KU LEUVEN

FACULTY OF
ENGINEERING SCIENCE

An SSI-based approach towards decentralized authorization with transitive delegations

Rawad Ghostin

Thesis submitted for the degree of
Master of Science in Engineering:
Computer Science, option Secure
Software

Supervisors:
Prof. dr. ir. Wouter Joosen
Dr. ir. Davy Preuveneers

Assessors:
Ir. Tim Van Hamme
Ir. Alexander van den Berghe

Assistant-supervisors:
Ir. Tim Van Hamme
Ir. Pieter-Jan Vrielynck

Academic year 2021 – 2022

Preface

This thesis would not have been possible without the support of many people.

I am extremely thankful to my supervisor, Wouter Joosen, for giving me the opportunity to research this topic and for his confidence in my work. I would also like to extend sincere gratitude to my co-supervisor, Davy Preuveneers, for his valuable insights and constructive suggestions.

Special thanks go to my mentors, Tim Van Hamme and Pieter-Jan Vrielynck, for their excellent guidance during the process, including weekly meetings, continuous advice, and reading of my numerous revisions.

Finally, I am particularly grateful for people close to me and whom I am fortunate to have by my side. My parents, Eid and Hoda, who inspired me and always stood behind me. My girlfriend, Paola, for being an unconditional and unwavering source of care and support. My close friends, for enabling some entertainment during demanding working days.

Rawad Ghostin

Contents

Abstract

In this thesis, we introduce an architecture for decentralized authorization with support for transitive delegation. The solution is designed to answer some use cases, such as building access control, where delegating access rights is functionally necessary, and centralized authorization presents essential security weaknesses. Today, the vast majority of access control solutions are designed to rely on centralized trust. However, centralized services are single points of failure and attacks. A malfunction in the central server can bring the whole system to a halt, compromising its availability and reliability.

Requirements analysis of the use case yielded three core requirements. First, the system must not rely on centralized trust. Second, users can delegate their permissions to other users. Third, permission delegations must be kept confidential to protect sensitive information. State-of-the-art solutions have been developed recently to tackle these requirements. However, these systems suffer from essential drawbacks that affect their security and performance. The current solutions assume that users' cryptographic keys are pre-shared securely, a problematic assumption in practice. Additionally, the existing solutions suffer from poor time performance in their most essential operations: authorizing and delegating permissions.

To address these issues, this thesis presents an alternative approach to the problem; it covers an architecture that satisfies the same requirement as the state of the art while offering consequential improvements. Firstly, the presented solution improves the performance drastically. For instance, the time to authorize a permission of depth 15 is up to 5 times faster than the state of the art. These improvements have been achieved by minimizing the amount of data to be encrypted and by using compression. Secondly, the architecture removes the assumption that users' keys are pre-shared securely; instead, it tackles the problem by relying on Self-Sovereign Identity (SSI). In SSI, the trust anchor is a blockchain, allowing users to prove their association with a public key without relying on centralized third parties, such as a certificate authority. Furthermore, this thesis offers an SSI-based authorization framework for decentralized access control. State-of-the-art SSI covers authentication only, but not authorization. However, authentication is not an end by itself; instead, it is a means to decide the level of access to grant users. As part of the solution, we present an architecture for SSI-based authorization, a standalone solution for decentralized access control, decoupled from the transitive delegation problem.

List of Figures and Tables

List of Figures

List of Tables

List of Abbreviations and Symbols

Abbreviations

CA	Certificate Authority
DID	Decentralized Identifier
DOS	Denial of service
IDP	Identity Provider
OOB	Out-Of-Band
PAP	Policy Administration Point
PDP	Policy Decision Point
PoC	Proof of concept
SSI	Self-Sovereign Identity
VC	Verifiable credential
VDR	Verifiable data registry
VP	Verifiable presentation

Chapter

1 Introduction

In 1993, *The New Yorker* magazine published a cartoon[1] showing two computer-savvy dogs, captioned "On the Internet, nobody knows you're a dog", denoting the anonymity of internet users at that time. Indeed, the great potential of the Internet and its accessibility led its ecosystem to grow at an unexpectedly rapid pace. Consequently, the Internet has been built gradually, sometimes inefficiently, leaving important parts without a standard and up to personal judgment. Such is the way for online authentication and authorization. Since 1993, technology usage has evolved, and while the Internet is still missing a standard identity layer, authentication and authorization components have become essential in many applications. In fact, many systems require to control the access to resources in order to provide a functionally correct and secure service.

Today, the vast majority of authorization solutions are designed to rely on centralized trust. For example, some prevalent solutions are the OAuth[2] framework, Active Directory[3], or most commonly custom solutions built by the application developers. However, in some cases, access control based on centralized trust poses essential security problems.

Firstly, a centralized service is a single point of attack. A central server storing security policies and controlling access to information is an attractive target to adversaries. An attacker compromising such a server is critical, as they are able to perform attacks with high impact. For example, they can tamper with access control policies to compromise sensitive resources. By revoking permissions for all employees to access a company building, the attacker hinders the company's operations and produces significant damage.

Secondly, a centralized service is a single point of failure. A malfunction in the central server can bring the whole system to a halt, making it less available and reliable.

Thirdly, central servers are managed by administrators whose integrity and reliability are to be trusted. Human factors, such as incompetence or corruption, threaten the system's security.

Finally, a central server storing security policies is a trove of sensitive information posing confidentiality and privacy concerns. For instance, a corrupted cloud administrator

[1] The cartoon was displayed in The New Yorker (vol. 69, no 20) issued July 5th, 1993.

[2] https://datatracker.ietf.org/doc/html/rfc6749

[3] https://docs.microsoft.com/en-us/windows-server/identity/ad-ds/active-directory-domain-services

or an attacker compromising such a server can deduce an organization's internal structure, reconstruct its organizational hierarchy and recognize high-value targets.

In response to these weaknesses, some current systems are increasingly avoiding centralized authorization, favoring models with decentralized trust instead. The WAVE[2] and DeFIRED[27] frameworks, which offer state-of-the-art decentralized access control with support for permission delegation, are examples of such systems. These frameworks are particularly beneficial in a scenario where physical access to buildings is to be delegated and controlled, as detailed in the following section 1.2. In this scenario, the drawbacks of centralized authorization become critical. However, the state-of-the-art frameworks suffer from some weaknesses, mainly in performance and scalability and make the problematic assumption that cryptographic keys are pre-shared securely.

In this thesis, we present an alternative architecture to decentralized access control with support for permission delegation that improves on the current state-of-the-art.

1.1 Problem Statement

Research has been conducted since 2019 into developing systems for decentralized authorization with support for transitive permission delegation. The WAVE[2] and [27] frameworks are the most important solutions. However, these frameworks suffer from essential drawbacks that affect their security and performance.

Firstly, state-of-art solutions assume that users' public keys have already been exchanged securely. However, this assumption referred to as the key exchange problem, is problematic in practice[12]. How can an identity prove their *ownership* of a public key? Today, systems typically authenticate keys by introducing a centralized third party, such as a certificate authority, responsible for assuring that an entity and a key are truly associated. However, this method relies on centralized trust, which is a single point of failure and attack[17].

Secondly, state-of-the-art solutions suffer from relatively poor performance in their most essential operations: permission authorization and delegation. The performance of WAVE and DeFIRED is negatively affected by their reliance on network communication subject to latency and their use of Identity-based encryption (IBE), which is relatively slow. Poor performance hinders those systems' usability and scalability, making them a less desirable option to be deployed in practice.

This thesis presents an alternative solution to the problem of decentralized authorization with support for transitive delegation that aims to improve the state of the art by addressing these issues.

1.2 Use case: Building access control

We consider a set of buildings owned by a *property owner* with building being managed by a *building manager* (*c.f* figure 1.1). Portions of the buildings are rented to tenants, for example, a company *"Corp"* that is installing its branch on the second floor of a building. Buildings are equipped with cyber-physical resources, such as digital door locks, cameras, and a wide variety of smart devices for which the ultimate authority is the

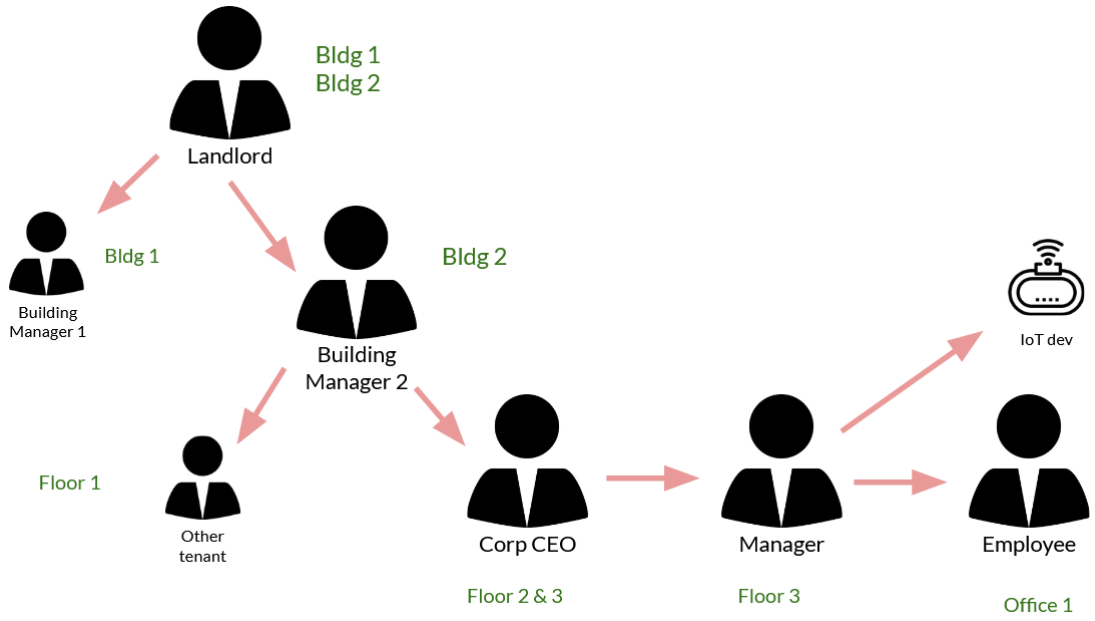


FIGURE 1.1: Example of building access control. The arrows represent a permission delegation.

property owner. However, to ease management, the property owner wishes to *delegate* the access for the building's resources to each corresponding building manager. In turn, building managers must further delegate a portion of their privileges to tenants, allowing them to control the area they rent. In the case of our example, the CEO of Corp should obtain rights over resources located on the floor they rent. Similarly, tenants can further delegate their rights to other users at their discretion. For instance, the CEO of Corp delegating access rights to employees.

This use case calls for three core requirements necessary for its functional correctness and security: *Transitive delegation*, *No reliance on centralized trust*, and *Confidentiality of topology*, each of which detailed in the following chapter 2.

1.3 Contributions of this thesis

This thesis presents an architecture for decentralized access control with support for transitive delegations. The presented architecture answers to the same requirements of state-of-the-art solutions while offering consequential improvements.

Firstly, the presented solution offers a drastic performance increase over the state of the art. For instance, the time to authorize a permission of depth 15 is up to 5 times faster than the state of the art. These improvements have been achieved by minimizing the amount of data to be encrypted and using compression.

Secondly, the drawbacks of the state-of-the-art mentioned in section 2.2 have been

mitigated. Mainly, the presented solution tackles the key exchange problem by using Self-Sovereign Identity (SSI) which relies on a blockchain as a trust anchor instead of a centralized Certificate Authority.

Thirdly, the presented architecture is built on top of an SSI layer, offering several building blocks that significantly simplify the system's complexity.

Furthermore, state-of-the-art SSI, a relatively recent technology, is only designed to support authentication but not authorization. As part of the solution, we present an architecture for SSI-based authorization. This architecture is a generic standalone solution for decentralized access control, decoupled from the transitive delegation problem.

1.4 Structure

The thesis is structured as follows.

We start by introducing background material relevant to building the solution. Chapter 3 introduces Self-Sovereign Identity (SSI), a decentralized identity technology employed by the architecture.

Then, we cover the architecture of the solution. Chapter 4 presents an architecture for decentralized permissions, that is further extended in chapter 5 to support confidential and transitive permission delegation. Chapter 6 presents an architecture for SSI-based authorization framework. The framework is a generic architecture for decentralized access control, attuned to function with the decentralized permissions outlined in previous chapters.

Next, chapter 7 details the development of a Proof-Of-Concept solution used to concretize the architecture and evaluate the performance. Chapter 8 covers the evaluation and compliance of the solution. In this chapter, the performance of the proof-of-concept is evaluated, and the solution is reviewed against all initial functional and non-functional requirements to ensure its full compliance. Chapter 9 further discusses the architecture's quality. An elementary security analysis of the solution is carried out with a reduced threat model and an availability analysis. In addition, the most important limitations of the system are outlined. Furthermore, the solution is compared to the state of the art.

Finally, chapter 10 concludes the thesis.

Chapter

2 Requirements and gap analysis

In this chapter, we cover specifications necessary in validating our solution. First, a requirements analysis is conducted to outline functional and non-functional requirements fundamental in addressing the problem of decentralized authorization with transitive delegations.. Then, we explore state-of-the-art solutions that address this problem, analysing their essential drawbacks with an aim of presenting an improved solution.

2.1 Requirements analysis

In this section, we present an overview of the requirements considered essential in addressing the problem of decentralized authorization with transitive delegations.

We define the following system entities:

- **Owner:** Entity that owns the protected resource.
- **Holder:** Entity that receives a permission.
- **Issuer:** Entity that issues a permission. The issuer can either be the owner or any other holder which has received a permission and is delegating it.
- **Authorization enforcer:** Software component that enforces access control to the protected resource.

Transitive delegation and revocation

Users of the system are allowed to delegate their permissions at their discretion. A tenant renting an area of the building can delegate their rights (integrally or partially) to other users. For example, the CEO of Corp can delegate the rights to an office to a manager that further delegates the rights to an employee. Therefore, a permission delegation must be *transitive*.

Moreover, permissions revocation must be transitive as well. Revoking a user's permission must automatically revoke all permissions generated by a delegation of that user's permission. For example, the building manager revoking Corp's CEO permissions to the floor they rent must automatically revoke the permissions for all Corp's employees.

We refer to this requirement as **CORE_DELEGATION**.

No reliance on centralized trust

The authorization system must not rely on centralized trust. As indicated in the introduction section 1, centralized access control is a single point of attack and failure. An attacker that compromises the central server gains access to all resources, in our case, the building's cyber-physical devices. They can spoof their identity to access unauthorized areas of the building or conduct a DOS attack to render the building unavailable.

We refer to this requirement as **CORE_DECENTRALIZED_TRUST**.

Confidentiality of topology

Permission delegations contain information involving the issuer and recipient users and the privileges they grant. Such information is considered sensitive private information. A tenant renting an apartment must be able to issue access to other persons, such as their family or a janitor, without the building manager having access to this information.

Furthermore, the information gained with the knowledge of permissions delegation raises security concerns. For example, an attacker knowing the permission delegations of Corp can deduce the organization's internal structure, reconstruct its organizational hierarchy and recognize high-value targets.

Therefore, permissions delegations must be confidential. Users in the system must not have access to permission delegation information except the ones involving them directly, as in the permission to which they are the immediate issuer or recipient.

We note that this requirement is relaxed solely for the property owner, the ultimate authority over the resources, to enable auditing.

We refer to this requirement as **CORE_CONFIDENTIALITY_TOPOLOGY**.

Other requirements

- **INTEGRITY**: The integrity of a permission must be verifiable by the verifier.
- **AUTHENTICATION:** An adversary that compromises a permission object cannot reuse it to escalate their privileges. That is, holders presenting a permission must be authenticated and their liveness must be asserted.
- **AUDIT:** All events related to authorizing access to a controlled resource must be auditable by the owner of this resource. As such, `CORE_CONFIDENTIALITY_TOPOLOGY` is relaxed for the owner of a resource; they must have knowledge of the entity accessing the resource as well as how the hierarchy of delegations was constructed.
- **CRYPTOGRAPHY_STRENGTH:** All encryption and digital signature keys must provide at least 112 bits of security [5] [4].
- **NONREPUDIATION:** A holder must explicitly either accept or refuse a permission that is being granted to them. Until accepted, the permission is ineffective. This mechanism serves as a protection mechanism to prevent issuing permissions to unaware users.

- **HOLDER_REVOCATION:** Holders can revoke permissions that were granted to them. Holder revocation allows permissions to be user-centric, which aligns with the principle of self-sovereignty mentioned in chapter 3.

- **PERFORMANCE_TIME:** The processing time of interactive operations must be lower than one second to retain the user's attention and flow of thought [20]. In our system, such operations are deciding the authorization outcome and issuing or delegating a permission.

- **PERFORMANCE_SIZE:** Permission objects size must not exceed `150 kB` to reduce cost and efficiency of storage and bandwidth, as well as reduce time of network transfer. We select `150 kB` as to ensure permissions objects can be transferred on the network in less than one second in the vast majority of countries[1].

- **USABILITY:** An issuer can consult a list of all permissions they directly granted. A holder can consult a list of all permissions they received.

2.2 State of the art

Traditional authorization models are based on accounts whose information is stored in centralized databases. However, the centralized model presents many weaknesses in distributed settings, mainly suffering from poor scalability, reliability, and robustness [1]. Moreover, centralized access control relying on a central authorization server presents a single point of failure and attack [17]. For example, in the WoSign incident, the Chinese certificate authority accidentally issued certificates for `*.github.com` to the wrong users [31] [17], allowing them to spoof the identity of `github.com`. Such incidents put in perspective how the failure of the trust anchor compromises the security of an entire system.

To tackles these issues, several approaches for decentralized access control have been proposed [17] [1] [7], mostly relying on distributed file systems, such as IPFS, and macaroons. A survey of decentralized authorization for distributed file systems is covered in [19]. Recently, research has been conducted to leverage the blockchain technology to achieve decentralized and distributed authorization [21] [10] [32]. The blockchain is an interesting subject of study for access control due to its unique security properties: it is a distributed database that is tamper-resistant and which no centralized entity controls. However, these solutions do not support transitive permission delegation and revocation yet.

To address the use case stated in section 1.2, research has been conducted since 2019 to develop decentralized access control solutions with transitive delegation mechanisms. The WAVE[2] and [27] frameworks address the three core requirements stated in section 2.1. In this section, we present these state-of-the-art solutions and analyze their drawbacks.

[1] https://www.speedtest.net/global-index

2.2.1 WAVE framework

WAVE[2] is a decentralized authorization framework with transitive delegation developed in 2019 at UC Berkeley. The framework is currently deployed on more than 800 production devices to control access to buildings and web services. WAVE answers the three core requirements presented in the use case section 2.1. However, the framework has some drawbacks that affect its security, performance, and usability.

Firstly, WAVE assumes that users' cryptographic keys have already been exchanged securely. This assumption, known as the key exchange problem, is problematic in practice[12]; how can two parties exchange and authenticate keys securely? In public key cryptography, key authentication is typically solved by introducing a centralized third party, the Certificate Authority, that authenticates a provided public key. However, this method falls back on centralized trust, which violates the core requirement `CORE_DECENTRALIZED_TRUST`.

Secondly, WAVE's time performance is relatively poor. As measured later in chapter 8, WAVE's operations are relatively slow due to it's usage of Identity-Based Encryption (IBE) which negatively affects its performance.

Finally, WAVE does not allow recipients of permissions to decline receiving a permission or revoke permissions they hold, which can raise repudiation concerns, as detailed explained by Vrielynck(2021) in his paper on DeFIRED [27].

2.2.2 DeFIRED

DeFIRED is another decentralized authorization framework that supports transitive permission delegation. The framework was developed in 2021 at KU Leuven and answers the three core requirements of the use case section 2.1. DeFIRED improves over WAVE by allowing its users to decline or disprove receiving a permission and revoke permissions they received. However, the framework suffers from similar drawbacks to WAVE: DeFIRED employs IBE that affects its performance negatively, as well as assumes that cryptographic keys are pre-shared securely, which as explained in the previous section, is problematic in practice.

Chapter

3 Background on Self-Sovereign Identity

In this chapter, we present Self-Sovereign Identity (SSI), an approach for decentralized identity that does not rely on centralized trust, thus, avoiding single point of failures and attacks. The chapter is structured as follows. First, we introduce existent models for digital identity and discuss their associated drawbacks. Then, we present SSI and explain the technology behind it. Finally, we discuss the extension of SSI to cover authorization.

3.1 The three models for digital identity

3.1.1 The centralized model

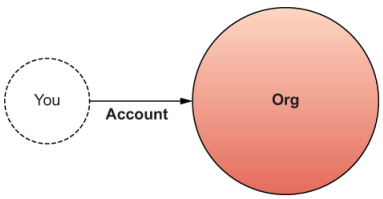


FIGURE 3.1: The centralized identity model. The user is as a dotted circle representing that their identity does not exist without their account at the organization. (Figure taken from [23])

The central identity model (figure 3.1) is the simplest form of digital identity, where users are assigned an identity by registering an *account* with a website, application,or service. The relationship between the user and the organization is created via shared secrets, typically a username and a password used to authenticate the user.

In the centralized identity model, a user's *identity* does not exist outside of the organization's system. That is, the organization lends the user credentials to authenticate themself to their service and manages all their data. However, their identity no longer exists after the user's account is deleted.

While being a simple way to digital identity, this model suffers from many drawbacks. First, the scenario of sharing secrets is repeated for every website, app, or service, putting

the burden on the user to remember and manage all their usernames and passwords. Secondly, organizations' centralized databases become a trove of sensitive information about their users. These databases are attractive targets to malicious adversaries, as has been shown by the frequent sizable data leaks. Moreover, the user has no choice other than to trust the organization with their data; and since each organization manages information security at its discretion, this poses consequential security and privacy threats to the user. Thirdly, users' identity data is not portable or reusable outside the organization.

