `c`, an integer
- `p1`, an affine point
- `ec`, an elliptic curve
- `FE`, a field type ($\mathbb{F}_q$ or $\mathbb{F}_{q^2}$)

Output: Affine point

`def scalar_mult_jacobian(c, p1, ec, FE)`:
Perform a scalar multiplication between an integer and a jacobian point.
Input:

- `c`, an integer
- `p1`, a jacobian point
- `ec`, an elliptic curve
- `FE`, a field type ($\mathbb{F}_q$ or $\mathbb{F}_{q^2}$)

Output: Jacobian point

`def G1Generator(ec)`:
Return the generator point in $\mathbb{G}_1$.
Input: `ec`, an elliptic curve
Output: Jacobian point

`def G2Generator(ec)`:
Return the generator point in $\mathbb{G}_2$.
Input: `ec`, an elliptic curve
Output: Jacobian point

`def G1Infinity(ec)`:
Return the point at infinity in $\mathbb{G}_1$.
Input: `ec`, an elliptic curve
Output: Jacobian point

`def G2Infinity(ec)`:
Return the point at infinity in $\mathbb{G}_2$.
Input: `ec`, an elliptic curve
Output: Jacobian point

`def untwist(point, ec)`:
Convert a twisted point (in $\mathbb{F}_{q^2}$) to an untwisted point (in $\mathbb{F}_{q^{12}}$).
Input:

- `point`, an affine point
- `ec`, an elliptic curve

Output: Affine point

`def twist(point, ec)`:
Convert an untwisted point (in $\mathbb{F}_{q^{12}}$) to a twisted point (in $\mathbb{F}_{q^2}$).
Input:

- `point`, an affine point
- `ec`, an elliptic curve

Output: Affine point

`def eval_iso(P, map_coeffs, ec)`:
Evaluate the isogeny over jacobian coordinates for hashing to the BLS12-381 elliptic curve.
Input:

- `P`, a jacobian point
- `map_coeffs`, an isogeny map
- `ec`, an elliptic curve

Output: Jacobian point

`def mx_chain(P)`:
Perform an addition chain for multiplication.
Input: `P`, a jacobian point
Output: Jacobian point

`def mx_chain(P)`:
Perform a fast cofactor clearing.
Input: `P`, a jacobian point
Output: Jacobian point

fields.py

It defines classes and methods to work with fields and field extensions.

`class Fq`:
This class defines an element of a finite field modulo a prime q. It contains several functions, including the ones to perform arithmetic operations (add, subtract, multiply, power, negate), comparisons, conversions, and copies.

`def __init__(self, Q, value)`:
Input:

- `Q`, an integer
- `value`, an integer

Output: Field element object

`class FieldExtBase`:
This class defines an element of an extension of a field. It contains several functions, including the ones to perform arithmetic operations (add, subtract, multiply, power, negate), comparisons, conversions, and copies. Its main variables are:

- `extension`, an integer
- `embedding`, an integer
- `basefield`, a field
- `Q`, an integer
- `root`, a field element of the basefield

`class Fq2`:
This class defines an element of the extension field $\mathbb{F}_{q^2}$. It is a subclass that inherits from the superclass `FieldExtBase` with:

- `extension = 2`
- `embedding = 2`
- `basefield = Fq`

`class Fq6`:
This class defines an element of the extension field $\mathbb{F}_{q^6}$. It is a subclass that inherits from the superclass `FieldExtBase` with:

- `extension = 6`
- `embedding = 3`
- `basefield = Fq2`

`class Fq12`:
This class defines an element of the extension field $\mathbb{F}_{q^{12}}$. It is a subclass that inherits from the superclass `FieldExtBase` with:

- `extension = 12`
- `embedding = 2`
- `basefield = Fq6`

Then, there are other variables useful to compute operations, such as:

- `roots_of_unity`, used for computing square roots in $\mathbb{F}_{q^2}$
- `frob_coeffs`, Frobenius coefficients for raising elements to q^i

hashes.py

It contains methods to perform hash functions.

`class shake_256`:
This class defines the SHAKE256 function.

`def __init__(self, m)`:
Input: `m`, a message (string or bytes)
Output: Updated SHAKE256 context

`def update(self, m)`:
Continue hashing of a message by consuming the next chunk of data.
Input: `m`, a message (string or bytes)
Output: Updated SHAKE256 context

`def read(self, n)`:
Compute the next piece of XOF output.
Input: `n`, an integer (number of bytes to return)
Output: Stream of bytes

Then, there are other hashing-related functions.

