Bern University
of Applied Sciences

Bachelor's Thesis

Unlinkability of Verfiable 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 May 20, 2024

- Engineering and Computer Science
- Computer Science

Abstract

Here an abstract might be placed.

Contents

1. Introduction

Self-sovereign Identity (SSI)[1] is a concept where individuals can control their digital identity and what data is shared with whom. In the real world physical credentials like an ID or a driver's license are used, which makes selectively disclosing information hard, as there is more information on the credentials as there is needed in one presentation. If we digitalize these credentials, we create the option for individuals to selectively disclose information. Verifiable credentials (VCs)[11] are a type of digital credentials that can be used to represent physical credentials. These digital credentials support multiple signing themes, one of which is the BBS Signature scheme (BBS)[9]. This signature scheme is based on the trust triangle seen in figure 1.1, where there are three involved parties.

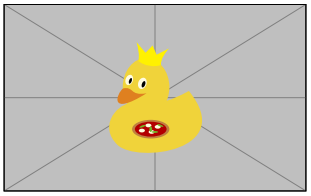

Figure 1.1.: The trust triangle

These parties are the issuer, holder and verifier. The issuer supplies the credential to the holder, which stores it. The holder is then able to present it to a verifier, which checks its validity and content. The verifier uses the signature to also check the authenticity and integrity of the supplied credential. Besides signing, this signature scheme also supports proof creation, which enables unlinkability. Unlinkability is the concept that there is no link between each credential presentation and different verifiers. BBS also supports selective disclosure, to only disclose a subset of the information contained in the credential.

In this thesis we want to investigate the use of the BBS Signature Scheme in a real world scenario. We assume that the BBS Signature scheme has no flaws, and the promoted unlinkability of the scheme holds in every situation. We will also only use verifiable-credentials with data integrity, to not only protect the attributes of the credential (like the name, birthdate ...) but to also protect the meta-data of the verifiable credential. For the data integrity we will also use the BBS Signature Scheme.

The credentials need to send between the issuer and holder and between the holder and the verifier. For this thesis we will only look at the messaging between the holder

and the verifier and assume that the VC creation and transportation to the holder is secure. For the presentation of the credentials there is an extension of VCs called a Verifiable-presentation (VP)[11]. To present the VP we use OpenID-Connect for Verifiable-Presentations (OIDC4VP)[12]. For OIDC4VP we assume that all communication is secured and encrypted.

Why not login? First of we would like to simply the process of a returning customer. Think about how often you log in into Netflix, either with your email and password or with the cached login token. It would be very unnecessary to present your VC each time you would like to watch a show, so we would like recognition. To solve that problem we will look at Pseudonyms[6] in chapter **??**.

These Pseudonyms are used for a verifier to recognize already before seen individual. This extension of the BBS Signature scheme should not break the unlinkability of the BBS proofs as they are only linkable between one verifier and the holder, but will still be analyzed.

Lastly we will look at the problem, where if we need to present two different VCs (like a driver's license and the vehicle registration document), how can we proof that both documents are owned by the same holder. For that we will make use of link secrets[4], which is a secret value that only the holder knows. In this thesis we want to analyse these concepts in conjunction with the BBS Signature scheme, to see if there is any data leakage or if the unlinkability provided by the BBS Signature Scheme will break. For this we will investigate each concept step by step, based on their respective specifications, and implement them if needed.

2. The BBS Signature Scheme

3. Use Cases

3.1. Use Case 1 (why use vc)

3.2. Use Case 2 (why use pseudonyms)

3.3. Use Case 3 (why use link secrets)

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[11] 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, this can be attributes like birthday, first and last name and so on. After creation, the issuer send the VC to the holder.

Now the holder would like to present this VC to verifier. Using the VC, the holder can generate a Verifiable Presentation[11], which is very close to a VC in structure. It also contains meta-data that follows the same rule 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/roblesjoel
                   ↪ /BA_Thesis_BBS_Signatures/docs/context/
                   ↪ example_1.jsonld"
        ],
}
```

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 be an attribute that can 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 signed the VC and with that guaranties its integrity, or it contains a BBS proof, which enables selective disclosure between 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 look like this:

```
{
        "@context": [
                "https://www.w3.org/ns/credentials/v2",
                "https://w3id.org/security/data-integrity/v2",
                "https://raw.githubusercontent.com/roblesjoel/BA_Thesis_BBS_Signatures/docs/context/example_1.jsonld"
        ],
        "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**, **mandatory_attributes** 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/roblesjoel/BA_Thesis_BBS_Signatures/docs/context/example_1.jsonld"
        ],
        "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[5]
   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*[7]
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/roblesjoel/
      ↪ BA_Thesis_BBS_Signatures/docs/context/example_1.
      ↪ jsonld",
  ],
}
```

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*[8], 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 revealed. The hashed data then can be used as the header for the BBS Signature operation, and thus ensuring that no information about the proof is changed, but also 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 inputed 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[9], 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[3], 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*[5] 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 a VC signed by BBS as seen in Listing 4.5.

4.3.2. Derive selective disclosure VPs

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 same example as in chapter **??**.
In that example we forced the disclosure of the *last_name*. We define that no more information should be revealed, meaning that the *first_name* will not be revealed to the verifier.

We define following variables:
vc: The VC, **not containing** the proof object
base_proof: The proof object from the VC
selective_pointers: A array containing pointers on what should be revealed. As already mentioned, in this example we will not reveal anything, so this variable is an empty array []
ph: The presentation header as a byte array. We will define it as an empty array [] for this example

To create the derived VP we call the algorithm in chapter 2.1 of the Analysis document, with the inputs **vc**, **base_proof**, **selective_pointers** and **ph**

This derived VP may look like this:

```
{
        "@context": [
                "https://www.w3.org/ns/credentials/v2",
                "https://w3id.org/security/data-integrity/v2",
                "https://raw.githubusercontent.com/RockstaYT/BA_Thesis_BBS_Signatures/docs/context/example_1.jsonld"
        ],
        "type": [
                "VerifiableCredential"
        ],
        "credentialSubject": {
                "first_name": "Joel"
        },
        "proof": {
                "type": "DataIntegrityProof",
                "cryptosuite": "bbs-2023",
                "created": "2024-04-17T23:41:58.089Z",
                "verificationMethod": "did:key:zUC7De...",
```