The centralized identity model is the traditional way of digital identity and the most prevalent in the real world today.

3.1.2 The federated model

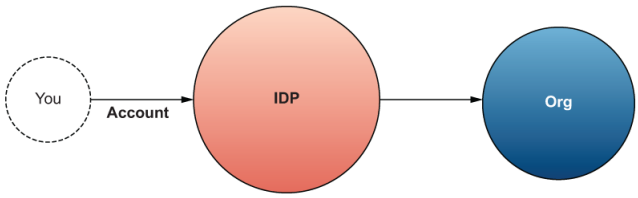


Figure 3.2: The federated identity model. (Figure taken from [23])

The federated identity model (figure 3.2) introduces a third-party organization that acts as an identity provider (IDP) between the user and other services. This addition enables Single-Sign-On (SSO), that is, to authenticate users to all services registered to the IDP with the same credentials provided to the users. Examples of SSO are "Login with Google"[1] and "Login with Microsoft"[2].

While federated identity alleviates some of the drawbacks of the central identity model, it still suffers from relevant problems. Firstly, the users and services need to trust the IDP completely, a man-in-the-middle that can monitor all their authentication activity. Secondly, big IDPs store vast amounts of sensitive data and are still attractive targets to malicious adversaries. Thirdly, there isn't a unique IDP that provides identity to all services. Instead, many IDPs exist, with the user still having to remember and manage their credential for each one.

Furthermore, as with centralized identity, federated identity users still do not have their own identity decoupled from any organization.

3.1.3 The decentralized model

The decentralized identity model avoids trusting any centralized third party becoming inherently decentralized. This model does not rely on organizations' accounts or third-party IDPs; instead, parties are independent entities that share a *trust connection*. Thus,

[1] https://developers.google.com/identity/sign-in/web/sign-in
[2] https://developer.microsoft.com/en-us/identity/add-sign-in-with-microsoft

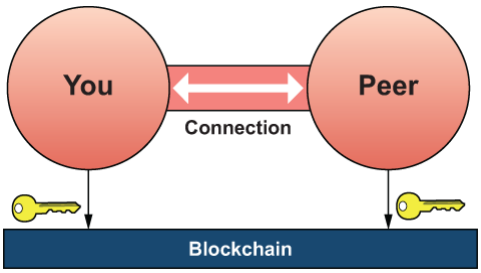


FIGURE 3.3: The decentralized identity model. (Figure taken from [23])

the relationship between the user and the service is direct and equal. Neither party controls the relationship; instead, it is a *peer-to-peer* connection that exists as long as both ends maintain it. In this way, a user's digital identity is global on the internet and decoupled from the services they use.

This model relies on a distributed database, such as a blockchain, to store and exchange users' public keys securely (*c.* figure 3.3). The public keys are employed to secure peer-to-peer communication channels between entities and verify entities' digital signatures. How decentralized identity works is explained later in section 3.3.

3.2 On the self-sovereignty

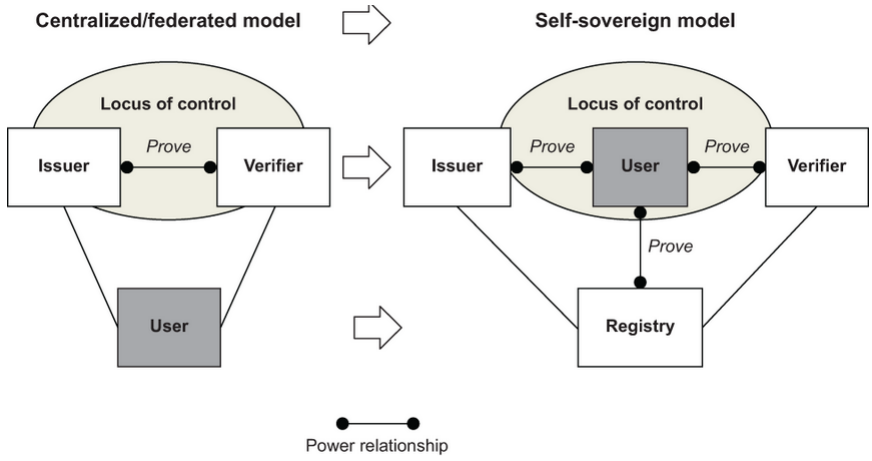


FIGURE 3.4: SSI transfers the control to the user. (Figure taken from [23])

The decentralized model for identity in the industry has been labeled ***Self-Sovereign Identity***, known as SSI. But what makes this model *self-sovereign*?

A *sovereign* entity, as per the Oxford English Dictionary[30], is an entity that is "free to govern itself; completely independent". Thus, the term *self-sovereign identity* refers to a person's identity that is independent of any other power or organization. With SSI, a

user's identity is solely managed by themselves and decoupled from the services they use; that is, a user is no longer identified by credentials attributed to them by services they sign up to. Instead, their identity is global, reusable on the internet, and only controlled by themselves.

In this way, SSI allows a *transfer of control* (illustrated in figure 3.4). That is, with the centralized and federated models, organizations and IDPs own users' identities. However, with SSI, the control is transferred to the users, allowing a truly user-centric interaction between users and services.

3.3 The SSI technology

In this section, we explain the building blocks of SSI to gain a better view of how it functions.

3.3.1 Verifiable credentials

Self-sovereign identity is fundamentally based on the concept of *verifiable credentials* (VCs).

A *credential* is "*any (tamper-resistant) set of information that some authority claims to be true about the subject of the credential—and which in turn enables the subject to convince others (who trust that authority) of these truths*", as stated by A. Preukschat and D. Reed in [23]. More concretely, a credential contains a set of claims about its subject (the *holder*) issued by an authority (the *issuer*). The integrity of a credential must be verifiable to allow holders to prove these claims to other entities (the *verifiers*). A credential verification could consist of checking that the presented credential has not been tampered with, its expiry date, when and by whom it has been issued. Some examples of real-life credentials are passports and identity cards whose integrity is verified visually and sensorially using physical markers.

SSI verifiable credentials[29] are digital credentials that can be digitally verified using cryptography. As illustrated in figure 3.5, the concept of VCs in SSI is directly derived from real-life credentials. A VC contains four main parts:

- **Credential Identifier:** Uniquely identifies the credential.
- **Credential metadata:** Meta information about the credential, such as its issuance and expiry date
- **Claims:** Claims about the subject.
- **Issuer signature:** Issuer's digital signature that allows verifiers to verify the VC's integrity.

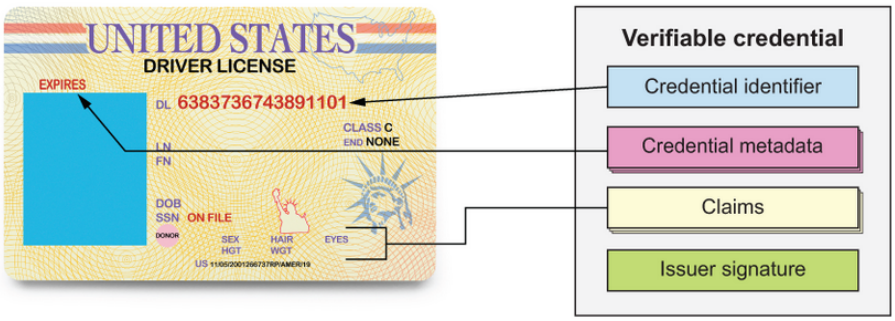


FIGURE 3.5: A real-life credential and a verifiable credential. The verifiable credential is a digital representation of a real-life credential (Figure taken from [23]).

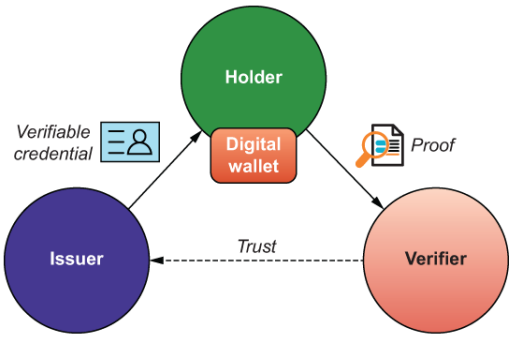


FIGURE 3.6: The trust triangle denoting the trust relationships between entities in an SSI system (Figure taken from [23]).

3.3.2 The trust triangle

SSI offers a decentralized identity model in which trust is established between entities in a peer-to-peer manner without the reliance on a centralized service. The relevant entities taking part in the SSI system are the following:

- **Issuers** are authorities that issue digitally signed verifiable credentials containing claims about their subjects. For example, issuers could be the government issuing a digital vaccination pass to a citizen.
- **Holders** receive VCs from issuers and store them in their wallet. Holders can present their VCs to verifiers to prove their claims. For example, holders could be citizens issued a digital vaccination pass from the government.
- **Verifiers** request and verify proofs of claims from holders. Verifiers could be any entity such as a person, an application, or an organization. For example, a verifier could be a company requiring its employees to be vaccinated.

The direct relation between all three entities is known as the *trust triangle*, illustrated in figure 3.6. This kind of peer-to-peer trust is analogous to real-life trust, for example,

in a scenario of a real-life ID card (VC) issued by the government (issuer) to a citizen (holder) and verified by a police officer (verifier).

3.3.3 Digital wallets

A digital wallet is a software installed on a holder's devices or in the cloud, responsible for storing and managing VCs the holder receives. Digital wallets offer the following features:

- **Key management system:** Handling of key generation, rotation, and revocation of cryptographic keys.
- **Encrypted storage:** Confidential storage for sensitive information such as VCs and cryptographic keys. Security is ensured through several defense mechanisms such as running inside a Trusted Execution Environments (TEE), for example, the Android Keystore System[3] on android devices, making the extraction of sensitive information an arduous task.

SSI wallets follow some essential design principles ensuring their security and alignment with the SSI philosophy. Firstly, wallets must be portable and implement open standards by default. In contrast to proprietary wallets whose capabilities are controlled by the vendor, an SSI wallet must aim to follow open standards allowing data portability. Thus, a holder is able to move their information from one wallet service to another at their discretion. Secondly, wallets must be consent-driven. A wallet must never execute actions that the holder has not explicitly authorized. Instead, the holder must give consent for every action, although holders can automate some consensual actions through policies. Thirdly, wallets must follow the privacy by design principles [8]. Mainly, wallets must implement end-to-end security, have privacy settings by default, and respect users' privacy. Finally, wallets must follow security by design. Wallets store sensitive information whose confidentiality is crucial; attackers compromising users' private keys enable high-impact damage. To mitigate threats, security by design principles [22] must be followed to help build a fundamentally secure application.

3.3.4 Decentralized Identifiers

On the internet, machines communicate through their assigned internet addresses, IP addresses, that identify them on the network. However, an IP address only identifies a machine, not the entity operating it. Today many services require users to authenticate themselves, that is, to corroborate their identity before they are authorized to access resources.

A Decentralized Identifier (DID) [28] is a globally unique URI [6] that identifies a digital identity. Each DID is associated with a private and public key (*c.f* figure 3.7). The identity holder publishes a digitally signed *DID document* containing their DID and its associated public key to the blockchain. The private key is kept confidential in the holder's wallet. To authenticate themselves, holders must prove their control over

[3] https://developer.android.com/training/articles/keystore

the DID that identifies their identity. For this purpose, they generate digital signatures to prove their possession of the private key associated with that DID. Since the DID document is publicly resolvable on the blockchain, verifiers can assert the proof's validity, thus the holder's identity.

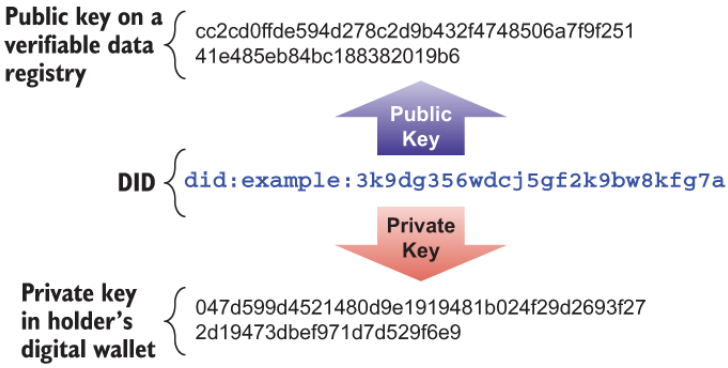


FIGURE 3.7: An example of a DID (Figure taken from [23]).

DIDs are designed to have the following properties:

- **Persistent:** Since DIDs are the identifiers of identities, they must be stable and never change. The DID must be independent of any particular representation or content of the digital identity.
- **Resolvable:** Entities in the system must be able to look up a particular DID and fetch relevant public information about the identity it identifies.
- **Verifiable:** Identity holders must be able to authenticate themselves, that is, corroborate their identity, by cryptographically proving their control over the DID. More concretely, identity holders prove that they control the private key complement to the public key associated with the DID.
- **Decentralized:** To avoid being a single point of attack or failure, DIDs must not rely on centralized trust, such as certificate authorities; instead, the identifiers must function on decentralized storage such as a blockchain.

3.3.5 Blockchain as a verifiable data registry

Trust connections are identified by the DIDs of the parties concerned. Before establishing a trust relationship, each participant must know the other party's DID to fetch their public key and initiate a secure peer-to-peer connection. So how do parties exchange their DIDs?

A DID can be stored in any verifiable decentralized storage, referred to as *verifiable data registry* (VDR), or exchanged directly between the two parties. However, a blockchain is the most appropriate due to its security guarantees. A blockchain is a tamper-resistant decentralized storage that no central party controls. Unlike other forms

of distributed storage, such as Distributed hash tables (DHT), blockchain sacrifices performance and efficiency for security; the ultimate objective of a blockchain is to offer tamper-resistant and robust storage that does not rely on any centralized party.

In the system, each user has a set of public cryptography keys: a secret key and its associated public key. Each piece of information to be published on the blockchain is digitally signed by its author. In the case of SSI, entities print a DID document containing their DID and public key. The DID document is signed with the entity's associated private key to prove their ownership of the document. In a blockchain, data is arranged in blocks that are securely tied together, forming a chain. Each block is linked to the previous block by containing its cryptographic hash. This way of chaining blocks makes it impossible to tamper with an individual block without having to recompute the hash of each of its consecutive blocks. However, computing a valid block hash is designed to be computationally very expensive, requiring significant time and energy to succeed. This mechanism makes data tampering a highly costly operation for an adversary to perform. Moreover, an additional security measure exists in the decentralization of the storage. After appending a block of data to the chain, the chain is replicated across all participants' nodes in the network, which will verify its validity before accepting it. Thus, a consensus of the participants is required for a block to be included in the blockchain. This measure makes crafting rogue chains a particularly arduous task as the adversary would need to compromise more than 51% of the participants to take over the consensus [25].

In this way, the blockchain becomes a trusted and authoritative storage of data that is not controlled by any central entity. In SSI, the blockchain stores DIDs and their associated public keys, acting as a *trust anchor* for key authentication. Unlike centralized trust anchors, such as certificate authorities, the blockchain is not a single point of attack or failure.

3.3.6 Credential revocation

In section 3.3.1, we introduced the concept of verifiable credentials, a set of claims made and signed by issuers that holders store in their wallets. In turn, holders can present proofs of claims to verifiers who can verify these proofs' validity before taking further application-specific actions. Since verifiable credentials serve as a means for authentication, they must be revocable to mitigate unauthorized users and limit the impact in case of private key leaks.

SSI revocation is based on cryptographic accumulators, zero-knowledge proofs that allow issuers to revoke credentials, and verifiers to verify whether a given credential has been revoked without having access to a complete revocation list. When presenting a credential, holders include a proof of non-revocation. After receiving the credential, the verifier tests the proof against a revocation registry published on the blockchain; if the credential is revoked, the holder's authentication fails. Otherwise, the normal flow of operations resumes.

In-depth coverage of cryptographic accumulators is considered out of scope for this thesis; however, a Hyperledger implementation of the mechanism is explained in detail in [15].

3.3.7 DIDComm

As explained in previous sections, SSI allows users to form direct trust relationships in a peer-to-peer fashion without having to trust any centralized third party, such as a certificate authority.

Trust relationships are materialized by creating a *DID-to-DID connection*, a peer-to-peer channel on which both parties can communicate securely. DID-to-DID communication is designed to offer the following properties:

- **Permanent:** The connection is never closed unless at least one party explicitly terminates it.
- **Confidential:** Messages transmitted over the channel are encrypted and signed, offering a robust defense against man-in-the-middle attacks.
- **Peer-to-Peer:** The connection links both parties directly without any intermediaries.
- **Mutually authenticated:** The identity of both parties must be corroborated.

Due to these security properties, DID-to-DID connections are used between entities to issue, request, or present verifiable credentials, trusted for secure communication between entities.

For this purpose, the DIDComm protocol [11] was created, a network communication protocol for DID-to-DID connections.

When two parties want to communicate, the first step is establishing a trust relationship, a DIDComm connection used to transmit messages securely. The connection is also used between entities to issue, request, or present verifiable credentials.

3.3.8 SSI: The big picture

In this section, we provide a big picture of the SSI technology that servers as a recapitulation to its operation (*c.f* figure 3.8).

SSI is a decentralized digital identity model aiming to avoid relying on any centralized trust. The model is built on top of DIDs, globally unique URIs that identify a digital identity, published on the blockchain along with their associated public keys. In SSI, holders store in their wallets verifiable credentials issued by issuers, entities who act as trusted authorities. The verifiable credentials contain claims about their holder, made and signed by the issuer. Holders of credentials can then present these claims to corroborate their identity, proofs which verifiers can verify using the corresponding issuer's public key. The model is built on top of DIDs, persistent globally unique URIs that identify a digital identity. Each digital identity has a DID along with a public cryptography key pair. While the private key is kept secret in the entity's wallet, the DID and the public key are published on the blockchain in a DID document to be publicly verifiable and resolvable by all entities. In this way, SSI offers a model for digital identity without relying on any centralized party, avoiding single points of failure and attack.

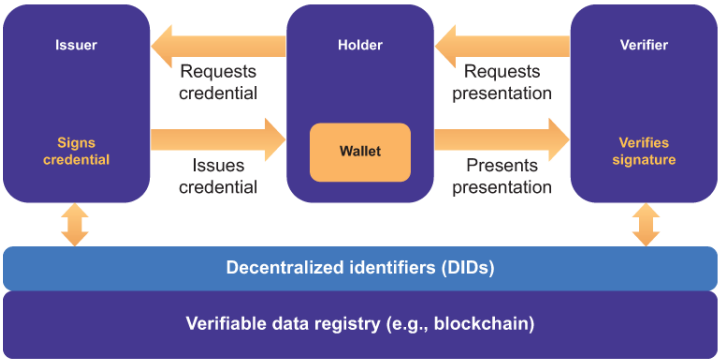


Figure 3.8: SSI, the big picture (Figure taken from [23]).

3.4 SSI-Based authorization

SSI is a model for decentralized digital identity in which there is no reliance on any centralized trust. The state-of-the-art SSI covers authentication only, but not authorization. However, authentication is not an end by itself; rather, it is a means for the system to decide the level of access a user is authorized. In this thesis, we look into SSI-based authorization as a way to achieve decentralized authorization. Moreover, as covered later in great details in chapters 4 and 6, we present an architecture for an SSI-based authorization framework.

Chapter

4 A protocol for decentralized permissions

As described in chapter 3, SSI offers a model for decentralized identity in which users hold decentralized credentials whose integrity is directly verifiable by referring to the blockchain. In our solution, we use SSI as a base layer to an architecture for decentralized permissions by storing permission objects in verifiable credentials. In this chapter, we first explore the process of presenting a verifiable credential, then present an architecture to model decentralized permissions on top of SSI.

4.1 SSI base layer for decentralized permissions

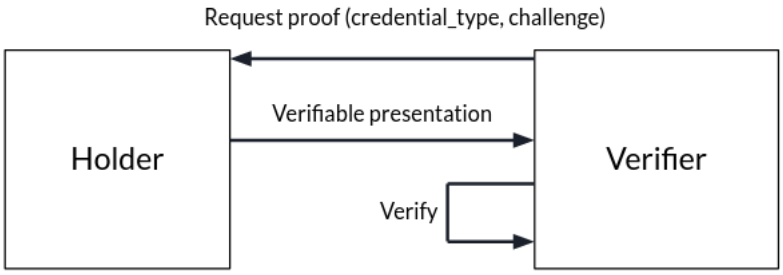
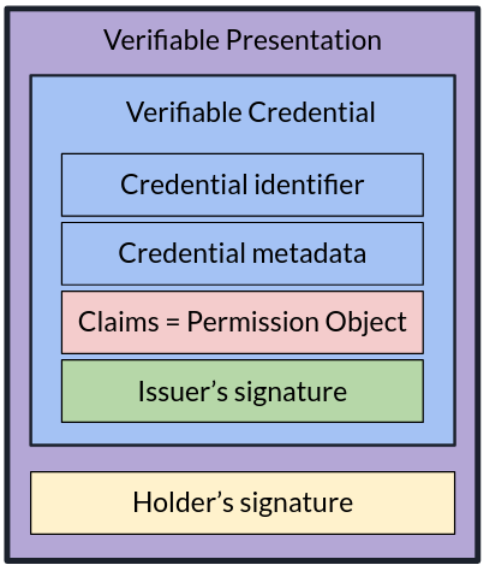


FIGURE 4.1: A credential request for verification purposes.