`def shake256_hash(m)`:
Perform a simple SHAKE256 hashing with a 64 bytes-long output.

Input: `m`, a message (string or bytes)
Output: Stream of bytes

`def hkdf_extract(salt, ikm)`:
Perform the HKDF extraction phase.
Input:

- `salt`, a stream of bytes
- `ikm`, input keying material, a stream of bytes

Output: `prk`, pseudorandom key, a stream of bytes

`def hkdf_expand(L, prk, info)`:
Perform the HKDF expansion phase.
Input:

- `L`, length, an integer
- `prk`, pseudorandom key, a stream of bytes
- `info`, additional information, a stream of bytes

Output: `okm`, output key material, a stream of bytes

`def I2OSP(val, length)`:
Convert a non-negative integer to an octet string (stream of bytes).
Input:

- `val`, an integer
- `length`, an integer

Output: Stream of bytes

`def OS2IP(octets)`:
Convert an octet string (stream of bytes) to a non-negative integer.
Input: `octets`, a stream of bytes
Output: Integer

`def expand_message_xmd(msg, DST, len_in_bytes, hash_fn)`:
Produce a uniformly random byte string using a cryptographic hash function.
Input:

- `msg`, a stream of bytes
- `DST`, domain separation tag, a stream of bytes
- `len_in_bytes`, an integer
- `hash_fn`, a cryptographic hash function

Output: Stream of bytes

`def expand_message_xof(msg, DST, len_in_bytes, hash_fn)`:
Produce a uniformly random byte string using an extensible-output function (XOF).
Input:

- `msg`, a stream of bytes
- `DST`, domain separation tag, a stream of bytes
- `len_in_bytes`, an integer
- `hash_fn`, an extensible-output function

Output: Stream of bytes

`def hash_to_field(msg, count, dst, modulus, degree, blen, expand_fn, hash_fn)`:
Compute the hash of a message into one or more elements of a field.
Input:

- `msg`, a stream of bytes
- `count`, an integer
- `dst`, domain separation tag, a stream of bytes
- `modulus`, an integer (q)
- `degree`, an integer (1 for $\mathbb{G}_1$, 2 for $\mathbb{G}_2$)
- `blen`, an integer
- `expand_fn`, an expand message function
- `hash_fn`, an extensible-output function

Output: Integer

`def hash_to_scalar(msg, n)`:
Compute the hash of a message into scalars.
Input:

- `msg`, a stream of bytes
- `n`, number of scalars, an integer

Output: Array of integers

`def map_message_to_scalar_as_hash(msg, dst)`:
Compute the hash of a message into a scalar.
Input:

- `msg`, a stream of bytes
- `dst`, domain separation tag, a stream of bytes

Output: Integer

key.py

It handles private key operations.

`class PrivateKey`:
This class defines a private key, which is a random integer between 1 and the group order r. It contains several functions, including the ones to perform comparisons, conversions, and get public keys.

`def __init__(self, value)`:
Input: `value`, an integer
Output: Private key object

opt_swu_g1.py

It implements the optimized simplified SWU map to BLS12-381 $\mathbb{G}_1$.

`def sgn0(x)`:
Return the sign of a field $\mathbb{F}_q$ element.
Input: `x`, a field element
Output: Integer (1 or -1)

`def osswu_help(t)`:
Perform the simplified SWU map by mapping a field $\mathbb{F}_q$ element to the curve .
Input: `t`, a field element
Output: Jacobian point

`def iso11(P)`:
Compute a 11-isogeny map.
Input: `P`, a jacobian point
Output: Jacobian point

`def opt_swu_map(t, t2)`:
Map from field $\mathbb{F}_q$ elements to point in $\mathbb{G}_1$ subgroup.
Input:

- `t`, a field element
- `t2`, a field element

Output: Jacobian point

`def map2curve_osswu(alpha, dst)`:
Map from bytes to point in $\mathbb{G}_1$ subgroup.
Input:

- `alpha`, a stream of bytes
- `dst`, domain separation tag, a stream of bytes

Output: Jacobian point

Then, there are other variables useful to compute operations, such as:

- `sqrt_mxi_1_cubed`, distinguished non-square in $\mathbb{F}_q$ for SWU map
- `EllP_a, EllP_b`, 11-isogenous curve parameters
- `xnum, xden, ynum, yden`, 11-isogeny map coefficients

pairing.py

It contains the Miller loop and the final exponentiation functions, among others, to work with pairings.

