KU LEUVEN

FACULTEIT INGENIEURSWETENSCHAPPEN

Secure and protected data aggregation in the decentralized web

Jesse Geens

Thesis voorgedragen tot het behalen van de graad van Master of Science in de ingenieurswetenschappen: computerwetenschappen, hoofdoptie Gedistribueerde systemen

Promotoren:

Prof. dr. ir. Wouter Joosen
Dr. Bert Lagaisse
Prof. dr. Vincent Naessens

Assessoren:

Prof. dr. Bart Jacobs
Dr. Emad Heydari Beni

Begeleiders:

Dr. Emad Heydari Beni
Ir. Kristof Jannes

Academiejaar 2021 – 2022

Preface

> *"I found myself answering the same questions asked frequently of me by different people. It would be so much easier if everyone could just read my database."*
>
> *– Sir Tim Berners-Lee*

Solid is an amazing technology that has the potential to change the web as we know it. The last year, I have witnessed amazing progress being made in this domain, and I am truly honored that I had the opportunity to be a part of it. It has been quite the roller coaster, combining my dissertation research with a semester spent abroad on an Erasmus exchange, but it was well worth the effort. There are an endless amount of people I want to thank for supporting me through this period, especially my friends and family.

First and foremost, my most sincere gratitude goes to my girlfriend Elise, who had to endure endless streams of complaints when I was stuck somewhere, and screams of joy when I made some progress. Without her support, this text would not be the same.

Secondly, I owe an immense amount of gratitude to my supervisors and assistant-supervisors. In particular, I want to thank Bert Lagaisse for allowing me to combine my dissertation while studying in Hungary, which was not a trivial task. Your feedback and guidance taught me a lot and greatly improved the quality of my work. I also want to thank Emad and Kristof for the great advice and encouragement every week. Your advice has inspired me many times and your kind words have always encouraged me. Finally, I also want to explicitly thank prof. Vincent Naessens, prof. Wouter Joosen and Dimitri Van Landuyt for their valuable feedback and suggestions.

Thirdly, I want to thank my family for supporting me not only through this dissertation, but throughout my whole study career. I owe a large part of my successes to their support. I especially want to thank my father and grandfather for inspiring me to study a technical domain, and for teaching me to always think critically.

Lastly, I want to thank my father and my friends Benjamin, Aram and Tobias for proofreading and improving this text.

Jesse Geens

Contents

Abstract

In recent years, the internet has become ever more centralized (Sandvine, 2021). This has many negative side effects, such as decreased competition and a lack of access to personal data (Relihan, 2018; Jarsulic, 2020). Solid (Capadisli et al., 2021) is a new draft W3C specification that aims to combat this by introducing pods. Pods are decentralized datastores where user data is kept for many different applications. In this manner, the user retains control over their personal data, and the same data can be used across different services. These pods provide a number of possibilities, but come with technological challenges such as secure and protected data aggregation across different pods.

Such aggregations impose privacy risks, as well as scalability issues. Ordinarily, a data aggregator would fetch complete resources and then anonymize the aggregation, which is privacy-invasive. Furthermore, aggregators may need to combine data from different services, which brings along authorization challenges. This dissertation presents a server-level middleware that aims to partially resolve these issues by presenting two novel technologies: privacy filters, and a new token mechanism that supports decentralized delegation.

Privacy filters is a technology that rewrites resources upon requests, based on the user's privacy settings and contextual parameters such as the requesting application and the resource's data type. This enables more granularity in the privacy of shared resources. To subsequently attain a token mechanism that supports decentralized delegation, this dissertation investigates the use of macaroons as a token mechanism in Solid. Macaroons not only provide decentralized token delegation, but are also much more efficient to generate and verify. Moreover, they allow third-party attestations, which is a useful property for realizing pods belonging to a group of agents.

The presented middleware has been evaluated on the basis of three use cases. Performance experiments have demonstrated that the overhead of privacy filters is tolerable for smaller resources. Rewriting resources of around 300KB using three transformations causes an overhead of around 50% in the request duration. Larger resources (around 3MB) have overheads that become a five-fold of the original request duration, but this issue can be lessened by making use of caching and precomputation. Further experiments on the generation and verification of macaroons have illustrated that macaroons offer large performance benefits. The throughput of generating and verifying macaroons is seven and eleven times higher respectively, compared to DPoP using the ES256 algorithm. Finally, there are also theoretical gains for decentralized delegation in a reduced interaction cost for realizing token delegation.

Samenvatting

De afgelopen jaren is het internet steeds meer gecentraliseerd geworden (Sandvine, 2021). Dit heeft vele negatieve gevolgen, waaronder verminderde concurrentie en een gebrek aan toegang tot persoonlijke gegevens (Relihan, 2018; Jarsulic, 2020). Solid (Capadisli et al., 2021) is een nieuwe, voorlopige W3C specificatie die dit probleem tracht op te lossen door het introduceren van datakluizen. Datakluizen zijn gedecentraliseerde opslagplaatsen voor data waar gegevens voor verschillende toepassingen worden bijgehouden. Op deze manier bewaart de gebruiker controle over zijn persoonlijke gegevens en kan dezelfde data door verschillende diensten gebruikt worden. Deze datakluizen brengen een hoop mogelijkheden met zich mee, maar evenzeer technologische uitdagingen.

Eén van zo'n uitdagingen is het mogelijk maken van veilige en beschermde gegevensaggregatie over verschillende datakluizen heen. Dergelijke aggregaties brengen privacyrisico's met zich mee, net als schaalbaarheidsproblemen. Typisch vraagt zo'n aggregator volledige bronnen op om vervolgens de resulterende aggregatie te anonimiseren, wat privacy-onvriendelijk is. Bovendien moeten dergelijke aggregators mogelijk gegevens van verscheidene diensten combineren, hetgeen autorisatie-uitdagingen met zich meebrengt. Deze thesis introduceert een middleware op het niveau van de server, die tracht deze problemen gedeeltelijk op te lossen dankzij het introduceren van twee nieuwe technologieën. Deze technologieën zijn privacy filters en een nieuw mechanisme voor toegangstokens dat gedecentraliseerde tokenafvaardiging ondersteunt.

Privacy filters is een technologie die bronnen herschrijft bij een verzoek, gebaseerd op de gebruiker zijn privacy-instellingen en contextuele parameters. Deze parameters omvatten welke toepassing het verzoek verzonden heeft en wat het datatype is van de verzochte bron. Dit laat toe om meer granulariteit te bereiken in de privacy van gedeelde bronnen. Om vervolgens een tokenmechanisme te bekomen dat gedecentraliseerde tokenafvaardiging ondersteunt, onderzoekt deze thesis het gebruik van macaroons als nieuw tokenmechanisme binnen Solid. Macaroons ondersteunen niet alleen gedecentraliseerde tokenafvaardiging, maar zijn ook efficiënter om te genereren en verifiëren. Bovendien laten ze attestaties van derde partijen toe (*third-party attestations*), wat een nuttige eigenschap is om datakluizen te bekomen die gedeeld worden door een groep gebruikers.

De middleware die deze thesis voorstelt werd geëvalueerd op grond van drie gebruiksscenario's. Voor privacy filters hebben prestatie-experimenten aangetoond dat de overhead van de middleware toelaatbaar is voor kleinere bronnen. Het

herschrijven van bronnen tot 300KB, gebruikmakende van drie transformaties, leidt tot een overhead van ongeveer 50%. Grotere bronnen (rond de 3MB) hebben echter een overhead die een vijfvoud wordt van de oorspronkelijke duur van het verzoek, maar dit probleem kan verminderd worden door gebruik te maken van caches of met behulp van voorberekeningen. Prestatie-experimenten die werden uitgevoerd op het genereren en verifiëren van macaroons hebben aangetoond dat hier sterke prestatievoordelen aan vasthangen. De doorvoer voor het genereren en verifiëren van macaroons is respectievelijk zeven en elf maal groter dan het DPoP systeem wanneer dit gebruik maakt van het ES256 algoritme. Ten slotte zijn er ook theoretische winsten voor gedecentraliseerde tokenafvaardiging, gezien dit mechanisme een verminderde interactiekost nodig heeft voor het afvaardigen van een toegangstoken in vergelijking met een gecentraliseerd mechanisme zoals DPoP.

List of figures and tables

List of figures

List of tables

List of code fragments

List of repositories

pepsa-component	https://github.com/jessegeens/pepsa-component
groupvault-demo	https://github.com/jessegeens/groupvault-demo
data-generator	https://github.com/jessegeens/data-generator
thesis-experiments	https://github.com/jessegeens/thesis-experiments

List of abbreviations

ABE Attribute-Based Encryption. vii, 25, 33, 34, 67

ACL Access Control List. ix, 11, 12, 22, 67

CP-ABE Ciphertext-Policy Attribute-Based Encryption. viii, 33, 34, 67

CSS Community Solid Server. viii, ix, 3, 11, 17, 36, 38, 40, 44, 50, 52–54, 62, 63, 65, 77

DPoP Demonstrated Proof Of Possesion. iv, vi, ix, 10, 38, 43, 51, 57–60, 63, 65, 66

GDPR General Data Protection Regulation. 1, 26, 70

GVS Group Vault Server. vii, 43–45

HMAC Hash-based Message Authentication Codes. 14, 15, 22

IBE Identity-Based Encryption. 33

IV Initialization Vector. 29

JWT JSON Web Token. 10, 48

KP-ABE Key-Policy Attribute-Based Encryption. viii, 33, 34

LDP Linked Data Platform. 12

MPC Secure Multi-Party Computation. 25, 30–32, 66

OP OpenID Provider. 9, 10

PETs Privacy-enhancing Technologies. 2, 3, 17, 20, 22, 25, 26, 37, 62, 65, 70

PII Personally Identifiable Information. 26, 27, 40, 62

PKCE Proof Key For Code Exchange. 9

RDF Resource Description Framework. vii, 6, 7, 11, 12, 30, 50

Chapter 1

Introduction

This dissertation presents a middleware for secure and protected data aggregation. Section 1.1 discusses the importance of such a middleware. Subsequently, a concrete problem statement is discussed in section 1.2. Afterwards, section 1.3 lists the contributions this dissertation makes. To conclude, section 1.4 gives an overview of the structure of the remaining parts of the text.

1.1 Motivation

In recent years, the internet has become more and more centralized (Sandvine, 2021). This has many negative side effects: powerful companies have control over vast amounts of data, providing new market insights or personalization options, which in turn lead to more market power (Relihan, 2018; Jarsulic, 2020). Lacking data on such large scales, smaller enterprises do not have access to these capabilities. This stifles competition as it makes it harder for these smaller companies to find a position in the market, limiting innovation.

Furthermore, the lack of access to personal data has also manifested itself at the user's side. It is often hard or impossible to share data between competing platforms, and users lack insights into what data is held by what companies. The introduction of the GDPR (European Commission, 2016) aimed to combat this by giving users the right to look at their personal data, but this still requires a lot of effort from the user as this data is splintered across many different services and platforms. This contrasts with an ever-increasing focus on data privacy from users (McKinsey, 2020).

To solve these data ownership problems, the Solid (Capadisli et al., 2021) specification was introduced. Solid is a very new technology that is receiving a lot of attention and large investments over the last few months. These investments come from multiple governments (De Tijd; De Tijd, 2021a; 2021b) as well as industry (TechCrunch, 2021). This will hopefully lead to an influx of developers and users into the ecosystem, which provides many opportunities from a data perspective. These data opportunities offer the most value when combined across pods, which highlights the need for secure data aggregation mechanisms.

1.2 Problem statement: data aggregation in the decentralized web

Solid is a web specification with the aim of decentralizing the web again. The specification introduces a number of concepts to achieve this. Every user (a person, automated agent, group, ...) is given his own WebID: an identity on the web. Furthermore, WebIDs often also have one or more coupled *pods*. Pods are personal online data vaults, and form a standardized way to store a user's data. This allows the user to re-use the same data across different services, and to fine-tune which applications have access to which data.

Solid is a recent specification, and as such there remain many open challenges. One of these challenges is secure aggregation of data across pods. Pods often hold valuable information that, when combined with information from other pods, can lead to useful insights for users, companies and other stakeholders. However, it is technically challenging to aggregate such data from different pods. Many aspects come in to play here. Firstly, data aggregation across Solid pods can lead to exposure of sensitive data. Hence, a mechanism to release data in a more fine-grained and controlled manner is necessary. Secondly, aggregating data across many pods implies that many authentication requests will have to be made. As such, a scalable, efficient and flexible authentication and authorization mechanism must be devised. Preferably, such a mechanism would also work in a decentralized manner, since complex aggregations can depend on many services.

The aim of this dissertation is thus to come up with possible solutions for secure and protected data aggregation in the Solid ecosystem. To concisely summarize this section, we formulate the following high-level research question.

RQ: Can we design a scalable middleware solution aimed at facilitating data aggregation in a secure and protected manner in a setting where data is distributed across various parties?

1.3 Contributions

This dissertation presents the Middleware for Aggregation Security in Solid (MASS), a conceptual server-side middleware for facilitating secure and protected data aggregation in Solid. Concretely, this dissertation makes the following contributions to the research domain of Solid:

1. An overview of literature in the domain of privacy-enhancing technologies (PETs), accompanied by a discussion of each technology and its usability within the context of Solid.

2. Privacy filters, a novel, customizable and seamless approach to limit the information leakage of data exposed through Solid pods. This mechanism is implemented in a prototype extension of the Community Solid Server[1] and evaluated.

3. A mechanism for realizing decentralized delegation of access tokens in Solid, which also enables more efficient generation and verification of tokens as well as tokens with third-party attestations. This mechanism is also accompanied by a prototype and benchmarked.

1.4 Dissertation outline

This section describes the layout of the remaining parts of this text. To better understand the context in which this dissertation is set, the next chapter introduces some necessary background information. It covers all the topics on which this dissertation is built, such as an in-depth explanation of the Solid protocol, as well as an introduction to macaroons, a novel type of access token.

Chapter 3 starts by introducing a number of use cases which highlight the need for an aggregation middleware. Furthermore, a generic architecture for data aggregation in Solid is introduced. This architecture aids in researching what the concrete bottlenecks are and is thus used to develop some concrete goals for the middleware. Based on the presented use cases and goals, a number of requirements and a relevant attacker model are proposed. The final section of this chapter then gives a system overview of the developed middleware, explaining on a high level its two main components.

The first building block necessary for such a middleware system concerns the privacy aspect. Therefore, chapter 4 studies a number of widely used privacy-enhancing technologies (PETs) and discusses their relevance in the context of Solid. Some of these PETs will be used in the developed solution, while devising solutions using the remaining applicable PETs is considered future work (discussed in chapter 8).

Chapter 5 subsequently introduces privacy filters, one of the components of the solution, in detail. This component focuses on rewriting requests to render data more private when it is requested by an aggregator or application. The chapter also discusses the concrete implementation of this component. The main findings of section 4 will be used to develop this solution.

Having developed a possible solution to the privacy problem, the next challenge is *(decentralized) delegation of access tokens*. This is discussed in chapter 6. Aggregators can be quite complex, consisting of multiple workers, and they need long-lived access to the data in a safe manner. Macaroons are studied as a possible remedy, and their advantages and drawbacks are discussed. This is done by first looking at how macaroons can enable group vaults. Subsequently, macaroons are compared to Solid's current token system in the context of delegating access tokens.

[1] https://github.com/CommunitySolidServer/CommunitySolidServer

To better understand the applicability of this dissertation's contributions in terms of performance and feasibility, chapter 7 validates and evaluates the results. This is done by discussing theoretical shortcomings, as well as performing some experiments to measure the performance of developed prototypes.

Finally, chapter 8 summarizes the main findings of this dissertation. The dissertation is then concluded by discussing potential areas of future work, which may hopefully inspire the reader to perform further research in this domain.

Chapter 2

Background

This dissertation investigates a middleware for realizing secure and protected aggregation of data exposed through the Solid protocol. Section 2.1 introduces the basic concepts of Solid, while section 2.2 introduces some necessary background knowledge on macaroons, a novel type of access token.

2.1 Solid

This section gives a detailed overview of the Solid protocol. The Solid protocol stems from earlier research into the Semantic Web, which will be introduced in section 2.1.1. Section 2.1.2 will then give an overview of the protocol itself, while the sections following this will each explain the authentication, authorization and linked data aspects of the protocol respectively.

2.1.1 The Semantic Web

> "The Semantic Web is not a separate Web but an extension of the current one, in which information is given well-defined meaning, better enabling computers and people to work in cooperation."
>
> (Sir Tim Berners-Lee)

In 2001, Sir Tim Berners-Lee — the inventor of the world-wide web — published an article in *Scientific American* entitled 'The semantic Web' (Berners-Lee et al., 2001). It envisioned a future where the Web was not just a collection of pages with meaningless links to each other, but a smart Web, where objects and relations have well-defined meanings.

Today, most resources accessible on the Web are meant for people to read, understand and process, but they are not adapted to automated processing by software agents. Computers can read information and format it correctly, but they lack a unified way to interpret the information: there is no way to process the *semantics*. The Semantic Web is a proposal to combat this problem. The Semantic Web tries to solve this problem by extending the current Web with new, compatible

standards and protocols to attach computer-readable meaning to information available on the Web.

Currently, markup languages such as XML enable users to arbitrarily structure documents the way they want it (e.g. using a tag called `<book>` and a tag called `<isbn>`). They then write software that parses this by telling the software that `<book>` means "book" and `<isbn>` means "isbn". However, other users may use different terms (e.g. `<isbn13>` instead of `<isbn>`), making information not universally parsable. The Semantic Web introduces the RDF, which encodes structure and information in sets of triples: a subject, a predicate and an object. These triples then create a knowledge graph called an RDF graph, where predicates relate subjects to objects. The objects in one triple can then be the subjects in another one. The figure below illustrates an example, where Tokyo is the subject of a linked data triple, and two predicates relate it to its area and the country it resides in. The object, Japan, may then be the subject of other RDF graphs, relating it for example to its population.

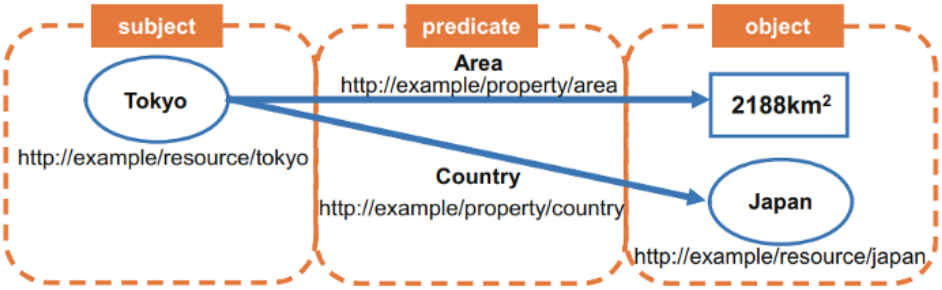


Figure 2.1: RDF representation of the city Tokyo, with predicates relating it to its area and its country. Source: Nakatani et al. (2018)

Subjects, predicates and objects are all identified by a URI, making them universal, and allowing anyone to define new predicates[1]. In this manner, webs of information are formed between related objects using definitions that can be found and understood by everyone.

This does not completely solve the problem of information interpretation, however. It is possible that multiple definitions exist for essentially the same object. To resolve this, the Semantic Web introduces a third component: ontologies. Ontologies are documents that formally define the relations among terms (Berners-Lee et al., 2001). A widely-used ontology is called *foaf*[2] (friend-of-a-friend): it is used to describe relationships between people.

