DEPARTMENT OF INFORMATICS

TECHNICAL UNIVERSITY OF MUNICH

Master's Thesis in Robotics, Cognition, Intelligence

# Verification of Digital Credentials Supporting Self-Sovereign Identity for Higher Education Institutions

**Pascal Yannick Herrmann**

DEPARTMENT OF INFORMATICS

TECHNICAL UNIVERSITY OF MUNICH

Master's Thesis in Robotics, Cognition, Intelligence

# Verification of Digital Credentials Supporting Self-Sovereign Identity for Higher Education Institutions

**Verifizierung von digitalen Leistungsnachweisen mit selbstbestimmter Identität für Hochschulen**

| | |
|---|---|
| Author: | Pascal Yannick Herrmann |
| Supervisor: | Prof. Dr. rer. nat. Florian Matthes |
| Advisor: | Felix Hoops, M.Sc. |
| Submission Date: | 15.10.2022 |

I confirm that this master's thesis is my own work and I have documented all sources and material used.

Munich, 15.10.2022 Pascal Yannick Herrmann

# Acknowledgments

First and foremost, I would like to thank my advisor, Felix Hoops, for the excellent support throughout the entire duration of this thesis, for many interesting discussions, valuable feedback, and for providing me with the opportunity to work on this exciting and relevant topic.

I would also like to thank Prof. Dr. Florian Matthes for supervising this thesis and for pushing forward the education and research in the field of blockchain technologies and Self-Sovereign Identity at Technical University of Munich.

At this point, I would also like to take the opportunity to thank Ulrich Gallersdörfer for establishing the BBSE lecture at TUM, which started my interest in this domain, and for being an excellent advisor in the SEBA Lab, which convinced me to also pursue my master's thesis at this chair.

Additionally, I also want to thank my fellow students, Carsten Sehlke and Gopi Mehta, for working with me on the DiBiHo project and making it a success.

Last but not least, I would like to thank Dr. Alexander Lenz for being a fantastic coordinator of the RCI program, for always providing support if needed, and for offering this unique study program which allowed me to pursue my Master's degree with this very special combination of subjects.

# Abstract

Traditionally, higher education credentials, such as Bachelor's or Master's degrees, have been issued by universities in the form of physical paper documents. Because they vary in structure and appearance across institutions worldwide, the verification of the trustworthiness of these documents is challenging. Often, this leads to manual and laborious verification processes for both, credential holders and relying parties. Recently, the World Wide Web Consortium has released a specification for *Verifiable Credentials* supporting the idea of Self-Sovereign Identity. Based on an interoperable, trusted data infrastructure, entities have full control over their digital identities and can issue and receive credentials as cryptographically verifiable claims.

In this thesis, we explore how W3C verifiable credentials can be applied to the context of higher education diplomas and in particular, how they can be verified by relying parties to ensure their contents are trustworthy. To this end, we identify a set of required verification checks, evaluating different aspects such as the credential's integrity or revocation status. In addition, we require the presenter to prove ownership over the submitted credential and confirm their personal identity by providing a cryptographically linked ID credential.

Furthermore, we design a concept for an interoperable, hierarchically structured trusted issuer registry, serving relying parties as a trust anchor for issuer identification. We also provide a proof of concept of how such a registry can be implemented as a smart contract on the Ethereum blockchain, which is maintained by a set of independent entities and governed on-chain by a majority voting based consensus mechanism.

We analyze and evaluate the landscape of available open-source SSI libraries and how they can facilitate the development of a verifier software. Ultimately, we apply the findings of our research questions to implement a prototype for a verification service that relying parties can easily integrate into their systems to verify given academic credentials.

# Contents

# 1. Introduction

## 1.1. Problem Statement and Motivation

Despite our ever-advancing technological progress and the increasing digitalization of our world, many of our most important and personally valuable documents are still only available to us in physical form - such as passports, driver's licenses, or university diplomas. Such *credentials* make up an important aspect of our identity – because they certify us specific entitlements, permissions, or qualifications – expressed as a claim about us made by a recognized issuer, like a government authority or university. The true value of these credentials, however, lies in the fact that we can present them to third parties, allowing us to prove them these specific claims, often enabling us to obtain specific privileges. [1]

Especially university diplomas are needed in this context for various life-defining situations, such as applying for an advanced study program, a job, or a visa for another country. The *Relying Party* – in this case, another university, a company, or a country's immigration department, needs to *verify* these credentials to ensure that the presented diploma is trustworthy. For example, this can include checks whether the diploma has really been issued by the claimed university, that it is a legitimate institution, and that its contents have not been manipulated.

Due to the fact that university degrees are still issued as physical paper documents and vary a lot in structure and appearance, the verification of degree diplomas issued by higher education institutions around the world is extremely challenging. To facilitate this verification process, relying parties often require applicants to present their credentials as notarized copies, signed by the issuing university in a sealed envelope, to ensure that the document has been issued by the claimed university and that the content is unchanged.

This process is not only extremely lengthy due to manual processing, physical mailing, and several involved parties – it is also tedious for students and causes expensive fees charged for shipping and certification. Also for the relying party, this manual verification process is time-consuming and expensive – and, on top of that, does not even provide a guarantee that the documents are indeed authentic since the handling of physical documents will always leave space for fraud. Furthermore, the issuing university often being involved in the verification process constitutes a serious lack of privacy, as it effectively allows the issuer to track which organizations the former student interacts with.

#### Digital Verifiable Credentials and Self-Sovereign Identity

Since nowadays, most forms of application interactions with relying parties exclusively happen digitally, the need for a digital solution to this problem is obvious. While trivial attempts towards digital credentials, such as photographing or scanning of physical degrees,

lack any form of reliable verification features, more advanced approaches, often based on signed PDF files, also could not succeed since they are not universally processable and typically depend on specific technologies, vendors, or software providers.

In the recent few years, the concept of *Self-Sovereign Identity (SSI)* [2] has come up, which promises a solution to this dilemma by providing an alternative approach to digital identity. By leveraging technologies such as asymmetric public-key cryptography, distributed ledger technologies, and cryptographic hash functions, it envisions a digital world in which entities have full control over their identity, without any dependence on a central authority or identity provider and at the same time, providing flexibility and interoperability towards technologies and implementations. The foundation of Self-Sovereign Identity forms the concept of a verifiable data registry, which allows to store and retrieve data based on a trusted and interoperable infrastructure.

In the context of degree credentials, as visualized in figure 1.1, SSI can enable learners and universities to establish their own digital identities and issue or receive diploma credentials that are linked to these identities and cryptographically secured. The learner has full control over their credentials and can freely decide with whom to share them, and relying parties can easily verify the credential without the involvement of the issuer or another additional party.

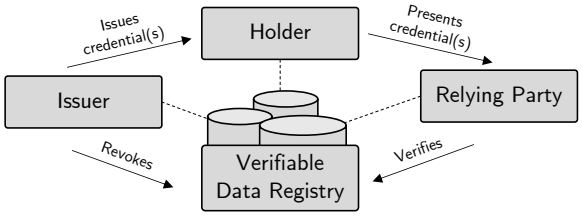


Figure 1.1.: The Primary Roles involved with the Exchange of VCs [2]

In 2022, the World Wide Web Consortium (W3C) proposed a specification to standardize *Verifiable Credentials* and *Decentralized Identity* [1, 3], which opens potential to establish a system of universally processable, digital, verifiable diploma credentials.

### DiBiHo

To explore the application of these W3C standards specifically to the context of higher education diplomas, TUM has launched the project DiBiHo – *Digital Credentials for Higher Education Institutions* (German: **Di**gitale **Bi**ldungsnachweise für **Ho**chschulen) [4]. DiBiHo is a joint project in collaboration with Hasso Plattner Institute (HPI) and the German Academic Exchange Service (DAAD). The project is funded by the German Federal Ministry of Education and Research (BMBF) with a total funding volume of € 1.94 Mio., and a duration of roughly two years, beginning in late 2020 and going until the end of 2022. The goal of the project is to "explore a trusted, distributed, and interoperable infrastructure standard for issuing, storing, presenting, and verifying digital academic credentials in a national and international context".

With these ambitions, TUM and its project partners are not alone but corporate closely with the *Digital Credentials Consortium (DCC)* - a consortium founded in 2018 by 12 internationally

leading universities, including TUM and HPI, whose mission is to build a global infrastructure for the issuance and exchange of digital academic credentials. [5] In 2020, the DCC published the white paper "Building the digital credential infrastructure for the future" [6], which serves as the basis and starting point for the work in DiBiHo.

DiBiHo began with an extensive phase of requirements engineering, which was accompanied by international stakeholder dialogues, expert interviews, and user workshops. As an important intermediate result of the project, a catalog of 46 functional and ten non-functional requirements was published. [7] DiBiHo's final and currently ongoing project phase consists of the development of a *Proof of Concept*. This primarily includes the implementation of a prototype system that supports the issuance of education achievement credentials, the transfer and storage in a wallet app, the sharing with relying parties, and the verification of credentials. [4] The three project partners represent different use cases in this system:

**HPI** is primarily involved in the issuer's side and targets to equip *OpenHPI*, their very own platform for massive open online courses, with a feature for learners to download their MOOC achievements as verifiable credentials.

**TUM** aims to issue diploma certificates for Bachelor's and Master's degrees as VCs by integrating this functionality directly into their student information system *TUM Online*. Furthermore, as one of Germany's largest universities, TUM also receives massive amounts of applications for their study programs. Therefore, TUM also represents the side of a relying party that verifies credentials that have been presented by learners applying for studying at TUM.

**DAAD** exclusively represents a relying party, which receives and verifies applications from students from Germany and abroad, primarily for scholarships and exchange programs.

The recently published thesis "Transforming a Digital University Degree Issuance Process Towards Self-Sovereign Identity" [8] covers the *issuance* process at TUM – in particular, the integration into TUM Online and the transfer of diploma credentials into the student's wallet.

In our thesis, we will focus on the *verification* aspect of the project and discuss and analyze how relying parties can verify education achievement credentials that have been issued from HPI's and TUM's prototype integrations. The goal of this thesis is the development of a prototype for a verification service (also called *Relying Party Service*) that addresses the five functional DiBiHo requirements specific to the relying party's demands for a verification component: [7]

**DiBiHo-13** As a relying party, I want to verify a credential so that I can be sure that the **contents of a presented credential are trustworthy**.

**DiBiHo-14** As a relying party, I want to **authenticate the issuer** of a presented digital credential so that I can be sure of the identity of the issuer, as well as their authorization to issue and, therefore, the validity of the credential.

**DiBiHo-55** As a relying party, I want to verify a credential **without the direct involvement of the issuer** so that I am not dependent on the issuer's infrastructure, especially when the issuing party does not exist anymore.

**DiBiHo-24** As a relying party, I want to **authenticate a learner** so that I can be sure that the subject of the presented digital credential is the same (or authorized) as the sender of the digital credential.

**DiBiHo-25** As a relying party, I want to **integrate the information inside a digital credential in my digital processes** so that I can use it in my further workflows.

## 1.2. Research Questions

To this end, we formulate three research questions that will form the foundation of this thesis.

As part of our first research question, we consider the verification process as a whole and want to answer what in particular must happen for a credential to be considered valid:

**RQ1**: What is an effective validity check for verifiable credentials?

- What checks must be performed, and in what order?
- What data must be fetched?
- What error states are possible? How should they be communicated?

After having defined the individual checks that need to happen as part of the verification, in our second research question, we will shift the perspective towards the development of our prototype and analyze if and how we can leverage existing libraries in the SSI domain to support our implementation:

**RQ2**: What SSI libraries shall be used in the implementation of the verification service?

- Which criteria must SSI libraries fulfill to be well-suited for this use case?
- What SSI libraries exist, and how do they fulfill these criteria?

For our last research question, we will focus on one specific aspect of the verification process that we consider a crucial component of the overall system – the authentication of the credential issuer. To this end, the DCC white paper introduces a trusted issuer registry that verifiers can rely on as a trust anchor to determine if a given credential issuer is a legitimate institution [6]. However, since the DCC has not yet designed a concept for this component, we dedicate it our third and final research question:

**RQ3**: How can a trusted issuer registry be designed and implemented to serve as a trust anchor for verifying issuers of digital credentials?

- Which type of infrastructure is suited?

- What information must be stored?
- How can new issuers be added while maintaining a high level of trust?
- How can the trusted issuer registry be implemented?

The ultimate **Goal of this Thesis** is to apply the findings from the above research questions and develop a proof of concept for a verification service that any relying party can integrate and use to verify if arbitrary credentials issued from TUM and OpenHPI are valid and can be considered trustworthy.

To approach this project, we followed a methodology that is largely resembled by elements of Scrum [9] and the Design Science Research Process model by [10]. The beginning phase of this thesis was mainly shaped by literature research to identify the problems and objectives that we have summarized in this section. From there, we followed an iterative approach centered around the design and development of the verifier service. Here, we aimed to have a functioning version of the prototype at any given time, in order to demonstrate its capabilities and evaluate how it fulfills our requirements. To this end, we conducted weekly scrum-inspired meetings within the DiBiHo developer team. This iterative approach allowed us to react to changing requirements and priorities and re-iterate accordingly.

## 1.3. Thesis Structure

In the following chapter 2, we will introduce some technical and theoretical background concepts that are required for an understanding of this thesis. We will then, in chapter 3, provide a glimpse at some other existing solutions for digital, verifiable credentials. In chapter 4, we will discuss what checks a credential needs to pass to be considered valid, which corresponds to our first research question. This chapter will form the foundation for our implementation of the verifier prototype and can therefore be considered the analysis and design phase of the thesis. After having described what our verifier has to achieve, in chapter 5, we will look at the question if and how we can leverage existing SSI libraries to leverage our implementation. To this end, we will analyze what requirements an SSI library should fulfill and evaluate different available candidate frameworks in how well they meet these criteria, corresponding to our research question 2. Then, in chapter 6, we will shift the focus to the trusted issuer registry to address our research question 3. We will explain the problem this registry should solve, analyze and explain different design choices, and present our implementation for an MVP solution. In chapter 7, we will look at the verification service in a broader context and discuss how relying parties can integrate it into their existing systems. We will also address some aspects of the development process and present our concept for a user interface to communicate the result of the credential verification. To wrap the thesis up, we will evaluate our solution in chapter 8 and address directions for future work before we come to the final conclusion in chapter 9.

# 2. Background

## 2.1. Cryptographic Basics

This thesis heavily relies on cryptographic concepts like hash functions, digital signatures, or public key cryptography. In the following, we will shortly introduce these concepts to allow readers who are unfamiliar with these topics to follow the thesis.

### Symmetric Cryptography

In symmetric cryptography, a *single key* or password is used for both encrypting data into ciphertext and decrypting it back into its original form. Consequently, if symmetric encryption is applied to messages sent between two parties, both of them need access to the same shared key. One of the most widely used symmetric encryption algorithms is *Advanced Encryption Standard (AES)*, which uses keys with a length of 128, 196, or 256 bits. [2, 11]

### Asymmetric Cryptography

Asymmetric cryptography, which is also called *Public Key Cryptography*, is based on pairs of two keys that are mathematically related: A *Private Key*, which should be kept secret by the owner, and a *Public Key*, which can be openly shared with everyone. If one key was used to encrypt a message, only the other key can be used to decrypt the resulting ciphertext. For the above example of secure message sharing, a sender can use the public key of the receiver to encrypt the message to be sent, resulting in a ciphertext message that can only be decrypted by the designated receiver's private key. Commonly used asymmetric cryptography algorithms include RSA or algorithms based on *Elliptic Curve Cryptography* (ECC). [2, 11]

### Cryptographic Hash Functions

Hash functions transform data of arbitrary input length into a fixed-sized output called the data's *hash value*. Additionally, they are deterministic, meaning that the same input will always hash to the same output. *Cryptographic* hash functions have additional important properties: They are *irreversible*, which means that given a hash value, it is computationally infeasible to retrieve the input of the hash function that led to the given hash. Additionally, they are *collision-resistant*, which means that it is also computationally infeasible to find multiple inputs that hash to the same value. Nowadays, especially the family of SHA algorithms (Secure Hash Algorithm) is widely used. Specifically, SHA-256 [12] is relevant for this thesis, which transforms inputs to hash values of 256 bits. [2, 11]

### Digital Signatures

A digital signature algorithm includes two functions, the *Signature Creation* which allows a signer to generate a cryptographic signature for an arbitrary message or data input, and the *Signature Verification*, which enables another party to verify that a certain entity has signed the data (*Message Authenticity*) and that the data has not been modified after signing (*Message Integrity*). Digital signatures rely on public key cryptography and assume that each signer owns a secret private key with the corresponding public key.

The **creation of a signature** for an arbitrary message typically consists of two steps: First, the signer hashes the message to obtain a condensed, fixed-sized representation of the data input, which is also called a "message digest". Then, the signer uses their private key to generate a cryptographic signature, similar to asymmetric encryption. The resulting signature is then shared together with the message.

For the **verification of a signature**, a verifier needs the original message itself, the actual signature, and the public key of the claimed signer. The verifier processes the data the exact same way as it was done before signing, e.g., by applying the same hash function and then uses the public key of the claimed signer to verify if the signature matches the data. If so, it proves that the owner of the corresponding private key has indeed signed the message and that it has not been altered. [2, 11, 13]

For specific types of data, an additional transformation called **canonicalization** may be performed before the hashing of the message. [14] This is often needed for data that can have multiple representations for the same information. An example of this are strings representing JSON objects, where the individual properties of the object can be in arbitrary order but still represent the same object. The canonicalization algorithm brings the data into one universal form to ensure that the signature will not be affected if the data was formatted or arranged without changing the information (like switching the order of JSON properties). A trivial canonicalization algorithm, in this case, could simply order all object properties alphabetically. Canonicalization is used for *JSON-LD proofs* [15], i.e., signatures that sign JSON-LD data, which we will cover in the next section.

There are many types of different signature algorithms. In the last years, the trend has mostly been going from RSA towards elliptic curve cryptography (ECC), particularly ECDSA (*Elliptic Curve Digital Signature Algorithm*) and EdDSA (*Edwards-curve Digital Signature Algorithm (EdDSA)* [16] - also based on ECC). Most often, ECDSA is used with the elliptic curve *secp256k1* (in short: ECDSA-secp256k1 [17]), while EdDSA is frequently used with *Curve25519* (in short: Ed25519 [16]). Among these two, Ed25519 is often preferred due to its slightly shorter public key size, faster performance, and increased security [11].

It is important that the signer and the verifier use the exact same algorithms for data transformation (canonicalization and hashing) and signature creation/verification. Therefore, these cryptographic primitives are typically bundled together into **Signature Suites**, allowing developers to integrate them into their applications. [18, 19, 20]


Figure 2.1.: Example of a JSON Web Token. (Based on [21])

## 2.2. Web Standards

### 2.2.1. JSON Web Token (JWT)

JSON Web Token (JWT) [22] is a standard for compactly representing claims that can be shared between different parties and verified for integrity. A JWT is typically shared in string format, consisting of three components that are separated by a dot character in the format `header.payload.signature` (as visualized in figure 2.1).

The **Header** is a JSON object specifying the algorithm that was used to create the JWT signature. A commonly used algorithm is *HMAC SHA256* (short: `HS256`) [23], which uses a shared secret in combination with a hash function (e.g., SHA256). Public/private key algorithms, like RSA or ECDSA, are also often used to sign JWTs.

The **Payload** represents a claim asserted by the signer. It can contain arbitrary JSON data, but has a set of reserved properties, such as `iss` (issuer), `sub` (subject), `aud` (audience), `ita` (timestamp of issuance), and `exp` (expiration time). [21]

Both the header and payload get base64 URL encoded and concatenated with the `.` separator. The **Signature** is then created for this input with the specified algorithm and also appended to the string.

The resulting JWT is compact in size and URL safe. This makes JWTs a popular choice for authentication purposes – especially websites often issue users a bearer token in form of a JWT after a successful login. In principle, JWTs can be seen as a simplified version of VCs, which we will cover in the next sections. [22, 24]

### 2.2.2. JSON for Linked Data (JSON-LD)

For many years, JSON has been the default format for serializing and exchanging data on the web. It is simple to generate and parse across all programming languages and, at the same time, human-readable. [25] However, when processing data from different sources, e.g., data that was retrieved by following hyperlinks in a given JSON document, the interpretation of these foreign JSON documents becomes more difficult: Documents from different data

```
/* Some JSON-LD File */
{
  "@context": "https://json-ld.org/contexts/person.jsonld",
  "name": "John Doe",
  "homepage": "http://john-doe.com/",
}
```

Listing 2.1: Example of a JSON-LD Document

```
/* https://json-ld.org/contexts/person.jsonld */
{
  "@context": {
    "name": "http://schema.org/name",
    "homepage": {
      "@id": "http://schema.org/url",
      "@type": "@id"
    }
  }
}
```

Listing 2.2: Example of a Context File

sources usually vary in structure and use different property keys, leading to conflicts and inability of automatic processing.

To this end, W3C has proposed *JSON-LD* [25], which is a JSON-based format to serialize linked data. It provides a concept to make documents universally machine-readable by uniquely defining how each of its properties shall be interpreted. JSON-LD is fully backwards-compatible, as it is based on the regular JSON format but extended with a set of reserved key properties. Most notably, JSON-LD introduces a notion of `@context`. The context defines how values of the JSON document shall be interpreted by mapping their keys to a unique type identifier. This enables interoperability and allows processing linked data from various sources, for example, by following links across the web from one document to the other, resulting in graphs of machine-processable data.

Listing 2.1 shows a minimal example of a JSON-LD document and its referenced context file in listing 2.2. The latter specifies that all values of properties with the key `name` are of a type with the unique identifier `http://schema.org/name`. It also defines that values of properties of the key `homepage` are from the unique type `http://schema.org/url` and represent an identifier itself that can be resolved. [25]

## 2.3. Decentralized Identifiers (DIDs)

*Decentralized Identifiers* [3] – in short, DIDs – form a core concept of Self-Sovereign Identity. They uniquely identify arbitrary subjects – such as organizations, persons, or items. In contrast to typical, centralized identifiers – like e-mail addresses, phone numbers, user names, or social security numbers, DIDs are designed to be decoupled from central authorities or identity providers. The owner of a DID alone controls it, independently of any authority or organization. DIDs can be associated with cryptographic key material. This allows DID controllers to authenticate themselves as the owner of a DID when interacting with different parties – in a privacy-preserving way, as no intermediate entity is involved. In 2022, DIDs have been standardized by the World Wide Web Consortium, which opens the potential to establish DIDs as a new form of digital identification.

On a technical level, DIDs are Uniform Resource Identifiers (URIs) that consist of a standardized format as visualized below:

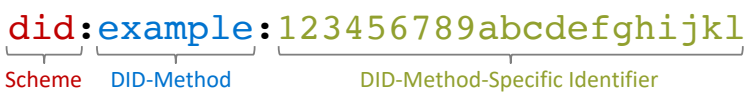


Figure 2.2.: Overview of the DID Format

Each DID refers to a so-called *DID Document*, which is a JSON-LD file containing information like public key material to authenticate the DID's controller or service endpoints that allow third parties to interact with the DID subject. The *Controller* of a DID is the entity that has the capability to make changes to the DID document, such as updating or deleting it. The *DID Subject* is the entity that a DID document refers to. In many cases, the subject and controller are the same entity, which is, however, no requirement. The mapping between a DID and its corresponding DID document is stored in a *DID Registry*. This registry is often a decentralized data storage, for example, a blockchain like Ethereum or Bitcoin. However, DIDs are completely flexible in supporting arbitrary infrastructures for DID document storage, which is one of the key ideas and benefits of decentralized identity. To account for this great variety of different storage options, a DID identifier includes the *DID Method*, which specifies the protocol of how a DID document can be retrieved or updated. The process of retrieving a DID document for a given DID from the registry in which it is stored is called *DID Resolving*. An overview of the DID architecture and its individual components is shown in figure 2.3.

### DID Document

An exemplary DID document is shown in listing 2.3. It contains the following properties: [3]

**ID:** The `id` contains the DID under which the document is resolvable. It refers to the DID subject.

**Verification Method:** The `verificationMethod` property contains one or multiple public keys that can be used to authenticate or authorize transactions with the DID subject. Typically, each key consists of the following properties:

- An `id`, which uniquely identifies the public key by concatenating the document's DID with a URL fragment that identifies the key resource within the DID document. The id will be included as metadata in any created proofs so that the verifier can retrieve the key material from the signer's DID document.
- A `controller`, which typically corresponds to the DID of the document.
- A `type` that defines what sort of key is used.
- And the actual public key material, which varies depending on the key type (in this example, it includes the property `publicKeyMultibase`).

**Verification Relationships:** The `assertionMethod` and `authentication` properties define for which purposes a key can be used. While the former is used to specify how a DID subject is expected to sign claims, the latter describes how the DID subject can be authenticated, e.g., for challenge-response protocols.

```
{
  "@context": "https://www.w3.org/ns/did/v1",
  "id": "did:web:dibiho.org:TUM",
  "verificationMethod": [
    {
      "id": "did:web:dibiho.org:TUM#key",
      "controller": "did:web:dibiho.org:TUM",
      "type": "Ed25519VerificationKey2020",
      "publicKeyMultibase": "z6Mktt7yUbwJqvGiHcnaw7aRs
      ↪ XvTFgMD2MbH6c3WcXEg36LA"
    }
  ],
  "authentication": [
    "did:web:dibiho.org:TUM#key"
  ],
  "assertionMethod": [
    "did:web:dibiho.org:TUM#key"
  ],
  "service": [{
    "id": "did:web:dibiho.org:TUM#StatusList:1",
    "type": "StatusList",
    "serviceEndpoint": "status:web:https://dibiho.org/
      ↪ credentials/status/1.json"
  }]
}
```

Listing 2.3: Example of a DID Document

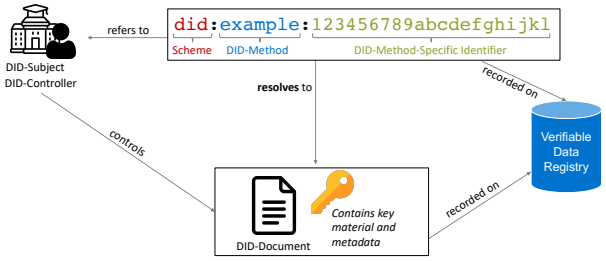


Figure 2.3.: Overview of the DID Architecture and the Relationship of the Basic Components. (Based on [3])

**Services:** The `services` property contains a list of services, defined by an `id`, `type`, and `serviceEndpoint`, that third parties can use to interact with the DID subject.

### DID Methods

A DID method defines how a specific DID resolving and storage scheme is implemented. Typically, it is specific to a certain type of verifiable data registry in which the DID document is stored. In particular, a DID method must define how the four CRUD operations are implemented for DID documents of this method scheme: **C**reate, **R**ead (also called *DID Resolution*), **U**pdate, **D**elete (also called *DID Revocation*). Currently, there exist specifications for more than 140 different DID methods. In the following, we will present four DID methods that we consider important within the DiBiHo project. [3, 26]

## 2.3.1. did:key

**did:key** [27] is the simplest possible DID method, as it does not involve a DID registry at all. Instead, the DID identifier directly contains its public key as well as a short prefix value which indicates the type of the key:

```
did:key:z6MkhHjQRt5HRTpGD7kwtZ8VZeydDHM5j3U7WaFHu5PiQ9uX    // z6Mk...: Ed25519 key
did:key:zQ3shokFTS3brHcDQrn82RUDfCZESWL1ZdCEJwekUDPQiYBme   // zQ3s...: Secp256k1 key
```

To *resolve* the DID document from a `did:key` identifier, the resolver simply "constructs" a DID document that includes the given public key as one and only verification method. The DID *creation* for this method is also trivial and just consists of the generation of a public-private keypair, with the former one being included into the DID identifier.

The advantages of did:key are obvious: it does not require a connection to a DID registry for both creation and resolving and is therefore fast, free, and easy to set up. However, the lack of a DID registry also leads to serious drawbacks – since nothing is stored, DID documents cannot be updated. Not only does this pose a major security risk because key rotation is not possible and DIDs cannot be revoked – but it also severely limits the usefulness of this method since many of the benefits that DIDs offer – like adding a service or additional keys to a DID document – are not possible. Therefore, while did:key is a convenient DID method for testing and development purposes, it is usually not recommended to use it for any production purposes.

### 2.3.2. did:web

**did:web** [28] is another very simple type of DID method. Here, the DID includes a Fully Qualified Domain Name (FQDN) with an optional path that redirects to a web server where the DID document is stored as a simple `did.json` file – secured by a TLS/SSL certificate, like a typical website.

```
did:web:dibiho.org          // DID Doc @ https://dibiho.org/.well-known/did.json
did:web:dibiho.org:TUM      // DID Doc @ https://dibiho.org/TUM/did.json
```

In order to resolve a DID of this method, the resolver simply needs to map the DID to the corresponding web URI and perform an HTTP GET request to obtain the DID document from the server. The `did.json` file representing the DID document can either be located in the `.well-known/` directory in the root of the domain – or at an arbitrary path on the server, in which case each "`/`" character in the path will be converted to a "`:`" character in the DID.

To create a DID for this method, it is necessary to register a domain at the Domain Name System (DNS) and forward it to a server that is controlled by the creator. There, they can create the `did.json` file and update it as needed. For revocation, the file can be deleted or the domain unregistered.

This DID method has several very significant benefits: DIDs for this method are very easy to set up, as the process is the same as hosting any other arbitrary file on the web. It is cheap or even free, assuming that the controller already operates a website and a domain. In contrast to most other DID methods, the identifier itself is human-readable, and it is easy for anyone to associate the DID with the organization behind it. And lastly, the DID document can be easily updated with arbitrary data for free.

However, did:web also comes with serious drawbacks, mostly resulting from the fact that the storage registry is not decentralized but rather a conventional web server. Since the web server can be down at any time, or a domain may expire, the DID may not always be reachable or can even become permanently inaccessible. For this reason, it is usually not recommended to use did:web for production applications; however, until DIDs will be more widely adopted, did:web may be a good transitional solution.

### 2.3.3. did:ethr

**did:ethr** [29] is - opposed to the two previously discussed DID methods - a fully decentral DID method that lives on the Ethereum blockchain. In did:ethr, each DID identifier is tied to an Ethereum account and its corresponding secp256k1 key pair. To obtain the identifier, the public key pair is appended to the method-specific prefix:

```
did:ethr:0x02f7fbd7af3819b0e67fbd0a12a22cbb27cbb75201cae6d853f1b4d02c8e0b59c5           // Mainnet
did:ethr:goerli:0x02f7fbd7af3819b0e67fbd0a12a22cbb27cbb75201cae6d853f1b4d02c8e0b59c5    // Goerli Testnet
```

The creation of a new DID for this method is therefore as easy as generating a regular Ethereum account. By default, all DIDs resolve to their public key – hence, no transaction is required to create a new DID, which brings the huge advantage that the creation of did:ethr identities is free of gas.

In addition, did:ethr also lets controllers make changes to their DID document. To this end, the controller can send a transaction to the *Ethereum DID Registry* (proposed in ERC-1056 [30]), a smart contract of which a single instance is deployed to each Ethereum network, and that serves as a registry to record all changes that have been applied to did:ethr documents.

Instead of storing the DID document like a file in the registry, the registry smart contract emits an event for every transaction changing a DID document. For the resolution of a DID, the resolver queries the registry smart contract via read-only operations for all events associated with the given DID and then aggregates the event data off-chain to dynamically construct the current version of the DID document.

To conclude, since did:ethr allows free and simple DID creation and is a fully decentralized DID method based on the popular Ethereum blockchain, it is a highly recommended DID method for many use cases as it leverages all the benefits of SII. The only downside is that changes to the DID document require transactions to the blockchain which leads to gas costs and can get expensive if frequent updates to the DID document are needed.

### 2.3.4. did:iota

**did:iota** [31] is a DID method for the IOTA distributed ledger. Compared to other distributed ledger technologies, like Ethereum, that are based on a blockchain, IOTA stores its transactions on a directed acyclic graph data structure - the *IOTA Tangle*. Another important characteristic of IOTA is the lack of miners, which is possible because each IOTA transaction validates at least two previous transactions – and therefore, eliminates the need for transaction fees.

```
did:iota:main:FFkVSV9FaXXKeW3PYzDJatuWCLu1RX5GsBiNQwRoSGY5 // stored on IOTA Mainnet (explicitly)
did:iota:CJ4unDbviP4JvG7zooPfq4j2zDB8zksDYwKM7fn3hY4i      // stored on IOTA Mainnet (implicitly)
```

Internally in did:iota, DID documents are represented as a chain of linked *DID messages* that are published to the IOTA tangle. The tangle has "no understanding of DID messages and acts purely as an immutable database". Consequently, all verification of the DID messages and the resulting DID document must happen on the resolver's side. [31]

```
{
  "id": "1234567890",
  "type": "<message-type-uri>",
  "from": "did:example:alice",
  "to": ["did:example:bob"],
  "created_time": 1665942024,
  "expires_time": 1666028424,
  "body": {
    "message_type_specific_attribute": "and its value",
    "another_attribute": "and its value"
  }
}
```

Listing 2.4: DIDComm Plaintext Message Structure

```
{ < DID Document Properties >,
  "service": [{
     "id": "did:example:123456789abcdefghi#didcomm-1",
     "type": "DIDCommMessaging",
     "serviceEndpoint": {
       "uri": ["https://example.com/endpoint"],
       "routingKeys": ["did:example:somemediator#somekey"]
     }
  }]
}
```

Listing 2.5: DIDComm Service Endpoint

To update a DID document, a DID controller can publish a message that defines the new state of the DID document to the tangle index corresponding to their DID identifier and sign the message with the public key contained in the current version of the DID document.