```
                    "proofPurpose": "assertionMethod",
                    "proofValue": "u2V0..."
            }
}
```

Listing 4.10: Derived VP

As you can see, the *last_name* is not shown as it is not being revealed.

4.3.3. Verify derived VP

A verifier has now received a selectively disclosed VP, which he would like to verify.

We define following variable:
secured_document: This is the VP that was received from the holder

To verify the VP we follow these steps:

- Set **unsecured_document** to the copy of **secured_document** but with the *proof* value removed
- Set **proof_config** to the copy of **secured_document.proof** but with *proofValue* value removed
- Set **proof** to the copy of **secured_document.proof**
- 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 3.1 of the Analysis Document, with **unsecured_document** and **proof** as the input
- Set **bbs_header** to the concatenation of **proof_hash** and **mandatory_hash**
- Set **verify** to the result of the BBS Proof Verification algorithm defined in *THE BBS Signature Scheme*[9] with the inputs **pk**, **bbs_proof**, **bbs_header**, **ph**

If **verify** is true, the verifier knows that the VP is valid and was not tampered with.

4.3.4. Security Considerations of VCs

Important!!! Signatures in VC also are identifying, vc changes from sig to proof
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

4.3.4.1. 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 a solution like ALLOSAUR[10]. With this solution 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.

4.3.4.2. 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*[2]:

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

Listing 4.12: Example: Sails VC as statements

Now 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> .
```

```
12          _:c14n3 <https://windsurf.grotto-networking.com/
               ↪ selective#year> "2020"^^<http://www.w3.org
               ↪ /2001/XMLSchema#integer> .
13          _:c14n4 <https://windsurf.grotto-networking.com/
               ↪ selective#sailName> "Kihei" .
14          _:c14n4 <https://windsurf.grotto-networking.com/
               ↪ selective#size> "5.5E0"^^<http://www.w3.org
               ↪ /2001/XMLSchema#double> .
15          _:c14n4 <https://windsurf.grotto-networking.com/
               ↪ selective#year> "2023"^^<http://www.w3.org
               ↪ /2001/XMLSchema#integer> .
16          _:c14n5 <https://windsurf.grotto-networking.com/
               ↪ selective#sails> _:c14n0 .
17          _:c14n5 <https://windsurf.grotto-networking.com/
               ↪ selective#sails> _:c14n1 .
18          _: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) where 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 and 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 label 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.

4.4. OpenID for Verifiable Presentations

Now that we know how to create and sign VCs/VPs, we want to be able to send them 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)[12]. The Goal of this chapter is to analyze if OIDC4VP leaks data or even breaks linkability.

4.4.1. How does OIDC4VP work?

OIDC4VP is a very easy to used 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, in just four easy steps these two parties can start a new session, present the VP and close the session.

The see the magic of this protocol in depth, please read chapter 4 of the Analysis document.

Let's use the example from chapter **??**.

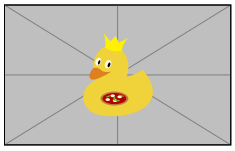

Figure 4.1.: Flowchart of OIDC4VP session

The flowchart in figure 4.4.1 shows the cross device flow of a OIDC4VP session, so 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.

4.4.2. Request

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 in turn is our beloved 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 4.14: Example of a presentation definition

Such an object is called **presentation_definition**. Now 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
   - The *id* inside a input_descriptor entry is used to identify a specific entry. It **MUST** be present for each entry in the *input_descriptors* array
   - The *format* describes the format of the requested credential. It **MUST** be present
     - 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.
     - 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 it *bbs2023*.
   - Next we have the *constraints* object. This defines the constraints of the requested credential. It **MUST** be present
     - 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
     - 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
       * The *path* defines the path of a field that must be present. $ is the root of the presented credential. In listing 4.14, the first entry in the *fields* array defines that the **type** attribute must be present in the presented credential
       * If *path* is present we may also use *filter* which lets us filter for specific entries. Again, in listing 4.14, 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*

* In the case of selective disclosure, we can also use *path* to define which attributes inside the **credentialSubject** must be presented. In listing 4.14 we define that *$.credentialSubject.first_name* must be presented.

Now that we know how this 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).

But we are not finished there. 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.

4.4.3. Response

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 just is a VP. This VP must be Base64url encoded. I can also be an array of VPs, which each are Base64url 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 4.15: Example of a presentation submission

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.

4.5. BBS with Pseudonyms

4.6. BBS with Link Secrets

4.7. BBS with Blind Signatures

A. VC Preparation

B. VP Derivation

B.1. Derive VP

This algorithm handles all the data for a selective disclosure proof creation.

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

The output is an altered VC **reavealed_document** only containing the data to be revealed.

We follow these steps:

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

B.2. Create disclosure data

This algorithm creates the data for the selective disclosure of a VP.

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

The output is an object containing the BBS proof, a label map, mandatory indexes, selective indexes and the revealed document.

We follow these steps:

- 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 B.3
- 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**
- Set **combined_pointers** to the concatenation of **mandatory_pointers** and **selective_pointers**
- Set **group_definitions** to an object with following key-value pairs: *mandatory* - **mandatory_pointers**, *selective* - **selective_pointers** and *combined* - *combined_pointers*
- Set **groups** and **label_map** to *response.groups* and *response.labelMap* respectively of the *canonicalizeAndGroup* algorithm specified in the DataIntegrity for ECDSA specification[8], passing the **label_map_factory_function**, **group_definitions** and **vc**.
- Set **combined_match** to **groups.combined.matching**