`def int_to_bits(i)`:
Convert integers to bits.
Input: `i`, an integer
Output: List of integers

`def double_line_eval(R, P, ec)`:
Evaluate an equation for a line tangent to R at the point P.
Input:

- `R`, an affine point
- `P`, an affine point
- `ec`, an elliptic curve

Output: Field element

`def add_line_eval(R, Q, P, ec)`:
Evaluate an equation for a line between R and Q at the point P.
Input:

- `R`, an affine point
- `Q`, an affine point
- `P`, an affine point
- `ec`, an elliptic curve

Output: Field element

`def miller_loop(T, P, Q, ec)`:
Perform a "double and add" algorithm for the ate pairing.
Input:

- `T`, an integer
- `P`, an affine point
- `Q`, an affine point
- `ec`, an elliptic curve

Output: Field element ($\mathbb{F}_{q^{12}}$)

`def final_exponentiation(element, ec)`:
Map the result of the Miller loop to a unique field $\mathbb{F}_{q^{12}}$ element.
Input:

- `element`, a field element ($\mathbb{F}_{q^{12}}$)
- `ec`, an elliptic curve

Output: Field element ($\mathbb{F}_{q^{12}}$)

`def ate_pairing(P, Q, ec)`:
Perform one ate pairing.
Input:

- `P`, a jacobian point
- `Q`, a jacobian point
- `ec`, an elliptic curve

Output: Field element ($\mathbb{F}_{q^{12}}$)

`def ate_pairing_multi(Ps, Qs, ec)`:
Perform multiple ate pairings at once.
Input:

- `Ps`, a list of jacobian points
- `Qs`, a list of jacobian points
- `ec`, an elliptic curve

Output: Field element ($\mathbb{F}_{q^{12}}$)

parameters.py

It contains the BBS+ signature scheme public parameters:

- `ciphersuite_id (csid) = "BBS_BLS12381G1_XOF:SHAKE-256_SSWU_RO_"`
- `xof_no_of_bytes = 64`
- `octet_scalar_length = 32`
- `octet_point_length = 48`
- `INITSALT = "BBS-SIG-KEYGEN-SALT-"`
- `hash_to_curve_g1_dst = "BBS_BLS12381G1_XOF:SHAKE-256_SSWU_RO_"`
- `hash_to_field_dst = "BBS_BLS12381FQ_XOF:SHAKE-256_SSWU_RO_"`
- `message_generator_seed = csid + "MESSAGE_GENERATOR_SEED"`
- `blind_value_generator_seed = csid + "SIGNATURE_BLINDING_VALUE_GENERATOR_SEED"`
- `sig_domain_generator_seed = csid + "SIGNATURE_DST_GENERATOR_SEED"`

util.py

It provides utility functions needed to use the BBS+ signature scheme.

`def generateSecret()`:
Generate a random secret.
Input: None
Output: Stream of bytes

`def generateNonce()`:
Generate a random nonce.
Input: None
Output: Stream of bytes

`def get_messages(messages, indices)`:
Return selected messages from an array.
Input:

- `messages`, a list of bytes
- `indices`, a list of integers

Output: List of bytes

`def get_messages(messages, indices)`:
Return selected messages from an array.
Input:

- `messages`, a list of bytes
- `indices`, a list of integers

Output: List of bytes

`def get_remaining_indices(length, indices)`:
Return the remaining indices from the length of an array.
Input:

- `length`, an integer
- `indices`, a list of integers

Output: List of integers

`def flatten_json(json_data):`
Flatten a JSON object.
Input: `json_data`, a JSON object (dict)
Output: JSON object (dict)

`def unflatten_json(flat_json):`
Unflatten a flat JSON object.
Input: `flat_json`, a JSON object (dict)
Output: JSON object (dict)

`def attributes_to_indices(json_data, attributes):`
Return the indices of the attributes of a JSON object.
Input:

- `json_data`, a JSON object (dict)
- `attributes`, a list of bytes

Output: List of integers

`def get_selected_attributes(json_data, indices):`
Return the selected attributes from a JSON object.
Input:

- `json_data`, a JSON object (dict)
- `indices`, a list of integers

Output: JSON object (dict)

`def json_to_array(json_data):`
Convert and normalize a JSON object to an array of bytes.
Input: `json_data`, a JSON object (dict)
Output: List of bytes

`def b58_encode(data):`
Encode bytes in a base58 string.
Input: `data`, a stream of bytes
Output: String

`def b58_decode(data):`
Decode a base58 string in bytes.
Input: `data`, a string
Output: Stream of bytes

Network

In the `network` folder there are the modules that allow communication between parties via sockets and simulate storing and retrieving data on a verifiable data registry, such as the *IOTA Tangle*.

connect.py

It contains a function to connect via socket.