RDF, however, is only something conceptual (a collection of subject, predicate and object). In order to actually use it, RDF must be stored in documents. This is possible in multiple ways, i.e., multiple representation formats exist. The most popular ones are Turtle (Prud'hommeaux and Carothers, 2014) and JSON-LD (Champin et al., 2020). Turtle has the advantage that it allows defining prefixes at the top of the file,

[1] Unless they are literals, such as the area in the example

[2] http://xmlns.com/foaf/spec/

which are aliases for the long URIs needed to relate the subjects/predicates/objects. To talk about the foaf ontology, the following must be defined at the top of the file:

```
@prefix foaf <http://xlmns.com/foaf/0.1/>
```

Then, `foaf` can be used in the rest of the Turtle file, instead of `<http://xlmns.com/foaf/0.1/>`. This makes it very human-readable compared to other formats. Additional existing formats are RDF/XML and N-Triples, but they are less common.

2.1.2 The Solid Protocol

> "Solid is a proposed set of conventions and tools for building *decentralized applications* based on Linked Data principles."
>
> (solidproject.org)

Solid (Capadisli et al., 2021) is a proposed W3C specification that wishes to decentralize the web by giving people control over their data. Solid aims to realize this by giving people an online datastore called a *pod* (personal online datastore). Users can then log in to applications using an authentication mechanism provided by Solid, and the application can in turn access data stored in the user's pod.

By storing all the data inside the user's pod instead of on the application server, the user retains complete control over their data. They can flexibly choose what data to share with what applications. Furthermore, storing the data in a pod reduces vendor lock-in: when the user wishes to switch from one service to another, he can simply give the new application access to the data he already possesses.

In Solid, there are two types of resources: linked and non-linked data resources. Non-linked data resources are the usual kinds of data that we access on the web right now (binary, text, images, ...), while linked data follows the RDF specifications by using a content representation such as Turtle. These linked-data resources thus follow the principles of the Semantic Web.

2.1.3 Authentication

Needless to say, some of the data stored in a user's pod should be protected, such that not every agent on the web can read it. To achieve this goal, Solid needs an authentication mechanism. Formally, authenticating a user is defined as the act of verifying a user's claimed identity (Capadisli et al., 2021). As such, two components are needed: a way to form identities, and a way to verify that a user possesses an identity.

Identities

Identities in solid (both for users and other agents) follow the WebID standard (Sambra et al., 2014), where URIs act as universal identifiers. An example WebID is:

`https://jessegeens.solidcommunity.net/profile/card#me`

These WebID URIs can identify several things: users, software agents, or other things (even if they do not exist on the web). The WebID URI references a WebID Profile Document. This document contains information about the agent who is the referent of the WebID URI. However, an important distinction must be made. When the *#me* is included in the URI (or more in general, the hashtag), this URI refers to an agent. When the hashtag is omitted, the URI refers to the document describing this agent.

```
@prefix foaf: <http://xmlns.com/foaf/0.1/>.
@prefix solid: <http://www.w3.org/ns/solid/terms#>.

<>
    a foaf:PersonalProfileDocument;
    foaf:maker <http://localhost/jesse/profile/card#me>;
    foaf:primaryTopic <http://localhost/jesse/profile/card#me>.

<http://localhost/jesse/profile/card#me>

    solid:oidcIssuer <http://localhost/>;
    a foaf:Person.
```

Listing 2.1: Example WebID profile document

Verifying identities

While the WebID standard provides identities, it does not yet verify them. The actual authentication (or, verifying that the agent actually controls the WebID he claims to control) in Solid uses the WebID-OIDC protocol. This protocol draws inspiration from OAuth2 (Hardt, 2012) and from OpenID Connect (Sakimura et al., 2014). The workflow below, drawn from the WebID-OIDC repository, explains the basic mechanism of the protocol from the point-of-view of the user:

1. **Initial Request**: Alice (unauthenticated) makes a request to bob.example, receives a `HTTP 401 Unauthorized` response, and is presented with a 'Sign In With...' screen.

2. **Provider Selection**: She selects her WebID service provider by clicking on a logo, typing in a URI (for example, `alice.solidtest.space`), or entering her email.

3. **Local Authentication**: Alice gets redirected towards her service provider's own Sign In page, thus requesting `https://alice.solidtest.space/signin`, and authenticates using her preferred method (password, WebID-TLS certificate, FIDO 2 / WebAuthn device, etc).

4. **User Consent**: (Optional) She's presented with a user consent screen, along the lines of "Do you wish to sign in to `bob.example`?".

5. **Authentication Response**: She then gets redirected back towards `https://bob.example/resource1` (the resource she was originally trying to request). The server, `bob.example`, also receives a signed ID Token from `alice.solidtest.space` that was returned with the response in point 3, attesting that she has signed in.

6. **Deriving a WebID URI**: `bob.example` (the server controlling the resource) validates the ID Token, and extracts Alice's WebID URI from inside it. She is now signed in to `bob.example` as user `https://alice.solidtest.space/#i`.

7. **WebID Provider Confirmation**: `bob.example` confirms that `solidtest.space` is indeed Alice's authorized OIDC provider (by matching the provider URI from the *iss claim* with Alice's WebID).

(github.com/solid/webid-oidc-spec)

Solid OpenID Connect

While the previous section gave a high level overview of the authentication from a user's point of view, this section takes a look at the technical details of Solid's version of OpenID Connect. OpenID Connect is a layer on top of OAuth2 that provides identities, and the protocol makes use of the PKCE specification (Sakimura et al., 2015).

In Solid, the authentication server (called the OP) can differ from the server that offers the resources (called the Resource Server). When the user enters the URI of his WebID document, the application can fetch the user's WebID Profile and read it (see listing 2.1). The application can then determine the URL of the OpenID Provider (OP) (located under `solid:oidcIssuer`) and retrieve the configuration of the OP. This is referenced by a fixed location, `.well-known/openid-configuration`. In the next step, the client makes an authorization request. This authorization request contains a code challenge and a code verifier (as defined in the PKCE specification). The verifier is created by generating a cryptographically random string, and the code challenge is created by applying an algorithm such as SHA256 to the code verifier. The verifier is then stored by the application. The code challenge, on the other hand, is not stored but forwarded to the OP in an authorization request.

In Solid, every agent has a WebID, and so too does the application. When the OP receives the authorization request, it fetches the client's WebID. The authorization request contains a redirect URL, and the OP makes sure that the redirect URL is also listed in the client's WebID, to prevent malicious redirects. The user then logs in at the OP, after which it generates a code which is sent back using the redirect URL. Finally, the client then generates a DPoP client key pair and header (see next section). This key pair consists of a public and private key, where the private key is used to sign a JWT (Jones et al., 2015).

All the steps for performing an authentication request have then been fulfilled, and a request can be made to the token endpoint specified in the openid configuration file (Morgan et al., 2022).

DPoP

Demonstrated Proof of Possesion (DPoP) (Fett et al., 2021) is a new technology that improves upon bearer tokens (see section 2.2.1) from a security point-of-view, by preventing replay attacks. It utilizes a proof-of-possession mechanism where a DPoP header token proves that the sender (i.e., the client application) possesses a private key, without revealing this key. It is the main mechanism used for authentication in Solid. In the mechanism, the first step is that the client generates a public-private key pair. To then construct a DPoP header token, the client creates a JWT and signs it using the generated key. This token includes the public key in its header, and a number of fields in its body (see listing 2.2).

```
{
    "htu": "https://jessegeens.solidcommunity.net/data/private.ttl",
    "htm": "POST",
    "jti": "cfd9287d-2900-5c44-c2cb-dc2cd6eabd3b",
    "iat": 1645966951
}
```

Listing 2.2: DPoP token body

The fields in this body provide a number of protections. The `htu` field limits the token to usage on a single resource. The `htm` field limits the token to a specific HTTP method. These two fields prevent malicious pods from using the token to relay it to another pod and fetch another resource (essentially, a protection against man-in-the-middle attacks). The `iat` field contains a unix timestamp of when the token was generated. Finally, the `jti` field contains an identifier for the token, used to prevent replay attacks.

2.1.4 Authorization

Authorization within the Solid project builds on the Web Access Control (WAC) standard (Capadisli, 2021). The WAC standard provides a method to define authorization conditions for resources in a pod using an ACL. Every resource is coupled with an ACL resource - either directly, or by inheriting it from a parent container (see next section). These ACL resources are using the ACL ontology, usually in a turtle file[3]. Authorization conditions consist of three elements: access objects (the associated resource), access modes (*read*, *write*, *append* or *control*) and access subjects (the agents performing the request).

The access objects in an ACL resource are the resources the ACL resource refers to. This can be both a normal resource (LDP-RS or LDP-NR, see 2.1.5) as well as a container. It is not mandatory for resources to have an associated ACL resource, as these can be inherited. This mechanism ensures that there is no need to create duplicate ACL resources for similar resources, as well as ensuring proper protection for newly created resources. When a resource is accessed, the server will first look for a directly associated ACL resource. When none is found, the server will walk up the container tree (starting from the current container the resource is located in, then the parents, up until the root container). Once an ACL resource applying to one of the parent containers is found, this is applied to the requested resource.

Possible access modes, then, are either *read*, *write*, *append* or *control*. Reading and writing are the usual operations. Append is a subset of writing: data can be added to the resource, but not deleted. This is useful for, for example, logging applications, ensuring that logs cannot be removed. Since appending is a subset of writing, the writing authorization implies the appending authorization (no resource can allow writing but disallow appending). Finally, control means being able to modify the ACL of the resource and generally implies ownership of the resource.

Lastly, access subjects are those agents who request a certain operation on the resource. Generally, these can be divided into four categories. The first one is *every agent*, i.e., the resource is publicly accessible. A second option is authenticated agents only (with no restrictions on who these agents are), which may be useful for auditing purposes. Thirdly, resource access can be restricted to agents with a specific WebID. Finally, this can also be extended to groups of agents (where the group has a single WebID). An example ACL resource, taken from the Community Solid Server, is presented below.

[3]Other RDF representations are also allowed, but support for turtle files is mandatory

```
# Root ACL resource for the agent account
@prefix acl: <http://www.w3.org/ns/auth/acl#>.
@prefix foaf: <http://xmlns.com/foaf/0.1/>.

# The homepage is readable by the public
<#public>
    a acl:Authorization;
    acl:agentClass foaf:Agent;
    acl:accessTo <./>;
    acl:mode acl:Read.

# Only the owner has (full) access to every resource in their pod.
<#owner>
    a acl:Authorization;
    acl:agent <https://pod.geens.cloud/podjesse/profile/card#me>;
    # Set access to root storage folder
    acl:accessTo <./>;
    # All resources will inherit this authorization, by default
    acl:default <./>;
    # The owner has all of the access modes allowed
    acl:mode
        acl:Read, acl:Write, acl:Control.
```

Listing 2.3: Example ACL Resource

2.1.5 Linked Data Platform and Containers

Solid mainly relies on the LDP protocol (Speicher et al., 2015) for resource management. The LDP protocol uses HTTP for accessing and modifying resources on a server. LDP Resources, or LDPRs, are HTTP resources that conform to a number of conventions. There are multiple types of resources:

1. RDF Sources, also called LDP-RSs (LDP RDF Source)
2. Non-RDF Sources, such as images or binary data, which are called LDP-NRs
3. Containers, a concept for bundling related resources together. These are also called LDPCs (LDP Container).

To support the access and modification of these resources, LDP servers must support a number of HTTP methods on the resources. The `GET` method is mandatory and returns the requested resource, if certain conditions are met (e.g., does the agent have the correct authorizations to access the resource). The `POST` and `PUT` methods are optional, and allow agents to create new resources or modify existing ones. Similarly, the `DELETE` method is optional and allows agents to delete resources. The `HEAD` and

`OPTIONS` methods are mandatory to be implemented by servers and are similar to the methods defined in the HTTP/1.1 protocol. Below is an example of a request for a basic container and the accompanying reply (from Mihindukulasooriya and Menday, 2015):

```
GET /alice/ HTTP/1.1
Host: example.org
Accept: text/turtle

HTTP/1.1 200 OK
Content-Type: text/turtle; charset=UTF-8
Link: <http://www.w3.org/ns/ldp#BasicContainer>; rel="type",
      <http://www.w3.org/ns/ldp#Resource>; rel="type"
Allow: OPTIONS,HEAD,GET,POST,PUT,PATCH
Accept-Post: text/turtle, application/ld+json, image/bmp, image/jpeg
Accept-Patch: text/ldpatch
Content-Length: 250
ETag: W/'123456789'

@prefix dcterms: <http://purl.org/dc/terms/>.
@prefix ldp: <http://www.w3.org/ns/ldp#>.
<http://example.org/alice/> a ldp:Container, ldp:BasicContainer;
  dcterms:title 'Alice`s data storage on the Web' .
```

Figure 2.2: HTTP request for publicly exposed basic container

2.2 Macaroons

Macaroons are a type of authorization credential developed for decentralized distributed systems. This section gives some background on why macaroons were developed, followed by the technical details of how they work. As will become clear, macaroons offer many advantages in an inherently decentralized system such as Solid. Macaroons will form an important part of the solutions proposed by this dissertation, and thus warrant a thorough introduction.

2.2.1 Introduction

Macaroons were developed at Google in 2014 by Birgisson et al.. They were created to solve a common authorization problem, for which no sufficient solution existed yet at the time. In 2014, the Cloud was an emerging technology which began to see widespread adoption. In the Cloud, a microservices architecture is often used, where a piece of software consists of multiple, smaller services. This architecture style has many advantages, such as improved scalability and testability, but it also comes with many challenges regarding authentication and authorization.

In particular, services operated by different owners and stakeholders are often used (for example, using third-party APIs), which necessitates sharing data across multiple trust domains. However, existing solutions are often not sufficient for solving this problem. A very commonly used authentication mechanism is bearer tokens. Bearer tokens act as a proof that the bearer of the token should have access to a particular resource. However, this implies that anyone able to steal the token can also get access to the resource. Other mechanisms, such as DPoP (see section 2.1.3), rely on public-key cryptography, which comes with a high computational cost.

To tackle these challenges, macaroons were introduced. Macaroons are similar to bearer tokens, but with some key differences. Firstly, macaroons introduce a concept called *caveats*. Caveats are requirements which the request, to which the macaroon is attached, must fulfill in order for the macaroon to be valid. These caveats are embedded in macaroons and cannot be removed thanks to the use of HMAC signatures. Examples of caveats are:

1. `user = jesse`
2. `ip = 193.190.253.145`
3. `time < 1649081943`
4. `@google.com logged-in as jesse.geens@gmail.com`

As can be seen from the examples, there are two types of caveats. The first type of caveat is called a *first-party caveat*. These are caveats that can be checked locally by the server receiving the request. In the example, caveats 1, 2 and 3 are first-party caveats. The second type of caveat is a *third-party caveat*. These are caveats which must be validated by a third-party server. This is done by returning a so-called *discharge macaroon*. The discharge macaroon is handed out by the third-party server and proves to the first-party server that the embedded caveat is indeed fulfilled. In the example, the fourth caveat is a third-party caveat. These discharge macaroons can in turn also contain additional caveats.

An additional important property of macaroons is called *decentralized delegation*. Often, a service also depends on another service to perform some of its work. Traditionally, a service has a token that is valid only for that service, and cannot safely transfer its token to another service. Solutions exist, such as the OAuth On-Behalf-Of flow, but these require a lot of communication between the service and the token endpoint. With decentralized delegation, services can pass their macaroon on to other services, with at least the same caveats. A service can also further attenuate the macaroon before handing it off. Finally, macaroons can also be sealed, which makes them unable to be modified further.

All of this is made possible by the structure of macaroons, which is explained in the next section. Macaroons use Hash-based Message Authentication Codes (HMAC), which are computationally very efficient.

2.2.2 Structure of macaroons

Macaroons work in a fashion somewhat similar to blockchains. Macaroons are made up of a chain of messages, the caveats. These messages are used to form a chain of HMAC values, of which only the terminal value will be stored in the macaroon. This final value acts as the signature, and can be recalculated by the macaroon-issuing service to verify its authenticity.

The first message of a macaroon is a (public) *key identifier*. This identifier maps to a secret root key. This root key is used to generate the first signature such that:

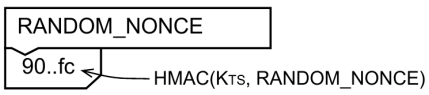
$$sig_1 = \text{HMAC}(K_r, \text{msg-id})$$

with K_r the secret root key, and msg-id the first message (usually the key identifier). The next signature is then calculated based on the first caveat C_1 as:

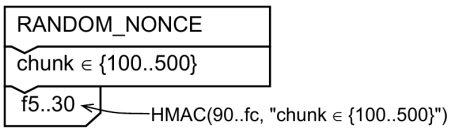
$$sig_2 = \text{HMAC}(sig_1, C_1)$$

which is repeated until all caveats have been incorporated. By keeping the root key secret, the signature of a macaroon cannot be forged. However, the identifier of the root key is needed to ensure that the target service knows which root key to use for signature verification. This mechanism is illustrated in figure 2.3.

Since the final signature is visible, macaroons can be further attenuated by the service that makes requests: a new caveat C_k can be added by generating a new signature $sig_{k+1} = \text{HMAC}(sig_k, C_k)$.



(a) sig_1, based on a random nonce (identifier) and the root key



(b) sig_2, based on a caveat and sig_1

Figure 2.3: Illustration of the macaroon signature composition. Source: Birgisson et al. (2014)

Third-party caveats are more difficult to implement. As explained in the previous section, third-party caveats are caveats that must be *discharged* by a third-party server. This is realized using *holder-of-key proofs*. In order to fulfill a third-party caveat, a service must also include a discharge macaroon that acts as a proof that the caveat is fulfilled. As these discharge macaroons are also macaroons, they also have a key identifier. However, in the case of discharge macaroons, this key identifier's purpose is not only to encode which root key was used - it also embeds the predicate that must be verified by the third party.

The way this is realized is by having the first-party server create a new root key for the discharge macaroon. This root key is encrypted with the third-party server's public key. When the requesting application receives the encrypted root key, it can pass it on to the third party server, which generates a new discharge macaroon by

decrypting the root key and verifying the embedded requirements. The third-party caveat in the original macaroon contains a key identifier pointing to this root key, as well as the location of the third party server and an encrypted copy of the root key. This copy is encrypted with the current signature of the macaroon, so that it can be decrypted when the macaroon is being validated. This is illustrated graphically in figure 2.4.

