Ca' Foscari
University
of Venice

Master's Degree Programme
in Computer Science

Final Thesis

Improving the trust model of Self-Sovereign Identity on blockchain

Supervisor
Ch. Prof. Andrea Marin

Graduand
Alessio De Biasi
Matriculation Number 870288

Academic Year
2022 / 2023

Abstract

In the digital world, when users want to prove something about their identities (e.g., age, or degree grade), they need to provide pictures of physical documents (e.g., ID cards, or degree certificates).

However, these documents may reveal on the identity of the users more than what the users want to (e.g., the ID card reveals the birthdate, but also the home address). Moreover, a malicious user can reuse those pictures, so to pretend to be another user.

Self-Sovereign Identity, together with the blockchain technology, gives back to the users the full control over the information they share about their own identities. In this case, when the users want to prove a claim about their identities to another entity (called *verifier*), they send a Verifiable Credential, which is tightly tied to them and cannot be reused by any other user.

The verifier uses publicly-available information stored on the blockchain to establish the validity of that credential, but can accept the credential only if the entity that has issued it (called *issuer*) is trusted. This trust is often established by means of invitations, and requires the verifiers to obtain the invitation from each of the issuers they want to trust.

In this thesis, we propose an extension of the current trust model, allowing a verifier to trust an issuer without obtaining any invitation from them, provided that another entity, trusted by the verifier, trusts (directly or indirectly) the issuer, effectively creating a chain of trust.

Keywords- Self-Sovereign, Blockchain, Solidity, Digital identity, Chain of Trust

Contents

1 Introduction 1
- 1.1 The solution we are looking for . . . 2
- 1.2 Structure of the thesis . . . 3

2 Digital identity 4
- 2.1 Digital identity management systems . . . 4
- 2.2 Centralized identity . . . 4
- 2.3 Federated identity . . . 5
- 2.4 Self-Sovereign identity . . . 6
- 2.5 Summary . . . 7

3 State of the art 8
- 3.1 Implementations . . . 9
  - 3.1.1 Sovrin . . . 9
  - 3.1.2 veramo . . . 10
  - 3.1.3 everest . . . 10
  - 3.1.4 Evernym . . . 10
  - 3.1.5 Midy . . . 10
  - 3.1.6 self key . . . 11
  - 3.1.7 civic . . . 11
  - 3.1.8 Other implementations . . . 11
- 3.2 Our approach to the problem . . . 11
  - 3.2.1 Digital certificates . . . 12
- 3.3 Summary . . . 13

4 Data representations 14
- 4.1 Resource Description Format . . . 14
  - 4.1.1 RDF canonicalization . . . 17
- 4.2 JSON-LD . . . 17
- 4.3 Summary . . . 20

5 Decentralized Identifiers **21**
5.1 DID scheme . . . 21
5.2 DID URL . . . 22
5.3 DID Document . . . 22
5.3.1 Verification methods . . . 24
5.3.2 EcdsaSecp256k1RecoveryMethod2020 . . . 25
5.3.3 Services . . . 25
5.4 CredentialRegistry . . . 26
5.5 DID method . . . 27
5.6 DID resolver and DID URL dereferencer . . . 27
5.6.1 DID resolution . . . 27
5.6.2 Did URL dereferencing . . . 29
5.7 Summary . . . 30

6 Verifiable credentials **32**
6.1 Trust model . . . 33
6.2 Data Integrity Proofs . . . 35
6.2.1 JSON Web Signature . . . 36
6.2.2 EcdsaSecp256k1RecoverySignature2020 . . . 37
6.3 Zero-Knowledge Proofs . . . 39
6.4 Credential status . . . 39
6.5 Verifiable Presentations . . . 39
6.6 Risk of correlation . . . 40
6.7 Summary . . . 40

7 Blockchain **41**
7.1 Blockchain ownership . . . 41
7.2 Ethereum . . . 42
7.3 Accounts and wallets . . . 42
7.3.1 Smart contract . . . 42
7.3.2 EVM . . . 43
7.3.3 Gas . . . 43
7.3.4 Networks . . . 44
7.3.5 Oracles . . . 45
7.3.6 Consensus algorithms . . . 45
7.3.7 Digital signatures . . . 47
7.3.8 Attacks . . . 48
7.4 Summary . . . 48

8 Implementation **49**
8.1 Chains of trust 49
8.2 Authentication 50
8.3 Role of the blockchain 50
8.4 library 50
8.5 CRUD operations 51
8.5.1 Create 51
8.5.2 Read 51
8.5.3 Update 51
8.5.4 Delete (deactivate) 52
8.6 DID document 52
8.6.1 Reification 52
8.7 Services 53
8.8 Trust certification 53
8.9 Invitations 55
8.10 DID resolution 56
8.11 DID URL dereferencing 56
8.12 Chain resolution 56
8.13 RevocationList2023 57
8.14 Smart contract 58
8.15 Privacy 59
8.16 Security 60
8.17 Summary 61

9 Case studies **62**
9.1 Summary 63

10 Conclusion and future work **64**

Bibliography **66**

Chapter 1

Introduction

Nowadays, more and more online services the users commonly access (like, for instance, e-mail clients, e-commerce websites, social networking services, or video streaming platforms) request them to authenticate. To perform the authentication, these services typically ask the users to fill a web form with an e-mail (or a username) and a password.

But what if the service needs to verify whether the user satisfies or not specific requirements? For instance, a cinema may require users to prove they are over 18 before booking tickets for watching a horror movie, while a car rental service may require users to demonstrate they have a valid driving licence.

In all such cases, users are typically asked to provide pictures of physical documents (e.g., ID cards or driver licences). However, those pictures are difficult to be automatically processed by a machine. Moreover, sharing the whole picture of a document may lead to identity thefts.

Digital identity is a way to solve these problems. The use of machine-readable documents (like JSON documents) together with the use of cryptographic material (like private and public keys) solve all the previously-mentioned problems.

Self-Sovereign Identity is a step further in the digital identity ecosystem. In particular, when a user wants to prove specific claims about their identity (like name, surname, birthdate, home address, etc.), they need to request to a trusted party, called *issuer*, a Verifiable Credential, which is a JSON document containing all the claims the user wants to prove. The Verifiable Credential is then digitally signed by the issuer, and made available to the user.

The user can then share that Verifiable Credential to any entity, called *verifier*, that requests the user to prove the claims contained in the credential. In particular, the *verifier* makes use of publicly available information to verify the validity of the claims inside the received Verifiable Credential.

However, the *verifier* can accept the Verifiable Credential only if they trust the *issuer*. For instance, the cinema can accept a Verifiable Credential presented by a user only if it has been issued by a trusted entity, like the registry office.

Current implementations of Self-Sovereign Identity establish this trust between the *verifier* and the *issuer* by means of invitations. These invitations are digital documents (e.g., JSON documents) containing all the information needed by the verifier to fetch the publicly available information that can then be used to verify the Verifiable Credentials issued by the *issuer* and presented by any user.

This, however, requires a verifier to obtain an invitation from each *issuers* they want to trust, which poses a scalability problem.

Indeed, imagine an employer that requires candidates for a job offer to provide their degree certificate under the form of Verifiable Credentials. To accept the credential, the employer, which acts as *verifier*, needs to obtain the invitation from any possible university in the country.

To overcome this limitation, we propose an extension to the currently-available standards so to allow *verifiers* to also trust *issuers* without obtaining any invitation from them, provided that another *issuer*, which is trusted by the *verifier* and, hence, the *verifier* has obtained an invitation from, trusts the *issuer*. This creates a chain of trust, in a way similar to the one created by digital certificates, but applied to Self-Sovereign Identity.

Therefore, supposing that the Department of Education trusts any university in the country, the employer is not forced anymore to obtain an invitation from all the universities, but the employer just needs to obtain the invitation from the Department of Education.

This however, poses new challenges a new security risks that will be detailed in this thesis. we propose also a new way to manage VC revocation.

1.1 The solution we are looking for

Establishing a chain of trust of issuer allows to reduce the computational burned on verifiers. Indeed, they do not need to retrieve an invitation from each of the issuers they want to trust, but they just need to trust an entity that, transitivey, trusts the other issuers.

This, somehow, delegates the several checks (like the trstiness of the next issuer of the chain) to another issuer, and not to the user. Indeed, it may be difficult for a user to know if an issuer is trustable or not. However, in may be easy for other issuers to do so, maybe bacause they have access to documents or because they force the new issuers of the chain to present several guanatees and documentation on their trustiness.

Consider, for example, an employer may require the candiates for a job to present a degree certificate, in the form of VC. To trust any university that could issue degree certificates, the employer is forced to improt the invitation from each of the universities. However, the employer has no access to documentation that proves that a univeristy is trstworthy.

This implies that the employer has to conduct some researches on the trusworth of each of the universities, which requires time and may not be automated.

Moreover, the employer is forced to periodically check whether new university borns, otherwise the employer may finish to reject valid Verifiable Credentials because they have been issued by a university that have not imported the invitation.

Establishing a chain of trust allows the employer to trust only a single entity, like the Department of Instruction of the governemtn. It will e then that entity that, before relasing the trust certification to a university, conducts all the necessary processes to ensure the tristworthiness of the univeristy.

Eth smart contracts are recuded in the number oand complexity of the operations -> Allow Ethemreum smart contract to validate VC

Chain of trust where anyone can be part of a chain.

A system where any user can join, and any verifier to user information to validate VC

1.2 Structure of the thesis

This thesis is structured as follows:

- In chapter 2 we will detail they different types of digital identities, putting in evidence how Self-Sovereign Identity differentiates from the other types;
- In chapter 3 we will describe how the literature tackles the problem of implementing a chain of trust in the Self-Sovereign Identity ecosystem, how our implementation differentiates from those proposals, and why the currently-available implementations of Self-Sovereign Identity are not ready to implement chains of trust;
- In chapters 5 and 6 we will explain in details the two main building blocks of Self-Sovereign Identity, that are Decentralized Identifiers and Verifiable Credentials;
- In chapter 7 we will detail what the Ethereum blockchain is, and how it is useful to implement SSI;
- In chapter 8 we will present our implementation of Self-Sovereign Identity that supports the creation of chains of trust. We will also detail the design choices we have made so to reduce or eliminate potential security risks;
- In chapter 9 we present two case studies that make use of our Self-Sovereign Identity implementation;
- In chapter 10 we conclude the thesis by detailing possible additions to our implementation.

Chapter 2

Digital identity

In this chapter we will detail the types of digital identity management systems, focusing on how Self-Sovereign Identity superseeds the others.

The digital identity is “a means for people to prove electronically that they are who they say they are and distinguish different entities from one another” [1, Sec.2].

Therefore, we can think a digital identity to be the digital counterpart of an ID card, which states our name, our surname and our birthdate.

2.1 Digital identity management systems

A Digital Identity Management System (IDMS) is a collection of policies and technologies that ensure that only relevant users have accesses to specific resources, like application, systems or specific services [2, Sec. I].

IDMS can be categorized into three types, based on where the identity of the user is actually stored. The following sections will details these three types.

2.2 Centralized identity

In centralized identity, the user, to authenticate, is requested to sign up on the online service they are accessing to, typically by providing an e-mail and a password[3, Ch. 4]. This implies that the user is creating a digital identity on each of the online services they are accessing, which forces the user to remember (or to write down) many passwords.

In centralized systems, the information the user shares with an online service are not under the control of the user, i.e., the online service can, at any time, delete those information, effectively deleting the identity of the user[4].

Moreover, all the information the user shares about their identity with an online service cannot be automatically shared with other services, i.e., the identity is not

portable. This implies that the user has to sign up on each service they access, and they need to share the same information to each service, and this may be annoying for the user [4].

However, if the users prove their identity by providing pictures of physical documents (like ID cards), these pictures are not automatically verifiable, and they need the human interaction to extract the information and validate them, which may require to directly contact the entity that has issued the document [2, Sec. C1]. However, this entity may not be always online, which implies that the verification process of the document may require several days to be completed.

2.3 Federated identity

In this type of digital identity management systems, all the information related to the identity of the user are managed by a special entity called Identity Provider (IdP)[3, Ch. 5].

Google, Facebook or the Italian SPID are all examples of federated identity management systems.

When the user wants to prove claims about their identity to an entity, the users are first requested to authenticate to the IdP, though mechanisms like Single Sign-On (SSO), whic returns back to the user a proof of the successful authentication[5, Sec. III C]

This proof can be then shared to any online service requesting the user to authenticate. Indeed, the online service, when receiving the authentication proof, can validate it by contacting the IdP and asking if the proof is valid or not. Then, if the proof is valid, the Idp shares some information about the digital identity of the user with the online service. This implies that the IdP asks the user for consent, typically by sending a notification to a user's device like the smartphone [5, Sec. III C]. Only if the user consents, the IdP will share the requested information with the online service,

With respect to centralized identity management systems, all the pieces of information related to the identity of the users are stored on servers under the control the Identity Provider, and not of the single service. Moreover, the user needs only to authenticate to the IdP to prove claims about their identity to other services, hence the user does not have to authenticate to each of the services they access, possibly using different emails and passwords. Therefore, the digital identity of the user can be onstructed from physical documents only once, and this is done by the IdP [4].

Moreover, the user can control how much information the IdP shares with each online service. For instance, in centralized identity systems, a user sending a picture of the ID card is sharing with the service the name, surname and birthdate, but also the home address, and this may not be required to be shared. With federated models, the user can select to share only the name, surname and birthdate with the online service, without sharing also the home address [5, Sec. III C].

However, the IdP can still delete, at any time, all the information related to the digital identity of the user, which will prevent the user from proving their identity to other services. [4]. Moreover, the IdP may track the activity of the user. Indeed, any service where the user present the authentication proof is required to contact the IdP so to validate the proof. This implies that the IdP may track the activity of the user, knowing exactly to which services the user is trying to authenticate to. This problem does not affect centralized identity management systems, since, when the user authenticates to an online service, the other are not contacted at all.

Note that there is still the problem, for the Identity Provider, to check the correctness and the validity of the pictures of physical documents the user provided, like for the centralized systems.

2.4 Self-Sovereign identity

This arguments explained in this section, unless otherwise states, are based on [6].