- Set **mandatory_match** to **groups.mandatory.matching**
- Set **combined_indexes** to the ordered list of **combined_match.keys**
- 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
- For each key in **mandatory_match** find its index in **combined_indexes** and add it to **mandatory_indexes**.
- Set **selective_match** to **groups.selective.matching**
- Set **mandatory_non_match** to **groups.mandatory.nonMatching**
- Set **non_mandatory_indexes** to the ordered list of **mandatory_non_match.keys**
- 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.
- For each key in **selective_match** find its index in **non_mandatory_indexes** and add it to **selective_indexes**
- Set **bbs_messages** to the values of **non_mandatory.values** as byte arrays
- Set **bbs_proof** to the result of the bbs proof operation defined in chapter 3.5.3 of *The BBS Signature Scheme*[9], with the input **public_key**, **bbs_signature**, **bbs_header**, **ph**, **bbs_messages** and **selective_indexes**

- Set **reavealed_document** to the result of the algorithm described in chapter 3.4.13 of *Data Integrity ECDSA Cryptosuites v1.0*[8], with the inputs **vc** and **combined_pointers**
- Set **deskolemized_n_quads** to the joined string from the **groups.combined.deskolemizedNQuads** array. In this thesis we do not use a delimiter
- Set **canonical_id_map** to the result of the JSON-LD canonicalization algorithm
- 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
- For each entry in **canonical_id_map**:
  - Add an entry to **verifier_label_map** where the key is the current value of the **canonical_id_map** entry and the value is the entry where the key the in **label_map** is the same as the current key from **canonical_id_map**
- Return an object with following key-value pairs: *bbsProof* - **bbs_proof**, *verifierLabelMap* - **label_map**, *mandatoryIndexes* - **mandatory_indexes**, *selectiveIndexes* - **selective_indexes** and *revealedDocument* - **reavealed_document**

B.3. Parse the base proof

This algorithm parses a base proof value back to objects.

The input of this algorithm is a base proof **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:

- Check that the **base_proof** start with a *u*, indicating that it is a base64-url encoded string.
- 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*[5]
- Check that **decoded_proof_value** starts with the proof header bytes *0xd9*, *0x5d* and *0x02*
- Set **components** to the cbor decoding described in *RFC 8949*[3], starting with the fourth byte in **decoded_proof_value**
- 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]**

- Return **base_proof_object**

With that we parsed the base proof bytes back to objects.

B.4. Serialize the proof value

This algorithm serializes a 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*[8] with **label_map** as the input
- Initialize **proof_value** as an 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[3] on the components array, where CBOR tagging **Must NOT** be used.
- Concat **proof_value** and **cbor**
- Set **derived_proof** to the value of the base64-url encoding defined in chapter 5 of "RFC 4648"[5] but without padding as mentioned in chapter 3.2 of the same RFC
- Return **proof_value**

C. VP verification

C.1. 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:

- Set **proof_hash** to the hashed (using SHA-256) canonical representation of the **proof_config** using the *Universal RDF Dataset Canonicalization Algorithm*[7]
- 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 C.2 with **proof** as the input
- Set **label_map_factory_function** to the algorithm in chapter 3.4.3 of *Data Integrity ECDSA Cryptosuites v1.0*[8]
- Set **nquads** to the result of the algorithm *labelReplacementCanonicalize* described in chapter 3.4.1 of *Data Integrity ECDSA Cryptosuites v1.0*[8] with the inputs **document** and **label_map_factory_function**
- Set **mandatory** and **non_mandatory** to an empty array
- For each entry in nquads, the key is **key** and the value is **value**, separate them into mandatory and non mandatory
  - If **mandatory_indexes** contains **key**, add **value** to **mandatory**
  - Else add **value** to **non_mandatory**
- Set **mandatory_hash** to hash of **mandatory** using SHA-256. We concatenate the elements of the array without a separator
- 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**

C.2. 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:

- Make sure that **proof_value** starts with *u*, which indicates that it is a base64-url encoded value with no padding
- Set **decoded_proof_value** to the decoding of **proof_value**
- Make sure that the three first bytes of **decoded_proof_value** are *0xd9*, *0x5d* and *0x03* in that order
- Set **feature_option** to *baseline*
- 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
- Replace the second object in **components** with the result of the algorithm in chapter 3.5.6 of *Data Integrity ECDSA Cryptosuites v1.0*[8], with the second object of **components** as input value
- 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**

D. OpenID Connect for Verifiable Presentations 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

Define vp token and presentation submission, and presentation definition
The flow between these three parties is shown following flow chart

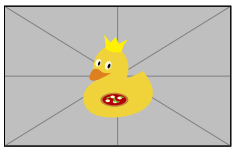

Figure D.1.: Full flowchart of OIDC4VP session

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 D)

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 endpoints then keeps a map 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 D)

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 following information (Step 4 in figure D):

- **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 ...

4. After scanning the QR Code, the holder generates the VP corresponding with the **presentation_definition**. 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** !!! Add nonce to VP (Step 5 in figure D)

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, as the holder doesn't directly notify the verifier that the presentation is done. ... (Step 6 in figure D)

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 D)

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, ... Session fixation??? The endpoint then responds with the corresponding *vp_token* and *presentation_submission* (Step 8 and 9 in figure D)

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 D)

And with that we now know how the whole process for OIDC4VP works!

E. Sandbox

A. First appendix Chapter

Bibliography

[1] 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.

[2] 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.

[3] 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.

[4] Rolf Haenni. Pseudocode algorithms for bbs signatures with link secrets. Unpublished, 2024. Accessed on March 25, 2024.

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

[6] Vasilis Kalos and Greg Bernstein. Bbs per verifier linkability. https://basileioskal.github.io/bbs-per-verifier-id/draft-vasilis-bbs-per-verifier-linkability.html, 2024. Accessed on March 25, 2024.

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

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] 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.