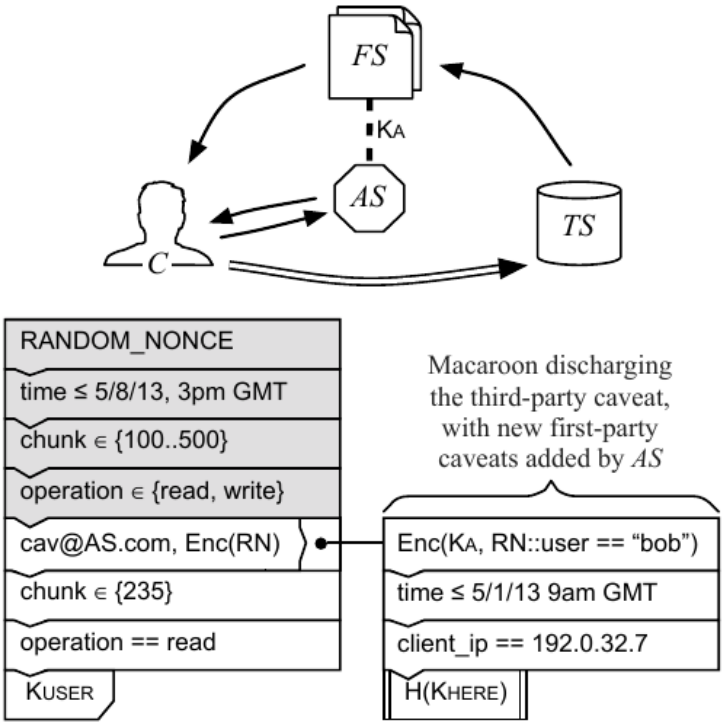


Figure 2.4: Illustration of third-party caveats and the linked discharge macaroons. The macaroon is issued by TS (Token Service, grey), extended with additional caveats by FS (File Service, white, left), one of which includes a third-party caveat to AS (Authorization Service). K_A is AS's public key. Source: Birgisson et al. (2014)

Chapter 3

Use cases, requirements and system overview

This chapter starts by introducing three use cases that illustrate the problem statement. To illustrate the problems that occur in these use cases more clearly, a generic data aggregation architecture is subsequently presented. This architecture will highlight a number of problems that currently exist and, together with the use cases, will serve to derive a number of requirements and an adversary model.

Thereupon, based on all these requirements, a high-level overview of the middleware system is given. This middleware, called the Middleware for Aggregation Security in Solid (MASS), is partially implemented as a prototype on top op the Community Solid Server. The next chapters then starts by studying a number of PETs in order to determine which technologies are usable in MASS. The chapters following this finally discuss the components of the solution in detail.

3.1 Use cases: health and finance

This section describes three use cases which highlight the need for a middleware for secure and protected data aggregation in Solid. These use cases illustrate the complexities and challenges that come with developing such a middleware, and will serve as a guide throughout the development of the solution. The use cases are located in the domains of health and finance, since these traditionally come with sensitive data. Furthermore, data in these domains can also lead to interesting insights, forming ideal candidates for data aggregation.

3.1.1 Exercise data

In this use case a user stores her exercise data from an application such as Strava[1] on her Solid pod. She wishes to export this data to a ranking board application to see which of her friends runs the fastest. However, she does not wish to share her exact heart rate since this is sensitive data and might leak information about her

[1] An application for tracking running and cycling sessions, see https://www.strava.com

fitness. This data is stored in the TCX file format[2]. An advantage of this format is that it is supported by popular tools such as Strava and can be imported/exported by exercise trackers such as Garmin devices.

3.1.2 Personal finance

This use case describes a scenario wherein a user has stored all his transactions on his Solid pod. He wishes to see some trends and statistics about his spending, and compare this to similar households. An example of such a statistic is "how much do I spend on groceries every week?". However, expenses are very sensitive data, especially when these are exact numbers and store locations. Therefore, the middleware should filter out the most sensitive information while still keeping enough data to generate relevant statistics. This can be done, for example, by removing direct identifiers and perturbing exact spending at individual transactions. For example, for entries of the type "Bob spent €5,82 at Colruyt Leuven on 8/11/2021 16:53", the user wishes that this is modified into something similar to "User87532 spent €5 at Supermarket on 8/11/2021". This way, trends in the spending are kept (by perturbing exact amounts to nearby integers, and replacing exact stores with store types). Information such as "you spent €400 in supermarkets this month" will still be available (and relatively accurate), without giving away exact details. For this use case a custom data format is used, since most standard data schemes for financial information are overly complex for a proof-of-concept.

3.1.3 Aggregated view on personal health data streams

The third use case is taken from one of the SolidLab[3] challenges. Concretely, the challenge *aggregated view on sensitive personal health data streams* is studied[4].

This use case describes a scenario wherein a caretaker wishes to gain insights on all her patients, without knowing exact patient details. These insights are provided through a dashboard, which shows some key statistics about the patient population. Examples of such statistics are average heart rates, average number of steps taken throughout the day, what percentage of the population woke up before 9 am, etc. Of course, the collection and aggregation of such data does not come without issues. Data is derived from activity trackers and IoT sensors, which regularly update data in the user's pod. A secure aggregator must then collect data from all these pods, combine these into aggregate statistics, and write this to a new pod accessible to the caretaker.

[2]Schema: https://www8.garmin.com/xmlschemas/TrainingCenterDatabasev2.xsd
[3]A research project initiated by the Flemish government (De Tijd, 2021a)
[4]See https://github.com/SolidLabResearch/Challenges/issues/16

3.2 A generic data aggregation architecture in Solid

All of the above use cases make use of an aggregation system that collects sensitive data and displays an aggregated view of some kind. Such an aggregation system in Solid will necessarily consist of a number of components, each with their own responsibility. Figure 3.1 illustrates a reference architecture of such a system in Solid.

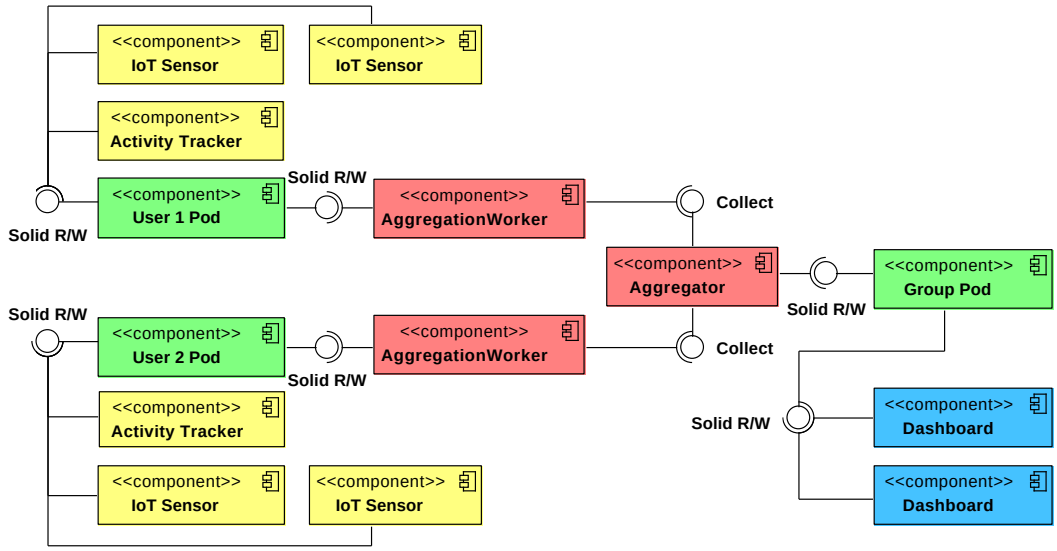


Figure 3.1: Reference architecture for data aggregation in Solid

This reference architecture highlights a number of challenges that make aggregation insecure, unscalable, or privacy-invasive. Concretely, the following issues are remarked:

1. IoT Sensors and activity trackers are embedded devices that regularly write data to a Solid pod. As these are resource-constrained devices, those write operations must be efficient. As section 2.1.3 explains, the currently used mechanism uses public-key cryptography, which is computationally expensive.

2. The aggregator component (illustrated in red on the figure) reads complete resources directly from the user's pod. Privacy-wise, this is unsafe and unnecessary, as not all data present in the resource may be necessary for performing the aggregation. Some method of restricting the data exposed to the aggregator must be devised.

3. The aggregator component can consist of multiple worker nodes in different networks. Similarly, services can require data from a Solid pod but also depend on other services that also need access to this data. This is currently not supported in Solid, and existing flows (such as OAuth On-Behalf-Of) impose bottlenecks on the token endpoint.

4. Group pods holding data of users from different pod providers are not supported as of yet. This comes with challenges in the authentication domain.

All of these problems can be mitigated (at least partially) in the domain of the Solid server, which provides the user's pods. A number of improvements here could thus help alleviate them. The problems can be divided in two categories: resource management and authentication management. To solve these, a solution will need to consist of two parts: one focused on rendering exposed resources more private, the other focused on improving the authorization mechanism. In the next section, requirements for the needed solution will be derived based on the stipulated problems.

3.3 Requirements

This section introduces a number of requirements, based on the use cases and architecture presented in the previous sections. First, functional requirements with a major impact on the architecture are discussed. The second subsection then discusses a number of non-functional requirements. These requirements will help designing an effective solution, and their realization will also be evaluated later.

3.3.1 Functional requirements

Flexibility for the supported data scheme Solid builds further upon the principles of Linked Data, and Solid servers can store nearly any type of data. As MASS aims to be a general solution, it should work with any type of structured data, i.e., it must provide a way to anonymize *any* type of textually represented structured data. Thus, flexibility for the supported data scheme is an essential part of MASS. Concretely, MASS should not hardcode data schemes nor which PETs should be applied to them: these should be able to be plugged in flexibly and selected automatically, to ensure that any data scheme can be supported. This forms a technical challenge, as a sufficient level of abstraction must be developed to be able to support any data scheme, while ensuring that the leakage requirements[5] are not violated.

Automatically select PETs Different applications may be trusted differently, just as different data types may contain more or less sensitive data elements. An important aspect to solve is thus being able to distinguish different *privacy levels*: required levels of anonymization. MASS must therefore not only be able to adapt to different data schemes, but must also support different privacy levels for every data scheme. The selected PET is highly dependent on the data scheme and required level of privacy, therefore MASS must automatically select PETs based on the data type of the requested resource and the requested privacy level. This privacy level must be determined based on the context of the request, i.e., what data type is requested and by which application.

[5] Leakage requirements specify what data can (not) be leaked to certain applications

Extensibility The Solid project is still very much a work in progress. This implies that the specification will likely be modified many times in the future, and additional features will be added. If MASS wants to successfully keep interacting with Solid servers, the architecture should be designed such that it can easily be extended in order to support newly released features and modifications.

Decentralized access token delegation As aggregators may consists of multiple worker nodes or may be dependent of other services, the middleware solution must also enable a mechanism for decentralized access token delegation. In other words, an application must be able to delegate its access token without having to request a new token at the token endpoint. The user should explicitly give access for this.

3.3.2 Non-functional requirements

Intuitive to use Solid aims to become the de facto standard for web applications in the future. Consequently, it must be intuitive for non-technical users to use this technology. There can be no technical jargon, and it must be easy to set up. Since MASS aims to follow this philosophy, the proposed solution must be intuitive to use for non-technical users. Concretely, this means that it should be opaque to the users which concrete PET is applied for which use case. On that account, a number of *privacy levels* should be created and presented to the user in a simple manner: a higher privacy level means more data protection but less utility. The user will then be able to select between a number of privacy levels, without needing to understand the technical details behind the scenes. Examples of concrete transformations can be given, to make the effects of selecting a certain level more comprehensible.

Testability Since MASS operates dynamically, there are many opportunities for unnoticed bugs to appear in the code. Some bugs may only appear under the presence of a specific combination of transformations in the configuration files. Therefore, testability is an important aspect. Increased testability gives stronger guarantees of the correct behavior of MASS, which is essential in the context of a privacy-enhancing technology.

Minimal overhead Since MASS operates as a middleware, it will be called on every resource request. The overhead associated with this should be minimized, such that the used Solid server does not become unusable due to running the middleware.

3.4 Adversary model

The attacker model considered in the design and development of MASS is an honest-but-curious attacker, i.e., an attacker that follows the protocol but will try to read as much information as possible within this confinement. This model is considered since the middleware handles the case of untrusted aggregators and applications. Untrusted in this case means that the user wishes to use the aggregator or application, but wants to minimize the amount of data exposed to it. Thus, the adversary is a

Solid application to which the user grants access and that can thus access any data to which the ACLs give it access. However, there is no fine-grained privacy control, i.e., on the level of a single resource.

It is important to note the goal of MASS in terms of disclosure prevention here. As will be explained in section 4.5, many PETs try to prevent identity disclosure (in datasets containing data from many different users). However, since every Solid pod is linked to a WebID, the main goal of this middleware is to prevent attribute disclosure. Accordingly, the attackers considered try to gain knowledge of *data attributes*. Handling identity disclosure is also an important aspect of a privacy-aware software system, but in the case of data aggregators this must be handled by the aggregators themselves.

3.5 System overview

To solve the problems and fulfill the requirements stipulated above, MASS consists of two main components. The first component is called *privacy filters* and focuses on improving data privacy for data exposed to aggregators (or other applications). The second component is an implementation of macaroons in Solid, i.e. a new access token mechanism. This mechanism supports decentralized delegation, group vaults, and is computationally very efficient (to support data writes by IoT sensors, ...).

Privacy filters are a mechanism to dynamically modify requested resources by restricting sensitive data attributes. When a resource is requested for aggregation, it often contains a lot of unnecessary and private information. To resolve this, privacy filters dynamically modify certain attributes of the resource. This is realized by loading a number of configuration files on to the server (or the user's pod), which describe what *privacy tactics* ought to be applied to a certain data scheme, for a given *privacy level*. Privacy levels are a granular way to select how much to trust a certain application or aggregator (i.e., how private should the resource be rendered). The next chapter studies a number of PETs to determine which are usable in MASS. Privacy filters are then discussed in detail in chapter 5.

To alleviate the authentication and authorization problems that come with data aggregation in Solid, this dissertation further investigates the use of macaroons in Solid. Some background information on macaroons has been given in section 2.2. Macaroons support decentralized delegation out-of-the-box, and also support *caveats* (requirements which must be fulfilled for a token to be valid), as well as third-party attestations. These properties can improve aggregation security, for example by embedding the required privacy level inside the access token, as well as providing a natural mechanism for realizing group vaults using the third-party attestations. Lastly, macaroons use Hash-based Message Authentication Codes instead of public-key cryptography. This creates performance improvements that enable IoT devices or activity trackers to more efficiently write to Solid pods. The integration of macaroons in Solid is discussed in detail in chapter 6. The combination of these two components can enable better and more secure data aggregation in Solid. Figure 3.2 illustrates more concretely the flow of data aggregation using these two new mechanisms.

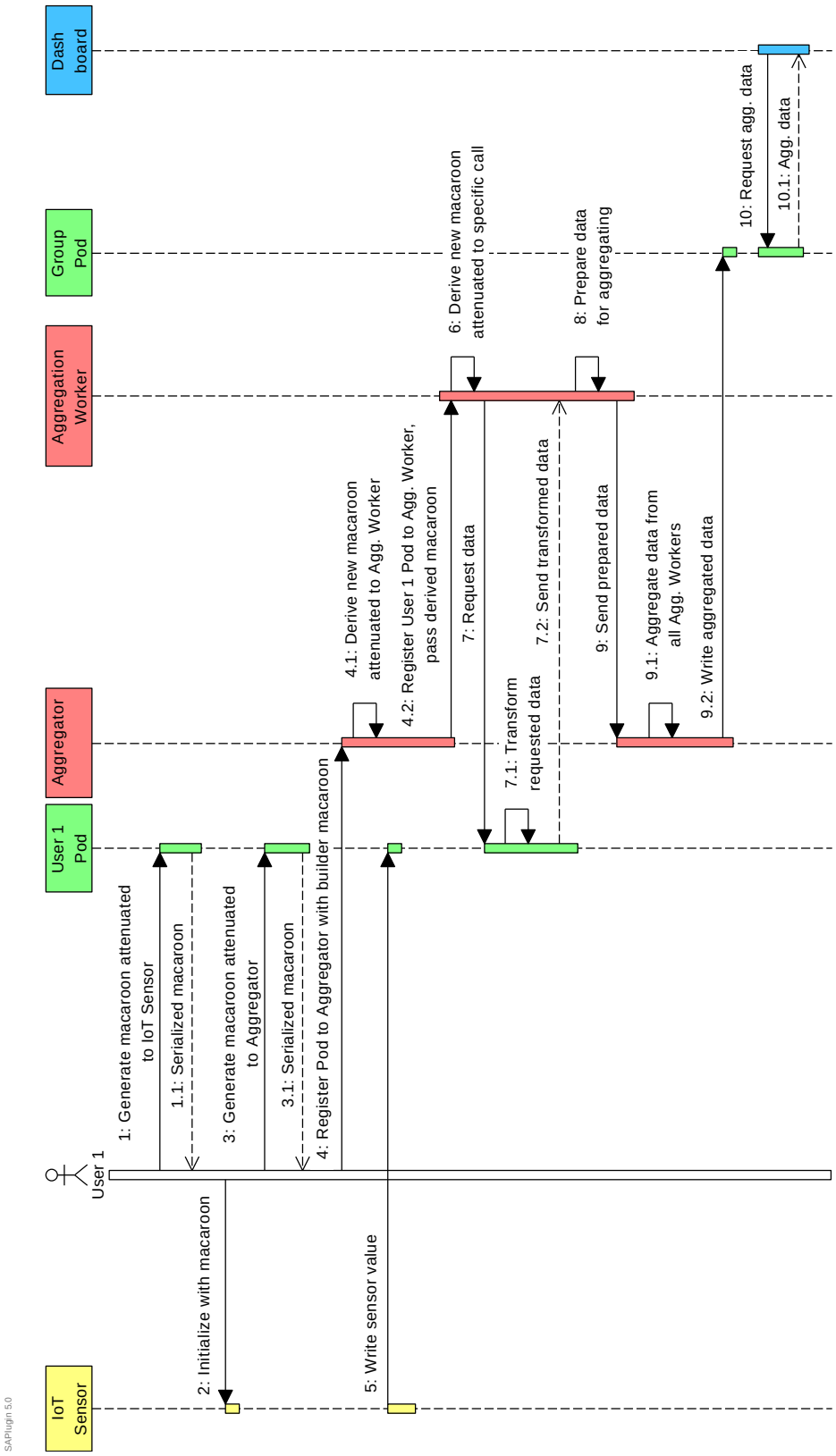


Figure 3.2: Aggregation flow

Chapter 4

Study of privacy-enhancing technologies

This chapter studies a wide range of PETs, discussing how they work and analyzing whether they are applicable in the context of enhancing data privacy in Solid. As explained in section 1.4, a component for enhancing privacy is first studied and developed. This analysis forms the basis for selecting a number of PETs that will be supported in the privacy-enhancing component of MASS.

Section 4.1 starts with an introduction to the concept of PETs, explaining why this kind of technology is needed and what the basic concepts are. The next sections will each treat one specific PET. First, transformation-based approaches are handled by section 4.2. Section 4.3 then continues with encrypted databases, which is followed by Secure Multi-Party Computation. Section 4.5 handles two types of statistical privacy, and the final section discusses Attribute-Based Encryption.

4.1 Introduction

This section introduces the notion of PETs, starting with an explanation of their necessity. Next, this section presents some of the basic concepts of PETs, which come back in the later sections where concrete technologies are discussed.

4.1.1 The need for privacy-enhancing technologies

The concept of privacy has been around for a long time. Even in the 14th century already, cases were brought before court for eavesdropping and opening letters (Holvast, 2009). Initially, it was more understood in the context of being let alone, especially in one's house or personal properties. Since the dawn of the information age after the Second World War, the concept started to shift more towards privacy in the context of personal information and publications on the topic started to appear. The definition did not change much however, remaining very similar to the one introduced in 1891 by Warren and Brandeis (Holvast, 2009):

> "Privacy is described as a right to be let alone and a right of each individual to determine, under ordinary circumstances, what his or her thoughts, sentiments, and emotions shall be when in communication with others."
>
> (Samuel Warren, Louis Brandeis)

The ubiquity of personal data collection in recent years has strengthened privacy concerns. Especially since the Web has become extremely centralized, with only six companies accounting for 43% of global internet traffic (Sandvine, 2021), there has been a growing demand for technologies that help protect user privacy.