In the Self-Sovereign Identity ecosystem, we can distinguish between three entities that interact [1, Sec. 2]:

1. The user, which is the entity wants to prove claims about their identity;
2. The verifier, which is the entity that asks the user to prove specific claims about their identity;
3. The issuer, which is the entity that issues to the user a proof stating that the claims claimed by the user are correct and valid.

In particular, the issuer issues to the user a Verifiable Credential, which is a digital document containing the claims together with a proof of correctness. Th user can then present the Verifiable Credential to any verifier that asks the user to prove the claims contained in the Verifiable Credential.

The verifier will then use publicly available information to validate the proof of correctness of the Verifiable Credential. If the proof is verified, then the issuer can be sure the claims in the Verifiable Credential are correct.

Verifiable Credentials are digital documents (like JSON documents) which can be automatically analyzed by computers. There is no need to take pictures of physical documents that require human intervention to be verified and digitalized.

Since the information used to validate Verifiable Credentials are public and not stored in a server under the control of the issuer, Verifiable Credentials to be immediately verified, without the need for the verifier to contact the issuer. This implies also that the issuer has no way to track the activity of the user, because the verifier does not need to contact the issuer to validate the Verifiable Credential [4].

Moreover, the issuer may issue “atomic” Verifiable Credentials, each of which containing the minimum information possible [4]. This allows the user to choose which information to share with the verifier.

For instance, the registry office may issue the user 4 different VC, one containing the name and surname, one containing the birthdate, one containing the home address and one containing the gender. If the users are requested to prove only their name, surname ean birthdate, users can send the first two Verifiable Credentials together. This implies that the user shares with the verifier only the information the verifier needs to know [4].

In addition, Verifiable Credentials are tightly linked to the user. This implies that, verifiers cannot reuse that Verifiable Credentials to pretend to authenticate as a user (see 6). This prevents identity thefts.

Finally, Verifiable are not stored online, but they are stored on a device under the control of the user. This implies that the user has full control over them. For instance, the user can make copies the Verifiable Credentials so to distribute them across multiple devices all under the control of the user. Therefore, the user is not forced to use a specific device to prove claims about its identity, but they can use any device they want [1, p. 6.2].

2.5 Summary

In this section we have seen how Self-Sovereign Identity management systems differentiate from the centralized and federated ones. In particular, we have seen that all the information related to the identity of the user are stored in devices under the control of the user, and the user can decide which information to share to the verifiers. This is a big step further in the digital identity ecosystem, since it puts the user in the center and protects them from tracking and identity thefts.

Chapter 3

State of the art

In the literature, there are few solutions that propose the creation a chain of trust for the issuers of Verifiable Credentials. Considering the solution we are looking for, as explained in section 1.1, we excluded from these solutions the ones that require the issuer to delegate to another issuer the possibility to issue Verifiable Credential as if it were issued by the first issuer.

From the remaining solutions we selected the ones that are similar to the solution we are looking for. In particular, [7] proposes to establish a chain of trust for the issuers by using the Domain Name System, adding additional information to the DNS records. The new information are called trust lists, which are lists where issuers insert other trusted issuers.

However, the Ethereum smart contracts are not allowed to make DNS reoslution calls, unless they use oracles 7.3.5, which sould be avoided as much as possible.

Moreover, there is no way for an issuer A to avoid to be added to the trust list of another issuer B to its trust list, is the issuer to add new members (in my implementation, issuers issue certifications, and it is up to the members decide to use it or not).

Another solution is proposed in [8]. This solution makes use of Verifiable Credentials to establish the chain of trust. In partiular, if the issuer A wants to be part of a chain of trust having B as parent, the issuer A needs to ask to issuer B a Verifiable Credential that allows A to be added to the chain of trust. Moreover, the solution imposes no limit on the depth of the chain of trust.

However, [8] requires the issuers to be authorized by governing authorities, like government agencies, to be part of a chain of trust, and we want a solution where any entity can be part of a chain of trust, without the need to be authorized by issuers with an high level of trust (like governments).

Consider, for example, [9], where the authors propose the use of Verifiable Credentials to track food in the food supply chain. This implies that a farmer, when selling a vegetable, issue food certifications in the form of Verifiable Credentials so to guarantee specific food standards are met.

Imagine that a group of farmers create a consortium. In this case, verify should trust each single farmer in the consortium to properly verify Verifiable Credentials attesting the quality of the food. Using a chain of trust is a solution to this problem. Indeed, there could be an issuer that represents the entire constortium, and each farmer is part of a chain of trust having the consortium issuer as parent.

In this way, each farmer still issue Verifiable Credentials attesting the quality of their food, but any verifier needs only to trust the consortium issuer to accept those Verifiable Credentials. Moroever, if, in the furutre, additional farmers become part of the consortium, there is no need for the verifiers to directly trust them, because they trust the consortium issuer.

Using the solution proposed in [8] may not be fasible for small consortiums because the farmers will require a governance authority to onboard them in the system, and to issue a Verifiable Credential that allows the farmers to become part of a chain of trust. However, to do so, the governance authority may require additional documentation that the farmers may not be able to produce, or it may require the farmers to prove they met some requirements that they cannot meet.

The solution suggested by the same authors of [8] in [10] is a step allows any issuer (also non-governance ones) to insert other issuers in a chain of trust. However, there is still the requirement that at least one member (typically the root) of the chain of trust must be a governance autority.

3.1 Implementations

We also analyzed some of the ready-to-use implementations of Self-Sovereign Identity that make use of the blockchain, so to see if some of them already allow the establishment of chains of trust, or if they can be extended to support them.

In this section we briefly present them, but none of them can be used to implement our solution because none of them supports chain of trusts or can be exetended to support them.

3.1.1 Sovrin

This arguments explained in this section are based on [11].

Born in 2016, Sovrin [1] is stores the verification information on public and permissioned blockchain that uses Hyperledger Indy, which is an open-source project implementing a blockchain.

Sovrin allows any user to issue verifiable credentials, but users, to use the Sovrin network, must first be onboarded by trusted entities that are part of the network. Users

[1] https://sovrin.org/

can then use a mobile application, offered by Sovrin, to issue Verifiable Credentials, and to import invitations so to trust other entities.

In our implementation, many of the design choices are inspired to Sovrin.

3.1.2 veramo

As the official site states, veramo [2] allows any deveoper can create an application that makes use of Self-Sovereign Identity by using the APIs veramo makes available. In particular, the APIs are ready to be used in Node.js environments or in React applications. Veramo allows also adding additional functionalities via plugins.

3.1.3 everest

Everest [3], as the official site states, provides a Self-Sovereign Identity application that allows users to create their digital identities. To do so, the users must use the client application made available by Everest. However, Everest recently made available an SDK allowing developers to directly interact with the blockchain that stores the information without using the client application Everest provided..

3.1.4 Evernym

Evernym [4], is a Self-Sovereign Identity solution that provides deveolpers a mobile SDK that interacts, via REST APIs, with the backend server that accesses information stored on the blockchain.

Initially, Evernym supported only the Sovrin network, but they recently added the support for integrating other blockchains.

3.1.5 Midy

Midy [5] is an evolution of Evernym. In particular, as stated in [12], Midy provides a mobile application that the users can use to create digital identities. These identities are directly created from physical documents. The user just takes a picture of the document, and the Midy team will take care of verifying the document and to update the digital identity of the user accordingly.

[2] https://veramo.io/
[3] https://everest.org/
[4] https://www.evernym.com/
[5] https://www.evernym.com/

3.1.6 self key

Self Key [6] provides its mobile application, for the users, and SDK for servers. It provieds also SDK to realize custom client application they primarily allow users to authenticate to other website, which make use of the SDK provied by self-key to autehnticate. it uses Etehreum as blockchain.

3.1.7 civic

Civic [7] allows decentralized

it may use deveral blockchains, like Etehreum or solana

it scans the physical documents that is then manually verified by Cicvic and added to the digital identity

They do not use VC but similar things called attestations, which do not follow the W3C standards it supports api to access the data stored on-chain, so to verify VC.

3.1.8 Other implementations

We have also considered other publicly available but not so popular solutions, but they still do not support for the creation of a chain of trust, or they cannot be extended to support it.

3.2 Our approach to the problem

In our implementation, we allow each user to become an issuer and to be part of a chain of trust. Like proposed by we use VC to establish this trust.

We want a way for smart contracts to fully verify the identity trust certifications, without the need to call external smart contracts or external entities.

Moreover, we want a way for the verifiers to always retrieve the chain of trust of the issuers, and to retireve all the ifnromation needed to verify if the trust certification is still valid or not.

Finally, we want to a way to clearly state if a trust certification has been revoked, and that this can be directly verified on-chain.

with respect to the already-available possible implementations available in the literature, descpribed in the presious sections, we dont't want the issuers to be government authorities, but they can be any user.

[6] https://selfkey.org/

[7] https://www.civic.com/

Note that, we are not taking into account concepts like reputation. This is by design. We do not want issuers to be denied to be part from chains of trust because their reputation is not so high , maybe due to previously security breaches that allowed attackers to stela sensitive data.

We want also a way for the verifiers to not trust specific issuers, even if they are part of chains of trust where there are trusted issuers. This is because an entity may loose its trustowrhitness, and we want verifiers to blacklist those entities.

3.2.1 Digital certificates

What we have implemented look very similar to digital certificates. So, why not directly using digital certificates to create chains of trust.

Digital certificates have problems. First of all, the must be always available, other verifiers cannot verify the chain of trust of an issuer. If the digital certificate is hosted on a server under the control of the issuer, the server may be down. In this case, the verifier will not be able to obtain the digital certificate,

Moreover, it requires DNS to properly resolve the server holding the certificate. Finally, since the server is under the control of the issuer, it may track the activities of the users, in particular, to which verifiers the user present VC. Indeed, only that verifiers will ask the issuer its digital certificate.

A solution may be store the digital certificate in the blockchain. This resolves the availability problem, because the blockchain is higly-available. It also solves the problem of traking if the blockchain in public ad unpermissioned, because there is no eneity that controls the blockchain, hence there is no way to track the verifier back to the account they use to access the blockchain.

Howver, thesre is still a problem. Blockchains often puts a limit in the maximum size of the transactions. This implies that X.509 digital certificates may exceed that limit. Moreover, they contain also many information that are not necessary to prove the trust (i.e., a descriptive name of the entity the certificate belongs to). Finally, and this is a big issue, smart contracts cannot parse and verify digital certificates in an efficient way. This requires to perform several operations, and the blockchain put a limit in the number of operations that can be performed, or on their complexity. Validating a X.509 digital certificate may require very complex operations, which are costly.

Therefore, instead of using digital certificates, we use VC while are lightweight and, using a specific algorithm to create digital proofs, they can be easily verified by smart contracts.

Digital certificates may also be stored on other online services, which are cccejtralized and they are not owned, e.g., Interplanetary File System. However, these solutions cannot be directly used by smart contracts to retrieve information.

3.3 Summary

Chapter 4

Data representations

In this chapter, we will analyze two formats that allow to represent data in a compact and standard form. In particular, we will analyze the RDF format, which will be then used to compute the digital proofs inserted in Verifiable Credentials (see 6.2.2), and we analyze the JSON-LD format, which is used to represent DID documents (see 5.3) and Verifiable Credentials (see 6).

4.1 Resource Description Format

Resource Description Format (RDF) is a format that allows to represent pieces of information as a directed graph [13, Sec. 1.1].

In particular, the information to represent are organized in an RDF graph, which is a set of triples in the form:

$$(\texttt{subject}, \texttt{predicate}, \texttt{object})$$

where the `subject` and the `object` are two nodes of the graph linked together by an arc, while `predicate` is the label of that arc. In case there is no label on the arc connecting the `subject` and the `object`, the predicate is empty.

The RDF graph, therefore, will contain all the triples that describe all the nodes and arcs in the graph.

RDF categorizes the nodes of the RDF graph in 3 distinct categories, based on their content. In particular:

1. Internationalized Resource Identifier (IRIs), which are like Uniform Resource Locators (URLs), but they also allow the use of any UTF-8 character [14, Sec. 1.1];

2. Literals, which are used to define constant values like numbers, strings or boolean values. These literals are sequences of characters that represent the value (like `0.1`, or `true`) together with an IRI that uniquely identifies the type of the value

(like `http://www.w3.org/2001/XMLSchema#string` or
`http ://www.w3.org/2001/XMLSchema#boolean`);

3. Blank nodes, which are nodes that are neither IRIs nor literals. These nodes are not associated with any identifier. This implies that, using the RDF format, it is not possible to distinguish between two blank nodes.

   However, there are cases where it is imperative to associated with each node a unique indentifier. In these cases, it is possible to assign to each bloank node a so-called blank node identifier, which will uniquely identify the blank node, allowing to distinguish it from the others.

   The RDF standard does not impose any particular constraint on structure on the blank node identifiers, because it highly depends on the implementation [13, Sec. 3.4]

RDF imposes some restrictions on the information that can be stored in the nodes of the graph. In particular, as explained in [13, Sec. 3]:

- Subjects must be IRIs, or blank nodes;

- Predicates must be IRIs or empty strings in case there is no label on the arc connecting the subject and the object;

- Objects must be IRIs, literals or blank nodes.

The graph represented in figure 4.1.1 is an example of a directed graph that can be represented in the RDF format. The figure highlights in green the nodes that are IRIs, in yellow the nodes that are literals, and in blue the node that is a blank node.

