Bern University
of Applied Sciences

Bachelor's Thesis

Unlinkability of Verifiable Credentials in a practical approach: Report

Course of study	Bachelor of Science in Computer Science
Author	Joël Gabriel Robles Gasser
Advisor	Prof. Dr. Annett Laube, Prof. Dr. Reto Koenig
Expert	Dr. Andreas Spichiger

Version 1.0 of June 13, 2024

- Engineering and Computer Science
- Computer Science

Abstract

This thesis analyzes the unlinkability of verifiable credentials (VCs) in conjunction with the BBS Signature Scheme (BBS) in a real-world implementation. Initially, the key features and concepts of verifiable credentials are analyzed. From those key features, a minimal usable subset is chosen and explored for its functionality and used to build examples. Algorithms for transforming VCs so that they can be signed by BBS are described. Furthermore, algorithms for creating and verifying selective disclosure verifiable presentations (VPs) are provided. Security concerns and the unlinkability are then analyzed, and solutions are proposed if needed. The messaging between the holder and verifier is handled by OpenID Connect for verifiable presentations. This technology is also analyzed for security concerns. The research concludes that the implemented methods achieve the desired unlinkability, with potential future enhancements identified to further improve the unlinkability of verifiable credentials in digital identity systems.

Contents

1 Introduction

In today's digital world, digital identities are mostly managed by centralized entities, like big corporations or governments. This may be easier for individuals as they don't need to manage their data. As the individuals don't have the control how their data is handled, this approach leads to problems with privacy and security.

Self-sovereign Identity (SSI)[10] is a concept where individuals can control their digital identity and what data is shared with whom. The idea is to allow those individuals to store their data on their own devices, without centralized entities playing middleman. With the data on the device of the individual, they may selectively disclose information, and thus preserve privacy. Another important aspect of SSI is the unlinkability between presentations, meaning that there is no link back to an individual when different data is presented to multiple verifying parties. Today's centralized entities always know which page an individual visited or what they bought, as they can link all this information back to one individual. A good example for this are targeted ads, which are based on previously revealed information.

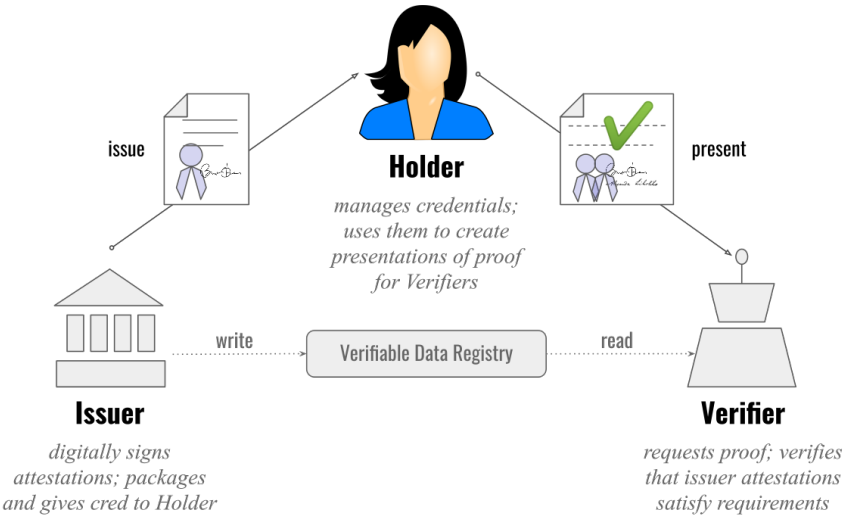

Figure 1.1: The trust triangle[1]

[1] https://upload.wikimedia.org/wikipedia/commons/5/51/VC_triangle_of_Trust.svg

Besides the individual that owns the data, there are more involved entities in SSI, seen in the trust triangle in figure 1.1. The issuer is the entity issuing credentials. A comparison of a credential might be an identity card, which is issued by a state. The third entity is the verifier. This party requests data from a holder, then verifies the response. An example would be the request and the subsequent presentation of a government issued ID when the identity of a holder is challenged. The verifying party is able to trust the presented data, because it is backed by the issuer. For this to work, the issuer must play fair (honest but curious) and does not issue malicious credentials. Only then can the verifier establish trust, hence the name the *trust* triangle.

One of the key technologies used in this thesis enabling SSI, is the BBS Signature Scheme (BBS)[6]. This cryptographic signature scheme provides mechanisms for both important SSI features, selective-disclosure and unlinkability. This enables the creation of secure and private verifiable credentials and the digital representation of physical credentials.

It is important to understand the difference between physical and digital credentials. First we need to establish that physical credentials can only be used in the physical world, where digital credentials can be used both in the physical and digital world and provide more security and privacy. It is possible that a verifier would accept a physical credential, where all the information that is not needed is taped over, as long as they can verify the validity of the credential (even though one would get weird looks). With that, selective disclosure is achieved. Also, for example when signing a contract, no identifying data is stored by the verifier, as they only check if the individual really is who they say they are. And with that we have the unlinkability. But in the digital world this is different, everything is logged and saved. An individual cannot use their physical credential for a digital interaction. That is why physical credentials need to be translated to digital credentials. Selectively disclosing information gets easier for an individual, when instead of having to change the tape over the hidden information, they can just select what they want to share. But the translation also brings new problems with it. All physical credentials from the same type, like a Swiss government ID, have general security mechanisms, so using those mechanisms for individual verification does not reveal any information. In the digital world, cryptographic signatures are used as a sign of validity, integrity and authenticity of the data. Those signatures are individual per VC. This means that each credential has its own identifier: the signature that is used to protect it.

BBS has a solution for this problem. Besides creating the signatures that protect the data and letting the individuals selectively disclose that data, there is a mechanism that allows to randomize the signature, without invalidating it. This allows the verifier to confirm an unlinkable presentation.

The credentials also need to be persisted in some type of construct. In the physical world, it is written on a neat little credit-card sized piece of plastic. That card also has

all the security on/in it. In the digital world persistence is usually done with files. These files contain data in a structured way, for example JSON. In our thesis we are using the verifiable credentials data structure (VC)[8], a JSON extension. For a holder of a VC it is also easy to store. Instead of adding an ID card to a wallet, a holder adds the persisted VC to a digital wallet on their device.

In this thesis we want to investigate the use of the BBS Signature Scheme in a real world scenario. We assume a working BBS Signature Scheme within the security setting, comprising unlinkability and data integrity.

The VCs need to be sent digitally from issuer to holder and presented from a holder to a verifier. For this thesis we will only look at the messaging between the holder and the verifier and exclude VC creation and transportation to the holder. Furthermore we will use an extension of VCs called verifiable presentations (VP)[8]. To send a VP between a holder and a verifier, we use OpenID Connect for Verifiable Presentations (OIDC4VP)[9]. We assume secure channels are used for the OIDC4VP communication.

Now that we know the different technologies and what they do, we can define the goal of this thesis. We want to analyze if the unlinkability provided by BBS breaks when using it in conjunction with VCs and OIDC4VP in a real world scenario.

2 The BBS Signature Scheme

The BBS Signature Scheme[6] is a multi-message signature scheme, supporting selective-disclosure and the proof of the knowledge of a signature which in turn creates unlinkability. These features makes this signature scheme an integral part of SSI.

But what do these features really mean?

Normal cryptographic signatures are only able to sign one message at a time, which is viable for digital credentials which contain multiple messages. To sign those credentials, a multi-message signature is needed, which is provided by BBS. The resulting signature provides data integrity for the messages. Important to note is also the fixed size of the signature. No matter the amount and sizes of the messages, the signature length stays the same, thus removing any links between the signature and the signed messages.

Now we have a signed set of messages and we would like to present them. All the messages can be presented and verified together using the original signature. The holder may want to only present a sub set of the original messages, selectively disclosing them. To verify a BBS signature, the verifier would need all the data that was signed, so revealing only a part of the data would not work. To be able to that, the holder must generate a new BBS signature, called a BBS proof, which is generated using the original signature. This new signature is valid for the sub set of the revealed messages with the added protection of proving to the verifier the knowledge of the original signature. This last part is important, as a verifier only trusts the original issuer of a credential and therefore only trusts the original signature.

The holder has generated a BBS proof and presented it to a verifier. Now the same holder would like to present the same data to the same verifier again. The easy way would be to use the same proof from the previous presentation. This would create a link to the previous presentation, as the proof value would be an identifier. To prevent the linking of presentations, a new proof must be created for each presentation, as these proofs are random for each generation. With that we have unlinkability between presentations.

3 Use Case

A university issues their diplomas as verifiable credentials to their students. Those contain multiple attributes, like degree, name, university and so on. A student would now like to join an online education platform. Only students with a degree in computer science are allowed to to join the platform.

The student can now generate a selectively disclosed VP, which only contains the degree prooving that they studied computer science. All the other attributes stay hidden. The student can now present that VP and is able to join the online course. As each VP is unlinkable because of the BBS proof, the activity of the student stays secret and thus ensures their privacy.

4 Verifiable Credentials and Verifiable Presentations

In this chapter, verifiable credentials (VC) and verifiable presentations (VP) are analyzed for data leakage and linkability. Then we will investigate how we need to manipulate these credentials and presentations, so that they can be signed by the BBS Signature Scheme.