Not only from the *Data Subject*'s perspective is there a growing concern for data privacy. With the introduction of the GDPR in 2018, companies have a legal obligation to take care of user data, especially if it is sensitive. This clashes with the rise of the Software- and Platform-as-a-service (Saas/PaaS) business models, where companies outsource data storage and management. However, this is not always possible and comes with risks. If a *Data Processor*[1] (a third party that processes data on behalf of a Data Controller) leaks data, the *Data Controller* risks violating the user's trust (and legal sanctions).

In both cases, privacy-enhancing technologies play a big role in reducing the risk of data leaks and preserving user privacy. Borking et al. introduced the following definition, which was later adopted by the European Commission[2]:

> "Privacy-enhancing technologies is a system of ICT measures protecting informational privacy by eliminating or minimising personal data thereby preventing unnecessary or unwanted processing of personal data, without the loss of the functionality of the information system."
>
> (Borking et al.)

4.1.2 Data (de-)identification

The previous section gave an introduction to the concept of privacy. In IT systems, this applies to personal data. But when is data personal? In the context of digital privacy, the term *PII* is used widely. The term applies to data elements that contain references to attributes that can, either directly or indirectly, disclose someone's identity. A distinction can then be made between three types of data (Van Landuyt et al., 2021):

- **Direct identifiers**: data attributes that directly identify an individual. Examples are national identification numbers, home addresses, e-mail addresses, full names, etc.

[1] Data Subject, Data Processor and Data Controller are roles defined in the GDPR. For a concise but correct definition, please refer to https://ec.europa.eu/info/law/law-topic/data-protection/reform/rules-business-and-organisations/obligations/controller-processor/what-data-controller-or-data-processor_en

[2] https://eur-lex.europa.eu/LexUriServ/LexUriServ.do?uri=COM:2007:0228:FIN:EN:PDF

- **Indirect identifiers**: also called quasi-identifiers, these are data attributes that can be combined with other data attributes to identify an individual, but do not disclose an identity on their own. Examples are birth dates, family names, IP addresses etc.
- **Non-PII elements**: data attributes that contain no reference at all to a person's identity.

Hoepman notes that much like security, privacy is a core property of software systems, and is heavily architecture-dependent. It is therefore of paramount importance that privacy considerations are taken into account from the outset and through the whole development cycle. This design philosophy is named *privacy by design* and it is one of the requirements listed in the GDPR. For regular software design, software developers have used software design patterns to solve common architectural problems. Likewise, Hoepman (2012) and Van Landuyt et al. (2021) propose a number of *privacy design strategies*, which will be used extensively. Based on these strategies, the next section discusses privacy-enhancing data transformations.

4.2 Transformation-based approaches

Transformation-based approaches are approaches that modify datasets in a way such that the resulting dataset contains less PII. The transformation-based approaches studied in this section all happen on a row-by-row basis. Since data transformations *modify* the dataset, it is important to note that this has an impact on the utility of the data. Some transformations may be more powerful at anonymizing the data, but at the same time make the data less valuable or even unusable. A balance must be struck between data privacy and utility, where the choice for a certain data transformation is very context-dependent. This section introduces a number of data transformations.

For an overview of common architectural tactics for data de-identification, a literature study by Van Landuyt et al. is examined. This study includes 163 investigated articles and builds a taxonomy for data de-identification architectural tactics, mostly consisting of transformation-based approaches. Table 4.1 presents their proposed taxonomy of data transformation tactics.

Such transformations seem very usable in the context of a middleware that anonymizes data. They are often fairly simple transformations, that can range in privacy strength and resulting utility. Because these transformations are so dynamic, they are perfect to adapt to different trust levels of applications, allowing more impactful transformations on data sent to applications that are less trusted. Another big advantage of an approach taking advantage of data transformations is that this can be realized statelessly. As the data is sanitized on a row-by-row or record-by-record basis, independently of the other records, there is no need to keep track of state. This heavily simplifies possible architectural designs.

Tactic name	Description
Remove	The targeted and deliberate omission of PII from the data record or data set.
Select / Filter	Same as Remove applied to a copy of the data while keeping the original data.
Transform	The application of a holistic transformation (not targeted on PII elements only) on data elements that is destructive in terms of ability to distinguish these data elements.
Mask	Overwriting data elements with other values for the purpose of obfuscating the content.
Encrypt	Using cryptographic means to encode the PII attribute values / records / datasets.
Randomize	Tactics that involve the modification of PII attribute values/records by injecting artificial random elements.
Aggregate	Generalize, combine and group data elements.
Blur	Reduce the accuracy of the data so that distinguishing characteristics become indiscernible.
Pseudonym	The systematic replacement of direct identifiers with surrogates, whereby the mapping between surrogate and identity is kept separately.
Placeholder	The systematic replacement of direct identifiers with a surrogate that is the same for all members of the population.
Interchange	The targeted mingling of data elements across members of the population.
Synthetic	The replacement of data elements with a synthetic placeholder which is a constructed representation specifically aimed at conveying discriminatory features.

Table 4.1: Overview of architectural tactics involving data transformations, by Van Landuyt et al. (2021)

4.3 Encrypted database systems

Encrypted database systems are database systems that store encrypted data, while maintaining some functionality over this data. It is a vast research field, with Fuller et al. (2017) being a formative work. The need for encrypted database systems arises from the rise of the PaaS model, whereby software and database deployment is outsourced to third parties such as Amazon Web Services and others. Since these cloud platforms store sensitive data (customer data, economically valuable information, ...), it is important for certain users to guarantee that these providers are unable to access their stored data. This is realized by encrypting the data, but this brings along problems. As these databases are often very large, information is retrieved through complex queries. These queries may not support operating on

encrypted data by default, so some new techniques are necessary (Heydari Beni et al., 2019).

Many types of encryption exist for these database systems, ranging from very secure to less secure, but in turn providing more functionality. Popular encryption schemes, sorted from least to most leakage, are (Rafique et al., 2021; Heydari Beni et al., 2019):

- **Random/probabilistic encryption**: Random encryption is an encryption scheme in which encrypting the same plaintext results in a different ciphertext. This is usually achieved by making use of the AES encryption scheme, and using a randomly generated IV. This scheme provides the strongest data protection, but it offers no useful operations on the data.

- **Deterministic encryption**: Deterministic encryption works similar to random encryption, but now the IV is kept constant. This results in a scheme which gives the same ciphertext for the same plaintext input. This is less secure, but allows for equality comparisons.

- **Order-revealing encryption**: In the order-revealing encryption scheme, the order relationship between plaintext is preserved in the ciphertext. Therefore, it is possible to do order-comparisons between encrypted data elements, allowing functionality such as range queries.

- **Order-preserving encryption**: Order-preserving encryption is an outdated form of order-revealing encryption. Its usage is discouraged, since it has a larger leakage than order-revealing encryption while providing the same level of functionality (Boneh et al., 2016).

A new type of encryption scheme that has recently started to become popular is *homomorphic encryption*: it allows processing on encrypted data by supporting mathematical operations on the encrypted data, such as multiplication and addition. It has become popular over the last decade, with Gentry (2009) being a breakthrough work and Google recently releasing an open-source library[3] supporting fully homomorphic encryption. While somewhat homomorphic encryption already has practical applications, this is less the case for fully homomorphic encryption, with the main hurdle being that it is very computationally intensive.

- **Somewhat homomorphic encryption**: Somewhat homomorphic encryption schemes, also called partially homomorphic encryption schemes, are homomorphic encryption schemes that support either addition or multiplication, but not both (Pisa et al., 2012). An example of an additive homomorphic encryption scheme is the Paillier Cryptosystem (Paillier, 1999). A widely used multiplicative homomorphic encryption scheme is the ElGamal Cryptosystem (ElGamal, 1985).

[3] https://github.com/google/fully-homomorphic-encryption

- **Fully homomorphic encryption**: Fully homomorphic encryption schemes are homomorphic encryption schemes that support both addition and multiplication on encrypted data. This is incredibly powerful, since any function can be represented as a circuit consisting of multiplication and addition gates. It can, theoretically, thus compute any function over encrypted data. However, it is still too computationally intensive for a number of practical applications (Naehrig et al., 2011; Evans et al., 2018). Despite that hurdle it remains a promising technology, allowing applications such as encrypted search queries (Gentry, 2009), and every year more efficient implementations are proposed.

While encrypted database systems form a very active research domain, their possible applications within Solid are rather limited for a number of reasons. First and foremost, Solid is not a database system and does not provide data with support for queries (at the moment). Instead, data fetched through the Solid protocol is in the form of resources, which map to files on a filesystem. In the future, Solid may support RDF queries on the resources, but since this is rather specialized implementing such a system would also be a complex task. The second and most important point is that these encryption systems are ideal for when a single client needs access to a database. However, this maps badly to Solid's decentralized nature, where multiple clients (which are trusted differently and have different access control rights) access multiple Solid pods. This complicates the encryption possibilities substantially, especially regarding key management.

Homomorphic encryption does have some interesting properties that would benefit from being researched in the context of Solid. Homomorphic encryption could be used to give some encrypted data to an application that then performs computations on it. In this way, an application could aggregate data from a number of different pods, perform some calculation on it, and then return this data to the clients giving them insights. This scenario is further developed as future work in section 8.2 under FW 1. This fits well in the goals of this dissertation, but then focused on the software that performs the aggregation instead of the middleware.

4.4 Secure Multi-Party Computation

MPC is an exciting research field that has seen a large amount of research over the last few decades, while the techniques and modern computers have only recently evolved enough to allow practical applications. It is a technique that allows any number of parties to secretly compute any function over some encrypted input data, while only revealing the requested output to the parties. The basic techniques for Multi-Party Computation stem from a groundbreaking paper by Yao (1986) (for two-party computation), which was later improved to support the multi-party case by Goldreich et al. (1987).

MPC has a somewhat unusual security model, since the adversary can be a party to the computation instead of being an outsider. The adversary model thus considers some adversarial entity, which manages to exert control over a subset of the parties partaking in the computation. Such a controlled party is called *corrupted*. Often,

a *monolithic adversary* is considered: if there are multiple corrupted parties, it is assumed they work together (Orlandi, 2011). When an adversary controls multiple corrupted parties, several *corruptions strategies* are possible (Lindell, 2020). In the *static corruption model*, the set of corrupted parties is fixed from the onset and does not change during execution of the protocol. In the *adaptive corruption model*, the adversary can corrupt parties during the execution, but once a party has been corrupted it remains corrupted until the end of the protocol. Finally, there is the *proactive security model* (Canetti and Herzberg, 1994), where corrupted parties can become honest again (for example, because a breach has been detected and the systems are recovered).

Of course, many protocols exist for facilitating computation between multiple parties, even HTTP would support that. The goal of MPC is to provide a *secure* protocol. Often, the *real-ideal*[4] definition of security is taken to define when such a system is secure. However, this does not provide many practical guidelines. Lindell (2020) lists a number of minimal (but not exhaustive) requirements, which every secure protocol should fulfill:

> 1. **Privacy**: No party should learn anything more than its prescribed output.
> 2. **Correctness**: Each party is guaranteed that the output it receives is correct.
> 3. **Independence of Inputs**: Corrupted parties must choose their inputs independently of the honest parties' inputs.
> 4. **Guaranteed Output Delivery**: Corrupted parties should not be able to prevent honest parties from receiving their output.
> 5. **Fairness**: Corrupted parties should receive their outputs if and only if the honest parties also receive their outputs.
>
> (Lindell (2020, p.2))

The actual computation, then, is usually based on something called a *garbled circuit* (Yakoubov, 2017). The function to be computed is represented as a boolean circuit whose gates are encrypted. The explanation of the complete protocol is out of scope for this dissertation, but it is described in Lindell (2020); Evans et al. (2018). For some use cases, generic MPC protocols may not be the most efficient and specific protocols that are much faster may exist. A very common example is private set intersection.

Because of the decentralized nature of MPC, this forms a very interesting technique to research in Solid. It could be used to securely compute insights over data stored in multiple pods, without even needing to hand out an encrypted version of the data. Computations are performed on the pods itself, and only *secret shares*

[4] In the real-ideal security definition, an adversary cannot harm the real protocol more than what would be possible in an ideal protocol (Goldreich et al., 1987)

are sent out[5]. While this is very interesting, it also entails a very large extension of the Solid protocol or a Solid server. Moreover, this redefines the concept of a pod, making it also a compute engine instead of just a storage space. This scenario is also considered future research and is elaborated in section 8.2 under FW 2.

4.5 Differential Privacy & *k*-Anonymity

Differential privacy and k-Anonymity are both types of *statistical* privacy. In differential privacy, *"participating in a database does not substantially increase the risk to the user's privacy"* (Dwork, 2006). This definition follows from Dwork's impossibility proof of absolute disclosure prevention. She proved that even if a database has very little information (such as the average height of a person for every country), even that information can lead to a privacy breach in the presence of side information. Imagine that a person's height is sensitive information, and that it is known that Alice's height is 2cm less than the average height. Then the database containing average heights discloses Alice's height, leading to a privacy breach. Because this is impossible to circumvent, Dwork introduces the relative notion: a disclosure is just as likely whether the individual takes part in the database or not. In short, the technique works by adding specifically chosen random noise to the answer of a query, to statistically provide the ϵ-differential privacy guarantees. A more rigorous explanation can be found in Dwork (2006, p9-11).

k-Anonymity was introduced by Sweeney (2002), who defined it in terms of de-identifying persons from so-called *quasi-identifiers* (i.e. indirect identifiers) in released data sets. Sweeney (2000) notes that 53% of the U.S. population can be uniquely identified with only the following attributes: place, gender and date of birth. In this context, it is very difficult to correctly identify indirect identifiers, since it is not known beforehand with which data this can be combined to determine unique identities. If a dataset is k-anonymous, however, it is very hard to uniquely identify individuals (given that no other dataset is released — see footnote 6). A data release is k-anonymous if "the information for each person contained in the release cannot be distinguished from at least k-1 individuals whose information also appears in the release" (Sweeney, 2002). Thus, every unique combination of identifiers maps to at least k individuals, making individual identification unlikely[6].

While both technologies are very interesting, there are a number of crucial aspects that stop them from being usable in the privacy-enhancing middleware this dissertation aims to develop. First of all, both methods are applicable in datasets with *multiple* users. There is no point in using these methods on data originating

[5] A secret share is a part of the data that is indistinguishable from a random string. It is used in MPC to perform the computations. When x is a confidential piece of data, you could build two secret shares by generating a random p and handing out $x + p$ and p. Only by combining the two secret shares can the original secret x be reconstructed

[6] There exist attacks on k-anonymity, but most can be resisted by following the accompanying policies stipulated in Sweeney (2002). Most of these attacks stem from the fact that k-Anonymity does not compose: combining multiple k-Anonymous datasets does not lead to a combined dataset that is also k-Anonymous

from a single user. For differential privacy, this is because it is defined as your participation in the database having a negligible impact (defined by ϵ). However, when the database only consists of records belonging to a single user, this is impossible. On the other hand, k-Anonymity lacks that it focuses on a different aspect than what the middleware tries to achieve. This is because k-Anonymity focuses on preventing *identity disclosure*, while in the middleware itself the goals are to lower the risks of *attribute disclosure*. However, k-Anonymity may be an interesting approach to prevent identity disclosure on an aggregation system (i.e. the component that collects data across many pods, or the `Aggregator` in figure 3.2).

4.6 Attribute-based encryption

Attribute-Based Encryption (ABE) is an encryption scheme that was introduced by Sahai and Waters (2005). In their paper, the authors developed a new type of Identity-Based Encryption (IBE). Traditional IBE systems typically use a single string to represent a user's identity; Sahai and Waters developed a scheme where the identity is not represented by a string but by a set of descriptive attributes. In their scheme, encrypting documents happens with a set of required attributes included. Decryption can then only happen by users whose identity contains all the attributes (or, where there is a big enough set overlap, in their original scheme). However, Sahai and Waters's development focused mostly on using IBE with biometric identities. A number of follow-up papers extended this idea to make it more suitable for ABE, using systems such as KP-ABE and CP-ABE.

Goyal et al. (2006) extend the original paper with Key-Policy Attribute-Based Encryption (KP-ABE) and make it more suitable for fine-grained access control. They do this by introducing a new access structure based on a tree consisting of `AND` and `OR` gates[7], and the leaves containing attributes. Decryption of the message is then allowed when the attributes satisfy the tree. This access structure, called the policy, is stored in the private key. In the original model, *all* attributes had to match approximately (within the predefined set distance), so this new access structure is much more expressive. KP-ABE associates the ciphertexts with these descriptive attributes, and the user's keys with the policy.
This scheme works similar to secret sharing schemes, where a number of parties (respectively, attributes) must collude to be able to reconstruct the secret. The main difference is that secret sharing allows cooperation between different parties, but KP-ABE does not. If this were allowed, some security requirements could be bypassed. This property is called *collusion resistance*, and is an essential property of ABE systems (Bethencourt et al., 2007).

As an example, take the case of government health data that is encrypted with the attributes `government AND health`. You would not want a government worker with attribute `government` to be able to collude with a nurse with attribute `health` to decrypt this data.

[7] Actually, the tree consists of nodes that act as threshold gates. `AND` gates are realized as n-out-of-n threshold gates, while `OR` gates are realized as 1-out-of-n threshold gates

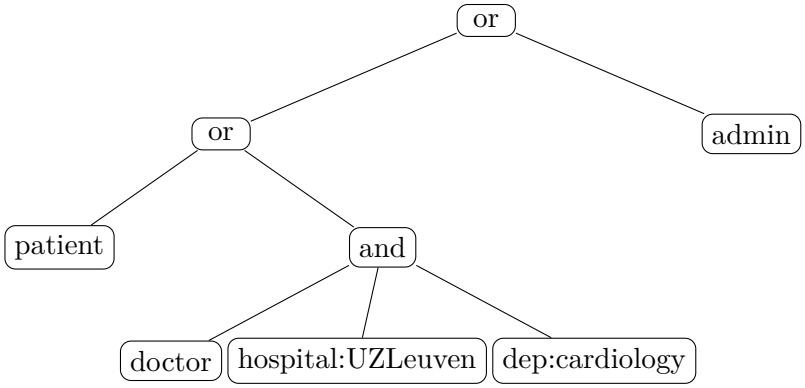


Figure 4.1: Example ABE policy

Ciphertext-Policy Attribute-Based Encryption (CP-ABE) then, is introduced by Bethencourt et al. (2007). CP-ABE reverses the situation of KP-ABE, by associating the private keys with descriptive attributes, and the ciphertexts with the access structure (also called the policy). Because of this, the secret sharing technique can no longer be used and Bethencourt et al. introduce a new private key randomization technique. The table below highlights the main conceptual differences between KP-ABE and CP-ABE.

associates-with	KP-ABE	CP-ABE
Ciphertext	sets of descriptive attributes	policies
Key	policies	sets of descriptive attributes

Table 4.2: Conceptual differences between KP-ABE and CP-ABE

In the context of Solid and a privacy-enhancing middleware, ABE provides a number of useful applications. The transformation-based approaches in section 4.2 protect against the threat of untrusted applications. ABE, on the other hand, could be interesting to use when the Solid pod is not trusted completely. Extremely sensitive data (such as doctors reports or police case files) could then be stored on Solid pods, guaranteeing data security even when the pod is compromised or access control systems are bypassed.