`def connect(address, port, callback_applogic)`:
Connect via socket to an address and a port.
Input:

- `address`, a string
- `port`, an integer
- `callback_applogic`, a function

Output: None

netmessage.py

It defines classes and methods for communications over a network.

`class NetMessage`:
This class defines a message to be exchanged over a network. It contains several functions to send and receive data, lines, and messages.

`def __init__(self, channel, message_type, message_buffer_size, fragment_size, socket_buffer_size)`:
Input:

- `channel`, a socket connection
- `message_type`, a string
- `message_buffer_size = 64 KB`, an integer
- `fragment_size = 1 KB`, an integer
- `socket_buffer_size = 8 KB`, an integer

Output: NetMessage object

`def send_data(self, data, debug_log)`:
Send data via socket.
Input:

- `data`, a stream of bytes
- `debug_log`, a boolean

Output: None

`def recv_data(self, data_length, timeout_sec, debug_log)`:
Receive data of a given size via socket.
Input:

- `data_length`, an integer
- `timeout_sec`, a floating-point number
- `debug_log`, a boolean

Output: Stream of bytes

server.py

It contains a class to instantiate and run a server.

`class Server`:
This class defines a server with an address and a port. It contains a function to start a socket for listening and accepting connections.

`def __init__(self, address, port, logger)`:
Input:

- `address`, a string
- `port`, an integer
- `logger`, a logger object

Output: Server object

`def run(self, callback_applogic)`:
Start a socket for listening and accepting connections.
Input: `callback_applogic`, a function
Output: None

tangle.py

It contains functions to simulate read and write operations on a verifiable data registry, such as the IOTA Tangle.

`def read_I_PK_fromTangle()`:
Read the issuer's public key from the Tangle.
Input: None
Output: Tuple

- `W`, a stream of bytes
- `H_s`, a jacobian point
- `H_d`, a jacobian point
- `H`, a list of jacobian points

`def write_I_PK_onTangle(W, H_s, H_d, H)`:
Write the issuer's public key on the Tangle.
Input:

- `W`, a stream of bytes
- `H_s`, a jacobian point
- `H_d`, a jacobian point
- `H`, a list of jacobian points

Output: None

Demo

The demo is composed of three parties: issuer, holder, and verifier.

The elements they produce, such as signatures and proofs, are serialized and deserialized following the BBS+ signature scheme IETF specification [59]. The elliptic curve points, such as public keys, generators, and commitments, are serialized and deserialized following the pairing-friendly curves IETF specification [52].

Moreover, to perform multi-messages BBS+ operations over a credential, the parties must canonicalize the JSON credential and its attributes must become an array of messages. A proposal at the W3C Credentials Community Group [75] specifies how to flatten a JSON by using the JSON Pointer Normalization [76].

In this way, the following JSON object:

```
{
  "name": "John Doe",
  "university": {
    "name": "Politecnico di Torino",
    "faculty": "Computer Engineering",
    "role": "Student"
  },
  "courses": ["Cybersecurity", "Cryptography"]
}
```

Listing B.1. Example of a credential to be flattened

Becomes the following array:

```
[
    "/name:John␣Doe",
    "/university/name:Politecnico␣di␣Torino",
    "/university/faculty:Computer␣Engineering",
    "/university/role:Student",
    "/courses/0:Cybersecurity",
    "/courses/1:Cryptography"
]
```

Listing B.2. Example of a flattened credential

Customization

The data used during the demo can be customized by modifying the values in the `data.py` file. It is possible to change the credential, its attributes, and which attributes to disclose.