To resolve a DID, the resolver queries all DID messages from the tangle index corresponding to the DID. It then orders the messages based on the linkages to previous messages and verifies their signatures to dynamically produce the most current state of the DID document.

With its free transactions, did:iota can be an interesting alternative to other distributed ledger technologies, where updating DID documents is often associated with considerable costs. However, it should be noted that most IOTA nodes prune zero-balance addresses after a certain amount of time for performance reasons. Therefore, it is important that the resolver is connected to a "permanode" that keeps all the previous messages. [32, 33]

### 2.3.5. DIDComm

DIDComm [34] is a routing and messaging protocol built on top of DIDs that allows for communication between different entities that control a DID. A DIDComm message consists of a structured plaintext message, as shown in listing 2.4. To send a message, the sender first resolves the receiver's DID document containing a `DIDCommMessaging` service endpoint specifying where messages should be delivered (see listing 2.5), and their public key material, which the sender uses to encrypt the plaintext message. The sender also adds authentication by signing the message with their own private key. A software agent then arranges the delivery accordingly.

## 2.4. Verifiable Credentials (VCs) and Verifiable Presentations (VPs)

#### Verifiable Credentials (VCs)

In 2022, the World Wide Web Consortium proposed a standard to establish a digital equivalent to physical credentials – *Verifiable Credentials (VCs)* [1]. Similar to physical credentials that contain information identifying the subject (like its name or a photo) and the issuer, as well as specific claims asserted by the issuer about the subject and some security features to validate the credential, W3C Verifiable Credentials can represent all this information in digital

```
{
  "@context": [
    "https://www.w3.org/2018/credentials/v1",
    "https://dibiho.org/contexts/CredentialDemo.json",
    "https://w3id.org/security/suites/ed25519-2020/v1"
  ],
  "id": "https://dibiho.org/credentials/1311",
  "type": [
    "VerifiableCredential",
    "TestDiplomaCredential"
  ],
  "issuer": {
    "name": "Technical University of Munich",
    "logoUrl": "https://dibiho.org/TUM/logo.svg",
    "id": "did:web:dibiho.org:TUM"
  },
  "credentialStatus": { ... },
  "expirationDate": "2030-01-01T00:00:00Z",
  "credentialSubject": {
    "id": "did:web:dibiho.org:Lucas.Learner",
    "degree": {
      "title": "Master of Science (M.Sc.)",
      "major": "Informatics",
      "grade": "1.3 (very good)",
      "graduationDate": "2022-10-15"
    }
  },
  "issuanceDate": "2022-09-15T16:29:18Z",
  "proof": {
    "type": "Ed25519Signature2020",
    "created": "2022-09-15T16:29:18Z",
    "verificationMethod": "did:web:dibiho.org:TUM#key",
    "proofPurpose": "assertionMethod",
    "proofValue": "zEkhRLLf....te4WMTvdXgQeem"
  }
}
```

Listing 2.6: Example of a Verifiable Credential

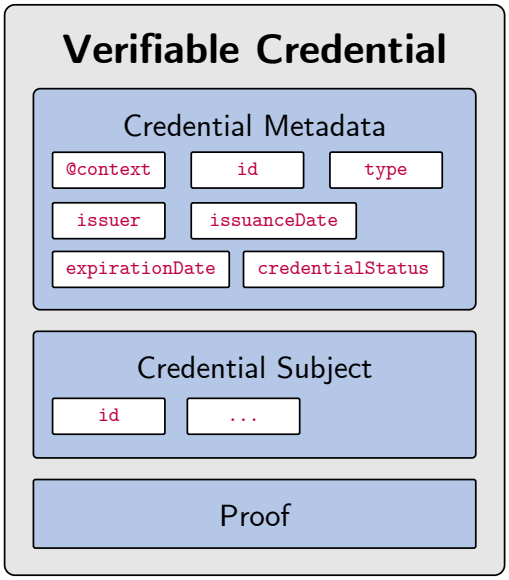


Figure 2.4.: Structural Overview of a VC. (Based on [1, 8])

form. These digital counterparts do not only bring the advantage that they are machine-readable and can be presented digitally, but by leveraging cryptographic features, they are also tamper-proof, privacy-preserving, and more trustworthy than traditional credentials. [1]

On a more technical level, verifiable credentials are JSON-LD documents that contain three components: A *Claim* that an issuer asserts about the credential subject, some *Metadata* that describes the credential and identifies the issuer, and a *Proof* that contains a signature from the issuer, that a verifier can use to validate the integrity and authenticity of the credential. An example of a VC is shown in listing 2.6. VCs typically contain the following properties: [1]

**JSON-LD Context:** A `@context` property, like every JSON-LD document. All VCs must include the W3C VC *Base Context URI* `https://www.w3.org/2018/credentials/v1`.

**Credential ID:** The `id` of the credential should uniquely identify the VC.

**Credential Type:** The `type` of the credential allows a processor to determine if they can handle a specific credential. It can contain a list of several values but must always include the generic type `VerifiableCredential`.

**Issuer:** The `issuer` property contains the issuer's DID and optionally additional information about the issuer. The DID must resolve to the issuer's DID document which stores their public key material needed for verification.

**Issuance Date:** The `issuanceDate` specifies the timestamp when a VC starts to become valid. A VC can also contain further datetime-valued properties, such as an `expirationDate`, which we will explain in more detail in section 4.5.

**Credential Subject:** The `credentialSubject` must be specified through a DID. Typically, it is a structured object that also contains further information about the credential and its subject.

**Credential Status:** The optional `credentialStatus` property may contain information that the verifier can use to check if the credential has been revoked.

**Proof:** The `proof` property contains the signature of the signer, along with further information that is needed for the verification, such as the signature type and the DID URL to the public key.

#### Verifiable Presentations (VPs)

Whenever a credential holder presents a VC to a relying party, such as a student applying to a company, they would typically not just provide the raw credential but rather share it in the form of a *Verifiable Presentation (VP)* [1]. A VP can be seen as a digital package containing one or multiple verifiable credentials and a signature from the presenter, which proves their ownership over the presented credential(s). It also contains additional properties, like a challenge parameter, that is typically provided by the relying party to prevent replay attacks. Figure 2.5 shows an overview of the structure of a VP, and listing 2.7 provides a concrete example. In general, VPs consist of the following properties: [1]

**JSON-LD Context:** The `@context` defining the JSON-LD contexts. Again, the W3C VC *Base Context URI* must be included here.

**Type:** The `type` must include `VerifiablePresentation`.

**Presented Credential(s):** The `verifiableCredential` property contains one or multiple verifiable credentials.

**Proof including Challenge:** The `proof` property contains a proof signed by the holder. It additionally contains a unique `challenge` parameter that is prescribed by the relying party and must be signed by the holder in order to prevent replay attacks.

## 2.5. DiBiHo

After having introduced the theoretical and technical basics of this thesis, we will now look closer at the technical details of the DiBiHo prototype implementation. Particularly, we will focus on the implementation for Technical University of Munich.

```
{
  "@context": [
    "https://www.w3.org/2018/credentials/v1",
    "https://w3id.org/security/suites/ed25519-2020/v1"
  ],
  "type": [
    "VerifiablePresentation"
  ],
  "verifiableCredential": [
    { < Credential 1 > },
    { < Credential 2 > }
  ],
  "proof": {
    "type": "Ed25519Signature2020",
    "created": "2022-09-15T16:29:19Z",
    "verificationMethod": "did:web:dibiho.org:Lucas.Learner#key2",
    "proofPurpose": "authentication",
    "challenge": "bc22571a899988abaf1233b4ad1a7581",
    "proofValue": "z33gyHptuDqerNWCizBEnBY...yr3K74wZ3Ak2Q8QL"
  }
}
```

Listing 2.7: Example of a Verifiable Presentation

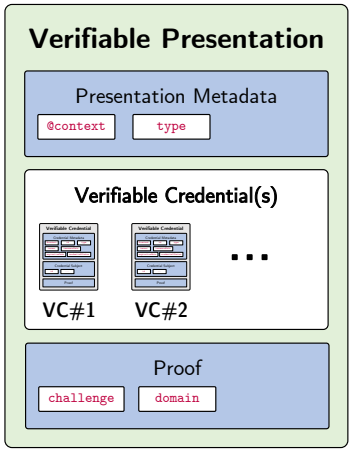


Figure 2.5.: Structural Overview of a VP. (Based on [1, 8])

### DiBiHo System Architecture

Since TUM not only issues degrees to their students but also verifies degrees from applicants, TUM acts both as an issuer and as a relying party. An overview of TUM's part of DiBiHo's system architecture is shown in figure 2.6. It consists of the following components:

The core **Issuer Service** is developed primarily by HPI. It is provided as open source software and can be installed by any institution that desires to issue VCs, like TUM. It provides an internal HTTP API that takes requests containing arbitrary data, which the issuer service signs and returns in form of a valid VC. Consequently, the issuer service has access to the private keys of the issuing organization. It is only accessible in a secured network by authorized applications. [35] Besides issuing, it also manages revocation of credentials and updates the credential status in a verifiable data registry accordingly.

The **TUM Online Bridge** connects the TUM Online student information system with the issuer service. It consists of an issuance watchdog service that periodically checks for new degrees in TUM Online that have been entered by the central examination office. Whenever new degrees are present, it queries the issuer service to generate VCs and stores them in a database. It also provides a credential collection UI plugin for TUM Online that students (i.e., *holders*) can use to transfer their VCs into their wallet. [8]

The **Wallet** stores all the credentials of a user. It allows them to manage their DIDs and securely stores the private keys. In DiBiHo, we primarily assume that learners use the DCC's *Leaner Credential Wallet*, which we will introduce in section 3.1.1.

The core **Verifier Service**, which we develop as part of this thesis, provides functionality to verify any given credentials issued by DiBiHo. It queries several verifiable data registries to resolve the issuer's and holder's DIDs, verify the credential status, and authenticate the issuer. We will cover the exact procedures for these checks in the following chapters. Similar to the core issuer service, the core verifier service will be provided as open source software

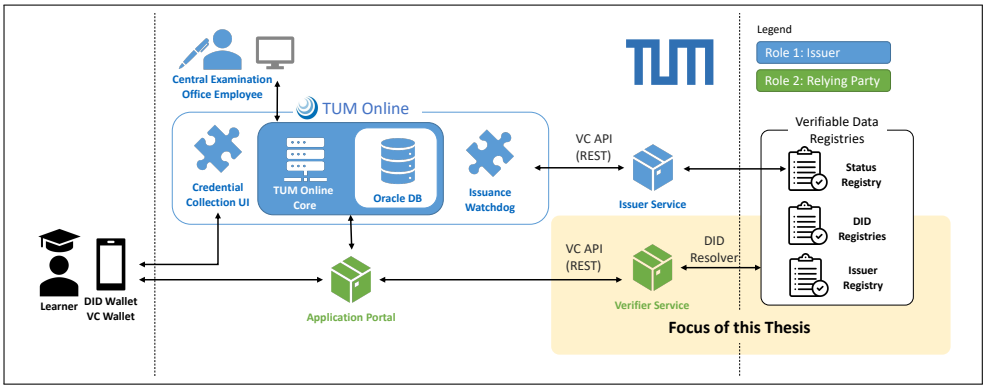


Figure 2.6.: Overview of TUM's Part of DiBiHo's System Architecture

that relying parties who desire to validate VCs can download and integrate into their systems through a provided REST API. It is important to note that the verifier service primarily acts on an "envelope level" and checks whether a credential is *valid*. It does not perform any business logic on the specific contents of a VC to decide whether to accept or reject a VC (e.g., rejecting students with too bad grades). Such rules should be implemented by the relying party's application after querying the verifier service:

To this end, TUM's vision is to integrate the verifier service into the **Application Portal** so that applicants can simply submit their VCs from their wallet, which the application portal validates, by querying the verifier service. If valid, the portal can automatically check if a student fulfills the required educational background to start a degree. This integration, however, is out of the scope of this thesis, as we will focus on the development of a generic core verifier service that can be integrated by arbitrary relying parties, such as TUM, DAAD, or other universities or organizations.

### Credential Data Format

The actual diploma content (like grade, major, degree, etc.) of VCs issued by TUM is represented in the ELMO format. ELMO is an exchange format specifically for learner information, like degrees, diploma supplements, or transcripts of records. It is used by many higher education institutions in Europe. The credential type issued by TUM is therefore named `ElmoDiplomaCredential`. [8, 36]

The exact format of the credential contents, however, is not relevant for this thesis, since as discussed, the verification happens on the credential-level and acts on the envelope layer which is agnostic to its contents.

# 3. Related Work

In the following sections, we present related work and different approaches towards digital verifiable credentials that have applications in the higher education context.

## 3.1. Digital Credentials Consortium (DCC)

The *Digital Credentials Consortium (DCC)* was founded in 2018 by twelve internationally leading universities from North America and Europe, including TU Munich and Hasso Plattner Institute. Its mission is to "create a trusted, distributed, and shared infrastructure that will become the standard for issuing, storing, displaying, and verifying academic credentials, digitally". [5] In 2020, the DCC released the White Paper *Building the digital credential infrastructure for the future* [6], describing the requirements, features, and components of such a system, which serves as a foundation for several projects developed by DCC as well as DiBiHo. Primarily driven by MIT, the DCC currently develops two major projects, called *Learner Credential Wallet App* and *Verifier Plus*, which we will introduce below.

### 3.1.1. Learner Credential Wallet

The *Learner Credential Wallet (LCW)* [37, 38] is a cross-platform mobile wallet specifically designed for storing and sharing digital education credentials conforming to the W3C standard. It is developed in React Native and available as open source project on GitHub under the MIT License. While still under active development, a pilot version of the app is already publicly available in both the App Store and Google Play. Due to DiBiHo's close cooperation with the DCC, allowing us to contribute to the development process of the app, we primarily consider LCW as the default wallet application for credentials issued by DiBiHo.

In the current version of the app, the LCW allows users to add and store credentials into the mobile wallet by either scanning a QR code or following a deep link that has been provided by a conforming issuer. Both of these options are implemented by DiBiHo's Credential Collection UI plugin developed for TUM Online [8].

The LCW app also offers functionality to view and explore stored credentials and to share selected credentials as a verifiable presentation through the mobile operating system's internal sharing mechanism. However, a major lack of functionality in this regard is that currently, the LCW app does not provide any functionality to specify the challenge parameter that should be used to construct a verifiable presentation. Currently, the app simply generates a random value as challenge parameter, making the created VP essentially useless since the challenge is normally provided by the relying party. However, we have already addressed this issue with the DCC and are confident that a functionality upgrade will be released soon.

### 3.1.2. Verifier Plus

Verifier Plus is another project developed by the DCC that is currently under development and expected to be released in the following months. It is closely related to the LCW, as it will allow users to share credentials from the app with anyone through a link. The LCW app will upload the credential to a public server operated by DCC that verifies the VC and stores it. Everyone who has access to the link can inspect the credential and verification result, which makes sharing easy, even with entities that don't have any VC-related software installed. [39]

While similar to our DiBiHo verifier at first glance, the focus of Verifier Plus is very different from ours. Verifier Plus goes a cloud-based approach by verifying a credential on a central server – which can then be inspected by arbitrary relying parties. For DiBiHo, we want to allow relying parties to verify credentials themselves, without being dependent on another party – and additionally, prescribe a unique challenge parameter for each exchange transaction, while Verifier Plus only evaluates stand-alone credentials. Additionally, we have strict requirements about identifying the credential holder, which Verifier Plus does not support.

## 3.2. Open Badges

Open Badges [40] is a standard for *digital badges* that was originally released in 2011. These badges visually represent skills or achievements that the holder has earned from some institution. A learner can collect several badges in a wallet and share them with others. Technically, Open Badges are .png or .svg image files that contain invisible metadata in JSON-LD format that conforms to the Open Badges standard. The data describes information about the issuer, the achievement, and the recipient. Additionally, it also includes information that allows relying parties to verify the badge.

In the last major version v2.0, an issuer can decide between two different types of verification models: For *Hosted Verification*, badges include an HTTPS URL where the issuer publicly hosts a JSON file that contains the badge's information. For *Signed Verification*, the badge contains a signed JWT and an URL to the corresponding public key, which should typically be hosted on the issuer's website.

The badge also includes personal data that allows to identify the holder. Most often, the email address is used for this. The identity data can be either included in plain text, or as a hash to protect the holder's privacy.

Compared to VCs, Open Badges 2.0 have several drawbacks: holders are most often identified by their emails, which are no permanent identifiers, but administrated by the email providers. There is no concept to present badges as in VPs that allows a relying party to verify if a holder has authorized the sharing of a badge. Also, badges depend on data hosted by the issuer, which can also pose a privacy concern. [41, 42]

To leverage the advantages of VCs, a proposal for a new Open Badges version 3.0 has been released that makes the data model of Open Badges fully compatible with VCs: Badges will be issued as VCs of type `OpenBadgeCredential` that follow the W3C VC data model. The issuer and recipient of badges can be specified as DIDs which makes Open Badges compatible

with regular VC wallets and eliminates the dependency on data hosted on the web. Still, Open Badges can be "baked" into .png and .svg files. [43]

## 3.3. OpenCerts

OpenCerts is a platform for issuing and verifying digital academic credentials that has been developed as part of Singapore's Smart Nation initiative. [44] It is built on top of the OpenAttestation library, another project of the Government Technology Agency of Singapore. OpenAttestation is "an open-sourced framework to endorse and verify documents using the [Ethereum] blockchain." [45]

To prove that a certificate was issued, there are two options: either via a hash lookup in an Ethereum smart contract, or via signing with a DID. The first option requires the issuer to deploy a *Document Store* smart contract to the Ethereum blockchain, which serves as registry for issuance logs. Instead of storing every single certificate's hash, multiple certificates can be aggregated by forming a Merkle Tree whose leaves represent the individual certificate hashes. For verification, the relying party will re-compute the Merkle Root and check if is contained in the document store on the blockchain. For the second option, the hash of the certificate gets signed with a did:ethr and the signature is stored directly in the certificate.

In order to identify issuers, OpenAttestation leverages the Domain Name System (DNS). Based on RFC-1464 [46], it is possible to "store arbitrary string attributes" as a `DNS-TXT` entry in a domain's DNS record. OpenCerts suggests to store the Ethereum address of the document store smart contract in the DNS entry. By including the corresponding domain into the document, a bi-directional binding between the document issuer and the domain is established.

OpenAttestation also supports selective disclosure through the way that a document's hash is computed: Instead of feeding the entire document into the hash function, each attribute is hashed individually, and a document's "targetHash" is computed over the alphabetically sorted list of all property hashes. To hide an attribute, the holder would just remove an attribute and add its hash to the certificate.

Furthermore, OpenAttestation also provides support for customized document rendering: An issuer can host their own web-based renderer that uses CSS and JavaScript to visualize JSON data. A verifier will embed this template using an *iframe* after successful verification.

While currently, OpenAttestation documents are not compatible to the W3C VC standard, a new version of the library is under development that promises to adapt the VC data model to support interoperability with VC-compatible software. [45]

The OpenCerts project adds features on top of OpenAttestation specifically for education certificates. Most notably, it introduces a data schema for digital education supporting degree diplomas and transcripts containing a list of courses. It also provides a central registry in form of a publicly hosted, unsigned json file, that associates the Ethereum address of document store smart contracts to the institutions behind it. [44]

To conclude, while some of OpenCerts features represent interesting first approaches towards several use cases that we also envision for DiBiHo, especially the verifier lacks

certain features like a concept for holder identification or challenge verification, and a more comprehensive approach towards a trusted issuer registry. Additionally, it is closely tied to specific technological choices, like a web-hosted renderer or a document store on the Ethereum blockchain.

## 3.4. European Blockchain Services Infrastructure (EBSI)

The European Blockchain Services Infrastructure (EBSI) is a project launched in 2018 by 29 countries (all EU member states plus Norway and Lichtenstein) and the EU commission. EBSI's vision is "to leverage blockchain to create cross-border services for public administrations, businesses, citizens and their ecosystems to verify information and make services trustworthy." [47]

The basis of EBSI forms a distributed ledger which is operated by a set of nodes across the participating countries. It is a public permissioned blockchain that employs the *Proof of Authority* consensus. It is based on Hyperledger Fabric and Hyperledger Besu (a permissioned Ethereum-based network).

EBSI introduces the DID method **did:ebsi** that is compatible with the W3C standard. It differentiates between two types of DIDs: [47, 48]

1) **Legal Entities**: DID Documents for legal entities are stored on the EBSI ledger. In order for a legal entity to register a DID, they need to obtain an *Authorization* to use the EBSI services from an *Authorization Issuer*.

2) **Natural Persons**: DIDs for natural persons are tied to a specific JWK key pair. Similar to did:key, they can be generated by anyone and are never registered in any DID registry. Therefore, they do not support updates to the DID document, e.g., for key rotation.

Before a legal entity can issue accepted VCs, they need to obtain an *Accreditation* to become a **Trusted Issuer (TI)**. Accreditations are issued by **Trusted Accreditation Organizations (TAO)** and are restricted to specific use cases and credential schemas. For example, a TAO could be the national Ministry of Education of an EBSI member state that issues an accreditation to a university to issue diploma credentials. TAOs have the purpose of extending the trust network and can also accreditate other TAOs. A set of **Root TAOs** represent a trust anchor in EBSI's chain of trust. Public information about trusted issuers and their accreditations is stored in a trusted issuer registry on the EBSI ledger. [47, 49, 48]

In 2021, eleven universities from different EU countries launched a first pilot project for different cross-border use cases, including the exchange of transcripts or university diplomas. In conclusion, driven by the European Commission and ministries across Europe, EBSI provides potential to become a blockchain solution supporting use cases for digital public services and potentially beyond. However, being a highly governed and regulated system and running on a permissioned blockchain, it also comes with limitations in terms of flexibility and interoperability. [47]

# 4. Credential Verification Checks

In the following, we will present the verification checks that we have identified to be essential in order to evaluate whether or not a given credential is considered *valid*.

For each check, we will begin with a short introduction and some examples, explaining why the check is relevant. We discuss how the check is performed and if necessary for better understanding, cover some high-level implementation aspects (for in-depth implementation details, we refer to the open source implementation of our verifier prototype on GitLab). Lastly, we will provide a summary of all possible outcomes and results of each check. Thereby, we use the following convention:

- If a check returns at least one **Error**, the check is considered as *failed*.
- If a check returns **neither an Error nor a Warning**, the check is considered as *passed*.
- If a check returns at least one **Warning** but **no Errors**, the check is *not* necessarily failed; a warning rather represents some information that may be relevant for the relying party and might impact their judgment of the verification result.

After covering all relevant checks, we will come to an intermediate conclusion and further discuss how the overall verification status shall be interpreted.

## 4.1. Sanity Checks

We start the verification process by performing a set of simple sanity checks to evaluate whether the provided input is a VC (or VP) conforming to the W3C data model as described in section 2.4. [1] To this end, we check if the uploaded input contains the set of mandatory properties (such as `@context`, `type`, `issuer`, `credentialSubject`, `proof`, etc.) and whether their values have the correct types (e.g., DIDs, date time, etc.). If any of these properties are missing or have an incorrect value, we return early i.e., skip all remaining checks, and show the user an error message, as displayed in table 4.1.

Furthermore, we also verify whether the credential is a valid JSON-LD document, i.e., the provided context must be resolvable and define all of the credential's properties. Otherwise, we return an error. While it would be possible to restrict the accepted JSON-LD contexts to a set of white-listed context URIs, we allow arbitrary custom contexts and leave the decision to accept or reject a VC based on its semantic contents to the relying party.

| Type | Error Code and Message |
|---|---|
| *Error* | `NOT_A_VC`<br>The provided input is not a verifiable credential. |
| *Error* | `NOT_A_VP`<br>The provided input is not a verifiable presentation. |
| *Warning* | `NO_CREDENTIALS`<br>The uploaded VP does not contain any credentials. |
| *Error* | `LD_CONTEXT_NOT_RESOLVABLE`<br>A JSON-LD context file could not be resolved. |
| *Error* | `UNDEFINED_JSON_LD_PROPERTY`<br>Not all credential's properties are defined in the provided JSON-LD context. |

Table 4.1.: Overview of Possible Results of the Sanity Checks

## 4.2. Signature and Proof Verification

If the sanity checks have been passed we can be sure that the input conforms to the W3C standard and we will perform the actual verification. We will begin with the *Signature and Proof Verification Check*. While this is only one of many checks that need to pass in order for a credential to be considered as valid, it is one of the most fundamental verification checks, as the entire concept of VCs and VPs is based on signatures and proof verifications. Consequently, it is also one of most general checks, as the credential's signature needs to be verified for every given VC - regardless of the credential type or application domain. [1]

A valid credential signature proves that a) the DID that claims to have signed the credential has *indeed* signed it; and b) the credential has not been modified after it has been signed; that is, the credential's *integrity* is confirmed [13, 15].

This point is particularly important since essentially, VCs are normal JSON files that anyone could manipulate with a simple text editor and enter arbitrary information. As the smallest semantic edit will immediately invalidate the signature, the modified credential can easily be evaluated as invalid.

An example proof property of a VC is shown in figure 4.1. It must contain the following information: [1]

- The `type` of the proof, which usually includes the concrete crypto suite that was used to create the signature and consequently should be used for verification (e.g., `Ed25519Signature2020` or `EcdsaSecp256k1Signature2019`)
- The `verificationMethod`, which contains a reference to the public key of the entity that signed the credential. Typically, the key material is contained in a DID document and referenced by a DID URL.
- The actual signature (here, the property of the signature field is named `proofValue`, though the name may vary depending on the signature type).

```
{
    < other VC properties >,
    "proof": {
        "type": "Ed25519Signature2020",
        "created": "2022-09-23T18:49:13.561Z",
        "verificationMethod": "did:web:dibiho.org:TUM#key",
        "proofPurpose": "assertionMethod",
        "proofValue": "z5jy5qMe6VVESfyjBrd2f7ZEHLv75LgGWyocNe68UMRzdwM2UomcBadL8uwmBjVzoPEWDxMny4bbaiWupW6svxToJ"
    }
}
```

Listing 4.1: Example of a VC's Proof Property

In general, we can split the Signature and Proof Verification Check in two parts: [15] First, the verifier needs to obtain the proof signer's public key material, by resolving the URI or DID specified in `proof.verificationMethod`. Depending on the DID method being used, this may include fetching the DID document with the key information from the web, a decentralized ledger, or another sort of DID registry.

Then, once the verifier has access to the public key material, the credential, and the signature, it can perform the actual signature verification. Thereby, the exact procedure depends on the proof type – as explained in section 2.1, it typically includes a transformation of the data input into a canonicalized form which is then fed into a message digesting algorithm like a hash function to produce a fixed-sized input, the *data to be verified*. This must be the exact same data that the issuer also used for the signature creation. It is then - along with the public key and the signature value - passed to the signature algorithm which mathematically verifies the integrity and authenticity of the credential, returning a boolean response, i.e., true = valid, or false = invalid – in which case we will return an error. [15, 19]

Because the signature verification involves many low-level cryptographic operations, it is never recommended for developers to implement such algorithms themselves for safety reasons, unless they are cryptography experts. Luckily, different open source crypto suites developed by cryptographers are available online that provide utilities for the above operations. Besides such low-level crypto suites that handle pure signature verifiction, also several libraries and frameworks exist that support the verification specifically on a VC and VP level. In section 5, we will analyze different functional and non-functional requirements that a framework needs to support for our use case and evaluate a list of available SSI frameworks for the implementation in our verifier prototype. [18, 19, 20]

As mentioned above, if the signature is valid, we know that the proof signer's DID has indeed signed the credential and that the credential has not been manipulated. As one additional simple check at this point, we need to make sure that the DID entered as `issuer.id` in the credential is the same as the proof signer's DID. Otherwise, we will return a mismatch error. An overview of different errors of this check is shown in table 4.2.

In this section, we mainly talked about the verification of the *credential*'s signature that has been signed by the *issuer*. However, *verifiable presentations* also include a signature, that has been created by the credential *presenter* and also needs to be verified. The actual proof and signature verification works exactly as discussed in this section. Because VPs can also include

| Type | Error Code and Message |
|---|---|
| *Error* | `CREDENTIAL_SIGNATURE_INVALID`<br>*The signature of the credential is invalid.* |
| *Error* | `DID_RESOLVE_ERROR`<br>*The DID document of the issuer could not be resolved.* |
| *Error* | `VC_ISSUER_PROOF_MISMATCH`<br>*Credential issuer must match the verification method controller.* |
| *Error* | `PROOFTYPE_NOT_SUPPORTED`<br>*Did not verify any proofs; insufficient proofs matched the acceptable suite.* |

Table 4.2.: Overview of Possible Results of the Signature Check

a list of multiple credentials, whenever a VP is verified, we need to verify the VP's signature, and additionally, all of the individual credential's signatures that are contained in the VP.

## 4.3. Trusted Issuer Check

If a credential's signature is valid, a verifier can be sure that the credential has been issued by a certain DID, and that its contents are authentic and have not been modified. However, this information is only useful if we can tell who the controller of this issuer DID is – i.e., which real world organization is associated to the given issuer DID. Since anyone can easily generate new DIDs and issue credentials with arbitrary content, a credential from an unknown issuer DID has zero information value and should never be trusted. Every verifier therefore needs a *trust anchor*, allowing them to check whether credentials issued by a given DID are trustworthy.

For many application domains involving VCs, this problem is relatively easy to solve, since often, the number of potential VC issuers is very small and a simple pre-defined set of white-listed DIDs is sufficient to assess the trustworthiness of an issuer. In many other cases, all issued VCs are governed by the same organization – that often also provides their own verifier software. E.g., a transit company issuing train tickets as VCs can simply maintain their own set of white-listed issuer DIDs and integrate it accordingly in their verifier i.e., ticket inspector app.

However, for the use case of verifying international higher education institution credentials, this problem becomes significantly more challenging as it is impossible to know all higher education institutions worldwide and they are completely independent organizations. To this end, we designed and implemented a concept for a *trusted issuer registry*, which the relying party can query to evaluate whether a given issuer DID is a trusted entity, and which allows to uniquely identify the organization behind that DID. [6]

In chapter 6, we will discuss the analyzed requirements for the trusted issuer registry for our use case and governance aspects, as well as technical choices regarding the underlying infrastructure, the data needed to be stored and implementation details in more depth.

We can already conclude that the trusted issuer registry has the following characteristics that

are relevant for this chapter:

- The relying party can configure which trusted issuer registry they rely on as trust anchor. Also, multiple registries can be queried.
- The minimal required information that each registry entry must contain for a registered DID to uniquely identify the issuer, are the university's name and website URL.
- Multiple registry resolution methods are supported:
  - *Ethereum*: The registry is represented by an Ethereum smart contract. The contract needs to conform to a pre-defined interface; the actual implementation is up to the smart contract maintainer. We provide an MVP implementation for a registry contract that can be maintained by several entities that must achieve consensus by a majority vote for every change of the registry data.
  - *Web*: The registry is wrapped into a signed VC in order to prove integrity and prevent man-in-the-middle attacks. It can contain a proof set of signatures from multiple entities that must approve the information stored in the registry.
- A registry can optionally link to sub-endorsed registries, allowing for hierarchical structuring and administration.

In terms of the implementation of the trusted issuer check, we simply loop through the list of configured trusted issuer registries, query them by using the appropriate driver (i.e., Ethereum smart contract interface or via HTTP request) and check whether the credential's issuer DID is registered and whether the returned institution name matches the issuer name in the credential – in which case the check is passed.

If there is a name mismatch, the check fails and the credential is considered invalid. This is to ensure that no registered university can issue a valid credential with the `issuer.name` property (which is often prominently displayed in many wallet apps) of a another university (e.g., a DID registered as TUM is prevented from issuing a valid credential in the name of "Harvard University").

If all of the configured registries have been queried but no issuer entry has been found, the check is failed because the identity of the issuer could not be confirmed. In case a trusted issuer registry cannot be reached or the signature of a web-based registry is invalid, we return an additional warning for the user's information. An overview of all possible results of the Trusted Issuer Check is shown in table 4.3.