Conceptually, the user in ABE schemes then maps to a Solid application, while the encrypted data is stored at the Solid pod. The (Solid) user can then specify which applications map to which attributes, while simultaneously deciding which access structures should be associated with a Solid resource. Because of this association, CP-ABE is the most suitable type of ABE to use in such a middleware. Investigating ABE as a mechanism for realizing sharing of sensitive data within Solid is out of scope for this dissertation, so researching possible (attribute-based) encryption techniques in this context may be interesting future research. This is discussed in section 8.2 under FW 4.

Chapter 5

Privacy filters

As discussed chapter 3, the goal of this chapter is to find a solution to the information leakage problems that come with data aggregation in Solid, and with requesting resources in general. This problem arises due to the fact that no distinction is made between the trust level given to client applications. These applications either get the full resource (when they have the necessary authorization), or they don't get the resource at all. However, since mostly structured data is handled by Solid, it should be possible to restrict which parts of a resource are exposed and allow more granular access control (i.e., within a resource).

As illustrated by figure 5.1, improved privacy always comes with a trade-off regarding data utility. Evidently, data with less or less precise attribute values will be more private, but this same property also renders the data less useful. Making this trade-off is very context-dependent (how much is the application trusted, what kind of data is requested etc.). As such, rendering data more private must happen dynamically, at the time of a request, such that details of the request can be taken into account when determining what transformations ought to happen.

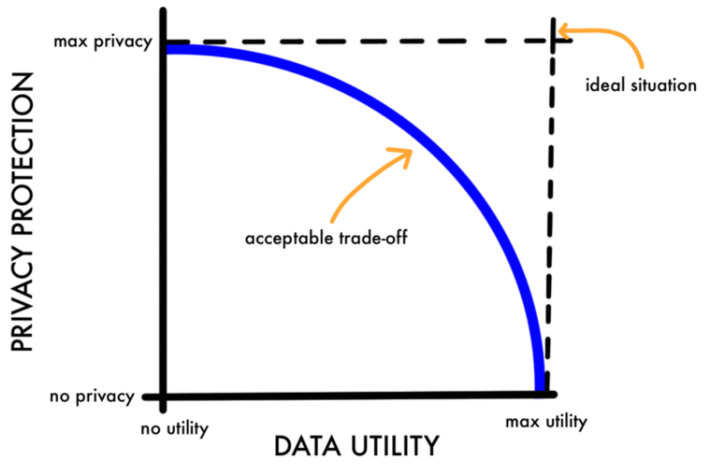


Figure 5.1: Illustration of the privacy-utility trade-off, from Nelson (2015)

This chapter introduces the concept of *privacy filters*, which dynamically strip away attributes of exposed data in accordance with some predefined rules. A prototype that implements privacy filters is developed for the Community Solid Server[1] as a part of the MASS middleware. The concrete implemenation of privacy filters as a software component was originally named PePSA (short for Privacy-enhancing Plugin for Solid Applications). However, since it forms a part of the complete middleware, the specific implementation will hereafter also be referenced to as MASS. This implementation provides a middleware in the Solid server which provides more fine-grained privacy control options.


```
{
"accountOwner": "Jesse Geens",
"IBAN": "BE66123456783456",
"saldo": 665.53,
"currency": "EUR",
"history": [
    {
    "from": "BE17954824458821",
    "to": "BE66123456783456",
    "from_name": "Mark Jansen",
    "to_name": "Jesse Geens",
    "amount": 33.7,
    "timestamp": 1645447425,
    "description": "Payback
    ↪ borrowed money"
    }, {
    "from": "BE66123456783456",
    "to": "BE15954347627130",
    "from_name": "Jesse Geens",
    "to_name": "Steffie Smits",
    "amount": 10.0,
    "timestamp": 1645440365,
    "description": "Drinks"
    }
]}
```


(a) Data before applying privacy filter


```
{
"accountOwner": "J.G.",
"IBAN": "BE42429824824354",
"saldo": 712.1171,
"currency": "EUR",
"history": [
    {
    "from": "BE17954824458821",
    "to": "BE42429824824354",
    "from_name": "Mark Jansen",
    "to_name": "J.G.",
    "amount": 34.8,
    "timestamp": 1645440000,
    "description": "Payback
    ↪ borrowed money"
    }, {
    "from": "BE42429824824354",
    "to": "BE15954347627130",
    "from_name": "J.G.",
    "to_name": "Steffie Smits",
    "amount": 9.45,
    "timestamp": 1645440000,
    "description": "Drinks"
    }
]}
```


(b) Data after applying privacy filter

Figure 5.2: Example of data treatment by privacy filter, applied to financial transactions data

[1] https://github.com/CommunitySolidServer/CommunitySolidServer

In the developed prototype privacy filters are represented by a JSON document (per data scheme), describing which transformations ought to happen to which data attribute. An example of such a configuration file can be found in the appendix, in listing A.2. Users can then specify how much privacy they want (in general, or for specific data types) in an abstract way using privacy levels, which are introduced in the next section. When a request is received, MASS checks what data type it is, which application requested the data and what level of privacy the user has requested. Subsequently, a number of transformations are applied to the data (as specified by the configuration file), after which the transformed data is returned to the application.

An example of what such a rewrite can look like is provided in figure 5.2. In this example, transformations such as *pseudonym* (on the `accountOwner`, `name`, `IBAN` and `from`/`to` fields) and *perturbation* (on the `saldo` and `amount` fields) are applied. A complete list of supported transformations is discussed in section 5.2.1. The next section first discusses the earlier introduced concept of privacy levels, which provide granularity in the performed transformations. The subsequent sections then discuss how privacy filters were implemented as a part of MASS by giving a detailed description of some important design decisions, followed by an overview of the architecture. Evaluation of the developed prototype is discussed in chapter 7.

5.1 Privacy levels

In order to provide an intuitive mechanism for selecting which data is transformed, the concept of privacy levels is introduced. Privacy levels form an abstraction above the concrete data transformations and PETs that are applied to the data before it is passed on to the application. This ensures that non-technical users can use a privacy-enhancing middleware, without needing to understand the technical details of the technologies and tactics that are employed.

MASS is configured with a default privacy level, but also supports specific privacy levels for certain data schemes. For example, a user could configure MASS to use privacy level 2 by default, but privacy level 4 on bank transactions, since he does not want to expose this data. This makes the middleware more context-sensitive and allows for strong configurability. The privacy-utility trade-off can then also be taken into account for specific applications that need data with very high utility to deliver usable results. To make clear to the user what data is exposed an explanation should also be included (for every data type) which specifies concretely what will happen to the data under a certain privacy level.

Another important aspect is that the developers of privacy filters for specific data schemes must be aware of what privacy levels map to what leakage or tactics. This is best determined by privacy experts as it is a very complex topic. However, appendix A.1 shows an example of such a mapping that was used to guide the development of configuration files for the prototype of MASS.

5.2 Architecture

MASS is implemented as an extension of the Community Solid Server. The complete code and instructions can be found at https://github.com/jessegeens/pepsa-component. First, a number of important design decisions are discussed in this section. Afterwards, the concrete implementation of privacy filters as an extension of the CSS is reviewed.

5.2.1 Design decisions

In every architectural design, important decisions have to be made based on some sort of cost-benefit analysis: there is no free lunch. Similarly, this is the case with the design of MASS. This section describes the reasoning behind a number of important design decisions that have been made during the modelling and development.

Positioning

A first important aspect of the design of MASS is where it is logically positioned. The choice was made to position the middleware as an extension of an existing Solid server (in our case, the CSS). In this regard it is a true middleware. Furthermore, this results in significantly sped up development compared to a stand-alone implementation, as well as better performance. A disadvantage of this approach is that it leads to more centralization: if a user is using a pod provided by a third-party not running MASS, there is no way to enable this. Extending an existing solid server also means choosing one specific server to support, making the solution incompatible with other servers. However, developing the middleware as an extension of a Solid server makes it such that it does not really interact with the Solid protocol itself. As such, the middleware becomes much easier to keep up-to-date with the quickly evolving Solid specification.

The other possibility that was considered would be to build MASS as a proxy. Requests from the Solid application would be directed to MASS instead, which fetches the data, sanitizes it, and then replies with the sanitized data. This is a much less centralized option, but comes at a big performance cost. There is more network usage and double the number of HTTP requests, for example. However, there is also a very important technical limitation. As was discussed in section 2.1.3, Solid uses DPoP tokens. As can be seen in listing 2.2, the `htu` parameter of the DPoP token body prevents this token from being used in another pod, essentially making the proxy solution impractical when communicating with a Solid server using this authentication mechanism.

Supporting multiple data schemes

Solid supports storing nearly any type of resource, both linked and non-linked. The data that is most prone to leakage in the context of a data aggregator is structured data. Structured data can take many forms, and any good middleware should be independent of the types of structured data that are requested or pass through it.

However, MASS needs to perform operations on the data that passes through it and should thus have some context as to how it should handle a specific data scheme.

This is realized by providing MASS with an internal library of parsers for each content representation, which support a predefined set of transformations. For example, there is a component that performs pseudonymization on data that is represented in JSON. At startup, the software reads a number of configuration files, one for every supported data scheme. These files then specify how a data scheme should be detected, and what transformations ought to be applied to it. Some disadvantages of this approach are that this requires more up-front coding to support this library of transformations for content representations. There is also a computational cost associated with having to parse all these configuration files and apply the rules expressed in them, instead of directly executing code. Despite these drawbacks, there is also a very large advantage to this approach: supporting new data schemes becomes a much easier and faster task. The only work that needs to be performed to support a new data scheme would be to write a scheme configuration file, something that can be done in a few hundred lines of JSON. The appendix contains the JSON Schema used to describe such configuration files (see listing A.1) as well as an example of such a scheme configuration file (see listing A.2). This configuration file is aimed specifically at use case 3.1.2.

Another possibility that was considered would be to have separate components, where each component is responsible for performing transformations on the input data of a specific data scheme. The components would be loaded dynamically at run-time, using dependency injection technology such as *components.js*[2]. This would result in great flexibility: the transformation component can perform virtually any transformation. However, a first big disadvantage of this strategy is that it results in a significant development time for each component (and thus, for every supported data scheme). This can result in a lack of support for many (popular) data schemes, making MASS significantly less useful. Another major downside is that this imposes new weaknesses from a security point-of-view, as untrusted code is imported in and executed on the Solid server.

Supported transformations

Table 5.1 gives an overview of which transformations are currently supported. The choice for supported transformations is mostly based on the discussion from section 4.2, which described a taxonomy of architectural tactics involving data transformations. Some tactics could, however, not be mapped to an equivalent tactic in our middleware. An example of this is the *Aggregate* tactic as MASS does not support grouping data elements, since data is treated on a per-attribute basis. Similar arguments hold for other tactics that are not supported. The *Perturbation* tactic is based on the *Blur* tactic from table 4.1, but renamed to convey its applicability to numeric values. Similarly, *Encrypt* is supported along with the similar *Hash* tactic.

[2] https://componentsjs.readthedocs.io/en/latest/

Transformation	Description	Type
Remove	The targeted and deliberate omission of PII from the data record or data set.	$any \rightarrow null$
Pseudonym	The systematic replacement of direct identifiers with surrogates, whereby the mapping between surrogate and identity is kept separately.	$string \rightarrow string$
Perturbation	The insertion of randomized noise into the values of the data to hide exact values	$num \rightarrow num$
Random	Tactics that involve the modification of PII attribute values/records by injecting artificial random elements.	$any \rightarrow any$
Encrypt	Using cryptographic means to encode the PII attribute values / records / datasets.	$any \rightarrow string$
Hash	Using cryptographic hash functions to obfuscate the PII attribute values / records / datasets in a deterministic fashion.	$any \rightarrow string$

Table 5.1: Overview of data transformations supported in MASS, based on Van Landuyt et al. (2021)

5.2.2 Implementation in CSS

The implementation of MASS is realized as an extension of the Community Solid Server, leveraging components.js to substitute a component of the CSS with a component provided by MASS. An overview of the architecture of the CSS is presented in figure A.1.

Concretely, the main component of MASS is the `AnonymizingHTTPHandler`, which extends the `OperationHttpHandler` class provided by the CSS. Using a custom components.js configuration, this new class is injected into the CSS by giving it as an argument to the `ParsingHttpHandler`. On diagram A.1 in the appendix, this is located under `AuthenticatedLdpHandler`. A configuration file for the CSS specifying the dependency injection of the `AnonymizingHTTPHandler` can also be found in the appendix, under listing A.3.

Figure 5.3 gives an overview of the system architecture of MASS. Components that are part of a vanilla version of the CSS are colored in blue. The red component, `AnonymizingHTTPHandler`, is the main component of the middleware and is the component that is injected into the CSS. The parsers (in yellow) are also included dynamically through the configuration file (listing A.3), such that other content representations can also easily be supported. The `ParserSelector` can then, based on the content representation that is defined in the detected data scheme, select the correct parser to perform the execution of all specified transformations. Figure 5.4 illustrates the flow of how a privacy filter is applied to an incoming request in the CSS.

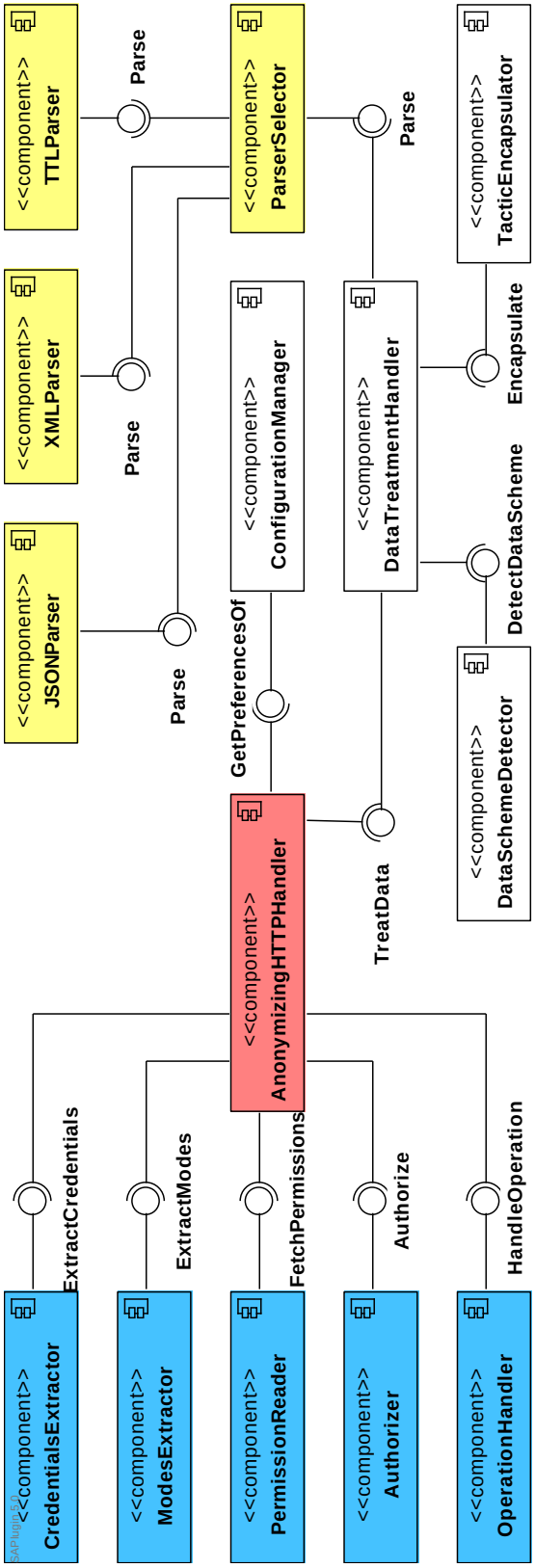


Figure 5.3: Overview of MASS privacy filter architecture

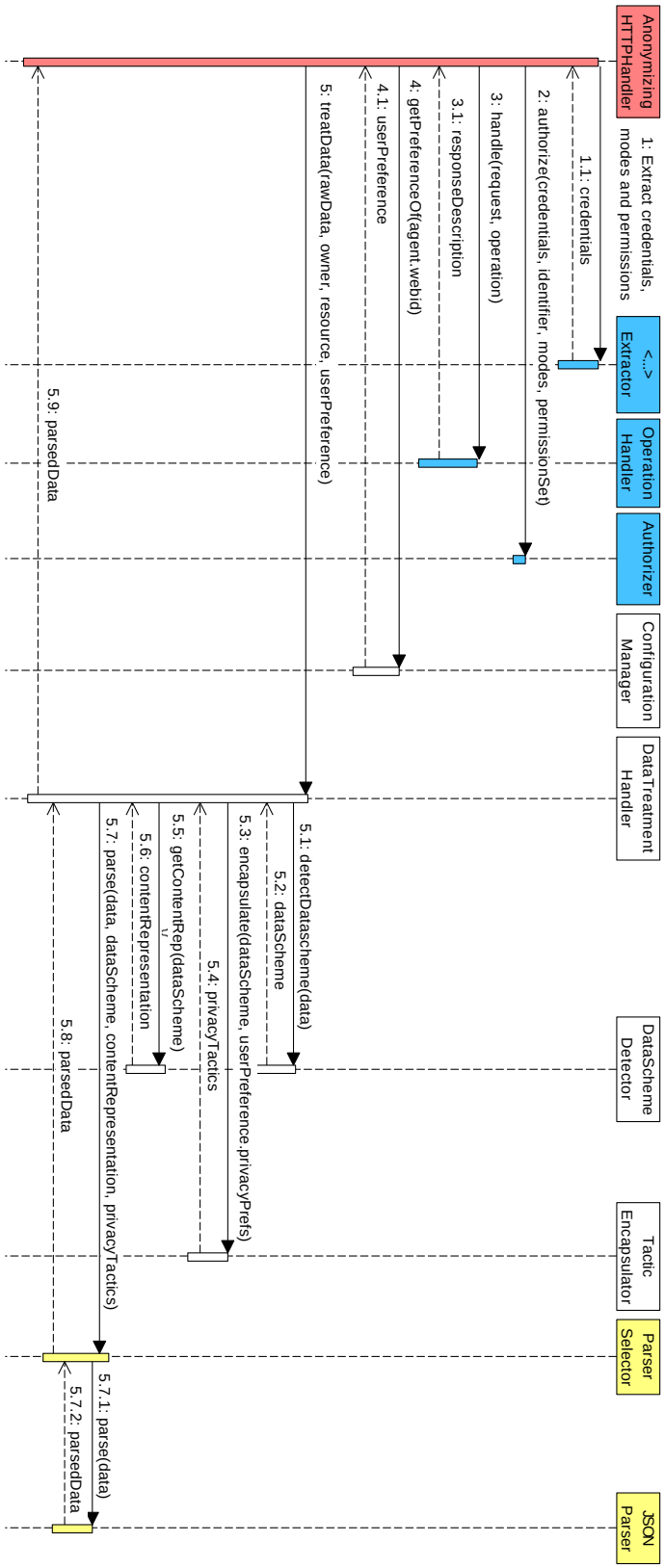


Figure 5.4: Flow of MASS request rewrite

Chapter 6

Macaroons as a novel access token mechanism in Solid

Section 2.2 gave an introduction to macaroons, a novel kind of access token. This chapter discusses ways in which macaroons can mediate the problems stipulated in chapter 3. The first part of this chapter focuses on group vaults, or pods that are controlled by multiple users. The second part of the chapter focuses on how macaroons can enable decentralized access token delegation, by highlighting how token delegation works traditionally and illustrating the improvements macaroons bring here.