4.1 What are VCs and VPs?

Verifiable credentials[8] are JSON-LD data models, designed to represent different types of digital credentials. The idea is to be able to translate physical credentials, like an ID or a driver's license, into the digital world. These VCs are generated and signed cryptographically by an issuer, like a government. In this thesis we will use the BBS Signature Scheme to sign the VC. The content of VC is meta-data, like the life-time or an ID, and an object containing the data that wants to be stored, most of the time, data that belongs to a specific holder. In the case of a government-issued ID, these can be attributes like birthdate, first and last name and so on. After creation, the issuer sends the VC to the holder.

Now the holder would like to present this VC to a verifier. Using the VC, the holder can generate a verifiable presentation[8], which is very close to a VC in structure. It also contains meta-data that follows the same rules as the meta-data in a VC. The only difference between a VP and a VC is that a VP contains a whole VC instead of the holder data. In the case of a VP, we also want to use a cryptographic signature to protect the content against tampering. Here we also use the BBS Signature Scheme.

In the next sections we will look at the VC concepts that are used in this thesis. Please note that there are more concepts, which are out of scope.

4.2 Used VC concepts

In this section we will define all the VC concepts that will be used in this thesis.

4.2.1 @context

The **@context** attribute is used to map human-friendly IDs like *type* to an URL. These URLs then help the system understand what the content of the VC is. The value of the **@context** attribute is an ordered list, where the first entry must be *https://www.w3.org/ns/credentials/v2*.
We will also use the context *https://w3id.org/security/data-integrity/v2* to signify that the content of the VC is protected. Lastly we use a custom context, which describes the content in the credentialSubject.

Our custom context looks like this:

```
{
  "@context":{
    "first_name": "https://schema.org/givenName",
    "last_name": "https://schema.org/familyName",
    "birth_date": "https://schema.org/birthDate"
  }
}
```

Listing 4.1: Example custom context

The complete **@context** which we will use in our example, where the different contexts are represented by URLs, looks like this:

```
{
  "@context": [
    "https://www.w3.org/ns/credentials/v2",
    "https://w3id.org/security/data-integrity/v2",
    "https://raw.githubusercontent.com/robles..."
  ],
}
```

Listing 4.2: Example context

4.2.2 credentialSubject

The **credentialSubject** contains a set of objects, which each contain one or more statements about the subject of the credential.
In our example we only have one object with three statements about the subject, namely *first_name*, *last_name* and *birth_date*.
Such a credentialSubject may look like this:

```
{
  "credentialSubject":{
    "first_name": "John",
    "last_name": "Doe",
    "birth_date": "1.1.1970"
  }
}
```

Listing 4.3: Example credentialSubject

4.2.3 ID

We are also able to add an **ID** to the VC. This ID can be used to identify a VC or to use it for revocation purposes.
There are two places where an **ID** can be added:

1. In the root of the VC, such that the whole VC can be identified
2. Inside an object of a credentialSubject (see chapter 4.2.2) to identify a specific subject

These IDs must be *URLs* and be either a UUID, a DID or an HTTP URL.

4.2.4 type

The **type** defines what the context of a VC is. It is an array containing either URLs to the description of the VC or can be attributes that must be mapped through **@context**. If we use the standard mapped attributes, the type set must either include *VerifiableCredential* or *VerifiablePresentation*. If we want, we can also add a more specific type, in which we define more information about the VC.
In this thesis we won't use any custom types.

4.2.5 proof

The **proof** inside a VC guaranties its integrity. But it can also do much more. In this thesis the proof will either contain a BBS Signature, which signs the VC and with that guaranties its integrity, or it contains a BBS proof, which enables selective disclosure and unlinkability between holder and verifier.
The proof object contains following key-value pairs:

- **type**: The type of the proof. In this thesis we use the type *DataIntegrityProof*, as it defines that the proof is there to protect the integrity of the data
- **cryptosuite**: Which cryptosuite was used. In this thesis we use *bbs-2023*, as we use the BBS Signature Scheme

- **created**: A timestamp when the proof was created
- **verificationMethod**: Which verification method is used. This value must be a string that maps to a URL. In this thesis we will use a URL to a public key
- **proofPurpose**: The purpose of the proof. There are two different valid values for this attribute: *assertionMethod* and *authentication*. In this thesis we will use *assertionMethod*
- **proofValue**: The proof value. In this thesis this is either a BBS signature or proof

4.3 VCs and BBS

This section contains all the different algorithms to sign a VC, derive a new VC which is used for presentation, create a BBS proof based on the derived VC, wrap the VC in a VP, present it and finally to verify the selectively disclosed VP.

In chapter 4.2 we defined all the different VC/VP concepts that will be used in this thesis. With that information, we can now build an example VC, that we will use to demonstrate the results of the algorithms.
The VC that we will use looks like this:

```
{
  "@context": [
    "https://www.w3.org/ns/credentials/v2",
    "https://w3id.org/security/data-integrity/v2",
    "https://raw.githubusercontent.com/robles..."
  ],
  "type":  ["VerifiableCredential"],
  "credentialSubject":{
    "first_name": "John",
    "last_name": "Doe",
    "birth_date": "1.1.1970"
  }
}
```

Listing 4.4: Example VC

4.3.1 Sign a VC

The first step in the process is to sign a VC. These algorithms are normally called by an issuer, which generated the VC. As the next step we define some variables, that will be used as the inputs to our algorithms:
vc: The VC document, see listing 4.4
hmac_key: A 32-byte random string, which is used to initialize a HMAC

verification_method: The URL to the verification method
mandatory_attributes: A set of attributes which are mandatory for the holder to disclose to the verifier. For this example we will set the *first_name* as mandatory. This looks like this: ["credentialSubject/first_name"]

We now define the main algorithm, which calls the sub-algorithms and handles the data between them:

1. Set **proof_config** and **canonical_proof_config** to the respective entry in the result of the algorithm described in chapter 4.3.1.2, with the inputs **vc.@context** and **verification_method**
2. Set **transformed_document** to the result of the algorithm described in chapter 4.3.1.3, with the inputs **vc**, **mandatory_attributes** and **hmac_key**
3. Set **hash_data** to the result of the algorithm described in chapter 4.3.1.4, with the inputs **canonical_proof_config** and **transformed_document**
4. Set **base_proof** to the result of the algorithm described in chapter 4.3.1.5, with the inputs **hash_data** and **mandatory_attributes**
5. Set **signed_vc** to the result of the algorithm described in chapter 4.3.1.6, with the inputs **vc**, **proof_config** and **base_proof**

After these steps have the signed VC in **signed_vc**. This algorithm is normally used by issuers, which generate the VC and sign it. After that, they send the secured VCs to their respective holder, which in turn can use them to create VPs. Such a signed VC might look like this:

```
{
  "@context": [
    "https://www.w3.org/ns/credentials/v2",
    "https://w3id.org/security/data-integrity/v2",
    "https://raw.githubusercontent.com/robles..."
  ],
  "type": [
    "VerifiableCredential"
  ],
  "credentialSubject": {
    "first_name": "John",
    "last_name": "Doe",
    "birth_date": "1.1.1970"
  },
  "proof": {
    "type": "DataIntegrityProof",
    "cryptosuite": "bbs-2023",
    "created": "2024-05-20T11:40:13.934Z",
    "verificationMethod": "did:key:zUC7D...",
    "proofPurpose": "assertionMethod",
    "proofValue": "u2V0ChdhAWFC..."
  }
}
```

Listing 4.5: Signed VC

In the coming sections we will define all the sub-algorithms used to generate **signed_vc**.

4.3.1.1 Shuffled label map

This algorithm shuffles the content of a canonical ID map, such that nobody without the hmac key knows the original position of the attributes in the VC.

The required inputs of this algorithm are a random 32-bit long bit-string named **hmac_key** which will be used to initialize a *HMAC* and **canonical_id_map** which contains the map.

The output is a function called **label_map_factory_function**, which shuffles canonical id maps.

To shuffle the map, we follow these steps:

1. Create a function with the name **label_map_factory_function**, which has one required input named **canonical_id_map**. Add these steps to the function:

a) Initialize **bnode_id_map** to a new map.

b) For each entry in the **canonical_id_map**, we split them up into *key* and *value*

   i. Set **hmac_result** to result of a HMAC-digest operation, which was initialized with the **hmac_key**, with the input of *value*

   ii. Add a new map entry into **bnode_id_map**, where the key is the *key* and the value is **hmac_result** encoded in base64-url encoding defined in chapter 5 and 3.2 of RFC 4648[3]

c) Initialize **hmac_ids** to the sorted values of **bnode_id_map**

d) For each entry in **bnode_id_map**, split up into *key* and *value*

   i. Concatenate **b** with index of *value* in **hmac_ids**. Set the result of this concatenation as the value of the current entry in **bnode_id_map**

e) Return **bnode_id_map**

2. Return **label_map_factory_function**

4.3.1.2 Create the proof config

This sub algorithm creates the skeleton of the proof config object, which will be appended to the VC. This object contains the information about the proof to be generated, like what cryptosuite was used, what type of proof it is etc.

The required inputs of this algorithm are the URL of the verification information **verification_method** and the context of the VC **@context**.