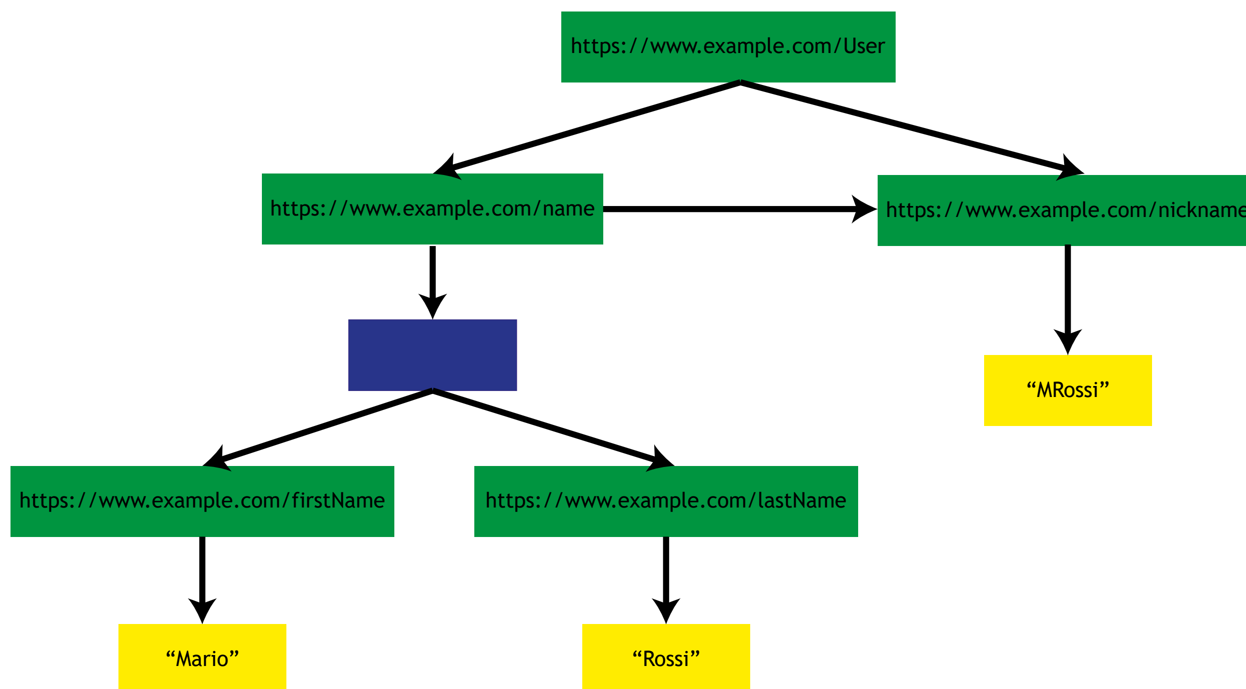

Figure 4.1.1: Example of an DRF graph

Listing 4.1 contains the representation of the graph in figure 4.1.1 using the RDF format, supposing to use (A) as a blank node identifier for the blank node.

RDF Listing 4.1: RDF representation of the graph

```
[https://www.example.com/User, https://www.example.com/Pred_1,
    https://www.example.com/name],
    [https://www.example.com/User, https://www.example.com/Pred_2,
    https://www.example.com/nickna
me],
    [https://www.example.com/name, , https://www.example.com/nickname],
    [https://www.example.com/name, https://www.example.com/Pred_3, (A)],
    [(A), , https://www.example.com/firstName],
    [(A), , https://www.example.com/lastName],
    [https://www.example.com/firstName,,"Mario"],
    [https://www.example.com/lastName,,"Rossi"],
    [https://www.example.com/nickname,,"MRossi"]
```

4.1.1 RDF canonicalization

As explained in the previous section, the RDF format represents nodes and arcs of a directed graph in the form of set of triples. Since it is a set, changing the order of triples does not change the described graph.

However, to create digital proofs starting from an RDF graph, we need a way to represent it as a unique string. Indeed, if we change the order of the triples in the set, the described graph is still the same, but the computed digital signature will be different.

The RDF Dataset Canonicalization algorithm is the solution to that problem. Indeed, given an RDF graph in input, the algorithm produces a string in output, and this string is guaranteed to be unique independently of the order of the tuples in the set [15, Sec. 1].

This allows creating the same digital signature out of the same RDF graph represented in two different ways using the RDF format [15, Sec. 1.1]. In particular, as described in [15, Sec. 1.1], the canonicalization algorithm will deterministically assign to each blank node a unique blank node identifier, so that the same black node will be associated to the same identifier, independently of the order of the tuples in the set that represents the RDF graph. Then, the algorithm will order the tuples in the set describing the RDF graph and, finally, it represents the set as a sequence of characters.

The canonicalization algorithm guarantees, therefore, that, starting from two RDF representations of the same graph, the resulting string will be the same.

The listing 4.2 shows the result of the application of the canonicalization algorithm to the graph depicted in figure 4.1.1.

RDF-CANON Listing 4.2: RDF caninicalization of the graph

```
_c14θ. ....%TODO:
```

4.2 JSON-LD

As the standard documentation [16, Sec. 1] states, “Linked Data is a way to create a network of standards-based machine-interpretable data across different documents and websites.” In particular, “starting from one piece of Linked Data, and application can follow embedded links to other pieces of Linked Data that are hosted on different sites across the Web.”

JSON-LD (JavaScript Object Notation for Linked Data), is a format that represents Linked Data using a JSON object [16, Sec. 1].

A conformant JSON-LD object is a JSON object where each property is an IRI [16,

Sec. 3]. The IRIs allows to unambiguously identify each of the values fo the properties specified in the JSON-LD document. Moreover, by accessing the resource identified by the IRI, we can retrieve a web page containing human-readable information explaining the meaning of property. In addition, to avoid misinterpreting a property name, each JSON-LD document is associated with a type, specified using the `@type` property.

Consider the example in listing 4.3. As you can see, the JSON-LD document is a JSON object describing a person. Indeed, from the `@type` property, we know that the type of the entity described in the JSON-LD context is `https://schema .org/Person` and, at that URL, we find a web page describing all the properties that can be associated to a person.

In particular, we can find:

- The `https://schema.org/givenName` property, which describes the first name of the person;
- The `https://schema.org/familyName` property, which describes the last name of the person;
- The `https://schema.org/url` property, which describes a URL associated with the person.

JSON-LD **Listing 4.3: Example JSON-LD document**

```
{
    "@type": "https://schema.org/Person",
    "https://schema.org/givenName": "Jane",
    "https://schema.org/familyName": "Doe",
    "https://schema.org/url": {
        "@id": "https://www.janedoe.com"
    }
}
```

However, the JSON-LD document in listing 4.3 is quite verbose because it repeats the prefix `https://schema.org/`. To reduce this verbosity, the JSON-LD format allows for the specification of contexts [16, Sec. 3.1]. In particular, using contexts, the JSON-LD document can be compacted by condensing all the IRIs used as names in the property. The JSON-LD parser will then take care of appending to each property name the URL of the context specified in the `@context` property.

For example, the JSON-LD document in listing 4.3 can be condensed in the document in listing 4.4, which is more compact and still allows a JSON-LD parser to correctly reconstruct the original document in listing 4.3.

JSON-LD **Listing 4.4: Example JSON-LD document with context**

```
{
    "@context": "https://schema.org",
    "@type": "Person",
    "givenName": "Jane",
    "familyName": "Doe",
    "url": {
        "@id": "https://www.janedoe.com"
    }
}
```

Note that, in the example, we have specified only one context in the `@context` propery. However, the JSON-LD standard allows for specifying multiple contexts in the `@context` property, in the form of a JSON array [16, Sec. 4.1]. In this case, each property name and type is prepended with the context URL that defines the meaning of that porperty.

In listing 4.5 there is an example of a JSON-LD document using multiple contexts. In particular:

- The `Person` and `Place` types together with the `address`, `givenName` and `familyName` properties are prepended with the `https://schema.org` URL, because it defines the meaning of those properties and types;
- The `geometry` and `coordinates` properties are prepended with `https://geojson.org/geojson-ld/geojson-context.jsonld`” URL, because it defines the meaning of those properties.

JSON-LD **Listing 4.5: Example JSON-LD document with context**

```
{
    "@context": [
        "https://schema.org",
        "https://geojson.org/geojson-ld/geojson-context.jsonld",
    ],
    "place": {
        "@type": "Place",
        "address": "1600 Amphitheatre Pkwy, Houston, TX",
        "geometry": {
            "coordinates": [107.04, 115.5]
        }
    },
    "owner": {
        "@type": "Person",
```

```
        "givenName": "Jane",
        "familyName": "Doe"
    }
}
```

A JSON-LD document can be represented as an RDF graph [16, Sec. 10], in which property names (after having being preprended with one of the context URLs) are the subjects, and their values are the objects. The predicates are absent, i.e., they are all empty strings.

For instance, the JSON-LD in listing 4.3 can be represented with the directed graph in figure 4.2.1

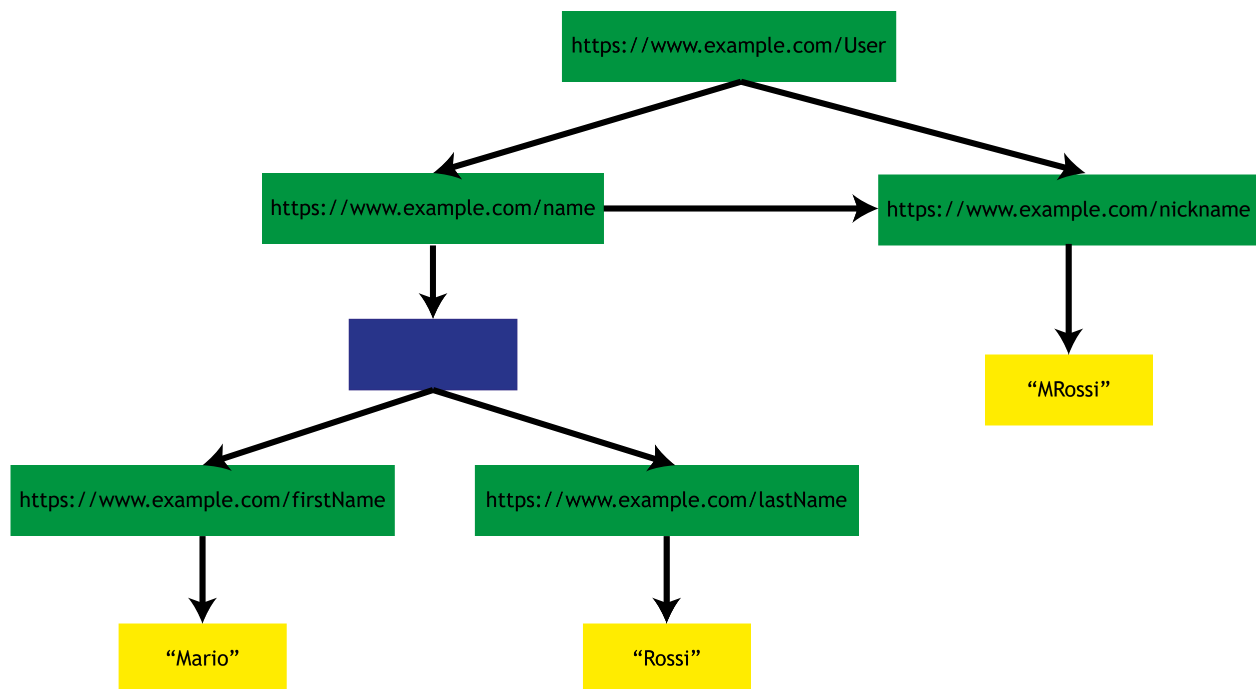

Figure 4.2.1: RDF graph computed from the JSON-LD document in listing 4.3

4.3 Summary

In this section we have detailed two RDF and JSON-LD. The first allows representing pieces of information as directed graph, which can then be uniquely represented as a string using the RDF canonicalization algorithm. The second, instead, allows information represented as JSON objects to be then represented as RDF graphs.

These two formats will be useful when computing digital signatures of Verifiable Credentials, as explained in 6.2.2.

Chapter 5

Decentralized Identifiers

In this chapter we will explain the first of the two building blocks of Self-Sovereign Identity, that is Decentralized Identifiers. These identifiers allow to uniquely identify each user but, at the same time, to guarantee their privacy.

As the W3C standard states, Decentralized Identifiers (DIDs) are a new kind of globally unique identifiers that allow users to prove control over them [19, Sec. 1]. The W3C standard refers to the users with the term *DID subject*, since they are the subject identified by the DID.

To guarantee more privacy for the users, they are suggested to use different DIDs when interacting with different entities [19, Sec. 10.2]. Indeed, if the user presents the same DID to different entities, they can collude to track the activity of the user. If, instead, the user presents two different DIDs, the entities receiving them are not able to know if the subject of those DIDs is the same user or two different users.

The W3C standard allows any entity to be identified using Decentralized Identifiers, also non-human ones. For instance, DIDs can identify the occupant of a specific role, like the CEO of a companyThis reduces discrimination, since any entity, when receiving a DID, cannot exclude a specific category of users (like, for instance, IoT devices) to perform some actions because there is no way to know if the DID subject is a physical person or not.

5.1 DID scheme

[19, Sec. 3.1]

The DID scheme defines the structure of the Decentralized Identifiers [19, Sec. 3.1]. The W3C standard requires DIDs to follow the structure contained in listing 5.1.

DID Listing 5.1: Basic DID scheme

```
did:<method>:<method-specific>
```

In particular, any DID must start with the string `did:`, followed by a string that identifies the DID method (see 5.5), and another string that uniquely identifies the DID subject according to the DID method.

5.2 DID URL

A DID URL is an identifier that identifies the location of a network resource [19, Sec. 3.2]. In particular, a DID URL is just a DID with the addition of the components that can be found in URLs.

The W3C standard specifies the usage of the query string to identify services [19, Sec. 3.2.1], while the fragment is used to identify verification methods.

DID URLs can be used, for instance, to reference parts of DID documents like verification methods or service (see 5.3).

5.3 DID Document

A DID document contains information about DID subject [19, Sec. 1.1]. In particular, as described in [19, Sec. 1.1], the DID document describes the cryptographic material the DID subject can use to prove control over the DID, as well as the cryptographic material that can be used to make assertions like Verifiable Credentials (see 6). Moreover, DID documents list URLs that can be used by other entities to establish a communication channel with the DID subject.

An example of a DID document is presented in listing 5.2. In particular:

- The DID subject is contained in the `id` property;
- The `authentication` property specifies the verification methods (see 5.3.1) describing the cryptographic material the user can use to prove their control over the DID;
- The `assertionMethod` property specifies the verification methods (see 5.3.1) describing the cryptographic material the user can use to make assertions, like Verifiable Credentials (see 6);
- The `service` property specifies the endpoints (see 5.3.3) where the DID subject can be contacted to exchange information.

, while the `authentication`, .