Bibliography

[1] P. Sneier, "On the Internet, nobody knows you're a dog", The New Yorker, vol. 69, July 1993

[2] The Laws of Identity, https://www.identityblog.com/stories/2005/05/13/TheLawsOfIdentity.pdf

[3] A. Preukschat and D. Reed, "Self-Sovereign Identity: Decentralized digital identity and verifiable credentials", Manning Publications Co., 2021, ISBN: 9781617296598

[4] Introduction to Trust Over IP (v2.0), https://trustoverip.org/wp-content/uploads/Introduction-to-ToIP-V2.0-2021-11-17.pdf

[5] World Economic Forum - A Blueprint for Digital Identity, https://www3.weforum.org/docs/WEF_A_Blueprint_for_Digital_Identity.pdf

[6] "Commission implementing regulation (EU) 2015/1502 of 8 September 2015 on setting out minimum technical specifications and procedures for assurance levels for electronic identification means pursuant to Article 8(3) of Regulation (EU) No 910/2014 of the European Parliament and of the Council on electronic identification and trust services for electronic transactions in the internal market", Official Journal of the European Union, vol. L 235/7, September 2015, pp. 4–14

[7] A Field Guide to Internet Trust, https://identitywoman.net/wp-content/uploads/TrustModelFieldGuideFinal-1.pdf

[8] M. Sporny, D. Longley, M. Sabadello, D. Reed, O. Steele, and C. Allen, "Decentralized Identifiers (DIDs) v1.0", W3C Recommendation, July 2022

[9] M. Sporny, D. Longley, and D. Chadwick, "Verifiable Credentials Data Model v1.1", W3C Recommendation, March 2022

[10] Documents published at W3C, https://www.w3.org/standards/types

[11] T. Berners-Lee, R. T. Fielding, and L. M. Masinter, "Uniform Resource Identifier (URI): Generic Syntax", RFC-3986, January 2005, DOI 10.17487/RFC3986

[12] T. Bray, "The JavaScript Object Notation (JSON) Data Interchange Format", RFC-8259, December 2017, DOI 10.17487/RFC8259

[13] M. Sporny, D. Longley, G. Kellogg, M. Lanthaler, P.-A. Champin, and N. Lidström, "JSON-LD v1.1", W3C Recommendation, July 2020

[14] D. Ferraiolo, J. Cugini, and R. Kuhn, "Role-Based Access Control (RBAC): Features and Motivations", 11th Annual Computer Security Applications Conference, New Orleans, Louisiana, United States, December 11-15, 1995, pp. 241–248

[15] C. T. Hu, D. Ferraiolo, D. Kuhn, A. Schnitzer, K. Sandlin, R. Miller, and K. Scarfone, "Guide to Attribute Based Access Control (ABAC) Definition and Considerations", Special Publication (NIST SP), National Institute of Standards and Technology, Gaithersburg, MD, August 2019, DOI 10.6028/NIST.SP.800-162

[16] E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.3", RFC-8446, August 2018, DOI 10.17487/RFC8446

[17] B. Zundel, I. Hernandez, C. Allen, Z. Larson, and K. Dow, "Engineering Privacy for Verified Credentials", W3C Draft Community Group Report, April 2022, https://w3c-ccg.github.io/data-minimization

[18] Selective Disclosure - Handling Personal Information, https://learn.mattr.global/docs/concepts/selective-disclosure (Accessed on 2022-07-14)

[19] ISO, "Evaluation criteria for IT security - Part 1: Introduction and general model", ISO/IEC 15408-1:2009, 2009

[20] A terminology for talking about privacy by data minimization: Anonymity, Unlinkability, Undetectability, Unobservability, Pseudonymity, and Identity Management, https://dud.inf.tu-dresden.de/literatur/Anon_Terminology_v0.34.pdf

[21] P. A. Grassi, M. E. Garcia, and J. L. Fenton, "Digital Identity Guidelines", Special Publication (NIST SP), National Institute of Standards and Technology, Gaithersburg, MD, June 2017, DOI 10.6028/NIST.SP.800-63-3

[22] S. Goldwasser, S. Micali, and C. Rackoff, "The Knowledge Complexity of Interactive Proof-Systems", Proceedings of the Seventeenth Annual ACM Symposium on Theory of Computing, New York, NY, USA, 1985, pp. 291–304, DOI 10.1145/22145.22178

[23] M. Bellare and O. Goldreich, "On Defining Proofs of Knowledge", Advances in Cryptology — CRYPTO '92 (E. F. Brickell, ed.), Berlin, Heidelberg, 1993, pp. 390–420, DOI 10.1007/3-540-48071-4_28