To present a requested credential for verification (figure 4.1), the holder constructs a verifiable presentation (VP) and transmits it to the verifier over an established DIDComm channel. A VP (figure 4.2) is a structure that contains a verifiable credential (VC) along with a proof of authenticity of the holder, which in most cases is their digital signature over the VC. The proof of authenticity prevents the issuer or any adversary that compromises the VC from spoofing the holder's identity. To protect against replay attacks, the verifier enforces a challenge-response protocol, in which a unique challenge must be included within the data to be signed. Therefore, the architecture of the




```
{
 "@context": [ "https://www.w3.org/2018/credentials/v1" ],
 "type": "VerifiablePresentation",
 "verifiableCredential": [{
    "@context": [ "https://www.w3.org/2018/credentials/v1" ],
    "id": "http://example.edu/credentials/1872",
    "type": ["VerifiableCredential", "PermissionCredential"],
    "issuer": "https://example.edu/issuers/565049",
    "issuanceDate": "2010-01-01T19:23:24Z",
    "credentialSubject": { <Permission object> }
  },
  "proof": { "type": "Ed25519Signature2020", [...] }
 }],

 "proof": {
    "type": "Ed25519Signature2020",
    "challenge": <challenge from verifier>,
    "jws":<digital signature of holder>,
     [...]
 }
}
```

Figure 4.2: Structure of a verifiable presentation [29][23]. In our case, a permission object is implemented as claims.

verifiable presentation readily answers to the security requirement AUTHENTICATION.

Upon receiving the VP, the verifier performs the following actions:

1. Authenticates the VP. This step consists of verifying both the holder's and VC issuer's signatures. To perform this operation, the verifier fetches the holder's and issuer's verification keys from the blockchain using their DIDs.

2. Reads the `claims` section and processes it for further application-specific actions.

Our solution uses SSI as a base layer to model decentralized permissions by implementing a permission object as claims in a verifiable credential. In this manner, a permission becomes a user-centric object stored client-side in the holder's wallet. Any verifier with access to the blockchain can publicly verify the integrity of the credential.

Moreover, with the permission being a VC, an issuer can revoke it using the SSI revocation mechanism described in chapter 3.

4.2 The permission object

The permission object is implemented as JSON claims included in the `credentialSubject` section of the verifiable credential. The permission object consists of the following fields:

- `claims`: The permission claims, which contains the following fields:
  - `iss`: *Issuer*. The DID of the permission issuer (in our case, the permission owner). This field is required for integrity verification and application-specific usage, such as defining access control policies.
  - `sub`: *Subject*. The DID of the holder of the permission. This field is required to ensure proper access control and prevent an adversary from reusing a stolen permission.
  - `iat`: *Issued at*. Issuance POSIX timestamp. The issuance time is required to issue permissions that only become effective in the future. Furthermore, the issuance time can be further processed for application-specific usages such as time-sensitive access control policies.
  - `exp`: *Expiry*. Expiry POSIX timestamp serves as a second mechanism for revocation and limits the impact of stolen permissions.
  - `resource_uri`: URI [6] of the protected resource. The field is a URI to allow greater adaptability and interoperability with various kinds of resources and protocols, for instance, an HTTP endpoint (`http://`), a local file (`file://`), or a git repository (`git://`).
  - `operations`: A list of allowed operations that can be performed on the resource. Operations are application-specific, and their definition is the application developer's responsibility. For example, for an HTTP endpoint, possible operations might be the HTTP methods [13].
- `auth`: The permissions security proofs, contains the following fields:
  - `Signature`: Issuer's digital signature over the `claims`. The signature is a protection mechanism against permission tampering and a way to authenticate the issuer. Thus, this answers the security requirement `INTEGRITY`. We note that any employed cryptographic key must offer at least 112 bits of security to be compliant with `CRYPTOGRAPHY_STRENGTH`. Throughout this project, we use ED25519 signatures, which offer 128 bits of security [16].

An example of a permission object can be found in appendix A (Listing 1).

4.3 Enforcing access control

4.3.1 Permission verification

Authorizing a permission holder involves verifying the integrity of the permission they present. For this purpose, the verifier must check the following conditions in the following order:

1. **Valid format.** All fields are present and contain valid values.

2. **Authentic.** The permission's security proofs (`auth`) must be verified. In the case of a digital signature, the verification requires fetching the issuer's verification key from the SSI blockchain using their DID (`iss`).

3. **Not revoked.** The permission has not been revoked. This check consists of verifying the revocation status of the SSI verifiable credential containing the permission object.

4. **Effective.** The permission is not expired (`exp`), nor issued in the future (`iat`).

The algorithm for permission verification is implemented in the pseudo-code function `is_valid` as follows:

```
    def is_valid(perm_credential):
    """
      Verifies the validity of a permission
      - perm_credential := The permission verifiable credential
                                          presented by the client.
      return := boolean, true if valid else false
    """
    perm = deserialize(perm_credential.credentialSubject) # permission
                                         object

    # Valid format
    if not is_valid_format(perm):
        return False

    # Authentic
    if not is_valid_signature(perm):
        return False

    # Not revoked
    if SSI_is_revoked(perm_credential):
        return False

    # Effective
    if (perm.claims.exp <= now) or (perm.claims.iat > now) or (perm.
                                        claims.exp < claims.iat):
        return False

    return True

boolean is_valid_signature(perm):
    """
      Verify the presented permission's digital signature.
      - perm := The permission object presented by the client.
      return: boolean, true if signature valid else false
    """
    issuer_verkey = SSI_Blockchain.get_verkey_of(perm.iss)
    return check_valid_signature(
             signature=perm.auth.Signature,
             verkey=issuer_verkey,
             to_check=perm.claims)
```

4.3.2 Request authorization

After verifying the permission, the issuer decides whether to authorize the holder to access the protected resource, possibly based on access control policies.

An authorization algorithm is presented in pseudo-code as follows. The following functions are employed:

- `is_authorized`: Returns an authorization decision given the client request, their presented permission credential and the access control policy.
- `policy_allows`: Determines whether a client request with a presented permission is allowed according to the given policy. It is the responsibility of the application developer to implement this function according to their application's requirements.

```
def is_authorized(request, perm_credential, policy):
    """
      Compute an authorization decision.
      - request := The client's request.
      - perm_credential := The permission verifiable credential
                                        presented by the client.
      - policy := A corresponding access control policy.
      return := boolean, true if authorized else false
    """
    return is_valid(perm_credential) and policy_allows(request,
                                      perm_credential, policy)


def policy_allows(request, perm_credential, policy):
    """
      Checks whether a client's request is allowed
      according to the access control policy,
      given a permission credential they presented.
      - request := The client's request.
      - perm_credential := The permission verifiable credential
                                        presented by the client.
      - policy := A corresponding access control policy.
      return := boolean, true if allowed else false
    """
    ### To be implemented by application developer
```

4.4 SSI wallets for builtin security features

Storing permission objects as verifiable credentials in a digital wallet allows us to take the opportunity of available security features implemented in secure wallet solutions.

Accepting a permission offer

Upon receiving a credential offer by an issuer, the wallet application notifies the holder of the event and requires them to either accept or decline the offer explicitly. Only upon

acceptance is the credential transmitted and stored in the wallet. In our case, a holder has the ability to decline a permission offer, thus answering to the security requirement NONREPUDIATION.

Delegatee revocation

The permission objects being solely stored client-side in the holder's wallet, a holder that deletes the permission effectively revokes it. Thus, this answers the security requirement HOLDER_REVOCATION. We note that, unlike the DeFIRED framework, our architecture does not support proving the non-possession of a permission credential.

Listing permissions

Wallets allow users to list their issued and received VCs, thus directly answering to the requirement USABILITY.

Additional security features

Secure and mature wallet solutions offer several important security features that are useful to our system, such as:

- Secure key management.
- Secure storage of credentials.
- Regulatory compliance.
- Secure backup, synchronization and replication.
- Ideally follows privacy by design [8] [23].

Chapter

5 Extending decentralized permissions with support for transitive delegation

In chapter 4 we established an architecture for decentralized permissions in which issuers can directly grant a privilege to holders. In this chapter, we extend the architecture to support *transitive permission delegation* as per the requirement CORE_DELEGATION. A permission delegation allows holders to delegate the permissions they received to other entities as a way of delegating them their privileges. In the following, we start by introducing transitive delegation concepts. Then, we describe an architecture for delegable decentralized permissions. Finally, we extend the architecture with a mechanism for enforcing the confidentiality of hierarchy and meet the requirement CORE_CONFIDENTIALITY_TOPOLOGY.

5.1 Transitive delegation concepts

In this section, we define essential concepts of transitivity and delegation required to design an architecture for delegable decentralized permissions.

Let the following sets used in the forthcoming sections:

- $Resources$: Set of all resources in the system. A resource is any information or operation whose access is to be controlled.
- $Entities$: Set of all entities in the system. Entities are users such as owners, issuers and holders.
- $Operations$: Set of all possible operations on resources in the system.
- $Permission$: Set of all possible permissions in the system.

Furthermore, we define the function $ownerOf : Resources \rightarrow Entities$ to return the owner of a resource in the system.

5.1.1 Delegation and transitivity

We define the function $privilegesOf : Entities \times Resources \rightarrow Operations$ to return a set of operations the entity is authorized to perform on the resource.

We define the function $delegated : Entities \times Entities \times Resources \rightarrow \{\bot, \top\}$ such that:

$$delegated(Iss, Sub, R) = \begin{cases} \top & \text{Iss has delegated a permission for resource R to Sub} \\ \bot & otherwise \end{cases}$$

We note that a *permission delegation* results in an issuance of privileges, such that the holder is partially or totally granted the issuer's operations on the resource:

$$\forall(Iss, Sub \in Entities; R \in Resources) :$$
$$delegated(Iss, Sub, R) \implies privilegesOf(Sub, R) \subseteq privilegesOf(Iss, R)$$

We describe a permission delegation to be *transitive* , for each permission z delegated from a permission y delegated from a permission x, the privileges that z grants are less or equal than the privileges that x grants. That is:

$$\forall(x, y, z \in Entities; R \in Resources) :$$
$$delegated(R, x, y) \wedge delegated(R, y, z) \implies privilegesOf(z, R) \subseteq privilegesOf(x, R)$$

5.1.2 Permissions Tree

We consider a protected resource $R \in Resources$ to which permissions have been issued/delegated to several entities in the system.

Let $G(N, E, R)$ be a digraph such that:

- N: Set of nodes; all entities having privileges to R (owner and holders of permissions).
- E: Set of edges; each edge represents a permission delegation, therefore

$$\forall(n_1, n_2 \in N) : (n_1, n_2) \in E \iff delegated(n_1, n_2, R)$$

- R: The resource being subject to access control.

We refer to $G(N, E, R)$ as the *permission tree* of the resource R. The permission tree is a representation of the infrastructure *topology*. We note that the root of the tree is the owner of the resource, from which all delegated permissions originate $(root(G) = ownerOf(R))$.

An example of a permission tree is illustrate in figure 5.1.

5.1.3 Hierarchy

Let $G(N, E, R)$ be the permission tree of a resource $R \in Resources$. Let the edge $p \in E$ be a delegated permission granting privileges to R. Let $H(p)$ be the path in G from p to the root of the tree, $root(G) = ownerOf(R)$. We refer to $H(p)$ as the *hierarchy* of p which represents the succession of permission delegations originating

from the owner O till the child permission p. An example of a permission hierarchy is represented in figure 5.1.

Moreover, we define the function $parentOf : Permissions \rightarrow \{\emptyset, Permissions\}$ to return the parent of a delegated permission, such as:

$$\forall(R \in Resources; P \in Permissions) :$$

$$parentOf(P) = \begin{cases} \text{First element of } H(P) & \text{issuer of P} \neq ownerOf(R) \\ \emptyset & \text{issuer of P} = ownerOf(R) \end{cases}$$

5.2 Delegable permission model

In this section, we extend the (non-delegable) architecture for decentralized permissions described in chapter 4 to support transitive delegation.

The non-delegable permission object (section 4.2) can be modelled as the following 4-tuple P ignoring security proofs and timestamp fields (`auth`, `iat`, and `exp`):

$$P := (Iss, Sub, R, Ops)$$

with:

- $R \in Resources$: `resource_uri`, the resource subject to access control.
- $Iss \in Entities$: `iss`, the permission issuer.
- $Sub \in Entities$: `sub`, the permission recipient.
- $Ops \in Operations$: `operations`, set of granted operations on the resource.

We extend the permission model to support transitive delegations by defining the delegable permission P as the following 6-tuple:

$$P := (Own, Iss, Sub, R, Ops, H := parentOf(P))$$

with:

- Own: The resource owner $ownerOf(R)$, to be preserved across delegations.
- H: The parent of P, $parentOf(P)$, the permission which direct delegation yielded P. By storing the parent permission in the child, each permission effectively stores its complete hierarchy recursively.

This concept is illustrated in figure 5.1 with an example permission tree and delegable permissions.

Having established a model for delegable decentralized permissions, we define the delegable permission object as an extension of the permission object described in section 4.2, with two new fields:

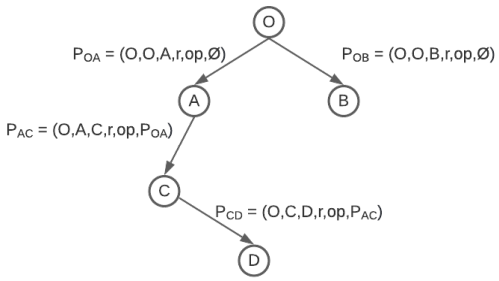


Figure 5.1: Example permission tree of resource r with delegable permissions. - The hierarchy of P_{CD} is $H(P_{CD}) = [P_{AC}, P_{OA}]$. As such, P_{CD} can read its hierarchy $H(P_{CD})$ recursively via its parent P_{AC}.

- `owner`: DID of the resource owner. We note that, for anonymity and increased decentralization, it is recommended that the owner give the resource it's proper digital identity; that is, the resource to be assigned a DID and public key pair. In that case, the `owner` field contains the DID of that resource. This mechanism also eases potential transfers of ownership.
- `hierarchy`: The hierarchy of the permission object. In practice, the encoded parent permission object is stored. The choice of the encoding algorithm is the responsibility of application developers, however, we recommend `Base64` for its adequate compromise in speed of execution and output size.

An example of a delegable permission object can be found in appendix A listing 2.

5.3 Delegable permission verification

Verifying a delegable permission consists of checking the validity of its hierarchy and each permission that constitutes it. For this purpose, all the following conditions must be met.

- **Hierarchy continuity.** The hierarchy must be continuous as per the definition of transitivity section 5.1. A hierarchy is continuous if the issuer of each child permission is the subject of its parent permission.
- **Originates from the owner.** The sequence of delegations must ultimately originate from the owner of the protected resource.
- **Does not exceed maximum depth.** We introduce a maximum allowed depth of delegations, MAX_DEPTH, to avoid hefty permissions and mitigate DOS attacks.

- **Non-increasing privileges.** Each child permission grants equal or fewer operations than its parent permission.
- **All permission are authorized.** Each permission of the hierarchy is individually verified similarly to non-delegable permissions as described in section 4.3.2 with the function `is_authorized`. This condition is necessary to mitigate attacks in which an adversary crafts their own permission. Moreover, the condition enable *transitive revocation* as required by `CORE_DELEGATION`. We note that, as indicated in section 4.3.2, using the `is_authorized` function requires the application developer to implement the `policy_allows` function according to their application's requirements.

The algorithm for verification of delegable permissions is presented in pseudo-code as follows:

```
def is_authorized_delegable(request, perm_credential, policy):
    """
      Compute an authorization decision for a delegable permissions.
      - request := The client's request.
      - perm_credential := The delegable permission credential
                                        presented by the client.
      - policy := The relevant access control policy
      return := boolean, true if authorized else false
    """
    depth = 0
    curr_cred = perm_credential

    while True
        # Condition: Does not exceed maximum depth
        depth += 1;
        if depth > MAX_DEPTH:
            return False

        # Condition: All permissions are authorized - c.f section "
                                          Permission verification"
                                          for is_authorized
        if not is_authorized(request, curr_cred, policy):
            return False

        # Condition: Originates from the owner
        # Current permission is initial permission issued by owner,
                                          All permissions up to this
                                          point verified, authorize
        if curr_cred.permission_object.iss == policy.owner.did:
            return True

        # Condition: Originates from the owner
        # End of hierarchy, the root of the hierarchy is not the owner
        if curr_cred.permission_object.hierarchy is None:
            return False

        try:
            parent_cred = get_parent_permission(curr_cred);
```

```
        except:
            return False

        # Condition: Hierarchy continuity
        if curr_cred.permission_object.iss != parent_cred.
                                              permission_object.sub:
            return False

        # Condition: Non-increasing privileges
        if not curr_cred.permission_object.operations is_subset_of
                                              parent_cred.
                                              permission_object.
                                              operations:
            return False

        curr_cred = parent_cred


def get_parent_permission(perm_credential):
    """
        Given a permission credential, return the previous permission
                                              credential in the hierarchy
                                              .
        - perm_credential := A delegable permission credential
        return := The previous permission object
    """
    encoded_parent_cred = perm_credential.permission_object.hierarchy
    return deserialize(decode(encoded_parent_cred))
```

5.4 Confidentiality of topology

The architecture for delegable permissions presented in section 5.2 extends our system to support transitive permission delegation. However, since each permission stores its own hierarchy, holders of permissions gain knowledge of a section of the permission tree: the sequence of delegations originating from the resource owner till their permission, including the issuers involved. Knowledge of the topology raises privacy concerns, as per the LINDDUN framework [9], it presents linkability, identifiability, and detectability threats. An adversary possessing a permission object can compile a partial list of entities in the organization, infer their role in the organizational hierarchy, and deduce persons of interest to be targeted in further attacks. In this section, we design a mechanism to provide confidentiality of topology and meet the core requirement CORE_CONFIDENTIALITY_TOPOLOGY.

5.4.1 Requirements analysis

Achieving confidentiality of topology must meet the following requirements:

- **OWNER_FULL_VIEW:** Only the owner of the protected resource and the software guard responsible for protecting it can have full knowledge of the topology.

- **ISSUERS_LIMITED_VIEW:** To mitigate privacy threats, issuers (except the owner) must not be able to infer any topology information besides the owner's identity and the identity of holders to which they directly delegated permissions.
- **HOLDERS_LIMITED_VIEW:** To mitigate privacy threats, holders (except the owner) must not be able to infer any topology information besides the owner's identity and the identity of the issuers that directly delegated them permissions.
- To stay compliant with requirement CORE_DELEGATION, holders must still be able to delegate permissions, and issuers must still be able to revoke permissions they delegated.
- To be compliant with requirement CRYPTOGRAPHY_STRENGTH, all encryption and digital signature keys must provide at least 112 bits of security.

5.4.2 Encryption scheme

General flow

As detailed in section 5.2, delegable permission objects store their own hierarchy required for verification. To achieve confidentiality of topology, the hierarchy must be confidential. We distinguish two main cases in this process, permission delegation and permission verification. Delegated permissions are issued with an encrypted hierarchy that only the resource owner can decrypt. Then, resource owners verifying delegable permissions consists of executing the delegable permission algorithm (section 5.3) while decrypting the hierarchy before being able to process it.

Encryption process

The encryption process (figure 5.2) occurs during a permission delegation and consists of the issuer encrypting the hierarchy of the child permission to be transmitted to the holder. Since the hierarchy is stored recursively by having each child permission contain its parent, effectively, the issuer encrypts their own permission credential, the parent, to be kept in the `hierarchy` field of the child permission. During the encryption process, the issuer performs the following actions:

1. Read the resource owner's DID from their own permission object, then fetch its corresponding public key from the SSI blockchain.
2. Fetch their private key from their SSI wallet.
3. Compress their own permission object.
4. Using the previously fetched private key and the owner's public key, use mutually authenticated public-key encryption to encrypt the compressed permission object.
5. Encode the encrypted permission object to be conveniently stored in the `hierarchy` field of the child permission to be issued. The choice of the encoding algorithm is the responsibility of application developers, however, we recommend `Base64` for its adequate compromise in speed of execution and output size.

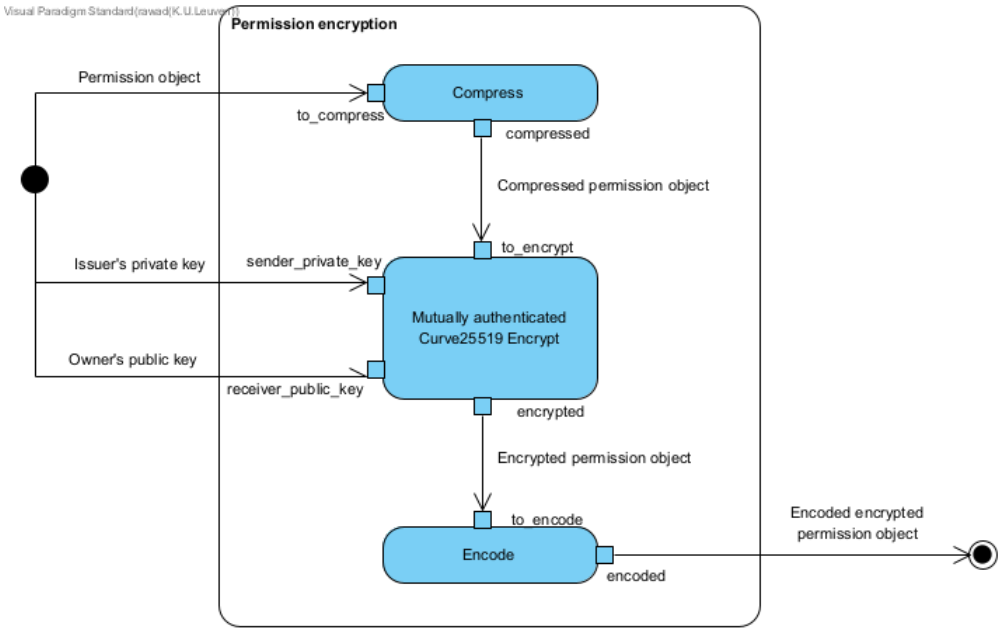


Figure 5.2: Activity diagram denoting a permission encryption.