## 4.4. Credential Status Check

An important component of any digital credentials ecosystem is a mechanism that allows revocation of previously issued credentials. There are different reasons why an issuer might need to revoke a credential. In the academic context, universities reserve the right to revoke degrees in certain cases, e.g., if fraudulent activities, such as plagiarism, are discovered after the degree has been awarded.

| Type | Error Code and Message |
|---|---|
| *Warning* | `INVALID_WEB_ISSUER_REGISTRY_PROOF`<br>*The web-based trusted issuer registry does not have a valid proof. It may have been manipulated by an unauthorized party.* |
| *Warning* | `TRUSTED_ISSUER_REGISTRY_WEB_CONNECTION_ERROR`<br>*Error while trying to fetch trusted issuer entry from web.* |
| *Warning* | `TRUSTED_ISSUER_REGISTRY_FILE_CONNECTION_ERROR`<br>*Error while trying to fetch trusted issuer entry from file.* |
| *Warning* | `ETH_CONNECTION_ERROR`<br>*Error while trying to fetch trusted issuer entry from ETH node.* |
| *Error* | `ISSUER_NAME_MISMATCH`<br>*The issuer's name does not match the entry in the trusted issuer registry.* |
| *Error* | `NO_TRUSTED_ISSUER`<br>*The issuer's DID is not listed in any of the trusted issuer registries.* |

Table 4.3.: Overview of Possible Results of the Trusted Issuer Check

A much more frequent reason to revoke a previously issued VC is, however, the ability to correct errors in the issuing process or to update credentials - e.g., if a credential has been issued by mistake or contains errors such as spelling errors or an incorrect grade. Even credentials that have initially been issued correctly might need to be updated at a later time if the signed data becomes outdated (e.g., because a student changes their name), in which case the credential can be revoked and a new credential will be issued.

The VC data model contains the property `credentialStatus` which allows the issuer to include information into the VC that lets the verifier check if the credential has been revoked in the meantime. [1]

#### Credential Revocation via StatusList2021

StatusList2021 [50] is a standard for specifying credential status information proposed by the W3C Credentials Community Group. It aims to provide a privacy-preserving and yet performant and space-efficient approach to publish status information for large sets of credentials.

Privacy is an important consideration for the design of status list mechanisms. When the status information for every individual credential is published in its own status resource (e.g., having its own URL), the publisher of the status information (which is typically the credential issuer) will know exactly what credential is being verified. By analyzing the status check request's metadata, such as the IP address, the issuer may even conclude from which relying party a status check request is coming, effectively allowing the issuer to track which relying parties a credential holder interacts with, which constitutes a serious lack of privacy. [50]

To solve this problem, StatusList2021 proposes to place status information for a large number of VCs into one common list to achieve *herd privacy*: once a verifier retrieves the

```
{
  < VC properties ... >
  "credentialStatus": {
    "id": "https://dibiho.org/credentials/status/1#1411",
    "type": "StatusList2021Entry",
    "statusPurpose": "revocation",
    "statusListIndex": "1411",
    "statusListCredential": "https://dibiho.org/credentials/
      ↪ status/1"
  }
}
```

Listing 4.2: Credential Status Field of a VC

```
{
  "@context": [
    "https://www.w3.org/2018/credentials/v1",
    "https://w3c-ccg.github.io/vc-status-list-2021/contexts/v1.jsonld",
    "https://w3id.org/security/suites/ed25519-2020/v1"
  ],
  "id": "https://dibiho.org/credentials/status/1",
  "issuer": "did:web:dibiho.org:TUM",
  "issuanceDate": "2022-08-22T08:56:53.839Z",
  "type": [
    "VerifiableCredential",
    "StatusList2021Credential"
    ],
  "credentialSubject": {
  "id": "https://dibiho.org/credentials/status/1#list",
  "type": "StatusList2021",
  "statusPurpose": "revocation",
  "encodedList": "H4sIAAAAAAAAA-3OsREAEBAAwR-RUAlKVToykeiHZLeAm4
      ↪ vYamToKZW7nFMeK78HAAAAAABY2u8BAAAAAACA04gJiuEEjdQwAAA"
  },
  "proof": { ... }
}
```

Listing 4.3: Example of a Status List Credential

list, the issuer does not know for which specific credential the verifier checks the status information for.

However, hosting such lists that contain status information for massive amounts of credentials can become very expensive in terms of performance and space-efficiency – both, on the issuer's as well as on the verifier's side. To keep the information stored for each credential minimal, the status information for all VCs are expressed as simple binary values, i.e., 0 = valid or 1 = revoked. These bit values of mulitple credentials are concatenated to a bitstring list, with a minimum size of 131,072 entries, which is equal to 16 KB. The index in the bitstring that represents the status of a specific credential is then added to the `credentialStatus` property in the VC, along with the link to the statusList.

To bring down the storage requirements further, the bitstring is compressed by using the GZIP compression algorithm [51] and applying base64 encoding [52] to the result. Since most entries of the list are zero, the compression makes the resulting encoded list very compact. [50]

The statusList is then wrapped into a signed VC itself, which gives the verifier a way to check if the contents of the status list credential are authentic. The VC containing the status list is hosted on the web. To activate the status check for a given VC, a URL to a statusList and the index that represents the credential's status is entered into the credentialStatus property.

The example in listing 4.2 shows a `credentialStatus` property of a VC conforming to the StatusList2021 standard. [50] It contains the following properties:

- the `type` property must be set to the value `StatusList2021Entry` to declare that the credential status conforms to the StatusList2021 standard
- the `statusPurpose` describes what kind of status information can be checked through the provided data. It may take the following values:
  - `revocation`: *permanently* cancels the validity of a VC.

  - `suspension`: *temporarily* prevents the acceptance of a VC.
  - arbitrary additional values can be defined by the issuer.

- the `statusListCredential` contains a URL to the VC of type `StatusList2021Credential`, hosted on the web, that contains the encoded status list
- the `statusListIndex` provides the index in the de-compressed status bitstring list that corresponds to the binary status information for the credential

The status list credential is a minimal VC that contains the encoded credential status list and the `statusPurpose` in the `credentialSubject`, along with the typical VC properties (e.g., issuanceDate, proof, etc.). An example is shown in listing 4.3. In order to check the revocation status of a credential that supports the `StatusList2021` standard, a verifier has to proceed as follows: [50]

1. Retrieve the *Status List Credential* from the URL provided in the `statusListCredential` field of the VC's `credentialStatus` property
2. Check if the status purpose of the `credentialStatus` property matches the purpose defined in the *Status List Credential* (e.g., "revocation")
3. Verify if the *Status List Credential* has a valid signature proof
4. Decompress the `encodedList` of the *Status List Credential*, by applying base64-decoding and expanding the output with the GZIP decompression algorithm
5. If the bit at position `credentialIndex` in the decoded list is equal to `1`: the credential is revoked. If it is `0`, the credential is not revoked.

#### Universal Status List

While the StatusList2021 Standard provides a scalable approach to verify and store status information for VCs while still protecting privacy, it creates limitations since the location of the status list is directly defined in the credential as an HTTPS URL and universities are required to host the status list on the web.

As discussed in section 2.3.2, a significant drawback of web-based resources is the lack of guaranteed availability. Servers can have outages any time or can be permanently shut down, which would make it impossible for relying parties to check the status information of a presented VC. Additionally, if the location of the status list changes for whatever reason, the verifier will never get notice of this, since the URL was directly entered and signed in the VC.

To overcome these issues, the DiBiHo project team proposes *UniversalStatusList2022*, a simple yet powerful modification of StatusList2021, that adds an abstraction layer on top, leveraging the properties of DIDs to obtain a more flexible and general approach towards credential status management. [53]

While the credential status entry is similar to StatusList2021 [50], containing information like the status list index and the reference to the status list, we do not directly store the URL of

```
"credentialStatus": {
    "id": "did:web:dibiho.org:TUM#StatusList:1#1311",
    "type": "UniversalStatusListEntry",
    "statusPurpose": "revocation",
    "statusListIndex": "1311",
    "statusListCredential": "did:web:dibiho.org:TUM#
      ↪ StatusList:1"
}
```

Listing 4.4: Universal Status List Entry

```
{
  "@context": "https://www.w3.org/ns/did/v1",
  "id": "did:web:dibiho.org:TUM",
  "verificationMethod": [ { ... } ],
  "authentication": [ ... ],
  "assertionMethod": [ ... ],
  "service": [
    {
      "id": "did:web:dibiho.org:TUM#StatusList:1",
      "type": "StatusList",
      "serviceEndpoint": "status:web:https://dibiho.org/credentials/status/
      ↪ 1.json"
    }
  ]
}
```

Listing 4.5: DID Document with StatusList Service

the status list in the credential, but instead, we rather reference a DID-service, from where the verifier can obtain all information to fetch the actual status list. In the example in listing 4.4, the status information can be retrieved at the service `StatusList:1` in the DID document of `did:web:dibiho.org:TUM`.

The corresponding DID document with the service definition is shown in listing 4.5. The service entry must be of type *StatusList* and the *serviceEndpoint* field contains the actual location of the statusListCredential which is of the format `status:<method>:<URI>`.

Similar as in DID resolution, different methods are supported to retrieve the status list credential, making this mechanism available to a wide set of different technologies (such as decentralized ledgers) and implementations. At this time, we support the methods `status:web` and `status:eth`.

The status method `status:web` contains a URL that points to a default `StatusList2021Credential`. The verification of the credential status from there works the same as discussed for `StatusList2021` above. The status method `status:eth` contains an address of an Ethereum smart contract deployed by the status list publisher that must conform to an interface which provides the encoded credential status list. The verifier fetches the status list data from the smart contract and proceeds with the verifiction as discussed above.

By leveraging DID Documents to store the location of the statusList and by supporting different methods to fetch them, `UniversalStatusList` solves the problems mentioned above and provides a flexible and generic revocation mechanism.

In our prototype, we support both, the `StatusList2021` standard, as well as DiBiHo's `UniversalStatusList` specification, currently supporting drivers for `status:web` and `status:ethr`. To achieve backward compatibility with older credentials, we also add support for the `RevocationList2020` standard [54], which is the predecessor of `StatusList2021` and very similar for the most part, with the only major differences being the lack of the status purpose field, as it only supports revocation, and a difference in the list compression algorithm, using ZLIB [55] instead of GZIP.

| Type | Error Code and Message |
|---|---|
| *Error* | `INVALID_STATUSLIST_PROOF`<br>*The status list does not contain a valid signature.* |
| *Error* | `CREDENTIAL_REVOKED`<br>*The credential has been revoked by the issuer.* |
| *Error* | `STATUS_CHECK_ERROR`<br>*An error occured while fetching the status information.* |
| *Error* | `COULDNT_RESOLVE_DID_DOC`<br>*Could not fetch the universal statuslist service entry.* |
| *Warning* | `NO_CREDENTIAL_STATUS_INFORMATION`<br>*The credential does not contain any revocation status information.* |
| *Error* | `UNSUPPORTED_CREDENTIAL_STATUS`<br>*Currently, RevocationList2020Status, StatusList2021Entry, UniversalStatusListEntry are supported.* |
| *Error* | `UNSUPPORTED_UNIVERSAL_STATUS_LIST_METHOD`<br>*Currently, status:web and status:ethr are supported.* |

Table 4.4.: Overview of Possible Results of the Status Check

| **Property Name (Current Version)** | **Property Name (Future Version)** | **Must be Before/After Verification Timestamp** |
|---|---|---|
| `issuanceDate` | `validFrom` | *BEFORE* |
| `expirationDate` | `validUntil` | *AFTER* |
| | `issued` | *BEFORE* |

Table 4.5.: Overview of date-time properties required for credential verification according to the VC Data Model Specification and its expected next version.

## 4.5. Timestamp Checks

The VC data model contains several optional and mandatory properties valued as date-time strings, which represent points in time in the past or future that are important for verification purposes (summarized in table 4.5). The *Timestamp Check* verifies whether a credential is valid based on those property-values and the current timestamp during the verification. [1]

The `expirationDate` expresses the date and time after which a credential becomes invalid. While Bachelor's and Master's degrees typically have an unlimited validity period, other credential types, such as student ID credentials might only be valid during a limited period of time, e.g., until the end of the semester. Even for degrees without a natural expiration date, issuers might decide to set a "technical" expiration date until the VC becomes invalid and credential holders have to renew their VC, which adds additional security as new keys can be

used to sign the renewed credential.

According to the VC data model specification, the mandatory property `issuanceDate` expresses the date and time when a credential becomes valid (i.e., *not* when it was issued) and is therefore also allowed to lie in the future. In the next version of the VC data model specification, the issuanceDate property is expected to become deprecated and replaced by the fields `issued` and `validFrom` [1], which helps to better differentiate the start of a credential's validity and the timestamp of it's issuance. The `validFrom` property therefore allows the issuer to set the start date of a credential's validity to a date other than the issuance date. In practice, this allows universities to issue credentials before an official graduation date, which is a highly likely scenario, as the graduation date is often the formal date of the official graduation ceremony, while most students have completed their degree requirements already weeks or months earlier. Since especially for large graduation cohorts, the issuance of all credentials can take several hours and might involve complex administrative processes, having the option to issue credentials in advance makes the issuing process more flexible.

The `issued` property is expected to be added in the next version of the VC data model specification and expresses the timestamp when a VC was signed by the issuer, and therefore, should be in the past, i.e., before the current verification timestamp. Relying parties might also decide to enforce stricter verification policies, e.g., require a credential to be issued within the last year. In theory, if an issuer loses trust at some point in time (e.g., because their private signing key was compromised; or a university loses accreditation), a verifier could require that the issued date lies before that specific event. However, as the issued date is set by the issuers themselves, it cannot be guaranteed to be correct i.e., equal to the real timestamp of issuance. For such verification feature, an objective notion of time or order would be needed, such as the block-index of an issuance log with the credential's hash on a public blockchain. While we do not consider issuance logging as a core requirement in DiBiHo, we believe that this direction leaves potential for future work that we will further address in chapter 8.5. [1]

The implementation of these timestamp based checks is relatively trivial, as we only need to compare the timestamps of the credential property values with the current system timestamp, as shown in table 4.5. For the case that the relying party might also be interested of the verification status at another point in time, different than the current system timestamp, the configuration allows to specify a lower bound for the expirationDate, and an upper bound for the validFrom date.

## 4.6. VP Challenge Verification

All of the previous checks happened on the credential level, which means that we run them individually for every credential contained in a VP. If all of the previously explained checks have passed for a credential, it can be considered *valid*. However, whenever a relying party needs to make a decision concerning a person submitting a credential, the sole fact that a pure "standalone" credential is valid is *not* sufficient - since it also needs to be verified whether the person submitting the credential is actually the holder of the credential (otherwise, one could just upload any other valid credential, e.g., a friend's credential or a VC someone posted on

| Type | Error Code and Message |
|---|---|
| *Error* | `CREDENTIAL_ISSUANCEDATE_IN_FUTURE`<br>*The issuance date of the credential is in the future.* |
| *Error* | `CREDENTIAL_EXPIRED`<br>*The credential has expired.* |
| *Error* | `CREDENTIAL_ISSUED_IN_FUTURE`<br>*The issued date of the credential is in the future.* |
| *Error* | `CREDENTIAL_VALID_IN_FUTURE`<br>*The validFrom date of the credential is in the future.* |
| *Error* | `CREDENTIAL_VALID_IN_PAST`<br>*The validUntil date of the credential is in the past.* |

Table 4.6.: Overview of Possible Results of the Timestamp Check

the internet). This is where the concept of VPs comes in, that also include a signature proof from the presenter in order to authenticate themselves as the owner of a specific DID.

The presenter's proof does not only sign the included VCs, but also a uniquely generated *challenge parameter* that was provided by the relying party, which fulfills several important purposes: [1]

- The relying party can be sure that the VP has been generated specifically for them, as it contains their unique challenge.
- The presenter proves that they have access to the private key of the DID that signs the VP, i.e., control the DID.
- Replay Attacks are prevented (e.g., that the same VP is submitted twice).

While we have addressed the signature verification procedure in section 4.2, the *VP Challenge Check* verifies the correctness of the challenge parameter. In theory, this check is trivial and not more than a basic string comparison between the relying party's *provided challenge* and the presenter's *signed and submitted challenge*. At closer consideration, however, it turns out that this check is significantly more complex and opens a set of new questions:

- How is the challenge generated?
- How is the challenge provided to the learner?
- How is the VP submitted to the relying party?
- How is the originally generated challenge parameter retrieved and associated with a submitted VP?

While the middle two bullets are rather a question of *Relying Party - Holder - Interaction* which we will cover later in section 7.1, in the current section, we will focus on the outer two aspects, i.e., challenge generation and challenge retrieving for cross-checking with the submitted VP.

### Stateful Challenge Generation and Verification

In principle, a challenge parameter can take any string value of arbitrary length. Ideally, it should be unique so that for every submitted VP, a new challenge will be required to avoid replay attacks. Furthermore, the challenge parameter should not be guessable by the presenter, which makes auto-incremented challenge numbers unsuitable.

The simplest yet completely viable option is to create the challenge via pure random value generation. A common pattern for generating unique values are UUIDs (Universally Unique IDentifiers) [56]. UUIDs are typically 128 bit long sequences (i.e., consisting of 16 bytes or 32 hexadecimal characters) that are considered unique without the need of a registration process, meaning we do not need to check if we have ever created a challenge parameter with the same value before.

As an additional security measure, a relying party can also define an expiration timestamp for a challenge. For example, if a learner does not submit their VP with the requested challenge within 24 hours, a new challenge may be required.

Once a challenge has been generated, the relying party can store it into a database, along with the expiration date. In order to be able to verify at a later time, once a presenter submits a VP, whether the challenge parameter in the submitted VP matches the earlier generated challenge that was given to the presenter, the database entry must be associated with an ID that identifies the particular exchange transaction. This exchange ID or transaction ID would typically come from the relying party's system that the learner interacts with (e.g., it could be an application ID coming from an application portal). Alternatively, the challenge payload could also be identified with a user ID or with the presenter's DID.

Once a presenter submits their VP, the relying party will retrieve the corresponding challenge parameter that was given to the presenter from the database, check if it matches the challenge in the submitted VP, and if the challenge has not yet expired. If all of this applies, the VP Challenge Check is passed - otherwise, an error will be returned as shown in table 4.7. After the challenge has been received, the challenge database entry will be cleared, in order to prevent another submission with the same challenge.

### Stateless Challenge Generation and Retrieval

While the above approach is relatively simple and straight forward, one important characteristic is that it requires some form of state management (e.g., a database to store and retrieve generated challenges). In the following, we will explain an alternative approach for *stateless* challenge generation, that does not entail this constraint.

The idea is - instead of purely randomly generating a challenge - to use a JWT [22] as challenge parameter (a similar concept is also described in [57]), which brings the following advantages:

1. All metadata that is required for verification - such as the expiration timestamp and a transaction ID - can be stored directly in the challenge JWT, so that no database is required.

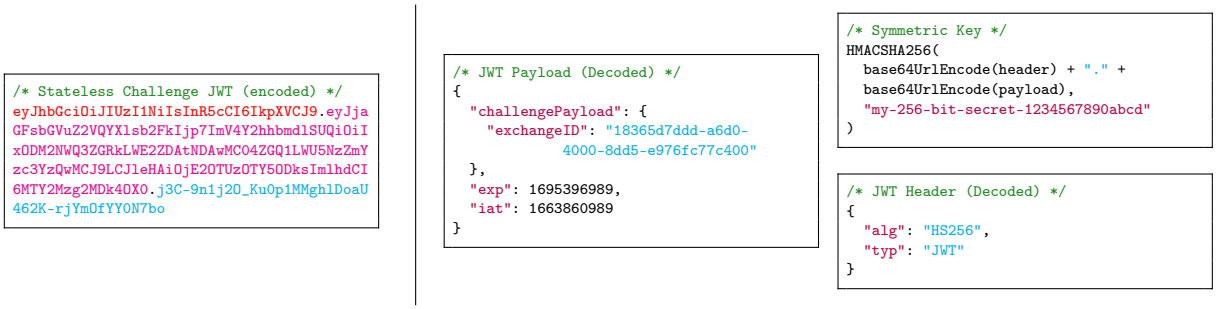


Figure 4.1.: Example Challenge JWT Representing a Stateless Challenge Parameter

2. The JWT is signed with a secret only known to the relying party. The same secret will be used for verification, making it impossible for other entities to manipulate the challenge metadata or create valid challenge JWTs themselves.

By storing all the data relevant for challenge verification directly *in* the challenge – secured by a signature to ensure integrity, there is no need to store it in a database. An example JWT challenge is shown in figure 4.1. At time of verification, the verifier checks if the JWT is decodable using the secret only known to the relying party. The verifier also checks whether the challenge has not yet expired, and whether the transaction ID is as expected. If all of this applies, the VP Challenge Check is passed.

Compared to the stateful approach, we see the following advantages:

- No database setup is needed, which simplifies the overall setup, making the verifier software more accessible and easier scalable.
- Challenge generation and challenge cross-check can happen on different devices or systems that can be completely disconnected. However, both must be controlled by the relying party, in order to have access to the JWT secret.
- If the database in the stateful approach is a distributed system, leading to additional latency, the stateless approach may be faster as no database call to another system is required.
- There is no problem of spamming challenge generation API requests or storage limits, as nothing needs to be stored. [57]

However, there is also one important drawback: since no information is stored in the stateless approach, a challenge token does not get de-activated after submission of a VP. Consequently, presentation replays cannot be entirely prevented – until the expiration timestamp has been reached and the challenge JWT becomes invalid. However, replays are only possible as long as the transaction ID remains the same; e.g., a student might present another credential with the same challenge parameter only to the same application, which seems acceptable. This

| Type | Error Code and Message |
|---|---|
| *Warning* | `JWT_CHALLENGE_EXPIRED`<br>*The JWT Challenge Token is expired.* |
| *Error* | `JWT_CHALLENGE_PAYLOAD_MISMATCH`<br>*Invalid JWT challenge transaction ID payload.* |
| *Error* | `JWT_CHALLENGE_SIG_INVALID`<br>*The JWT challenge token signature is invalid.* |
| *Warning* | `SKIPPED_CHALLENGE_CHECK`<br>*Skipped Ownership Check (no challenge provided by the verifier).* |
| *Error* | `CHALLENGE_MISMATCH`<br>*Challenge Mismatch* |

Table 4.7.: Overview of Possible Results of the Challenge Check

shows the importance of having a limited challenge validity timeframe and a transaction ID that uniquely identifies the exchange session.

Overall, we believe that the stateless challenge generation and verification is an interesting direction that offers some promising benefits. We do acknowledge that in real-world-systems, where databases are widely used anyway, the mentioned advantages become less significant. However, particularly for simple test and demo setups, the stateless challenge approach has proven to be quite valuable during the development of our prototype, as it might also be in certain scenarios, where the challenge generation should happen completely separated from the VP verification.

In our verifier implementation, we support both approaches and let the relying party either specify an expected challenge parameter or set a flag to indicate that they expect a JWT challenge that must match a specified transaction ID payload.

## 4.7. Credential Holder DID Consistency

Besides the challenge parameter defined by the relying party, a verifiable presentation contains a list of credentials and a signature signed by the presenter. If the VP's signature is valid, it proves that 1) the presenter created the signature specifically for the verifier as they prescribed the challenge; and 2) the presenter therefore has access to the private key, i.e., authenticated as the owner of the DID which was used to sign the proof.

However, as the presenter can wrap arbitrary credentials into a VP, it is not necessarily guaranteed that the presented credentials belong to the same DID, and therefore, to the presenter. For instance, `did:web:dibiho.org:Lucas.Learner` could sign a verifiable presentation, but include a credential from `did:web:dibiho.org:Stella.Student`. Therefore, we only consider VCs as valid whose credentialSubject's DID is equal to the DID signing the presentation.

The Credential Holder DID Consistency Check therefore operates on a per-credential-basis, but is only evaluated if the credentials are submitted in form of a verifiable presentation, as

| Type | Error Code and Message |
|---|---|
| *Error* | `CREDENTIAL_HOLDER_PROOF_SIGNER_MISMATCH`<br>*Credential Holder must be same as Presentation Signer.* |

Table 4.8.: Overview of Possible Results of the Holder DID Consistency Check

it compares each credentialSubject's DID to a the presentation signer's DID. If both do not match, the credential is invalid.

For the case that one presenter owns multiple DIDs and wants to present credentials issued to different DIDs, they have to submit them in separate VPs to individually prove ownership over all these DIDs.

## 4.8. Credential Holder Identification

By signing a verifiable presentation, a presenter can prove that they are the owner of the credential that is being submitted, i.e., have access to the private key corresponding to the DID set as credential holder. However, knowing the DID of the presenter does not necessarily allow the relying party to determine who the presenter is in the real world, i.e., identifying the natural person behind the DID by personal information such as first name, last name, birth date, and birth place.

In section 4.3 we already discussed a similar problem, as we needed a way to identify the organization (e.g., university) behind the DID of the credential *issuer*. We solved this problem by querying a publicly accessible trusted issuer registry which provides a mapping from an issuer DID to the corresponding university. The registry serves as trust anchor and is maintained by a trusted entity (such as a national university association or consortium like DCC) or in a self-managed way (e.g., based on a voting mechanism).

While a similar solution concept (e.g., in form of a "Student DID Registry") would be conceivable for the problem of holder identification, such a direction turns out to be less suited because both problems, although similar on a high level, have fundamentally different characteristics:

Typically, universities are publicly known institutions who can leverage established IT infrastructure (e.g., domains and TLS certificates) and already verified communication channels for the purpose of self-identification and communication of their DIDs. Along with the fact that universities are often organized in different associations and consortiums like DCC, this makes the setup and maintenance of a trusted issuer registry certainly feasible. Most notably, the number of issuer organizations that need to be recorded in a trusted issuer registry is comparably small and limited - for example, in Germany there are currently 422 higher education institutions (of which 108 are universities) [58]. On the other hand, obviously, the number of students is of a completely different order of magnitude and many times that of issuers: Alone in the winter semester 2021/2022 there were 2,941,915 students enrolled in Germany (1,725,461 at universities) [59] - with new students coming in every year. Maintaining a public registry for tens (or, internationally, for hundreds) of millions of students

would not only be challenging in terms of operating concepts, governance, and storage cost, but also create additions problems such as privacy concerns – making this approach clearly less practicable.

Another potential solution approach would be to directly include personal information identifying the credential holder into the VC. Traditional physical diplomas typically contain the degree holder's first name(s), last name, birth date, and birth place, which could also be added inside the `credentialSubject` field of a diploma credential. While this represents the solution closest to the real world, it is not ideal as it leads to different problems:

When a student presents multiple credentials, each credential containing the personal information introduces information redundancy and can cause verification anomalies (e.g., one credential contains the name "Lu**c**as Learner" and another credential "Lu**k**as Learner"), which ideally should be avoided. Also, if a learner's personal information changes (e.g., because a change of family name, for example, due to marriage), all credentials are outdated and need to be updated (e.g., revoked and re-issued) individually by the corresponding issuers. Additionally, storing the student's information directly inside the credential might lead to privacy concerns (e.g., even if a verifier might not require to see the student's birth date, it is still always included in the credential).

#### Identity Credential Linking

To overcome these challenges, the DiBiHo project team introduces a concept called Identity Credential Linking. An *Identity Credential* is a separate VC that is typically issued to students after an in-person identity check. The identity credential contains the student's personal information (i.e., first name(s), last name, date of birth, place of birth; see listing 4.6 for an example), providing a mapping between a student's real identity and their DID. The identity credential can be issued by the student's university and the identity check could take place during the enrollment procedure. The identity credential can then be considered as a type of digital student ID and could even supersede traditional physical student ID cards in the future. However, an identity credential may also be issued by another organization (e.g., a different university or a state authority) as long as it is a trusted issuer. [53]

If a student wants to present their diploma credential, in addition, they must also include an identity credential into the verifiable presentation. While the issuers of the two credentials do not need to be the same DID or organization, the holder DID of both credentials match to prove the identity of the diploma holder.

To further ensure integrity, an identity credential is "linked" to a diploma credential by storing the identity credential's SHA256-Hash [60] into the diploma credential's property `identityCredentialHash`. This linking mechanism allows the relying party to check if the student presented the exact same identity credential which was used by the issuer to sign the degree.

It is important to note that by leveraging the properties of a hash function, the diploma credential still contains the same level of information that it would if all the personal data was directly entered into the credential - the only difference is the form of representation of the credential, which is now split apart in two credentials, that are both needed for verification

```
{
    "@context": [ ..., "https://dibiho.org/contexts/elmoLearner", ... ],
    "type": [
        "VerifiableCredential",
        "IdentityCredential",
        "ElmoIdentityCredential"
    ],
    "issuer": { "id": "did:web:dibiho.org:TUM", ... },
    "credentialSubject": {
        "id": "did:web:dibiho.org:Lucas.Learner",
        "elmo": {
            "givenNames": "Lucas",
            "familyName": "Learner",
            "bday": "2000-05-10T00:00:00Z",
            "placeOfBirth": "Bamberg, Germany"
        }
    },
    "expirationDate": "2032-01-01T00:00:00Z",
    "id": "https://example.com/credentials/1872",
    "credentialStatus": { ... },
    "issuanceDate": "2022-09-15T16:29:18Z",
    "proof": { ... }
}
```

Listing 4.6: Example of an Identity Credential

purposes. This concept of using a linked identity credential brings the advantage, that the ID credential creates a single source of truth for all identity data: Even if a student presents multiple diploma credentials, all diploma credentials can link to the same identity credential. Because the student's data is stored in one single identity credential instead of multiple diploma credentials, this eliminates the problem of inconsistencies of the personal information between different diplomas. Additionally, storing all personal data in a single ID credential may also open up new possibilities for update mechanisms in case the student's personal information changes. In section 8.4, we discuss a potential update concept based on linking different identity credentials to provide a changelog of personal information history. Lastly, the outsourcing of all identity information from the diploma into a separate credential also results in increased privacy. Some relying parties might only require a presenter's DID, while not being interested in additional personal data. In such cases, the student can choose to only present their diploma credential without the corresponding identity credential, and would not share any PII.

### Verification Logic

Since the Identity Credential Check requires cross-checking different credentials (at least two: one identity- and one diploma credential), it can only be performed if a VP is verified, but not if a standalone VC is verified. However, different to previous checks that also only happened during VP verification (such as the challenge check in section 4.6), the Identity Credential Check will be evaluated on a *per-credential*-basis, i.e., if and only if a VP is submitted, we perform the identity check individually for every of the presented credentials.

In order to keep the verifier backward compatible with VCs that do not conform to the DiBiHo specifications, such as VCs without an `identityCredentialHash`, we perform the following verification logic:

| Type | Error Code and Message |
|---|---|
| *Error* | `MULTIPLE_ID_CREDENTIALS`<br>*More than one identity credentials are provided. Currently, this verifier only supports a single identity credential per VP. If you have multiple identity credentials, please submit them in separate VPs.* |
| *Error* | `MISSING_ID_HASH`<br>*This credential requires an identity credential hash; none is provided.* |
| *Error* | `NO_ID_CREDENTIAL_SUBMITTED`<br>*This VC requires a matching identity credential, but none is provided in the VP.* |
| *Error* | `ID_CRED_MISMATCH`<br>*The provided identity credential does not match the specified hash.* |
| *Warning* | `NO_IDENTITY_INFORMATION_IN_VC`<br>*This VC does not contain a link to an identity credential. Therefore, the identity of this DID could not be confirmed. Please check this credential manually if it contains information that proves it belongs to the presenter.* |

Table 4.9.: Overview of Possible Results of the Identity Check

- An identity credential is mandatory for every credential of type `ElmoDiplomaCredential`. Furthermore, the relying party can also configure a list of additional credential types that must have an identity link.
- If the verified credential is of a non-mandatory type, but yet *does* include a property `identityCredentialHash`, the verifier **will also require** an identity credential. The intuition here is, that if an issuer explicitly added an ID hash to a credential, it implies that the hashed data is part of the credential and therefore, should be presented and verified.
- If either of the above two points holds we perform the identity check by evaluating the following: If an identity credential is contained in the VP *and* the identity credential itself is valid (e.g., is neither revoked nor has an invalid signature) *and* the identity credential's hash matches the `identityCredentialHash` property defined in the diploma credential – then the check is passed. Otherwise, the check is failed because the holder's identity could not be confirmed.
- If the credential to be verified does not contain an identity hash *and* it is not of a type that requires an identity check, the identity check will be **skipped**. However, we will display a warning to the relying party informing them that no identity information could be assigned to the presenter's DID. Further, we recommended the relying party to inspect the credential manually to check whether it contains any information that helps identifying the natural person behind the holder's DID.