The output of this algorithm is the proof config **proof_config** and the canonicalized proof config **canonical_proof_config**.

To generate the proof config, we follow these steps:

1. Create an empty **proof_config** object
2. Set **proof_config.type** to *DataIntegrityProof*
3. Set **proof_config.cryptosuite** to *bbs-2023*
4. Set **proof_config.created** to the current time
5. Set **proof_config.verificationMethod** to the URL of the verification method **verification_method**
6. Set **proof_config.@context** to **@context**
7. Set **canonical_proof_config** to the canonical representation of the **proof_config** using the *Universal RDF Dataset Canonicalization Algorithm*[4]

8. Return an object containing **proof_config** and **canonical_proof_config**.

If we follow the instructions, we get a JSON object like this:

```
{
  type: "DataIntegrityProof",
  cryptosuite: "bbs-2023",
  created: "2024-05-20T14:25:12.540Z",
  verificationMethod: "did:key:zUC7De...",
  proofPurpose: "assertionMethod",
  "@context": [
    "https://www.w3.org/ns/credentials/v2",
    "https://w3id.org/security/data-integrity/v2",
    "https://raw.githubusercontent.com/robles...",
  ],
}
```

Listing 4.6: Example Proof Config

This object is then canonicalized into this:

```
_:c14n0 <http://purl.org/dc/terms/created> \"2024-05-20T14
    ↪ :25:12.540Z\"^^<http://www.w3.org/2001/XMLSchema#
    ↪ dateTime> .
_:c14n0 <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> <
    ↪ https://w3id.org/security#DataIntegrityProof> .
_:c14n0 <https://w3id.org/security#cryptosuite> \"bbs
    ↪ -2023\"^^<https://w3id.org/security#cryptosuiteString>
    ↪ .
_:c14n0 <https://w3id.org/security#proofPurpose> <https://
    ↪ w3id.org/security#assertionMethod> .
_:c14n0 <https://w3id.org/security#verificationMethod> <did:
    ↪ key:zUC7De...> .
```

Listing 4.7: Example Proof Config canonicalized

Note that **@context** is missing from the statements. This is because it is used by the canonicalization algorithm to determine the type of each attribute in the object.

After those steps we now have the skeleton of the proof config object and its canonicalized version.

4.3.1.3 Transform the VC

This algorithm transforms a VC into statements, sorted into mandatory and non-mandatory ones. These statements then can be used as an input into the BBS signature algorithm.

The required inputs are a VC **vc**, a 32-byte long **hmac_key** to initialize an HMAC and a set of mandatory pointers **mandatory_attributes**.

The output of this algorithm is an object containing the mandatory and the non-mandatory attributes.

To generate the statements, we follow these steps:

1. Define **label_map_factory_function** as the function which is returned from **create_shuffled_id_label_map_function** defined in chapter 4.3.1.1 with the input **hmac_key**
2. Set **group_definitions** to a map where *mandatory* is the key and the value is **mandatory_attributes**
3. Set **groups** to *response.groups* of the *canonicalizeAndGroup* algorithm specified in the *DataIntegrity for ECDSA specification*[5], passing the **label_map_factory_function**, **group_definitions** and **vc**
4. Set **mandatory** to the value from **groups.mandatory.matching**
5. Set **nonMandatory** to the value from **groups.mandatory.nonMatching**
6. Return an object named **transformed_data** containing *mandatory* set to **mandatory** and *nonMandatory* set to **nonMandatory**

This object may look like this:

```
{
  mandatory: {
  [
    2,
    "_:b0 <https://schema.org/givenName> \"John\" .\n",
  ],
  [
    3,
    "_:b1 <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> <
      ↪ https://www.w3.org/2018/credentials#
      ↪ VerifiableCredential> .\n",
    ],
    [
    4,
    "_:b1 <https://www.w3.org/2018/credentials#
      ↪ credentialSubject> _:b0 .\n",
  ]
  },
  nonMandatory: {
  [
    0,
    "_:b0 <https://schema.org/birthDate> \"1.1.1970\" .\n",
  ],
  [
    1,
    "_:b0 <https://schema.org/familyName> \"Doe\" .\n",
  ]
  },
}
```

Listing 4.8: Return object of the VC transformation

And with that we transformed the VC into canonical statements that can later be signed by BBS.

4.3.1.4 Hash the proof data

In this algorithm we hash the proof data. This data contains the information about the proof, but also the mandatory attributes, which must be revealed. The hashed data then can be used as the header for the BBS Signature operation, and thus ensures that no information about the proof has changed. It also guarantees that all the correct mandatory data is revealed, as the header value must be the same in the sign, sign verify, proof generate and proof verify process.

The input for this algorithm is the proof config as canonical statements **canonical_proof_config** and a transformed document **transformed_data**.

The output is an object **hash_data** which is a copy of the **transformed_data**, the new hashes of the proof and mandatory data.

For the hashing, we follow these steps:

1. Set **proof_hash** to the result of the hashing of **canonical_proof_config** using SHA-256.
2. Set the **mandatory_hash** to the result of the hashing of **transformed_document.mandatory** using SHA-256. Be aware that mandatory is an array that first needs to be concatenated. In this thesis we don't use any delimiter for concatenation
3. Initialize **hash_data** as a copy of **transformed_data**, then add the key-value pairs *proofHash* - **proof_hash** and *mandatoryHash* - **mandatory_hash**
4. Return **hash_data**

After following these steps, we have an object like this:

```
{
  mandatory: {
  [
    2,
    "_:b0 <https://schema.org/givenName> \"John\" .\n",
  ],
  [
    3,
    "_:b1 <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> <
      ↪ https://www.w3.org/2018/credentials#
      ↪ VerifiableCredential> .\n",
    ],
    [
    4,
    "_:b1 <https://www.w3.org/2018/credentials#
      ↪ credentialSubject> _:b0 .\n",
  ]
  },
  nonMandatory: {
  [
    0,
    "_:b0 <https://schema.org/birthDate> \"1.1.1970\" .\n",
  ],
  [
    1,
    "_:b0 <https://schema.org/familyName> \"Doe\" .\n",
  ]
  },
  proofHash: [34, 31, 47, 124, 176, 144, ...],
  mandatoryHash: [51, 239, 227, 180, 185, ...]
}
```

Listing 4.9: Return object of the VC transformation

Now all the data is in the correct format to generate a BBS Signature.

4.3.1.5 Generate and serialize the signature

This algorithm generates a BBS Signature using the input data.

The inputs for this algorithm are the hashed data **hash_data** and a hmac key **hmac_key**.

To generate the BBS Signature, we follow these steps:

1. Set **proof_hash**, **mandatory_hash**, **mandatory** and **non_mandatory** to the corresponding value in **hash_data**
2. Set **bbs_header** to the concatenation of **proof_hash** and **mandatory_hash**
3. Set **bbs_messages** to the messages in the **non_mandatory** array, which are represented as byte arrays encoded with UTF-8
4. Set **bbs_signature** to the result of the BBS signature operation, defined in chapter 3.5.1 of the BBS Signature Scheme[6], with **bbs_header** and **bbs_messages** as the inputs
5. Initialize **proof_value** to the BBS proof header bytes 0xd9, 0x5d and 0x02
6. Set **components** to an array containing (in this order): **bbs_signature**, **bbs_header**, **public_key**, **hmac_key**, **mandatory_pointers**
7. Set **cbor** to result of the application of the CBOR-encoding[2], where CBOR tagging **Must NOT** be used
8. Set **proof_value** to the concatenation of **proof_value** and **cbor**
9. Set **base_proof** to the value of the base64-url encoding defined in chapter 5 of *RFC 4648*[3] but without padding as mentioned in chapter 3.2 of the same RFC
10. Return **base_proof**

base_proof now contains a valid BBS Signature, generated with the contents of the VC.

4.3.1.6 Add signature to VC

This algorithm adds a BBS Signature to the proof object of a VC.

The inputs to this algorithm are a VC **vc**, the base proof **base_proof** and a proof config **proof_config**.

The output of this algorithm is a signed VC **vc**.

To add the proof to a VC, we follow these steps:

- Add a new key-value pair *proof* - **proof_config** to **vc**
- Add a new key-value pair *proofValue* - **base_proof** to **vc.proof**
- Return **vc**

After following these steps we now have a VC signed by BBS as seen in listing 4.5.

4.3.2 Derive selective disclosure VP

As a holder of a secured VC, one would like to present that VC to a verifier. For that we create a verifiable presentation (VP), which in turn is just a VC containing *@context*, *type* and the VC as *verifiableCredential*. We also want to use the selective disclosure provided by the BBS Signature Scheme, so that we don't need to disclose all the information contained in the VC. We will use the signed VC seen in listing 4.5.
In that example we forced the disclosure of the *first_name*. We define that we want to reveal the *last_name*.

As the inputs of the algorithms, we define following variables:
vc: The VC, **not containing** the proof object
base_proof: The proof object from the VC
selective_pointers: An array containing pointers on what should be revealed. In this example we also want to reveal the *last_name*, so the array looks like this: ["/credentialSubject/last_name"]
ph: The presentation header as a byte array. We will define it as an empty array [] for this example
verification_method: The URL to the verification method

This algorithm handles all the data for a selective disclosure proof creation. First, it generates the derived VC, that will be included in the VP. Then the BBS Proof is created and replaces the original BBS Signature. Lastly a VP is created, the VC is added and a new BBS proof is generated for the VP. After all these steps, the VP is ready to be presented.