JSON-LD **Listing 5.2: Example of DID document**

```
{
    "@context": [
        "https://www.w3.org/ns/did/v1",

    "https://identity.foundation/EcdsaSecp256k1RecoverySignature2020/lds-ecdsa-secp256k1-reco
        very2020-2.0.jsonld",
        "https://www.ssicot.com/did-document",
        "https://ssi.eecc.de/api/registry/context/credentialregistry"
    ],
    "id": "did:ssi-cot-eth:5777:1234567890abcdef123456789abcdef123456789",
    "authentication": [
        {
            "id":
    "did:ssi-cot-eth:5777:1234567890abcdef123456789abcdef123456789#issuer-authentication-key",
            "type": "EcdsaSecp256k1RecoveryMethod2020",
            "controller":
    "did:ssi-cot-eth:5777:1234567890abcdef123456789abcdef123456789",
            "blockchainAccountId":
    "eip155:5777:d14DaC2057Bd0BEbF442fa3C5be5b2b69bbcbe35"
        }
    ],
    "assertionMethod": [
        {
            "id":
    "did:ssi-cot-eth:5777:1234567890abcdef123456789abcdef123456789#vc-issuing-key",
            "type": "EcdsaSecp256k1RecoveryMethod2020",
            "controller":
    "did:ssi-cot-eth:5777:1234567890abcdef123456789abcdef123456789",
            "blockchainAccountId":
    "eip155:5777:e14D5C265fBdfB7bF442fa3C30ed1b2b69bbe1e59"
        }
    ],
    "service": [
        {
            "id":
    "did:ssi-cot-eth:5777:123456789abcdef123456789abcdef123456789abc#issuing-service",
            "type": "CredentialRegistry",
            "serviceEndpoint": {
                "registries": [
                    "https://www.issuer-service.com/verifier",
                    "https://www.new-issuer-service.com"
                ]
            }
```

The information contained in the DID documents can be modified at any time. The entities that can make changes to a DID document are called DID controllers [19, Sec. 2]. Typically, the DID subject is also a DID controller, meaning that the DID subject can always make changes to their DID document. However, the standard allows the DID subject to specify other users, identified with other DIDs, that are allowed to make changes to the DID document.

5.3.1 Verification methods

A verification method describes a set of information that can be used by other entities to authenticate or authorize the interaction with the DID subject [19, Sec. 5.2].

An authentication is a verification method that contains information that can be used to authenticate the DID subject [19, Sec. 5.3.1]. For instance, it can contain the public key paired with a private key that the DID subject can use to authenticate to an external service by proving the knowledge of the private key. In particular, an entity can authenticate the DID subject by making them use the private key to sign a particular message. If the entity can then decrypt the message using the public key contained in the verification method, then the DID subject has proved the control over the DID and, hence, they are authenticated.

Another type of verification methods are assertion methods [19, Sec. 5.3.2]. These verification methods can be used by the DID subject to express claims, like Verifiable Credentials 6.

The W3C standard allows the DID subject to specify distinct verification methods to authenticate and to express claims, so to avoid an attacker to reuse public cryptographic material for purposes it is not intended to be used to. For instance, if the DID subject uses its private key sign expressed claims, an attacker may reuse the signature to authenticate as a user. The W3C standard allows not to incur in these situations by using different keys for authenticating the DID subject and for assert claims.

All the verification methods are uniquely identified by a DID URL (see 5.2). The cryptographic material associated to them is determined by the specific type of verification method. For instance, some verification methods may contain public keys, while others may contain references Ethereum accounts (see 5.3.2).

The verification methods include also the controller of that verification methods, which may not be necessarily the DID subject. The controller expressed the DID that can prove control over the cryptographic material that is described by the verification method.

For example, a parent controlling the DID of the child may insert, as an authentication verification methods of the child, a public key that is under the control of the parent. This allows the parent to authenticate as the child in any online service where the child can be authenticated [19, Sec. 2]. The controller property of the verification method will contain the DID of the parent, effectively disallowing the child to modify or remove that verification method.

5.3.2 EcdsaSecp256k1RecoveryMethod2020

The EcdsaSecp256k1RecoveryMethod2020 standard, detailed in [17] details the format of cryptographic material the DID subject can use to authenticate or to to express claims by using the private key associated with an Ethereum account (see 7.3).

The EcdsaSecp256k1RecoveryMethod2020 standard represents verification methods as shown in the listing 5.3 where the `blockchainAccountId` property contains the address of the Ethereum account whose private key can be used by the DID subject to authenticate or assert claims.

The address of the Ethereum account is prepended with the ID Ethereum chain the account belongs to, according to the EIP-155 standard (see 7.3.4).

JSON **Listing 5.3: Example of an EcdsaSecp256k1RecoveryMethod2020 verification method**

```json
{
    "id": "did:example:123#key-0",
    "type": "EcdsaSecp256k1RecoverySignature2020",
    "blockchainAccountId": "eip155:1:0x89a932207c485f85226d86f7cd486a89a24fcc12"
}
```

5.3.3 Services

Services describe service endpoints, which are network addresses like URLs that can be contacted to communicate or interact with the DID subject [19, Sec. 2].

Each service, like verification methods, is uniquely identified by a DID URL [19, Sec.5.4]. In this way, we decouple the location of the service from the identifier that identifies it. Indeed, if the DID subject moves the service to another location, the subject is just required to change the service endpoint in the DID document, without the need to contact all the entities that use the service warning them that the service endpoint has changed.

Additional information may be included to further describe the service, like a description of the steps to perform to establish an encrypted communication channel when accessing the service endpoint [19, Sec. 5.4]

A service described in the DID document may be used, for instance, by an online website that wants to authenticate the users. In particular, the website may present a QR code to the user that wants to autenticate, and this QR code encodes the DID URL of the login service. Once the user takes a picture of the QR code, the user's device can contact the service identified by the DID URL to start the authentication process, e.g., a challenge-response protocol where the server challenges the user to prove they have control over a specific DID.

5.4 CredentialRegistry

The CredentialRegistry is a type of services allowing the DID subject to express service endpoints where users can ask for Verificable Credentials [18, Sec 6.2.3].

In particular, by sending an HTTP request to one of the listed service endpoints specifying a DID, the user will receive a verifiable credential. The listed endpoints,typically require the users to authenticate by proving control over the specified DID.

Listing 5.4 (source [18, Example 30]) contains an example of the specification of a CredentialRegistry service in a DID document.

JSON **Listing 5.4: Example of an CredentialRegistry service**

```json
{
    "service": [
        {
            "id": "did:example:123#vcregistry-1",
            "type": "CredentialRegistry",
            "serviceEndpoint": {
                "registries": [
                    "https://registry.example.com/{credentialSubject.id}",
                    "https://identity.foundation/vcs/{credentialSubject.id}"
                ]
            }
        },
        {
            "id": "did:example:123#vcregistry-2",
            "type": "CredentialRegistry",
            "serviceEndpoint":
    "https://ssi.eecc.de/api/registry/vcs/{credentialSubject.id}"
        }
    ]
```

```
}
```

5.5 DID method

The DID method describes all the operations that can be performed on a DID document [19, Sec. 8].

In particular, the DID method describes which CRUD operations are allowed on the DID document (like addition, modification and deletion of verification methods and services), and which entities can perform such operations.

Note that the DID methods are not required to implement all the CRUD operations [19, Sec. 8]. For instance, the DID method may allow the addition and deletion of verification methods, but not their modification.

5.6 DID resolver and DID URL dereferencer

The DID resolver is a system component that, given in input a DID, returns the DID document describing it [19, Sec. 2]. The DID URL dereferencer, instead, is a system component that, given in input a DID URL, retruns in output the resource identified by that URL [19, Sec. 2].

DID resolver and DID URL dereferencers may accepts specifying DID and DID URL that conform one or more DID methods [19, Sec. 7].

5.6.1 DID resolution

DID resolution is the operation of resolving a DID into the DID document describing it [19, Sec. 7.1].

The W3C standard allows to perform the DID resolution using two different functions[19, Sec. 7.1]:

1. `resolve`, which given a DID, resolves it to the associated DID document, and returns the document as a sequence of key/value pairs;

2. `resolveRepresentation`, acts like the `resolve` one but, instead of returning a sequence of key/value pairs, it returns the information in a specific data representation (like a JSON-LD document;

Since the `resolve` function is quite restrictive in the representation of the returned information, we will only consider the `resolveRepresentation` function, because it is more flexible by allowing to return DID documents in different formats.

The DID resolution process can be customized by passing to the DID resolver specific options [19, Sec. 7.1]. For instance, specifying to the `resolveRepresentation` function the JSON object presentedin listing 5.5, we are requesting the resolver to return the DID document as a JSON-LD document. In this case, if the resolver does support that representation, a `representationNotSupported` error is generated.

JSON **Listing 5.5: Example of DID resolution options**

```json
{
    "accept": "application/did+ld+json"
}
```

The process of resolution must return a JSON-LD object with the structure presented in listing 5.6.1 (real values are subsituted with `...`), where:

- `didDocumentStream` contains the resolved DID document in the requetsed representation, e.g., JSON-LD document;
- `didDocumentMetadata` contains some metadata about the DID document, like the date it was created, or the date it was last update or whether the DID has been deactivated;
- didResolutionMetadata, instead, contains some metadata on the result returned in *didDocumentStream*, like the MIME type of the representation used to represent the DID document.

JSON **Listing 5.6: Example of the result of DID resolution**

```json
{"@context":"....",
    "didResolutionMetadata": {"conetntTRype":""},
    "didDocumentStream":{"..."},
    "didDocumentMetadata":{
        "created":"....",
        "updated":"...",
        "deactivated"}
```

In case of errors, the result will be a JSON-LD object the structure presented in listing 5.7, where `error` is be a string that identifies the error. For example [18, Sec. 10]:

- *invalidDid* states that the DID is not conformant to the syntax exposed in 5.1;
- `notFound` states that the DID document has not been found;
- `methodNotSupported` states that the DID resolver does not support the DID method;

- `interalError` states that there were problems in performing the action (like not able to contact the netwrok to retrive the information).

JSON **Listing 5.7: Example of the result of DID resultion in case of errors**

```
{"@context":"....",
    "didResolutionMetadata": {"conetntTRype":""},
    "didDocumentStream":{"..."},
    "didDocumentMetadata":{
        "created":"....",
        "updated":"...",
        "deactivated"}
```

5.6.2 Did URL dereferencing

DID URL dereferencing is the process of taking a DID URL and returning the resource identified by that DID URL [19, Sec. 7.2]. Examples of returned resources can be an authentication, an assertion method or a service.

Like for the DID resolution (5.6.1) the DID URL dereferencing process can be customized by specifying additional options. For instance, specifying as options the JSON oject presented in listing 5.5, we are requesting the DID URL dereferencer to represent the returned resource as a JSON-LD document. As for the DID resolution, in this case the specified representation is not supported by teh DID URL dereferencer, a `representationNotSupported` error is generated.

The DID URL dereferencing process must return a JSON-LD object with a structure similar to the object presented in listing 5.8 where:

- `contentStream` contains the resource identified by the DID URL;
- *contentType* expresses the MIME type of the resource contained in `contentStream` property;
- `contentMetadata` contains some metadata relative to the `contentStream`, like creation date or the last update date.

  Note that the metadata highly depends on the type of resource contained in the `contentStream` property. For instance, if the DID URL references an entire DID document, `contentMetadata` must be the same structure we find in the `didDocumentMetadata` property of the JSON-LD object returned when performing a DID resolution (see 5.6.1).

JSON Listing 5.8: Example of the result of DID URL dereferencing

```
{
    "@context": "....",
    "dereferencingMetadata": {
        "contentType": "...."
    },
    "contentStream": {
        "..."
    },
    "contentType":{
    "..."
}}
```

In case of errors, isntead, the result returned by the DID URL dereferenced will be a JSON-LD object with the structure presented in listing 5.9, where the `error` property is a string identifying the error that occured. For example[18, Sec. 10]:

- `invalidDidUrl` states that the ID URL is not conformant to the DID URL syntax exposed in 5.2;
- `notFound` states that the resource referenced by the DID URL has not been found;
- `methodNotupported` states that the DID resolver does not support the specified DID method;
- `interalError` means that there were problems in performing the action

JSON Listing 5.9: Example of the result of DID URL dereferencing in case of errors

```
{
    "@context": "....",
    "dereferencingMetadata": {
        "error": "..."
    },
    "contentStream": {},
    "contentMetadata":{}}
```

5.7 Summary

In this chapter we have detailed the first of the two building blocks of SSI, Decentralized Identifiers. We described why DID are important in SSI, how they guarantee privacy of

the user, how DID documents are useful to obtain information related to the subject of the DID (like public keys, or services), and how to retrieve the DID document (or a part of it) associated with a DID.

Chapter 6

Verifiable credentials

Verifiable Credentials are the second main building block of SSI. They can be used by the user to prove the coorectness of specific claims. In particular, the verifiable credential is digitally signed by the issuer of the credential, which gurantees the VC has not been tampered with, and guarantees the authenticity, i.e., guarantees that the issuer has really isssued that VC.

VC are JSON-LD objects 4.2 and they contain the claims the user is claiming. In addition, additional infromation are present, like the identifier of the issuer that has issued it, the issuance date so not to use VC that has been issued in the future, expiration date, so to reject VC that have been expired, and the proof guaranteeing the integrity of the VC (6.2).

An example of VC is the listing 1.

```
    {@context:[ ... ],
id:....,
credentialSubject:{....},
type:[VerifiableCredential, ..],
    issuer:"",
issuanceDate:"",
expirationDate:"",
credentuialStatus:{ ... },
proof:{ ... }}
```

As you can see, the VC contains the follwoing info:

- The `@context` field contaisns the JSON-LD contexts, Note that the URL must be the first one and must be always present to properly resolve the terms of the standard;
- The `id`, whhich uniquely identifies the VC. However, this may provoke a correlation 6.6;

- The `credentialSubject` field contains the claims of the verifiable credential. The claims effectlively present depend on what the user wants to prove;
- A set of types, which describe the meaning of the fields in the credentialSubject. Note that VerifiableCredential is required;
- The issuer of the VC, indentified using a DID (see 5);
- The issuance date, before which the VC cannot be considered valid;
- The expiration date, after which the VC cannot be considered valid;
- credentialStatus, which contain ifnromation that can be used to retrieve the status of the verifiable credential (e.g., revoked or suspended). The ifnromation contained depend on the stype of credential status;
- The proof, which integrity protects the content. The fields in the proof depend on the type of proof generated.