[24] On Interactive Proofs and Zero-Knowledge: A Primer, https://medium.com/magicofc/interactive-proofs-and-zero-knowledge-b32f6c8d66c3

[25] P versus NP problem, https://www.britannica.com/science/P-versus-NP-problem

[26] J.-J. Quisquater, M. Quisquater, M. Quisquater, M. Quisquater, L. C. Guillou, M. A. Guillou, G. Guillou, A. Guillou, G. Guillou, S. Guillou, and T. A. Berson, "How to explain zero-knowledge protocols to your children", Advances in Cryptology — CRYPTO '89, 9th Annual International Cryptology Conference, Santa Barbara, California, USA, August 20-24, 1989, Proceedings, 1990, pp. 628–631, DOI 10.1007/0-387-34805-0_60

[27] O. Goldreich, S. Micali, and A. Wigderson, "Proofs That Yield Nothing but Their Validity or All Languages in NP Have Zero-Knowledge Proof Systems", J. ACM, vol. 38, July 1991, pp. 690–728, DOI 10.1145/116825.116852

[28] O. Goldreich, "Foundations of Cryptography", vol. 1, Cambridge University Press, 2001, ISBN: 0-521-79172-3

[29] T. P. Pedersen, "Non-Interactive and Information-Theoretic Secure Verifiable Secret Sharing", Advances in Cryptology — CRYPTO '91 (J. Feigenbaum, ed.), Berlin, Heidelberg, 1992, pp. 129–140, DOI 10.1007/3-540-46766-1_9

[30] C. P. Schnorr, "Efficient Identification and Signatures for Smart Cards", Advances in Cryptology — CRYPTO '89 Proceedings (G. Brassard, ed.), New York, NY, 1990, pp. 239–252, DOI 10.1007/0-387-34805-0_22

[31] A. Fiat and A. Shamir, "How To Prove Yourself: Practical Solutions to Identification and Signature Problems", Advances in Cryptology — CRYPTO '86 (A. M. Odlyzko, ed.), Berlin, Heidelberg, 1987, pp. 186–194, DOI 10.1007/3-540-47721-7_12

[32] D. Pointcheval and J. Stern, "Security Proofs for Signature Schemes", Advances in Cryptology — EUROCRYPT '96 (U. Maurer, ed.), Berlin, Heidelberg, 1996, pp. 387–398, DOI 10.1007/3-540-68339-9_33

[33] M. Bellare and P. Rogaway, "Random Oracles Are Practical: A Paradigm for Designing Efficient Protocols", Proceedings of the 1st ACM Conference on Computer and Communications Security, New York, NY, USA, 1993, pp. 62–73, DOI 10.1145/168588.168596

[34] S. Goldwasser and Y. Kalai, "On the (In)security of the Fiat-Shamir paradigm", 44th Annual IEEE Symposium on Foundations of Computer Science, 2003. Proceedings., 2003, pp. 102–113, DOI 10.1109/SFCS.2003.1238185

[35] M. Blum, P. Feldman, and S. Micali, "Non-Interactive Zero-Knowledge and Its Applications", Proceedings of the Twentieth Annual ACM Symposium on Theory of Computing, New York, NY, USA, 1988, pp. 103–112, DOI 10.1145/62212.62222

[36] S. Bowe, A. Gabizon, and I. Miers, "Scalable Multi-party Computation for zk-SNARK Parameters in the Random Beacon Model", Cryptology ePrint Archive, Paper 2017/1050, 2017, https://eprint.iacr.org/2017/1050

[37] D. Wong, "Real-World Cryptography", Manning Publications Co., 2021, ISBN: 9781617296710

[38] N. Bitansky, R. Canetti, A. Chiesa, and E. Tromer, "From Extractable Collision Resistance to Succinct Non-Interactive Arguments of Knowledge, and Back Again", Proceedings of the 3rd Innovations in Theoretical Computer Science Conference, New York, NY, USA, 2012, pp. 326–349, DOI 10.1145/2090236.2090263

[39] M.-H. Nguyen, S. J. Ong, and S. Vadhan, "Statistical Zero-Knowledge Arguments for NP from Any One-Way Function", 2006 47th Annual IEEE Symposium on Foundations of Computer Science (FOCS '06), October 2006, pp. 3–14, DOI 10.1109/FOCS.2006.71