To create the derived VP, we follow these steps:

1. Set **bbs_proof**, **label_map**, **mandatory_indexes**, **selective_indexes** and **revealed_document** to their corresponding entry in the response object from the algorithm described in chapter 4.3.2.2, with the inputs **vc**, **base_proof**, **selective_pointers** and **ph**
2. Set **new_proof** to a copy of **base_proof**
3. Replace **new_proof.proofValue** with the result of calling the algorithm in chapter 4.3.2.3, with the inputs **bbs_proof**, **label_map**, **mandatory_indexes** and **selective_indexes**

4. Set **revealed_document.proof** to **new_proof**
5. Create an empty object **vp**
6. Add the key-value pair *@context* - **revealed_document.@context** to **vp**
7. Add the key-value pair *type* - **["VerifiablePresentation"]** to **vp**
8. Create a map with **verifiableCredential** as the key and an array, with **revealed_document** as the only entry, as the value
9. Add **verifiableCredential** to **vp**
10. Set **proof_config** and **canonical_proof_config** to the respective entry in the result of the algorithm described in chapter 4.3.1.2, with the inputs **revealed_document.@context** and **verification_method**
11. Add **proof_config** to **vp**
12. Set **bbs_message** to **vp** as a string
13. Set **bbs_signature** to the result of the BBS signature operation, defined in chapter 3.5.1 of the BBS Signature Scheme[6], with **bbs_message** and the public key stored at **verification_method** as the inputs
14. Add the key-value pair *proofValue* - **bbs_signature** to **vp.proof**
15. Return **vp**

This derived VP may look like this:

```
{
  "@context": [
    "https://www.w3.org/ns/credentials/v2",
    "https://w3id.org/security/data-integrit...",
    "https://raw.githubusercontent.com/ro..."
  ],
  "type": [
    "VerifiablePresentation"
  ],
  verifiableCredential: {[
    {
      "@context": [
        "https://www.w3.org/ns/credentials/v2",
        "https://w3id.org/security/data-integrity/v2",
        "https://raw.githubusercontent.com/ro..."
      ],
      "type": [
        "VerifiableCredential"
      ],
      "credentialSubject": {
        "first_name": "John",
        "last_name": "Doe"
      },
      "proof": {
        "type": "DataIntegrityProof",
        "cryptosuite": "bbs-2023",
        "created": "2024-06-09T14:40:03.606Z",
        "verificationMethod": "...",
        "proofPurpose": "assertionMethod",
        "proofValue": "u2V0Dhd..."
      }
    }
  ]}
}
```

Listing 4.10: Derived VP

As you can see, *first_name* and *last_name* are shown in the VP, but *birth_date* is not. This is the version of the VP that will be presented to the verifier.

4.3.2.1 Parse the base proof

This algorithm parses the BBS signature in the proof object back into its respective components.

The input of this algorithm is a proof object **base_proof**.

The output is an object **parsed_base_proof** containing the parsed base proof values.

We follow these steps to parse the base proof:

1. Check that the **base_proof** start with an *u*, indicating that it is a base64-url encoded string
2. Set **decoded_proof_value** to the result of the base64-url decoding with no padding as described in *The Base16, Base32, and Base64 Data Encodings*[3]
3. Check that **decoded_proof_value** starts with the BBS proof header bytes *0xd9*, *0x5d* and *0x02*
4. Set **components** to the CBOR decoding described in *RFC 8949*[2], starting with the fourth byte in **decoded_proof_value**
5. Set **base_proof_object** to the key-value pairs *bbs_signature* - **components[0]**, *bbs_header* - **components[1]**, *public_key* - **components[2]**, *hmac_key* - **components[3]** and *mandatory_pointers* - **components[4]**
6. Return **base_proof_object**

4.3.2.2 Create disclosure data

This algorithm generates all the data needed to create a selective disclosure proof. It also generates the derived VC.

The inputs of this algorithm are a VC **vc**, the proof containing the signature **base_proof**, the selective pointers **selective_pointers** and a presentation header **ph**.

The output is an object containing an BBS proof, a label map, mandatory indexes, selective indexes and the document that we want to reveale (the adjusted VC).

To generate all this data, we follow these steps:

1. Set **bbs_signature**, **bbs_header**, **public_key**, **hmac_key** and **mandatory_pointers** to their respective value in the response of the algorithm described in chapter 4.3.2.1
2. Define **label_map_factory_function** as the function which is returned from the **create_shuffled_id_label_map_function** defined in chapter 4.3.1.1 with the input **hmac_key**
3. Set **combined_pointers** to the concatenation of **mandatory_pointers** and **selective_pointers**

4. Set **group_definitions** to an object with following key-value pairs: *mandatory* - **mandatory_pointers**, *selective* - **selective_pointers** and *combined* - **combined_pointers**
5. Set **groups** and **label_map** to *response.groups* and *response.labelMap* respectively of the *canonicalizeAndGroup* algorithm specified in the *DataIntegrity for ECDSA specification*[5], passing the **label_map_factory_function**, **group_definitions** and **vc**
6. Set **combined_match** to **groups.combined.matching**
7. Set **mandatory_match** to **groups.mandatory.matching**
8. Set **combined_indexes** to the ordered list of **combined_match.keys**
9. Set **mandatory_indexes** to an empty array. We want to compute the position of the mandatory indexes in the **combined_match** array, so that the verifier knows which indexes were mandatory to reveal
10. For each key in **mandatory_match** find its index in **combined_indexes** and add it to **mandatory_indexes**
11. Set **selective_match** to **groups.selective.matching**
12. Set **mandatory_non_match** to **groups.mandatory.nonMatching**
13. Set **non_mandatory_indexes** to the ordered list of **mandatory_non_match.keys**
14. Set **selective_indexes** to an empty array. This time we want to compute the position of the selective indexes in the **non_mandatory_indexes** list. This list will be used for the BBS proof generation process, to define which messages are going to be revealed
15. For each key in **selective_match** find its index in **non_mandatory_indexes** and add it to **selective_indexes**
16. Set **bbs_messages** to the values of **non_mandatory.values** as byte arrays
17. Set **bbs_proof** to the result of the BBS proof operation defined in chapter 3.5.3 of *The BBS Signature Scheme*[6], with the input **public_key**, **bbs_signature**, **bbs_header**, **ph**, **bbs_messages** and **selective_indexes**
18. Set **revealed_document** to the result of the algorithm described in chapter 3.4.13 of *Data Integrity ECDSA Cryptosuites v1.0*[5], with the inputs **vc** and **combined_pointers**
19. Set **deskolemized_n_quads** to the joined string from the **groups.combined.deskolemizedNQuads** array. In this thesis we do not use a delimiter

20. Set **canonical_id_map** to the result of the JSON-LD canonicalization algorithm with the input **deskolemized_n_quads**

21. Set **verifier_label_map** to an empty map. This maps the canonical blank node identifiers like *c14n0* from the revealed VC to the blank node identifiers created by the issuer

22. For each entry in **canonical_id_map**, where key is *key* and the value is *value*:
    a) Add an entry to **verifier_label_map** where the key *value* and the value is the entry in **label_map** with the key *key*

23. Return an object with following key-value pairs: *bbsProof* - **bbs_proof**, *verifierLabelMap* - **label_map**, *mandatoryIndexes* - **mandatory_indexes**, *selectiveIndexes* - **selective_indexes** and *revealedDocument* - **revealed_document**

4.3.2.3 Serialize the proof value

This algorithm serializes the BBS proof value.

The inputs of this algorithm are a BBS proof **bbs_proof**, a label map **label_map**, an array of mandatory indexes **mandatory_indexes**, an array of selective indexes **selective_pointers** and a presentation header **ph**.

The result of this algorithm is the serialized proof value **proof_value**.

We follow these steps:

- Set **compressed_label_map** to the result of calling the algorithm in chapter 3.5.5 of the *Data Integrity ECDSA Cryptosuites v1.0*[5] with **label_map** as the input
- Initialize **proof_value** as a byte array, where the first three bytes are *0xd9*, *0x5d* and *0x03*
- Set **components** to an array containing (in this order): **bbs_proof**, **compressed_label_map**, **mandatory_indexes**, **selective_pointers** and **ph**
- Set **cbor** to result of the application of the CBOR-encoding[2] on the components array, where CBOR tagging **Must NOT** be used.
- Set **derived_proof_value** to the concatenation of **proof_value** and **cbor**
- Set **derived_proof** to the the base64-url encoding of **derived_proof_value**, as defined in chapter 5 of "RFC 4648"[3] but without padding as mentioned in chapter 3.2 of the same RFC
- Return **proof_value**

4.3.3 Verify derived VP

The VP was presented by a holder to a verifier. The verifier now wants to verify the received VP.