An overview of the different errors and warnings resulting from the identity check is shown

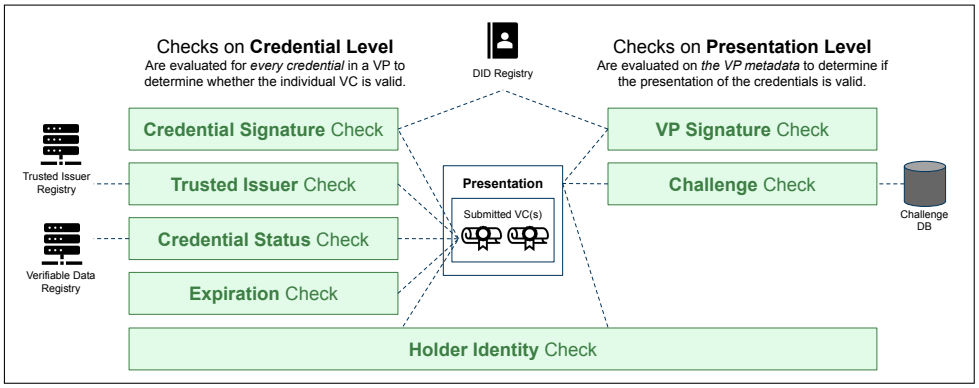


Figure 4.2.: Overview of the Performed Verification Checks

in table 4.9.

## 4.9. Conclusion and Overview

### Overview of Checks and Overall Validity

In the previous sections, we have discussed a set of different verification checks. Some of these checks happened on a *per-credential-level*, meaning they are performed for every credential in a VP individually to determine whether these VCs are valid. These per-credential-checks cover different aspects of the credential's lifecycle: if the entity who originally issued the credential is a legitimate institution; if the content has remained unchanged since then; if the credential has not been revoked in the meantime; and if the expiration date still isn't reached. Additionally, several checks are performed on the *presentation-level* to ensure that the presenter is the owner of the included credentials and intentionally presents them to the specific relying party. Additionally, we identify the natural person behind the presenter's DID by an identity credential, which must be referenced in all diplomas. Figure 4.2 displays an overview of the discussed verification checks. [1]

We consider a **standalone credential** as *valid*, if *all* checks that are performed exclusively on a credential level are passed, i.e., no check returned an error. Note that this excludes the holder identity check – as by verifying a standalone credential, we only know the holder's DID, but cannot make any assumptions about the controller behind it. Therefore, we consider the use case of *pure credential verification* as only suitable for testing and debugging purposes, but never for making decisions about a credential holder (e.g., accepting or rejecting an applicant), as we also need to check the credential holder's identity and DID ownership through a VP.

Therefore, we *always* require individuals to submit their credentials in form of a VP. We consider a **presented credential** as *valid*, if *all* checks that are performed on a credential level *and* all checks that are performed on a VP level are passed, i.e., no check returned an error.

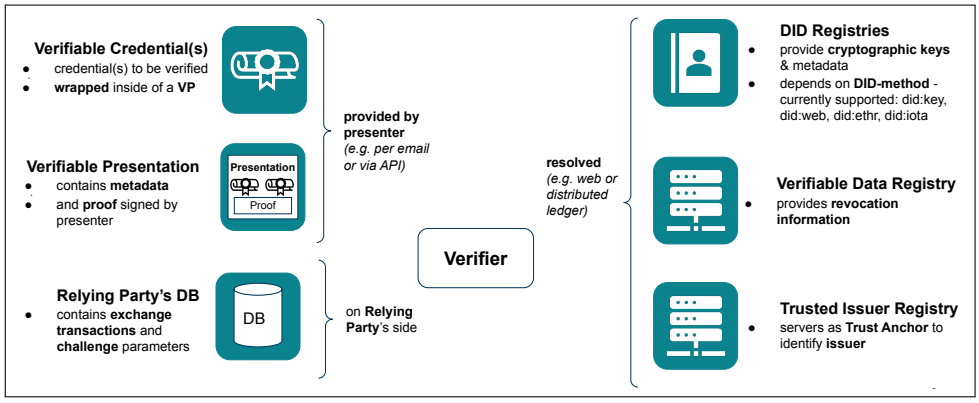


Figure 4.3.: Overview of the Data Sources Used During Verification

### Overview of Data Sources

In the previous sections, we discussed the different datasources that are needed for verification. An overview of them is shown in figure 4.3. We can conclude that an important and recurring concept was the support of different resolution methods or drivers, enabling the integration of different infrastructures, technologies, or implementations, which promotes interoperability and prevents lock-in-scenarios. This is one of the foundational ideas behind Self-Sovereign Identity, that we also leveraged for our implementation of the credential status and trusted issuer registries.

We also presented the idea of stateless challenge generation, which reduces the dependencies to additional infrastructure and can help making the verification and VP exchange setup easier and more accessible.

One aspect that we haven't addressed in the previous sections, is how the relying party obtains a VP from the presenter and how the interaction between the presenter's wallet and the relying party looks like, particularly concerning the exchange of the challenge parameter. While strictly speaking, this aspect is not a responsibility of the verifier service (which takes a given VP as input and returns the verification result), we consider it an important component of the overall DiBiHo project and will therefore further address this topic in section 7.1.

### Verification Check Order and Behavior at Failed Checks

The only part of research question 1 that we haven't addressed yet is whether there is a specific order in which the individual verification checks must be performed.

In principle, almost all of the above checks are independent from each other, i.e., do not require a certain outcome from another check. The only exception here is the Holder Identification Check, which requires the identity credential to be valid, and therefore should be evaluated last.

All the other checks can be evaluated in any order, or preferably in parallel. Since, as discussed above, the entire VC is considered invalid if a single check fails, the verifier can return early once any check yields an error, effectively skipping all the remaining checks.

However, in order to give the relying party a detailed overview about the entire verification status, we decided to not return early if a check fails. Instead, we always evaluate all the checks, and report the complete verification result to the relying party, providing a detailed analysis of the verification status, narrowed down to individual credentials and verification checks. In section 7, we will discuss further details regarding the API response format and the communication of the verification result to the user.

# 5. Analysis of Available SSI Libraries

The field of verifiable credentials and Self-Sovereign Identity is still relatively new. Many standards and specifications are still in an early draft mode and the issuance of official documents as VCs is currently only supported by very few organizations. Consequently, there are not yet any clearly established software libraries or frameworks for the issuing and verification of VCs. While in the developer community, VCs still represent a small niche area, the field has started to grow and more and more new open source projects are getting started. Because of the novelty of the topic and the highly dynamical environment, the range of available software frameworks that support the interaction with VCs is very vague and unclear.

To this end, we aim to analyze and evaluate what software frameworks in the context of SSI are available and if and how we can leverage them for the development of DiBiHo. We will begin this chapter by analyzing functional and non-functional requirements that a framework for our use case should support, particularly with an emphasize on the credential verification side. Then, in the second part, we will look at different available frameworks and evaluate how they fulfill these requirements.

## 5.1. Requirements

### 5.1.1. Functional Requirements

Most of the verification checks that we have discussed in chapter 4 extend the VC standard by custom components that we specifically designed to support the requirements of DiBiHo, such as the concept of identity credential linking, decentralized credential status checks, the stateless challenge validation, or the querying of a trusted issuer registry to identify issuer institutions.

However, as we explained earlier, especially the *Credential Proof Verification Check* is an entirely standardized verification check, as the proof verification is a core concept of VCs and is clearly described in the W3C specifications [15]. At the same time, it is the most complex verification check, as it requires low-level cryptographic operations to verify a signature's validity. For security reasons as well as for convenience we therefore aim to leverage a framework to perform the credential proof verification, which we consider as one of the core functional requirements a framework should fulfill. As discussed in section 4.2, the check can be split into two parts, **DID resolution** to obtain the proof signers public key material, and the actual **signature verification**. We will further analyze the functional requirements for these two aspects below.

### DID Resolution

DID Resolution [3] is one of the core concepts in SSI. It is easily explained: given a DID, resolve, i.e., retrieve the DID document corresponding to that identifier from the decentralized data registry in which it is stored. For degree credential verification, we need to resolve the university's DID as well as the learner's DID to retrieve their public key material against which we verify the signatures. Besides for signature verification, we also need to resolve DID documents to obtain further information, such as the endpoints of UniversalStatusLists (see section 4.4) or other services such as domain linking (see section 6.4.2).

What makes decentralized identifiers so powerful is the concept of different DID methods, allowing entities to choose what type of infrastructure, which specific network, and what implementation they want to use to store and manage their own DID documents. However, this variety of DID methods also makes the actual implementation of DID resolution a challenging problem.

A core requirement of DiBiHo is therefore, to support DID resolution for **multiple DID methods** in order to fully take advantage of the benefits that DIDs offer. Since there are specifications and drafts for well above a hundred different DID methods [26], an important question we were facing in the beginning of the project was which particular DID methods we should require our verifier to support.

Thereby, we came to the following conclusions: Mainly for testing and development purposes, we require support for **did:key** and **did:web**. The former is convenient as it does not require the hosting or querying of a distributed DID registry. The latter can easily be hosted on our project website `dibiho.org`, allowing for flexible modifications of the DID document as well as human-readable identifiers. It goes without saying that both DID methods should not be considered for real-world-usage, as they come with certain limitations as discussed earlier. Therefore, we also decided to support **did:ethr**, since it is a mature, fully decentralized DID method operating on the widely used Ethereum blockchain. An additional advantage is that the creation of `did:ethr` DIDs is free of gas and does not require a transaction to the blockchain, unless the DID document shall be modified. In order to demonstrate the concept of interoperability, we decided to include at least one more fully decentral DID method. Here, our choice fell on **did:iota**, as it caught our interest, since different to blockchain-based distributed ledgers, IOTA stores transactions on a directed acyclic graph for higher scalability and allows free updates of DID documents due to the lack of transaction fees. [61] We believe that these four described choices constitute a solid representation of different DID methods. As a minimum requirement for DID resolution, a framework should support these methods out-of-the-box or allow an integration of unsupported DID methods.

A process that is closely related to DID resolution is **DID URL Dereferencing**. Instead of just returning the DID document, it dereferences a DID URL (consisting of additional components like a DID path "`<DID>/...`", a DID query "`<DID>?...`", or a DID fragment "`<DID>#...`") into a specific resource within the DID document. [3] DID dereferencing is typically required to retrieve specific resources, such as a key that signed a credential proof. Therefore, it should also be supported by the framework. [3]

| Suite Name | Canonicalization Algorithm | Message Digest Algorithm | Signature Algorithm |
|---|---|---|---|
| `Ed25519Signature2018` [20] | RDF [14] | SHA-256 [12] | Ed25519 [16] |
| `EcdsaSecp256k1Signature2019` [18] | RDF [14] | SHA-256 [12] | ES256K [17] |
| `Ed25519Signature2020` [19] | RDF [14] | SHA-256 [12] | Ed25519 [16] |

Table 5.1.: Overview of several Crypto Suites for Signing and Verifying JSON-LD Proofs. Note that Ed25519Signature2018 and Ed25519Signature2020 use the same algorithms and mostly differ through the syntax of how the public key is presented.

#### Signature Verification for VCs and VPs

Besides DID Resolving, a core functional requirement for SSI frameworks to suit our needs is the support of signature verification. As we have discussed in section 2.1, the signature verification for JSON-LD documents (like VCs and VPs) typically consists of canonalizing the input document and feeding it through a hash function to obtain the *document digest* of a fixed size length, which is then fed into the actual verification algorithm, along with the provided signature.

What makes the signature verification challenging is the fact that many different signing algorithms and proof types exist. Therefore, the question arises which proof types and signature algorithms we should support. As discussed earlier, nowadays the Ed25519 algorithm is often preferred over ECDSA, due to its short key size and fast performance. However, the latter is in fact used by most public blockchains, including Bitcoin and Ethereum [62, 11]. Since `did:ethr` initializes each DID with a default ECDSA-secp256k1 key pair, we also want to support these key types so that they can be used for credential and presentation signing. We therefore conclude that an important functional requirement for signature verification is the support or easy integration of both, the **Ed25519** and **ECDSA-secp256k1** signature algorithms. An overview of commonly used signature suites for VCs is shown in table 5.1.

Besides the raw signature verification, several further steps need to happen to verify **Linked Data Signature Proofs** [15]: Such as calling the DID resolver to retrieve the required key material or validating that the public key's purpose matches the one in the proof. Furthermore, specifically for VCs and VPs, additional checks are needed, such as ensuring that the proof signer is the same as the credential holder. Because all these steps are critical to achieve a correct verification result, we prefer a framework that directly operates on a VC / VP level, i.e., supports signature verification specifically for **verifiable credentials** and **verifiable presentations**.

#### Desired Tooling Functionalities

While we mainly look for a framework to use for the implementation of the verifier service, ideally it should also support **signing** of credentials and particularly VPs. This is important beause currently, the Learner Credential Wallet App does not allow the generation of VPs for

specific challenge parameters. Therefore, we plan to provide users a CLI-based tool to create VPs.

Additionally, a few further non-trivial tools are needed as part of the verification: On the one hand, we need to **decode revocation bitstrings** that have been compressed according to the StatusList2021 specification. Additionally, for the stateless challenge verification, we require support for **creating and verifying JWTs**. While these functionalities are no must-have criteria for a selection of an appropriate framework, support of them would surely be a benefit.

### 5.1.2. Non-Functional Requirements

Besides our functional requirements, we define the following set of non-functional requirements that a framework should ideally fulfill to be suited for our implementation of the DiBiHo Verifier Service:

**Open Source** A mandatory requirement that a framework must meet is that it is provided as open source software. Because verification of degree credentials is a critical use case, it is of great importance that the user can inspect the entire source code if necessary, in order to understand how the verification is implemented. This requirement is also important to allow relying parties to run the verifier locally and customize it in case modifications to the source code are needed. The software should be offered under a license that permits modification and distribution free of charge, as well as private and commercial use. Commonly used licenses supporting these permissions are *Apache License 2.0*, *MIT License* and *BSD 3-Clause Revised License* [63].

**Programming Language** Since we develop our verifier service in **JavaScript / Node.js**, this is also the preferred programming language that a framework should support. As a minimum requirement, a framework should provide Node.js bindings in order to integrate it into the verifier service. Ideally, the framework should be written natively in JavaScript or TypeScript, as this opens the possibility to distribute the verifier in form of a mobile app (e.g., via React Native), a desktop app (e.g., via Electron) or as a pure front-end application.

**Extensibility** Another core requirement that is closely related to the functional requirements discussed above is the possibility of extending the framework. As interoperability is a key concept behind Self-Sovereign Identity, the framework should be flexible to **easily integrate additional DID methods and signature schemes**. Additionally, the framework should be flexible enough to integrate it into existing systems or combine it with additional verification checks, since as addressed above, our verifier service will need a set of customized verification procedures in order to satisfy DiBiHo's requirements.

**Documentation.** The quality of the documentation is an important aspect of a framework, as it significantly contributes to how fast developers can understand the framework and

use it to leverage their own development. The documentation should be up-to-date, complete, and easy to follow. Ideally, it should include code samples, tutorial guides, API references, and annotated function interfaces.

**Maturity.** We consider the maturity of a framework as the level of how well the software can be used in a production setting. This presumes the correct and robust implementation of all required functionality. It also includes how actively the project is developed and maintained and how fast bugs are reported and errors are fixed. While this criteria is hard to measure and requires comprehensive studying and testing of the software, specific metrics such as the number of stars on GitHub can indicate how widely a framework is used. Particularly the number and relevance of projects depending on that framework can help to derive an assessment.

## 5.2. Library Evaluation

Since we defined the open source availability as a core requirement for our frameworks, we primarily identified possible candidates by searching on GitHub for the keywords *Self-Sovereign Identity* and *Verifiable Credential(s)* as well as their respective acronyms. In this chapter, we will introduce seven different libraries and frameworks and discuss how they fulfill our previously defined requirements.

### 5.2.1. DIDKit

DIDKit (GitHub: *spruceid/didkit*) [64] is a cross-platform SSI toolkit that is written in Rust, but provides bindings for a set of platforms including Node.js, Python, Java, iOS, and Android. The developers explain their choice for Rust with its expressive type system and memory safety. In terms of functionality, DIDKit provides a wide set of features including issuance and verification of VCs and VPs, as well as key generation and DID creation, and support for both the JSON-LD and JWT proof format. Currently, a total of nine different DID methods are supported, including `did:web`, `did:ethr`, and `did:key`. DIDKit also provides a CLI and an HTTP API that follows the *VC-API* specifications. [35]

DIDKit particularly stands out for its simple set-up and use, as it works as a high-level "out-of-the-box" framework. It provides all required functionality in a single library with minimal installation and configuration efforts.

However, in our trials with the Node.js and WebAssembly bindings, the framework could not convince completely, as some of the advertised features were not working as expected. For example, while the documentation lists several supported proof types, including `EcdsaSecp256k1Signature2019`, it was unable to verify VCs with such signatures. The documentation also does not explain if or how support for additional proof types can be enabled. Because DIDKit is written in Rust, the debugging of framework-internal functions is significantly harder and the extensibility is limited if used in projects that are developed in JavaScript.

In conclusion, we currently consider DIDKit not mature enough, especially if used in an Node.js environment. However, the vision of offering an out-of-the box toolkit for all required functionality makes DIDKit a very interesting framework that might become a preferred choice in the future.

### 5.2.2. IOTA Identity

IOTA Identity (GitHub: *iotaledger/identity.rs*) [65] is an SSI library developed by the IOTA Foundation. With more than 240 stars on GitHub, it is one of the most popular SSI frameworks thanks to the popularity of the IOTA distributed ledger.

The framework primarily implements the `did:iota` method, supporting all four defined CRUD operations – generation, resolution, updating, and revocation. Different to most other ledger-specific DID method implementations, IOTA Identity is far more than a pure reference implementation. Additionally, it provides a rich set of SSI related functionalities, such as key generation and storage management, VC and VP signing and verification, all the way to revocation status checking using their own defined credentialStatus type `RevocationBitmap2022` [66]. Basically, this status method stores the revocation bitstring directly in the DID document instead of in a separate resource, since frequent updates to the DID document are no problem in IOTA thanks to the absence of transaction fees. Additional advanced features, as support for the DIDComm messaging protocol, are currently under development, and further privacy enhancing functionality like selective disclosure and zero knowledge proofs are planned for the future.

Similar to DIDKit, IOTA Identity is also implemented in Rust, but provides WebAssembly bindings that can be called from Node.js or TypeScript.While obviously, the framework is centered around IOTA, its core components (such as VC verification or DID management and storage) are agnostic about the distributed ledger technology, with the exception of higher level functionalities that provide convenience methods for IOTA integrations. In terms of DID resolution, `did:iota` is the only DID method provided by default, but other methods can be added by attaching custom handlers to the generic `Resolver` class. However, similar to DIDKit, the documentation does not specify how or if custom resolvers can be added in Node.js.

In conclusion, we think that IOTA Identity's main advantage is the great integration into the IOTA platform. While the framework can be used with other platforms as well to a certain degree, there is currently no significant benefit of using IOTA Identity over other more generic frameworks – especially if developing in a programing language other than Rust, as this leads to additional disadvantages as discussed in the previous section.

Nonetheless, we believe that IOTA Identity is a great example for a good integration of SSI into a general purpose distributed ledger technology like IOTA. With its concepts like fee-less transactions, there is great potential for novel SSI applications, and thanks to IOTA's relatively big user base, it may help to further push forward the adoption of SSI.

### 5.2.3. Universal Resolver (Docker-Based)

One of the big challenges of DID resolution is the high number and the variety of different DID methods. While most DID methods provide a reference implementation for their specific resolver, the interfaces between different method specific resolvers vary, and are often implemented on different tech stacks. To solve this, the Decentralized Identity Foundation has started a project called *Universal Resolver* (GitHub: *decentralized-identity/universal-resolver*) [67] - it's goal is to provide a single unified interface that can resolve any DID method.

The idea is, that the Universal Resolver provides a universal HTTP API, and routes each request to a method specific driver. Because as mentioned, the tech stacks between drivers can vary a lot, each driver is distributed in a docker image, allowing each method's developer to bundle all required dependencies into their container.

A driver must wrap their resolver into a simple HTTPI API that exposes the GET endpoint `http://<image-url>/1.0/identifiers/<your-did>`. The universal resolver is ran via Docker Compose, a tool that allows defining and running multi-container applications [68]. To add new drivers, developers can simply add the driver's Docker image into the `docker-compose.yml`, and update a central config file with the pattern of the DID method scheme, so that the correct container is queried if a DID of that method is getting resolved. New drivers can be added via pull request and currently there are drivers for more than 40 different DID methods collected.

A big advantage of the Universal Resolver is obviously the high number of supported DID methods. Thanks to the use of Docker containers, implementors have maximal flexibility over the underlying technology and the development of new drivers is quick and straight forward. Also the setup and integration of the Universal Resolver is simple, since everything is distributed via Docker containers and exposed by a single HTTP API.

In our tests, however, some of the available drivers were outdated, wrongly configured, or not working at all. The maturity of the Universal Resolver obviously depends on the specific driver implementations for each required DID method.

We believe that the Universal Resolver is a very promising project, since it is impossible for a singly developer team to integrate every existing DID. By collecting drivers to obtain a universal interface that can easily be integrated by developers, holders and issuers can choose from a wide range of different DID methods. This allows them to use a DID method that suits their needs best and take full advantage of the benefits that SSI offers.

A significant drawback of the Universal Resolver, that comes with its flexibility, is the additional overhead that results by running each driver in a Docker container. This makes it less convenient to run the resolver locally, due to higher performance requirements and longer image download and installation time. Therefore, the Universal Resolver is well suited to be operated by a dedicated trusted host, e.g., in a local network of a relying party organization. However, in the current stage of DiBiHo, our main goal is to minimize the access barriers to setup a verifier, and prefer to easily run a lightweight application locally.We therefore conclude, that while we do see a lot of potential in the Universal Resolver and a lot of possible applications for the future, it is less suited for DiBiHo, as we rather prefer a lightweight resolver mechanism – if it comes with the cost of a slightly lower number of supported DIDs,

this is acceptable.

### 5.2.4. DID Resolver (JavaScript-Based)

As an alternative to the previously described docker-based Universal DID Resolver, the Decentralized Identity Foundation also provides a *Universal Resolver for JavaScript Environments* (GitHub: *decentralized-identity/did-resolver*) [69].

This framework, which is written in TypeScript, defines a simple and lightweight interface to resolve DID documents from decentralized identifiers. The library itself does not implement any DID resolver implementations, but provides an interface that developers can use to release compatible drivers via npm. Any installed drivers can easily be registered with a generic resolver class provided by the framework, that also handles additional functionality like caching mechanisms if needed. All configured DID methods can then be resolved with a universal call to `resolver.resolve(did)`.

Therefore, the main difference to the previously discussed approach is, that here, all drivers need to be implemented in JavaScript/TypeScript instead of running in Docker containers. This makes the resolver significantly more lightweight and easier to integrate. Obviously, the higher requirements for drivers also lead to fewer available driver implementations. However, for both `did:web` and `did:ethr`, a resolver implementation for this interface already exists. For our other two required DID methods, `did:key` and `did:iota`, an implementation was not yet available. However, it was trivial to implement a driver ourselves by wrapping the reference implementation of did:key and did:iota into the required interface format.

An overview of these two different universal resolver architectures is shown in figure 5.1. While this JavaScript-based universal resolver approach supports less DID methods than the Docker-based resolver, we could easily integrate all our desired DID methods into one universal interface, and at the same time, have a lightweight resolver that can be directly imported into the Node.js verifier service. We therefore conclude that this approach is the best solution for DiBiHo. At this stage, it is not our goal to integrate as many DIDs as possible, but rather to have stable drivers for all supported DID methods and have an extensible interface that easily allows the integration of new drivers if needed. An additional option, that combines the best of both approaches, could be to query the docker-based universal resolver via HTTP request as a backup option whenever a given DID method is not supported by the local resolver.

### 5.2.5. Verifiable Credentials JS Library

The Verifiable Credentials JS Library (GitHub: *@digitalbazaar/vc*; in short: *vc-js*) [70] is a library for both, signing and verifying verifiable credentials as well as verifiable presentations. It is written in JavaScript and supports Node.js as well as browser runtimes. The library is developed and maintained by DigitalBazaar, the company that primarily contributed to the *VC Data Model*, *Decentralized Identifiers* and *JSON-LD* W3C standards.

Instead of being bundled as a single library, vc-js is provided as a set of different open source npm modules that work on different abstraction levels. As such, the pure *@digitalbazaar/vc*

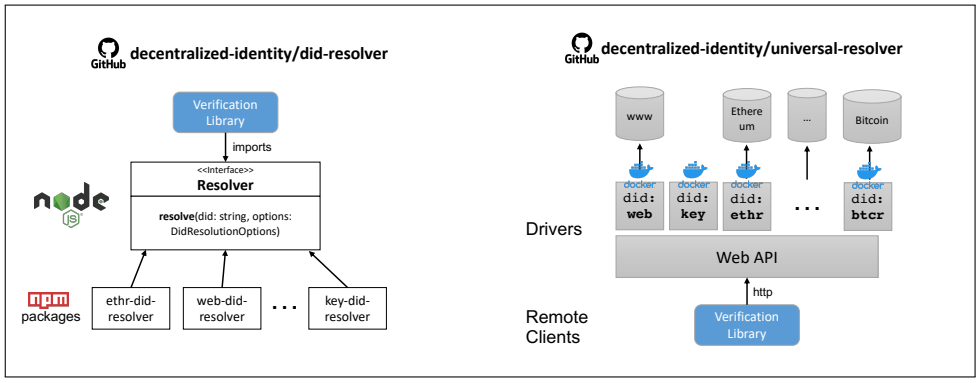


Figure 5.1.: Comparison of Different Universal Resolver Architectures

module does not contain any crypto suites or DID resolvers itself. Rather, the developer has to pass both to the respective sign or verify functions. To this end, DigitalBazaar offers separate npm modules for both, Ed25519 as well as ECDSA signatures. For DID resolving, the developer needs to define their own `DocumentLoader` function, which takes a given URI (e.g., an HTTPS URL or a DID) and returns the corresponding resource, by calling the respective HTTPS client or DID resolver and dereferencer.

To verify a credential, vc-js first performs a set of sanity checks, to confirm that the credential conforms to the VC data model. Then, for the actual signature verification, it calls the library *jsonld-signatures* (also developed by DigitalBazaar) which handles the more general verification of linked data proofs. An optional *checkStatus* function can be passed to vc-js for the case that the credential contains any status information.

Additionally, DigitalBazaar also provides an implementation of the StatusList2021 standard (that also has been primarily authored by DigitalBazaar), as well as a module specifically for encoding and decoding compressed revocation bitstrings (*@digitalbazaar/bitstring*).

In terms of extensibility, vc-js is a great option due to its modularized design. In particular, new DID resolvers can easily be added to the DocumentLoader function, and new crypto suites can be added for both signing and verification. For the downsides, vc-js is generally more complex to setup than the previously discussed libraries, as different modules need to be installed and the document loader must be defined. This means the developer is responsible for providing a DID resolver and dereferencer.

Overall, vc-js convinces through its range of functionality, the modularized design allowing for high extensibility, and the fact that it is developed by an organization that significantly contributes to several VC related standards. It is a mature framework that is used by many developers and several projects (including DCC's sign-and-verify project and the learner credential wallet). vc-js is also used in the implementation of DiBiHo's issuer service, which is an additional benefit, since using the same framework for issuing and verification ensures that both sides are always compatible and eliminates the problem of mismatching versions.

We conclude that the @digitalbazaar/vc framework meets our requirements well with regards to signature verification. As vc-js does not come with a DID resolver, we decided to

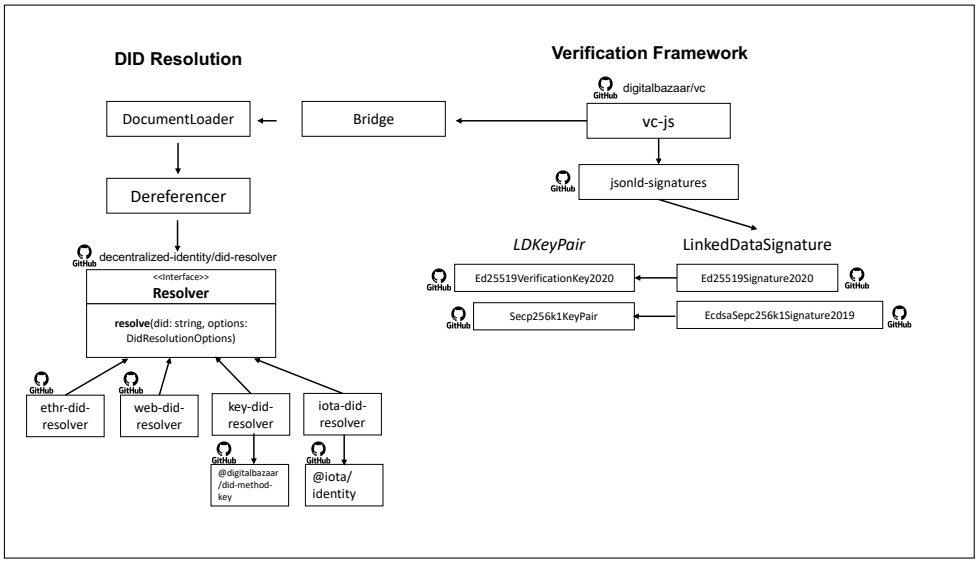


Figure 5.2.: Overview of Imported Modules for DID Resolution and Signature Verification

combine it with the JavaScript-based universal resolver interface discussed in section 5.2.4. By integrating it into a custom document loader, this results in a system that can deal with various DID methods and effectively verify signatures of different proof types. An overview of the installed modules interacting with each other is shown in figure 5.2.

### 5.2.6. DCC Sign and Verify Modules

The Digital Credentials Consortium has created a fork from the previously described *@digitalbazaar/vc* framework, as well as from some of its underlying modules like *@digitalbazaar/jsonld-signatures* [71]. According to the documentation in the repository, the reason for this fork was to provide compatibility for TypeScript and ReactNative. Besides these compatibility changes, there are currently no significant changes in terms of functionality in comparison to the original repository.

Since we develop our verifier in plain JavaScript and therefore do not need TypeScript support, and there are currently no major differences between both versions, we decided to stick with the original version of the framework – especially, since also DiBiHo's issuer service is based on the DigitalBazaar framework. The fact that DCC bases their development on a fork of vc-js proves that it is a powerful and mature framework. At the same time, it also shows how the space of available frameworks is constantly changing and developing. We recommend to monitor both versions of the fork for the case that bigger changes will be introduced; and if necessary, switch to the other branch.

Additionally, the DCC also maintains two further projects, *sign-and-verify-core* and *sign-and-verify*. The former works on top of their forked version of vc-js and adds resolvers for the DID methods `did:web` and `did:key`. Additionally, it checks if the issuer DID is contained in a static list of whitelisted issuers. The latter repository wraps an HTTP API around and

adds a database integration. For DiBiHo however, these repositories do not provide relevant additional features.

### 5.2.7. Veramo

Veramo (GitHub: *uport-project/veramo*) [72, 73] is a high-level SSI and verifiable data framework which is written in TypeScript and available to run in Node.js, in the browser, or on mobile applications via React Native.

Designed to be flexible and modular, it consists of a core framework (called Veramo *agent*) and several plugins. The agent is responsible for managing the different plugins and interactions between between them. Depending on which plugins are enabled, Veramo provides a wide set of functionality, including issuing and verifying of VCs and VPs; as well as creating and managing keys and DIDs. There also exist further plugins supporting additional functionality, such as message handling via DIDComm. While the plugin system provides a high feature range and excellent extensibility, it comes at the cost of ease of use, since the setup of enabled plugins needs to be configured and the documentation only provides limited guidance.

An interesting aspect is that for verification of JSON-LD-based VCs, which is handled by the `credential-ld` -plugin, Veramo uses *@digitalcredentials/vc*, i.e., the DCC's fork of *vc-js* (as Veramo also runs in TypeScript). This means, if used exclusively for verification purposes, Veramo does not add significant additional value over directly using vc-js.

We believe that Veramo is a very powerful and mature framework. It offers many benefits, such as multi-platform support for running it in browser or via React Native. Written in TypeScript, the code is clean and structured with well documented interfaces. It also contains useful tools to store and manage different keys and DIDs and provides an easy-to-use command line interface, which also makes it a good choice for end-users to manage their VCs. The plugin-system also makes Veramo very extensible and easy to customize. However, since under the hood, Veramo also essentially uses vc-js, it does not provide additional benefits for the purpose of verification. Therefore, we decide to stick with vc-js to prevent additional overhead of another framework layer. With a set of additional useful features, even though not relevant for our use case, we believe that Veramo can play a major role in the future for SSI frameworks.

## 5.3. Conclusion

In this chapter, we shed some light on the landscape of available SSI frameworks, by first analyzing different requirements that a framework should fulfill, followed by an analysis of available open source candidates - particularly with a perspective on the implementation of our DiBiHo verifier. We identified DID resolution and signature verification as core functional requirements – with a particular emphasis on extensibility of the framework, to support interoperability – a fundamental concept within SSI.