Verifiable credentials make use of DIDs. This allows more privacy for the user. For instance, two verifiers cannot collude by sharing their infromation to know on the user much more than the user has shared with the single verifier. Indeed, since the user may use two different DIDs to present VC to the two verifiers, they cannot merge the ifnromation of the same user because they do not know whcih DIDs belong to the same user.

6.1 Trust model

The trsut model of VCs is depisced in the picture 6.1.1.

In particular:

- The user trusts the issuer;
- The verifier trusts the issuers;
- The issuer does not trust the user. It trust them only when the user authenticates, e.g ., it proves the control over a DID;
- The veirfier does not trust the user. It trum them only after authentication, so that the user has proven the control ofver the DID reported in the credentialSubject.id field.
- The issuer does not know the verifier, so no trust is required.

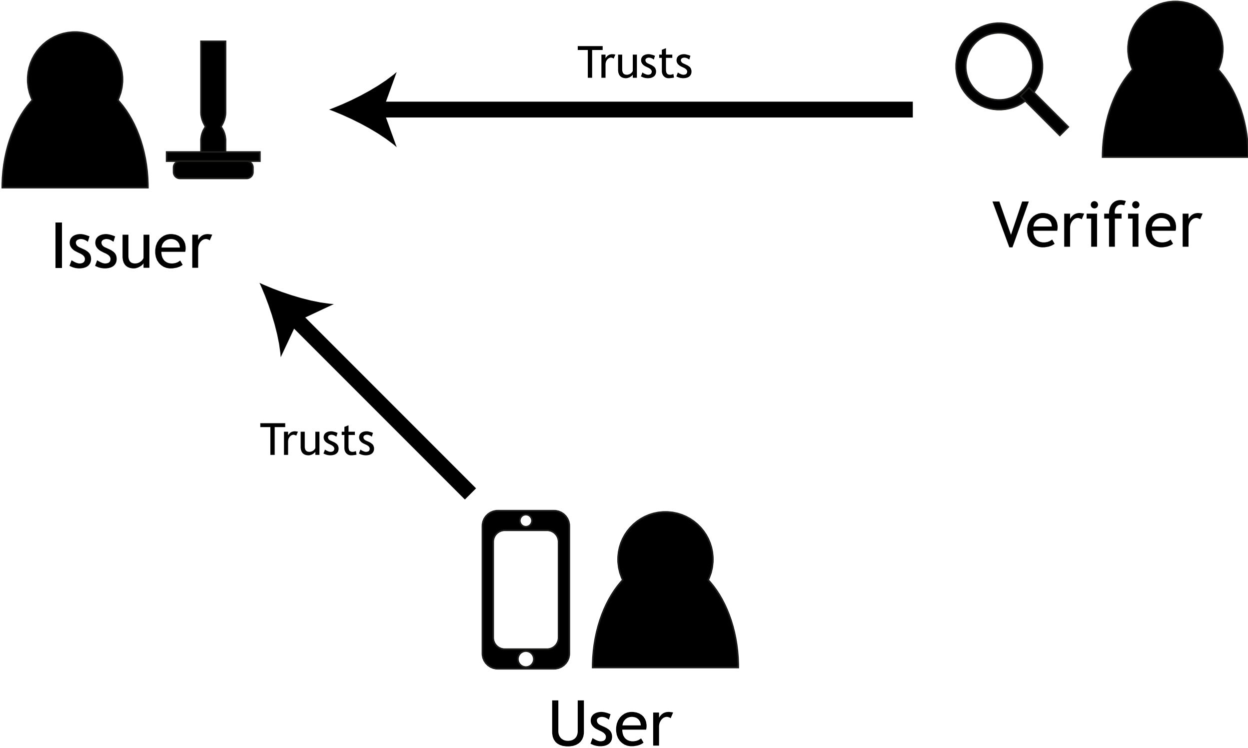

Figure 6.1.1: Trust model assumed by Verifiable Credentials

In our implementation, we will extend this trust model by allowing the verifier to accept VC issued by non non-trusted issuer provided that another issuer, which is trusted by the verifier, trusts the issuer of the VC.

The verifiable data registry is a foruth component that holds information that allow to tamper-evident and to be a correct record of which data is controlled by which entity.

The verifiable credentials trust model is as follows:

The verifier trusts the issuer. To establish this trust, a verifiable credential is expected to either: Include a proof establishing that the issuer generated the verifiable credential; Have been transmitted in a way clearly establishing that the issuer generated the verifiable credential and hat the verifiable credential was not tampered with in transit or storage. The holder and the verifier trust the issuer to issuer true (i.e., not false) credentials about the subject, and to revoke them quickly when appropriate; The holder trusts the repository that stores and protects the access to the holder's verifiable credentials. In particular, the holder trusts the repository to store the credentials securely, not to release them to anyone other than the holder, and not to corrupt or loose them while they are in its care All entities trust the verifiable data registry to be tamper-evident and to be a correct record of which data is controlled by which entity. Note that:

The issuer and the verifier do not need to trust the repository storing the holder's verifiable credentials; The issuer does not need to know or trust the verifier.

6.2 Data Integrity Proofs

Data Integrity Proofs allows the issuer to integrity-protect the content of VC. This allows users to present the VC without requiring the verifier to directly contact the issuer to check the validity of the ifnromation stored in the VC.

Typically, Data Integrity Proofs typically are digital signatures, so to guarantee also te authenticity of the VC/VP, i.e ., to prove that the VC/VP has been issued by the issuer claiming to have issued it.

Indeed, using the digital signature, if the public key of the issuer is publicly available, anyone can verify that the issuer specified in the VC is effectively the one that has signed it.

To create a digital signature, as described by the standard, the VC should first be canonicalized so to represent it in a unique form, independently of the order of the fields in it (see 4.1.1 for additional details). Then, the result of the canonicalization is hashed with an hashing algorithm and the resulting hash is signed with the private key of the issuer.

Transformation algorithm: takes the VC and computes a string from it Hashing algorithm: takes the result of the transformation algorithm and computes an hash Signature algorithm: takes the hash and computed the digital signature.

The proof is associated with a proof purpose. This allows the proof not to be used in contexts where the user don't want to. For instance, a proof with proof purpose=assertionMethod cannot be used to authenticate the user. This allows not to take proofs of verifiable credentials and try to use uthem to athenticate an attacker as the user.

6.2.1 JSON Web Signature

JSON Web Signature (JWS) allows to represent contennt that is integrity-protected. In particular, it is composed of 3 parts, separatect by a dot `.` character:

1. Header, which is a JSON object containing infromation of the JWS, like the algorithm used to digitally sign the content. It must contain at least the `alg` header parameter, which specifies the algorithm used to compute the digital signature. It may also contain the crit parameter, which specifies the array fo header parameters that the receipient must know and process in order ot validate the JWS.

   JWS distinguish between protected and unprotected headers: the first are integrity protected with the gital signature, the second does not. In the implementation we realized, we make use only of protected headers.

2. Payload,which is the content to integrity protect;

3. signature, which is the digital signature of the payload and the protected header, separated by a dot `.` character

To allow JWSs to be placed into URLs or contenxts where there are reserved symbols, each part of the JWS is base64url-encoded (see 6.2.1).
In listing 4 there is an example of JWS.

```
aaaa.bbbbbb.cccccc
```

where:

- aaaa is the header, which is the base64url encoding of the JSON object ....;
- bbbb is the payload, which is the base64url encoding of the string .....;
- ccccc is the digital signature, creted using the .... algorithm.

There are cases where encoding the payload is not necessary or it in unfeasible. Consider for instance, a case where you want to integrity protect a movie file. These file may be tenth of gigabytes. In this case, base64url encoding is unfasible because the reuslting encoded payload will much bigger. The Unencoded Payload option, allows to create JWSs with an undencoded payload. To not create trubles with teh standard JWS, it adds the b64 paremetr to the ehader and the crit header must contain the string b64.

There are also cases where the payload does not need to be sent. In particular, it the recipient can reconstruct it in some way, it is not neccessary to include it in the JWS. A JWS like so, uses the Detached Payload option appendinx F This allows to reduce the information sent to just the header and the digital signature, saving bandwith and reduucing the latency. However, the recipient must be able to recontrsuct the whole payload, otherwise it cannot verify the signature. This type of JWS is used to guarantee the integrity of content that is known both by the sender and the receiver.

It is possible to distinguish detached payload from non detached payload by checking if the payload is empty or not in the JWS: if it is, then the payload is detached.

Base64 and Base64URL

Base64 is a format to encode an arbitrary sequence of bits. In particular, the sequence of bits is consireder as a sequence of groups of 6 bits. Each group is then considered as an unsigned integer number ranging from 0 to 63. Finally, each of these numbers is converted to a character according to the following rules:

1. numbers from 0 to 9 are converted to characters from '0' ot '9';
2. numbers from 10 to ... are converted to upper case letters from 'A' to 'Z',
3. numbers from .... to 61 are converted to lower case letters from 'a' to 'z';
4. number 62 is converted to the charatcert +;
5. number 63 is converted to the character /

The base64 format cannot be used in contexts where / and + have a sepecific meaning. For instance , in URLs, the / character is the separator of the paths coponents

Base64URL format is similar to base64, but it substitutes the caharcter + with - and the character / with _, leaving all the other as they are. The resulting string can be now inserted in URLs.

This allows to easily use base64 encoders and decoders to also encode enad decote base64url -encoded strings. Indeed, it is sufficient to subitute + and / after with - and _encodning with ase64, and subsitute - and _with + and / before decoding.

6.2.2 EcdsaSecp256k1RecoverySignature2020

This is a type of data integruty proof

It uses RDF caninocalization as transformation algorithm, SHA256 for the hashing algorithm and ECDSA with recovery bit as signature algorithm.

The resulting signature is inserted in a JWS with detached payload and unencoded payload option. Therefore, the unendoced header of the JWS will look like listing 3.

```
    {
  "alg": "ES256K-R",
  "b64": false,
  "crit": ["b64"]
}
```

Note the use of b64 due to unencoded payload, and the b64 in the crit header paramter because the recipient must know and understand the meaning of the b64 header parameter before processing the JWS.

A proof created with this type of data integrity proof algorithm looks like the following

```
    {
  "type": "EcdsaSecp256k1RecoverySignature2020",
  "created": "2020-04-11T21:07:06Z",
  "verificationMethod": "did:example:123#vm-3",
  "proofPurpose": "assertionMethod",
  "jws": "eyJhbGciOiJFUzI1NkstUiIsImI2NCI6ZmFsc2UsImNyaXQiOlsiYjY0Il19..pp9eiLCMfN4E
tgUaTfByDaaB6IZbXsnvIy5AUIFjbgaiFNtq9-3f8mP7foD_HXpjrdWZfzlwAE"
}
```

For instance, consider the VC in listing 1 and 2. They contain the same ifnromation but, if we compute the hash directly from them , we end up with two different hashes and, therefore, with two different digital signatures.

RDF dataset canonicalization allows to express an RDF dataset into a standard format, so that it can be digitally-signed for later verification.

```
{
    proof: ...
    credentialSubject:..
    issuer:...
    issuanceDate: ...
}
```

```
{
    issuer:...
    issuanceDate: ...
    proof: ...
    credentialSubject:..

}
```

Therefore, to avoid this problem, we need a form of canonicalization that, given the two VC before, or any other permutation, gives in output the same string.

6.3 Zero-Knowledge Proofs

Zero-Knwoledge Proofs are ways to create claims without revailing the data of the claims. cite ZPK

For instance, we can proove to be over 18 without revealing the birthdate.

The verifier can verify the claims are correct because the ZPK are cryptographically tight to the claims they are generated from.

ZPKs are a step further in the SSI ecosystem by enancing the privacy of the user. VC support for ZPK.

6.4 Credential status

VC can be revoked at any time due to different factors. To support correctly the credential status, VC allows to specify URLs that, once dereferenced, result in information telling the status of the VC.

The only registered credential status standard in the VC ext reg is teh CredentialStatusList2017 An improvement is CredentialStatusList2021 , but both do the same. This standard allows to revoke verifiable credentials. In particular, once a VC is revoked, it cannot be used anymore, i.e., the verifier must discard it.

In particular, each issuer is associated with a string of bits, where in position i there is 1 if the i-th verifiable credential has been revoked. The bit string is then packed, so to reduce its size.

In the VC, a URL is placed together with an index x. When this URL id dereferenced, the bit string is returned. The verifier can know if the VC has been revoked by checking if at index x there is a 1 or 0.

This allows to know which VCs has been revoked. Indeed, the verifier can cache the result and, if receiving a VC with index y, can check if the y-th bit of the bit string is 1. If it is, then the VC is revoked for sure. If it is 0, then the verifier must dereference the URL so to obtain an updated version of the bit string.

However, the standard requires the bit string to be reified at creation, i.e., it must be created a bit string of a specific size filled all with 0, and this size cannot be changed.

6.5 Verifiable Presentations

Verifiable presentations allow the user to present in a single unit one or more verifiable credentials. Moreover, the user presenting the presentation may not be the subject to which the verifiable credentials are issued to. The standard refers to such a user as ”holder”.

In case the holder is not the subject, then it is up to the verifier accepting or not the credentials. Indeed, to properly delegate the holder to use the credential of a subcjet, a special verifiable credential stating the delegation should be issued by the subject, and included in the presentation.

6.6 Risk of correlation

To avoid the risk of correlating the user, i.e., for a verifier to understand that behind two different DIDs there is the same users, identifiers in the verifiable credentials should be omimitted when possible.

For instance, connsider the following two VCs:

```
credSubj: {id: did:example:123, name: Mario surname: Rossi}

credSubj: {id: did:example:456, name: Mario surname: Rossi}
```

Even if the two subjects have the same name and surname, the verifier cannot be sure they are the same user. Indeed, they may be omonimous.