We define following variable:
secured_document: This is the VC inside the received VP, so **vp.verifiableCredential**

To verify the VP we follow these steps:

1. Set **bbs_signature** to **vp.proof.proofValue**
2. Remove the key-value pair **vp.proof.proofValue**
3. Set **bbs_message** to **vp** as a string
4. Set **vp_pk** to the value stored at **vp.proof.verificationMethod**
5. Set **verified_vp** to the result of the BBS Signature Verification algorithm defined in *The BBS Signature Scheme*[6] with the inputs' **vp_pk**, **bbs_signature** and **bbs_message**
6. If **verified_vp** is false, reject the presented VP
7. Set **pk** to the value returned when using the URL from **secured_document.proof.verificationMethod**
8. Set **unsecured_document** to the copy of **secured_document** but with the *proof* value removed
9. Set **proof** to the copy of **secured_document.proof**
10. Set **bbs_proof**, **proof_hash**, **non_mandatory**, **mandatory_hash**, **selective_indexes**, **ph** and **feature_option** to their respective entries in the result of the algorithm described in chapter 4.3.3.1, with **proof** as the input
11. Set **bbs_header** to the concatenation of **proof_hash** and **mandatory_hash**
12. Set **disclosed_messages** to the contents of the **non_mandatory**, each UTF-8 encoded
13. Set **verify** to the result of the BBS Proof Verification algorithm defined in *The BBS Signature Scheme*[6] with the inputs **pk**, **bbs_proof**, **bbs_header**, **ph**, **disclosed_messages** and **selective_indexes**

After running the algorithm, the verifier has the variable **verify**. If it's *true*, the verifier knows that the VP is valid and was not tampered with.

4.3.3.1 Parse Proof Value

This algorithm parses the proof value.

The input is a proof value **proof_value**.

The output is a map containing the BBS proof, a label map, the mandatory indexes, the selective indexes, the presentation header and the feature options.

We follow these steps:

1. Make sure that **proof_value** starts with *u*, which indicates that it is a base64-url encoded value with no padding
2. Set **decoded_proof_value** to the decoding of **proof_value**
3. Make sure that the three first bytes of **decoded_proof_value** are *0xd9*, *0x5d* and *0x03* in that order
4. Set **feature_option** to *baseline*
5. Set **components** to the result of the CBOR decoding of the **decoded_proof_value** starting with the fourth byte. The resulting array must have 5 elements
6. Replace the second object in **components** with the result of the algorithm in chapter 3.5.6 of *Data Integrity ECDSA Cryptosuites v1.0*[5], with the second object of **components** as input value
7. Return a map containing the key-value pairs: *bbsProof* - **components[0]**, *labelMap* - **components[1]**, *mandatoryIndexes* - **components[2]**, *selectiveIndexes* - **components[3]**, *presentationHeader* - **components[5]** and *featureOption* - **feature_option**

4.3.3.2 Create Verify data

This algorithm generates the data for the proof verification process.

The input of this algorithm are the VP without the proof value **document**, the proof **proof**, and the proof without *proofValue* **proof_config**.

The output of this algorithm is a label map containing the BBS proof, the proof hash, the non mandatory indexes, the mandatory hash, the selective indexes and the feature options.

We follow these steps:

1. Set **proof_hash** to the hashed (using SHA-256) canonical representation of the **proof_config** using the *Universal RDF Dataset Canonicalization Algorithm*[4]

2. Set **bbs_proof**, **label_map**, **mandatory_indexes**, **selective_indexes**, and **feature_option** to their respective entries in the result of the algorithm described in chapter 4.3.3.1 with **proof** as the input
3. Set **label_map_factory_function** to the algorithm in chapter 3.4.3 of *Data Integrity ECDSA Cryptosuites v1.0*[5]
4. Set **nquads** to the result of the algorithm *labelReplacementCanonicalize* described in chapter 3.4.1 of *Data Integrity ECDSA Cryptosuites v1.0*[5] with the inputs **document** and **label_map_factory_function**
5. Set **mandatory** and **non_mandatory** to an empty array
6. For each entry in **nquads**, the key is **key** and the value is **value**, separate it into mandatory and non mandatory
   a) If **mandatory_indexes** contains **key**, add **value** to **mandatory**
   b) Else add **value** to **non_mandatory**
7. Set **mandatory_hash** to hash of **mandatory** using SHA-256. We concatenate the elements of the array without a separator
8. Return a map containing the key-value pairs: *bbsProof* - **bbs_proof**, *proofHash* - **proof_hash**, *nonMandatory* - **non_mandatory**, *mandatoryHash* - **mandatory_hash**, *selectiveIndexes* - **selective_indexes**, *presentationHeader* - **ph** and *featureOption* - **feature_option**

4.4 Security Considerations of VCs

The combination of BBS and VC raises two security questions:

1. If we want to use IDs to identify a revoked VC, we run into linkability problems
2. The RDF canonicalization algorithm creates data leakage which leads to linkability

4.4.1 IDs in VCs

While creating a VC, the issuer might add an ID to the VC to enable revocation. A verifier can then check if the ID of the VC, which was presented to them, is in the revocation list of the issuer, and thus accept or reject the presentation.
But there is a big problem with this. If the ID is just added to the VC and is revealed to multiple verifiers, they can link the presentations together based on the ID. With that we would break the unlinkability provided by BBS.

To preserve the unlinkability between the presentations, the holder would need to be able to prove that the ID of the VC is not in the revocation list of the issuer.

As a solution to that problem, we can use something like ALLOSAUR[7]. With this a holder can prove to a verifier that his ID is not in the revocation list, without presenting a unique identifier. In this thesis we will assume that the concept shown in ALLOSAUR works without compromising unlinkability. We also won't look at how it works, as this would be out-of-scope for this thesis.

4.4.2 RDF canonicalization algorithm

In the algorithms described in the Analysis Document, we use RDF multiple times to transform the JSON-LD Document into statements. Using this algorithm can lead to data leakage which in turn can lead to linkability.

Let's look at an example. We will use the example given in chapter 6.1 of *Data Integrity BBS Cryptosuites v1.0*[1]:

```
{
        "@context": [
                "https://www.w3.org/ns/credentials/v2",
                {
                "@vocab": "https://windsurf.grotto-networking
                   ↪ .com/selective#"
                }
        ],
        "type": [
                "VerifiableCredential"
        ],
        "credentialSubject": {
                "sails": [
                {
                        "size": 5.5,
                        "sailName": "Kihei",
                        "year": 2023
                },
                {
                        "size": 6.1,
                        "sailName": "Lahaina",
                        "year": 2023
                },
                {
                        "size": 7.0,
                        "sailName": "Lahaina",
                        "year": 2020
                },
                {
                        "size": 7.8,
                        "sailName": "Lahaina",
                        "year": 2023
                }
                ]
        }
}
```

Listing 4.11: Example: Sails VC

We use the RDF canonicalization algorithm and get these statements:

```
[
        _:c14n0 <https://windsurf.grotto-networking.com/
           ↪ selective#sailName> "Lahaina" .
        _:c14n0 <https://windsurf.grotto-networking.com/
           ↪ selective#size> "7.8E0"^^<http://www.w3.org
           ↪ /2001/XMLSchema#double> .
        _:c14n0 <https://windsurf.grotto-networking.com/
           ↪ selective#year> "2023"^^<http://www.w3.org
           ↪ /2001/XMLSchema#integer> .
        _:c14n1 <http://www.w3.org/1999/02/22-rdf-syntax-ns#
           ↪ type> <https://www.w3.org/2018/credentials#
           ↪ VerifiableCredential> .
        _:c14n1 <https://www.w3.org/2018/credentials#
           ↪ credentialSubject> _:c14n4 .
        _:c14n2 <https://windsurf.grotto-networking.com/
           ↪ selective#sailName> "Lahaina" .
        _:c14n2 <https://windsurf.grotto-networking.com/
           ↪ selective#size> "7"^^<http://www.w3.org/2001/
           ↪ XMLSchema#integer> .
        _:c14n2 <https://windsurf.grotto-networking.com/
           ↪ selective#year> "2020"^^<http://www.w3.org
           ↪ /2001/XMLSchema#integer> .
        _:c14n3 <https://windsurf.grotto-networking.com/
           ↪ selective#sailName> "Kihei" .
        _:c14n3 <https://windsurf.grotto-networking.com/
           ↪ selective#size> "5.5E0"^^<http://www.w3.org
           ↪ /2001/XMLSchema#double> .
        _:c14n3 <https://windsurf.grotto-networking.com/
           ↪ selective#year> "2023"^^<http://www.w3.org
           ↪ /2001/XMLSchema#integer> .
        _:c14n4 <https://windsurf.grotto-networking.com/
           ↪ selective#sails> _:c14n0 .
        _:c14n4 <https://windsurf.grotto-networking.com/
           ↪ selective#sails> _:c14n2 .
        _:c14n4 <https://windsurf.grotto-networking.com/
           ↪ selective#sails> _:c14n3 .
        _:c14n4 <https://windsurf.grotto-networking.com/
           ↪ selective#sails> _:c14n5 .
        _:c14n5 <https://windsurf.grotto-networking.com/
           ↪ selective#sailName> "Lahaina" .
        _:c14n5 <https://windsurf.grotto-networking.com/
           ↪ selective#size> "6.1E0"^^<http://www.w3.org
```

```
            ↪ /2001/XMLSchema#double> .
19        _:c14n5 <https://windsurf.grotto-networking.com/
            ↪ selective#year> "2023"^^<http://www.w3.org
            ↪ /2001/XMLSchema#integer> .
20 ]
```

Listing 4.12: Example: Sails VC as statements

The data leakage problem arises in one specific case. Let's say the holder only discloses information about the sails with the size 7.0 and 7.8, so the last two objects of the credentialSubject.