In total, we analyzed 7 different open source frameworks and identified their individual

| Framework | Supported DID-Method By Default | VC Verification | VC/VP Issuing | Supported Suites by Default | Ease of Use | Extensibility | License | Programming Language | GitHub Stars |
|---|---|---|---|---|---|---|---|---|---|
| **spruceid/didkit** | ethr, key, web + 6 more | Yes | Yes | ED | Very Easy | Limited | Apache License 2.0 | Rust (Node.js Bindings available) | 162 |
| **iotaledger/identity.rs** | iota | Yes | Yes | ED | Easy | Limited | Apache License 2.0 | Rust (Node.js Bindings available) | 243 |
| **decentralized-identity/ universal-resolver** | +40 DID Methods | No | No | None (Only Resolver) | Easy | Good | Apache License 2.0 | Java + Docker Containers | 392 |
| **decentralized-identity/ did-resolver** | None (Interface only) | No | No | None (Only Resolver) | Easy | Good | Apache License 2.0 | TypeScript | 162 |
| **digitalbazaar/vc** *("vc-js")* | None (requires document loader) | Yes | Yes | ED + EC | Requires Setup of docloader, submodules | Good (modularized) | BSD-3-Clause License | JavaScript | 106 |
| **digitalcredentials/sign-and-verify-core** | key, web | Via vc-js | Via vc-js | ED | Medium | Medium | MIT License | TypeScript | 1 |
| **uport-project/veramo** | ethr, web | Via vc-js | Via vc-js | ED | Requires plugin/agent configuration | Good (Plugin-System) | Apache License 2.0 | TypeScript | 248 |

Table 5.2.: Overview of the Analyzed Frameworks.

pros and cons. An overview of the different analyzed candidates is shown in table 5.2. The analyzed frameworks varied a lot in functionality – from some that were solely focussing on DID resolving, over pure sign and verifier libraries, all the the way to comprehensive SSI suites. In terms of verification, most frameworks are able to perform basic signature verification and run a set of checks to confirm the conformance to the VC data model. The abstraction levels differed a lot - from configuration heavy frameworks to others offering high-level convenience methods out of the box - but often at the cost of extensibility.

In summary, we concluded that *@digitalbazaar/vc-js* best fits our needs, as it reliably handles our core functional requirements, has great extensibility, and is developed by the contributors of the corresponding W3C standards. For DID resolving, the concept of a universal resolver is essential – since a single developer team will never be able to implement all possible DID methods. Therefore, establishing common interfaces for which implementors can provide drivers is crucial.

A drawback that all frameworks shared was the relatively weak documentation. In many cases, the documentation was incomplete and code examples often outdated. Obviously, this aspect is also related to the currently relatively small number of users and developers in this domain. However, all considered frameworks had their own strengths – there is no one-fits-all solution, even though, some frameworks were close to that and might become mature enough at some point in the future. We recommend closely monitoring the landscape of available SSI frameworks, as it is rapidly developing.

Because in our verifier use case, we need to call the SSI library only in few places in the code – primary for DID resolution and signature verification, all calls to the SSI framework can easily be abstracted and decoupled - which makes it easy to exchange the framework, if ever needed.

# 6. Trusted Issuer Registry

As introduced in section 4.3, a trusted issuer registry is a vital component of DiBiHo. In this section, we will analyze what requirements a trusted issuer registry should fulfill and explore and evaluate different approaches for the implementation and maintenance.

### Core Functionality

The overall goal of a trusted issuer registry [6] is to provide assurance that a given issuer DID is indeed associated with a legitimate higher education institution. Furthermore, it should provide a mapping from the issuer's DID to some form of registry entry data structure that provides information to uniquely identify the institution controlling the DID. This stored mapping information must be trustworthy as it serves the verifier as trust anchor to decide whether or not a given credential issued by a certain DID is considered valid.

Because the higher education sector is dynamic, since new institutions are getting established, and at the same time, certain DIDs might lose trust status (e.g., because their key got compromised), the trusted issuer registry should be dynamic as well – rather than being a simple static issuer list – and support new institutions being added or existing entries being updated or removed.

### Registry Maintainers

In the following, we will use the term *Registry Maintainer* for whoever entity that controls the information stored in a trusted issuer registry, i.e., decides what organizations shall be added or removed.

The maintainer could be a single individual, an organization, or a set of independent entities. If multiple maintainers control a registry, there may exist different permission concepts. For example, every maintainer could have full permission to add, update, or remove entries. Alternatively, some form of permission or consensus mechanism can be implemented to govern any changes to the registry data.

A core responsibility of a maintainer is to thoroughly examine any data entry for correctness before it is added to the registry. This includes a verification whether the DID to be added indeed belongs to the claimed institution, which can involve manual verification procedures. Essentially, the trusted issuer registry simplifies the complex task of verifying an issuer DID's identity for the relying party by shifting this responsibility to the registry maintainers.

## 6.1. Registry Design Analysis

We will start by discussing some general requirements and choices related to the high-level design of our trusted issuer registry.

### 6.1.1. Multiple Registry Instances

The Higher Education Institutions (HEIs) landscape is extremely heterogeneous: There exist many different types of HEIs – such as universities and universities of applied sciences (German: *Fachhochschulen*); distance learning universities; public and private universities; research institutes; community colleges; and many more. Consequently, many different degree types exist, and internationally, education systems may vary significantly between different countries.

Therefore, a single registry that contains every single HEI worldwide does not seem very feasible. It would be practically challenging to manage and maintain due to the amount and variability of institutions. Because registry maintainers need to decide about which organizations should be added, they typically require having some understanding of the higher education sector (e.g., to detect if a DID controller asking to be registered is a "fake university"). Also, different relying parties may have different policies about the requirements a DID controller needs to fulfill to be considered as a "Trusted Issuer".

For these reasons, instead of enforcing a concept of a single instance registry, we propose to relax this constraint and support multiple trusted issuer registry instances. If needed, different registries can co-exist and can be controlled by different maintainers, e.g., for different countries by national university associations, or consortiums like DCC.

To allow relying parties to meet their individual policies about what trust anchors should be used for verification, the relying party should be free to configure their own custom list of registries that they query for issuer identification.

### 6.1.2. Multiple Resolution Methods

An important question for the design of a trusted issuer registry that we already formulated as part of RQ2 is what type of infrastructure is suitable to store the data. While the storage must be trustworthy and openly accessible, multiple types of infrastructures and storage concepts come into question, such as blockchains, which can be divided into permissionless and permissioned blockchains – as well as distributed file systems or web-based systems. [74] All of these infrastructures have their own set of advantages and disadvantages, for example:

- Web-based systems are inexpensive to host and easy to maintain. However, they come with drawbacks, since servers can be down, are prone to several types of manipulations, and require trust in a single entity.
- Decentralized networks, such as public blockchains have the advantage that they are not dependent on a single entity. Since a network of independent nodes maintains their own copy of the ledger, the system does not rely on individual servers that

can be temporarily down or permanently unreachable. Smart contracts allow the implementation of sophisticated consensus or permission mechanisms. However, a big drawback here is that decentralized networks typically come with a price that needs to be paid for transactions in order to be validated and added to the ledger.

Because we support multiple registry instances, there will be multiple registry maintainers that may have different requirements on the registry's infrastructure. Therefore, we decided to follow a similar approach as discussed in chapter 4.4 and support multiple resolution drivers analogously to the concept of DID resolution. This allows maximum flexibility and supports different infrastructures, technologies, and implementations. For example, a registry of university issuers might be operated as a smart contract on the Ethereum blockchain, while a registry of issuers of MOOCs might be hosted on the web.

### 6.1.3. Hierarchical Structuring by Referencing Sub-Registries

Besides containing a set of trusted universities and their DIDs, it may be beneficial for registries to optionally contain a set of referenced "sub registries", i.e., other trusted issuer registries that are endorsed by the maintainers. This simplifies the maintenance and management of a trust network, as it allows hierarchical structuring and administration of registries:

A top-level registry could be maintained by an international organization like DCC. It could contain references to sub registries maintained by each country's national university association or state education authority. They could reference further trusted sub registries for different regions or higher education types.

This hierarchical structuring makes maintenance significantly more manageable. Each registry contains a smaller number of university entries, and its maintainers are familiar with the specific type of institution, country, or education system. During verification, the verifier can query the top-level registry, and if the queried DID is not included there, recursively query the sub-registries in a graph search fashion.

## 6.2. Trusted Issuer Registry Data Model

### 6.2.1. Information Needed for Issuer Identification

Another important question we want to address is what information should actually be stored in a trusted issuer registry. In its most minimal form, a trusted issuer registry can consist of a simple set of white-listed DIDs. The verifier performs a basic inclusion check to evaluate if a credential's issuer DID is contained in the registry, resulting in a boolean response from the registry.

However, due to the aforementioned variety of higher education institutions, a simple boolean information about the issuer's inclusion in the registry is not very expressive. Rather, the relying party should know exactly which higher education institution has issued the credential. Therefore, the trusted issuer registry should map an issuer DID to a registry entry data structure that contains information to uniquely identify the issuer organization. To this

end, we primarily require the institution's **name** which typically already uniquely identifies a university. However, for further disambiguation, we also require the institution's **location** and **URL** - similar to the DCC's trusted issuer registry MVP (which currently is a basic and unsigned JSON list hosted on GitHub) [75]. As discussed in section 4.3, the verifier will cross-check this information stored in the registry with the `issuer.name` property in the credential to prevent registered DID controllers from issuing degrees in the name of another institution. To support the visual identification, we also add a **hash of the issuer's logo image** to the registry and cross-check it accordingly.

Besides information to identify the university, we also evaluated further types of data that could be added to the registry. This included the storage of contact details (like the university's e-mail address or phone number) or styling data for dynamic rendering of credentials. However, we concluded that the trusted issuer registry should primarily focus on its key purpose of issuer identification. This is especially important since data like contact details or styling properties are subject to change and would require frequent updates to the trusted issuer registry. As the registry confirms that an issuer's DID and claimed identity are authentic, any further information can be self-administered by the university and could rather be stored on the university's website or directly in the DID document.

### 6.2.2. Issuer and Registry Authorization Concepts

The trusted issuer registry could also define authorizations that clearly describe what credentials the registered universities are allowed to issue. In the following, we will discuss if and how such authorizations should be stored in the trusted issuer registry.

#### Decoupling University Accreditations from the Trusted Issuer Registry

On a high level, authorizations could restrict the credential types that are accepted from a specific issuer. For example, an online course provider like OpenHPI could only be authorized to issue AchievementCredentials, but no ElmoDiplomaCredentials. This concept could also be extended to restrict specific properties of the credentials being issued, e.g., a university of applied sciences might not be allowed to issue PhD credentials – or to majors, e.g., a technical university might not be allowed to issue a Law degree. All of these rules could be implemented directly in the trusted issuer registry.

However, we believe that detailed semantic rules about university accreditations that define what types of degrees from a specific institution are accepted should *not* belong in the trusted issuer registry: The primary goal of the trusted issuer registry is to **identify** organizations based on their **DIDs**. This is already a complex problem, as it requires the maintainers to verify whether the DID to be added indeed belongs to the claimed institution.

Once a relying party knows who the organization is, they can decide how to acknowledge the degree from a certain university (e.g., "don't accept law degrees issued by TUM"). However, this is a different problem, completely unrelated to DIDs or VCs, that also applies to physical degrees, where a university can print a degree with arbitrary contents. Consequently, there already exist datasets (such as university rankings or accreditation lists) that solve this

problem, which have been created by organizations that focus on the task of *degree evaluation*. These datasources could be integrated by the relying party's business logic after the issuer's identity has been confirmed by the trusted issuer registry. A coupling of this task with DID Identification would add unnecessary complexity, compared to solving both problems independently. We conclude that detailed rules about what specific degrees are accepted from which universities are important – however, since this is unrelated to DIDs, it should not be placed into the trusted issuer registry.

#### Authorizations to Restrict Scopes of Sub-Registries

However, certainly there are also reasons that promote the definition of some form of authorization directly in the registry. This becomes particularly important if a registry is managed in a hierarchical fashion – for example, by endorsing sub registries, as discussed before. If all sub registries have full permission, a single malicious sub registry could add arbitrary DIDs to the list of trusted issuers that can issue accepted credentials without restrictions. Therefore, it is important to have a mechanism that allows to technically restrict the scope of lower links in the trust chain.

We believe that hierarchically structured registries will majorly be structured based on their geography (e.g. institutions in Germany) and on the type of higher education institution (e.g., universities vs. MOOC providers like OpenHPI). Therefore, we decided to specify the list of allowed top-level `credential.type` properties that an institution can issue (such as DiplomaCredential vs. AchievementCredential). Since we also store the location of each issuer, this lets us restrict the scope of sub registries to only add institutions of a specific type or regions.

We believe that this model is a good MVP towards a trusted issuer registry that captures the specifics of the higher education sector. Of course, more complex trust models can be established (for example, as EBSI demonstrates [48]). However, this is mostly a question of governance and is beyond the scope of this thesis.

## 6.3. Authentication of New Issuers to be Added to the Registry

In order to register a new university in the trusted issuer registry, the registry maintainer is responsible for making sure that this newly added DID is really controlled by the claimed institution. This is not a trivial task since it requires the maintainer to ensure that:

- The university requesting a DID registration is a **legitimate university** (i.e., no "fake university")
- And that the sender of the DID registration request can authenticate as a **representative of the university** who is authorized to submit a DID registration request on behalf of the university.

Ideally, the registry maintainer should have good knowledge of the local higher education sector to reliably identify a university. This is easiest if the maintainer already has an

established secure communication channel with the university (e.g., via a public key, but this may also include physical in-person contact with authorized university staff). This would be the case if a registry is maintained by a national university association as discussed earlier, which strengthens the point of hierarchical structuring.

If a registry maintainer does *not* have any existing connection to the organization to be potentially added, the authentication becomes more difficult. Ideally, the university should publish their DID on multiple verified channels (such as their website, and on verified social media channels) to prove that the DID belongs to them. In section 6.4, we will introduce how the concept of domain linking may be used to support the authentication of potentially new registry members.

It is ultimately up to the maintainer's judgment to assess if the information in the registration request is valid and should be added to the registry – which again, is mostly rather a question of governance and policy.

As there may be different standards or *trust levels*, depending on how well the DID owner can prove that they represent a given university, we also add the `trustLevel` property to each registry entry, which can be set for each issuer. The relying party might choose a required minimum trust level threshold in order to accept a credential.

In the following section, we will discuss approaches for linking domains to DIDs, which could be used by maintainers to confirm the authenticity of a DID that claims to represent a given university.

## 6.4. Domain Name Linking to Support Issuer Authentication

Among the best-known and most widely adopted digital identifiers are Fully Qualified Domain Names (FQDNs) [76]. Domain names are registered in the Domain Name System (DNS) which primarily serves to map human-readable domain names to IP addresses of web servers. In the last decades, due to the enormous rise of the internet, and supported by secure communication protocols like TLS, domain names became an established digital identifier to represent the real-world organizations behind these domains.

For example, everyone can easily tell that the domain `www.tum.de` refers to a German organization called TUM. Even if a someone (e.g., a relying party) does *not* know this specific web domain, they can easily find out to which institution it belongs and whether it is a legitimate university, e.g., by analyzing incoming references linking to this domain (e.g., from university ranking websites) or by leveraging methods like the Google Page Rank. In the following, we will discuss if and how we can leverage domain names, linked to DIDs, in order to identify issuers.

### 6.4.1. did:web

In contrast to most other DID methods, which typically have complex and cryptographic identifiers, did:web [28] identifiers correspond to human-readable domain names. Therefore, a simple approach to solving the problem of issuer identification (even without the need for

an issuer registry) could be to encourage the use of the did:web method, as it uniquely maps a DID to a website, from where the verifier can easily derive the legitimacy of the issuer.

To increase the level of security, only DID documents that are stored in the .well-known directory of the domain root should be accepted by the verifier (i.e., `did:web:TUM.de` but not `did:web:TUM.de:students:ge82sap`) to prevent users who are not the domain owner but have access to a sub-directory from issuing credentials in the domain's name.

However, while this approach is straightforward to implement, we strictly discourage the use of did:web for anything other than test and demo purposes. As discussed in section 2.3.2, they cannot be considered as truly decentralized identifiers, due to their dependency on being actively hosted on a running web server. Therefore, we refrain from giving did:web identifiers a special treatment in the verification process but explore a related but more flexible concept in the following subsection.

### 6.4.2. Well-Known DID Configuration

*Well-Known DID Configuration* is a draft for a standard proposed by the Digital Identity Foundation, allowing DID owners to prove that they are the same entity that also controls a domain [77].

It requires the domain owner to define a *DID Configuration Resource* – a file named `did-configuration.json` that must be located in the `/.well-known` directory [78] of their domain root. The corresponding URI for the DID Configuration Resource of TUM would look as below:

```
https://tum.de/.well-known/did-configuration.json
```

An example of such a DID Configuration Resource is shown in listing 6.1. It contains the property `linked_dids`, which is a list of any number of DIDs that are linked to the corresponding domain. Each linked DID is represented by a verifiable credential of type `DomainLinkageCredential`. Essentially, these are minimalistic VCs whose credential subject contains the **DID** (in the `id` field) and the **domain** (in the `origin` field, which must match the domain from which the resource was retrieved) that should be linked together. To prove the ownership of the domain owner over the DID, the credential contains a proof with a signature signed by the linked DID.

In order to enable the discovery of any linked domains from a given DID, the DID document can contain a service endpoint that references the linked domains, as shown in listing 6.2.

For a verifier to check if an issuer's DID is linked to a domain, they can simply resolve the issuer's DID document and look for any services of type `LinkedDomains`. If such a service is included (e.g., for the `serviceEndpoint: "https://tum.de"`), they fetch the DID Configuration Resource from the domain's well-known directory over a secure HTTPS connection via GET request. Then, the verifier checks if the configuration contains a `DomainLinkageCredential` that has been signed by the corresponding DID, which proves that the entities controlling the DID and the domain can be considered the same.

Essentially, this approach provides a simple and effective way to bind DIDs to domains. In

```
{
  "@context": "https://identity.foundation/.well-known/did-configuration/v1",
  "linked_dids": [
    {
      "@context": [
        "https://www.w3.org/2018/credentials/v1",
        "https://identity.foundation/.well-known/did-configuration/v1"
      ],
      "issuer": "did:example:z6MkoTHsgNNrby8JzCNQ1iRLyW5QQ6R8Xuu6AA8igGrMVPUM",
      "issuanceDate": "2020-12-04T14:08:28-06:00",
      "expirationDate": "2025-12-04T14:08:28-06:00",
      "type": [
        "VerifiableCredential",
        "DomainLinkageCredential"
      ],
      "credentialSubject": {
        "id": "did:example:abcde12345",
        "origin": "https://tum.de"
      },
      "proof": {
        "type": "Ed25519Signature2018",
        "created": "2020-12-04T20:08:28.540Z",
        "jws": "eyJhbGciOiJFZERTQSIsImI2NCI6ZmFsc2UsIm...J1iJlkSmB4JaDQ",
        "proofPurpose": "assertionMethod",
        "verificationMethod": "did:example:abcde12345#someKey"
      }
    },
    < Further Linked DIDs >
  ]
}
```

Listing 6.1: .well-known DID Configuration Resource

```
"service": [
   {
     "id":"did:example:abcde12345#institutionWebsite",
     "type": "LinkedDomains",
     "serviceEndpoint": "https://tum.de"
   }
 ]
```

Listing 6.2: Linked Domain Service in a DID Document

comparison to the previously discussed linking via the did:web method, any arbitrary DID methods can be used for this approach (except methods like did:key, which do not support adding services to a DID document).

### 6.4.3. TLS-endorsed DIDs

In a recent work on TLS-endorsed Smart Contracts (TeSC) [79], Gallerdörfer et al. propose a solution for the similar problem of linking Ethereum Smart Contract addresses to web domains. They leverage the established Transport Layer (TLS) / Secure Socket Layer (SSL) infrastructure and store signatures on-chain in smart contracts, which have been created by the domain owner using the private key of their TLS certificate. Verification happens off-chain by retrieving the domain's TLS certificate, verifying the signature in the smart contract, and recursively verifying the chain of Certificate Authority (CA) certificates until a trusted root certificate is reached.

A similar concept would also be conceivable for DIDs, e.g., by storing a domain along with a signature `sign(DID || domain)` in the DID document that is signed by the key of the domain's TLS certificate. The actual verification could happen in a similar fashion to TeSC.

### 6.4.4. Drawbacks of Domain-Based Approaches

The above two approaches have shown effective mechanisms to link DIDs to web domains. However, we want to highlight that an issuer verification purely based on domain linking is not sufficiently suited to provide relying parties assurance about the authenticity of diploma VCs.

First of all, anyone can register arbitrary domain names as long as they are still available, which leaves room for malicious parties to set up domains similar to famous universities in order to fool relying parties. Typically, such malicious domain names could contain small typos (like harward.com), a wrong but realistic domain name (like cambridgeuniversity.org), or simply a different top-level domain (like tum.org). This concept is also called *Typosquatting* [80] and can lead to serious consequences if a relying party believes that the typosquatted domain really represents the pretended institution.

Recent work has shown promising results on effective defense strategies, such as [81] for the context of TLS-endorsed smart contracts by training a learning-based model to detect typosquatted domain names. However, even with such defense strategies, relying solely on domain bindings poses security risks. Especially since it is also possible for a malicious party to set up a domain for a non-existing, completely imaginary fake university what a relying party might not notice. This amplifies the point that a trusted issuer registry should be curated by a responsible and trusted maintainer.

For the context of higher education domains, it should be noted that the dedicated top-level domain ".edu" exists, which can only be registered by accredited U.S.-based postsecondary institutions [82]. Therefore, if a VC can be linked to an .edu domain, it is safe to assume that the VC has been issued by a university. However, the ".edu" domain is currently only available to U.S.-based institutions and can not be used by foreign universities anymore. Additionally, as already discussed, all web-domain linking based approaches will always require that the web server is reachable. This is a strong assumption that not always holds, especially since one of DiBiHo's requirements is to verify credentials without the involvement of the issuer.

We conclude that domain-linking-based identification is a valuable approach that can help relying parties or registry maintainers to identify organizations behind DID controllers. However, as these approaches have drawbacks and still require manual judgment to map the domain to the real-world organization, we do not see them as an alternative to a trusted issuer registry. However, we believe that such approaches can be very helpful for registry maintainers to handle registration requests from new, unknown institutions by checking if their DID is linked to their respective website domain.

## 6.5. Consensus Concepts for Registries Maintained by Multiple Entities

In the previous sections, we have always assumed that the maintainer of the registry is a single entity that can autonomously decide what information should be added to the registry. However, giving complete trust to a single entity can be a risky trust model since the single individual may turn malicious at any time. Therefore, a trusted issuer registry should have a concept to support decision-making involving multiple independent entities.

In the following, we will discuss two different concepts that can be used to implement a consensus mechanism for registries that are maintained by *multiple* entities.

#### Proof Sets

So far, we have always considered VCs to contain exactly one `proof`, i.e., the signature from the issuer. However, a verifiable credential may also contain an array of several proof objects, which is called a *Proof Set* – it is useful whenever the same data needs to be signed by multiple entities – just like a set of signatures on a physical contract [15].

A verifiable credential with a proof set could therefore be used to represent a trusted issuer registry that is maintained by a set of independent entities (e.g., it could be maintained by the twelve founding members of the Digital Credentials Consortium). An example is shown in listing A.1. The `credentialSubject` stores the different registry entries for the set of registered issuers, and the `proof` property contains a list of all the required signatures from every single maintainer.

Such a registry credential with a proof set could even be hosted on the web while allowing verifiers to confirm that all maintainers approved the state of the registry. A relying party who decides to use this trusted issuer registry as a trust anchor would configure it accordingly, along with the required DIDs of the twelve consortium members. At verification, it will retrieve the newest version of the registry and simply check if all individual maintainers have signed the TrustedIssuerRegistryCredential with a valid signature – which means that all maintainers, i.e., consortium members, have approved this version of the trusted issuer registry.

#### Registries based on Smart Contracts

Obviously, the above approach of requiring an approval of *all* the maintainers is a very basic approach towards a consensus mechanism. It can work well in certain cases, but often, especially when independent parties maintain a registry, a complete consensus is not guaranteed.

Clearly, web-based resources are less suited to implement more sophisticated consensus mechanisms, especially since the entity that publishes the document to the web can decide what status of the registry is uploaded to the web server, and there is no clear notion of document history.

A much better approach towards consensus mechanisms is enabled by smart contracts – for example in the Ethereum network, which could be used to implement a decentralized trusted issuer registry. Smart contracts allow to store data on a blockchain and expose different functions which can be called by accounts in the Ethereum network in order to interact with the registry. In the following section, we will demonstrate how a smart contract can be used to implement a more elaborate form of consensus based on a voting mechanism.

## 6.6. MVP Implementation

For our prototype, we implemented two different registry resolution drivers to demonstrate how our strategy of preferring interoperability and encouraging flexibility over fixed tech-

```
struct IssuerEntry {
    string name;
    string url;
    Location location;
    string trustLevel;
    bytes32 logoImageHash;
    string[] allowedCredentialTypes;
}

struct SubRegistry {
    string identifier;
    bool canAddSubRegistries;
    string[] allowedLocations;
    string[] allowedCredentialTypes;
}
```

Listing 6.3: Registry Data Structure

```
interface TrustedIssuerRegistry  {
    event IssuerEntryChanges(string did, IssuerEntry issuerEntry);

    event SubRegistriesChanges(SubRegistry subRegistry, EventType eventType);

    function getIssuerEntry(string memory _issuerDid) external view returns (
      ↪ IssuerEntry memory issuerEntry);

    function getSubRegistries() external view returns (SubRegistry[] memory);
}
```

Listing 6.4: Trusted Issuer Registry Solidity Interface

nologies can be implemented in practice. One driver is based on a web-hosted registry VC, and the other is based on an Ethereum smart contract.

### 6.6.1. Web-Hosted Registry VC

The web-based resolution driver retrieves the registry VC from an HTTPS URL. It is wrapped into a VC of type `TrustedIssuerRegistryCredential` (as shown in listing A.1), secured by a signature to verify its integrity. Instead of one signature, a proof set with multiple signatures can be added, which makes this registry implementation suitable for organizations like DCC that consist of multiple entities but act as one consortium.

### 6.6.2. Ethereum Smart Contract Based Registry

In order to support a decentral trusted issuer registry, we also provide an MPV implementation for a trusted issuer registry based on an Ethereum smart contract. Listing 6.3 shows the structure of the `RegistryEntry` data model, and listing 6.4 shows the interface that registry smart contracts need to implement in order to be compatible with our verifier.

The data structure contains the earlier discussed properties to identify the real-world organization behind the issuer DID, i.e., name, URL, and location of the institution. It also contains a logo image hash, the trust level property, and a set of allowed credential types. The registry interface contains a function for checking if a given DID is contained in the registry, as well as a function to get the referenced sub registries.

We leave the exact implementation of the smart contract to the registry maintainer, such that they can decide what best suits their requirements. This includes choices about the exact data model that is used to manage the registry entries, any permission mechanisms about who can update the registry, and how the list of maintainers itself is administrated (e.g., if and how new maintainers can be added).

Whenever an issuer entry is added to the registry or updated, the smart contract should emit an **Ethereum event**. Ethereum events can be seen as publicly accessible logs that can be analyzed off-chain. This allows relying parties to track all changes to the registry, including timestamps indicating when issuers have been added or removed. It also provides

```
enum OperationType { Add, Remove }
enum PropertyType { IssuerEntry, Maintainer, SubRegistry }

struct Proposal {
    uint proposalId;
    uint votesCount;
    OperationType operationType;
    PropertyType propertyType;
}
```

Listing 6.5: Data Structure for Proposed Registry Changes

the possibility to subscribe to events and maintain a local copy of the registry, which can significantly improve the performance of registry lookups.

We implemented two different MVP smart contracts conforming to the above specifications: one that has direct setters for all data stored in the registry, protected by an `isOwner` modifier to only allow the authorized registry maintainer to perform changes to the registry data. In the other one, we demonstrate how a voting mechanism can be used to implement a trusted issuer registry in a smart contract that is maintained by multiple entities.

### Voting-Based Smart Contract

There are countless ways how a voting-based issuer registry could be implemented. As a comprehensive survey and analysis of different voting design schemes would exceed the scope of this thesis, here we focus on a simple MVP for a voting-based registry to demonstrate the idea behind this concept.

Our smart contract conforms to the interface described above, i.e., provides a mapping from a DID to issuer entries and stores a list of endorsed sub registries. Internally, our smart contract contains a list of maintainer addresses, which represent Ethereum accounts that administrate the registry, i.e., are authorized to cast votes regarding the data listed in the registry.

We implement a mechanism that every change to the data stored in the contract must be approved by the maintainers:

- This includes any type of change, i.e., **Adding**, **Updating**, or **Deleting** of data
- This also includes any type of data that is stored in the registry, i.e., **DID to Issuer Entry Mappings**, **Sub-Registries**, or the list of **Maintainers** (i.e., new maintainers can be added or maintainers can be kicked out of the registry)
- Any account in the Ethereum network can suggest a change to the registry. This allows institutions to "apply" themselves to be added to the registry (because of transaction fees in the Ethereum network, we do not expect a problem of spamming of proposed registry entries). Such a transaction will create a *Proposal* data object with a unique identifier for the specific operation type and property type, as defined in listing 6.5.
- **Only maintainers** can **vote** to approve a proposal by calling a function `vote(proposalid)`.

As voting logic, we use a simple thresholding logic:

- The threshold should not be too high (e.g., an approval should not be needed from all maintainers - because if there is a single inactive or malicious maintainer, no changes could be done to the registry anymore).
- The threshold should not be too low (e.g., an approval by a single entity should not be sufficient to make changes, since otherwise, a single malicious party could do harmful manipulations to the data).
- Therefore, we decide to employ the concept of majority voting, which means that **at least >50% of the maintainers** (for example, 7 out of 12) need to approve a transaction to the registry in order for the proposal to become effective, i.e., the data in the registry will be updated accordingly.
- All votes are recorded so that each maintainer can **only vote once** for a particular proposal.
- If a maintainer gets removed from the registry, all their **votes of proposals that are still pending will be removed**.

A complete listing of the MVP smart contract can be found in Appendix A. Obviously, many further mechanisms can be implemented in a smart contract. For example, certain registry maintainers could have different voting weights or there may be different voting thresholds depending on the proposed changes. We believe that smart contracts are a very powerful method to implement decentralized registries. This leaves space for future work and shows that the strategy of encouraging flexibility (by just requiring conformance to an interface) is important and opens much potential for different implementations.

## 6.7. Conclusion

For a trusted issuer registry specifically for the context of higher education institutions, we can make the following conclusions:

- The primary goal of the trusted issuer registry is to provide assurance that a given issuer DID is controlled by a **legitimate higher education institution** and to **reliably identify** the institution.
  - Therefore, the trusted issuer registry answers the question "**Who** issued a VC?"
  - It explicitly shall not answer the question "**How** do we acknowledge diplomas issued from that institution?" – This question is completely independent of DIDs and can be answered in the corresponding business logic once we identified the issuer.

- As the higher education landscape is complex, multiple registries and different resolution methods should be supported. The management of a registry can be facilitated by hierarchical structuring, based on geography and type of institutions or awarded degrees. These authorizations should be stored in the registry, to restrict the scope of sub registries.
- The establishing of a mapping from DIDs to real-world organizations is challenging. Maintainers need to verify if a DID is indeed controlled by the claimed organization. Approaches of linking DIDs to domain names can be supportive here, but manual judgement of the registry maintainers is always required.
- To increase trust the trust in a registry, it may be maintained by multiple independent entities, instead of a single controller. Smart contracts can be leveraged to implement comprehensive consensus mechanisms.
- Ultimately, all the addressed aspects – how a registry is structured, what data is stored, how new issuers are verified, and how consensus between different maintainers should happen – are mostly a question of policy and governance.
- For our prototype, we developed an MVP for a web-based registry, implemented as a signed credential; and for an Ethereum smart contract based registry that supports a majority voting mechanism.
- By only prescribing a smart contract interface, leaving the implementation up to the owner, and by supporting several infrastructure types and making the list of queried registries configurable for the verifier, we keep the trusted issuer registry flexible and configurable so that different relying parties and registry maintainers can use what works best for them.

# 7. Service Integration

So far, we have discussed the credential verification in detail and which checks need to be performed to determine the validity of a VC or VP. In this chapter, we will consider the verifier in a broader context and discuss how it can be integrated into existing systems. To this end, we will first explore how an exchange of VPs from a holder to the relying party can be established. Then, we will introduce our DiBiHo Verifier prototype and explain how it can be integrated by relying parties through different APIs.

## 7.1. Credential Exchange from the Wallet to the Verifier

While the previous chapter on credential verification assumed that the verifying party has already received the learner's credential(s) in the form of a VP, this chapter will address how this credential exchange between the learner and the verifier can happen.