Decryption process

The decryption process (figure 5.3) occurs when a resource owner is verifying a presented delegable permission. The process consists of decrypting the `hierarchy` field prior to processing it during the delegable permission verification 5.3. To this effect, the verifier (owner) performs the following actions:

1. Read the issuer's DID from the presented permission object, then fetch its corresponding public key from the SSI blockchain.
2. Fetch their private key from their SSI Wallet.
3. Decode the `hierarchy` field of the presented permission.
4. Using the previously fetched private key and the issuer's public key, use mutually authenticated public-key encryption to decrypt the compressed permission object.
5. Decompress the permission object.

The decryption process results with the (plain text) parent permission of the presented permission, at which point the hierarchy verification algorithm can resume as detailed in section 5.3.

The presented design readily meets the requirements `CORE_DELEGATION` listed in section 2.1, and requirements `OWNER_FULL_VIEW`, `ISSUERS_LIMITED_VIEW`, and `HOLDERS_LIMITED_VIEW` listed in section 5.4.1. Moreover, We use ED25119 for encryption which provides 128 bits of security [16], thus staying compliant with requirement `CRYPTOGRAPHY_STRENGTH` as well.

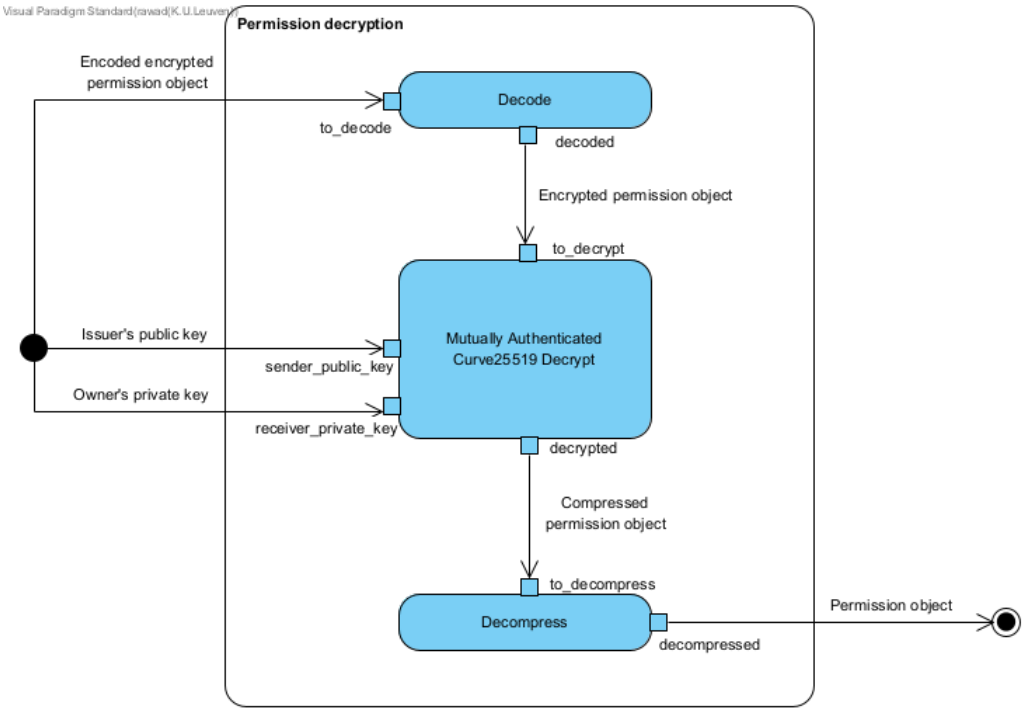


FIGURE 5.3: Activity diagram denoting a permission decryption.

5.4.3 Compression to improve performance

In the encryption and decryption processes, compression improves performance. Firstly, compression reduces the size of delegated permissions, thus decreasing the latency when they are transmitted over the network. Secondly, in our Proof-Of-Concept to be presented in chapter 7, the compression operation proved to be substantially faster than the cryptographic operations. Therefore, compressing the hierarchy before encrypting it improves the performance of the delegation and verification operations as will be reported in chapter 8. The performance improvement is due to reducing the amount of data to be encrypted and decrypted.

5.4.4 Caching to improve performance

The encryption and decryption process require querying the blockchain to fetch a public key corresponding to a given DID. This operation is subject to network latency, is relatively slow [24], and is unavailable during potential blockchain downtime. To improve the performance and the availability, we can add a caching mechanism to store frequently accessed keys. The caching strategy must take into account the probability of key changes.

Chapter

6 An architecture for SSI-based access control

As covered in chapter 3, SSI is a model for decentralized digital identity in which there is no reliance on any centralized trust. The state-of-the-art SSI covers authentication only, but not authorization. However, authentication is not an end by itself; instead, it is a means for the system to decide the level of access a user is authorized. In this chapter, we present an architecture for an SSI-based authorization framework for decentralized access control.

6.1 Authorization system

In chapter 4, we describe an architecture for decentralized permissions, in which a holder receives and stores user-centric permission objects. Permissions are implemented as verifiable credentials, for which the integrity and authenticity can be verified by any entity (the verifier) with reference to the blockchain. This section presents an architecture for an SSI-based authorization system responsible for controlling access to resources. The boundaries of our system lie in the `AuthorizationEnforcer` component described in detail in the forthcoming sections.

6.1.1 Overview of the authorization process

The `AuthorizationEnforcer` component (Figure 6.2) is a gateway responsible of enforcing access control to a resource. The component is deployed between one or several `Resource` components and the holder's application `ClientApp`. The `AuthorizationEnforcer` is accessible to the client as an HTTP API via an `HTTPServer` secured by HTTPS (Figure 6.3). Moreover, both the `ClientApp` and the `AuthorizationEnforcer` access an underlying SSI layer via an `AriesAgent`, an SSI client responsible for providing an SSI wallet service, managing DIDComm channels, and communicating with the SSI blockchain.

To operate on protected resources, holders must present a valid permission credential to the `AuthorizationEnforcer` which verifies its integrity, then decides whether to grant or deny the operation based on configured access control policies.

The authorization process is represented in a `AuthorizationSession` state machine figure 6.1, and happens as follows:

1. To operate on a protected resource, the client (holder) sends a corresponding HTTP request to the `HTTPServer` which forwards all requests to the `AuthorizationEnforcer`. At this step, the state of the session is `NEW`.

2. Via the SSI layer (depicted in purple), the `AuthorizationEnforcer` and the client establish a secure Out-Of-Band (OOB) DIDComm channel. During this step, the state of the session transitions from `CONN-WAIT-ACCEPT` to `ESTABLISHED`.

3. Over the OOB channel, the `AuthorizationEnforcer` requests a permission credential required to access the requested resource. The client presents the permission issued to him by a trusted issuer. During this step, the state of the session transitions from `CREDS-WAIT` to `CREDS-RECEIVED`.

4. Via the SSI layer, the `AuthorizationEnforcer` verifies that the presented credential is authentic, in which case it extracts the permission object. At this step, the state of the session is `CREDS-VERIFIED`.

5. Based on configured access control policies, and with the presented permission object, the `AuthorizationEnforcer` decides whether to authorize or deny the client's request. The session transitions to state `DEC-AUTHORIZED` or `DEC-DENIED` respectively.

6. If denied, the request is blocked. If authorized, the request is forwarded to the `Resource` component. Finally, the `Resource`'s response is forwarded back to the client.

A failure or cancellation at any step of the process results in the `AuthorizationSession` being transitioned to the state `DEC-DENIED/CANCELED`.

In the forthcoming sections, we follow a top-down approach to the `AuthorizationEnforcer` component in greater details.

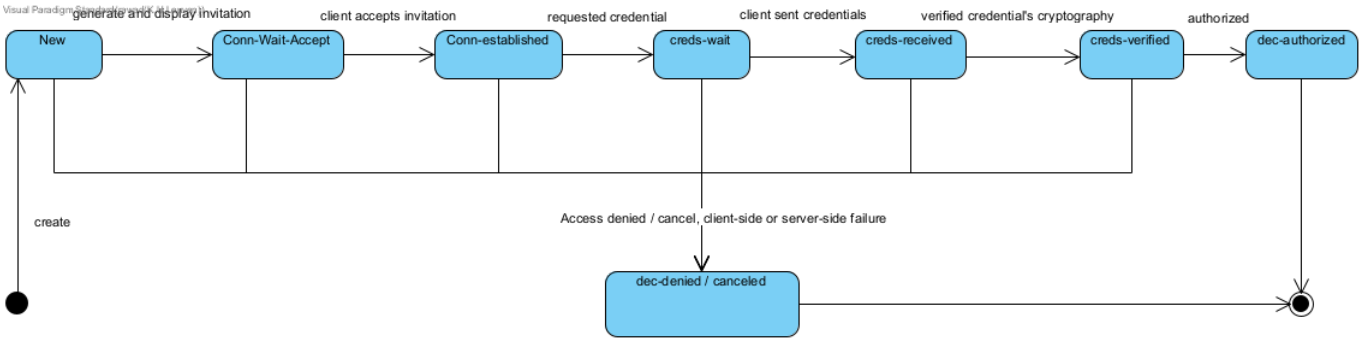


FIGURE 6.1: State machine representing an authorization session.

6.1.2 AuthorizationEnforcer pipeline

The `AuthorizationEnforcer` component depicted in figure 6.5 is responsible of enforcing access control to protected resources. A request entering the `AuthorizationEnforcer` follows an authorization process in a pipeline composed of three stages illustrated in figure 6.4.

1. **State Establishment stage.** In this stage, all information required from the client to compute an authorization decision is acquired. In our case, is required a permission credential proving the client's privileges to access the protected resource. The components responsible for conducting the State Establishment stage are depicted in gray throughout the report: the `Controller` and `SessionStore` components.

2. **Authorization Decision stage.** In this stage, an authorization decision is computed based on the presented permission and configured access control policies. The components responsible for conducting the Authorization Decision stage are depicted in dark green throughout the report: the `AuthorizationEngine` component.

3. **Operation stage.** If the authorization decision is positive, at this stage, the client's request is forwarded to the protected resource. The components responsible for conducting the Operation stage are depicted in red: the `ResourceOperator` component.

During the authorization process, the underlying SSI layer is frequently accessed to perform operations such as requesting and authenticating a verifiable credential. The access to the SSI later is mediated by a helper service, the `AriesService` component, responsible for encapsulating SSI operations to the `AriesAgent` as well as providing a caching mechanism.

The primary sequence diagram of the system illustrated figure 6.6 depicts the interaction of different components through all stages of the authorization process.

In the forthcoming sections, we explain each of the three stages in greater depth.

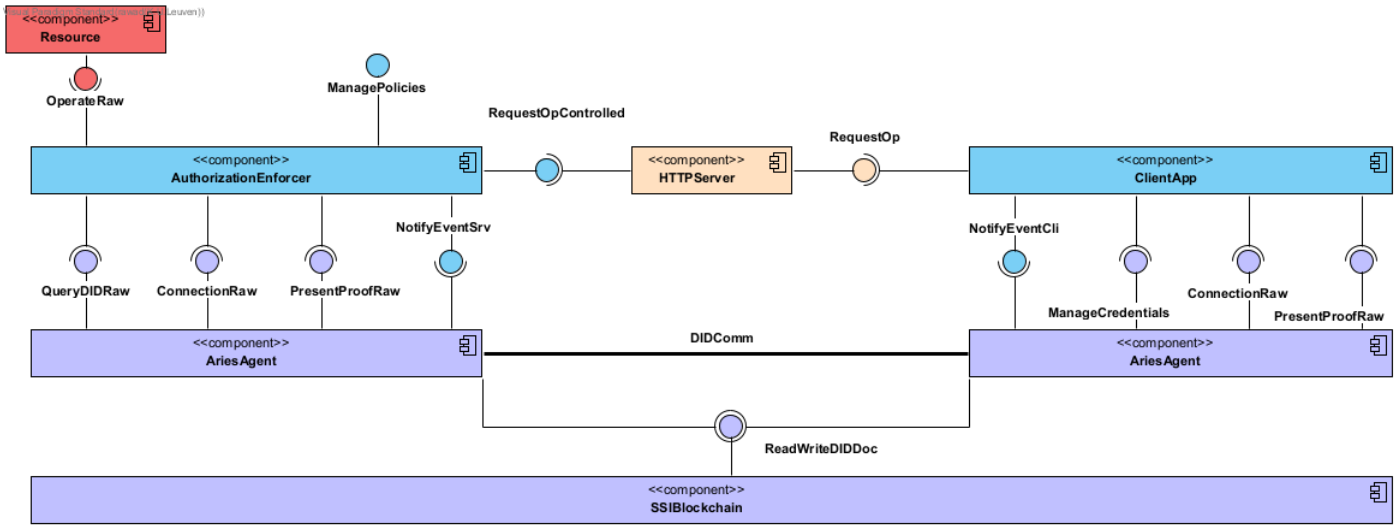


Figure 6.2: Primary component diagram of the authorization system. The architecture is scoped to the AuthorizationEnforcer component; all other components lay outside the architectural boundaries of the system.

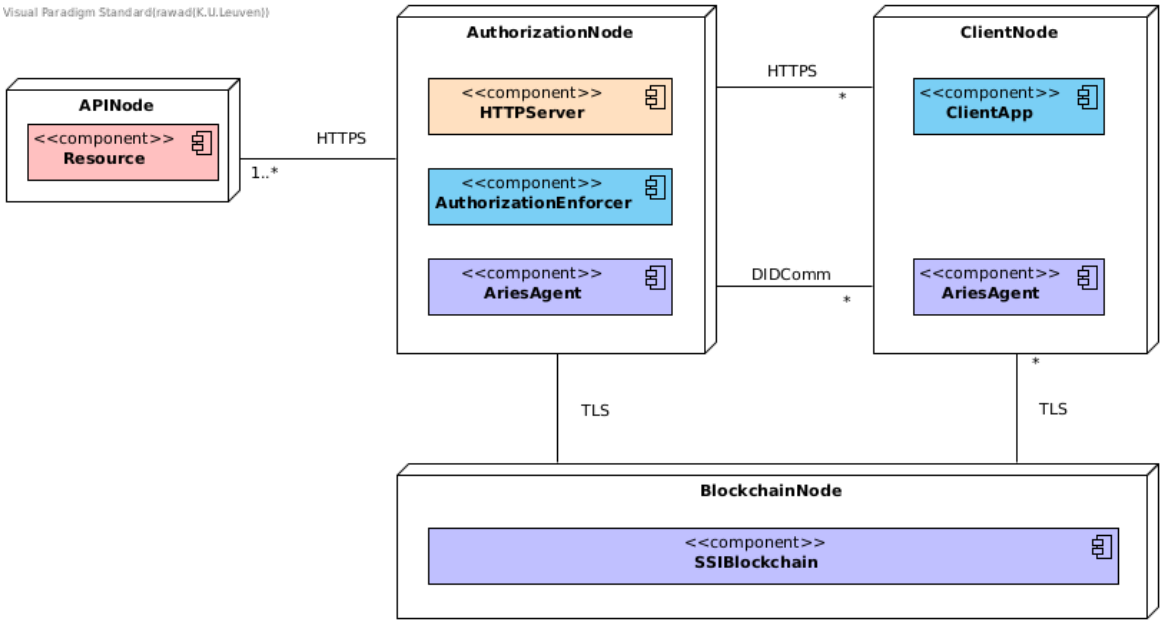


Figure 6.3: Deployment diagram of the authorization system.The architecture is scoped to the AuthorizationEnforcer component; all other components lay outside the architectural boundaries of the system.

FIGURE 6.4: Pipeline of the authorization process.

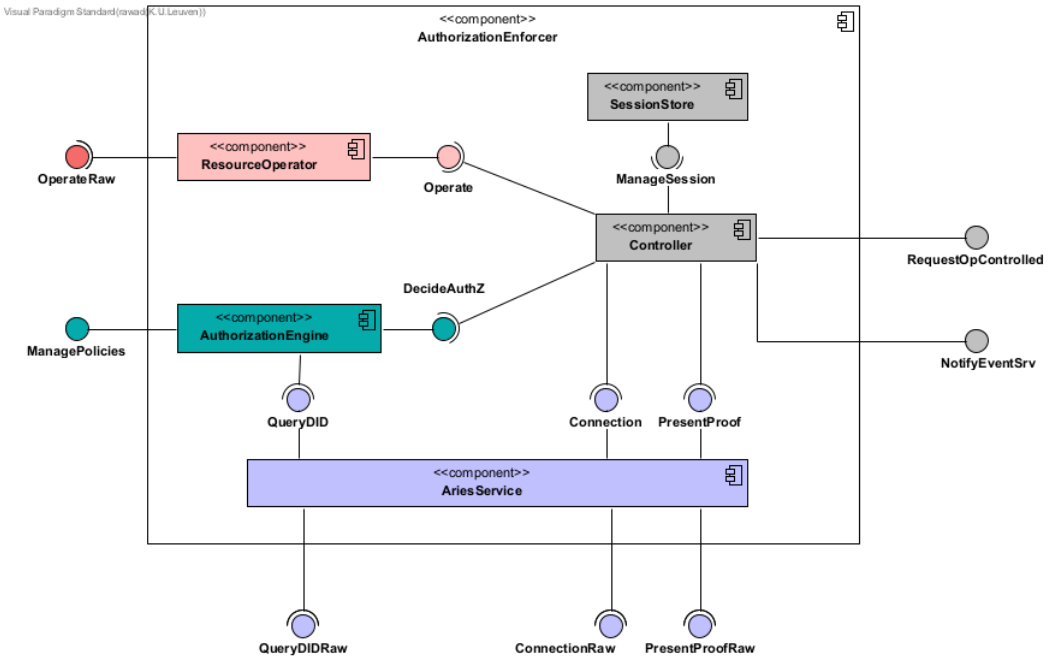


FIGURE 6.5: Decomposition of the AuthorizationEnforcer component

6.2 State Establishment stage

During the authorization process, clients requesting a protected resource must present all the information necessary to prove their privileges on the resource. Thus, before making an authorization decision, a permission credential must be acquired from the client. We refer to this process as *state establishment*.

The state establishment process is conducted by the `Controller` which is responsible for handling all input messages to the `AuthorizationEnforcer`. Input messages can be of two types: HTTP requests sent from the client, or asynchronous notifications spawning from the underlying SSI layer. To handle messages, the `Controller` is composed of two sub-components (Figure 6.7):

- `HttpRequestHandler`: Component responsible of handling the HTTP requests forwarded by the `HTTPServer`. The messages are received via the interface `RequestOpControlled`.
- `SSIEventHandler`: Component responsible for handling asynchronous events from

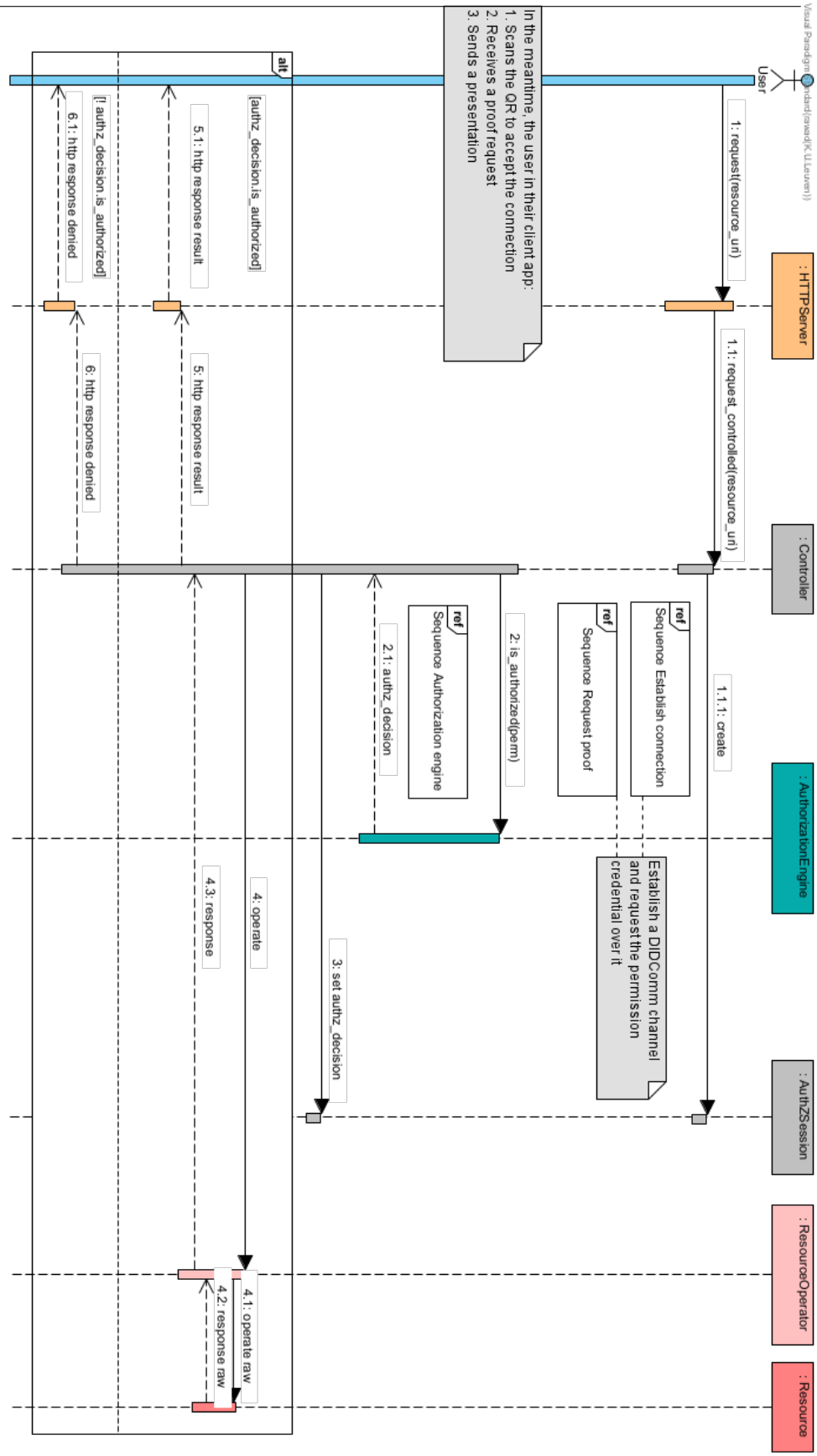


Figure 6.6: Sequence diagram depicting the operation of the AuthorizationEnforcer component

the underlying SSI layer via its `NotifyEventSrv` interface.