Now, the holder receives a new credential, where information about a sail, which has an entry before the disclosed sails (size 7.0 and 7.8), has been changed. Let's say that the year for the size 6.0 sail has been updated to 2024. When the RDF algorithm is now run again on the new VC, we get these statements:

```
[
        _:c14n0 <https://windsurf.grotto-networking.com/selective#sailName> "Lahaina" .
        _:c14n0 <https://windsurf.grotto-networking.com/selective#size> "6.1E0"^^<http://www.w3.org/2001/XMLSchema#double> .
        _:c14n0 <https://windsurf.grotto-networking.com/selective#year> "2024"^^<http://www.w3.org/2001/XMLSchema#integer> .
        _:c14n1 <https://windsurf.grotto-networking.com/selective#sailName> "Lahaina" .
        _:c14n1 <https://windsurf.grotto-networking.com/selective#size> "7.8E0"^^<http://www.w3.org/2001/XMLSchema#double> .
        _:c14n1 <https://windsurf.grotto-networking.com/selective#year> "2023"^^<http://www.w3.org/2001/XMLSchema#integer> .
        _:c14n2 <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> <https://www.w3.org/2018/credentials#VerifiableCredential> .
        _:c14n2 <https://www.w3.org/2018/credentials#credentialSubject> _:c14n5 .
        _:c14n3 <https://windsurf.grotto-networking.com/selective#sailName> "Lahaina" .
        _:c14n3 <https://windsurf.grotto-networking.com/selective#size> "7"^^<http://www.w3.org/2001/XMLSchema#integer> .
        _:c14n3 <https://windsurf.grotto-networking.com/selective#year> "2020"^^<http://www.w3.org/2001/XMLSchema#integer> .
        _:c14n4 <https://windsurf.grotto-networking.com/selective#sailName> "Kihei" .
        _:c14n4 <https://windsurf.grotto-networking.com/selective#size> "5.5E0"^^<http://www.w3.org/2001/XMLSchema#double> .
        _:c14n4 <https://windsurf.grotto-networking.com/selective#year> "2023"^^<http://www.w3.org/2001/XMLSchema#integer> .
        _:c14n5 <https://windsurf.grotto-networking.com/selective#sails> _:c14n0 .
        _:c14n5 <https://windsurf.grotto-networking.com/selective#sails> _:c14n1 .
        _:c14n5 <https://windsurf.grotto-networking.com/
```

```
            ↪ selective#sails> _:c14n3 .
19          _:c14n5 <https://windsurf.grotto-networking.com/
            ↪ selective#sails> _:c14n4 .
20 ]
```

Listing 4.13: Example: Updated sails VC as statements

You can see how the ordering is different, and how the blank nodes (_:c14nx) were assigned differently. Even if we didn't disclose any information about the two smaller sails before and after the VC update, a verifier could easily deduce that the content of the VC has changed, which is leaking data. Depending on the size of the VC or what was already revealed, this can even lead to linkability.

There is an easy solution to this problem. As an input to the RDF algorithm we need to pass a function, which takes the canonical labels and replaces them with another value. If we now add a HMAC (using SHA-256) to this function, we can randomize how the canonical label values are replaced each time a VC is created.

Note: It was suggested that the possibility to use KMAC with SHA3-256 should be added to the specification. This was declined by the working group on the basis that SHA-256 is more common.

5 OpenID for Verifiable Presentations

Now that we know how to create and sign VCs/VPs, we want to be able to send them to the other parties. In this thesis, we will only look at the communication between a holder and a verifier, so how VPs are transmitted. As the message layer we use OpenID for Verifiable Presentations(OIDC4VP)[9]. The goal of this chapter is to analyze if OIDC4VP leaks data or even breaks linkability.

5.1 How does OIDC4VP work?

OIDC4VP is a very easy to use protocol. There are just two parties, the verifier and the holder (which can be split up into holder and wallet, but for the sake of simplicity we merge them). After the initial interaction between a holder and verifier, these two parties can start a new session, present the VP and close the session.

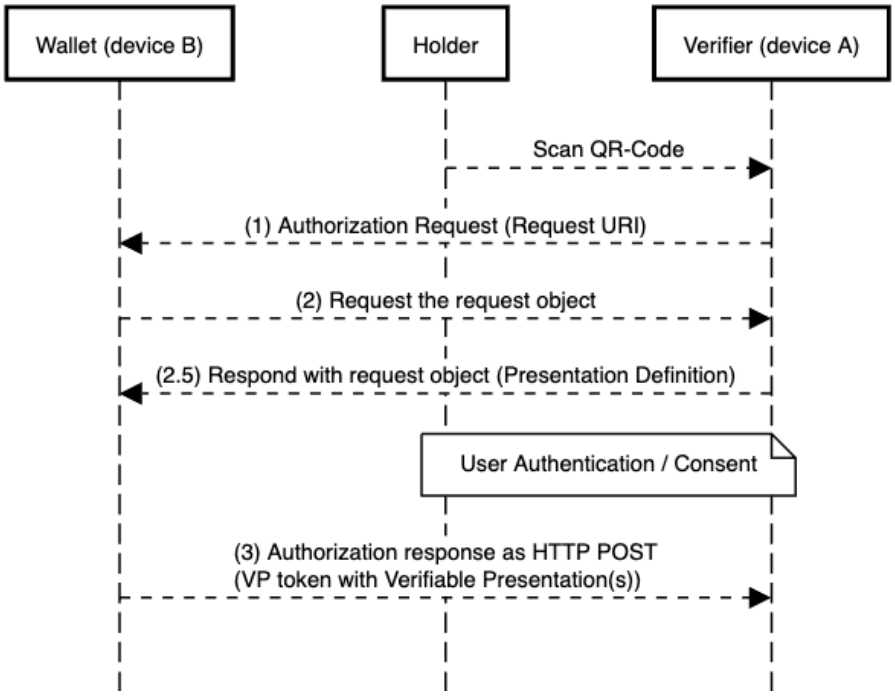

Figure 5.1: Flowchart of OIDC4VP session

The flowchart in figure 5.1 shows the cross-device flow of a OIDC4VP session, meaning the flow, if the initial interaction was not made on the same device, where the VP is

saved (for example scanning a QR code from your laptop screen with your phone).

In the coming chapters we will examine each step in the session for data leakage and linkability.

It's important to note that in this draft, VC and VP can be used interchangeably. OIDC4VP doesn't really mean that it is only for sending VPs between parties, it is a protocol that defines how the messaging between a holder and a verifier works, independent of VC or VP.

Also, all the following chapters use HTTP POST/GET requests for the communication.

5.1.1 Presentation definition

The first step in the session is the authorization request. In this step the verifier communicates the requirements of the presented VP to the holder, like what type of credential or in which format. The verifier can also define which individual attributes must be revealed, which is the wanted selective disclosure.

Such an object might look like this:

```
{
  "id": "example with selective disclosure",
  "input_descriptors": [
    {
      "id": "ID card with constraints",
      "format": {
        "ldp_vp": {
          "proof_type": [
            "bbs2023"
          ]
        }
      },
      "constraints": {
        "limit_disclosure": "required",
        "fields": [{"path": [
              "$.type"
            ],
            "filter": {
              "type": "string",
              "pattern": "IDCardCredential"
            }
          },{"path": [
              "$.credentialSubject.first_name"
            ]
          },
        ]
      }
    }
  ]
}
```

Listing 5.1: Example of a presentation definition

Such an object is called **presentation_definition**. There are many new attributes in this object, so let's define them from the top to the bottom.

1. The *id* in the root is used to identify a specific presentation_definition. It **MUST**

be present

2. The *input_descriptors* is an array describing the different credentials that are requested. It **MUST** at least contain one entry
   a) The *id* inside an input_descriptor entry is used to identify a specific entry. It **MUST** be present for each entry in the *input_descriptors* array
   b) The *format* describes the format of the requested credential. It **MUST** be present
      i. For this thesis we use one of two entries for the *format* object, *ldp_vc* stating that the requested object must be a VC or *ldp_vp* stating that it must be a VP.
      ii. The *proof_type* is an array which defines which proof types are accepted. It **MUST** be present. In this thesis the only proof type that is accepted is *bbs2023*.
   c) Next we have the *constraints* object. This defines the constraints of the requested credential. It **MUST** be present
      i. If we want to use selective disclosure, we must define *limit_disclosure* as *required*. If we don't want selective disclosure, *limit_disclosure* is not present
      ii. The *fields* entry is an array which defines which fields must be present, and what the content of those fields might be. It **MUST** be present
         A. The *path* defines the path of a field that must be present. $ is the root of the presented credential. In listing 5.1, the first entry in the *fields* array defines that the **type** attribute must be present in the presented credential
         B. If *path* is present we may also use *filter* which lets us filter for specific entries. Again, in listing 5.1, the first entry filters for the type *IDCardCredential* inside the **type** attribute with *patter*, meaning that *IDCardCredential* must be present. We also define the type of the value, which in this case is a *string*
         C. In the case of selective disclosure, we can also use *path* to define which attributes inside the **credentialSubject** must be presented. In listing 5.1 we define that *$.credentialSubject.first_name* must be presented

Now that we know how a presentation definition looks, we can integrate it into the authorization request in two different ways:

- Send the whole object as a part of the initial HTTP POST request
- Send a *presentation_definition_uri* as part of the initial HTTP POST request, which then allows the holder to retrieve the object from a static server

But why are there two different integrations?
If everything is done on the same device, it is easy to send all the data in the background. But if we are using two devices, we might need to scan a QR Code with the device containing the credential from a second device. There might be the problem that the size of the *presentation_definition* is too large for a QR Code (max. of 4296 alphanumeric characters).