There are different possible use cases that may require a VP exchange from a holder to another party. In this chapter, we will focus on the VP exchange as part of an online application use case, in which a credential holder applies to a relying party - e.g., for a study program, a job, or a visa. We assume that most applicants will fill out these online applications from their PC due to the convenience of having a larger screen and access to the keyboard. We also assume that most learners will store their credentials in a *wallet application*, which securely stores the private keys and facilitates the creation of cryptographic signatures to create VPs. Specifically, we assume that the wallet is a mobile application on the user's smartphone, which allows them to carry their digital credentials always with them, secured by biometrical security features of the phone. In our proof of concept, we primarily support an exchange with the Learner Credential App developed by DCC.

Summarized, the following entities are involved in a credential exchange:

- The **Credential Holder** is a person that owns one or multiple credentials, e.g., diploma credentials.
- The **Wallet** is a software application that the credential holder uses to store their credentials and private keys. We assume that the wallet is a mobile application installed on the credential holder's phone.
- The **Relying Party** is a third party, e.g., a company or university, that the credential holder interacts with.
- The **Application Portal** is a distributed system hosted by the relying party through which the credential holder interacts with the relying party, e.g., to apply for a job or

study program. We assume that most users interact with the application portal through a web browser on a PC.

- The **Exchange API Server** consists of a set of endpoints which we want to define in this chapter, that let the credential holder transfer their credentials to the relying party. The API server is hosted by the relying party, but it is neither directly a part of the application portal nor of the verifier service.
- The **Verifier Service** is a service that verifies given VPs, as discussed in the previous sections. We assume that the relying party operates an instance of the verifier service in their internal network.

In the following subsections, we will discuss what steps need to happen to transfer a VP from a holder to a relying party.

### 7.1.1. General Exchange Process

The credential exchange process not only includes the transportation of the credential from the user to the verifier. Additionally, the verifier needs to define the specifics of the required presentation (such as the challenge parameter and desired credential types), which the wallet then has to produce accordingly. There exist several specifications for protocols for the interaction with VC wallets for the use case of credential presentation to relying parties (e.g., [35, 83, 57, 84]). Many of these protocols share a similar pattern that we have abstracted here as a general exchange process (a sequence diagram is shown in figure 7.1), which we will explain in the following.

#### 1. Application Portal Generates QR Code to Start Exchange with Wallet App

The process sets in after a credential holder has begun their online application in the relying party's application portal. In order to initiate a credential exchange from the wallet, the wallet needs to obtain certain metadata about how it can reach the exchange API server. An efficient way to transmit this information from the application portal to the wallet is via a QR code that can be scanned by the phone's camera. The following information should be included in the QR code, as it is needed by the wallet in order to initiate the exchange:

- A URL that references the API endpoint for the exchange-initiation request.
- A unique identifier (such as a UUID, application ID, or transaction ID) that the API server can use to identify to which transaction or application the request coming from the wallet belongs. From now on, we will simply call this identifier *Exchange ID*.

The application portal generates a QR code encoding the above information, which can be scanned by the credential holder using their smartphone's camera. For a good user experience, the information in the QR code should be represented as a deep link that - once scanned - leads the user directly into the wallet app and starts the exchange. If the wallet is on the same

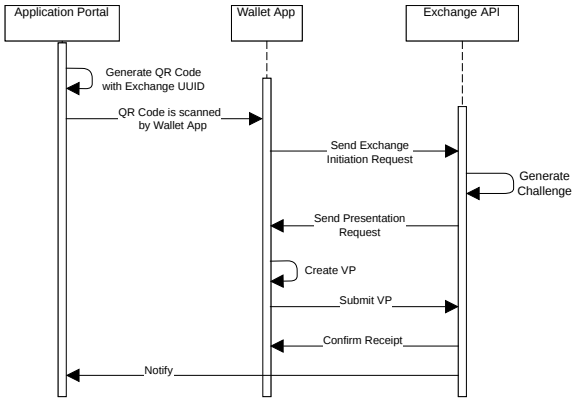


Figure 7.1.: Sequence Diagram of the General Credential Exchange Process

device that the applicant started the online application with (e.g., the learner has started the application from their cell phone's browser), a clickable deep link can be directly shown in the application portal that redirects to the wallet.

#### 2. Wallet App sends Exchange Initiation Request to the API Server

Once the wallet has received the information above, it will send a POST request to the provided endpoint and include the exchange ID in order to fetch further metadata for the exchange.

#### 3. API Server responds with Presentation Request

The API server receives the Exchange Initiation Request and will process it to ultimately respond with a **Presentation Request**, describing the specifics of the expected verifiable presentation.

First, it may also verify whether the provided exchange ID is valid and that the sender is authorized to present a verifiable presentation. Then, the API server generates a challenge parameter specific to the exchange ID. The challenge parameter can be generated in a stateless fashion, as described in chapter 4.6, or via random generation and stored to a database.

The presentation request can also contain further metadata that specifies what kinds of credentials should be shared and how the VP should be generated. This can include specific desired credential types (such as `DiplomaCredential` only), JSON-LD contexts, properties (e.g., only credentials that have a credentialStatus property), or property values (e.g., only degree credentials with a GPA of 3.0 or better) as well as proof types (e.g., only Ed25519Signature2020). To explain to the user why the credential is needed, the presentation request can also contain a description attribute. Lastly, the presentation request must contain a submission endpoint to which the wallet is supposed to submit the VP.

**4. Wallet App creates Verifiable Presentation and sends it to the API Server**

After the wallet app has received the presentation request, it guides the user through the creation of the verifiable presentation by displaying the query description from the presentation request and showing a list of all credentials that fulfill the requirements defined in the presentation request.

Once the user has selected all credentials they want to include in the submitted VP, the wallet app automatically constructs a verifiable presentation by including the challenge parameter and signing the VP using the private key. It then sends the created VP to the submission endpoint.

**5. API Server Processes VP and Confirms Receipt**

After the API Server has received the verifiable presentation, it may first validate whether the exchange ID exists and the sender is authorized to submit a VP.

It stores the submitted VP to the server and directly requests a verification of the submitted VP from the verification service.

Lastly, it sends an HTTP 200 status response to the wallet to confirm the receipt of the VP and optionally hits a notification endpoint from the application portal to inform it about the completion of the exchange process.

At this point, the VP exchange process is finished as the application portal has received the requested credentials and can continue with arbitrary business logic. In the following, we will look at some concrete specifications for data models and protocols that aim to standardize this process.

### 7.1.2. VC API and Verifiable Presentation Request

The *VC API* is a draft proposed by the W3C Credentials Community Group for a concrete HTTP API for credential exchange [35]. It contains the following two API endpoints:

- `POST: /exchanges/{exchange-id}` : **Initiate Exchange**
- `POST: /exchanges/{exchange-id}/{transaction-id}` : **Continue Exchange**

The **Initiate Exchange Request** is sent from the holder to notify the verifier about the intent to present a VP. The request body can be empty [35] or can contain a `credentialQuery` that describes the credentials the user wants to share [85]. The API responds with a **Verifiable Presentation Request** (as shown in listing 7.1) which describes what kinds of credentials and proofs are accepted by the verifier.

The verifiable presentation request contains one or multiple *queries* that define which credentials are accepted by the relying party. A query of type `DIDAuthentication` specifies which DID methods and crypto suites are accepted to be used by the holder to create a proof [85]. A query of type `QueryByExample` is used to describe what kind of credentials are accepted by setting constraints on specific credential properties, such as JSON-LD context or credential type. It also contains a description why the credential is needed, which the wallet

```
HTTP/1.1 201 Created

{
  "verifiablePresentationRequest": {
    "query": [{
        "type": "DidAuthentication",
        "acceptedMethods": [{"method": "key"}],
        "acceptedCryptosuites": [{"cryptosuite": "ecdsa-2022"}]
     }, {
        "type": "QueryByExample",
        "credentialQuery": {
          "reason": "Please present your degree credential.",
          "example": {
            "@context": [
              "https://www.w3.org/2018/credentials/v1",
              "https://www.w3.org/2018/credentials/examples/v1"
            ],
            "type": "UniversityDegreeCredential"
          }
        }
      }],
      "challenge": "3182bdea-63d9-11ea-b6de-3b7c1404d57f",
      "domain": "example.edu",
      "interact": {
        "service": [{
          "type": "UnmediatedHttpPresentationService2021",
          "serviceEndpoint": "https://example.edu/exchanges/refresh-degree/0bc42ece-89f7-11ec-9af0-136dfa9e4dbb"
        }]
      }
    }
  }
}
```

Listing 7.1: Example of a Verifiable Presentation Request

app can display for user guidance. The VP request also contains a challenge and an optional domain parameter to avoid replay attacks. Lastly, the `interact.service` property specifies how the wallet should respond to the VP Request by providing a transport protocol (e.g., HTTP, DIDCommv2, WACI, OpenID Connect, etc.) and a service endpoint.

Once the wallet has generated and signed the VP, it submits it to the **Continue Exchange** endpoint via post request. Finally, the verifier responds to confirm the receipt of the VC. The response may also contain the verification status [85], or - if applicable (such as in refresh credential use cases) another VP. [35]

Overall, the VC-API seems to be in an early draft stage and not yet completely thought through, as many aspects are not described in detail. Particularly, it does not explain how a mapping between different devices or sessions can be established, e.g., if and how a QR code should be used to allow initiation from another device. Also, in the official examples, the `exchange-id` is rather used to indicate the purpose of the exchange (e.g., `refreshCredential`) instead of identifying the browser session (as in their refreshVC example use case, no separate device or session is involved). However, by simply using the `exchange-id` property to uniquely identify the exchange session, we can make this API definition work for our use case.

### 7.1.3. Presentation Exchange

The *Decentralized Identity Foundation* has released a Working Group Draft called *Presentation Exchange 2.0.0* [83], which proposes a standard for the data model that is used in an exchange

```
{
 /* VP / OIDC / DIDComm / or CHAPI outer wrapper would be here. */
  "presentation_definition": {
    "id": "32f54163-7166-48f1-93d8-ff217bdb0653",
    "input_descriptors": [
      {
        "id": "wa_driver_license",
        "name": "Washington State Business License",
        "purpose": "We can only allow licensed Washington State
         business representatives into the WA Business Conference",
        "constraints": {
          "fields": [
            {
              "path": [
                "$.credentialSubject.dateOfBirth",
                "$.credentialSubject.birthDate"
              ]
            },
            {
              "path": [
                "$.type"
              ],
              "filter": {
                "type": "string",
                "pattern": "DegreeCredential"
              }
            }
          ]
        }
      }
    ]
  }
}
```

Listing 7.2: Example of a Presentation Definition

```
{
  "presentation_submission": {
      "id": "a30e3b91-fb77-4d22-95fa-871689c322e2",
      "definition_id": "32f54163-7166-48f1-93d8-ff217bdb0653",
      "descriptor_map": [
        {
          "id": "wa_driver_license",
          "format": "jwt_vc",
          "path": "$.verifiableCredential[0]"
        },
        {
          "id": "employment_input",
          "format": "ldp_vc",
          "path": "$.verifiableCredential[1]"
        }
      ]
    }
  }
}
```

Listing 7.3: Example of a Presentation Submission

between a credential holder and a verifier. While it does not define how an exchange is initiated and finalized, it focuses on the steps in between in the exchange process, where the most communication between both parties happens, as the verifier defines what information they want presented, and the holder responds accordingly. The following two data models are specified:

1. A *Presentation Definition* formulates the requirements for the VP and the proof that a verifier expects to accept a submission from the holder.
2. A *Presentation Submission* is used by the holder to submit their VP and explain how it meets the requirements previously defined by the verifier.

While the above data models correspond to the steps 3 and 4 of the general exchange process, the Presentation Exchange specification is *transport envelope agnostic*, which means that it does not define any concrete API endpoints or messaging protocols that describe how the messages are sent between the holder and verifier – it rather can be used with arbitrary messaging mechanisms, such as DIDComm or HTTP. Furthermore, it is also *claim and proof format agnostic*, which means that the *claims* do not need to be VCs but could also be represented in another JSON-based claim format, such as JWTs. [83]

An example of a **Presentation Definition** is shown in listing 7.2. It can contain one or multiple input descriptors that describe the information the verifier requires from the holder

(e.g., one input descriptor for a driver's license and one for a degree diploma). Each input descriptor is defined by an ID and contains a name and a purpose property explaining why the specific information is requested. Furthermore, an input descriptor can set one or multiple constraints on data fields that credentials need to fulfill to be accepted.

In their terminology, the authors differentiate between the terms *Presentation Definition* and *Presentation Request*: The former can be seen as a component of a *Presentation Request*, which describes the type of VCs and proofs a holder should submit. The term *Presentation Request* is used by the authors for the "transport mechanism used to send a Presentation Definition from a verifier to a holder" [83] and may also contain additional information (like a challenge parameter).

A **Presentation Submission** (an example is shown in listing 7.3) is used by the holder to specify how the VCs submitted to the verifier match the requirements defined by the verifier. A descriptor map associates each input descriptor from the presentation definition to a JSON path of the corresponding submitted VC. Once the verifier receives a presentation submission, they process it to evaluate whether the submitted input indeed satisfies all input descriptors.

In conclusion, compared to the verifiable presentation request draft discussed in the previous section, the Presentation Exchange offers more specific queries to specify the desired VP that a relying party expects and offers more structured support for querying multiple inputs. In addition, the Presentation Exchange format is used in several exchange protocols as explained in the next sections.

### 7.1.4. WACI

*Wallet And Credential Interactions (WACI)* [57] is a draft published by the Decentralized Identity Foundation, which aims to provide a standard for interactions between a wallet and a relying party. It covers interactions for the following two use cases:

- *Offer / Claim*: An issuer transfers a credential to a holder.
- *Request / Share*: A verifier asks a holder to share credential(s) with them.

The interactions for both use cases share a similar process:

1. To initiate a request, a *QR code* is generated, which contains a *challengeTokenUrl* from where the wallet can fetch further information.
2. The wallet scans the QR code and makes a *GET-Request* to the challengeTokenUrl, receiving back a JWT called *Challenge Token* which contains information about the requested VC(s) and a callback URL where they should be sent.
3. The wallet sends a JWT called *Response Token* via POST request to the callbackUrl.
4. The verifier sends a *Callback URL Response* with simple HTTP Success status. Optionally, the response can also contain a *redirect URL* or another challengeToken, in case a follow-up exchange is needed.

The structure of the WACI standard is very similar to the VC API, and the general exchange process explained earlier. A key difference is that the challengeToken and the responseToken are required to be JSON Web Tokens (JWTs).

Furthermore, for the use case of credential request and sharing, the challengeToken must contain a *Presentation Definition*, and the responseToken sent to the wallet must contain the VP with a *Presentation Submission*, as defined in the previously described *Presentation Exchange* standard [83].

### 7.1.5. WACI-DIDComm Interop Profile

The *WACI-DIDComm Interop Profile* [84] is a draft proposed by the Decentralized Identity Foundation and describes an interoperable protocol for issuing and presentation of verifiable credentials. Its overall structure follows the WACI standard we just discussed. However, instead of using HTTP, this specification suggests to communicate via the DIDComm v2.0 messaging protocol. It also uses the presentation definition and -submission data models defined in [83].

The following messages are sent:

1. `from: verifier | - ` **Out-of-Band Invitation Message** (via QR code):

   This message only contains a sender (the verifier's DID) but no receiver. As every DIDComm v2.0 message, it contains an ID. To allow mobile app wallets to receive this data, the message is encoded into a QR code that can be scanned by the credential holder – DIDComm v2.0 calls this an "out of band invitation message".

2. `from: holder | to: verifier` **Propose Presentation**:

   Next, the holder sends a regular, minimal DIDComm v2.0 message to the sender DID.

3. `from: verifier | to: holder` **Presentation Request**:

   The verifier responds with a presentation request containing the challenge parameter and a presentation definition.

4. `from: holder | to: verifier` **Present Proof**:

   The holder sends the VP to the verifier, wrapped in a presentation submission.

5. `from: verifier | to: holder` **Ack Presentation**:

   Lastly, the verifier responds with a simple acknowledgement message containing a status property with the 3 predefined possible values OK, FAIL, PENDING.

A drawback that comes with the communication via DIDComm v2.0 is that it introduces additional overhead for the relying parties, as they *must* possess a DID and additionally support DIDComm v2.0 messaging. While this exchange protocol might be interesting for the future, at the current time, when DIDs are not yet widely adopted, we conclude that it is not a suitable solution to our use case.

### 7.1.6. Credential Handler API (CHAPI)

The Credential Handler API is a protocol standard that web applications can use to register event handlers in the browser for handling credential storage or request events. This allows different web applications that support CHAPI to interact with each other by responding to another website's credential request/store events. For example, a wallet web app may respond to a `store()` event that was sent from an issuer web app; or to a `get()`, i.e., request event that was sent from a relying party's web app.

Once a web app emits a CHAPI event, the user will then be shown an in-browser dialogue with a list of previously enabled credential handlers that are compatible with the request. Once the user selects a handler, the event will be sent to it, and it will handle the query.

Developers can enable CHAPI by importing the *CHAPI Polyfill* JavaScript library into their web frontend, which will add a set of global variables to the code that can be used by the application to send `request()` or `store()` events, or to register handlers to handle them.

A relying party can request specific credentials by defining a verifiable presentation request (as in section 7.1.2) to query for specific types of credentials and provide a query message.

CHAPI has been proposed by the W3C Credentials Community Group from authors representing Digital Bazaar and is currently in a draft stage. While we find the concept interesting, it currently does not support having different devices for the application portal (i.e., verifier) and wallet application, respectively, and therefore is currently not compatible with our defined use case. [86, 87]

### 7.1.7. Conclusion

In this section, we discussed different protocols for verifiable presentation exchange between a credential holder and a relying party. While strictly speaking, this exchange procedure is not a part of the verification process itself and therefore was not included in our initial thesis outline and research questions, we realized during the thesis that the exchange between the holder and relying party plays a very important role for the overall project and therefore decided to also touch this topic.

We concluded that DIDComm-based exchange and CHAPI are not suited to our use case. The remaining discussed protocols – VC-API and WACI, are very similar in their overall structure. A key difference is that VC-API uses the W3C CCG's *Verifiable Presentation Request* format to specify the required VCs from the holder, while WACI follows the DIF's *Presentation Exchange* specification. Overall, the latter currently appears to be the more thought-through and clearer defined specification and provides means to more precisely specify the expected VPs.

However, for our use case of diploma presentation, most of the additional features of the Presentation Exchange specification are not needed, and the functionality offered by the VC API standard is completely sufficient. Additionally, the VC-API is more lightweight and less complex to evaluate, which makes an integration into the LCW app easier.

Ultimately, the choice of the exchange protocol(s) that should be implemented by a relying party mainly depends on what is supported by the wallet apps. Because we work closely

with the DCC, we decided to primarily focus on the Learner Credential Wallet. Currently, the LCW lacks any VP exchange related features, and therefore, neither provides the ability for a relying party to provide a challenge parameter or a credential query nor allows sending the generated VP to a pre-defined endpoint. The implementation of a presentation exchange feature for the LCW is planned, but development has not started.

As we are dependent on the DCC's LCW app in this regard, we decided to implement our MVP for a VP exchange feature by following the VC-API: We expect that the LCW will also follow the VC-API, since DCC members have actively contributed to the VC-API and verifiable presentation request drafts [85], and it is also close to the existing LCW protocol for credential takeout. Additionally, the VC-API standard is more lightweight and easier to implement for wallet developers and, at the same time, completely sufficient for our use case.

However, we consider WACI and Presentation Exchange as promising protocols that might be established as commonly used standards in the future. Ultimately, any acceptor of VCs should not be limited to a single exchange protocol but rather strive to offer the best wallet integration possible by supporting several of the most widely adopted exchange standards. For the verifier service itself, the choice of the exchange protocol does not matter, since it will always be called by the relying party *after* the VP has been received.

## 7.2. Prototype Development

Since DiBiHo's goal is to perform a proof of concept for VCs in higher education institutions, an important aspect of the project – and particularly of this thesis – is the development of a prototype. As core functionality, the verifier service prototype provides a method to verify a given VP by performing all the verification checks we discussed in chapter 4, including a check against a trusted issuer registry, as explained in chapter 6.

#### Core Library and Integrations Concept

We designed our DiBiHo Verifier as a core library that can be integrated through different APIs to meet the varying requirements of different relying parties and other stakeholders. We provide the following interfaces:

- A **Command Line Interface (CLI)**, which allows VP verification from the command line and scripting of automations. It is targeted to be used by advanced users and developers.
- An **HTTP API**, which can be used by relying parties to integrate the verifier functionality into their applications. It is targeted to be used by larger organizations that run their own internal instance of the verifier service.
- A **Web Frontend**, which provides a user interface for manual VP verification and inspection. It is targeted to be used by smaller relying party organizations that primarily follow manual application processes – as well as by any individuals who desire to verify given VPs.

Since the verifier service is developed in JavaScript, it might also be distributed as cross-platform desktop application using Electron or as iOS and Android mobile applications via React Native, which further simplifies the access to the verifier for individuals and relying parties. As the entire DiBiHo project code will be published as open source project, relying party organizations can also choose to integrate the DiBiHo verifier as an npm module or customize it for their own needs.

#### Tech Stack

For our development, we use the following tech stack:

- We use **Node.js / JavaScript** as our main programming language, as it is widely used in blockchain applications and can be run both in the frontend and backend, and additionally provides many possibilities for integrations, as discussed above.
- We developed the web frontend in **React** to allow for a potential mobile app distribution via React Native. As CSS framework, we used **Tailwind.js** due to its flexibility, allowing for highly customized user interfaces.
- For blockchain-related development and testing, we use the **Goerli** Ethereum testnet, as well as a local **Ganache** blockchain, as provided in the **Truffle Suite**. Smart contracts have been developed in **Solidity**, and the blockchain integration into the verifier app was implemented via **web3**.
- We use **Docker Containers** to simplify the distribution and setup of the project and use docker-compose to configure the interaction between multiple containers.
- We created unit tests using **Mocha** and **Chest**.

#### DiBiHo.org

To facilitate the development and to provide a publicly accessible environment that can be used for demos, we registered the domain `dibiho.org` under which we host several `did:web` documents and additional resources.

We set up the DID `did:web:dibiho.org:TUM`, which represents a demo credential issuer identity for TU Munich. Additionally, we created the imaginary student persona *Lucas Learner*, with the DID `did:web:dibiho.org:Lucas.Learner` acting as our demo credential holder (see figure B.2).

We also host a static web-based revocation list credential with a fixed set of revoked indices, following the `StatusList2021` specification. The DID document of our test issuer also references this revocation list in a `UniversalStatusList2022` service. Furthermore, we host a web-based trusted issuer registry credential, similarly as explained in section 6.6.1. Lastly, the website also contains some documentation materials, including sample credentials and sample keys as well as additional resources like logo images and custom JSON-LD context files.

#### Unit Tests

As an important part of our prototype development process, we set up several unit tests and API tests, covering most possible error, warning, and success cases for the different verification checks. Since many different factors can cause a credential to become invalid and especially signature verification checks are hard to debug manually, automated tests are extremely helpful to monitor the overall code functionality and help locate bugs in the code. Currently, we obtain 75.6% code coverage and plan to add additional unit tests in the future.

#### Development Methodology

Because of the close interplay with other components of the DiBiHo project, especially the issuer service and the wallet integration, we chose an iterative methodology for our development process, allowing to get regular feedback and react accordingly to changing priorities and updated requirements. We held weekly scrum meetings within the DiBiHo developer team, in which we demonstrated new features and discussed next steps and new requirements. We also gave several demonstrations of DiBiHo to potential stakeholders, such as the DCC, the Bavarian State Ministry of Digital Affairs, and the chief information officer of TU Munich.

## 7.3. HTTP API

As discussed above, an important deliverable of this thesis is an HTTP API that allows relying parties to easily integrate the verifier service into their systems. In order to simplify the usage of our HTTP API, we followed the specifications of the VC API [35]. For our verification services, it contains the following 2 endpoints:

- `POST /credentials/verify` to verify a VC
- `POST /presentations/verify` to verify a VP

The request body contains the VC or VP, respectively, and an `options` object which can include a challenge parameter for VP verification. For DiBiHo, we also add an additional option `expectGeneratedJwtChallenge` if the stateless challenge approach should be used instead of a fixed challenge comparison.

The response object `verificationResult` contains three properties: a list of the performed **checks**, a list of the occurred **warnings**, and a list of the thrown **errors**. While the VC API defines the values of these objects as *strings*, we do not consider simple strings expressive enough for the relying party to obtain a detailed understanding of the verification result. Therefore, we decided to extend the API by returning a more specific representation of the errors, warnings, and checks. Concretely, instead of strings, we return objects that contain a unique `code` that identifies the warning/error; as well as a human-readable `message`. Additionally, we include the properties `check` and `credentialIndex` to exactly indicate

in which specific verification check and for which credential in the VP an error or warning occurred.

In addition, our API provides the two endpoints for the **presentation exchange** as discussed in chapter 7.1.2:

- `GET /api/exchanges/<exchange-id>`, which is called by the wallet using the URL in the QR code and returns a VP request.
- `POST /api/exchanges/upload/<exchange-id>`, to where the wallet submits the generated VP, which is then verified and stored on the server.

We also offer an endpoint `GET /api/exchanges/:exchange-id` that can be used to retrieve the submitted data of a given exchange ID. This API endpoint is, however, rather intended for demo purposes since, in a real-world scenario, the submitted exchanges would be managed in another system (e.g., the application portal).

More detailed documentation about our HTTP API can be found in the DiBiHo Verifier GitLab repository.

## 7.4. Command Line Interface

We also provide a command line interface, primarily for advanced users who prefer a CLI over a graphical UI. Additionally, it enables developers to easily create automations, such as for batch verification purposes. Once downloaded and installed, the CLI can be queried with the command `$ dibiho`.

The list of available commands is shown in the shell outprint in listing B.1. Obviously, a main feature is the verification of VCs and VPs, which can be done using the `$ dibiho verifyVC` and `$ dibiho verifyVP` commands. The user can specify a challenge with the `--challenge <challenge param>` option, or use the `--expectGeneratedJwtChallenge` flag to verify if a valid stateless challenge is presented. The verification result is shown ordered by verification check and credential, with failed checks being highlighted and explained with a status message. An example of how the verification result is presented in the CLI can be seen in figure B.1.

Besides verification purposes, the DiBiHo CLI can also be used to sign presentations, which we currently consider an important feature since presentation exchange is still not supported in the LCW wallet app. If an exchange is initiated through our UI frontend, the user is shown a copy & paste command that can be run by the presenter to automatically generate a VP with the corresponding challenge. Additionally, the CLI provides convenience functions to generate a DID, which can be useful if the user doesn't own one yet.

For testing and development, the CLI also supports signing of credentials. Here, the issuer can optionally specify a `--identityCredentialPath` of an identity VC, which will be automatically hashed and linked in the VC. The generated credentials can either be printed to the console or directly stored in the local file system.

The CLI also provides further tools, such as for DID resolving or encoding and decoding of revocation bitstrings. For more detailed documentation of the individual commands and their options, we refer to the DiBiHo Verifier GitLab repository.

## 7.5. User Interface

While we assume that most larger relying party organizations will use technical interfaces, such as the HTTP API, CLI, or NPM Module, to integrate the verifier into their existing systems, we also consider a front end web UI an important component of our prototype. Especially for smaller relying party organizations that manage job applications manually via traditional e-mail communication, a manual user interface will be helpful. They can simply ask candidates to attach a VP via e-mail. The verifier frontend can be downloaded and installed by any individual who requires to verify and inspect degree credentials – including students who want to explore their own credentials.

The web frontend's main functionality is to provide a UI for the verification of verifiable *presentations*. While we also developed a feature for standalone VC verification, this is only meant for debugging- and testing purposes but never intended to verify an applicant's credentials, as they should always be submitted in a VP to ensure verification of the presenter's identity. An important requirement for our user interface is to make VP verification as simple as possible, to make it accessible to every relying party without any need for technical knowledge.

#### Input UI

To enter a given VP into the verifier app, the UI contains a drag-and-drop field that allows the user to simply drag the VP file into the UI from their local file system or an applicant's e-mail attachment. Alternatively, we also provide the option to manually copy & paste a VP as JSON string – but we assume that in most cases, VPs will be shared as files.

The user can also manually enter the challenge that was given to the presenter. If integrated into an online application system, the requested challenge would rather be loaded automatically from the internal database. As an alternative to manual challenge input, the user can also select a checkbox to expect a stateless JWT challenge, which can be verified without any data lookup, which demonstrates the benefits of the stateless challenge approach. A screenshot of the VP input UI is shown in figure B.3.

#### Communication of the Verification Result

After entering the VP, the UI automatically calls the verification API and then presents the verification result to the user. We consider the presentation and communication of the verification result one of the most vital components of the UI since it is used by the relying party to make impactful decisions, such as accepting or rejecting a job candidate. Therefore, to make the verification result easy to interpret, we use the following logic:

If all checks on a VP level have passed, we will show some metadata about the VP, such as the presenter's DID and a list of all the contained VCs with their individual verification status – *valid*, *invalid*, or finished with *warnings*. If, however, any VP checks have *not* passed (e.g., because the challenge is invalid or the VP signature is invalid), then we consider the presentation invalid and will not show any credentials at all, regardless of the individual credential's status – since even a valid credential has no meaning if the presenter could not be authenticated as its owner. Screenshots of the verification result overview are shown in figure B.4.

#### Credential Verification Result - Detail View

Each included VC can be expanded to see details about the credential's content and verification status. One of the most important components of the detail view is a **Preview Rendering** of the credential, which displays the credential's most important properties in a human-readable way inspired by traditional certificates. This allows the relying party to quickly and easily find all relevant information, which they would look for in a physical degree, and strengthens the impression of dealing with an actual degree credential instead of just cryptographic JSON files. We believe this contributes significantly to the user experience and will help strengthen the adoption of VCs.

Different renderers can be applied to appropriately visualize different credential types – for example, by displaying logos, colors, credential properties, and text blocks. We currently support rendering for TUM's `ElmoDiplomaCredential` and `ElmoIdentityCredential` as well as HPI's `AchievementCredential`. New renderers can be added to the frontend code. Screenshots of the verification result detail view with different renderers are shown in figure B.5.

We also show important **metadata** about the VC and the verification result. This includes information such as the credential's ID, the issuance timestamp, and the holder's and issuer's DID. Since we analyzed in the previous chapters that the mapping of these DIDs to their corresponding real-world-entities is an important aspect, we show a *REGISTERED* checkmark containing the information retrieved from the trusted issuer registry below the issuer DID, identifying the institution that issued the VC. Analogously, we show a *LINKED ID* check mark below the holder DID that contains identifying personal information if a valid identity credential was linked to the VC.

For the case that a relying party still requires degree documents in PDF format, e.g., for archiving purposes, we also implemented a dedicated **Print and Export Mode** that allows to export the verification report for a given VP in PDF format. It consists of a cover page with the verification overview and an additional page for each individual VC result. Most importantly, it contains a verification timestamp and a disclaimer on every page, reminding the reader that this PDF file is only a snapshot of the verification status at the time of verification. For decision-making-purposes, the VP should always be verified through the DiBiHo verifier, as the verification result may have changed since the printing of the report. Screenshots of the exported verification report are shown in figure B.9.

An important consideration of the result UI is that we only show the above credential

information if the VC is *valid*. If any of the verification checks returned an error, we will neither show a preview nor any credential data to clearly illustrate that the VC is invalid and should not be trusted. Instead, we will show the list of all happened errors and potential warnings. This way, the user can easily see why a specific credential has been classified as valid or invalid. This information can help the presenter to act accordingly in order to obtain a valid VP.

If the VC checks finished without an error but with at least one warning, the overall verification status of the VC will be "*finished with warnings*". This doesn't mean that the credential can be clearly classified as valid or invalid but rather indicates to the user a reason why the warning was thrown and recommends how to proceed to determine whether a credential is valid. A common reason for a warning is that no ID credential has been linked to a VC. Since this means that no personal information can be associated with the holder's DID, the relying party is instructed to manually check the credential content to see if it contains any information describing the holder. To this end, the credential can be shown in *raw JSON mode* to be inspected directly from the UI. This raw inspection can also be helpful in the case that a credential of an unknown type is verified, for which no renderer is pre-installed. In this case, a generic *default credential renderer* will be shown, and the user can further explore the raw data. Screenshots for this functionality are shown in B.6.

To provide interested users with more detailed information about the result of the individual verification checks, we show a verification overview with the results grouped by the performed checks for each credential. However, we primarily consider this a testing- and debugging feature for advanced users and developers and may deactivate it by default, as relying parties will typically only care about the overall status and resulting warnings or errors.

As for future improvements, we think there is much potential to display additional credential properties (e.g., the master's thesis title) in a structured way and add further verification metadata (like the complete signature and key type).