However, consider the following two VCs:

```
id:”issuer/abc”, credSubj: {id: did:example:123, name: Mario surname: Rossi}

id:”issuer/abc” credSubj: {id: did:example:456, name: Mario surname: Rossi}
```

Even if the two verifiable credentials are issued to two different DIDs, the verifier can understand that the two VC have been issuet to the saem exact usser, which is using two different DIDs.

6.7 Summary

In this chapter we have exposed the concept of Verifiable Credential. In particular, we have underlined its importance in the SSI ecosystem, and how to integrity protect the content.

Chapter 7

Blockchain

Born in 2008 by a person with the psuedonim Satoshi Nakamoto, the blockchain is a way to store data in an immutable and distributed fashion. The blockchain is also called distributed ledger technology.

Nakamoto prposed its use so to exchange money, in form of Bitcoins, without the need of instritutions like banks or governments. Moreover, no authority controls the chain, hence anyone can exchange money.

The computers particupating in the maintainment of the blockchain, in the check of the transactions and in the proposal of new blocks are rewarned by freshly created Bitcoins, plus fees gives by the people sending the transactions included in the block.

Anyone can send a transaction to the network. This transaction is put on a glibal pool, called mempool, from which each miner will extract the transactions to put in the next block of the chain.

The fee model used by Bitcoin is quite unfair, becaseu block miners will be incentivated to insert in a new block all the transactions with a higher fee.

Every node contains the hash of the previous block and the hash of the current block. Therefore, it is not possible to modify a block without the need to modify all the subsequent blocks. The

7.1 Blockchain ownership

While the Bitcoin blockchain is public and has no owned, the idea of a blockchain has been implemented.

We can distinguish between permissioned and permissionless blockchain.s . The first, the users requires to be inserted in the newtrork in order to participate, in the other they can freely participate.

Blockchains may also be private or public. The first are accessible only by the ones that knws it , while the second are accessible by anyone.

7.2 Ethereum

Ethereum can be thought as a distributed computing system more than a way to exchange money. Indeed, it allows to execute programs, called smart contracts. Since the execution of a program requires computing power, anyone that wants to execute smart conracts need to pay.

The paymant is done in ETH, which is the native value of the Etehereum network. Since the value of ETH is high, we ususlayy work with submultiples of ETH, in particular, wei which is 10^{-18} ETH and gwei (giga wei) which is 10^{-9} ETH or, equivalently, 10^9 wei.

Etehreum is a public and unpermissioned blockchain, menaning taht anyone can participate and send transaction to the network.

Addresses are identifiers to uniquely identify accounts and smart contracts. In particular, they are 40 bytes-long, and the address is directly computed starting from the public key.

Any entity having an address can hold ETH. Therefore either accounts and smart contracts can hold ETH.

7.3 Accounts and wallets

An account is a user that can access the network to make transactions or to propose new blocks. To create an account, it is sufficient to generate a pair of public and private keys.

A wallet is just a container of the private and public keys.

Elliptic Curve Digital Signature Algorithm (ECDSA) is an algorithm thata llows to compute a public and a private key in a asymmetric key criptography context.

In particular, while the private key is just a random sequence of bytes, the public key is computed with an algorithm. Ethereum uses ECDSA with the secp256k1 constants to generate the public key.

Note that the balance is not stored in the wallet but it stored in the blockchain. Therefore, to access the balance of an accoutn, the user must access the blockchain and recontruct the whole history of deposit and withdraw done with that account in order to compute the balance of the account.

7.3.1 Smart contract

A smart contract is just a program than can be run by the nodes in the Etehereum network. In particular, it is like a class in object-oriented programming languages, and it exposes methods to users and other smart contracts.

Any method can be called by sending a transaction to the smart contract, specifying in the content of the transaction the index of the method to execute, and all the paremters it needs, properly encoded.

Smart constract, to be executed, must be deployed to the entwork. To do so, the creator of the smart contract needs to send to the enetwork a transaction, specifying as the sender the address of where the smart contract will be deployed, and the content f the transaction will be the code fo the contract.

A smart contract can call any method exposed by the smart contract itself. Moreover, the smart contract can also call methods exposed by other smart contracts. However, this requires the first smart contract to peroform a transaction, which has a much higher gas cost thatn calling a method inside the smart the smart contract itself.

7.3.2 EVM

The smart contracts need to be executed by any machine in the netwkork. Since the machines are different, smart contracts cannot be written in a machine-specific code. Therefore, smart contracts are written in byte-code, and then executed by an interpeter called Ethereum Virtual Machine (EVM).

The EVM is run by every node of the network.

Unlike most other applications, the EVM puts some limits in the size and in the complexity of the smart contracts. In particular, the smart contracts cannot have a size bugger than 24.576 KiB. Moreover, the more a method of the smart contract is complex, the more computing power Thta's why, for complex smart contracts, optimization is needed.

Typically, like for machine-language, smart contracts are not directly written in byte-code, but they are wiritten using a high-level language, called Solidity, that is then compiled to the EVM bytecode.

This also allows to easily create other languages to create smart contracts, provided that a compiler is able to produce bytecode the EVM can understand.

7.3.3 Gas

Any instruction executed by the EVM has a cost. However, instead of measuring it is ETH (or submultiples like gwei or wei), each instruction is asoociated with a cost measured in gas units. The more the instruction is complex, the more its cost in gas units.

The cost of each instruction is fixed. The use of gas allows to decouple the complexity of the instruction from its real cost in ETH.

Note that while methods that require to write on the blockchain require the user to send a transaction of the network, methods that perform only read-only operations can be directly executed by the client. This allows not to pay to execute read-only transactions because no transaction is sent.

However, even if the smart contract code is public, it is not so easy to compute the gas that a transaction uses. For instance, if the contract contains loops based on data stroed on the chain, it may be difficult to determine how many time the cycle will cycle.

Therefore, to properly pay the transaction, Etehreum instroduces the concept of gas limit. In parituclar a user, when sending a transaction, specifies in it the gas limit, and the value of the gas limit. For instance, if you specify a gas limit of 10000, then you pay the cost in ETH of 10000 gas. When the transaction is executed, the node executing the transaction will use up to the gas limit to execute the instructions. If the total gas used to execute the transaction is less than the gas limit, then the difference is refunded to the sender. For instance, if you set the gas limit to 10000 and only 6500 is used , then the remaining 3500 are refunded in ETH back to the sender. If, instead, the transaction exceeds the gas limit, the node aborts the execution of the transaction as soon as the gas limit is excceded. The transaction is then reverted, i.e., all the effects produced by the transaction are cancelled. The user will not receive any result back already computed by the method, and no gas is refunded.

The cost of the gas is proportional to the load of the network: the higher the load, the higher the cost of the gas. This allows to reduce the number of transactions sent to the netwkork because the cost higher.

The gas protects the network agains DoS and DDos attacks. Indeed, when the number of transactions sent to the netwok increases, the cost of the gas increses as well. Moreover, a transaction cannot run forever because at some point, it finishes all the gas that has been paid.

Note that the block proposer is not forced to pay for its transactions, i.e., it can add its transactions to the block it is porposing without paying.

7.3.4 Networks

There are several Etehreum blockchains available. The main chain is called mainnet. In this chain, all the transactions must be paid, meaning that the user has to lead ETH to their accounts before sending a transaction. Recall that also deploying a smart contract is a transaction.

Ethereum provides also two other test networks, called goerli and Sepolia. They can be used by developers to test the execution of their smart contracts in an environemnt which is similar to the one of the mainnet.

each netwrok is independent form the others, i.e., they contain different blocks.

However, the transactions are free, i.e., you do not have to pay to execute the transactions.

However, to avoid all the users to use test networks (so running applications that make use of the blockchain without the need to pay for transactions), test networks are by design much slower, so they can handle much less transactions that the main net.

There are also private Ethereum blockchains. This are typically used by developers to test their smart contracts before deploying them on the mainnet. With respect to testnets, local netwroks are faster because there is no network latency. Moreover, they do not impose any limit in the number of deployed contracts on in the number of transactions sent.

Ganache is an example fo local Etehereum blockchains. It simulates a blockchain with a bunch or accounts already loaded with ETH. Therefore, the devoper do not need to pay for executing the transactions, and have no latency due to the network becuase it is local. However, running an application on a local network is not robust, becuase if the node fails all the ifnromation are lost. Moreover, if the node is not connected to the Internet, the users cannot use the application anymore. Therefore local networks should be used only for development purposes only.

To uniquely identify each of these blockchains, the EIP-155 assigns a unique number that identifies each chain. Frr instance, the mainnet (the main Etehereum network) is identifie by 1, while a local private test network is identified with 1337.

7.3.5 Oracles

Smart contract cannot access extenal entities like files or URLs. This is because they might not be always available, or they may change during the time.

If it is imperative to access external entities, a smart contract can user oracles. However, oracles are difficult to use since they must always provide the same data every time it is required. Indeed, at any time, one can take the entire blockchain and execute all the transactions to retieve the current status of the chain. This is impossible if the oracle returns different resources every time it is called.

Therefore, oracles should be avoided when possible.

7.3.6 Consensus algorithms

Different nodes in the network see the transaction in different order. We need a way to choose the next block of the blockchain. Moreover, due to the network latency, some nodes may receive blocks after the other nodes, creating multiple views of the last block in the chain, e.g., if the blocks are A-B, a node may receive only A, hence one node says that the last block is B but the other says that it is A.

we need, tehrefore, a way to deterministically determine the real last block of the cahin when there are multiple options.

Etehereum uses two types of consensus algorithms that will be detailed in the following sections

Proof of Work

Before the London hard fork Proof of Work (PoW). In this type of consensus algorithm, all the nodes participating to the network are in competition to selve a mathematical problem, called puzzle.

In particular, the nodes are challenged to find the correct nonce that, added to the next block of the blockchain, will create a sequence of bytes whose hash begins with a specific amount of zeros.

Any node particupating in this competition can put in the next block of the blockchain whatever transaction it wants.

The winner of the competition will add the block to the blockchain, and receives all the fees from the transactions included in the block. All the other nodes have simply lose, and all the work they have done needs to be discarded.

The number of zeros varies so to maintain a specific mean rate of blocks produced per hour.

This type of consensus algorithm has been abandoned by Etehereum because it is not considered environemntally-friendly, since the miners are using compters to solve the mathematical problem, and they consume electricity to do so.

Proof of Stake

Proof of Stake (PoS)

Unlike Proof of Work, this solution is environmentally friendly because the comuters just need to listen to the transactions sent to the network and, every 12 seconds, they may be called to propose the next block of the chain. No computing power is used to create the new block, it is just composed by the chosen node and then propagated to the rest of the network.

To participate to the block poposal process, a node must stake at least 32 ETH. This means that the 32 ETH (or more) cannot be used by the user to make transactions, i.e., they are locked. In any time the user can un-stake the ETH. This are sent to a specific smart contract, called deposit smart contract.

Every 12 seconds a new block will be proposed. To do so 128 nodes are randomly chosen among all the nodes of the network, to form a comittee. The probability to be chosen depends on the number of ETH that are staked: the more you stake, the higher the probability to be chosen as the next block proposer.

Then, from the 128 nodes, 1 is randomly chosen, again based on the number of staked ETH. This chosen node will become the next block proposer. It will create the block by inserting the transactions it wants, it inserts the hash of the previous block and computes the hash of the resulting block

The block proposer will then propageate the created block to the other 127 chosen nodes of the network. These nodes will check the correctness of the block, i.e., if all

the transaction contained in the block are correct against the status of the network. They then vote for the block: ”yes, it is correct” or ”no, it contains some errors”. The block becomes part of the blockcchain only if at least 2/3 of the 127 nodes respond ”yes”.

If less than half of the 127 nodes are not reachable (online), or if the block proposer goes offline, then the no new block is added to the blockhain, and a new block will be proposed the next time slot.

The block proposer will receive great part of the the fees, while the remeining part is split among the 127 nodes that have participated.

A node can answer ”no” because transactions are received by the nodes in different orders. For example, if a user deposit 1 ETH and then withdraw 0.5 ETH, then the two transactions are logically correct. Indeed, if the block proposer receives these two transaction and inserts the second one , it kows it is correct because there is a transaction (that will be added).

All the received ETH by the varius nodes are added to the staked ETH, i.e., they acannot be use dto make transaction.

If a node does not behave correclty, part or all of the staked ETH will be burned, i.e., they will be sent to a wallet whose private key is not known by anyone, while everyone knows the public key.

Do not bahave correctly means: - Not responding while conatcted during the process of blockc proposal. This is because the other nodes are waiting for all the nodes to answer, and a malicious user that is selected to participate to the block proposal may delay tohe answer or not repond at all so to slow down or block the newtork; - Propose a block with wrong transactions inside. Indeed, a malicious user may insert a transaction ”A sends 100 ETH to B” even if A does not have 100 ETH. If A and B are two accounts owned by the maliclious users, that transaction may insert in the network more ETH that the ones that are in the network currently, and makes the malicious user rich even if it has no money.

After 12 blocks has been proposed, all the validators selected to validate the previous 12 blocks are asked to vote for the correct last block of the chain.

7.3.7 Digital signatures

Ethereum allows any account to digitally sign arbitrary sequences of bytes. Indeed, since every account is associated with a private and a public key, the user can use its private key to sign any sequence of bytes.

Etehreum allows smart contracts to verify those signatures. To do so, it makes use of the `ecrecover` function, which takes the hash of the message, the signature, and returns the address of the user that has signed the message. In case of errors, the function returns either 0 or an address that is different from the one of the user that has signed the message.

Of course, Etehreum does not allow smart contracts to generate digital signatures. Indeed, to do so, the smart contract requires to know the private key to use to sign. This implies that the user has to specify, when calling the mathod of the smart contract, as an argument the private key to use. However, recall that anything on the blockchain is public, included the arguments passed when calling a method of a smart contract. This implies that the private key of the user, specified as argument, will be published in a block of the blockchain, meaning that the account of the user will be compromized.

7.3.8 Attacks

An attacker may try to attack the network so to shuffling the order of the blocks (caller reorg) so to inlcude new blocks or exclude ones.

As exmplained by mouting a successful reorg attack would require the attacker to control at 66% of the total staked ETH.