Both the `HttpRequestHandler` and `SSIEventHandler` components manage session objects stored in the `SessionStore` to save the progress of the authorization process.

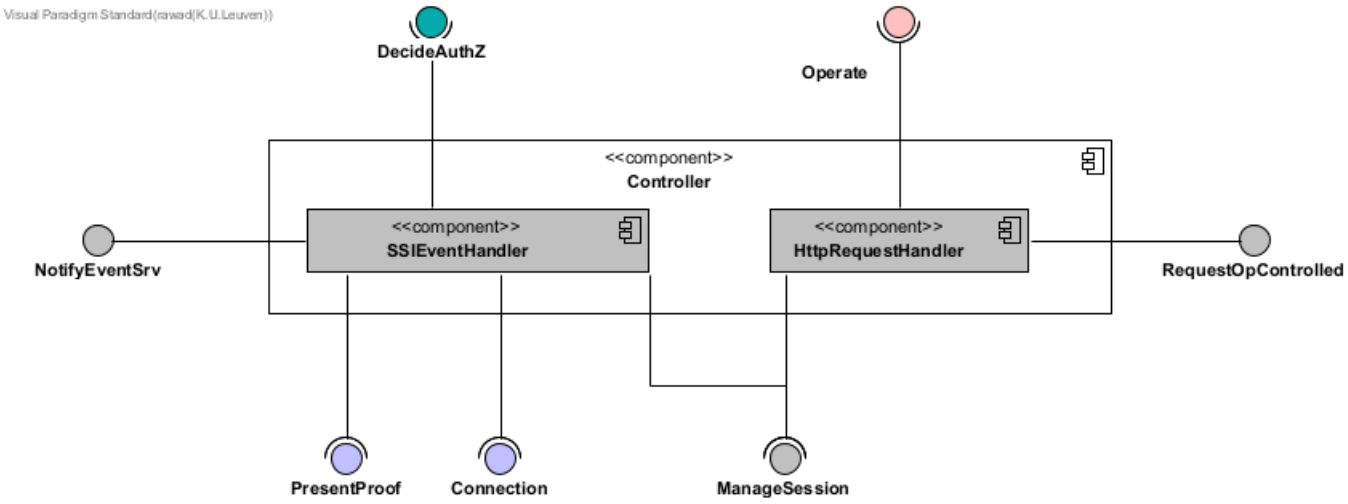


FIGURE 6.7: Decomposition of the Controller component.

The state establishment process occurs in two phases:

1. **Establishment of an OOB channel**: An Out-Of-Band DIDComm channel is established between the `AuthorizationEnforcer` and the client. This phase is further detailed in section 6.2.1.

2. **Presentation of credential**: Over the OOB channel, the `Controller` requests and receives the necessary permission credential to access the protected resource. This phase is further detailed in section 6.2.2.

6.2.1 Establishment of an Out-Of-Band channel

The *State Establishment* stage starts with establishing a DIDComm Out-Of-Band (OOB) channel. As described in chapter 3, DIDComm is a protocol that offers mutually authenticated peer-to-peer communication without relying on any third-party trust such as a certificate authority.

The connection establishment process is tracked in a `ConnectionSession` state machine object represented in figure 6.8. The connection establishment process is illustrated in both figures 6.9 and 6.8.

When a client requests a protected resource, the `HttpRequestHandler` creates a `ConnectionSession` object and generates a DIDComm connection invitation, thus transitioning the machine to state `WAIT-ACCEPT`. The connection invitation is displayed in the client's browser as a QR code to be scanned in their `ClientApp`. After accepting the invitation, the `AriesAgents` of the client and the `AuthorizationEnforcer`

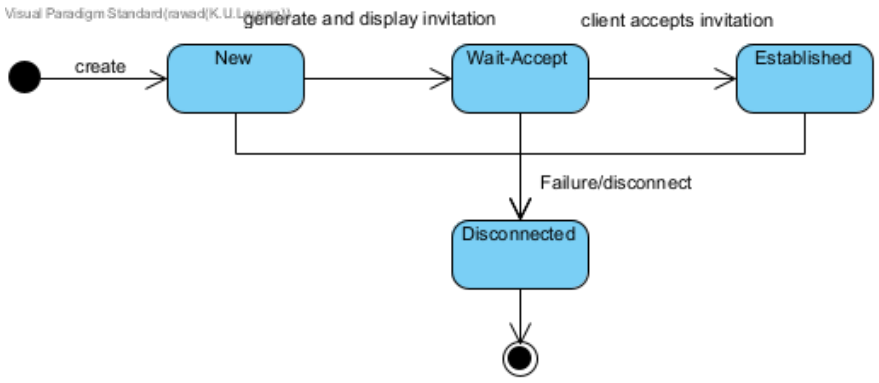


Figure 6.8: State machine representing a connection session.

have established a DIDComm OOB channel. Upon being notified of the connection establishment from the underlying SSI layer, the `SSIEventHandler` transitions the `ConnectionSession` to state `ESTABLISHED`. A failure or cancellation at any step results in the `ConnectionSession` being transitioned to the state `DISCONNECTED`.

6.2.2 Presentation of credential

After the OOB channel is established, the client and the `AuthorizationEnforcer` can communicate securely in a peer-to-peer fashion without relying on any third-party trust. The channel is used to acquire information from the client required to make the authorization decision. More precisely, a permission credential is required to prove the client's privileges over the protected resource. The credential request process is detailed in figure 6.10 and occurs as follows:

1. Via the OOB channel, the `SSIEventHandler` requests from the client a permission object to prove their privileges over the protected resource. The `AuthorizationSession` (previously illustrated in figure 6.1) is transitioned to the state `CREDS-WAIT`.

2. Via the OOB channel, the client presents the relevant permission credential. Upon reception, the `serverAriesAgent` notifies the `SSIEventHandler` of the event, which in turn transitions the `AuthorizationSession` to state `CREDS-RECEIVED`.

3. The `SSIEventHandler`, in communication with the underlying SSI layer, asserts the credential's integrity by verifying its digital signature. The client's public key is fetched from the blockchain for this operation. If authentic, the `AuthorizationSession` is transitioned to the state `CREDS-VERIFIED`.

After authenticating the presented permission credential, the `AuthorizationEnforcer` has acquired all the information required from the client to make an authorization decision. This point marks the end of the *State Establishment* stage.

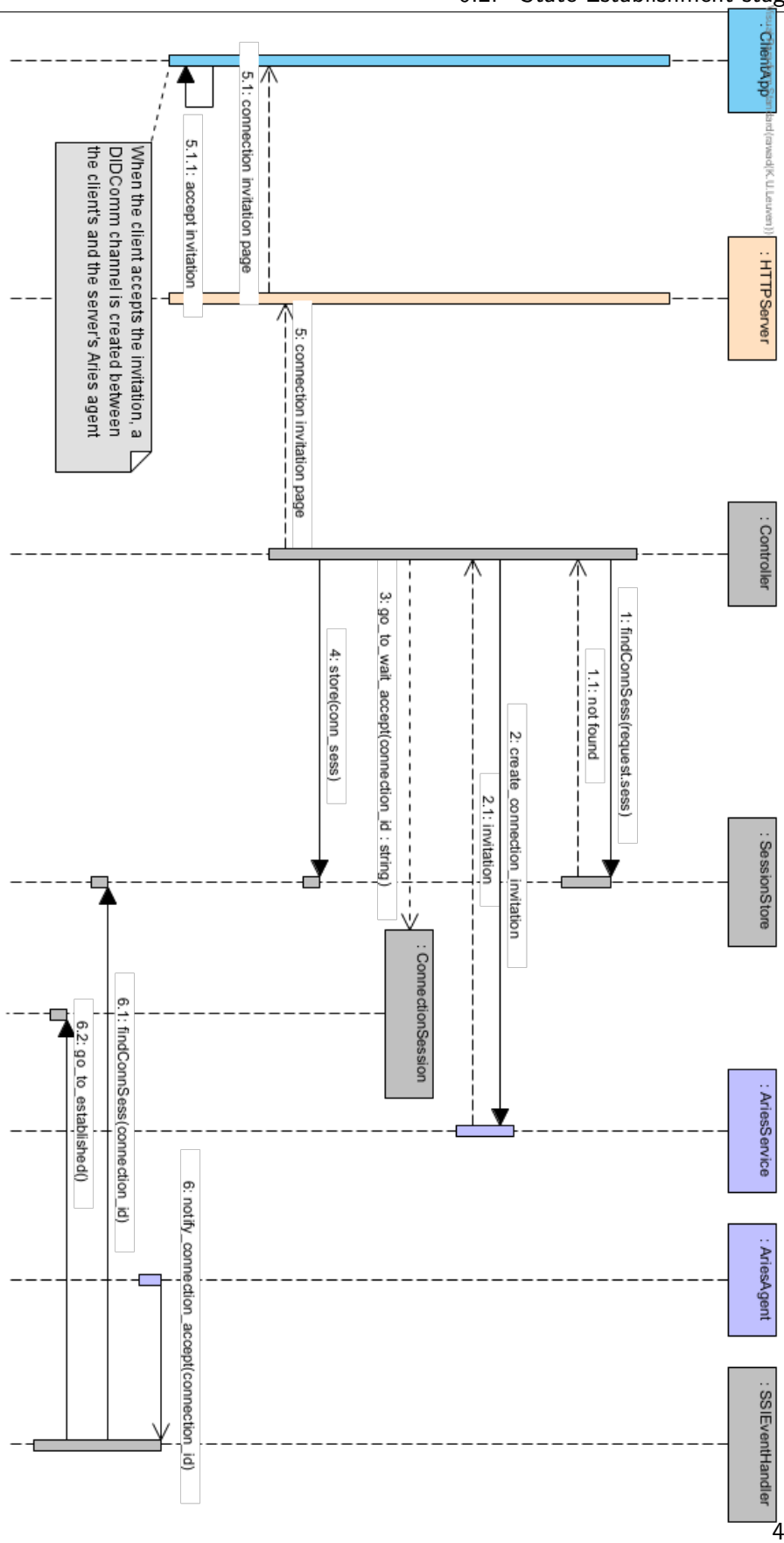


FIGURE 6.9: Sequence diagram for establishing an Out-Of-Band DIDComm channel.

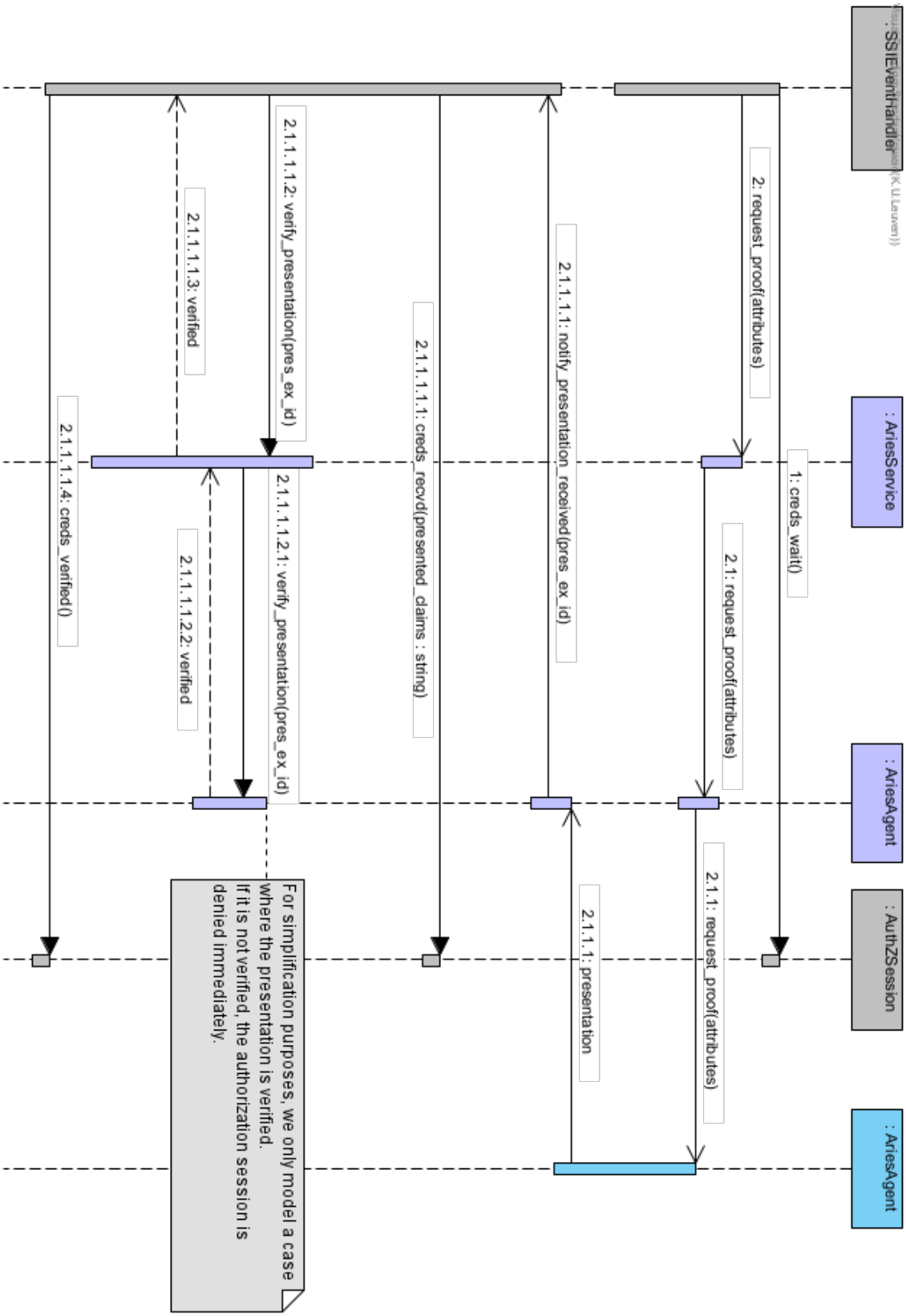


Figure 6.10: Sequence diagram for requesting a credential over an OOB DIDComm channel.

6.3 Authorization Decision stage

After completing the *State Establishment* stage, the `AuthorizationEnforcer` has received and authenticated a permission credential presented by the client. Consequently, the authorization process enters the *Authorization Decision stage*, in which the `AuthorizationEnforcer` decides whether the request must be granted or denied.

6.3.1 AuthorizationEngine component

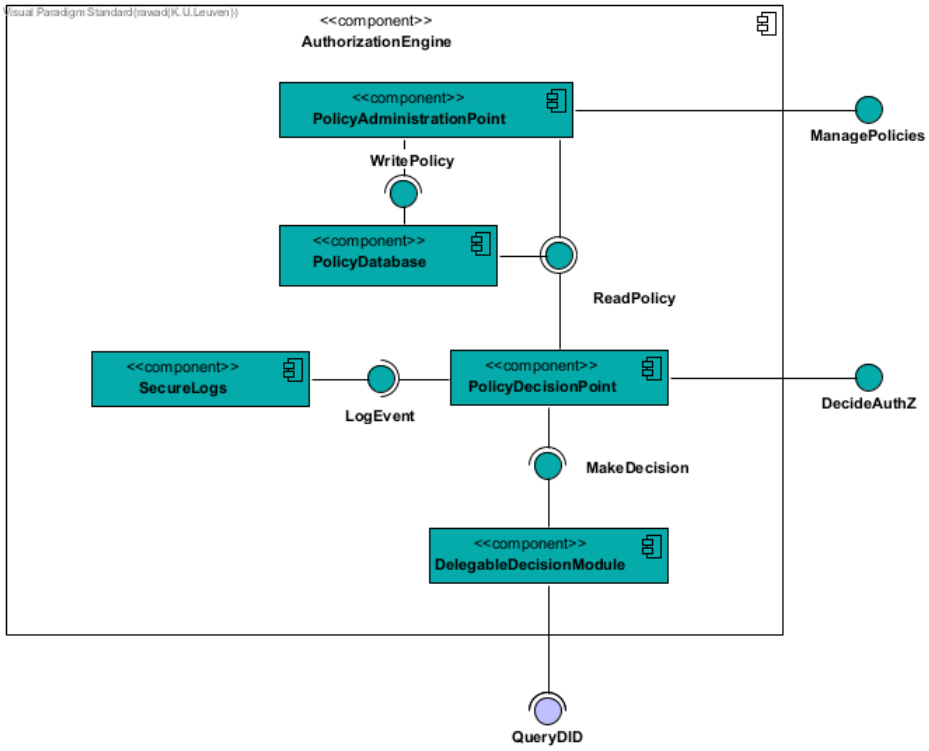


FIGURE 6.11: Decomposition of the Authorization Engine.

FIGURE 6.12: Class diagram of the MakeDecision interface.

The *Authorization Decision* stage is conducted by the `AuthorizationEngine` component detailed in figure 6.11, previously depicted in figure 6.5 for the bigger picture.

The `AuthorizationEngine` is composed of the following components that interact to output an decision:

- `PolicyDecisionPoint`: This component is a *facade* for the `AuthorizationEngine`. Given a permission object, it returns an authorization decision according to the configured access control policies. To improve modularity, the `PolicyDecisionPoint` delegates the decision-making to a configurable *decision module* that implements the `MakeDecision` interface.
- `MakeDecision`: Interface to be implemented by plugin decision modules. The interface is illustrated in figure 6.12.
- `DelegableDecisionModule`: Decision module that implements the `MakeDecision` interface, responsible of computing an authorization decision with support of delegable permissions. This component is further detailed in section 6.3.4.
- `PolicyAdministrationPoint`: Component via which an administrator manages access control policies.
- `PolicyDatabase`: Database to store policy objects as detailed in the following section 6.3.2.
- `SecureLogs`: Logging unit that stores access control events, ensuring that eventual security audits are possible, thus answering the requirement `AUDIT`. Designing a secure storage mechanism for logs is deemed out of scope in the project; however, we can suggest a remote server to store signed encrypted logs.

6.3.2 Access control policies

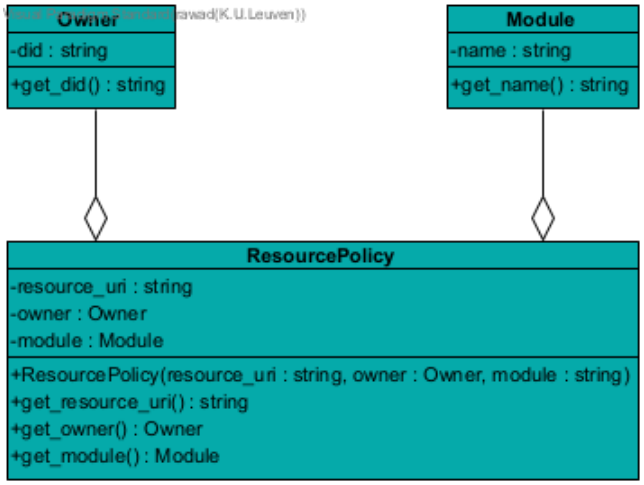


Figure 6.13: Class diagram of a ResourcePolicy.

The authorization decision is governed by pre-configured access control policies, `ResourcePolicy`, stored in the `PolicyDatabase`. `ResourcePolicy` objects (figure 6.13) are managed by the resource owner in the `PolicyAdministrationPoint`. The object contains the following fields:

- `resource_uri`: The URI of the resource governed by the current policy.
- `owner`: The DID of the resource owner.
- `module`: A decision module that implements the `MakeDecision` interface.

Policy rules are left to be expressed programmatically in the `DecisionModule`. While using a declarative Domain Specific Language (DSL) to express rules would have increased usability and maintainability, designing such language is out of scope for the project.

We note that the `PolicyDatabase`'s read and write interfaces, `ReadPolicy` and `WritePolicy` respectively, are separated with write access only provided to the `PolicyAdministrationPoint` in an effort to maintain the *Principle of Least Privilege* described in [14].

6.3.3 Authorization decision process

After acquiring a permission credential during the *State Establishment* stage (c.f 6.2), the authorization process enters the *Authorization Decision* stage to decide whether the client's request must be granted or denied. To this effect, the `SSIEventHandler` provides the `PolicyDecisionPoint` the acquired permission credential to compute an authorization decision. The `PolicyDecisionPoint` performs the following actions illustrated in figure 6.14:

1. Logs the request in the `SecureLogs` component.
2. From the `PolicyDatabase`, fetches the `ResourcePolicy` object corresponding to the requested resource (based on `Permission::resource_uri`).
3. Delegates the decision-making to the `DecisionModule` configured in the corresponding `ResourcePolicy` (based on `ResourcePolicy::module`). In our scope, we only consider the `DelegableDecisionModule`.
4. Logs the decision in the `SecureLogs` component.
5. Returns the decision back to the `SSIEventHandler`.