#### Exchange API Demo

To demonstrate how a credential exchange may look like, we also implemented a simulated *Application Portal* with VP exchange functionality. Therefore, this part of our prototype is targeted at the *presenter*, e.g., a job applicant. After clicking on an *apply* button, a unique exchange ID gets generated. While in a real-world application, this would happen in a separate system, we simply generate the ID in the frontend to simulate an exchange ID that would come from the job application management system.

The user gets shown a QR code for their specific exchange ID that they can scan with the wallet app to initiate and perform the exchange, as discussed in the previous section. Alternatively, instructions for manual submission can be displayed, which contain a copy-paste command for the DiBiHo CLI, which creates a VP for the unique challenge parameter and sends it to the submission endpoint for the corresponding exchange ID. Screenshots of the submission instructions are shown in figure B.7.

Once the credential holder submits the VP, it will be verified and stored in the file system of the verifier server (in a real-world system, the application portal would be notified with the

verification result and proceed accordingly). The relying party can access the result through the UI by the corresponding exchange ID or view a list of all submitted VPs, including their verification status (screenshots are shown in figure B.8). In this overview list, arbitrary data from the credential could be shown (like the holder DID, grade, or university) and also filter mechanisms could be applied, which demonstrates what new levels of automatization are enabled by digital verifiable credentials.

# 8. Evaluation and Future Work

In the previous chapters, we discussed the verification of VCs for higher education institutions and presented our DiBiHo verifier prototype. Our final solution fulfills all five DiBiHo requirements for the relying party service that we discussed in the introduction [7]: It allows relying parties to verify arbitrary credentials created with the issuer prototypes via TUM Online and OpenHPI to determine whether their contents are trustworthy (*DiBiHo-13*). The verification happens without requiring direct involvement from the issuer (*DiBiHo-55*) or other parties. The issuer is authenticated (*DiBiHo-14*) by querying a trusted issuer registry, and the learner is authenticated (*DiBiHo-24*) as the DID owner through a challenge-response-check and personally identified through the linked identity credential. Additionally, the verifier service can easily be integrated into the relying party's existing systems (*DiBiHo-25)* through different interfaces. In the following sections, we will discuss further specific aspects, such as performance, privacy, and security considerations, and suggest different directions for future work.

## 8.1. Performance Considerations

Since many relying parties are large organizations that have to handle massive amounts of online applications, the performance of the verification process is an important factor to consider for the integration into real-world systems.

As the verifier has to make several asynchronous requests to fetch remote information from the web and various verifiable data registries, the duration of a single VP verification depends largely on how long it takes to resolve all these requests. This varies a lot depending on the concrete did:method or storage type that is used. While resolving one of our *did:web* demo identifiers (hosted on GitHub Pages) took around *150ms*, resolving a *did:ethr* through an Infura node (both mainnet and goerli testnet) took significantly longer with an average of *1700ms*. A *did:iota* typically needed *350ms* to resolve, and obviously, *did:key* resolving was by far the fastest with only *3ms* since no asynchronous request at all was needed. A key factor that helps improve the performance and verification time greatly is leveraging **caching** to reduce latency. In particular, we can apply it for fetching the following documents:

**JSON-LD Context Files:** The context files can be cached offline permanently after downloading them once because they are meant to be static and do not update (in case a context gets a new version, it will typically be published under a new, versioned URL). Since certain contexts are contained in many credentials, caching them can contribute to decreasing the response time.

**Trusted Issuer Registry:** The trusted issuer registry can also easily be cached because the same registries are queried for every credential. The set of registry URLs is limited and defined by the relying party. Because the registry entries may change - even though not too often - a limited cache time should be configured, depending on the relying party's preference (e.g., an hour). For batch verifications, we simply fetch the current version of the trusted issuer registry in the beginning and keep it in the cache until the batch is processed.

**Credential Status Registries:** Analogously, retrieved status lists can be cached for a limited time, as each list contains status information for more than 100,000 credentials. This can save some unnecessary requests if a previous credential from the same status list was already verified shortly ago. Issuer DIDs and public keys could also be cached for a limited amount of time.

Obviously, caching of holder DIDs does not make sense since typically, all presenters have different DIDs. Therefore, the resolution of the holder DID heavily influences the overall verification duration. Particularly for storage methods that allow subscribing to update events, like a trusted issuer registry operated on Ethereum, the relying party can also subscribe to these events to maintain a local in-memory model of registry data that is always up-to-date and minimizes the need for requests to the blockchain.

To understand how computationally demanding the verification process is, we set up a script that first generates a batch of mock data and then verifies the set of all generated VPs. Concretely, the data generation script runs for `N` loops and generates the following: 1.) an issuer DID, 2.) a holder DID, 3.) an ID credential for the holder, signed by the issuer; 4.) a diploma credential for the holder, linked to the ID credential and signed by the issuer; and 5.) a VP, containing the ID credential and diploma credential, signed by the holder.

To measure pure computation time, independent of any latency and wait time for asynchronous HTTP requests, we used the did:key method and enabled a cache for the document loader for the duration of the batch processing.

On a typical consumer computer (MacBook Pro 2021, Apple M1 Pro Chip), the generation of 10,000 sets of mock data took us 2:43 minutes. The verification of all 10,000 generated VPs took us 6:46 minutes – which is a very satisfactory result, especially assuming that TUM receives fewer applications than that per day.

Of course, using DIDs other than did:key will add a delay to each request caused by the asynchronous waiting for the resolution of the documents, as discussed. However, since multiple requests can be processed in parallel and the overall computational effort for verification is relatively low, we conclude that performance-wise we do not see any challenges.

## 8.2. Privacy Considerations

While decentralized identifiers and verifiable credentials enable the digital processing of more and more important personal data that was previously processed manually and offline, this

also raises the importance of privacy. Our described prototype already offers several means of privacy protection, such as the concept of credential verification without the involvement of any third party, the holder's control over when and who they share credentials with, and the removal of all PII from the diploma credential by storing it in a separate identity credential. While this is a good starting point, SSI and verifiable credentials offer even more potential for further privacy-enhancing technologies.

A particularly important aspect that is often associated with privacy in data processing is the concept of **Data Minimization**, which means that only the amount of data should be shared that is necessary in order to reach a certain purpose. [88] A concrete method to achieve data minimization is through the concept of **Selective Disclosure**. It means that instead of sharing all information from a credential with the relying party, the holder can actively choose which attributes they want to disclose. There are several methods to implement such a concept:

**Atomic Credentials** means that an issuer does not only provide the holder a single credential containing all the data but instead issues several "atomic" credentials that only contain one or very few attributes of the claim (e.g., one VC just containing the grade, another VC just containing the degree type, and one more VC just containing the major). The holder can then simply choose which credentials they combine in a VP shared with the issuer. All credentials referring to one diploma must share a common identifier so that it is not possible to combine several atomic credentials from different diplomas. The advantage of this approach is that it is trivial to implement. In fact, both our issuer service and the verifier can easily deal with such atomic credentials: Essentially, these are normal VCs, only containing fewer data. The downside is that it might become cumbersome for holders to manage multiple credentials belonging to the same degree unless the wallet app provides dedicated features for managing atomic credentials. [89]

**Hashed Values:** Another straightforward approach towards selective disclosure is to issue all data in a single credential, but instead of stating confidential properties (such as the grade) in plain text, the credential only contains a hash of the value in combination with a secret nonce, which effectively "hides" the actual values. The holder possesses the plain values, including their corresponding nonces, and can choose which plain text values they provide to the relying party, along with the VC. The relying party can perform a default credential verification and then additionally check whether the provided values match their hashes in the signed VC. This additional check could either happen outside the core verifier module or be integrated directly. While such an implementation is straightforward, there are currently no established standards that specify formats and data models for this type of selective disclosure. [89]

**Selective Disclosure Signatures:** As another option, specific signature algorithms natively support selective disclosure, such as the *Camenisch-Lysyanskaya Signature*, which means that only a subset of values can be disclosed to the relying party without invalidating the signature. [89]

Another related field of privacy-enhancing methods are **Zero-Knowledge-Proofs (ZKPs)**, which allow the holder to prove the existence of a certain value without actually disclosing it to the relying party. ZPKs allows the holder to create *Predicate Proofs* that may answer a relying party's true-or-false-question, such as "Is the holder older than 18?". Using ZPKs, the holder could create a VP that proves the answer to this question without revealing their birthdate. [89]

## 8.3. Security Considerations

Because verifiable credentials may be used by relying parties to make impactful decisions, such as accepting or rejecting a job candidate, the security of the overall system is crucial.

Therefore, we highly recommend an in-depth security audit for the entire system (including the core verifier and issuer services as well as their integrating systems), performed by an external and specialized provider, in order to identify potential security vulnerabilities. That being said, in the following, we will address some components of the system that we believe can pose certain risks in terms of security.

#### Issuer's Key Security and Need for Issuance Logging and Monitoring

One of the main factors for the security of the overall system is the security of the private keys since if any malicious party obtains access to the key, they can issue arbitrary valid credentials. Therefore, it is extremely important to securely store the private key material and only give authorized personnel access to it.

In the unfortunate case that a private key gets knowingly compromised, the simplest measure an issuer can take is to remove this key from the DID document, which will automatically invalidate all credentials issued by that key. In case an issuer loses control over the DID, it would be removed from the trusted issuer registry, which automatically invalidates all credentials issued from this DID.

However, it may take a while until the fraudulent activity gets noticed and reported. In the worst case, it may never be noticed that an unauthorized party obtained an issuer's key material and secretly issues VCs. To this end, we recommend implementing an issuance logging mechanism that can be monitored for auditing purposes to detect suspicious activity. Such a logging system could be integrated into the issuer's system, but we also advocate for a form of publicly available issuance logging for all issued credentials to a verifiable, trusted data store. We will further address this direction in section 8.5.

Even with an issuing and monitoring system in place, great responsibility will always lie on the university to appropriately secure its systems, both from a technical and organizational perspective.

#### Trusted Issuer Registry Control by a Malicious Party

Since the trusted issuer registry serves the relying party as a trust anchor for the issuer identification, it is a critical target for attacks: In case a malicious party obtains the control

over a trusted issuer registry, they could add arbitrary DIDs which can issue valid credentials. We therefore strictly advise against registries that are maintained by a single entity. Instead, we recommend relying on registries maintained by several independent parties to increase the barriers that must be reached by a maintainer to obtain control over the registry. As discussed, this could be implemented via smart contracts or proof sets. It is important that registry maintainers act responsibly and only add entries to the registry of issuers that they could clearly authenticate. For this process, clear guidelines and policies should be established. A big advantage of the trusted issuer registry is that it is publicly available and can easily be monitored. We therefore highly recommend monitoring any updates to the registry and acting accordingly in case suspicious activity is detected.

#### Holder Key Security

The loss of a credential *holder's* private key (or potentially, even the voluntary sharing among different students) is a possible scenario that should also be considered. If Bob is somehow able to obtain Alice's private key and potentially even some of her credentials, Bob can present them as valid VPs to arbitrary relying parties. However, as the linked identity credential contains Alice's personal information, Bob won't be able to pretend that the degrees he is presenting are his own. However, being able to act in Alice's name is a serious case of identity theft which could have fatal consequences for Alice. This underlines how important private key security is in the context of decentralized identity. Luckily, wallet apps usually provide strong security mechanisms, such as encryption, 2-factor-authentication, and biometric authentication checks, so that the key material should normally be sufficiently well protected. If a leak happens, the DID should be revoked immediately.

## 8.4. Credential Updatability

In chapter 4.4, we already touched on the topic of credential updatability – which our described infrastructure supports since any issuer can revoke an outdated credential and re-issue it with the up-to-date information. This process is already a big step forward from physical documents, which typically must be mailed back to the issuing university, which officially destroys the document and then prints, signs, and sends back the new document. For VCs, the revocation and re-issuing process could be easily integrated into an administration UI in the student information system, like TUM Online.

However, the fact remains that the issuer is required for a credential to be updated. If the credential update is associated with any degree-related information, this makes perfect sense. But if the change of information is related to the credential holder's personal data, because, for example, their name changes, it is a very tedious process for the holder to ask every single issuer of all owned credentials to update the VCs accordingly. It might even be impossible, in the case that an issuer doesn't exist anymore.

By introducing the concept of identity credentials, we outsourced all holder-related personal information into a separate, single ID credential. The ID credential can be referenced by

every diploma credential to store all holder-related personal information in one single place. This opens the potential for new update concepts of the student's personal data, as it only needs to be updated in the ID credential. Obviously, the ID credential's signature has been created specifically for the original ID data, and simply editing the outdated information would invalidate the ID credential's signature. Therefore, a new credential must be issued from an identity credential provider, which makes sense, as any change of personal data should be combined with an in-person verification by an identity provider. However, as each diploma credential contains the hash of the original ID credential, the linking will be broken.

A possible solution that we could imagine for this problem is to link the new, updated ID credential to the outdated ID credential, which effectively results in a chain of linked credentials. For verification, holders who updated their information would provide the entire list of all ID credentials to the verifier so they can chain it back to the one linked in the diploma. By design, this mechanism provides a change log of the holder's identity data history. To prevent the holder from presenting only their outdated ID credential, we suggest introducing a new status *OUTDATED* that can be managed via a custom status list analogously to revocation information. We see huge potential in this approach, as it could provide an elegant solution to the real problem of updating personal data in issued diplomas.

A downside could be that the original diploma issuer is not asked to approve the ID credential's update. However, avoiding the need for issuer involvement was exactly the problem we wanted to solve. Since a trusted identity credential issuer has confirmed the update, we believe it is an acceptable assumption that the update is legitimate. It would also be possible to let issuers and relying parties configure policies about what types of identity credential updates are allowed. For example, only N character insertions/deletions might be accepted; or only name changes but no birthdate changes.

## 8.5. Issuance Logging

We already mentioned the concept of issuance logging, which brings several important advantages:

**Notion of Time:** As already discussed in the context of timestamp verifications, the `issued` timestamp can be filled by the issuer with arbitrary data, therefore not providing a reliable indication of the issuance time. By leveraging an issuance log on a trusted data infrastructure, we could obtain a clear timestamp indicating before when a given credential was issued.

**Monitoring and Auditing:** Additionally, an issuance logging concept could enable auditing mechanisms to detect if credentials are issued maliciously or by mistake. For example, if a single malicious university employee has access to the signing key, they could issue arbitrary credentials. However, if an issuing log entry is required, every unauthorized credential issuance would be noticed.

For privacy purposes, it is recommended that no personally identifiable information should be stored in the issuance logs. While the credential's hash could be sufficient to prove its

issuance, log entries might also contain further non-PII related data, like the type of credential issued, to allow for more specific monitoring mechanisms.

It is important that the logs are stored on a trusted, publicly available data storage similar to the trusted issuer registry. However, issuance logging is significantly more challenging because is requires much more write operations, since every issued credential must be logged. This can be challenging if a storage like Ethereum, which requires gas fees, is used.

A system that could serve as a reference model here is *Certificate Transparency (CT)* [90], which is a publicly available logging and auditing system for the issuance of TLS certificates, allowing anyone to monitor issuing Certificate Authorities (CA) and detect suspicious activity. CT works as a distributed, append-only log system. Log entries are cryptographically secured by Merkle trees, whose leaves represent the hashes of individual certificates, making the log entries tamper-proof and verifiable. [91]

To conclude, while a concept of issuance logging is not a core requirement of DiBiHo, since other than the previously discussed checks, it is not absolutely required to verify a credential's validity, it definitely increases the security of the overall system a lot. Therefore, we highly recommend to investigate this direction further.

## 8.6. Generalization to Use Cases other than Education

While in this thesis, we have primarily considered digital credentials in the higher education context, many of the concepts that we discussed are also applicable to completely different domains – for example, employment certificates.

Because most of the checks that we discussed happen on the envelope level and we implemented our prototype generalizable and configurable, it can easily be configured for usage in different domains:

- The relying party can configure additional trusted issuer registries, e.g., one registry specifically for companies that issue digital work certificates.
- Issuers can voluntarily add an identity credential link to their issued VCs. The relying party can configure additional credential types that require an identity check.
- Credential preview renderers for additional credential types can be added to the frontend code.

The remaining components of our implementation (such as presentation and challenge verification, credential status check, etc.) do not need additional configuration and work out of the box with arbitrary credential types.

To go a step further, we believe that there is potential for future work to make such customized verification checks and visualizations even more flexible to account for the many different use cases in which VCs can be applied:

### Generalized Customizable Credential Rendering

Since, as discussed, we consider the functionality of dynamic credential rendering after a successful credential verification very important, we see potential for future work in making this process more generic and customizable.

Instead of installing several credential renderers for different credential types in the verifier frontend, the entire styling could be specified by the issuer. While [92] specifies a set of pre-defined styling properties that issuers can enter directly in a VC, it only provides limited customizability and does not allow to update the appearance of a credential. [45] uses a web-based renderer, which provides customizability but adds a dependance on the issuer's infrastructure.

A more flexible approach towards *Decentral Rendering* could be to define a styling template service in the DID document so that the university can specify the rendering directly on a verifiable data registry, while still being able to update the appearance of a credential after issuing if needed.

### Generalized Customizable Verification Checks

The introduction of a customized identity credential check that TUM requires for diploma credentials has shown that different issuers might prescribe different verification checks for their credentials that are supplementary to the basic VC checks according to the W3C standard.

While we have implemented the ID credential check as a part of our overall verification routine, we see a lot of potential for approaches to abstract such additional customized verification rules that can be prescribed by issuers directly in the credential (e.g., by a property `requiredVerificationChecks`).

The declaration of these additional verification rules could happen in form of programmed verifier plugins or add-on scripts. Those could be managed in a registry and might be automatically executed by the verification software if required by an issuer or a relying party.

# 9. Conclusion

In this thesis, we explored the *Verification of Digital Credentials Supporting Self-Sovereign Identity for Higher Education Institutions*. To this end, we identified a set of checks that are necessary to determine whether a digital diploma credential can be considered trustworthy. Additionally, we designed a concept for an issuer registry, allowing to bridge the gap between real-world organizations and their digital identifiers. Furthermore, we analyzed and evaluated the landscape of available SSI libraries and how they can facilitate the development of a verifier software. Ultimately, we applied the findings of our research questions and implemented a proof of concept for a verification service, allowing relying parties to verify given academic credentials.

This was the last missing piece of the DiBiHo project, which now constitutes a fully functioning and integrated infrastructure for issuing, storing, transferring, and verifying digital academic credentials. While there are still some open questions, those are mostly related to operation concepts and governance-related aspects. Our proof of concept has shown that the technology is ready to implement a digital equivalent to physical credentials. As the next phase towards our mission, we recommend the realization of a pilot project at TUM, enabling students to optionally download their diplomas as verifiable credentials. We believe that a real-world operation in practice will help to answer these more governance-related questions.

As TUM takes a pioneering role in the development of digital credentials and by teaming up with strong partners in the DCC, we are confident that we can actively contribute to a global standard for verifiable credentials in the higher education sector. This makes it only a matter of time until our shared vision will become reality, allowing us to digitally store our lifelong record of education achievements in digital wallets and easily and securely present them to third parties. We think that the education sector is just the beginning – and by pushing forward the state of the art and adoption in this important and relevant field, we can make a huge step towards a new era of digital identity.

# A. Trusted Issuer Registry Prototype

```
{
    "@context": [
        "https://www.w3.org/2018/credentials/v1",
        "https://dibiho.org/contexts/Trusted.json",
        "https://w3id.org/security/suites/ed25519-2020/v1"
    ],
    "id": "http://dibiho.org/trusted/issuers.json",
    "type": [
        "VerifiableCredential",
        "TrustedIssuerRegistryCredential"
    ],
    "issuer": {
        "id": "did:web:dibiho.org:TUM"
    },
    "credentialSubject": {
        "id": "http://dibiho.org/trusted/issuers.json#registry",
        "trustedIssuers": {
            "did:ethr:goerli:0x02f7fbd7af3819b0e67fbd0a12a22cbb27cbb75201cae6d853f1b4d02c8e0b59c5": {
                "name": "OpenHPI",
                "location": {
                    "country": "DE",
                    "region": "Brandenburg",
                    "city": "Potsdam"
                },
                "url": "https://staging.openhpi.de",
                "logoHash": "75a76811925805afde1df4aac08953106904ae3f7785b41671bafa05bd30fd3e",
                "trustLevel": "HIGH",
                "credentialTypes": [ "VerifiableCredential", "AchievementCredential" ]
            },
            "did:key:z6MkfpaQUDMoASZmE9TKa3ZTM9VHM8CghdM32mbi3B8Bi5XC": {
                "name": "Technical University of Munich",
                "location": {
                    "country": "DE",
                    "region": "Bavaria",
                    "city": "Munich"
                },
                "url": "https://www.tum.de/",
                "logoHash": "4feca3df62f0c8523172888a4689c57e78af5e26a946524cb176b689a1556bfb",
                "trustLevel": "HIGH",
                "credentialTypes": [ "VerifiableCredential", "ElmoDiplomaCredential", "ElmoIdentityCredential" ]
            },
            < Further Issuer Entries >
        },
        "subRegistries": [
            {
                "identifier": "ETHR:0x4bda7c5557bfdd53d838f5361a09350106b4888f",
                "authorizedCredentialTypes": [ "VerifiableCredential", "ElmoDiplomaCredential" ],
                "authorizedCountries": [ "DE" ]
            },
            ...
        ]
    },
    "issuanceDate": "2022-09-15T12:10:59Z",
    "proof": [
        < Proof 1 >,
        < Proof 2 >,
        ...,
        < Proof N >,
    ]
}
```

Listing A.1: Trusted Issuer Registry Credential with a Proof Set Signed by Multiple Entities

```
pragma solidity >=0.7.0 <0.9.0;

import "./TrustedIssuerRegistry.sol";

enum OperationType {Add, Remove}
enum PropertyType {IssuerEntry, Maintainer, SubRegistry}

struct Proposal {
    uint proposalId;
    uint votesCount;
    OperationType operationType;
    PropertyType propertyType;
}

struct IssuerEntryProposal {
   string did;
   IssuerEntry proposedEntry;
}

contract VotingRegistry is TrustedIssuerRegistry {

    mapping (string => IssuerEntry) public trustedIssuers;
    mapping (address => bool) public maintainers;
    uint public maintainersCount = 0;
    SubRegistry[] public subregistries; // here we want to return the entire list

    mapping (uint => Proposal) public proposals;
    uint public proposalsCount = 0;
    mapping (uint => mapping (address => bool)) public votes;

    mapping (uint => IssuerEntryProposal) public issuerEntryProposals;
    mapping (uint => address) public maintainerProposals;
    mapping (uint => SubRegistry) public subregistryProposals;

    event NewProposal(uint proposalId);
    event MaintainerChanges(address _address, EventType eventType);

    modifier isMaintainer() {
        require(
            maintainers[msg.sender],
            "Sender not authorized."
        );
        _;
    }

    constructor () {
        maintainers[msg.sender] = true;
        maintainersCount = 1;
    }

    function getIssuerEntry(string memory issuerDid) external override view returns (IssuerEntry memory registryEntry) {
        return trustedIssuers[issuerDid];
    }

    function getSubregistries() external override view returns (SubRegistry[] memory) {
        return subregistries;
    }

    function createProposal(OperationType _operationType, PropertyType _propertyType) internal returns (uint id) {
        proposalsCount += 1;
        proposals[proposalsCount] = Proposal({
            proposalId: proposalsCount,
            operationType: _operationType,
            propertyType: _propertyType,
            votesCount: 0
        });
        emit NewProposal(proposalsCount);
        return proposalsCount;
    }

    function proposeIssuerEntry(string calldata issuerDid, IssuerEntry calldata entry) external {
        // create generic proposal that manages the votes
        uint proposalId = createProposal(OperationType.Add, PropertyType.IssuerEntry);

        // set type specific entry
        issuerEntryProposals[proposalId] = IssuerEntryProposal({
            did: issuerDid,
            proposedEntry: entry
        });
    }

```

```
    function proposeMaintainer(address maintainerAddress, OperationType _operationType) external {
        uint proposalId = createProposal(_operationType, PropertyType.Maintainer);
        maintainerProposals[proposalId] = maintainerAddress;
    }

    function proposeSubregistry(SubRegistry memory subRegistry, OperationType _operationType) external {
        uint proposalId = createProposal(_operationType, PropertyType.SubRegistry);
        subregistryProposals[proposalId] = subRegistry;
    }

    function handleVote(uint proposalId, PropertyType propertyType) internal isMaintainer returns (bool result) {
        require(proposals[proposalId].proposalId != 0, "Proposal does not Exist");
        require(proposals[proposalId].propertyType == propertyType, "Proposal for this PropertyType does not exist.");
        require(!votes[proposalId][msg.sender], "Sender has already voted!");
        votes[proposalId][msg.sender] = true;
        proposals[proposalId].votesCount += 1;
        return (100 * proposals[proposalId].votesCount) / maintainersCount > 50;
    }

    function voteMaintainerProposal(uint proposalId) external isMaintainer {
        if (!handleVote(proposalId, PropertyType.Maintainer)) return;

        if (proposals[proposalId].operationType == OperationType.Add) {
            require(!maintainers[maintainerProposals[proposalId]], "Address is already a maintainer.");
            maintainers[maintainerProposals[proposalId]] = true;
            maintainersCount += 1;
        } else {
            require(maintainers[maintainerProposals[proposalId]], "Address is no maintainer.");
            maintainers[maintainerProposals[proposalId]] = false;
            maintainersCount -= 1;

            // remove all casted votes (this could be made more efficient by tracking active votes for each maintainer)
            for (uint i = 1; i <= proposalsCount; i++) {
                if (votes[i][maintainerProposals[proposalId]]) {
                    votes[i][maintainerProposals[proposalId]] = false;
                    proposals[proposalId].votesCount -= 1;
                }
            }
        }

        emit MaintainerChanges(maintainerProposals[proposalId], proposals[proposalId].operationType == OperationType.Add ? EventType.Add :
    ↪ EventType.Remove);
        delete proposals[proposalId];
    }

    function voteIssuerEntryProposal(uint proposalId) external isMaintainer {
        if (!handleVote(proposalId, PropertyType.IssuerEntry)) return;
        trustedIssuers[issuerEntryProposals[proposalId].did] = issuerEntryProposals[proposalId].proposedEntry;
        emit IssuerEntryChanges(issuerEntryProposals[proposalId].did, issuerEntryProposals[proposalId].proposedEntry);
        delete proposals[proposalId];
    }

    function voteSubRegistryProposal(uint proposalId) external isMaintainer {
        if (!handleVote(proposalId, PropertyType.SubRegistry)) return;

        uint idx = subregistries.length;
        for (uint i = 0; i < subregistries.length; i++) {
            if (keccak256(bytes(subregistries[i].identifier)) == keccak256(bytes(subregistryProposals[proposalId].identifier))) {
                idx = i;
                break;
            }
        }

        if (proposals[proposalId].operationType == OperationType.Add) {
            require(idx == subregistries.length, "Sub Registry already exists!");
            subregistries.push(subregistryProposals[proposalId]);
        } else {
            require(idx < subregistries.length, "Sub Registry does not exist");
            subregistries[idx] = subregistries[subregistries.length-1];
            subregistries.pop();
        }
        emit SubregistriesChanges(subregistryProposals[proposalId], proposals[proposalId].operationType == OperationType.Add ? EventType.Add
    ↪  : EventType.Remove);
        delete proposals[proposalId];
    }

}
```

Listing A.2: Solidity Voting-Based Registry Contract

# B. Screenshots

The following figures show screenshots of the DiBiHo Verifier's User Interface and Command Line Interface.

*Please note: In order to avoid printing of unnecessarily many pages and to keep the appendix compact, the images have been reduced in size which may affect readability in the printed version of this thesis. For an optimal experience, we recommend viewing the digital PDF version of this thesis, which contains all graphics as scalable vector graphics, allowing to zoom into the graphics as desired.*

```
DiBiHo Verifier CLI

Options:
  -V, --version                                output the version number
  -h, --help                                   display help for command

Commands:
  verifyVP [options] <verifiablePresentation>  Verifies a Verifiable Presentation
  verifyVC <verifiableCredential>              Verifies a Verifiable Credential
  present [options]                            Creates and Signs a Verifiable Presentation from a set of Verifiable Credentials
  issue [options]                              Issues a Verifiable Credential
  generateDid [options]                        Generates a DID
  resolve <did>                                Resolves a DID Document
  decodeBitstring <bitstring>                  Decodes a Bitstring from a CredentialStatusList
  createBitstring [options] <revokedIndices>   Creates a Bitstring from a CredentialStatusList
  hashCredential <credentialPath>              Hashes a credential
  generateStatelessChallenge [options]         Generates a Stateless JWT Challenge Parameter based on some payload (e.g., presenter DID)
```

Listing B.1: Commands Available in the DiBiHo CLI

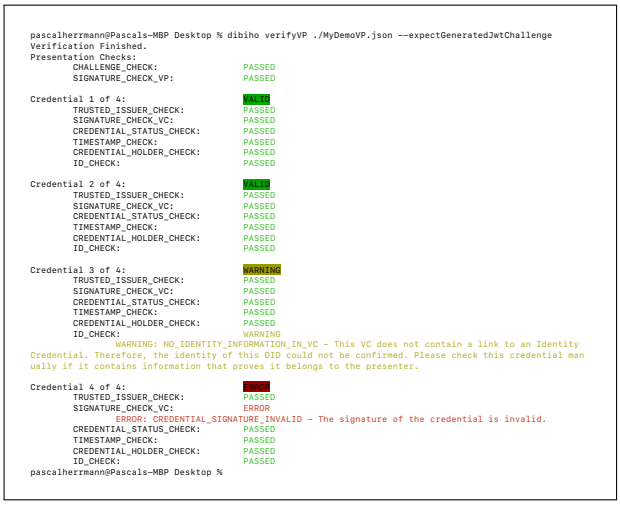


Figure B.1.: CLI Verification Output

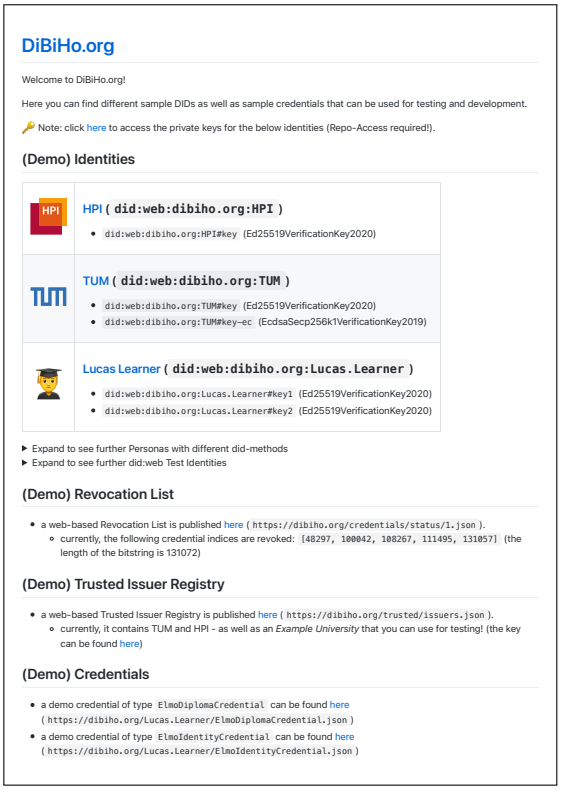


Figure B.2.: Screenshot of the demo identities hosted on the `dibiho.org` website