[40] E. Ben-Sasson, I. Bentov, Y. Horesh, and M. Riabzev, "Scalable, transparent, and post-quantum secure computational integrity", Cryptology ePrint Archive, Paper 2018/046, 2018, https://eprint.iacr.org/2018/046

[41] B. Bünz, J. Bootle, D. Boneh, A. Poelstra, P. Wuille, and G. Maxwell, "Bulletproofs: Short Proofs for Confidential Transactions and More", 2018 IEEE Symposium on Security and Privacy (SP), May 2018, pp. 315–334, DOI 10.1109/SP.2018.00020

[42] C. Paar and J. Pelzl, "Understanding Cryptography: A Textbook for Students and Practitioners", Springer Publishing Company, Incorporated, 1st ed., 2009, ISBN: 3642041000

[43] J. Camenisch, M. Drijvers, and A. Lehmann, "Anonymous Attestation Using the Strong Diffie Hellman Assumption Revisited", Trust and Trustworthy Computing (M. Franz and P. Papadimitratos, eds.), Cham, 2016, pp. 1–20, DOI 10.1007/978-3-319-45572-3_1

[44] S. D. Galbraith, K. G. Paterson, and N. P. Smart, "Pairings for cryptographers", Discrete Applied Mathematics, vol. 156, no. 16, 2008, pp. 3113–3121, DOI 10.1016/j.dam.2007.12.010 Applications of Algebra to Cryptography

[45] D. Boneh and X. Boyen, "Short signatures without random oracles", Advances in Cryptology - EUROCRYPT 2004 (C. Cachin and J. L. Camenisch, eds.), Berlin, Heidelberg, 2004, pp. 56–73, DOI 10.1007/978-3-540-24676-3_4

[46] D. Boneh and X. Boyen, "Short Signatures Without Random Oracles and the SDH Assumption in Bilinear Groups", Journal of Cryptology, vol. 21, 2008, pp. 149–177, DOI 10.1007/s00145-007-9005-7

[47] D. Freeman, M. Scott, and E. Teske, "A Taxonomy of Pairing-Friendly Elliptic Curves", Journal of Cryptology, vol. 23, 2010, pp. 224–280, DOI 10.1007/s00145-009-9048-z

[48] A. Menezes, T. Okamoto, and S. Vanstone, "Reducing elliptic curve logarithms to logarithms in a finite field", IEEE Transactions on Information Theory, vol. 39, no. 5, 1993, pp. 1639–1646, DOI 10.1109/18.259647

[49] P. S. L. M. Barreto, B. Lynn, and M. Scott, "Constructing elliptic curves with prescribed embedding degrees", Proceedings of the 3rd International Conference on Security in Communication Networks, Berlin, Heidelberg, 2002, pp. 257–267

[50] BLS12-381: New zk-SNARK Elliptic Curve Construction, https://electriccoin.co/blog/new-snark-curve

[51] BLS12-381 for the rest of us, https://hackmd.io/@benjaminion/bls12-381 (Accessed on 2022-07-29)

[52] Y. Sakemi, T. Kobayashi, T. Saito, and R. S. Wahby, "Pairing-Friendly Curves", IETF draft-irtf-cfrg-pairing-friendly-curves-10, July 2021, Work in Progress

[53] "Zcash Overwinter Consensus and Sapling Cryptography Review", https://research.nccgroup.com/wp-content/uploads/2020/07/NCC_Group_Zcash2018_Public_Report_2019-01-30_v1.3.pdf

[54] D. Chaum and E. van Heyst, "Group signatures", Advances in Cryptology – EUROCRYPT '91 (D. W. Davies, ed.), Berlin, Heidelberg, 1991, pp. 257–265, DOI 10.1007/3-540-46416-6_22

[55] D. Boneh, X. Boyen, and H. Shacham, "Short Group Signatures", Advances in Cryptology – CRYPTO 2004 (M. Franklin, ed.), Berlin, Heidelberg, 2004, pp. 41–55, DOI 10.1007/978-3-540-28628-8_3

[56] J. Camenisch and A. Lysyanskaya, "A Signature Scheme with Efficient Protocols", Security in Communication Networks (S. Cimato, G. Persiano, and C. Galdi, eds.), Berlin, Heidelberg, 2003, pp. 268–289, DOI 10.1007/3-540-36413-7_20