One of the problems that was identified in section 3.2 was the high computational cost of a token mechanism that uses public-key cryptography. A lower computational cost is realized by macaroons because of its hash-based signatures, instead of using public-key cryptography. Real performance gains of utilizing macaroons instead of DPoP are explored in section 7.2.2. Using macaroons for realizing group vaults and decentralized delegation is explored in the next sections.

6.1 Group vaults

Group vaults are pods which hold data of multiple users in a single pod. This could be useful in many scenarios, for example a pod which holds data for a family (such as incoming invoices, controlling light switches in the home, etc.). However, realizing secure group vaults poses problems due to the decentralized nature of Solid. Users in a group vault can have identities that are issued by different services. For example, users *Alice*, with WebID `https://alice.solid.org` (issued by `solid.org`), and *Bob*, with WebID `https://bob.pods.org` (issued by `pods.org`), would like to open a group vault. When a request is made to the Group Vault Server (GVS) (for modifying or reading a resource), a token must be included that authenticates Alice or Bob. However, the GVS is not the server that issued the token. Moreover, the token server that must be contacted for validating the user's identity is dependent on which user makes the request.

Macaroons (Birgisson et al., 2014) can bring a solution to this problem thanks to one of its features called *third-party attestation*. In this mechanism, a macaroon can contain a caveat (requirement) that must be attested by a third party. This third party can then generate a *discharge macaroon*, which serves as a proof that the caveat laid out in the original macaroon is fulfilled. In the case of group vaults, these third-party attestations can be used to prove to the group vault server that a user is authenticated by a third-party identity provider. This section explores an architecture for a group vault server which uses macaroons to authenticate its participants.

6.1.1 Group vault architecture

Group vaults are Solid pods that store data for a group of participants. Traditionally, such as for example in the Community Solid Server, the server that hosts the pod (the data) is the same server that acts as the token endpoint for verifying access tokens of the user. In the context of group vaults, the GVS does not issue tokens and as such can not verify them. Instead, the GVS keeps track of which users belong to the group and holds the data belonging to the group vault.

When a user wishes to access a group resource, it fetches a macaroon from the GVS, specifying his WebID and the group from which he would like to access resources. The GVS then issues a macaroon, limiting its use to that specific user, and including a caveat that the user is logged in at his identity provider. The user must then contact his own token endpoint and request a discharge macaroon, which serves as proof of his authentication. The user can then request resources from the GVS by including his macaroon and its associated discharge macaroon. If many successive requests must be made (for example, fetching the contents of a resource container), the token endpoint of the user must only be contacted once and the discharge macaroon (or one derived from this) can be re-used multiple times.

Figure 6.1 illustrates the flow of a user requesting a resource from a GVS. The GVS guarantees that only the correct server can create the discharge macaroon, by encrypting the necessary caveat key with the token endpoint's public key. The location of this public key can be found in the `openid-configuration` file. An example lay-out of the macaroon caveats can be seen in figure 6.2.

To verify the feasibility of using macaroons as access tokens for realizing efficient group vaults, a prototype has been developed in JavaScript, leveraging the macaroons.js library[1]. The prototype is not built following the Solid specification but implements all the necessary features for validating the general architecture. The source code of this prototype can be found at https://github.com/jessegeens/groupvault-demo.

[1] See https://www.npmjs.com/package/macaroons.js/v/0.1.0

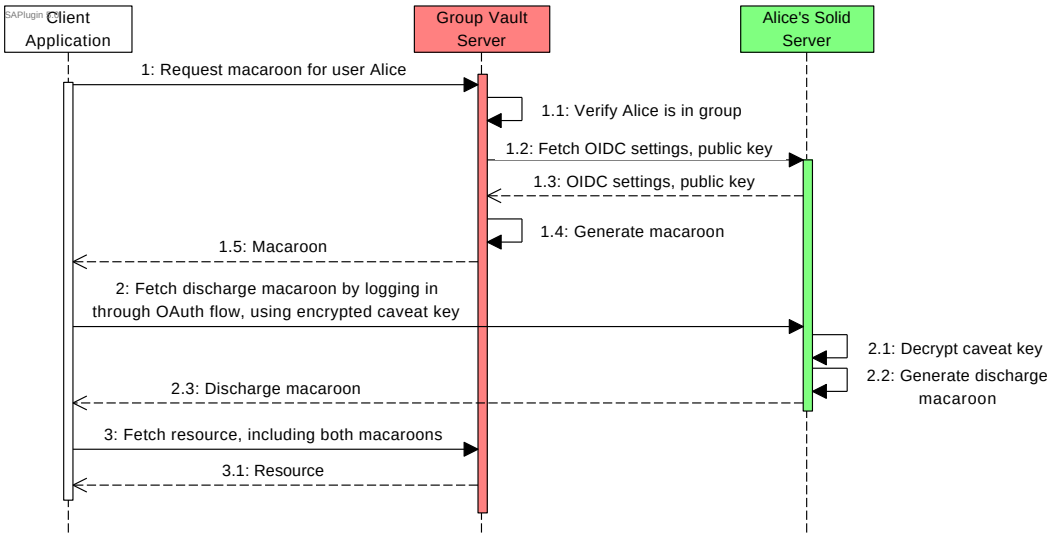


Figure 6.1: Authentication flow for accessing a resource from a Group Vault Server.

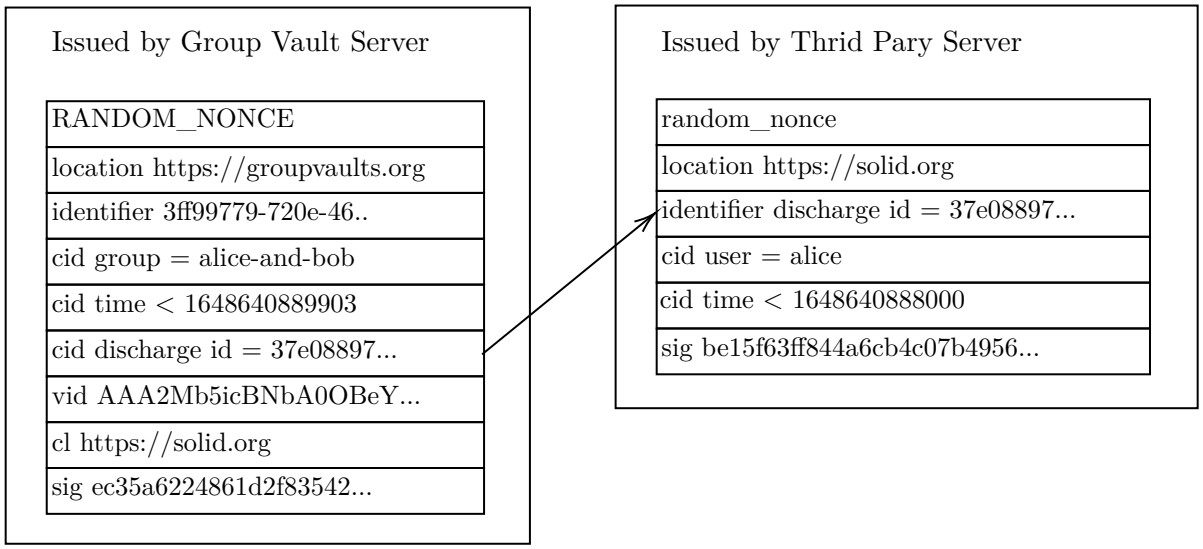


Figure 6.2: Example group vault macaroon and related discharge macaroon.

6.2 Decentralized delegation of access tokens

A second problem in the context of secure data aggregation is delegation of access tokens. In the case of more complex aggregations, there is often more than one node performing the aggregation. Another possibility is that a component performing data aggregations also needs data from another component or service, which also needs access to the user's pod.

6.2.1 Existing access token delegation mechanisms

Methods for delegating access tokens already exist, even within the OAuth standard. An example is the OAuth On-Behalf-Of flow[2]. This flow supports the delegation of an access token of *Service A* to *Service B*, propagating the access token's bound identity and permissions. This concept is called *access token delegation* because Service B acts on Service A's behalf and in its name. The On-Behalf-Of flow works as illustrated in figure 6.3 (protocol description based on footnote 2).

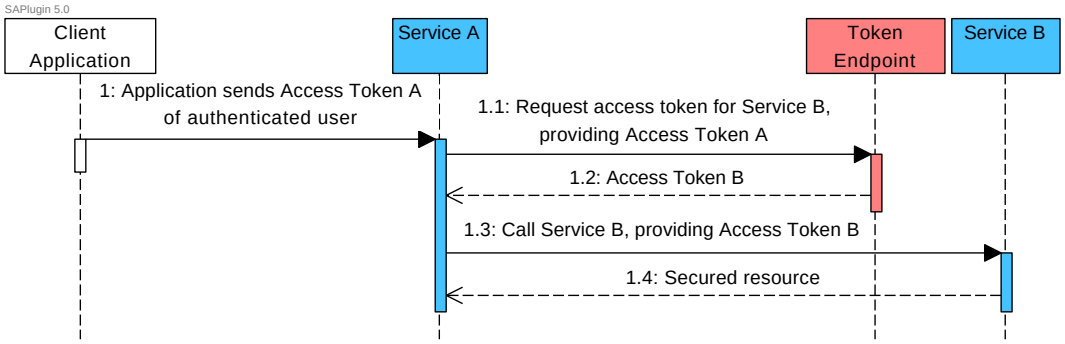


Figure 6.3: OAuth On-Behalf-Of flow

Concretely, the following steps are executed:

1. The client has authenticated the user and has an access token bound to this user's identity (Access Token A)
2. The client sends a request to Service A, providing Access Token A
3. Service A requests an access token for Service B on behalf of itself, providing Access Token A as part of the request
4. The Token Endpoint generates an access token, B, and returns it
5. Service A calls Service B, including Access Token B in the request
6. Service B returns the secured resource

Although this method functionally works, it has a number of important limitations in a decentralized environment. The main limitation is that this flow relies heavily on the Token Endpoint to generate new tokens every time an access token needs to be delegated. This creates a bottleneck when such tokens must often be generated, and makes the system more centralized. The next section will look into a different token system that resolves this issue.

[2]See https://docs.microsoft.com/en-us/azure/active-directory/develop/v2-oauth2-on-behalf-of-flow

6.2.2 Decentralized delegation of access tokens using macaroons

Macaroons have a number of interesting properties that make them a suitable access token for realizing decentralized delegation. Concretely, this decentralized delegation means that a Service A can delegate its access token to Service B, without having to contact the Token Endpoint for obtaining a new token.

Delegation of macaroons works out-of-the-box, because by default macaroons are not bound to a specific client (nor is a proof-of-possession mechanism employed). Such restrictions must be added by adding caveats that restrict the macaroon's usage to a specific application (and a specific user, target resource, etc.). By default, macaroons can also be extended with additional caveats. This enables a service that wishes to delegate its access token to add additional constraints on the token usage (such as making it usable only once, adding a more strict time restriction, etc.).

Since Solid uses OAuth as its authorization mechanism, a new access token mechanism should be compatible with OAuth. Fortunately, macaroons have already been used in production systems as access and refresh tokens. The original macaroons paper discusses macaroons in the context of OAuth (Birgisson et al., 2014, p.12), and the ForgeRock AM 7 Access Management system supports the use of macaroons as access tokens in OAuth[3].

Realizing access delegation naively may lead to severe vulnerabilities, where tokens can be stolen. Therefore, delegation should be disabled by default by limiting a macaroon to a certain WebID (the application's WebID). An application that requests access delegation should include this information when requesting an access token, specifying the WebIDs of the third-party services. Figure 6.4 illustrates the flow of delegating an access token in Solid, in the context of an aggregator.

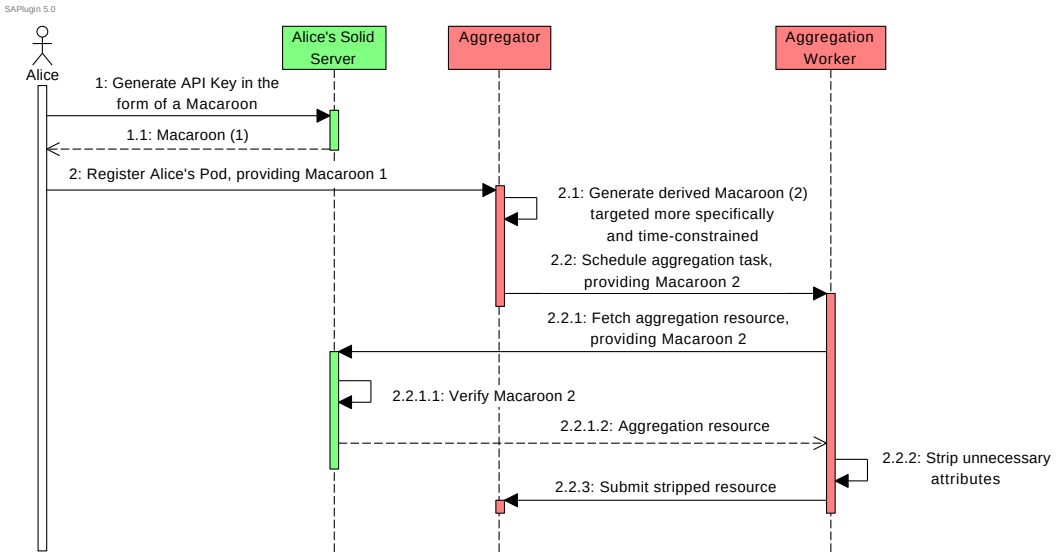


Figure 6.4: Decentralized delegation of macaroon in Solid data aggregator

[3]See https://backstage.forgerock.com/docs/am/7/oauth2-guide/oauth2-macaroons.html

As can be seen in figure 6.4, the `Aggregator` component does not need to contact `Alice's Solid Server` (the token endpoint) to generate a new macaroon to pass to the `AggregationWorker`.

Apart from not having to contact the token endpoint, the usage of macaroons in this context provides additional benefits as well. Firstly, when the `Aggregator` derives a new macaroon to delegate, this can be confined to the concrete request very precisely. For instance, the derived macaroon can be limited to a specific resource (such as `health-data/01042022.json`), to a specific method (such as `GET`), and within a very specific time frame (such as in the next sixty seconds).

Additionally, combining macaroons with the privacy filters proposed in chapter 5 also allows confining the tokens to a specific privacy level. In this way, a Solid server can rewrite the data to strip sensitive personal information even before passing it to the aggregator. This decreases the reliance on the aggregator to correctly perform such operations, increasing the trust level of aggregators. Furthermore, by already stripping a number of attributes in the Solid server, the system load of the aggregator is decreased since it must receive less data.

Finally, macaroons also prove to be a very safe and secure method of handing out API keys. Traditionally, API keys (whether they are in the form of bearer tokens, JWTs or other token types) require either extensive bookkeeping (in the case of bearer tokens) or verifying signatures (in the case of JWTs) to verify whether they are valid. However, none of them offer good anti-theft mechanisms apart from token revocation. Unfortunately, such token thefts can often go unnoticed. Nevertheless, since macaroons can be easily confined to an identifier (such as an IP address), they offer more extensive protection mechanisms. They also don't come with the downside of extensive bookkeeping for bearer tokens (since authorization limitations are embedded in the macaroon), and can be revoked easily by adding a caveat to check the token identifier in a revocation list. Concretely, the content of Macaroon 1 and Macaroon 2 would look like the macaroons presented in table 6.1. Refined caveats are hidden in the derived macaroon for simplicity.

RANDOM_NONCE
location https://pods.org
identifier 3ff99779-720e...
cid user = alice
cid operation = read
cid target = /health
cid expiry = 01-01-2024
cid webid = https://agg.com/...
cid privacy_level = 3
sig ec35a6224861d2f83542c7f2...

(a) Aggregator macaroon

RANDOM_NONCE
location https://pods.org
identifier 3ff99779-720e...
cid user = alice
cid operation = read
cid target = /health/010422.json
cid expiry = 01-06-2022 17:04:02
cid webid = https://agg.com/...
cid privacy_level = 3
cid ip = 193.190.253.145
sig b7da027f072ce6f5a64c35ab...

(b) Derived macaroon

Table 6.1: Illustration of macaroon delegation

Chapter 7

Evaluation

This chapter evaluates the solutions proposed in this dissertation. The first section discusses some theoretical limitations of the proposed solutions. The second section will then benchmark the performance of the developed prototypes. Finally, the last section validates the proposed solutions in the context of the use cases presented in section 3.1.

7.1 Theoretical limitations

This section describes a number of theoretical limitations that are inherent to the proposed solution. This section first discusses the limitations of privacy filters, followed by those of macaroons as access tokens.

7.1.1 Privacy filters

Privacy filters are a mechanism to dynamically rewrite resources to improve privacy. The currently proposed solution comes with a number of limitations, which are listed below.

No write-back Data is modified on a per-request basis, where certain attributes could be changed or removed. In the proposed solution, no mapping of this is kept (which might not even always be possible). As such, when an application modifies this data and tries to write it back, there is no way to reconcile these two versions. In the context of an aggregator that only reads data, this does not pose a problem, but it limits the use of privacy filters in other contexts.

Lack of domain knowledge Often, domain knowledge is needed to construct useful transformations. For example, to strip out the user's name while keeping other names, it is necessary to know the user's name. Similarly, when information is made less specific (for example, replacing "Colruyt" with "supermarket"), it is important to have a mapping of such strings. This means that either users must add some necessary information to the privacy filter configuration files, or that localized

filters (for example, containing information about Belgian supermarkets) should be sourced from somewhere.

Only structured data Since privacy filters, in the proposed solution, work on a per-attribute basis, they do not support performing transformations on unstructured data. For such data, more sophisticated techniques ought to be used (such as machine learning models). Techniques for this already exist (such as Aghasian et al. (2020) and Mehta et al. (2019)), but are complex to integrate and computationally very expensive to run on a per-request basis.

Data linkage Privacy filters only work on a single resource or dataset. However, very often de-anonymization happens because multiple datasets are linked together. Privacy filters only provide limited protection against such information disclosures since resources are transformed on a one-by-one basis without looking at the complete picture. Fetching multiple resources from a single pod may disclose information that should have been filtered out.

7.1.2 Macaroons as access tokens

While macaroons offer a number of advantages as access tokens, they also have shortcomings. This subsection lists some of the most important disadvantages.

Verification requires trust Verification of a macaroon happens by recomputing the signature of the macaroon, and confirming that the calculated signature matches the signature present in the macaroon. The first computed signature is computed based on the random nonce present in the macaroon and the *root key*. As such, this implies that a service that wishes to verify a macaroon must be in possession of the root key (or delegate verification to another service that possesses the root key).

When the target service (the service that receives a request) is running on the same server as the service handing out the macaroon, this does not pose a problem. The Community Solid Server, for example, works like this. However, scenarios were this is not the case are not impossible. For those scenarios, a mechanism must be devised where the verification is either delegated, or the service is trusted and receives the root key.

Caveats are not standardized The original macaroons paper defined what a caveat is and how macaroons are composed, but it did not define a language to express caveats in. Nor did it define in what way macaroons should be serialized for transportation over the internet (such as base64). Due to this limitation, multiple, incompatible libraries exist for using macaroons. Furthermore, since the expression of caveats is not even dependent on the used library, services that require third-party attestations must make sure that caveats are expressed in the same way (since a discharge macaroon may also contain caveats). Therefore, the Solid specification should be extended with definitions of which serialization should be used, and a unified way of representing caveats (for example, in RDF) must be defined. These

decisions have an impact on which existing libraries can be used, or whether custom libraries must be developed.