As the time of writing, the total amount of ETH staked is 19,741,828 ETH. This implies that a 66% attack sligjlty more tahan 13 million ETH, which implies thay have to stake almost 23 billion dollars (sine 1 ETH= 1,742.42 dollars) . It is improbable that an attacker can stake that amount of ETH and, even if they can, the economic loss in case the attack doe not succeed is so high to prevent a possible attack. With less that 66%, the attack is difficult to mount, because it requires the attacker to ”collaborate” with the validators off-chain, possibly indicung a netwrok partiction, or by delaying messages.

Another attack if the finality delay in which the attacker prevents the network from reaching an agreement on the next block to add to the chain. In this case, the attacker should be chosen to be the block proposer. The attacker will withdraw its block until many validators has voted for the preiouvos block in the chain to be the correct head. Then the attacker releases its block. If the timing is correct, some other validators will vote for the attacker’s block to be the last block of the chain. This prevents the newtrok to reach the 2/3 of the majority, meaning that there is no agreement on the last block of the chain, and the prcedure should be repeated. This will slow down the network.

There is a solution to this problem which identifies and excludes the validators that are not attesting at all, or they are attesting opposite to the majority of the network (which is still less than 2/3 of the staked ETH). To exclude these validators, part of their staked ETH is burned , so that, at some point, the validators attasting the oppoiste of the majority will hold less than 1/3 of the staked ETH, meaning that the 2/3 will reach an agreement. This solution is called inactivity leak.

7.4 Summary

In this section we have presneted the Etehereum blockchain, and why it is useful to store immutable ifnromation that must be publicly available.

Chapter 8

Implementation

Our implementation is made of two parts, a TypeScript library and one Etehereum smart contarct. The smart contract will take care of storing the ifnromation regarding teh DID documenst, The library is just a wrapper around the smart contract, allowing to bring to the clients amny of the checks and parsing logic, so to make the smart contract easier and gas efficient.

In particular, we propose a new DID method, the ssi-cot-eth DID method.

Realizing the smart contarct was a challenge. Indeed, the smart contract needs to provide a lot of functionalities, and we didn't want to split the smart contract into many others, because it increases the gas needed.

The smart contract requires the compiler to optimize the code, because the bytecode exceeds the limit of 24.576 bytes. Instead of splitting the smart contracts into several others so to overcome the limit in size, we preferred keeping all the code inside the same smart contract, so that the gas cost is lower.

8.1 Chains of trust

In our implementation we allow the creation of chains of trust. In particular, a DID may have at most one parent in the chain of trust. If the same issuer wants to be part of chains with different parents, it must create different DIDs.

However, there is no limit in the length of the chain, meaning that the chain of trust can contain an arbitrary number of issuers.

To establish the chain of trust of an issuer, this issuer must first create a new DID or use an already existing one under their control. Then, the issuer must ask to the parent of the chain of trust a trust certification, which is a VC digitally signed by the parent in the chain of trust.

Then, the issuer can call the `updateTrustCertification` method of the smart contract, providing the trust certification as paramster. The smart contract will validate

the trust certification and, if valid, the chain of trust is established.

The smart contratc performs the check so that a user cannot specify an invalid verifiable credential. This check could be done on the library, but a malicious user can also skip the library and directly call the smart contract.

8.2 Authentication

The user is authenticated on a per-transaction basis, i.e., it must authenticate on every request it sends to update its DID document.

To be authenticated, the user speicifes the DID URL that iodentifies one of its authentication methods. Then, the user uses their private key to send a transaction to the Etehereum blockchain. The TypeScript library will do so behind the scenes.

The method of the smart contract will then perform tha authentication of the sender of the transaction. In particular, it takes the DID URL, splits in DID and fragment, acccesses the DID document relative to the DID and extracts the authentication method with the specified fragment.

If the auth method is not found, then the user is for sure not authenticated. If, instead, the auth method is found, the retrieved address is checked against the address of the sender fo the transaction. Only if the two addresses match, the user is authenticated, and it can perfrom modification to its DID document (the smart contract will check if the user tries to make modifications to other DID documents).

8.3 Role of the blockchain

The blockchain is used to store the ifnromation of the DID Documents. In particular, the DID, list of authentication and assertion methods, services and trust certification.

To save gas, not all the information oare stored. Only the one that cannot be recomputed from the others. For instance, the type of verification methods (auth and assertion) is not stroed because it is assumed to be EcdsaSecp256k1RecoverySignature2020. Moreover, the controller of the verification methods is not stored because it is assumed to be the DID subject.

8.4 library

The library supports, for digital signatures, the EcdsaSecp256k1RecoverySignature2020 standard. This standard requires the use of ECDSA with sekp256k1 constants to create the digital signature. This allows the smart contract to verify the digital proof directly, without requiring to contact other contracts or extenarl entities.

The proof is ecoded as a JSON Web Signature with detached and unencoded payload. This allows to reduce the size of the JWS because it does not contain the payload. Moreover, it reduces the gas cost of executing the verification method on the smart contract because the payload to hash must not be base64url encoded.

Moreover, there is no way for an issuer A to avoid to be added to the trust list of another issuer B to its trust list, is the issuer to add new members (in my implementation, issuers issue certifications, and it is up to the members decide to use it or not).

8.5 CRUD operations

The TypeScript library we realized acts as the DID resolver for the ssi-cot-eth DID method.

Therefore, the library allows the user to make modifications to their DID documents, as well as resolving DID into DID documets, dereferencing DID URLs and resolving the chain of trust of an issuer. In my implementation, the DID subject is also the only DID controller, meaning that the user cannot deleage another user to make modifications to their DID document.

Any CRUD operation requires the user to pay a fee because an Etehreum transaction is sent to the network.

8.5.1 Create

To create a new DID the user must just create a new Eteherum account, hence a pair of public and private keys. The DID will be teh following:

did:ssi-cot-eth:chain id:address

where chain Id is the id of the chain where the smart contract containing all the DID documents is deploywed, while address is the address of the account, without 0x, that ahs been used to create a new DID. Call to the createNewDid function to do so.

You can also create a new DID from al already existing account, by providing the private key. Call the createNewDidFromAccount to do so.

8.5.2 Read

Refer to 8.10, 8.11 and 8.12 for additional details on read operations.

8.5.3 Update

Once the user has created a DID, it can make modifications to the asscoiated DID document. In particular, the user can associated with it any number of additional Etehereum accounts, by updateing the DID document and adding new Etehereum addresses.

The user can also update existing auathehtocation and assertion methods, or even remove them. ‘addAuthentication‘, ‘addAssertionMethod‘ and ‘addService‘ updateAuthentication‘, ‘updateAssertionMethod‘ and ‘updateService‘ ‘removeAuthentication‘, ‘removeAssertionMethod‘ and ‘removeService‘

8.5.4 Delete (deactivate)

The user can also deactivate the DID, whcih implies that the DID cannot be used anymore to authenticate the user, or to issue new verifiable credentials. However, the DID document remain available to be resolved. This allows, for instance, to validate trust ceritfications issued to other issuer so that they become part of a chain of trust. If the DID document is removed, any verifier asking for the chain of trust of an issuer, and in this chain of trust there is a deactivated DID, whould not be able to verify the trust certification.

Moreover, not removing the DID document allows also the verifier to still verify VC the issuer has issued before being deactivated. Therefor,e even if the issuer is deactiated, the users can still present VC to verifiers, and verifiers can still verify the digital signature of the VC because all the ifnroamtion required are still available n the DID document of the issuer.

To do so, the developer calls the deactivate function.

8.6 DID document

The DID document stores verification methods (auth and assertion) fo tyep ECSA..., services of any type, and the trust certification (if any).

To support the trust certification, we extended the standard by adding the trustCertification field to the document. It contains all the infromation that are required to reconstruct the Trust certification. In particular, it contains the issuer, the issuance date, the expiration date, the DID URL to derence to obtain the status of the certification, and the digital proof (creation date, verification method and digital signature).

we do not allow a user to specify verificztion methods that are controlled by other DIDs different from the DID subject.

an example of a DID document is the listing

8.6.1 Reification

Since the user, when creating a new DID, does not interact with the blockchain, no information is stored on the chain. This allows any user to create DIDs when they want, without worring about loading some ETH on the newly created accounts.

However, when someone asks for the DID document of that newly created DID, no information can be retrieved from the chain. Therefore, the smart contract will return a DID documetn generated on-the-fly. In particular, the DID document has no services, no trust certification, and the authentication equal to the assertion method, which are both the address of the Ethereum account associated with the DID, and whose address is directly placed in the DID, as described by the DID scheme.

Any further operaition that requires update the DID document, will force the reification of the document itself. This implies that the operations will be costly the first time, but they cost less the next times. In this case, the resolution process will returns, for the metadatda, the current date and time ffor update and creation.

8.7 Services

The library supports any type of service. In particular, since the services may contain several infromation other that the DID URL and the type, the library forces the develpers to serialize all the additional information in a string.

Thta string will be then stored on the blockchain.

Once the DID document is resolved or the DID URL referencing the service is dereferenced, the library will ask the developer to parse the string into the object containing all the ifnromation of the service.

8.8 Trust certification

The trust certification is a VC issued by the parent of the DID in the chain of trust and presented to the smart contract by the DID subject. The VC has the following format:

```
{
"@context": [
    "https://www.w3.org/2018/credentials/v1",
    "https://identity.foundation/EcdsaSecp256k1RecoverySignature2020/lds-ecdsa-s
very2020-2.0.jsonld",
    "https://www.ssicot.com/certification-credential",
    "https://www.ssicot.com/RevocationList2023"
],
"type": ["VerifiableCredential", "CertificationCredential"],
"credentialSubject": {
    "id": "<DID of the DID subject>"
},
"issuer": "<DID of the issuer>",
```

```
    "issuanceDate": " ... ",
    "expirationDate": " ... ",
    "credentialStatus": {
        "id": "<Certification status URL>",
        "type": "RevocationList2023"
    },
    "proof": {
        "type": "EcdsaSecp256k1RecoverySignature2020",
        "created": " ... ",
        "verificationMethod": "<Issuer verification method>",
        "proofPurpose": "assertionMethod",
        "jws": " ... "
    }
}
```

In particular, it must contain:

- CertificationCredential in the type,
- credentialSubject.id is the leaf issuer of the chain of trust;
- issuer is the parent in the chain of trust;
- issuanceDate and expirationDate express the date and time the trust certification becomes valid and the date and time the trust certification will become invalid and must be renewed;
- credentialStatus conbatin ifnromation to retrieve the current status of teh trust certification (like revoked);
- The proof is conformant to EcdsaSecp256k1RecoverySignature2020 type (see 6.2.2.

Note that we force trust certifications to be periodically renewed. This housld incentivize the issuers of the chain to behave well, otheriwise the parents may not renew the trust certification. Moreover, this forces the parent of the chain to periodically re-check the trustworthiness of the issuers.

Moreover, in order to accept the certification, the user must trust the issuer, directly or indicercly though a chain of trust. The library allows the user to select which issuuers to trust directly and which ones can be trusted though a chain of trust.

The smart contract will perform several operations:

- Check if it has been issued to the DID subject
- Check if it has been issued in the future

- Check if it is expired
- Check if it has been revoked;
- Check the proof: retireve the address of the issuer by resolving the DID URL in the verificationMethod field. Then the smart contract recomputes the RDF caninicalization of the VC. Finally, computes the hash, calls ecrecover to compute the address of the entity that has produced that signature. Finally, it checks if the two addresses match and, if so, the trust certification is added to the DID document, possibly replacing the already associated one.

It is possible to remove a trust certification by calling 'removeTrustCertification'. In this case, the subject will not be part of any trust chain, and becomes the root of other trust chains in case there are childern.

8.9 Invitations

To establish the trust between the verifier and the issuer, we propose the use of invitations. They are JSON documents containing a field, `did`, which contains the DID of the issuer.

Additional fields may be present, like the name of the issuer.

An example of invitation of teh Ca' Foscary University can be:

```
{
did: " ... ",
displayName: "Ca'Foscari univeristy",
...
}
```

We propose this invitation to be downloadable from the official site of the issuer, or can be sent from a certified source (like certified e-mails).

This part is delicate from a security prospective, because if the user imports a certifiacation from a malicious user, they can trust any verifiable credential issued by the malicious user.

However, the user can distinguish between peer-to-peer issuers and chains of trust issuers. This implies that the user, when receiving a VC, can be asked to accept the VC because issued by an issuer that they directly trust, or because the issuer is part of a chain of trust that contains an issuer that the user trsust.

This allows the verifier, in any case, to reject VC issued by an issuer Y in which their chain of trust contains the issuer X that is trusted by the verifier. The cahin of trust allows, however, the verifier to trust the VC even if the issuer Y is not directly trusted, since X is trusted by the verifier and part of the chain of trust of Y.

8.10 DID resolution

The library, as said before, acts as DID resolver for the ssi-cot-eth DID method. In particular, it allows, given a DID, to resolve the corresponding DID document.

We propose an extension of the standard, allowing in case of error messages, to include the `errorMessage` field in the returned response so to detail the cause of the error:

```
    {@context:....,
didResolutionMetadata: {error:"",
erroMessage;:.....},
didDocumentStream:{},
didDocumentMetadata:{}
    }
```

The only type acceptable is application/json+ld. No other didDocument metadata are added form the ones examplined in chapter 5.

8.11 DID URL dereferencing

The library, as part of the DID resolver job, can dereference DID URLs. In particular, it allows to resolve DID URL referencing auth and assertion methods, services and credential status.

We propose an extension of the standard, allowing in case of error messages, to include the `errorMessage` field in the returned response so to detail the cause of the error:

```
{@context:....,
dereferencingMetadata:{
error: "...",
erroMessage:----},
contentStream:{},
contentMetadata:{}}
```

The only type acceptable is application/json+ld. The library does not amike use of any metadat for the content.

8.12 Chain resolution

This is a new operation that, given a DID, returns the chain of trust ending on the specified DID. For each issuer on the chain, the trust certification is returned together

with its revocation status or if they are expired , so that the clint can know which issuers are part of the chain, and can validate the trust certifications against revocation and expiration.