After receiving the decision, the `SSIEventHandler` transitions the `AuthorizationSession` state machine (figure 6.1) to the final state `DEC-AUTHORIZED` if authorized, or `DEC-DENIED` if denied, effectively marking the end of the *Authorization Decision* stage. The authorization process is ready to move to the next stage, the *Operation stage*, in which the client's request is applied to the protected resource.

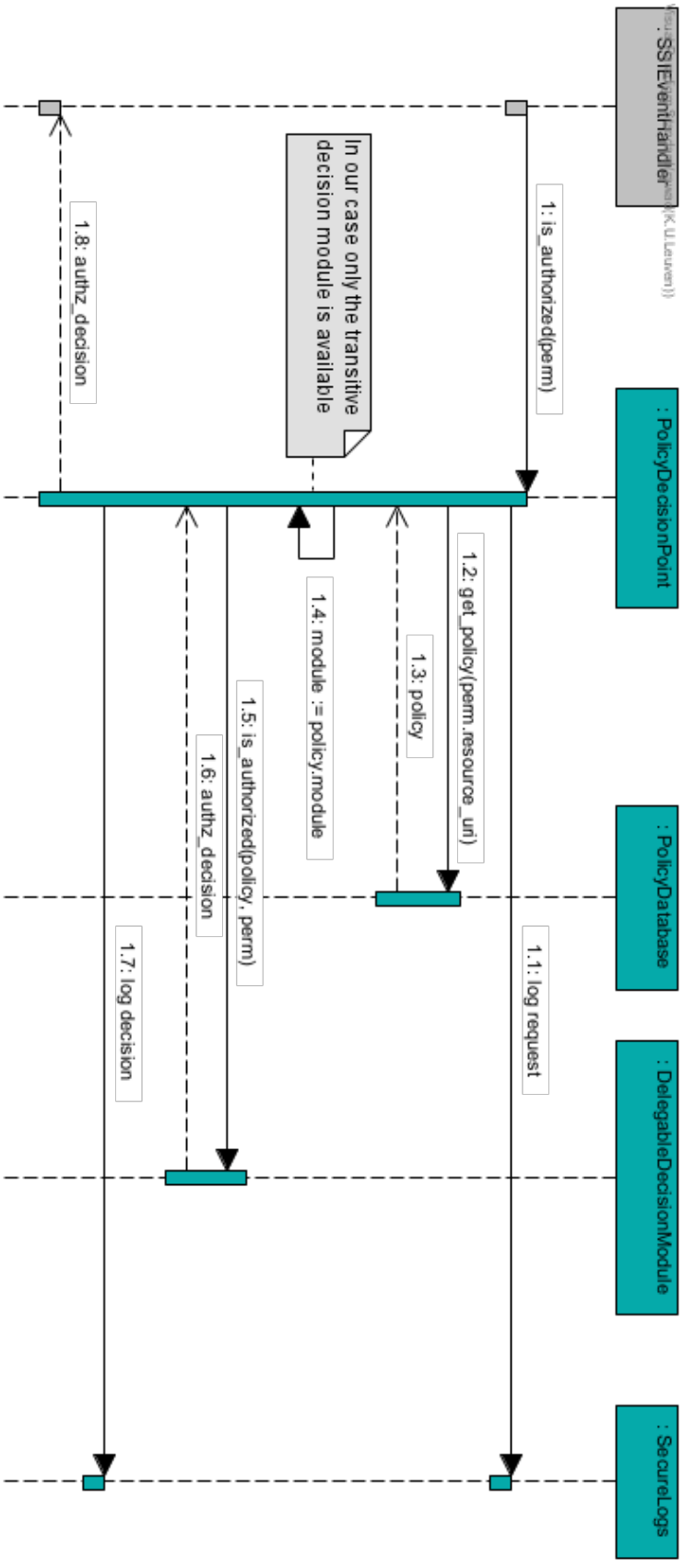


Figure 6.14: Sequence diagram for computing an authorization decision.

6.3.4 DelegableDecisionModule component

To compute an authorization decision, the `PolicyDecisionPoint` delegates the decision-making to a plugin *decision module* that implements the `MakeDecision` interface (figure 6.12), and configured in the resource's policy (in the field `ResourcePolicy::module`). It is the responsibility of the decision module to implement the decision-making algorithm in the inherited operation `is_authorized`. This design ensures that the `AuthorizationEngine` is easily extendable and can be customized to specific use cases.

The `DelegableDecisionModule` is a decision module responsible of computing an authorization decision with support for the delegable permissions presented in chapter 5. To this effect, the `DelegableDecisionModule` implements the delegable permission verification algorithm described in section 5.3. Since the algorithm requires application developers to implement the `policy_allows` function according to their application's uses, we consider our requirements. The objective of the `AuthorizationEngine` is to decide whether grant or deny a client's request based on the permission credential they provide and the configured access control policies. Therefore, in our implementation the following conditions must be checked:

1. **The client's request, the presented permission, and the policy target the same resource.**
2. **Matching owner.** The owner from which the delegated permission originates is the resource owner configured in the policy.
3. **Allowed operation.** The requested operation is one that is granted in the presented permission.

Hence, the `DelegableDecisionModule` implements the `policy_allows` function in pseudo-code as follows:

```
def policy_allows(request, perm_credential, policy):
    """
      Checks whether a client's request is allowed
      according to the access control policy,
      given a permission credential they presented.
      - request := The client's request.
      - perm_credential := The permission verifiable credential
                                        presented by the client.
      - policy := A corresponding access control policy.
      return := boolean, true if allowed else false
    """

    # The request, presented permission, and policy target the same
                                        resource
    if (policy.resource_uri != perm_credential.permission_object.
                                        resource_uri) or (policy.
                                        resource_uri != request.uri):
        return False

    # Matching owner
    if policy.owner.did != perm_credential.permission_object.owner:
```

```
        return False

    # Allowed operation
    if not request.operation in perm_credential.permission_object.
                                        operations:
        return False

    return True
```

6.3.5 Caching

The `AuthorizationEngine` requires issuers' verification keys to verify the digital signature of permission objects. Via the `AriesService`, the underlying SSI layer is employed to fetch verification keys from the blockchain given the corresponding DIDs. However, this operation can decrease the overall performance of the authorization process because it requires network communication that might suffer from instability and latency. Moreover, searching the blockchain is substantially slower than searching the local memory [26]. Therefore, we consider suggest issuers' verification keys in the `AriesService` to optimize the performance.

6.3.6 AuthorizationDecision object

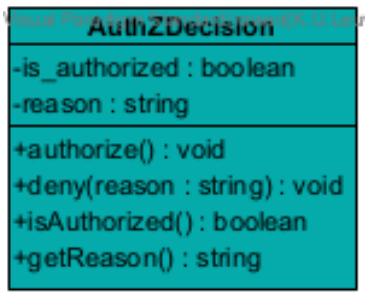


Figure 6.15: Class diagram of an AuthorizationDecision.

The *Authorization Decision* stage results in a decision describing whether to grant or deny the client's request. The decision is modeled as an `AuthorizationDecision` class illustrated in the class diagram figure 6.15. If the authorization decision is negative, a corresponding reason is stored, providing increased usability for the client and a means for debugging.

6.4 Operation stage

After completing the *Authorization Decision* stage described in section 6.3, the `Controller` has acquired an authorization decision, whether to grant or deny the client's request over the protected resource. If the decision is negative, the requested operation is dropped,

and the HttpRequestHandler answers the client with an HTTP 403 Forbidden error message. On the other hand, if the decision is positive, the Controller send the request to the ResourceOperator component (figure 6.5) to be applied to the resource Resource, then mediates its response back to the client. This process is represented in the primary sequence diagram 6.6.

The ResourceOperator component is responsible for applying a client's request to the protected resource. In the presented architecture, this component performs this operation without any without additional actions. However, the ResourceOperator can be extended with features such as a second logging unit for resource operations or a recovery mechanism in case an operation fails.

The presented authorization system is scoped to enforce access control to resources that are HTTP endpoints only. Since the AuthorizationEnforcer is accessible as an HTTP API, the client's requests are HTTP messages that can be forwarded to the resource as is without any adaptation. Providing access control to non-HTTP resources is deemed out of scope for our system; however, we can suggest two strategies to achieve compatibility. The first strategy consists of mapping the resource's interface to an HTTP API to be protected by the system. The second approach consists of extending the ResourceOperator with a variety of protocol adapter components to mediate the communication between the client and the resource while ensuring the translation between the two protocols.

After applying the requested operation on the resource and forwarding its response to the client, the *Operation* stage terminates, marking the end of the authorization process.

6.4.1 On managing decentralization

As illustrated in the deployment diagram 6.3, the AuthorizationNode is installed between the protected resource APINode and the user's ClientNode to ensure access control. We could imagine one AuthorizationNode component responsible for controlling the access to several resources owned by the same owner. In this matter, the owner controls the extent to which the system is decentralized. That is, we consider the context of building access control where the usage of smart devices must be authorized. An owner can install one AuthorizationEnforcer per smart device for complete decentralization. Alternatively, the owner can protect several devices with one gateway AuthorizationNode, thus decreasing the extent of decentralization. The more decentralized the system is, the more it is robust against single points of failure and attack; however, the more cumbersome and costly it becomes. It is the responsibility of the owner to ensure the right trade-off.

Chapter

7 Proof-Of-Concept

In chapters 4, 5, and 6, we described an architecture for decentralized access control that consists of decentralized permissions along with an SSI-based authorization system. The architectural boundaries of the system are limited to permission credentials and the `AuthorizationEnforcer` component. As such, other aspects of the system, including the client application and the user interface, are considered to be the responsibility of application developers or eventual architectural extensions. This decision aims to keep the architecture a minimal and cohesive secure base. In this chapter, we describe a proof-of-concept (POC) application whose objectives are to illustrate an example implementation of the architecture, prove its feasibility, and serve as a tool to conduct performance evaluation. The PoC consists of 2 modules, the client and the server, each of which is described in greater detail in the following sections.

7.1 Overview

The PoC consists of 2 modules, the client and the server. The client, represented as `ClientApp` in the primary 6.2 and deployment 6.3 diagrams of the system), is the application used by users (owners, issuers, and holders) to store, present and receive permission credentials. The server is the `AuthorizationNode` illustrated in the the deployment diagram 6.3, hosting the `AuthorizationEnforcer` and responsible for enforcing access control to protected resources.

Both modules are built as Django[1] web applications on top of an underlying SSI layer based on a Hyperledger Indy[2] blockchain, the British Columbia test blockchain[3]. Each module interacts with the SSI layer via an `AriesAgent`[4], a component that performs SSI operations accessible via an HTTP API. The `AriesAgent` readily provides several functionalities required by the modules, namely:

For both the client and server:

- A means to create, accept and interact with DIDComm channels.

[1] https://www.djangoproject.com/
[2] https://www.hyperledger.org/use/hyperledger-indy
[3] http://test.bcovrin.vonx.io/
[4] https://github.com/hyperledger/aries-cloudagent-python

- A means to fetch an entity's public key from the SSI blockchain.

For the client:

- A wallet to securely store and manage permission credentials.
- A means to present verifiable credentials to verifiers (the server).
- A means to receive credentials from issuers,and issue credentials to holders.

For the server:

- A service to request verifiable credentials from holders and verify their cryptographic integrity.

In the following sections, we describe the implementation of the client and server modules in further detail.

7.2 Client application

The client is an application for users to manage permission credentials they received, issue or delegate new permissions to holders, and present permissions to verifiers (the server). The client consists of one main page, depicted in figure 7.1, from which various operations are performed. Ideally, a client application should minimize the likelihood of users making mistakes that affect security. Hence simplicity, high usability, and accessibility are desired.

View and manage permissions

The client allows users to view permissions they issued, delegated, and received. This feature complies with `USABILITY`. Moreover, users can delete permissions they received, effectively amounting to a revocation, since permissions are user-centric objects stored client-side only.

Issue and delegate permissions

The client allows users to issue permissions to resources they own or delegate permissions they received to other users. These operations result in a connection invitation (figure 7.2) being displayed on the issuer's screen, to be accepted by the holder in their client application, and after which the permission credential is automatically transmitted over the DIDComm channel. We note that an adversary attempting to accept the invitation before its intended receiver would fail because of the `sub` field of the permission object which specifies the intended receiver's DID. An attacker cannot impersonate a DID without stealing their private key.

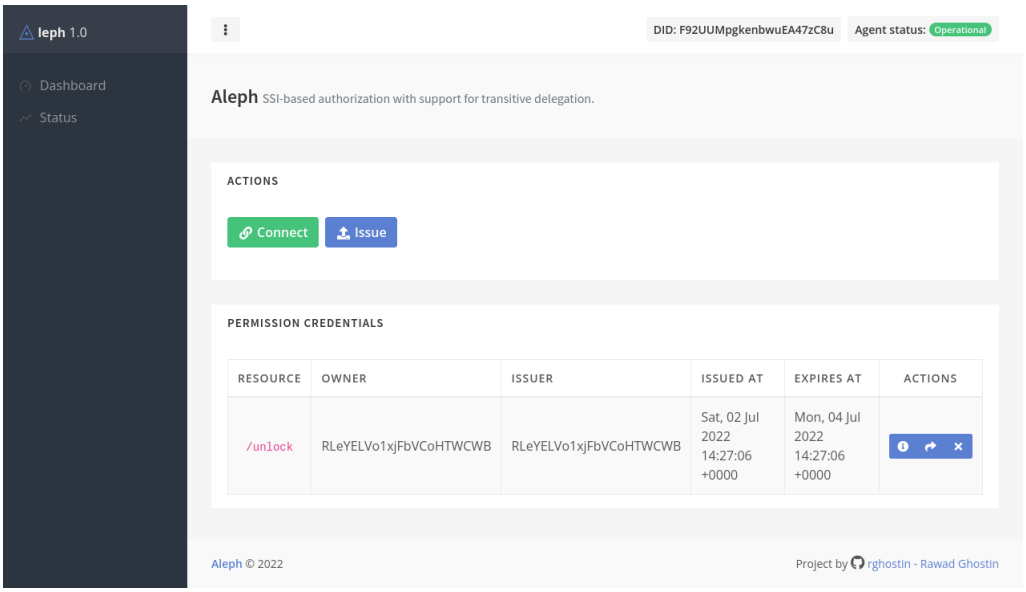


FIGURE 7.1: Screenshot of the client application.

Accept connection invitations

The client allows users to accept DIDComm channel invitations. After accepting a connection, it is the responsibility of the connection initiator to conduct the operations. Hence, if the connection is initiated by a verifier (the server), the client is requested to present a permission credential proving the user's privileges. If the connection is initiated by another client to issue or delegate a permission (as in figure 7.2), the client receives a permission credential to be stored in the wallet.

7.3 Server application

The server consists of an implementation of the `AuthorizationEnforcer` component and is responsible for enforcing access control to protected resources.

Users requesting a protected resource via their browser must prove their privilege to the server. Hence, an authorization page is displayed (figure 7.3) requesting the user to scan a connection invitation in their client app. The authorization process described in chapter 6 executes between the server and the client app to decide whether the client's request must be granted or denied. In the meantime, the user's browser is actively waiting for redirection to the protected resource if authorized or an *access denied* page otherwise.

Furthermore, the server offers an administration page to manage access control policies `ResourcePolicy`.

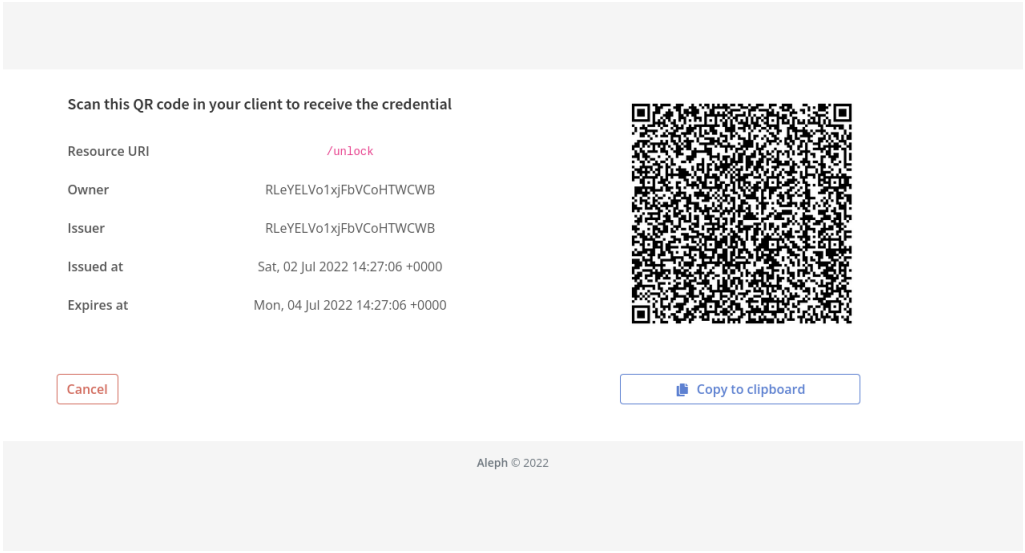


Figure 7.2: Screenshot of the client application.

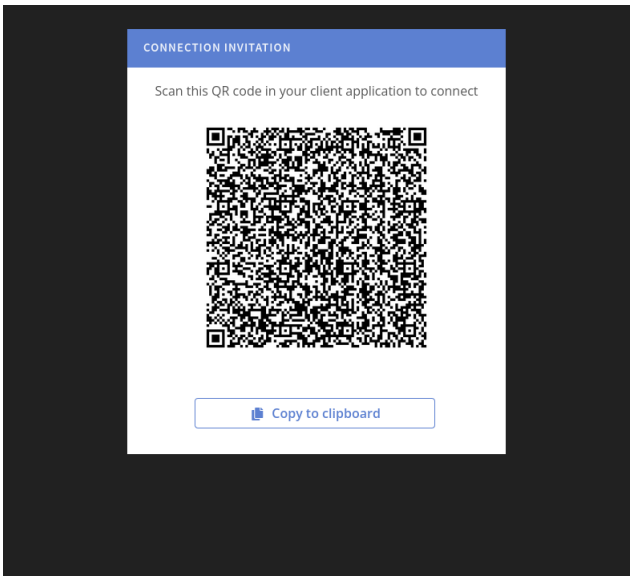


Figure 7.3: Screenshot of the server authorization guard.

Chapter

8 Evaluation

In this chapter, we evaluate the presented solution to be compliant with the requirements. First, a performance evaluation is conducted on the proof-of-concept software to measure the time of various operations and the size of permission objects. Then, we recapitulate the architecture's compliance with all requirements stated in section 2.1.

8.1 Performance evaluation

In this section, we evaluate the performance of core operations in our system, namely the time to delegate permissions, the time to make an authorization decision, the size of delegable permissions, and the time to generate a DIDComm invitation. For each essential operation, we also compare its performance to state-of-the-art solutions, the DeFIRED [27] and WAVE frameworks [2]. The performance evaluation is conducted using the PoC described in chapter 7 and employing the BCovrin test SSI Blockchain[1]. The tests ran on a single computer with an AMD Ryzen 9 3900X CPU and 32 GB of RAM.

8.1.1 Performance measures

Permission delegation time

The *permission delegation time* is the time required for an issuer to generate a permission object before transmitting it to the recipient. We measure the permission delegation time for delegation depths of 0 to 120. The results yielded by the test are represented in figure 8.5. A comparison with the state of the art is depicted in figure 8.6. We observe that a delegation of depth 15 takes around 0.0005 s, which is almost equal in the Wave framework compared to around 4 s in the DeFIRED framework. DeFIRED suffers from poor performance due to its usage of IBE to ensure topology confidentiality. The performance increase is explained by the fact that the presented solution relies on SSI, which employs more efficient cryptographic schemes, as covered in chapter 3. Moreover, the solution optimizes speed by minimizing the amount of data encrypted during delegation by using compression. As covered in section 5.4, The permission

[1] http://test.bcovrin.vonx.io

delegation time is substantially lower than one second for any realistic delegation depth, thus compliant with the requirement PERFORMANCE_TIME.

Permission delegation size

The permission delegation size is the size of permission objects generated by delegation. We measure the permission delegation size for delegation depths of 0 to 120. The results yielded the test are represented in figure 8.3. A comparison with the state of the art is depicted in figure 8.4. We observe that a delegation of depth 120 weighs around 90 kB compared to 20 kB in the DeFIRED framework. Similar data for the Wave framework could not be retrieved. In the comparison to our solutions, in DeFIRED framework, proof objects' size depends on the amount of delegations but not the size of the policy representation. This property allows DeFIRED to have smaller objects.
Permission delegation size is substantially smaller than 150 kB for any realistic delegation depth, thus compliant with the requirement PERFORMANCE_SIZE.

Authorization decision time

The authorization decision time is the time for the AuthorizationEngine to compute an authorization decision. We measure the authorization decision time for delegation depths of 0 to 120. The permissions used in the test all result in a positive authorization decision which is the worst case in execution time. The measures yielded in the test are represented in figure 8.5. A comparison with the state of the art is depicted in figure 8.6. We observe that in our system, a permission of depth 15 is authorized in around 0.005 s compared to 0.025 s in the Wave framework and 0.055 s in the DeFIRED framework; that is the time to authorize a permission of depth 15 is up to 5 times faster than WAVE and 11 times faster than DeFIRED. As stated in section 2.2, both the WAVE and DeFIRED frameworks suffer from poor performance due to its usage of IBE. The performance increase is explained by the fact that, first, the presented solution relies on SSI, which employs more efficient cryptographic schemes, as covered in chapter 3. Moreover, the solution optimizes speed by minimizing the amount of data encrypted during delegation by using compression. Furthermore, caching public keys (*c.f* section 6.3.5) avoids networks latency and blockchain operations giving the solution an additional performance increase.
The authorization decision time is substantially lower than one second for any realistic delegation depth, thus compliant with the requirement PERFORMANCE_TIME.