7.2 Evaluation of the performance

This section discusses a number of benchmarking experiments that have been performed to evaluate the performance of the proposed middleware. The first experiment focuses on the overhead of running privacy filters on top of Solid, comparing the request durations with and without privacy filters. The second experiment measures the performance improvement of using macaroons instead of using DPoP-based tokens by measuring the throughput of token generation and verification. The last experiment focuses on a theoretical improvement in the number of interactions necessary for delegating an access token. The code for the first two experiments can be found at https://github.com/jessegeens/thesis-experiments.

7.2.1 Experiment 1: Overhead of privacy filters

Experiment set-up

The goal of the first experiment is measuring the overhead of running privacy filters on top of Solid. Therefore, a number of requests are sent to the Solid server, comparing the times it takes to fulfill the request between a version with privacy filters (running the privacy-enhancing middleware) and a vanilla version. To provide an overview of the performance overhead that is general enough, the experiments have been run as follows:

1. Every request has been sent 100 times
2. Two types of resources have been requested, financial data (from the finance use case) and exercise data (from the exercise use case)
3. For every resource type, the number of attributes of the resource has been varied from 10 to 10000
4. For a fixed resource, the number of transformations has been varied from 1 to 7

The measured times are in milliseconds, and correspond to the complete end-to-end time (i.e., the time from when the request is sent out at the client to the time when it receives a response). To simulate a realistic environment, the set-up of the experiments is as described in figure 7.1.

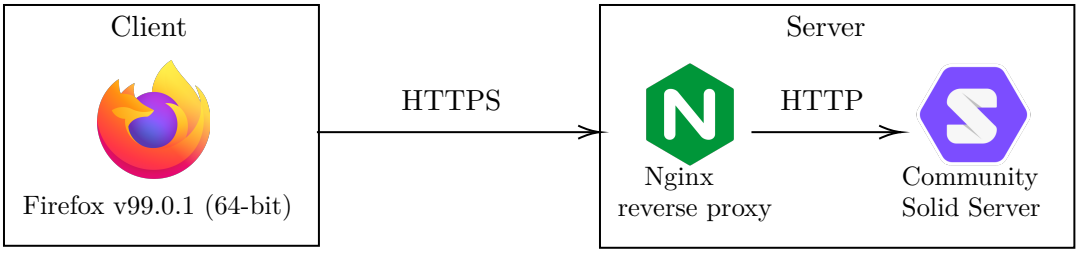


Figure 7.1: Set-up for Privacy Filter performance benchmark

The client runs on a laptop that uses an i5-6200U CPU with 16GB of memory, running on Manjaro Linux v21.2.6. The server runs on a DigitalOcean droplet[1] with 2 vCPUs and 4GB of memory. Since JavaScript is a single-threaded language, 2 CPU cores should be sufficient. The server runs on Ubuntu Server v21.10, which runs the Community Solid Server version 2.0.1 on NodeJS version 12.22.5.

```
{   "to": "HU47063860017040107800278243",
    "to_name": "Bob Langosh",
    "from": "MT47GPBH0064183561Y7J031999I81E",
    "from_name": "Billy Klocko",
    "amount": 653,
    "timestamp": 1607524526257,
    "description": "Illinois yellow networks" }
```

(a) Attribute for transaction data (JSON)

```
<TrackPoint>
    <Time>2020-08-16T00:59:27.969Z</Time>
    <Position>
    <LatitudeDegrees>40.7780135</LatitudeDegrees>
    <LongitudeDegrees>-73.9656795</LongitudeDegrees>
    </Position>
    <DistanceMeters>4671.900000000001</DistanceMeters>
    <HeartRateBpm xsi:type="HeartRateInBeatsPerMinute_t">
    <Value>153</Value>
    </HeartRateBpm>
    <SensorState>Absent</SensorState>
</TrackPoint>
```

(b) Attribute for exercise data (XML)

Figure 7.2: Example data attributes of resources.

[1] See: https://www.digitalocean.com/products/droplets

Two types of resources have been requested, with varying numbers of attributes (from 10 to 10 000). The data contained in these resources has been generated randomly[2], which resulted in files having sizes between 3KB and 3MB (or 900 000 lines, for the largest files). An example of an attribute in both resources is illustrated in figure 7.2.

To study the effect of the number of applied transformations, this number has also been varied. When no variation is done, we opted to use three transformations. Otherwise, when n transformations have been applied, these are the first n of the transformations listed in listing A.4 in the appendix.

Results

The tables below give an overview of the performance results. Table 7.1 handles the performance of the unmodified version of the CSS, while table 7.2 shows the performance results of the modified server. Both tables contain subtables for the different data types that have been requested, and illustrate results for a differing number of attributes. To correctly interpret this data, data between the two tables, within the same column and subtable should be compared. These have similar features (i.e. the same number of attributes and same data type), but have been delivered by the different server versions. For all performed measurements, the average, minimum, maximum, 95^{th} percentile and standard deviation are listed, along with the effective resource size (i.e. the response length).

To graphically illustrate these results, a number of boxplots are also presented in figures 7.3 and 7.4. Figure 7.3 illustrates the effect of varying resource sizes, both for the transaction data in JSON (figure 7.3a) as well as the exercise data in XML (figure 7.3b). Finally, figure 7.4 illustrates the effect of applying a varying number of transformations. When looking at these results, we notice that there is a minimal increase in request durations for resources of up to 1000 attributes (which is around 300KB). However, larger attribute amounts have a detrimental effect on the request duration. These results are discussed after listing the tables containing the results.

[2]The code used for generating this data can be found at https://github.com/jessegeens/data-generator

No. of attributes	**10**	**100**	**1 000**	**10 000**
Response length (bytes)	3 582	32 941	326 464	3 175 105
Average duration (ms)	147	193	307	1 059
Max. duration (ms)	500	518	659	2 305
Min. duration (ms)	83	99	176	782
Std. dev (ms)	80	75	102	222
95^{th} percentile (ms)	277	315	448	1 423

(a) Performance overview of unmodified CSS when requesting transaction data

No. of attributes	**10**	**100**	**1 000**	**10 000**
Response length (bytes)	7 321	61 028	598 887	6 005 836
Average duration (ms)	158	198	350	1 605
Max. duration (ms)	468	541	729	3 309
Min. duration (ms)	77	100	210	1 173
Std. dev (ms)	90	93	105	313
95^{th} percentile (ms)	402	395	582	2 203

(b) Performance overview of unmodified CSS when requesting exercise data

Table 7.1: Overview of performance of unmodified CSS

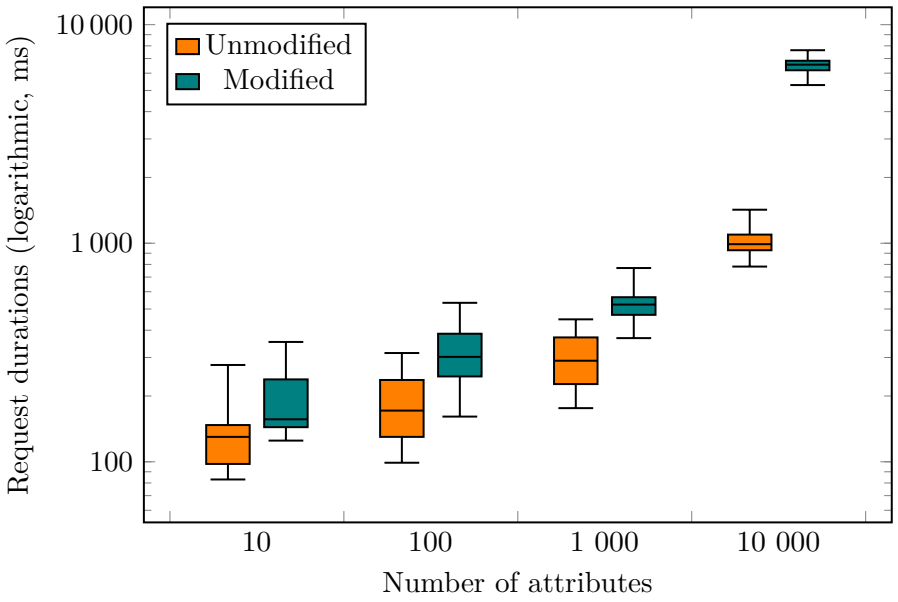
No. of attributes	**10**	**100**	**1 000**	**10 000**
Response length (bytes)	2 525	22 761	224 866	2 159 316
Average duration (ms)	198	328	543	6 596
Max. duration (ms)	647	594	876	8 921
Min. duration (ms)	125	161	368	5 293
Std. dev (ms)	91	114	109	644
95^{th} percentile (ms)	353	534	770	7 641

(a) Performance overview of modified CSS when requesting transaction data

No. of attributes	**10**	**100**	**1 000**	**10 000**
Response length (bytes)	4 735	38 192	373 551	3 755 500
Average duration (ms)	193	346	926	9 244
Max. duration (ms)	487	714	1436	11 798
Min. duration (ms)	119	182	702	8 028
Std. dev (ms)	92	130	142	634
95^{th} percentile (ms)	428	593	1 191	10 347

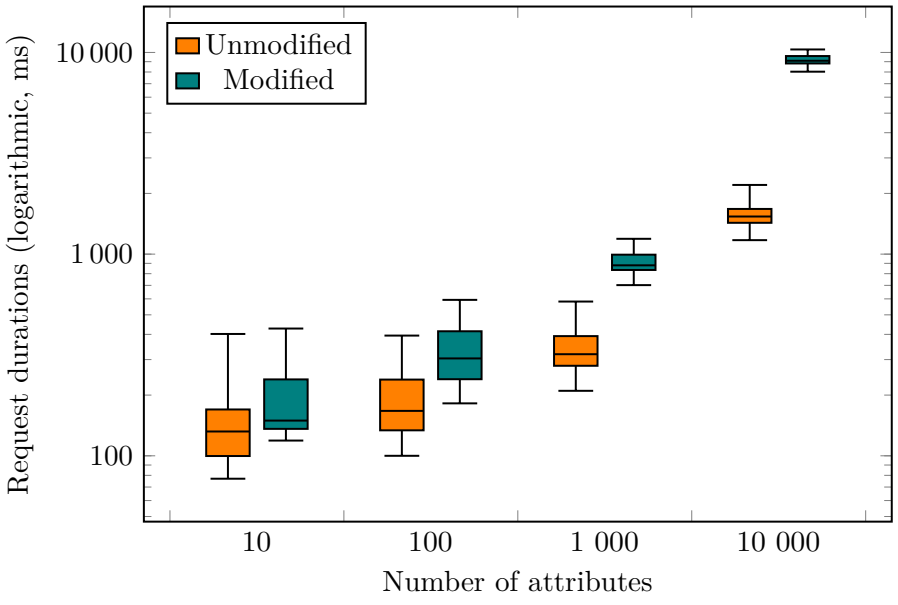
(b) Performance overview of modified CSS when requesting exercise data

Table 7.2: Overview of performance of modified CSS



(a) Transaction data (JSON)

.



(b) Exercise data (XML)

Figure 7.3: Boxplot of data request durations (y-axis, in ms, logarithmic) across variable attribute amounts (x-axis). Max values are 95$^{\text{th}}$ percentile

.

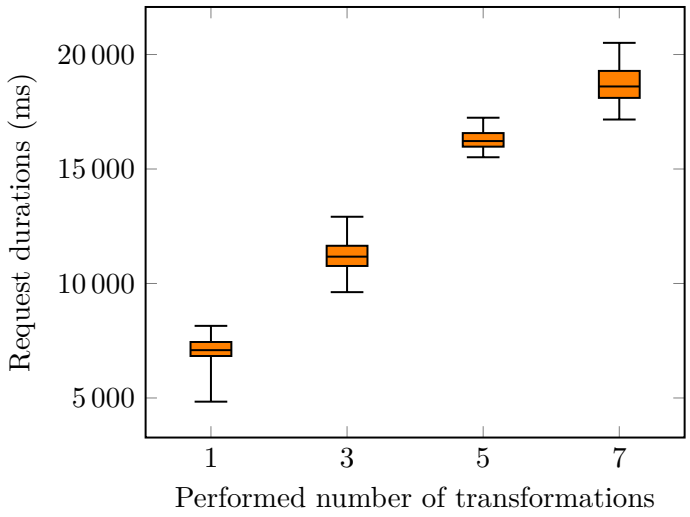


Figure 7.4: Boxplot of request durations (in ms, y-axis) with a different number of transformations performed (x-axis). Max values are 95$^{\text{th}}$ percentile

.

Discussion

As the first part of the experiment demonstrates, the overhead of running privacy filters for smaller resources (up to 500KB) is tolerable. For resources of 300KB, the overhead is around fifty percent in the case of financial data. However, the solution comes with a serious performance overhead for large resources, increasing the request duration with a factor of up to six. The second part of the expirement highlights that increasing the number of performed transformations has a detrimental effect on the performance and there are thus severe limitations on how many transformations can be performed. In some cases this can be a grave weakness, when many transformations are needed (e.g. transforming the name of a large number of supermarkets).

There are multiple explanations for these results. Firstly, a part of this overhead is attributable to a suboptimal implementation of how these filters are applied; for every filter, an iteration over the complete resource is performed. This leads to a time complexity of $\mathcal{O}(nt)$, with t transformations over a resource of n attributes. This limitation is caused by the library that is used in the prototype to perform the transformations, and as such, this issue can be partially resolved by using a more efficient algorithm and data structure. Secondly, this difference in request durations is likely also partially due to the fact that for smaller resources, the transit time of the resource forms a larger part of the duration. For larger resources, this forms only a smaller part of the request duration.

To lessen these limitations, the overhead can also be reduced by using *precomputation*, where transformed files are computed beforehand and stored, so that the transformations need not happen at runtime. However, this comes with a significantly increased memory cost, and should thus only be done for large files that are requested often. Caching strategies may aid here as well, but are not useful for the first request.

7.2.2 Experiment 2: Performance improvement of macaroons compared to DPoP

Experiment set-up

The goal of this experiment is to evaluate the performance gain of using macaroons instead of DPoP as access tokens. Concretely, this is realized by measuring the throughput of token generation and verification (i.e. how many tokens per second can be generated or verified). This metric is different than the one from the last experiment, as here throughput is measured instead of latency. This choice is based on several insights. First, the throughput of token generation is very high (over ten tokens per millisecond), which would cause precision problems when calculating the latency. Secondly, the experiments also aim to evaluate the solutions in the context of different goals. The first experiment concerned mostly the point-of-view of a client, who wishes to receive his resources as soon as possible, and as such, latency is the best metric for this case. However, optimizing token generation concerns the server itself, which wishes to be able to generate as much tokens as possible (i.e. make sure the bottleneck threshold is as high as possible). As such, measuring throughput is more suitable here.

For macaroons, the *macaroons.js*[3] library is used. For generating the DPoP tokens, a modified version[4] of the *dpop*[5] library is used. Verification of DPoP tokens relies on the *jose*[6] library, and follows an implementation similar to what is used in the *solid-access-token-verifier*[7] (but without dereferencing WebIDs).

All tests were performed on the same DigitalOcean droplet as in the previous experiment, namely one with 2 vCPUs and 4GB of RAM. This time, Node version 16.13.1 was used. The experiments measured the amount of tokens generated or verified per second. For every measurement, this was repeated 100 times. In the case of DPoP, a distinction was also made between which cryptographic algorithm was used. Since the *solid-access-token-verifier* library supports both ES256 (ECDSA with SHA256 as hash function) and PS256 (RSASSA-PSS with SHA256)[8], separate tests are run for both of these algorithms. As such, the following throughputs are measured (100 times):

1. Generation of macaroons
2. Generation of DPoP with ES256
3. Generation of DPoP with PS256
4. Verification of macaroons

[3] https://www.npmjs.com/package/macaroons.js
[4] The main modification was making use of the Node Crypto API instead of the WebCrypto API
[5] https://www.npmjs.com/package/dpop
[6] https://www.npmjs.com/package/jose
[7] https://www.npmjs.com/package/@solid/access-token-verifier
[8] Other hashing algorithms from the SHA-family (using 384 or 512 bits) are also supported, but these are not tested

5. Verification of DPoP with ES256
6. Verification of DPoP with PS256

In order to make a fair comparison, the following properties are embedded and validated in all cases:

1. Valid token (i.e. signature check)
2. Token not expired
3. Correct HTTP method used (i.e. the `htm` claim in DPoP tokens)
4. Correct target (i.e. the `htu` claim in DPoP tokens)

Results

Table 7.3 lists the results of the performed experiments. Concretely, the average, maximum, minimum throughput are listed, as well as the 5^{th} percentile results, the upper and lower quartile, and the standard deviation. The throughput is measured in generated and verified tokens/second respectively for tables 7.3a and 7.3b. Figure 7.5 illustrates these results visually in two boxplots. These results indicate that using macaroons can allow large performance benefits. They are discussed in detail in the section following the tables and figures.

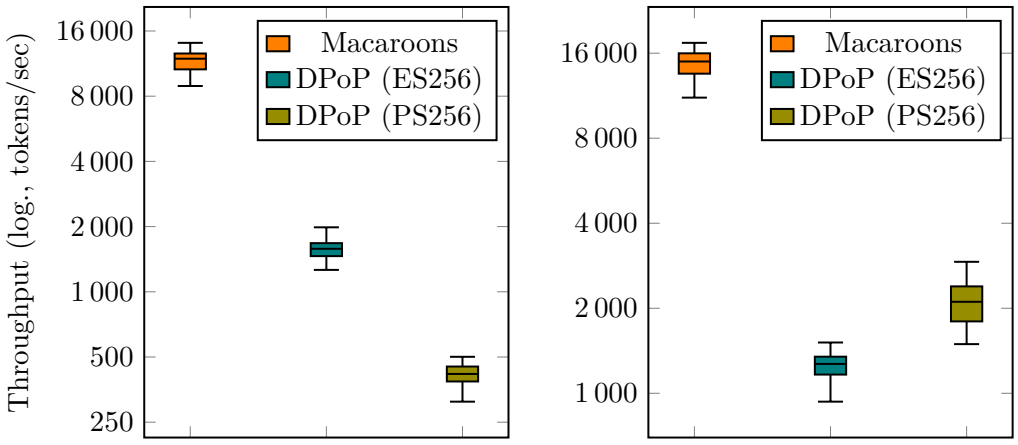
Access token type	**Macaroon**	**DPoP (ES256)**	**DPoP (PS256)**
Average througput	11 633	1 567	412
Max througput	14 105	1 985	501
Min throughput	7 461	1 045	263
Std. dev	1 480	180	54
5^{th} percentile	8 916	1 262	311
Upper quartile	12 595	1 677	452
Lower quartile	10 640	1 461	385

(a) Token generation throughput

Access token type	**Macaroon**	**DPoP (ES256)**	**DPoP (PS256)**
Average througput	14 539	1 244	2 095
Max througput	17 410	1 513	2 921
Min throughput	9 596	878	1 305
Std. dev	1 895	151	369
5^{th} percentile	11 138	934	1 493
Upper quartile	15 986	1 348	2 390
Lower quartile	13 540	1 165	1 797

(b) Token verification throughput

Table 7.3: Overview of token generation and verification throughput in tokens/second



(a) Boxplot of token generation throughput (b) Boxplot of token verification throughput

Figure 7.5: Boxplots of token throughput. Y-axes are logarithmic, min values are 5^{th} percentile

Discussion

These experiments demonstrate that macaroons are a much more efficient type of access token, both when it comes to generation as well as verification. Comparing macaroons with DPoP using ES256, which is the most common algorithm, shows that there is a more than sevenfold increase in throughput for token generation. For token verification, the increase is more than elevenfold. Comparing macaroons with DPoP using the PS256 algorithm yields similar results, but reversed for verification and generation. Furthermore, the experiments illustrate that the choice of cryptographic algorithm depends strongly on whether the generating or verifying device is more resource-constrained. When we assume that the generating device is more resource-constrained (as we did in the use cases, e.g. when it is an IoT device), ES256 seems to be the algorithm that is best suited. This result validates our assumption that macaroons are a much more efficient access token type. Looking at the resource usage graphs provided by DigitalOcean, the CPU seems to be the bottleneck.