It returns the last 10 trust certification in the chain. The process can be iterated with the other issuers, but typically it is not necessary because the chains are emant to be short.

The chain resolution accepts options to be customized. In particular, the accpet option can be used to specify the MIMe type of the returned chain. Currently, only application/jsn-ld is supported. Any other MIMe type will result in the representation-NotSupported error.

An example of result returned by this operation is the following:

```
    {
    @context:....,
    resolutionMetadata:{contentType:....},
 trustChain: [....],
chainMetadata:{}
}
```

where trustChain contains the sequwence of trust certgifications enabling the chain of trust, resolutionMetadata contains the MIMe type of the content of trustChain (in the fuiuture, additional type may be supported) and chainMetadata currently is an empty object, but in the future it may contain additional information.

If the issuer is not part of any chain of trust, the field `trustChain` contains an empty array, and no error is generated.

We also propose a represnetation for resultion errors:

```
    {
    @context:....,
    resolutionMetadata:{error:"",
erroMessage:""},
 trustChain: [],
chainMetadata:{}
}
```

where error is s tring identifying the error, like methodNotSupported if the DID method of the DID whose chain is to resolve is not the ssi-cot-eth, or internalError if there are some errors in contacting the blockchain. Like for did resolution and did url dereferencing, we add the error message explaining the cause of the error.

8.13 RevocationList2023

CredentialStatusList2021 is a good standard to represent

However, it requires the reification of the entire bit string when it is created. Moreover, it requires to encode such bit string. This may be a waste of space and a high gas cost if this information is stored on the blockchain.

However, if not stored on the blockchain, the verifiers may not be able to retrive the status of a VC, hence this will cause the VC to be discared.

Therefore, the information (revoked or not) must be stroed on the blockchain so to be publicly and highly available.

Moreover, if there are a lot of revocations, the array returned by the smart contract is very big, and the latency increases. This forces splitting the bit string into multiple pieces, and the computation of which piece contain the information we want.

Therefore, we propose a new credential status, that allows to retrieve the status information from any smart contract.

In particular, the verifiable credentials contain a DID URL in the credentialStatus field. This DID URL, once dereferenced, returns a JSON-LD object with the following form:

```
    {
    "@context": ["https://www.ssicot.com/RevocationList2023/"],
    "revoked": false
}
```

hence it contains just a field specifitying whetehre the speciifed VC is revoked or not.

The check of the revocation status can also be directly done in the smart contrct, by calling the resolveCredetialStatus method. The result is just a boolean value, an not a complex object like the one presented before. This allows to ease the resolution of credential statuses also on-chain.

To check for revocation, therefore, there is no need for the verifier to interact with a server under the control of the issuer. The interaction is directly with the blockchain.

To make the solution more gas efficient, theblockchain stores only the fragment part of the DID URl of the revoked verifiable credentials. Therefore, the space required is proportiinal to the number of revoked verifiable credentials.

Note that once a verifiable credential is revoked, there is no way to un-revoke it. However, there is no limit on the number of revoked credentials.

8.14 Smart contract

In this section, we report the costs, in gas units, of the execution of the methods exposed by the smart contract.

Note that, as explained in the previous section, the execution of any method on a DID document that is not reified requires its reification, which mean higher costs.

Therefore, we divide the computation of the cost in two categories, when the DID document is reified, an when it is not.

As said before, read-only methods do not cost any gas because they are directly executed on the client node.

removeAuthentication not reified (is is not possible because it is the last authentication)

Resolts give a rough idea, because the real gas cost depends also on the size of the inputs (i.e .,string lengths)

Method	gas cost if not reified (wei)	gas cost if reified (wei)
`addService`	508.401	180.469
`updateService`	/	47.778
`removeService`	/	58.952
`addAssertionMethod`	460.444	137313
`updateAssertionMethod`	368.204	42.285
`removeAssertionMethod`	297.320	54.184
`addAuthentication`	460.700	142.369
`updateAuthentication`	368.168	42.249
`removeAuthentication`	/	58.330
`updateTrustCertification`	635.832	143.062
`removeTrustCertification`	/	66.168
`revokeVerifiableCredential`	433.690	105.759
`deactivate`	365.166	40.039

Table 8.14.1:

8.15 Privacy

When creating a new DID, the user does not contact the blockchain and does not need to authenticate. Therefore, there is not transaction connecting the user with the DID or any other DID the user uses.

Anyone listening to the transaction will see some transactions. However, the issuer cannot track the users, because they do ot know which verifiers will ask for the resolution of the chain. Indeed, the issuer will see some accounts asking for the resolution of the chain.

Therefore, what the issuer sees is a bich of transaction made by several addresses to resolve the chain. If the verifiers are also issuer, they should use divverent Etehreum

accounts to issue VC and to verify VC. In this way, other issuers have no way to associate the DID the verifiers use to ask for the chin of trust to the verifier themeselves, hence the issuer has no way to track which issuers receive VC issued by the issuer.

8.16 Security

See 7.3.8 for a discussion on the possible attacks to the Etehereum network.

An attacker tries to modify DID documents that are not the one relayted to them -> the DID of the document to modify is directly extracted from the DID URL specified as authentication method when calling the method. Since the specified DID URL contains the DID, only the relative DID document can be modified by the attacker. However, since the attacker cannot insert its auth method to other DID documents, there is no way for the attacker to use one of its auth methods to autehtnicate and modify other DID documents.

An attacker calls updateTrustCertification using a ”fake” (crafted) certification, even with a valid signature (even from a government authority): -> The transaction is aborted, because the attacker cannot autehnticate as another user. -> The transaction is aborted because the data integrity proof of the VC cannot be validated.

An attacker is able to foll the issuer and make it issue a valid trust certification to them: Solution: the issuer can revoke the trust certification issued to the attacker.

An attacker steal a trust certification: -> They cannot use it to call updateTrustCertification because the credentialSubject.id field mentions a DID that is not under the control of the attacker, hence it cannot authenticate when calling updateTrustCertification as the subject of the VC.

An issuer loses its auth key: -> The DID is lost, because the issuer cannot authenticate anymore. Solution: use multiple

An issuer loses its assertion key: -> Solution: The issuer can revoke the stolen assertion key

An issuer is stolen its auth key: -> The DID is lost because the attacker can remove the auth key and add a new one.

If the issuer is part of a chain (but not the root), the parent in the chain could revoke the trust certification, the issuer can create a new DID, the parent issues a new ceritfication to the newly created DID, and the newly created DID re-issues the trust certification for all the children, whihich must take the new certification and call the updateTrsustCertification method to remove the previous certification and use the new one. If the issuer is the root, it has not certification to revoke, hence the first action is not to be executed.

Mitigation: periodic key rotation. Note that the children, nor the parent are compromized. This means that the attacker cannot autehnticate nor issue VC as the childer or the parent. The attacker can harm the children by revoking their trust certifications,

which impleis that they are not part of a chain of trust anymore. Note that the attacker must know the certification credentials to be revoked because they are not stored in the DID document of the issuer but in the DID documents of the children. Therefore, if not using a predictable scheme (like a counter), the attacker cannot revoke their credentials, unless it knows they are part of a chain of trust where there is also the compromized issuer. Moreover, the attacker cannot even know all the chains of trust the compromized issuer belongs to.

For instance, using a date and time as counter is not predictable, because the attacker cannot revoke any certification varying the date and time for any second, because it is very expensive.

An issuer is stolen its assertion key: -> Solution: The issuer can revoke the stolen assertion key (the attacker cannot do that typically is the issuer uses auth key != assert key) and insert a new one.

A user loses its auth key: -> The DID is lost, because the user cannot authenticate anymore. Solution, add more authentication keys, so that is one is lost the user can use another one.

A user is stolen its auth key: -> The DID is lost, because an attacker can authenticate as the user, and modify its DID document adding a new key under the control of the attacker and remove the one stolen. The user will be, therefore, not able to authenticate anymore. Mitigation: periodic rotation of the keys

8.17 Summary

Chapter 9

Case studies

In this chapter, we describe two case studies that use the library we implemented. They have been both realized by two people attending the University of Padua.

In the first one is to automatic subdividing the heritage of a person to their heirs after death. In particular, a person, before dying, will create its will in form of smart contract deployed on the Ethereum blockchain. All the ETH part of the heritage will then be sent to the contract.

When the person dies, anyone (not necessarily the hiers) can call a method exposed by the smart contract so to start the hertitage subdivision. To do so, the caller must provide a valid death certificate, under the form of Verifiable Credential. The smart contract will check the validity by calling the methods exposed by our smart contract.

In particular, it retrieves the chain of trust of the issuer of the death certificate, so to check if it is trusted by an entity that the death person has indicated when creating the will. After that, the smart contract will retrieve the Ethereum addresses of all the heirs indicated in the will by resolving the DID URLs that have been indicated by the death person. Finally, once retrieved the addresses, the smart contract will subdivide the heritage to all the heirs.

The second case study, instead, aims to expand the library so to use Zero-Knowledge Proofs. In particular, this solution is applied to an invented cinema that offers an online service allowing the users to book tickets for watching a movie. Since some movies cannot be seen by people under a specific age, the cinema, before booking a ticket for watching such a movies, requires the user to prove they are over a specific age.

To do so, the user creates a Verifiable Credential containing a Zero Knowledge Proof that proves the user is over a specific age without sharing their bitrh date, or any other information related to their identity.

The cinema, once received this Verifiable Credential, will validate it by retrieving the cryptographic material dereferencing DID URLs by using the library we have implemented.

9.1 Summary

In this section we exposed two case stduies that make use of the implementation we have realized , denoting their importance in the SSI ecosystem and how our implemebtation and idea esaed their implementation.

Chapter 10

Conclusion and future work

Self-Sovereign Identity is a big step further in the digital identity ecosystem. Indeed, compared to other solutions like centralized identity and federated identity, SSI guarantees much more privacy for the user. The ability to verify credentials without forcing verifiers to contact the issuer allows the verification at any time (the issuer must not be online). What we have proposed in an addition to SSI, allowing to reduce the burden of the verifiers, an allows any issuer to become part of a chain of trust.

The SSI implementation we have realized can be extended further. For instance, Verifiable Credentials need the definition of JSON-LD contexts, which are then used to properly caninicalize the VC when computing and verifying the Data Integrity Proof. However, these context definitions must be publicly available and always accessible, otherwise it would not be possible to create nor verify Verifiable Credentials.

Currently, these context definition files are published on web servers. While this enables the use of proxies so to cache these context definition files that, typically, change very often, there could be the chance that the servers are not available, an the verifier or the issuer ha no access to a cached copy of these files.

A possible solution to this problem would be to publish the JSON-LD context definition files on the blockchain. To do so, another smart contract should be deployed, and this smart contract will take care of managing the context definitions while using our smart contract to authenticate the user, so that a malicious user cannot modify a JSON-LD schema published by another user.

Another improvemente can be the addition of the generation and verification of Zero Knwoledge Proofs , so including directly in the library what a case study implements and extending it by adding the possibility to prove non-revocation, i.e., instead of using RevocationList2023 which requires to dereference a DID URL, you can directly include in the VC a proof that the VC has not been revoked.

Finally, RevocationList2023 can be extended by adding the possibility to suspend Verifiable Credentials. Indeed, with the implementation we porposed, VC can be only revoked and, once revoked, then cannot be used anymore. This forces the user to

request a new VC with the same claims if it wants to prove that claims again.

Indeed, imagine that, after an incident, the driving licence of a person is suspended while waiting for a judge to evaluate the respoinsibility of the driver. Using RevocationList2023, this implies that the VC representing the driving licence is revoked, otherwise the driver may use the VC to prove they have a valid driving licence.

However, it the judge judges that the driver is not responsible for the accident, the driving licence returns to be valid. This implies that the VC representing the diriving licence can become valid. Nevertheless, RevocationList2023 does not allow to un-revoke a revoked credential because revocation is permanent.

If we extend ReocationLists2023 to allow for suspension, then the VC driving licence will be suspended after the incident, and then unsuspended when the judge judges that the driver is not responsible. Therefore, the driver will not be forced to ask a new VC, but they can use the already issued one. This implies that, any verifier that has stored the verifiable credential, does not have to ask the new VC to the driver.

Bibliography

[1] URL: https://www.sciencedirect.com/science/article/pii/S1574013718301217.

[2] URL: https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=9927415.

[3] URL: https://www.mn.uio.no/ifi/english/people/aca/josang/publications/jfhdp2005-aisw.pdf.

[4] URL: https://sovrin.org/wp-content/uploads/2017/06/The-Inevitable-Rise-of-Self-Sovereign%20-Identity.pdf.

[5] URL: https://ieeexplore.ieee.org/abstract/document/5689468.

[6] URL: https://www.lifewithalacrity.com/2016/04/the-path-to-self-soverereign-identity.html.

[7] URL: https://dl.gi.de/server/api/core/bitstreams/d91fcb27-0129-499f-871d-c3b8ac336b63%20/content.

[8] URL: https://ieeexplore.ieee.org/abstract/document/9881606.

[9] URL: https://www.mdpi.com/1999-5903/13/12/301.

[10] URL: https://womencourage.acm.org/2021/wp-content/uploads/2021/07/87_extendedabstract.pdf.

[11] URL: https://sovrin.org/wp-content/uploads/Sovrin-Protocol-and-Token-White-Paper.pdf.

[12] URL: https://www.midy.com/wp-content/uploads/2023/05/An-Introduction-to-Midy-Whitepaper.pdf.

[13] URL: https://www.w3.org/TR/rdf11-concepts/.

[14] URL: https://www.ietf.org/rfc/rfc3987.txt.

[15] URL: https://www.w3.org/TR/rdf-canon/.

[16] URL: https://www.w3.org/TR/2020/REC-json-ld11-20200716.

[17] URL: https://github.com/decentralized-identity/EcdsaSecp256k1RecoverySignature2020.

[18] URL: https://www.w3.org/TR/did-spec-registries/.

[19] Manu Sporny, Dave Longley, Markus Sabadello, Drummond Reed, Orie Steele, and Christopher Allen. *Decentralized Identifiers (DIDs) v1.0. Core architecture, data model, and representations*. July 19, 2022. URL: https://www.w3.org/TR/did-core/ (visited on 06/13/2023).