Connection invitation generation time

The connection invitation time is the time for the Controller to generate a connection invitation. The connection generation time has been measured by computing the mean time over 100 connection generations. A comparison with the state-of-the-art connection invitation times is depicted in figure 8.7. We observe that generating a connection invitation in our system takes around 0.02 s compared to 0.16 s in the Wave and DeFIRED frameworks.

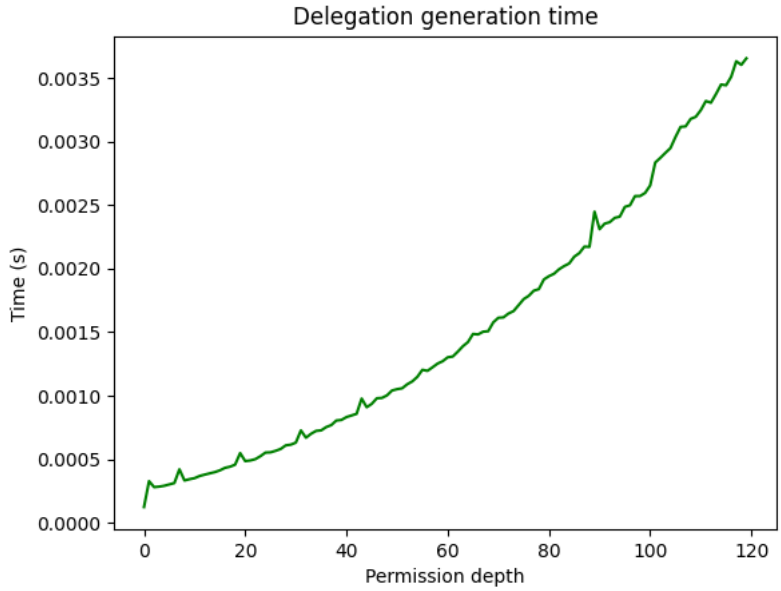


FIGURE 8.1: Time performance of permission delegation.

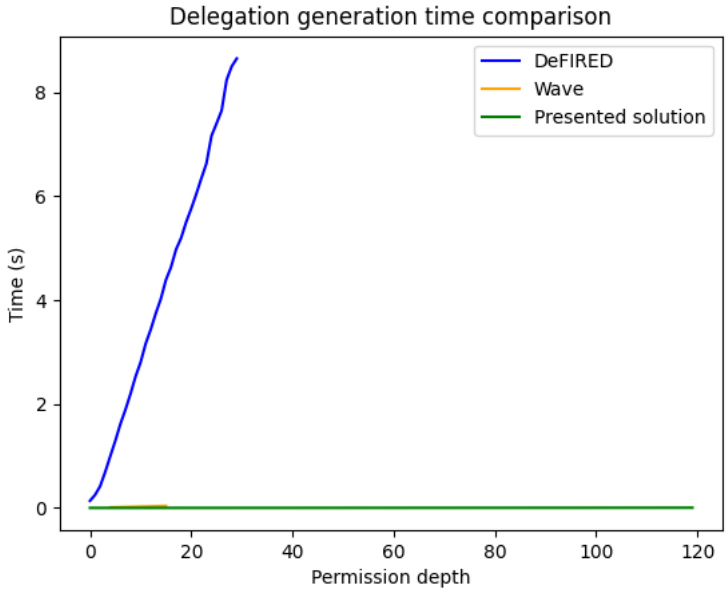


FIGURE 8.2: Comparison of delegation time with the state of the art.

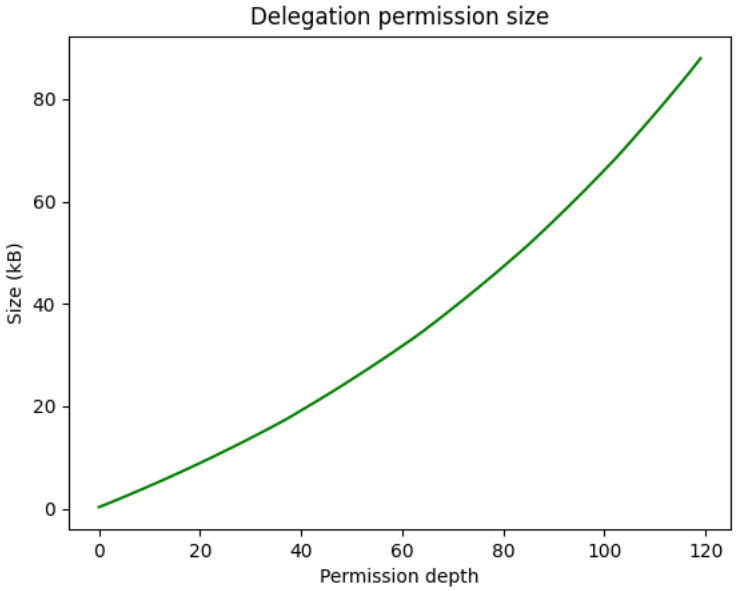


Figure 8.3: Size performance of permission delegation.

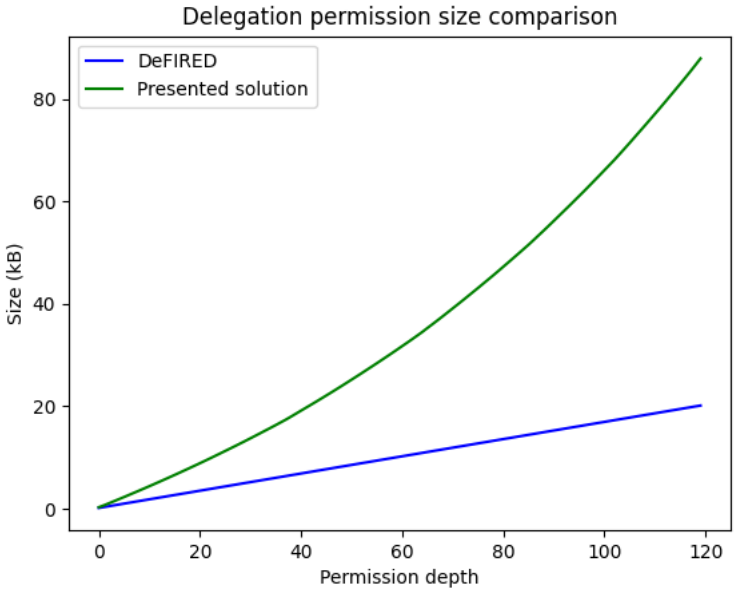


Figure 8.4: Comparison of delegation size with the state of the art.

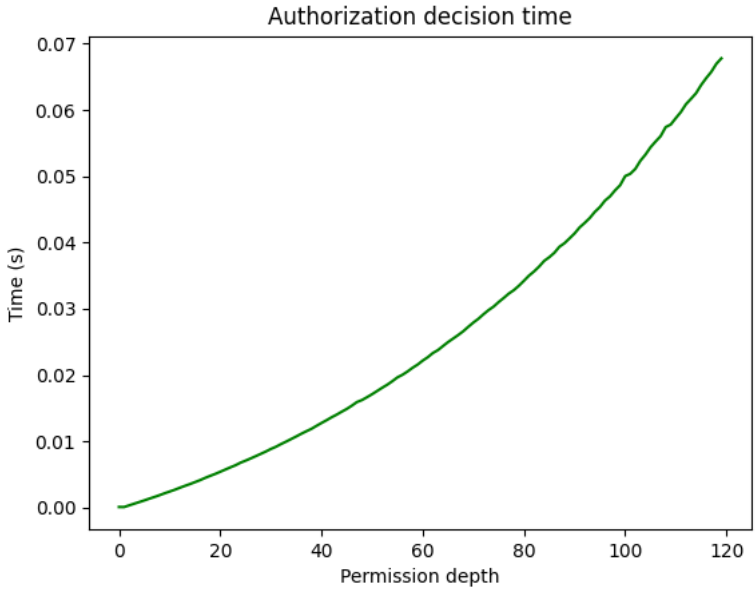


FIGURE 8.5: Time performance of authorization decision.

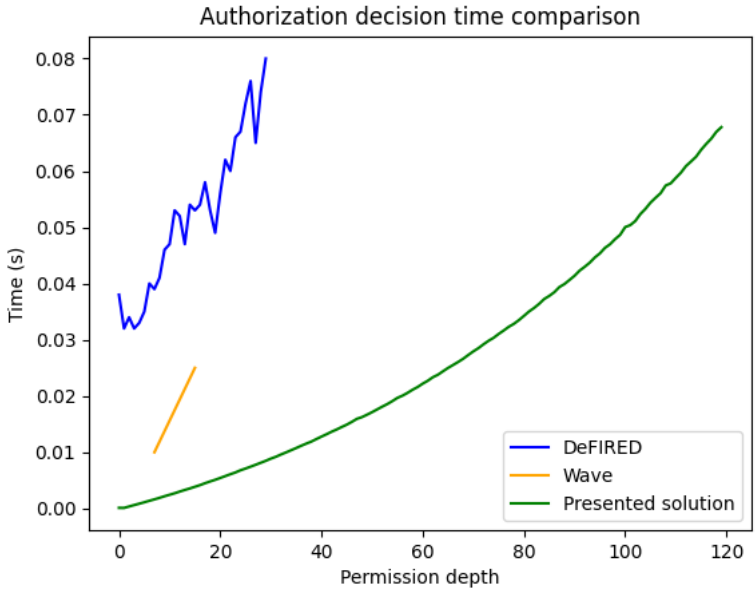


FIGURE 8.6: Comparison of authorization decision with the state of the art.

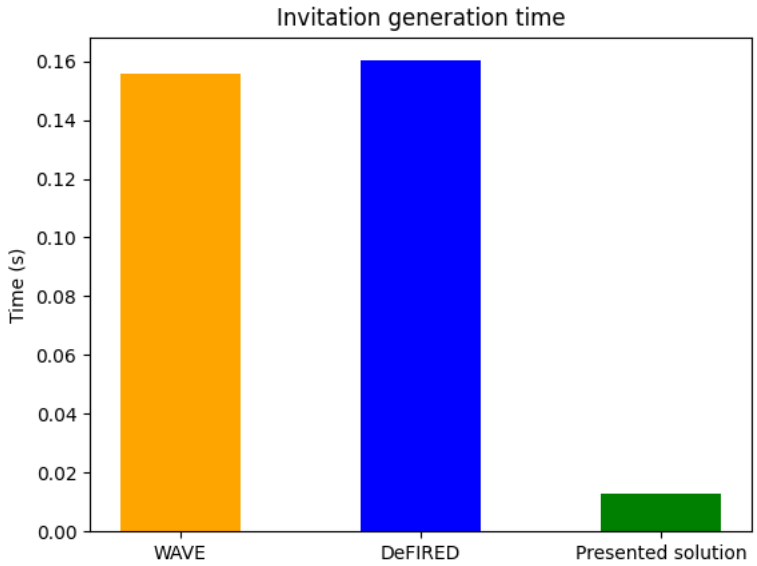


Figure 8.7: Comparison of connection invitation time with the state of the art.

8.1.2 Threats to validity

We consider some threats to the validity of the evaluation.

Firstly, the performance is potentially affected by the size of the blockchain. The performance tests have been conducted employing the BCovrin test SSI Blockchain[1], which at the time of writing (July 2022) stores around 371000 transactions. The size of the blockchain potentially influences the time for searching for a verification key, thus affecting our system's measured performance.

Secondly, compared to the WAVE and DeFIRED frameworks, the `AuthorizationEnforcer` caches verification keys. With caching, in some cases, the system avoids communicating with the blockchain and searching for a key, which increases performance. We have no information on whether the state-of-the-art frameworks use any form of caching.

8.2 Compliance validation

In this section, we validate the solution's compliance with the requirements. For this purpose, for each requirement in section 2.1, we outline how the architecture answers it and reference the sections responsible for covering the solution in detail.

- **CORE_DELEGATION:** *(Holders are allowed to delegate their permissions at their discretion. Permissions delegations and revocations must be transitive).* Delegable permission objects (*c.f* chapter 5) are an extension of decentralized permission objects that can be delegated granting the receiver privileges in a

transitive manner. Delegable permission objects can be revoked individually as an SSI verifiable credential is revoked. The delegable permission verification algorithm covered in section 5.3 ensures the transitivity of revocation.

- **CORE_DECENTRALIZED_TRUST:** *(The system must not rely on centralized trust to avoid single points of failure and attack).* The system is based on decentralized permission objects built on top of SSI, a model for decentralized digital identity. This architecture allows for avoiding centralized trust. We provide a compact overview of decentralized aspects in the presented solution. 1) In contrast with centralized authorization where organizations control user's identity through accounts, decentralized permissions are issued and received by entities whose identity is decentralized (*c.f* chapter 3). 2) Instead of being stored in centralized organizations databases, permissions objects are only stored in holders' wallets (*c.f* chapter 4). 3) All communications are DIDComm peer-to-peer channels that do not rely on centralized trust, such as an certificate authority, to secure the channel (*c.f* section 3.3.7). 4) The owner of the resources controls the extent to which access control to resources is decentralized. Owners can deploy an `AuthorizationEnforcer` per resource, thus allowing a complete decentralization. Alternatively, they can group several resources under an `AuthorizationEnforcer` gateway, thus reducing the system's decentralization. access control is enforced by components controlled by their owner only (*c.f* chapter 6).

- **CORE_CONFIDENTIALITY_TOPOLOGY:** *(The topology formed by permissions delegations of a resource must be confidential to all users except the owner of that resource).* Permission hierarchies are encrypted with the owner's public key, ensuring their confidentiality to all parties except the owner. Confidentiality of topology is covered in section 5.4.

- **INTEGRITY**: *(The integrity of a permission must be verifiable by the verifier).* Permission objects are implemented as verifiable credentials digitally signed by the issuer. Moreover, upon presentation, the permission is digitally signed by the holder. The verifier having access to other entities' public keys on the blockchain can verify the permission signatures and ensure their integrity. Permission objects are covered in 4.

- **AUTHENTICATION:** *(An adversary that compromises a permission object cannot reuse it to escalate their privileges. That is, holders presenting a permission must be authenticated, and their liveness must be asserted).* When presenting a permission, the holder digitally signs the permission along with a nonce issued by the verifier to mitigate replay attacks. This mechanism, covered in section 4.1, ensures authentication and liveness.

- **AUDIT:** *(All events related to authorizing access to a controlled resource must be auditable by the owner of this resource).* The `AuthorizationEngine` component of the `AuthorizationEnforcer` ensures the logging of all authorization events by the `SecureLogs` component. The architecture of the `AuthorizationEngine` is covered in section 6.3.1.

- **CRYPTOGRAPHY_STRENGTH:** *(All encryption and digital signature keys must provide at least 112 bits of security).* The architecture of permission objects and confidential permission delegation mandate mandate employing cryptographic keys of at least 112 bits of security (*c.f* 4.2 and 5.4). We assume that the underlying SSI layer meets this requirement as well. In practice, both our architecture and the underlying SSI layer employ ED25519 with 118 bits of security keys [16].
- **NONREPUDIATION:** (A holder must explicitly either accept or refuse a permission that is being granted to them). SSI wallets, which store permission credentials, are by design *consent-driven*. That is, no action is ever taken without the holder's authorization. When receiving a permission credential, holders must accept explicitly accept it. SSI wallets' builtin features are covered in section 4.4.
- **HOLDER_REVOCATION:** *(Holders can revoke permissions that were granted to them).* Permissions are user-centric verifiable credentials only stored client side in the holder's wallet. Therefore, deleting a permission from the wallet effectively results in its revocation. SSI wallets' builtin features are covered in section 4.4.
- **PERFORMANCE_TIME:** *(The processing time of interactive operations must be lower than one second to retain the user's attention and flow of thought).* As measured in section 8.1, delegation and authorization time is significantly lower than one second for any realistic delegation depth.
- **PERFORMANCE_SIZE:** *(Permission objects size must not exceed 150 kB bytes to reduce cost and efficiency of storage and bandwidth, as well as reduce the time of network transfer).* As measured in section 8.1, permission objects' size are significantly lower than 150 kB for any realistic delegation depth.
- **USABILITY:** *(An issuer can consult a list of all permissions they directly granted. A holder can consult a list of all permissions they received).* Permissions are implemented as SSI verifiable credentials stored in holders' wallets. SSI wallets allow users to consult credentials they issue and receive. SSI wallets' builtin features are covered in section 4.4.

We conclude that the presented architecture is compliant with all requirements.

Chapter

9 Discussion

In this chapter, we discuss the quality of the presented solution. First, an elementary security analysis of the solution is carried out with a reduced threat model and an availability analysis. The security analysis aims to elucidate the main threats that pose a security or privacy risk to the system and consider possible mitigations. Next, the most critical limitations of the system are outlined, uncovering potential drawbacks of the solution. Finally, the solution is compared to the state of the art, putting forward improvements and pitfalls.

9.1 Security analysis

9.1.1 Threat model

In this section, we identify and analyze the system's potential security and privacy concerns to build a threat model. The objective of threat modeling is to elucidate the main threats that pose a security or privacy risk to the system and consider possible mitigations. For the threat modeling, we rely on the data flow diagram of the system illustrated in figure 9.1.

Threats

- **Title:** Unauthorized access with phishing using connection invitation.
- **Type:** Spoofing, Elevation of privileges.
- **Description:** An adversary can gain unauthorized access to protected resources by tricking an unauthorized user into using their identity. The attack happens as follows. First, the attacker requests the resource to be presented with a connection invitation (data flows 1 till 2.2). Then, in a phishing attack, the attacker sends the connection invitation to a legitimate authorized user to trick them into accepting the invitation (simulating data flow 2.3). When the victim accepts the connection invitation and presents their credential, data flows 2.4 to 5.1 are executed, and the attacker is authorized to access the protected resource (data flow 5.2).
- **Likelihood:** High. The attack is simple and does not require the attacker's advanced technical skills or resources.

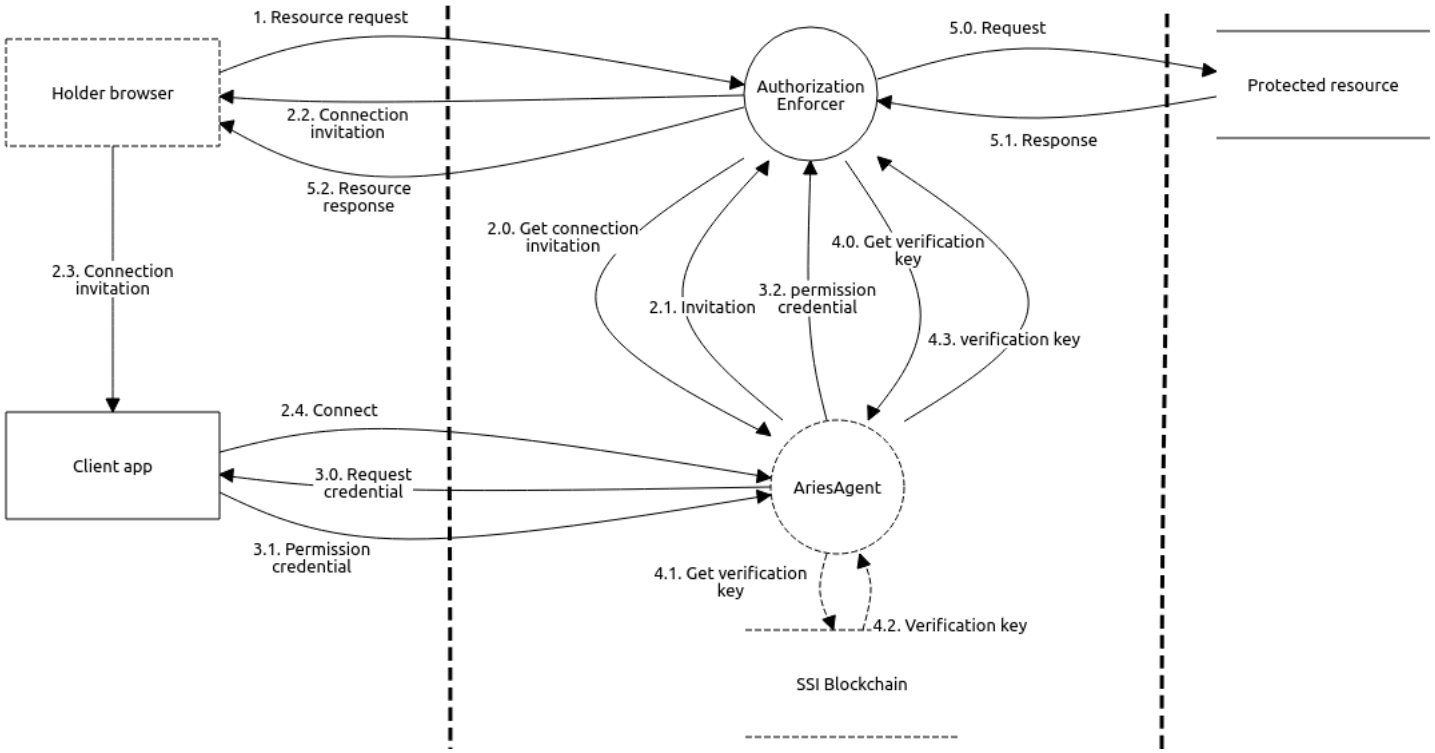


Figure 9.1: Data flow diagram of the system.

- **Impact:** High, as the attack results in unauthorized access to the protected resource.
- **Possible mitigation:** This attack is similar to attacks on the OAuth device code flow, for example, as covered in [18]. Research for effective defense mechanisms for such attacks is still ongoing. Suggestions involve restrictive access control policies that only allow whitelisted apps and devices to authenticate; and strong detection and prevention mechanisms to flag and deny suspicious requests.