We also want to tell the holder where to send the presentation to. In this thesis we will use the *direct_post* response mode, which defines that the response must be sent to an endpoint using an HTTP POST request. The verifier must also set the *response_uri* parameter with the corresponding uri.

5.1.2 Presentation submission

After the holder got the *presentation_definition* as well as the *response_uri* and prepared the VP, he can respond to the verifier.

For the response, the holder needs two parameters:

1. A *vp_token* which is just a VP. This VP must be base64-url encoded. It can also be an array of VPs, which each are base64-url encoded
2. A *presentation_submission*. This contains a map for the VC inside the VP

```
{
  "id": "Presentation example 1",
  "definition_id": "Example with selective disclosure",
  "descriptor_map": [
    {
      "id": "ID card with constraints",
      "format": "ldp_vp",
      "path": "$",
      "path_nested": {
        "format": "ldp_vc",
        "path": "$.verifiableCredential[0]"
      }
    }
  ]
}
```

Listing 5.2: Example of a presentation submission

Figure 5.2 is an example of a presentation definition. We already know the use of the *id*. The *definition_id* is just the id from the *presentation_definition*. The *descriptor_map* is an array that contains the definition for all the VPs inside the *vp_token*. In that array we have an *id* for that entry, the *format* of the corresponding credential in the *vp_token* and

the root *path* value.
We then have the *path_nested* object, which contains the *path* of the corresponding credential inside the root credential (like a VC inside a VP) and the *format* of that credential.

To finish the session, the holder just returns the *vp_token* and the *presentation_submission* to the verifier.

5.2 OIDC4VP Flow

In this chapter we will look at all the steps between the initial interaction until the verification is complete. We have 3 entities in this session:

- The holder. This entity has the VP that will be presented
- The verifier. This entity requests the VP and verifies it
- The verifier response endpoint. This entity is a part of the verifier. It receives the VP from the holder and stores it until the verifier requests it

The flow between these three parties is shown in the following sequence-diagram:

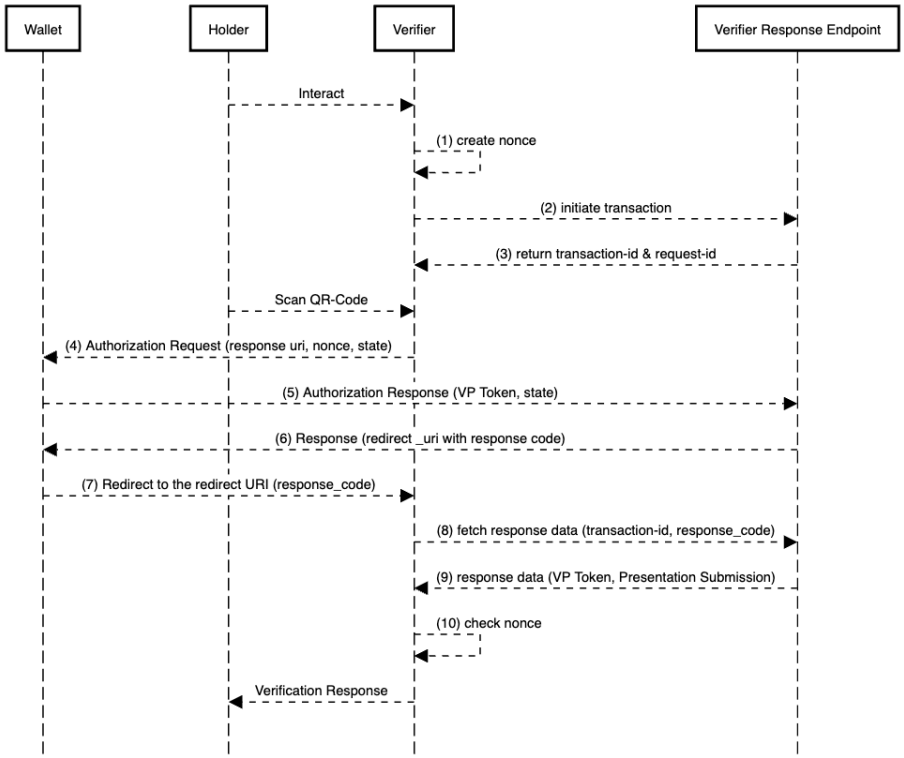

Figure 5.2: Full flowchart of a OIDC4VP session

With all this information, let's go step by step and look what exactly needs to be done:

1. The verifier wants to receive a VP from a holder. The first step is to generate a random number called a **nonce**. This value is used to counter replay attacks. These attacks try to use data from old sessions and re-submit them to a verifier. With the **nonce**, the verifier can check if there is an open session connected to it, and if not, the invalid presentation is rejected (Step 1 in figure 5.2)

2. The next step is to initiate a transaction between the verifier and the verifier response endpoint. The endpoint responds with two random numbers, the **transaction_id** and the **request_id**. The endpoint then keeps a map of both values as they belong together. The **request_id** is given to the holder, who uses it to tell the endpoint to which session (transaction) the presentation belongs. At the end of the session, a verifier can then fetch the presentation from the endpoint using the **transaction_id** (Step 2 and 3 in figure 5.2)

3. The verifier now generates a QR code for the holder to scan. In this thesis we will use the response mode *direct_post*. This mode allows to the holder to send the response to the verifier using a redirect. The QR code must contain the following information (Step 4 in figure 5.2):
   - **response_mode**: In this thesis, we set this value to *direct_post*
   - **response_uri**: This value contains the uri where the holder need to send the VP to
   - **client_id_scheme**: This value defines the type of the **client_id** value
   - **client_id**: In this thesis, this value must be the same as **response_uri**
   - **response_type**: This value defines the type of the response. For OIDC4VP this value must be set to *vp_token*
   - **nonce**: The **nonce** as defined above
   - **state**: The **request_id** as defined above
   - **presentation_definition**: The presentation definition as defined in 5.1.1

4. After scanning the QR code, the holder generates the VP corresponding with the **presentation_definition**. It is important to note that there are two special things that the holder needs to add to the VP. First the *nonce* needs to be added in the proof object of the VP. Then the *client_id* also needs to be added to the proof object, but as the domain. This domain value must also be used by the holder as an input to the BBS proof generation algorithm. This VP is called the *vp_token*. The holder also generates the *presentation_submission*. These values are then sent together with the **state** to the verifier response endpoint defined in **response_uri** (Step 5 in figure 5.2)

5. The verifier response endpoint receives the response from the holder. Using the *state* (which initially was the **request_id**), the endpoint can match the response from the holder to an open transaction with the verifier. Then the endpoint returns a *redirect_uri* and *response_code* to the holder. This points the holder back to

the verifier to message him that the presentation was sent to the endpoint. The code is again a random number, which is linked to the respective presentation. There would be a possible implementation without this response to the holder. In that implementation the verifier would continuously pull the new data from the endpoint to check if the VP was presented. This would enable a session fixation attack (Step 6 in figure 5.2)

6. The holder now has the *redirect_uri* and *response_code*. They now send an HTTP POST request to the *redirect_uri* with the *response_code* to the verifier. After sending the request, the session is finished for the holder (Step 7 in figure 5.2)

7. After receiving the *response_code* from the holder, the verifier now fetches the *vp_token* and *presentation_submission* from the endpoint. For that they send an HTTP POST request containing the **transaction_id** and **response_code**. The *transaction_id* is only known by the verifier, but it might be that the system is compromised. As the *response_code* is only known to the part of the system, where the holder was redirected to, so the compromised part of the system is not able to pull the data of the holder. The endpoint then responds with the corresponding *vp_token* and *presentation_submission* (Step 8 and 9 in figure 5.2)

8. In the last step, the verifier now verifies the nonce included in the *vp_token*. If that nonce is correct, they can verify the VP and either accept or reject it (Step 10 in figure 5.2)

5.3 Security concerns

Adding OIDC4VP to the VC/VP-BBS stack raises two new security concerns:

- We want to use nonce to avoid replay attacks
- Session fixation attacks are possible if the holder is not re-reouted

As all the data is generated randomly, no problems with unlinkability and selective disclosure is created.

5.3.1 Replay attacks

If a holder sends data to a verifier trought a channel that has a middle man in it, a replay attack is possible even if the data is encrypted. A replay attack is an attack, where data that was already presented once, is presented again to, for example, gain access to an account. To prevent such attacks, values called *nonce* are used. When a session is created, a *nonce* is given to the holder. The holder then can present data with that nonce once. After receiving the data, the verifier then removes the nonce from the active list, so that presenting data with the same nonce again is rejected. A basic way to create a *nonce* is a counter. Using a counter is only secure until the current value is known. To replay already presented data, an attacker would then just need to add a higher value

as the current counter. A more secure version would be to generate random numbers, and then adding those numbers to a active list. When a session is finished, the nonce of that session is removed from that list.