Figure B.3.: VP Verification Input UI

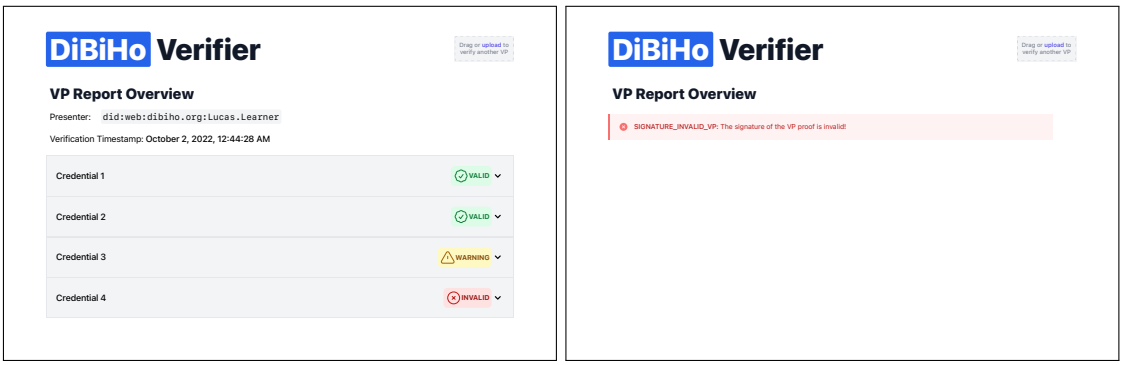


Figure B.4.: VP Verification Result – Overview

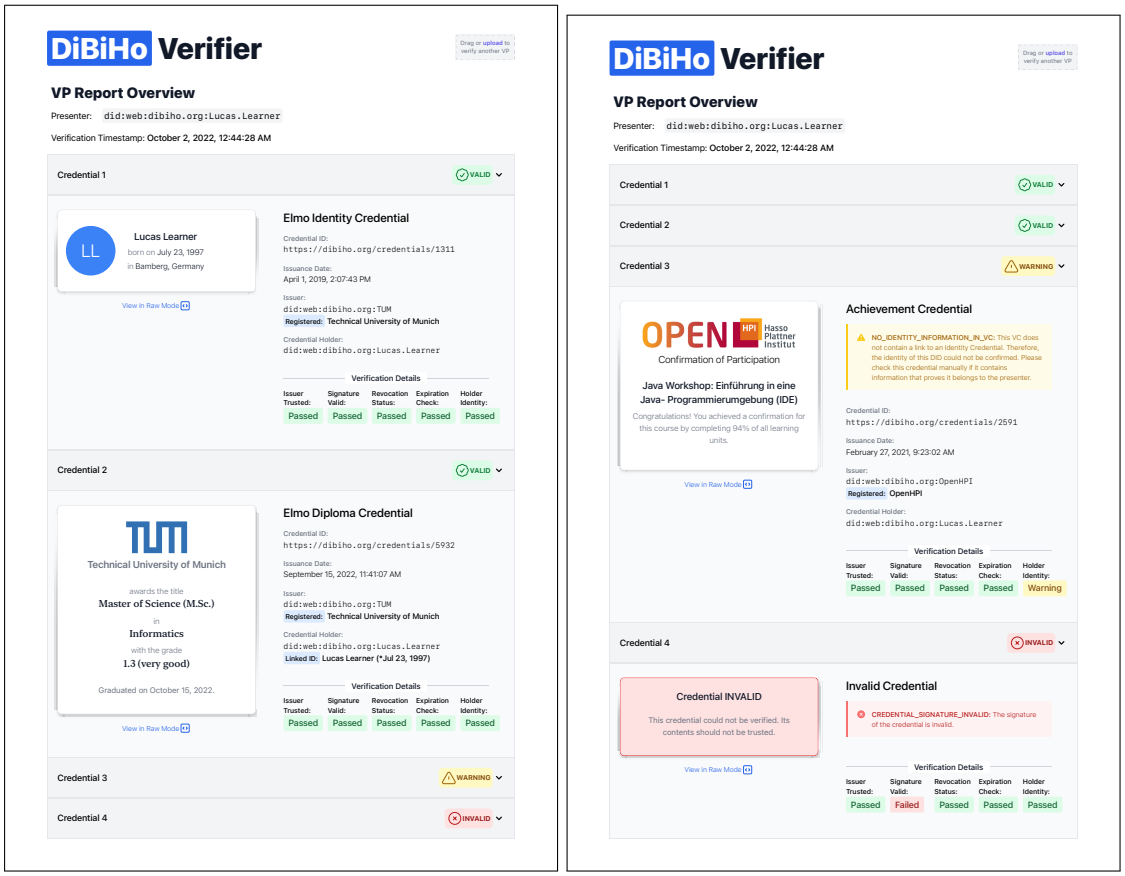


Figure B.5.: VP Verification Result – Detail View

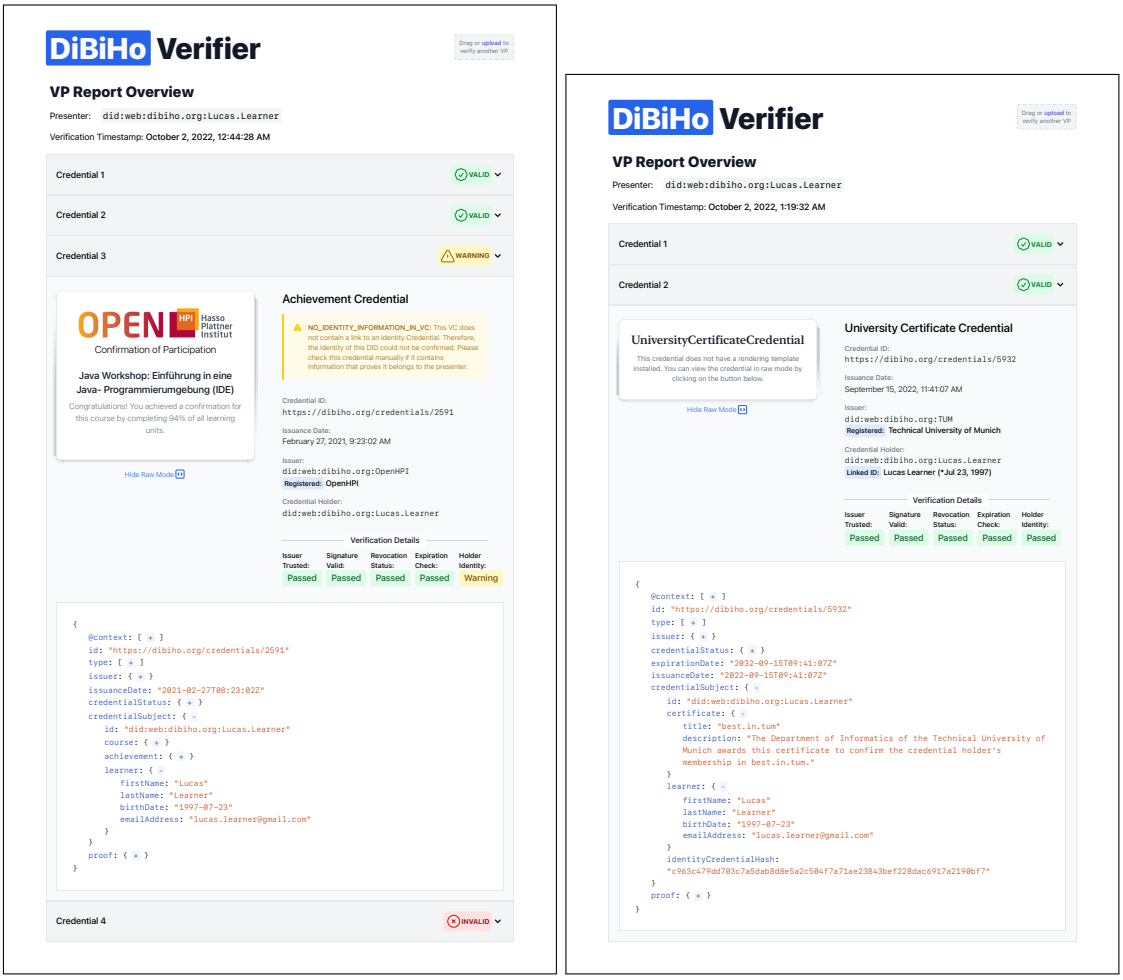


Figure B.6.: VP Verification Result – Manual Inspection

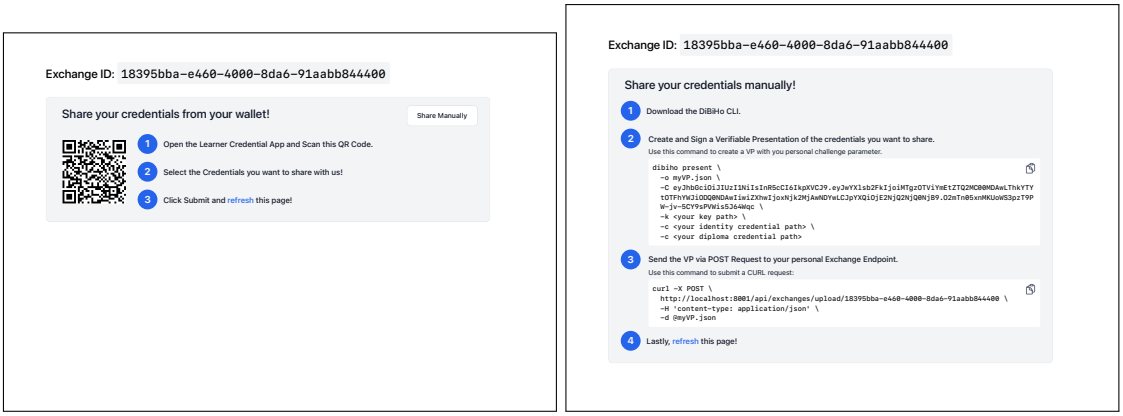


Figure B.7.: VP Exchange Instructions

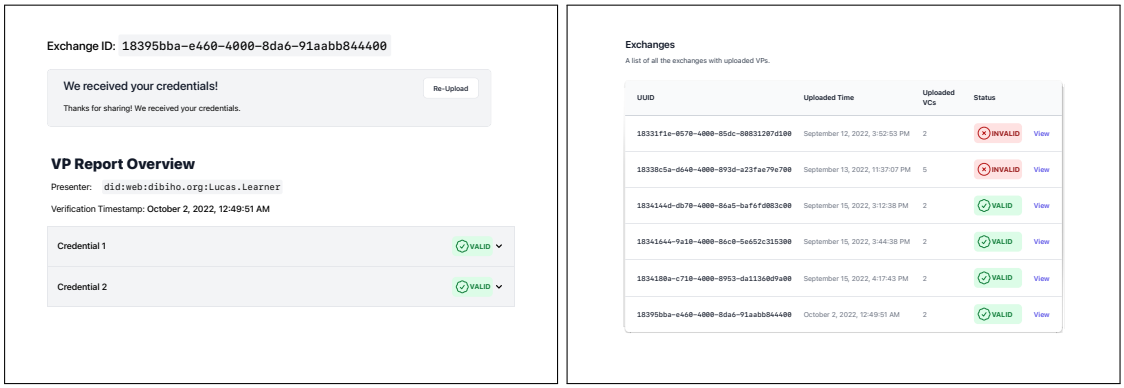


Figure B.8.: VP Exchange Succeeded

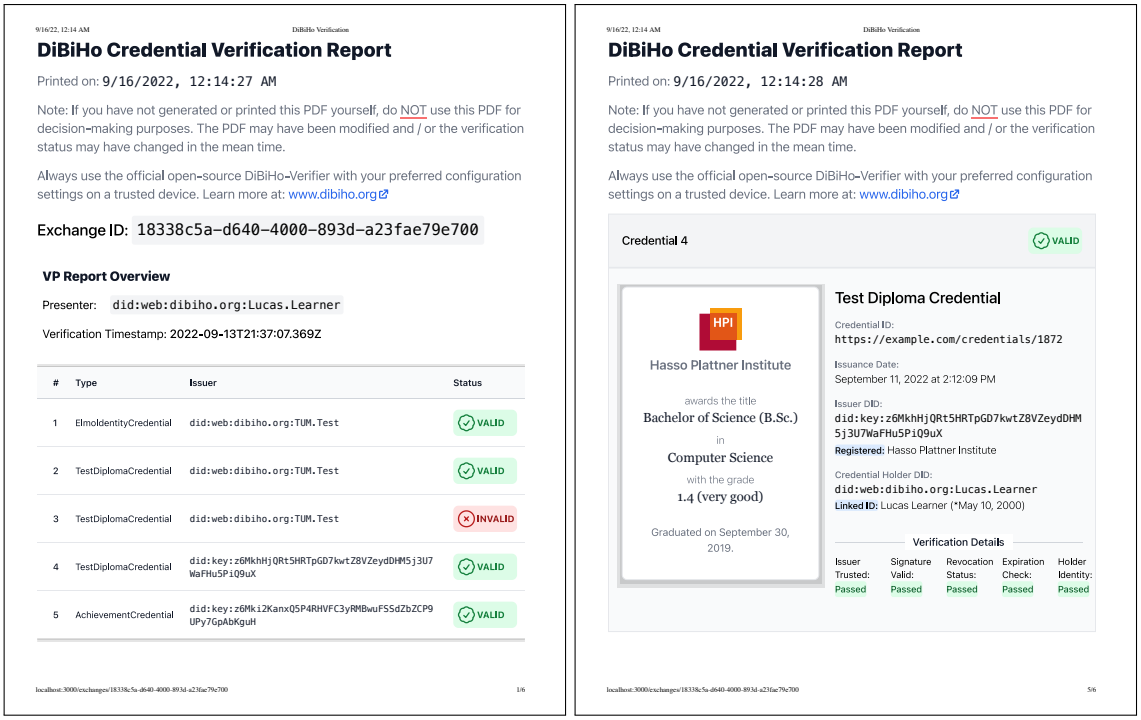


Figure B.9.: Exported / printed out Verification Report PDF. Left: Coverpage. Right: Example Page containing a valid Diploma Credential.

# List of Figures

# List of Tables

# Listings

# Bibliography


[1] M. Sporny, D. Longley, and D. Chadwick. *Verifiable Credentials Data Model v1.1*. W3C Recommendation. World Wide Web Consortium, Mar. 2022. URL: https://www.w3.org/TR/vc-data-model/.

[2] A. Preukschat and D. Reed. *Self-Sovereign Identity*. Manning Publications, 2021.

[3] M. Sporny, D. Longley, M. Sabadello, D. Reed, O. Steele, and C. Allen. *Decentralized Identifiers (DIDs) v1.0 - Core architecture, data model, and representations*. W3C Recommendation. World Wide Web Consortium, July 2022. URL: https://www.w3.org/TR/did-core.

[4] Matthias Gottlieb and Hans Pongratz. *DiBiHo: Digital Credentials for Higher Education Institutions*. 2022. URL: https://www.it.tum.de/en/it/dibiho/ (visited on 10/05/2022).

[5] *Digital Credentials Consortium - Website*. 2022. URL: https://digitalcredentials.mit.edu/ (visited on 10/06/2022).

[6] K. H. Duffy, H. Pongratz, J. P. Schmidt, J. Chartrand, S. Freeman, U. Gallersdörfer, M. Lisle, A. Mühle, and S. van Engelenburg. *Building the digital credential infrastructure for the future*. White Paper. Digital Credentials Consortium, 2020. URL: https://digitalcredentials.mit.edu/wp-content/uploads/2020/02/white-paper-building-digital-credential-infrastructure-future.pdf (visited on 10/05/2022).

[7] C. Meinel, H. Pongratz, A. Knoth, A. Mühle, M. Gottlieb, K. Clancy, D. Köhler, U. Gallersdörfer, B. Schwazwald, K. Assaf, F. Hoops, L. Peters, and E. Soldo. *Digitale Bildungsnachweise für Hochschulen - Requirements Engineering - Interim Report Version 1.0*. DiBiHo, 2022. URL: https://www.it.tum.de/fileadmin/w00bgh/www/dibiho/files/RequirementsEngineeringDiBiHo_v1.0.pdf (visited on 10/06/2022).

[8] C. Sehlke. *Transforming a Digital University Degree Issuance Process Towards Self-Sovereign Identity*. Master's Thesis. Sept. 2022.

[9] K. Schwaber and J. Sutherland. "The 2020 Scrum Guide". In: (2020). URL: https://scrumguides.org/scrum-guide.html.

[10] K. Peffers, T. Tuunanen, C. E. Gengler, M. Rossi, W. Hui, V. Virtanen, and J. Bragge. "Design Science Research Process: A Model for Producing and Presenting Information Systems Research". In: (2006). URL: https://jyx.jyu.fi/bitstream/handle/123456789/63435/Design%20Science%20Research%20Process.pdf.

[11] Svetlin Nakov. *Practical Cryptography for Developers*. 2018. URL: https://cryptobook.nakov.com/digital-signatures#digital-signature-schemes-and-algorithms (visited on 09/28/2022).

[12] D. Eastlake 3rd and T. Hansen. *US Secure Hash Algorithms (SHA and SHA-based HMAC and HKDF)*. RFC 6234. May 2011. URL: https://www.rfc-editor.org/rfc/rfc6234.

[13] *Digital Signature Standard (DSS)*. Federal Information Processing Standards Publication 186-4. National Institute of Standards and Technology, 2013. URL: https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.186-4.pdf.

[14] D. Longley and M. Sporny. *RDF Dataset Canonicalization - A Standard RDF Dataset Canonicalization Algorithm*. Draft Community Group Report. W3C Credentials Community Group, 2022. URL: https://w3c-ccg.github.io/rdf-dataset-canonicalization/spec.

[15] M. Sporny, D. Longley, and M. Prorock. *Verifiable Credential Data Integrity 1.0 - Securing the Integrity of Verifiable Credential Data*. W3C Editor's Draft. W3C Credentials Community Group, 2022. URL: https://w3c.github.io/vc-data-integrity/.

[16] S. Josefsson and I. Liusvaara. *Edwards-Curve Digital Signature Algorithm (EdDSA)*. RFC 8032. Jan. 2017. URL: https://www.rfc-editor.org/rfc/rfc8032.

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

[18] O. Steele. *Ecdsa Secp256k1 Signature 2019*. Draft Community Group Report. W3C Digital Verification Community Group, 2021. URL: https://w3c-ccg.github.io/lds-ecdsa-secp256k1-2019/.

[19] O. Steele, M. Sporny, and T. Looker. *EdDSA Cryptosuite v2020*. Final Community Group Report. W3C Credentials Community Group, 2022. URL: https://www.w3.org/community/reports/credentials/CG-FINAL-di-eddsa-2020-20220724/.

[20] M. Sabadello. *Ed25519 Signature 2018*. Draft Community Group Report. W3C Digital Verification Community Group, 2021. URL: https://w3c-ccg.github.io/lds-ed25519-2018.

[21] Auth0, Inc. *jwt.io*. 2022. URL: https://jwt.io/ (visited on 10/04/2022).

[22] M. Jones, J. Bradley, and N. Sakimura. *JSON Web Token (JWT)*. RFC 7519. May 2015. URL: https://www.rfc-editor.org/rfc/rfc7519.

[23] H. Krawczyk, M. Bellare, and R. Canetti. *HMAC: Keyed-Hashing for Message Authentication*. RFC 2104. Feb. 1997. URL: https://www.rfc-editor.org/rfc/rfc2104.

[24] Auth0, Inc. *Get Started with JSON Web Tokens*. URL: https://auth0.com/learn/json-web-tokens/ (visited on 10/06/2022).

[25] G. Kellogg, P.-A. Champin, D. Longley, M. Sporny, M. Lanthaler, and N. Lindström. *JSON-LD 1.1 - A JSON-based Serialization for Linked Data*. W3C Recommendation. World Wide Web Consortium, 2020. URL: https://www.w3.org/TR/json-ld11/ (visited on 10/05/2022).

[26] O. Steele, M. Sporny, and M. Prorock. *DID Specification Registries - The interoperability registry for Decentralized Identifiers*. W3C Group Note. World Wide Web Consortium, Aug. 2022. URL: https://www.w3.org/TR/did-spec-registries/.

[27] M. Sporny, D. Zagidulin, D. Longley, and O. Steele. *The did:key Method v0.7 - A DID Method for Static Cryptographic Keys*. Draft. W3C Credentials Community Group, Sept. 2022. URL: https://w3c-ccg.github.io/did-method-key/ (visited on 10/03/2022).

[28] M. Prorock, O. Steele, O. Terbu, C. Gribneau, M. Xu, and D. Zagidulin. *did:web Method Specification*. W3C Credentials Community Group, Aug. 2022. URL: https://w3c-ccg.github.io/did-method-web/ (visited on 10/03/2022).

[29] Veramo. *ETHR DID Method Specification.*, 2022. URL: https://github.com/decentralized-identity/ethr-did-resolver/blob/master/doc/did-method-spec.md (visited on 10/04/2022).

[30] P. Braendgaard and J. Torstensson. *Ethereum Lightweight Identity*. ERC-1056. Ethereum Request for Comments, 2018. URL: https://eips.ethereum.org/EIPS/eip-1056 (visited on 10/04/2022).

[31] J. Millenaar. *IOTA DID Method Specification*. Draft - Version 0.5. IOTA Foundation, 2022. URL: https://wiki.iota.org/identity.rs/specs/did/iota_did_method_spec (visited on 10/03/2022).

[32] I. Foundation. *IOTA Reclaim Identification Verification Process*. 2022. URL: https://blog.iota.org/iota-reclaim-identification-verification-process-e316647e06e6/ (visited on 10/12/2022).

[33] I. Foundation. *IOTA Identity is Coming to Layer 1*. 2022. URL: https://blog.shimmer.network/iota-identity-is-coming-to-layer-1/ (visited on 10/12/2022).

[34] S. Curren, T. Looker, and O. Terbu. *DIDComm Messaging v2.x*. Editor's Draft. Decentralized Identity Foundation, 2022. URL: https://identity.foundation/didcomm-messaging/spec/ (visited on 10/04/2022).

[35] *Verifiable Credentials API v0.3 - An HTTP API for Verifiable Credentials lifecycle management*. Draft Community Group Report. W3C Credentials Community Group, Aug. 2022. URL: https://w3c-ccg.github.io/vc-api/.

[36] W. Rygielski and M. Puzar and M. Stanic and R. Borge. *EMREX - Technical Description and Implementation Guide - v1.1*. 2020. URL: https://emrex.eu/wp-content/uploads/2020/01/Technical-Guide-to-EMREX.pdf (visited on 10/05/2022).

[37] K. H. Duffy, U. Gallersdörfer, J. Goodell, M. Lisle, B. Muramatsu, and P. Schmid. *Learner Credential Wallet Specification*. Digital Credentials Consortium, May 2021. URL: https://digitalcredentials.mit.edu/docs/Learner-Credential-Wallet-Specification-May-2021.pdf (visited on 10/06/2022).

[38] *Open Source Student Wallet Final Report*. Digital Credentials Consortium, Mar. 2022. URL: https://digitalcredentials.mit.edu/docs/Open%20Source%20Student%20Wallet%20Final%20Report%20-%20Public%20Web%20Version.pdf (visited on 10/06/2022).

[39] Digital Credentials Consortium. *Learner Credential Wallet Frequently Asked Questions*. 2022. URL: https://lcw.app/faq.html (visited on 10/06/2022).

[40] IMS Global Learning Consortium Inc. *Open Badges*. 2022. URL: https://openbadges.org/ (visited on 10/12/2022).

[41] K. Lemoie. *Open Badges as Verifiable Credentials*. 2021. URL: https://kayaelle.medium.com/in-the-w3c-vc-edu-call-on-june7-2021-we-discussed-open-badges-asserted-as-w3c-verifiable-90391cb9a7b7 (visited on 10/12/2022).

[42] *Open Badges v2.0*. IMS Final Release. IMS Global Learning Consortium, Inc., 2018. URL: https://www.imsglobal.org/sites/default/files/Badges/OBv2p0Final/index.html.

[43] *Open Badges Specification - V3.0*. Candidate Final. 1EdTech Consortium Inc., 2022. URL: https://imsglobal.github.io/openbadges-specification/ob_v3p0.html#decentralized-identifiers-and-self-sovereign-identity.

[44] Government Technology Agency (Singapore). *Open Certs - Documentation*. 2022. URL: https://docs.opencerts.io/ (visited on 10/12/2022).

[45] Government Technology Agency (Singapore). *Open Attestation - Documentation*. 2022. URL: https://www.openattestation.com (visited on 10/12/2022).

[46] R. Rosenbaum. *Using the Domain Name System To Store Arbitrary String Attributes*. RFC 1464. May 1993. URL: https://www.rfc-editor.org/rfc/rfc1464.

[47] European Commission. *EBSI | European Blockchain Services Infrastructure*. 2022. URL: https://ec.europa.eu/digital-building-blocks/wikis/display/ebsi (visited on 10/12/2022).

[48] *EBSI - Specifications*. 2022. URL: https://ec.europa.eu/digital-building-blocks/wikis/display/EBSIDOC/EBSI+DID+Method (visited on 10/12/2022).

[49] *EBSI - API Documentation*. 2022. URL: https://api.preprod.ebsi.eu/docs/apis (visited on 10/12/2022).

[50] M. Sporny and D. Longley. *Status List 2021 - Privacy-preserving status information for Verifiable Credentials*. W3C Draft Community Group Report. W3C Credentials Community Group, June 2022. URL: https://w3c-ccg.github.io/vc-status-list-2021/.

[51] P. Deutsch. *GZIP file format specification version 4.3*. RFC 1952. May 1996. URL: https://www.rfc-editor.org/rfc/rfc1952.

[52] S. Josefsson. *The Base16, Base32, and Base64 Data Encodings*. RFC 4648. Oct. 2006. URL: https://www.rfc-editor.org/rfc/rfc4648.

[53] A. Mühle and F. Hoops. *Proof of Concept Presentation: Digital Credentials for Higher Education Institutions – DiBiHo*. DCC Community Call. Sept. 2022.

[54] M. Sporny and D. Longley. *Revocation List 2020 - A privacy-preserving mechanism for revoking Verifiable Credentials*. W3C Draft Community Group Report. W3C Credentials Community Group, Apr. 2021. URL: https://w3c-ccg.github.io/vc-status-rl-2020/.

[55] P. Deutsch. *ZLIB Compressed Data Format Specification version 3.3*. RFC 1950. May 1996. URL: https://www.rfc-editor.org/rfc/rfc1950.

[56] P. Leach, M. Mealling, and R. Salz. *A Universally Unique IDentifier (UUID) URN Namespace*. RFC 4122. July 2005. URL: https://www.rfc-editor.org/rfc/rfc4122.

[57] A. Aman and E. Hedges. *Wallet And Credential Interactions*. Draft. Decentralized Identity Foundation, 2022. URL: https://identity.foundation/wallet-and-credential-interactions.

[58] Statistisches Bundesamt. *Hochschulen nach Hochschularten*. 2022. URL: https://www.destatis.de/DE/Themen/Gesellschaft-Umwelt/Bildung-Forschung-Kultur/Hochschulen/Tabellen/hochschulen-hochschularten.html (visited on 09/18/2022).

[59] Statistisches Bundesamt. *Studierende nach Bundesländern*. 2022. URL: https://www.destatis.de/DE/Themen/Gesellschaft-Umwelt/Bildung-Forschung-Kultur/Hochschulen/Tabellen/studierende-insgesamt-bundeslaender.html (visited on 09/18/2022).

[60] *Secure Hash Standard (SHS)*. Federal Information Processing Standards Publication 180-4. National Institute of Standards and Technology, 2015. URL: https://csrc.nist.gov/publications/detail/fips/180/4/final.

[61] IOTA Foundation. *IOTA for Business - Permissionlessinnovation*. 2022. URL: https://files.iota.org/comms/IOTA_for_Business.pdf (visited on 09/27/2022).

[62] Ethan Fast. *Cryptography behind the top 100 cryptocurrencies*. 2021. URL: http://ethanfast.com/top-crypto.html (visited on 09/28/2022).

[63] Open Source Initiative. *Licenses & Standards*. 2022. URL: https://opensource.org/licenses (visited on 09/28/2022).

[64] Spruce Systems, Inc. *DIDKit - A toolkit for Verifiable Credential and Decentralized Identifier functionality*. 2022. URL: https://www.spruceid.dev/didkit/didkit (visited on 10/05/2022).

[65] IOTA Foundation. *iotaledger/identity.rs - Implementation of the Decentralized Identity standards such as DID and Verifiable Credentials by W3C for the IOTA Tangle.* 2022. URL: https://github.com/iotaledger/identity.rs (visited on 10/06/2022).

[66] IOTA Foundation. *Revocation Bitmap 2022*. 2022. URL: https://wiki.iota.org/identity.rs/specs/revocation_bitmap_2022 (visited on 10/12/2022).

[67] Decentralized Identity Foundation. *decentralized-identity/universal-resolver - Universal Resolver implementation and drivers.* 2022. URL: https://github.com/decentralized-identity/universal-resolver (visited on 10/06/2022).

[68] Docker Inc. *Docker Compose - Overview*. 2022. URL: https://docs.docker.com/compose/ (visited on 09/28/2022).

[69] Decentralized Identity Foundation. *decentralized-identity/did-resolver - Universal did-resolver for javascript environments*. 2022. URL: https://github.com/decentralized-identity/did-resolver (visited on 10/06/2022).

[70] Digital Bazaar, Inc. *Verifiable Credentials JS Library (@digitalbazaar/vc)*. 2022. URL: https://github.com/digitalbazaar/vc-js (visited on 10/02/2022).

[71] *Digital Credentials Consortium - GitHub Profile*. 2022. URL: https://github.com/digitalcredentials (visited on 10/02/2022).

[72] Veramo. *uport-project/veramo*. 2022. URL: https://github.com/uport-project/veramo (visited on 10/02/2022).

[73] Veramo. *Veramo - Documentation*. 2022. URL: https://veramo.io/docs/basics/introduction (visited on 10/02/2022).

[74] DiBiHo. *State of the Art Analysis - Trusted Data Registry*. 2022. URL: https://gitlab.hpi.de/dibiho/development/state-of-the-art-analysis/-/wikis/Trusted-Data-Registry (visited on 10/05/2022).

[75] Digital Credentials Consortium. *MVP DCC Issuer Registry*. 2022. URL: https://github.com/digitalcredentials/issuer-registry/ (visited on 09/24/2022).

[76] P. Mockapetris. *Domain Names - Concepts and Facilities*. RFC 1034. Nov. 1987. URL: https://www.rfc-editor.org/rfc/rfc1034.

[77] D. Buchner, O. Steele, and T. Looker. *Well Known DID Configuration*. DIF Working Group Approved Draft. Decentralized Identity Foundation, 2021. URL: https://identity.foundation/.well-known/resources/did-configuration/.

[78] M. Nottingham. *Well-Known Uniform Resource Identifiers (URIs)*. RFC 8615. May 2019. URL: https://www.rfc-editor.org/rfc/rfc8615.

[79] U. Gallersdörfer and F. Matthes. "TeSC: TLS/SSL-Certificate Endorsed Smart Contracts". In: *2021 IEEE International Conference on Decentralized Applications and Infrastructures (DAPPS)*. 2021, pp. 95–100. DOI: 10.1109/DAPPS52256.2021.00016.

[80] J. Spaulding, S. Upadhyaya, and A. Mohaisen. *The Landscape of Domain Name Typosquatting: Techniques and Countermeasures*. 2016. DOI: 10.48550/ARXIV.1603.02767. URL: https://arxiv.org/abs/1603.02767.

[81] F. Hoops. *Threat Analysis, Evaluation, and Mitigation for Smart Contracts Endorsed by TLS/SSL Certificates*. Master's Thesis. Apr. 2021.

[82] Educause. *Eligiblity*. 2022. URL: https://net.educause.edu/eligibility.htm (visited on 09/18/2022).

[83] D. Buchner, B. Zundel, M. Riedel, and K. H. Duffy. *Presentation Exchange 2.0.0*. Working Group Draft. Decentralized Identity Foundation, 2022. URL: https://identity.foundation/presentation-exchange.

[84] O. Steele, B. Zundel, A. Aman, E. Hedges, J. Hensley, S. Curren, B. Richter, R. Quadras, and J. Caballero. *WACI-DIDComm Interop Profile*. Draft V1.0. Decentralized Identity Foundation. URL: https://identity.foundation/waci-didcomm/ (visited on 10/02/2022).

[85] D. Longley, M. Varley, and D. Zagidulin. *Verifiable Presentation Request v0.2*. Draft Community Group Report. W3C Credentials Community Group, Aug. 2022. URL: https://w3c-ccg.github.io/vp-request-spec/.

[86] D. Longley and M. Sporny. *Credential Handler API 1.0*. Draft Community Group Report. W3C Credentials Community Group, June 2021. URL: https://w3c-ccg.github.io/credential-handler-api/ (visited on 09/28/2022).

[87] W3C Credentials Community Group. *CHAPI for VC Verifiers*. URL: https://chapi.io/developers/verifiers (visited on 09/28/2022).

[88] D. Buchner, B. Zundel, J. Hensley, and D. McGrogan. *Engineering Privacy for Verified Credentials - In Which We Describe Data Minimization, Selective Disclosure, and Progressive Trust*. Draft Community Group Report. W3C Credentials Community Group, 2022. URL: https://w3c-ccg.github.io/data-minimization/.

[89] D. Chadwick, D. Longley, M. Sporny, O. Terbu, D. Zagidulin, and B. Zundel. *Verifiable Credentials Implementation Guidelines 1.0 - Implementation guidance for Verifiable Credentials*. W3C Working Group Note. W3C Credentials Community Group, Sept. 2019. URL: https://www.w3.org/TR/vc-imp-guide.

[90] B. Laurie, E. Messeri, and R. Stradling. *Certificate Transparency Version 2.0*. RFC 9162. Dec. 2021. URL: https://www.rfc-editor.org/rfc/rfc9162.

[91] Google. *Certificate Transparency - Working together to detect maliciously or mistakenly issued certificates.* URL: https://certificate.transparency.dev/ (visited on 10/06/2022).

[92] D. Buchner, B. Zundel, J. Hensley, and D. McGrogan. *Wallet Rendering*. Draft. Decentralized Identity Foundation, 2022. URL: https://identity.foundation/wallet-rendering/.