- **Title:** unauthorized access by using the holder's client application.
- **Type:** Spoofing, Elevation of privileges.
- **Description:** In data flow 2.4 and 3.1, an adversary with access to a holder's client application can use it to present the holder's permissions gaining unauthorized privileges. The adversary might attempt to access the client application by stealing the holder's device or an evil-maid attack.
- **Likelihood:** Low. This attack requires physical access and is not stealthy in the case of stealing the device as the holder likely notices their device missing.
- **Impact:** High, as the attack results in unauthorized access to the protected resource.

- **Possible mitigation:** The client application must enforce a local authentication mechanism, for example, a biometric, to unlock the access to the application and when presenting a permission. Moreover, the client application must lock itself after a short period of inactivity to limit the attack window.

- **Title:** Private key compromise.
- **Type:** Spoofing, Tampering, Denial of service, Elevation of privileges.
- **Description:** Compromising the private key of entities allows for several high-impact attacks such as hijacking the ownership of a resource or crafting rogue permissions. This threat is covered in great detail in section 9.1.1.
- **Likelihood:** Very low.
- **Impact:** Very high.
- **Possible mitigation:** Covered in 9.1.1.

- **Title:** Malicious holder grants unauthorized delegations.
- **Type:** Escalation of privileges.
- **Description:** A holder that is an adversary can delegate permission they received to unauthorized entities, thus granting them unauthorized access to protected resources.
- **Likelihood:** Medium.
- **Impact:** Medium.
- **Possible mitigation:** The likelihood and impact of the attack can be limited by extending the delegable permission model to support rules that must be verified before an entity can delegate its permission. For example, rules could cover whether a permission can be further delegated, the number of allowed delegations, and a blacklist or whitelist of recipient holders. Other mitigations include respecting the principle of least privileges [14] when granting rights to holders, and to monitor and audit logs constantly.

- **Title:** Delegation depth inference.
- **Type:** Information disclosure.
- **Description:** Delegable permissions, as covered in the architecture chapter 5 store their hierarchy recursively by containing their parent permission. As Curve25519 encryption maintains plain-text size, in some cases, an adversary holder can infer the delegation depth of their permission based on the length of the `hierarchy` field.
- **Likelihood:** Medium, the operation is easy to execute but yields no useful information.
- **Impact:** Low, the permission depth is not sensitive information nor can be used to conduct further attacks.
- **Possible mitigation:** Employ padding with the encrypted hierarchy.

Private key compromise

The system's security properties guaranteed to an entity are reduced to the confidentiality of the entity's private key. An adversary with access to private keys can conduct high-impact attacks compromising the system's security.

Firstly, we consider the case of an issuer key compromise. An adversary that compromises the private key of an issuer can use it to craft permission delegations on their behalf, thus employing it to delegate themselves unauthorized privileges to protected resources. In addition, if the adversary is a holder that received a permission credential from an issuer they compromised, they can tamper with their permission credential to grant themselves more privileges or extend the expiry date. Furthermore, compromising the private key of an issuer allows the adversary to revoke permissions they previously delegated by publishing revocation objects on the blockchain, thus affecting the availability of resources.

Secondly, we consider the case of a resource owner key compromise. An adversary with access to an entity's private key can modify the entity's DID document published on the blockchain. Hence, the adversary can exchange the public key assigned to the entity's DID with one they control, effectively hijacking their identity and hence, all resources they own. Moreover, an adversary that compromises an owner's private key can use it to produce legitimate signatures on behalf of the owner, thus employing it to craft permissions issuing themselves unauthorized privileges to protected resources.

Finally, we consider the case of a holder key compromise. An adversary that compromises a holder's private key can use it to craft for themselves a permission delegation, granting themselves unauthorized access to protected resources.

In the state-of-the-art SSI, private keys are stored and managed by wallets responsible for ensuring the confidentiality of the keys. The wallets provide several security layers such as key encryption and execution in Trusted Execution Environments (TEE), for example, the Android Keystore System[1] on android devices. We assume that the extraction of the keys from wallets is an arduous task. Thus, a private key leak is very unlikely.

9.1.2 Availability analysis

High availability is a desired and required property in the presented authorization system. Maintaining availability and robustness against DOS attacks is crucial in use cases such as building access control described in section 1.2. This section covers an availability analysis of the system occurring in three scenarios: blockchain downtime, `AuthorizationNode` failure, and resource unavailability.

Firstly, we consider the case of blockchain downtime. In SSI, the blockchain is the root of trust, holding all entities' DIDs and their associated public key. The system built on top of an SSI layer employs the blockchain to retrieve verification keys of entities given their DID. This operation occurs server-side during permissions verification and

[1] https://developer.android.com/training/articles/keystore

client-side during permissions delegation when the resource owner's public key is used to encrypt the hierarchy of the generated permission. Unlike delegations, an owner issuing a permission directly to a holder (depth 1) does not require access to the blockchain because there is no hierarchy to encrypt. Therefore, a blockchain downtime affects the availability of permissions delegation and verification. Nonetheless, we note that the caching mechanisms implemented both client-side (section 5.4.4) and server-side (section 6.3.5) in some cases can improve the availability. It is crucial to indicate that a blockchain downtime is highly unlikely as per the inherent design of the blockchain, which aims to maximize availability by being decentralized.

Secondly, we consider the case of resource downtime. A resource being inaccessible only causes requesting that resource to fail. The resources to which accesses are being controlled lay outside the architectural and trust boundaries of the system (as depicted in figure 9.1 and 6.2). Therefore, the system is not responsible for enforcing their availability. However, a resource being inaccessible does not affect the availability of any other component.

Thirdly, we consider the case of an unavailability in the `AuthorizationNode` (figure 6.3) hosting the `AuthorizationEnforcer`. All accesses to protected resources must be controlled by the `AuthorizationEnforcer` that employs an `AriesAgent` and is accessible via an `HTTPServer`. The unavailability of any of these components causes the `AuthorizationNode` to fail. Consequently, the resources protected by the failed node become unavailable. However, as indicated in section 6.4.1, it is recommended to minimize the number of resources protected by one `AuthorizationNode`, to maximize decentralization and, thus, availability.

Finally, we evaluate what it takes for an adversary to compromise the availability of the whole authorization system. In centralized systems, compromising the central authorization server is sufficient to shut down all access to all resources. However, the presented architecture provides decentralized trust and access control. Thus, compromising the availability of the whole system would consist of attacking each deployed `AuthorizationEnforcer`.

9.2 Limitations of the system

This section discusses some limitations of the system and their potential solutions. The system's limitations are characteristics of the design that might negatively impact its utility.

Static permissions limits updates

Decentralized permissions are static; issuers cannot inherently update the permissions they issued. In the current architecture, updating a permission consists of revoking it, then issuing a new updated version of it. The impact of this characteristic is further enhanced in delegable permissions, as revoking a permission that is an internal node in

the permissions tree by design results in a revocation of its whole sub-tree. Therefore, updating a delegable permission requires its whole sub-tree to be reissued. To address this limitation, we can suggest a potential solution based on the concept of refresh tokens [3]. The solution would consist of decoupling in the permission tree, distinguishing its structural topology to be more permanent (analogous to refresh tokens) and the privileges of each node to be more volatile (analogous to access tokens). Thus, the decoupling would result in two separate trees.

Required connectivity during issuance

During a permission issuance or delegation, recipients must be online to receive the credential. In most centralized authorization systems, permissions are managed directly in the authorization server; thus, issuing a permission to a user is a server-side-only operation. However, decentralized permission credentials are user-centric objects stored solely in the holder's wallet. Therefore, issuing a permission requires the user to be online to accept the credential into their wallet, resulting in a slight usability inconvenience. A potential solution to this limitation is to host users' SSI agents in the cloud, substantially increasing their availability to receive credentials. Then, when the user is back online, they will receive a credential offer to either accept or deny, in compliance with the requirement `NONREPUDIATION`.

Delegation order constraint

Delegations must be instantiated in chronological order with respect to the transitivity of the privileges they grant. As covered in chapter 5, delegable permission objects store their own parent permission. Therefore, an issuer cannot delegate privileges over a resource in anticipation of receiving their own privileges. A potential solution to this limitation is the following: upon a holder presenting a permission with a missing parent, the `AuthorizationEnforcer` queries its issuer to provide the parent permission. To increase the availability and usability of this process, it is recommended that the issuer's SSI agent be hosted in the cloud.

9.3 Comparison with the state-of-the-art

In section 2.2, we introduced the WAVE and DeFIRED frameworks, state-of-the-art solutions that offer a framework for decentralized authorization with support for transitive delegations. However, the frameworks suffer some drawbacks that affect their security, performance, and usability. In this section, we compare the presented solution with the state of the art and outline the mitigations of their essential drawbacks.

Firstly, the presented solution tackles the key exchange problem. Both state-of-art frameworks assume that users' cryptographic keys have already been exchanged securely. However, this assumption referred to as the key exchange problem, is problematic in practice[12]. Today's systems using public key cryptography typically authenticate keys by introducing a centralized third party, for example a Certificate Authority, responsible for assuring that an entity is truly associated with a given key. However, this method relies

on centralized trust, which violates the core requirement CORE_DECENTRALIZED_TRUST. The presented architecture tackles the key authentication in a decentralized manner by relying on SSI. In SSI, a blockchain acts as a decentralized algorithmic trust anchor. The association between an identity and its public key is proved cryptographically, as explained in further detail in section 3.3.4.

Secondly, the presented architecture offers a significant performance increase. Performance evaluations conducted on a proof-of-concept software (*c.f* performance evaluation section 8.1) demonstrate that the designed solution enables a significant performance increase: For instance, the time to authorize a permission of depth 15 is up to 5 times faster than WAVE and 11 times faster than DeFIRED. These improvements have been achieved by minimizing the amount of data to be encrypted and by using compression. Moreover, the performance of WAVE and DeFIRED is negatively affected by their use of Identity-based encryption (IBE) for identity authentication. The usage of IBE in the presented architecture is unnecessary and avoided, as the system is built on top of an SSI layer that authenticates identities more efficiently (detailed in chapter 3).

Thirdly, WAVE does not allow recipients declining to receive a permission or revoke permissions they hold, which can raise repudiation concerns, as detailed explained by Vrielynck(2021) in his paper on DeFIRED [27]. DeFIRED tackles these issues by introducing proofs of revocation objects. In contrast with DeFIRED, the presented solution does not require proofs. The system allows declining permissions by requiring holders to explicitly consent to receive permissions in their wallet. This feature is reflected in the requirement NONREPUDIATION. As for holder-side permission revocation, since permissions are solely stored client-side in the wallet, holders can simply delete permissions they received to revoke them effectively. This feature is reflected in the requirement HOLDER_REVOCATION. We note that the DeFIRED framework allows users to disprove ever receiving a permission; the covered solution does not offer this feature.

Chapter

10 Conclusion

10.1 Goal of this thesis

In this thesis, we introduced an architecture for decentralized authorization with support for transitive delegation. The solution is designed to answer some use cases, such as building access control, where delegating access rights is functionally necessary, and centralized authorization presents essential security weaknesses. A requirement analysis has been conducted to better understand the problem and build a functionally correct and secure solution. The analysis yielded three core requirements. First, the system must not rely on centralized trust to avoid single points of failure and attacks. Second, users can delegate their privileges over protected resources to other users. Third, permission delegations must be kept confidential to protect sensitive information.

State-of-the-art solutions have been developed recently to tackle these requirements. However, these frameworks suffer from essential drawbacks that affect their security and performance. The current solutions assume that users' cryptographic keys have already been exchanged securely. This assumption is problematic as in practice, secure exchange of public keys typically requires introducing a trusted third party, such as a certificate authority, which is centralized. Moreover, the current solutions suffer from poor time performances in their most essential operations: authorizing and delegating permissions.

This thesis aims to present an alternative approach to tackle the problem of decentralized authorization with support for transitive delegations while improving on the state of the art.

10.2 Approach

The thesis approaches the problem as follows.

First, requirements essential to building a compliant solution are defined. These requirements act as a reference in later stages to validate the solution.

Then, an architecture aiming to tackle the problem is presented; it consists of three main parts: 1) an architecture for decentralized permissions; 2) an extension of decentralized permissions that supports confidential transitive delegation; 3) an architecture for an SSI-based authorization framework to enforce access control to

sensitive resources. Additionally, a proof-of-concept software is developed to concretize the architecture and evaluate the performance.

Next, the compliance of the solution is covered. The proof-of-concept's performance is evaluated, and the architecture is tested to comply with all the previously established requirements.

Finally, the architecture is further analyzed qualitatively. A security analysis that consists of a reduced threat model and an availability analysis is carried out. In addition, the most important limitations of the system are outlined, and the system is compared to the state of the art.

10.3 Contributions

This thesis presents an architecture for decentralized access control with support for transitive delegations. The proposed architecture satisfies the same requirement as the state of the art while offering consequential improvements. Firstly, the presented solution improves the performance drastically. For instance, the time to authorize a permission of depth 15 is up to 5 times faster than the state of the art. These improvements have been achieved by minimizing the amount of data to be encrypted and by using compression. Secondly, the architecture removes the assumption that users' cryptographic keys are pre-shared securely; instead, it tackles the problem by relying on self-sovereign identity (SSI). In SSI, the trust anchor is a blockchain, allowing users to securely prove their association with a public key without relying on centralized third parties, such as a certificate authority.

Furthermore, this thesis offers an SSI-based authorization framework for decentralized access control. State-of-the-art SSI covers authentication only, but not authorization. However, authentication is not an end by itself; instead, it is a means to decide the level of access to grant users. As part of the solution, we present an architecture for SSI-based authorization. This architecture is a generic standalone solution for decentralized access control, decoupled from the transitive delegation problem.

In summary, the main contributions of this thesis are: 1) a drastic performance increase over the state of the art; 2) mitigation of essential drawbacks of state-of-the-art solutions such as assumed pre-shared keys; 3) a generic decentralized authorization framework for the SSI technology, which only covers authentication at the time of writing.

10.4 Future work

The system covered in this thesis aims to provide a foundation for SSI-based authorization and permission delegation. Nevertheless, the system can be further extended for better utility. We consider two suggestions: Firstly, the framework is currently only designed to provide access control to HTTP resources only. The component can be extended to allow easy integration with different protocols. Secondly, the current delegable permission model does not support any restriction on delegations. The model can be extended

to support delegation policies, thus allowing issuers to have more control over how permissions are further delegated.

A Appendix

```
{
    "claims":{
       "iss":"1dide5z8roRZ6a4WMciXne",
       "sub":"2did6hNgAFwF3BeRHzvpo",
       "iat":"1652553886",
       "exp":"1652726686",
       "resource_uri":"https://api.example.com/main-door",
       "operations": ["GET"]
    },
   "auth": {
       "ED25519Signature":"/BP4pHgFLYz2jTCcCFxUtMglvgSitg1Pqyv
       LLAQGPMspaX7z5egFFPR0adowzAzHRG3VPlsZJ29lOrFIo9u/DA=="
    }
}
```

Listing 1: An example of a (non-delegable) permission object.

```
{
    "claims":{
       "owner": "1dide5z8roRZ6a4WMciXne",
       "iss":"2did6hNgAFwF3BeRHzvpo",
       "sub":"3didEwMCFWFbEhZPoVZz8",
       "iat":"1652560000",
       "exp":"1652626686",
       "resource_uri":"https://api.example.com/main-door",
       "operations": ["GET"],
       "hierarchy": "ewogICAgImNsYWltcyewogCAggImlIH0KfQ[...]"
    },
   "auth": {
       "ED25519Signature":"/BP4pHgFLYz2jTCcCFxUtMglvgSitg1Pqyv
       LLAQGPMspaX7z5egFFPR0adowzAzHRG3VPlsZJ29lOrFIo9u/DA=="
    }
}
```

Listing 2: An example of a delegable permission object.

Bibliography

[1] A. Almutairi, M. Sarfraz, S. Basalamah, W. Aref, and A. Ghafoor. A distributed access control architecture for cloud computing. *IEEE Software*, 29(2):36–44, 2012.

[2] M. P. Andersen, S. Kumar, M. AbdelBaky, G. Fierro, J. Kolb, H.-S. Kim, D. E. Culler, and R. A. Popa. Wave: A decentralized authorization framework with transitive delegation. *28th USENIX Security Symposium (USENIX Security 19)*, 2019.

[3] auth0.com. What are refresh tokens and how to use them securely. *auth0.com, retrieved July 10th, 2022 from https://auth0.com/blog/refresh-tokens-what-are-they-and-when-to-use-them/*, 2021.

[4] E. Barker. Recommendation for key management, nist sp-800-57 part 1 rev2. *NIST*, 2020.

[5] E. Barker and A. Roginsky. Transitioning the use of cryptographic algorithms and key lengths, nist sp-800-131a rev2. *NIST*, 2019.

[6] T. Berners-Lee, R. T. Fielding, and L. Masinter. Uniform resource identifier (uri): Generic syntax. STD 66, RFC Editor, January 2005. http://www.rfc-editor.org/rfc/rfc3986.txt.

[7] A. Birgisson, J. G. Politz, Úlfar Erlingsson, A. Taly, M. Vrable, and M. Lentczner. Macaroons: Cookies with contextual caveats for decentralized authorization in the cloud. In *Network and Distributed System Security Symposium*, 2014.

[8] A. Cavoukian. Privacy by design: The 7 foundational principles. *Information and Privacy Commissioner of Ontario*, 2009.

[9] M. Deng, K. Wuyts, R. Scandariato, B. Preneel, and W. Joosen. A privacy threat analysis framework: Supporting the elicitation and fulfillment of privacy requirements. *Requir. Eng.*, 16:3–32, 03 2011.

[10] D. Di Francesco Maesa, P. Mori, and L. Ricci. A blockchain based approach for the definition of auditable access control systems. *Computers & Security*, 84:93–119, 2019.

[11] DIF. *DIDComm Messaging*. Available at https://identity.foundation/didcomm-messaging/spec/.

[12] W. Diffie and M. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, 22(6):644–654, 1976.

[13] R. T. Fielding and J. Reschke. Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content. RFC 7231, June 2014.

[14] M. Gegick and S. Barnum. Least privilege. *Cybersecurity & Infrastructure Agency, retrieved May 14th, 2022 from https://www.cisa.gov/uscert/bsi/articles/knowledge/principles/least-privilege*, 2005.

[15] D. Hardman. Credential revocation. *Hyperledger Indy, retrieved May 14th, 2022 from https://hyperledger-indy.readthedocs.io/projects/hipe/en/latest/text/0011-cred-revocation/README.html*, 2018.

[16] S. Josefsson and I. Liusvaara. Edwards-Curve Digital Signature Algorithm (EdDSA). RFC 8032, Jan. 2017.

[17] H. Kim and E. A. Lee. Authentication and authorization for the internet of things. *IT Professional*, 19(5):27–33, 2017.

[18] Microsoft. Microsoft delivers comprehensive solution to battle rise in consent phishing emails. *Microsoft, retrieved July 11th, 2022 from https://www.microsoft.com/security/blog/2021/07/14/microsoft-delivers-comprehensive-solution-to-battle-rise-in-consent-phishing-emails/*, 2021.

[19] S. Miltchev, J. M. Smith, V. Prevelakis, A. D. Keromytis, and S. Ioannidis. Decentralized access control in distributed file systems. *ACM Comput. Surv.*, 40:10:1–10:30, 2008.

[20] J. Nielsen. Response times: The 3 important limits. *NN Group. Retrieved May 12th, 2022 from https://www.nngroup.com/articles/response-times-3-important-limits/*, 1993.

[21] A. Ouaddah, A. Abou Elkalam, and A. Ait Ouahman. Fairaccess: a new blockchain-based access control framework for the internet of things. *Security and Communication Networks*, 9(18):5943–5964, 2016.

[22] OWASP. *Insecure Design*. Retrieved May 19th, 2022 from https://owasp.org/Top10/A04_2021-Insecure_Design/.

[23] A. Preukschat and D. Reed. *Self-Sovereign Identity, Decentralized digital identity and verifiable credentials*. Manning, 2021.

[24] D. Reed, J. Law, and D. Hardman. The technical foundations of sovrin. *The Technical Foundations of Sovrin*, 2016.

[25] S. Sayeed and H. Marco-Gisbert. Assessing blockchain consensus and security mechanisms against the 51% attack. *Applied sciences*, 9(9):1788, 2019.

[26] P. Thakkar, S. Nathan, and B. Viswanathan. Performance benchmarking and optimizing hyperledger fabric blockchain platform. In *2018 IEEE 26th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, pages 264–276, 2018.

[27] P.-J. Vrielynck, E. H. Beni, K. Jannes, B. Lagaisse, and W. Joosen. Defired: Decentralized authorization with receiver-revocable and refutable delegations. In *Proceedings of the 15th European Workshop on Systems Security*, EuroSec '22, page 57–63, New York, NY, USA, 2022. Association for Computing Machinery.

[28] W3C. *Decentralized Identifiers (DIDs) v1.0*. Available at https://www.w3.org/TR/did-core/.

[29] W3C. *Verifiable Credentials Data Model v1.1*. Available at https://www.w3.org/TR/vc-data-model/.

[30] Weiner. *The Oxford English Dictionary*.

[31] WoSign. Wosign incidents report. *WoSign, retrieved July 10th, 2022 from https://www.wosign.com/report/WoSign_Incidents_Report_09042016.pdf*, 2016.

[32] G. Zyskind, O. Nathan, and A. S. Pentland. Decentralizing privacy: Using blockchain to protect personal data. In *2015 IEEE Security and Privacy Workshops*, pages 180–184, 2015.