5.3.2 Session fixation

A session fixation attack works when an attacker steals a token from a holder and uses that to keep a session open. In OIDC4VP a session fixation attack can be used to get the unencrypted data from a presentation. If attacker infiltrated a verifier, they could get the transaction-id and request-id for a session (see figure 5.2). Using those IDs, the attack could then query the verifier backend to get the response data from the session. This attack can only happen if steps 6 and 7 in figure 5.2 are not implemented, thus letting the verifier querying the endpoint until they get a response.
To prevent this attack from happening, steps 6 and 7 in figure 5.2 should be implemented. This adds two security features to the flow. First a random response code is generated. The verifier and the attacker wouldn't receive the data from the backend with just the transaction-id and request-id, as they would also need the response code. If the holder then would just send back the response code to the verifier, the attacker would also have it, which defeats the point of that code. That is why the holder also gets a redirect uri from the endpoint.
The holder send the reponse code to the redirect uri, which is in another part of the verifiers system. With that the attacker never has all pieces and thus isn't able to access the presented data.

6 Sandbox

While researching for the integration of BBS into VCs, I came by the *Data Integrity BBS Cryptosuites v1.0*[1] specification. This specification contains all the necessary information about how to process a VC into statements, selectively disclose those statements, sign the VC and then also verify that VC.
Sadly, the specification was a bit of a mess (it was updated recently, so it is much better now). That is why I talked with Greg Bernstein, one of the authors of the specification, which linked me to his test implementation. Using that implementation I went through the specification and described all the steps in chapter 4. I also used the same implementation to generate all the VCs & VPs seen in this thesis.

If you, the reader, would also like to know how everything works, you can download the library here: `testurl`.

Here some quick tips how to use the library:

- All the files that can be changed or run are in the /examples folder
- Inside the /examples/input folder there are different files that can be changed. example.json contains the VC structure, mandatory.json the mandatory pointers and selective.json the selective pointers
- To run files, cd into the /examples directory and run 'node <filename.js>'
- Running createSignedBase.js generates a signed VC, which can be found in the /examples/output folder
- Running verifySignedBase.js verifies the generated VC and prints the result in the terminal
- Running deriveDocument.js generates a derived VC, using the before generated signed VC and the selective pointers. The derived VC can also be found in the /examples/output folder
- Running verifyDerivedDocument.js verifies the derived VC, and prints the result in the terminal
- If you want to create a custom context, create a new js file in /examples/contexts. Then import the new file into /examples/documentLoader.js and add the new url to the existing list

Do not forget to run 'npm i' in the root folder after downloading the library, to install all dependencies.

7 Future Work

In the timeframe of this thesis, not all planned technologies were analyzed. These can be part of future works to extend the features of this stack. These features are Blind BBS Signatures, BBS Pseudonyms and Link Secrets.

In this thesis, the use case presented in chapter 3 is only for demonstrative purposes. We don't yet have a concrete use case, where all the features of the described technologies really show their usefulness. With time, when digitalizing credentials becomes more common, one might find a use case where selective disclosure and unlinkability are very important.

An other point for future work would be the implementation and testing. This thesis only contains theoretical analysis of the different technologies. Implementations using different programming languages might create new problems which were not part of this thesis.

At last, the communication between an issuer and a holder must be analyzed. OpenID Connect for Verifiable Presentations would be used, but this was excluded from this thesis.

8 Conclusion

The most important thing to say is that **it works**! The goal of this thesis, to analyze the inner workings of the different technologies and to find solutions to keep the unlinkability intact, is reached.

It was very interesting to research for this project. After learning how BBS works under the hood the semester before for the Project 2 module, learning how to integrate it with other technologies was very interesting and challenging. After setting up the project, looking at the risks and arranging the different sprints and sub-goals for 3 weeks, I was ready to start researching.

The first step was to find out how the verifiable credentials work. After reading the VC specification for some time, I encountered two problems. The first problem was the sheer size of the specification. There were so many optional features or different ways to do something that I had to narrow it down. So after sometime testing, I defined a minimal set of features found in chapter 4.2. Based on that I could then create a VC and concentrate on how to add BBS into the mix, which was the second problem. After some research I found the *Data Integrity BBS Cryptosuites v1.0*[1] specification, which states exactly how to merge both technologies. Problem was, it was not comprehensible. In the time after my research into that topic, the specification was updated and is thus more readable, but it was really hard to understand. That is why I contacted one of the authors of the specification, Greg Bernstein, which referred me to his implementation. Using that implementation I then understood the different steps of the specification and wrote them up. After finishing analyzing all the algorithms, I searched for security problems. Two problems were found, one in the algorithm of the VC transformation (so that the VC could be signed) and a second one regarding identifying meta-data, which were both solved. At the end of the thesis a third problem popped up, but it was not analyzed because of time constraints. This problem is about the signing of a VP. The verifier doesn't trust the holder, so how can the holder sign the VP that he generated. This is something that should be analyzed in future work.

After finishing up researching VCs, the next topic was OpenID Connect for Verifiable Presentations. Here again, the first step was to read the specification. This time it was not as extensive as the VC specification, and there were not many different optional features, so it was more straight forward. So after a week of reading and documenting, this step of the process was done.

After that there was not much time left until the due date, so out of time constraint reasons, the research on Blind BBS Signatures, pseudonyms and link secrets was not done.

Another thing besides the messy VC-BBS specification creating time problems, was the documentation. For me, writing is very difficult and takes a lot of time and tries for getting it to an acceptable level. For example the research, documenting and implementation of VCs & VPs should have only take one week each. In the end it was 3 weeks of research and 4 weeks of testing Greg Bernsteins library and documenting the findings. At the same time I needed to prepare the book entry and the poster, create my defense to make a test presentation with expert and guests and also create the video, the small presentation and prepare the demo day. Each of those assignments took time away from the main task of documenting and thus leaving me with less time than expected.

At the end of the day, I'm happy with my results. I've learned a lot of new things about different technologies, but also about researching and documenting. It was a very interesting semester, with a lot of up and downs. I will use the acquired knowledge to implement this technology stack and to keep researching the missing features.

A big thanks goes out to my advisors, Annett Laube and Reto Koenig. Dealing with me on a weekly basis must have been tiring. They also really helped me to get throught documenting everything. Also a big thanks to my expert, Andreas Spichiger, who took time to listen to my presentations and critiqued it. This feedback helped me to create a better defense.

Also very important are my mother, my sister and my aunt. All these women generously helped me by proof reading the documentation, listening to my presentations and by being my guinea pigs to see if the text was abstract enough. Big thanks to them!

Last but not least, big thanks to my girlfriend. Without her I would have probably starved. Because of her I got a nice meal each evening, without needing to consume time cooking. Those meals gave me energy to keep going. And also a big thanks for helping me in coming up with ideas on how to explain certain things.

And on a small side note, also small thanks to myself on not giving up after learning that there is no time to implement something.

Bibliography

[1] Greg Bernstein and Manu Sporny. Data integrity bbs cryptosuites v1.0. https://www.w3.org/TR/vc-di-bbs/, 2024. Accessed on April 1, 2024.

[2] C. Bormann and P. Hoffman. Rfc 8949, concise binary object representation (cbor). https://www.rfc-editor.org/rfc/rfc8949, 2020. Accessed on April 6, 2024.

[3] S. Josefsson. The base16, base32, and base64 data encodings. https://datatracker.ietf.org/doc/html/rfc4648, 2006. Accessed on April 6, 2024.

[4] Dave Longley. Rdf dataset canonicalization. https://www.w3.org/TR/rdf-canon/, 2024. Accessed on April 1, 2024.

[5] Dave Longley and Manu Sporny. Data integrity ecdsa cryptosuites v1.0. https://www.w3.org/TR/vc-di-ecdsa/, 2024. Accessed on April 3, 2024.

[6] Tobias Looker, Vasilis Kalos, Andrew Whitehead, and Mike Lodder. The bbs signature scheme. https://datatracker.ietf.org/doc/html/draft-irtf-cfrg-bbs-signatures-05, 2023. Accessed on March 17, 2024.

[7] Michael Lodder Samuel Jaques and Hart Montgomery. Allosaur: Accumulator with low-latency oblivious sublinear anonymous credential updates with revocations. https://eprint.iacr.org/2022/1362,, 2022. Accessed on May 6, 2024.

[8] Manu Sporny, Dave Longley, David Chadwick, and Orie Steel. Verifiable credentials data model v2.0. https://www.w3.org/TR/vc-data-model-2.0, 2023. Accessed on March 19, 2024.

[9] O. Terbu, T. Lodderstedt, K. Yasuda, and T.Looker. Openid for verifiable presentations - draft 20. https://openid.net/specs/openid-4-verifiable-presentations-1_0.html, 2023. Accessed on March 24, 2024.

[10] TNO. Self-sovereign identity: a simple and safe digital life. https://www.tno.nl/en/technology-science/technologies/self-sovereign-identity/, 2022. Accessed on March 24, 2024.