[57] R. L. Rivest, A. Shamir, and L. Adleman, "A method for obtaining digital signatures and public-key cryptosystems", Commun. ACM, vol. 21, feb 1978, pp. 120–126, DOI 10.1145/359340.359342

[58] M. H. Au, W. Susilo, and Y. Mu, "Constant-Size Dynamic k-TAA", Security and Cryptography for Networks (R. De Prisco and M. Yung, eds.), Berlin, Heidelberg, 2006, pp. 111–125, DOI 10.1007/11832072_8

[59] T. Looker, V. Kalos, A. Whitehead, and M. Lodder, "The BBS Signature Scheme", IETF draft-irtf-cfrg-bbs-signatures-00, October 2022, Work in Progress

[60] Blind Signatures extension of the BBS Signature Scheme, https://identity.foundation/bbs-signature/draft-blind-bbs-signatures.html

[61] S. programs for functions on curves, https://crypto.stanford.edu/miller/miller.pdf

[62] M. Dworkin, "SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions", Federal Inf. Process. Stds. (NIST FIPS), National Institute of Standards and Technology, Gaithersburg, MD, August 2015, DOI https://doi.org/10.6028/NIST.FIPS.202

[63] E. Brier, J.-S. Coron, T. Icart, D. Madore, H. Randriam, and M. Tibouchi, "Efficient indifferentiable hashing into ordinary elliptic curves", Advances in Cryptology – CRYPTO 2010 (T. Rabin, ed.), Berlin, Heidelberg, 2010, pp. 237–254, DOI 10.1007/978-3-642-14623-7_13

[64] R. S. Wahby and D. Boneh, "Fast and simple constant-time hashing to the BLS12-381 elliptic curve", Cryptology ePrint Archive, Paper 2019/403, 2019, DOI 10.13154/tches.v2019.i4.154-179, https://eprint.iacr.org/2019/403

[65] D. Chaum, "Showing credentials without identification transferring signatures between unconditionally unlinkable pseudonyms", Advances in Cryptology — AUSCRYPT '90 (J. Seberry and J. Pieprzyk, eds.), Berlin, Heidelberg, 1990, pp. 245–264, DOI 10.1007/BFb0030366

[66] M. Chase, E. Ghosh, S. Setty, and D. Buchner, "Zero-knowledge credentials with deferred revocation checks", July 2020, https://github.com/decentralized-identity/snark-credentials/blob/master/whitepaper.pdf

[67] How does a verifier know the credential is yours?, https://www.evernym.com/blog/how-does-a-verifier-know-the-credential-is-yours

[68] J. Camenisch, M. Kohlweiss, and C. Soriente, "Solving revocation with efficient update of anonymous credentials", Security and Cryptography for Networks (J. A. Garay and R. De Prisco, eds.), Berlin, Heidelberg, 2010, pp. 454–471, DOI 10.1007/978-3-642-15317-4_28

[69] D. Cooper, S. Santesson, S. Farrell, S. Boeyen, R. Housley, and W. Polk, "Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile", RFC-5280, May 2008, DOI 10.17487/RFC5280

[70] J. Camenisch and A. Lysyanskaya, "Dynamic accumulators and application to efficient revocation of anonymous credentials", Advances in Cryptology — CRYPTO 2002 (M. Yung, ed.), Berlin, Heidelberg, 2002, pp. 61–76, DOI 10.1007/3-540-45708-9_5

[71] E. Brickell, J. Camenisch, and L. Chen, "Direct anonymous attestation", Proceedings of the 11th ACM Conference on Computer and Communications Security, New York, NY, USA, 2004, pp. 132–145, DOI 10.1145/1030083.1030103

[72] ISO, "Trusted Platform Module Library - Part 1: Architecture", ISO/IEC 11889-1:2015, 2015

[73] M. Jones, J. Bradley, and N. Sakimura, "JSON Web Token (JWT)", RFC-7519, May 2015, DOI 10.17487/RFC7519

[74] D. Hardt, "The OAuth 2.0 Authorization Framework", RFC-6749, October 2012, DOI 10.17487/RFC6749

[75] BBS+ Signature Suite for LDP using JSON Pointer Normalization, https://github.com/trinsic-id/json-bbs-signatures/blob/main/ldp-bbs-jpn.md

[76] P. C. Bryan, K. Zyp, and M. Nottingham, "JavaScript Object Notation (JSON) Pointer", RFC-6901, April 2013, DOI 10.17487/RFC6901