7.2.3 Experiment 3: Improvement in interaction amount necessary for access token delegation

In this experiment, we look at how macaroons perform theoretically in the context of their decentralized token delegation. This is done by taking a look at how many interactions are necessary to delegate an access token. We consider a token endpoint τ, a service α and a service β. In both scenarios, α wants to delegate the token it received from τ to β.

Calculation

Let δ be the cost of an interaction (which includes, a.o., the cost of the network) between α and τ, and η the cost of an interaction between α and β. Finally, we also assume that the generation of a new token has a certain cost π associated with it. For a DPoP token we let this cost be π_1, for a macaroon this cost is assumed to be π_2. Based on the results of experiment 2, we can estimate the relation between π_1 and π_2.

DPoP delegation In the case of token delegation with DPoP tokens, we become the following cost c_{t1}:

1. α requests token from τ $(c = \delta)$
2. τ generates DPoP token $(c = \pi_1)$
3. τ replies with token $(c = \delta)$
4. α requests second (to-delegate) token from τ $(c = \delta)$
5. τ generates second DPoP token $(c = \pi_1)$
6. τ replies with token $(c = \delta)$
7. α requests a resource from β, includes the second token $(c = \eta)$
8. β replies with the resource $(c = \eta)$

This comes to a total cost $c_{t1} = \sum c$ of $4\delta + 2\pi_1 + 2\eta$.

Macaroons delegation In the case of token delegation with macaroons, we become the following cost c_{t2}:

1. α requests token from τ $(c = \delta)$
2. τ generates macaroon $(c = \pi_2)$
3. τ replies with token $(c = \delta)$
4. α generates derived token $(c = \pi_2)$
5. α requests a resource from β, includes the second macaroon $(c = \eta)$
6. β replies with the resource $(c = \eta)$

This comes to a total cost $c_{t2} = \sum c = 2\delta + 2\pi_1 + 2\eta$, which makes a difference of $\Delta_c = c_{t1} - c_{t2} = 2(\delta + (\pi_1 - \pi_2))$ between the two types of access tokens.

Discussion

The above calculation has demonstrated that there are gains as long as the condition $\pi_1 - \pi_2 > -\delta$ is fulfilled. However, experiment 2 has demonstrated that $\pi_1 > \pi_2$ (by a factor of ~ 7). We can thus confidently conclude that there are also theoretical gains in the number of interactions necessary to perform token delegation. These improvements are most prevalent when the cost δ of a connection between a service α and a token endpoint τ are large. This can be the case, for example, when a token endpoint is under heavy load, or when it is located far from the service.

7.3 Validation of use cases

This section discusses whether the proposed solution fits all the criteria laid out by the use cases described in section 3.1.

7.3.1 Exercise data

The exercise data use case described a scenario wherein a user, who stores her exercise reports in her pod, wanted to export some of this data to form a scoreboard with her friends. By leveraging privacy filters, the heart rate of her activity data can be filtered out so she can safely share her data with an aggregator. Embedding the correct privacy level in the issued macaroon, which functions as an API key, ensures that incorrect settings cannot lead to leaking this data. Furthermore, the data necessary for constructing this overview can be stored in a group vault, such that all users have easy access to it. This can be realized with the use of macaroons as a token mechanism.

7.3.2 Personal finance

In the personal finance use case, a user wants to get a general overview of his spending without revealing the exact details such as exact amounts, store locations, IBAN number, etc. This can be realized using privacy filters as well: timestamps and amounts can be changed with the *perturbation* tactic. The IBAN number can be replaced with a pseudonym. Finally, also the store names can be changed while still keeping the correct category, albeit using a strategy that is a bit more complicated. Individual stores can be given a pseudonym (e.g. replacing "Carrefour" with "Grocery store") by using the *pseudonym* tactic combined with the *equalsCondition* field (which only applies the tactic if the specified condition is fulfilled). Since this spending data can be very large, decentralized delegation allows spreading the load over different worker nodes without overloading the token endpoints.

7.3.3 Aggregated view on personal health data streams

The final use case was based on a challenge in the context of SolidLab, and pertains to a dashboard that highlights aggregated personal health data, such as in the context of caretakers for elderly people. The SolidLab challenge listed a number

of requirements, some of which have been fulfilled. Since this dissertation focuses mostly on the privacy and scalability aspect of aggregation, other requirements in the challenge such as streaming capabilities have been ignored. However, MASS seems to be an acceptable proof-of-concept for the other criteria if a small number of modifications are added. Right now, the requested resource is re-parsed every time an update happens. It could be possible to keep track of which part of the resource is already transformed, and then only perform transformations on the updated part. This does not come with any problems as the transformations happen on a record-by-record basis and are stateless. Given this small modification, MASS allows efficient updates of the aggregation; since on the Solid server only a part of a resource needs to be parsed, and a lot of unnecessary data can already be stripped away before it reaches the aggregator. The last criterion, stopping personal information from leaking to the aggregator, can also be fulfilled given that the privacy filters have been correctly set up. Then, all PII can be stripped away before being transmitted.

7.4 Evaluation of requirements

Section 3.3 listed a number of functional and non-functional requirements. This section will discuss whether the proposed solution has managed to fulfill these requirements.

7.4.1 Functional requirements

Four functional requirements were stipulated. The first one was *flexibility for the supported data scheme*. This requirement has been fulfilled by leveraging an abstraction layer over the concrete data schemes, which allows configuration files to be written for every possible data scheme. The second requirement was *automatically select PETs*. This requirement has also been fulfilled, partially by the configuration files. These configuration files specify transformations in the context of a certain privacy level, which brings forth granularity in the applied transformations. Users can then configure a privacy level based on which application is requesting the data, or which data type is being requested. Based on this contextual information, the middleware selects the correct privacy levels, which maps to a corresponding set of transformations. The third requirement, then, was *extensibility*. This requirement has been realized by making the middleware a part of the Community Solid Server. In this way, modifications to follow updates to the Solid specification are automatically integrated by updating the server dependency. Furthermore, by leveraging the CSS's component injection, the middleware can easily be extended with more functionality, such as additional parsers for new content representations. Finally, the last functional requirement that was listed was *decentralized access token delegation*. This requirement has been attained by introducing macaroons as an access token mechanism, which support decentralized delegation out of the box.

7.4.2 Non-functional requirements

Subsequently, we take a look at which non-functional requirements have been fulfilled. Since non-functional requirements behave more like properties than features of a system, this is the place where trade-offs often have to be made. The first non-functional requirement that was listed was *intuitive to use*. This requirement has been partially fulfilled by making use of privacy levels, where concrete transformations are abstracted away. The developed middleware is only a prototype, however, so settings have to be made in a configuration file instead of through an intuitive user interface, but this can easily be resolved. Then, a description of what would happen to data of a certain data type under a certain privacy level could be given to make this more intuitive. In that sense, the middleware is intuitive to use. The second requirement was *testability*, which has also only been partially attained. By developing the middleware as an extension of the existing CSS, which makes use of a dependency injection system, allows components to be tested more easily. In addition, the CSS's test suite can be leveraged. However, the developed prototype does not come with additional tests. The final non-functional requirement that was listed was *minimal overhead*. For privacy filters, this requirement has not been completely fulfilled as these come with a significant overhead, especially when caching is disabled and when transformations are applied to large resources. For the access token system, the requirement has been fulfilled, as macaroons are more efficient than DPoP tokens.

7.5 Conclusion of evaluation

The evaluation has demonstrated that privacy filters are a feasible solution for smaller resources. Furthermore, it has also highlighted that macaroons as a novel access token in Solid provide large performance benefits, along with a number of other advantages. It has, however, also demonstrated that a number of limitations are present in the solution, mainly regarding the performance of privacy filters on large resources. Nevertheless, mechanisms exist for lessening this burden, such as making use of caches for transformed resources. Of course, this evaluation only considered the technical aspects of the solution. Privacy filters may provide a technical solution to restricting data leakage to untrusted applications, but it does not solve the problem of how to determine which applications can be trusted. For solving such a complex challenge, additional research will be necessary.

Chapter 8

Conclusion

8.1 Overview

The research question this dissertation aimed to answer, posed in section 1.2, asks whether we can design a scalable middleware solution aimed at facilitating data aggregation in a secure and protected manner in a setting where data is distributed across various parties. To answer this question, the MASS middleware has been developed in the context of this dissertation. Three use cases have been employed to guide the design of the middleware, from which concrete requirements were also derived. To guarantee the privacy-aspect of the middleware, a number of privacy-enhancing technologies have been analyzed in this context. This analysis has led to various possible future research directions, which are discussed in the next section.

The middleware introduces two novel components, privacy filters and a new access token system. Privacy filters are a mechanism to dynamically rewrite resources based on contextual information from the request, to reduce attribute leakage to untrusted applications. Privacy filters provide granularity in the provided level of privacy by introducing privacy levels, which map the requested level of privacy to a concrete set of privacy-enhancing transformations. In addition, this dissertation suggests using macaroons as a novel access token mechanism. Macaroons support decentralized delegation, which is a crucial part of a scalable aggregation system. Furthermore, macaroons are also more efficient to generate and verify than the currently used DPoP token mechanism. Lastly, macaroons support third-party attestations, which is a useful property for enabling group vaults.

The middleware solution proposed in this dissertation has also been thoroughly evaluated. This evaluation has demonstrated that the laid out requirements have, for the most part, been fulfilled. In particular, the middleware supports automatically selecting the right PETs across different data schemes. Implementing privacy filters as an extension of the CSS has made the middleware also very extensible. However, there are limitations on the performance of privacy filters for large resources, and the testability requirement has not been completely fulfilled as the developed prototype does not come with an additional test suite.

Additionally, the evaluation has demonstrated that macaroons bring a number

of performance improvements. These improvements are both practical, with an increased throughput of token generation and verification, as well as theoretical, with a decreased interaction cost necessary for delegating access tokens. Concretely, the throughput of generating tokens compared to DPoP with the ES256 algorithm has increased more than seven-fold by using macaroons, while the throughput of verifying the tokens increased more than eleven-fold. The cost of interactions for delegating an access token has decreased by a factor $2(\delta + (\pi_1 - \pi_2))$, where δ is the cost of an interaction between a service and the token endpoint, and $\pi_1 - \pi_2$ is the performance difference between DPoP and macaroons for generating access tokens.

8.2 Future work

FW 1 — Homomorphic encryption in Solid Homomorphic encryption is an encryption technology that allows for mathematical operations to be performed on the ciphertext. It was discussed in section 4.3. This makes it an interesting technology to make the Solid specification more privacy-friendly and robust against data leaks when data aggregations are performed. For example, a possible use case could be an application that collects salaries from a number of pods, and then gives back an encrypted answer to each pod containing the average salary. In this way, interesting insights can be gained without sacrificing personal information. However, this method also lacks somewhat in the need for a centralized party that performs the aggregation — it is not fully decentralized. Whether this is problematic depends heavily on the concrete use case. Additionally, some coordination is needed between the pods beforehand to make sure that the necessary data is encrypted in the same manner. Future research could investigate a potential implementation of a secure data aggregator for Solid based on homomorphic encryption schemes.

FW 2 — MPC in Solid Secure Multi-Party Computation is a cryptographic technique that allows for securely performing computations on data, without sharing this data between parties. While at first glance this might seem similar to FW 1, there are some major differences. Firstly, in the case of the homomorphic encryption, there was a single central party that performed the computation. In the case of MPC, though, this computation is entirely decentralized and happens on the parties themselves; data is never shared or exposed. This has many advantages as the MPC scenario clearly fits better in the decentralized nature of Solid. Nevertheless, there are disadvantages as well. In the homomorphic encryption scenario only few modifications to the Solid protocol would be required. The bulk of the work would happen in the party that performed the aggregation by requesting encrypted data from multiple pods. However, as MPC does not share any data and computations happen locally, an implementation of MPC in Solid would require many additions to the Solid protocol. Developing an architecture and improved specification for Solid could thus be interesting future research, with many possible applications.

FW 3 — Privacy levels Currently, this dissertation has proposed a number of so-called *privacy levels* (see section 5.1) which form a granular way to determine how much data may be handed over to an untrusted application. However, this was only a practical proposal lacking a rigorous definition. Future research could investigate possible ways to define privacy levels more rigorously, for example by finding some sort of leakage metric that determines the maximum leakage of data under a certain privacy level.

FW 4 — Attribute-Based Encryption Section 3.4 highlighted an honest-but-curious adversary model where applications are not completely trusted. However, an attacker model where the roles are reversed is also possible. This would be a scenario where the applications are highly trusted, but the Solid pod is stored at a third party and there is no way to verify that there are no vulnerabilities in the server source code. As such, this implies an active attacker that can deviate from the protocol and actively tries to bypass the Solid server's authentication and authorization mechanisms. Effectively, this means that the attacker can access resources to which it should not have access according to the ACLs as these resources can be stored on-disk. Should an attacker breach the server (for exampling, getting SSH access in some way), then he can read all resources stored on the server.

A good solution to this problem would be encryption, which would obscure the data stored in the Solid pod, and only parties with a correct key could access the data. However, in the decentralized context of Solid, this is hard to achieve. Multiple applications need to access the same data, and access policies for resources can be complex. In addition, these access policies must also be dynamic, meaning that it must be possible for a user to withdraw an application's access to a resource. Using traditional encryption methods will quickly lead to decidedly complex key management. By mapping applications to attributes and using Ciphertext-Policy Attribute-Based Encryption to encrypt resources with an access policy, this problem may be solved. As such, researching CP-ABE for securely storing data on Solid pods and sharing it with applications may be useful future research.

Appendix A

MASS configuration and code

A.1 Privacy level mapping

In our example data transformations, four levels of increasing privacy are proposed, based partially on the GDPR definition of sensitive personal data and on the types of identifiers defined in section 4.1.2. However, it is important to note that this is only a practical choice and forms a trade-off between granularity and necessary work (for setting up all the transformations required for a certain level). Privacy experts may have more valuable opinions on the correct number of privacy levels and how these should be communicated to a user. Future research could help to find a rigorous definition for these privacy levels, for example based on some leakage metric. This is elaborated in FW 3.

Level 1: all data No data transformations are applied, all data is passed on to the requesting application.

Level 2: Removal of sensitive personal data Sensitive personal data, as defined by the GDPR (European Commission, 2016), is removed from the dataset. This includes data consisting of racial or ethnic origin, political opinions, religious or philosophical beliefs, etc. Thus, the tactic *Remove* is applied to all data elements matching this definition.

Level 3: Pseudonymization/generalisation of direct identifiers Since level 3 is a stronger version of level 2, sensitive personal data is removed first. Additionally, direct personal identifiers are pseudonymized or generalized/perturbed. Concretely, the tactics *Pseudonym*, *Encrypt*, *Perturbation*, ... may be applied here, depending on the specific data attribute. Examples are the replacement of names by placeholders, the perturbation or adding a placeholder for birth dates (such that the exact date is obscured, but the age is still correct), the removal of street names and numbers while keeping larger geographic areas such as cities, etc. This makes the data still relatively accurate, while direct identification of the user is made impossible.

Level 4: Pseudonymization/generalisation of (in)direct identifiers In addition to direct identifiers, also indirect identifiers are now modified or removed. The same PETs and tactics are used, but are now applied more strictly and to more data attributes. For example, when perturbing birth dates, now the exact age is not kept exactly, but it is changed to a range within the exact age. Cities may also be perturbed when possible, but keeping for example the province or state. Other indirect identifiers such as genders may also be modified or removed.

A.2 Privacy filter JSON Schema

```
{
  "$schema":"https://json-schema.org/draft-07/schema",
  "$id":"https://jesse.geens.cloud/privacyfilter.schema.json",
  "title":"Schema for specifying privacy filters on detectable data schemes",
  "type":"object",
  "properties":{
    "schemeName":{
      "description":"The unique name of the data scheme",
      "type":"string"
    },
    "detector":{
      "description":"Describes how the data scheme should be recognized",
      "type":"object",
      "properties":{
        "contentRepresentation":{
          "type":"string",
          "description":"The content-representation of the input data. E.g. XML or JSON.",
          "enum":[ "xml", "json", "ttl" ]
        },
        "scheme":{
          "type":"string",
          "description":"The name of data scheme being detected"
        },
        "mechanism":{
          "type":"object",
          "description":"How the data scheme should be detected",
          "properties":{
            "mechanismName":{
              "type":"string",
              "enum":[ "filenameExact", "filenameContains",
                "bodyContains", "containernameExact" ]
            },
            "value":{
              "type":"string",
              "description":"The value on which the detection is based"
            }
          },
          "required":[ "mechanismName", "value" ]
        }
```

```
40        },
41        "required":[ "contentRepresentation",  "mechanism" ]
42      },
43      "transformations":{
44        "type":"array",
45        "items":{
46          "type":"object",
47          "properties":{
48            "level":{
49              "type":"integer",
50              "minimum":1,
51              "maximum":4
52            },
53            "tactics":{
54              "type":"array",
55              "items":{ "$ref":"#/$defs/PET" }
56            }
57          },
58          "required":[ "level", "tactics" ],
59          "additionalProperties":false
60        }
61      }
62    },
63    "required":[  "schemeName", "detector", "transformations" ],
64    "$defs":{
65      "PET":{
66        "type":"object",
67        "properties":{
68          "field":{
69            "type":"string"
70          },
71          "transformation":{
72            "type":"object",
73            "properties":{
74              "transformationName":{
75                "type":"string",
76                "enum":[ "remove", "pseudonym",  "perturbation",
77                  "hash", "encrypted", "random" ]
78              },
79              "pseudonym":{
80                "type":"string",
81                "description":"What to replace the field with"
82              },
83              "equalsCondition":{
84                "type":"array",
```

```
85                "description":"Only perform this transformation if the
                  ↪  value is in the array"
86              },
87              "perturbationFactor":{
88                "type":"number",
89                "description":"The maximum relative difference between
                  ↪  the perturbed value and the orginial value."
90              }
91            },
92            "required":[ "transformationName" ],
93            "allOf":[
94              {
95                "if":{
96                  "properties":{
97                    "transformationName":{ "const":"pseudonymization" }
98                  }
99                },
100               "then":{ "required":[ "pseudonym" ] }
101             },
102             {
103               "if":{
104                 "properties":{
105                   "transformationName":{ "const":"perturbation" }
106                 }
107               },
108               "then":{ "required":[ "perturbationFactor" ] }
109             }
110           ]
111         },
112         "fieldType":{
113           "type":"string",
114           "enum":[ "string", "integer", "float", "boolean" ]
115         }
116       },
117       "required":[ "field", "transformation", "fieldType" ],
118       "additionalProperties":false
119     }
120   }
121 }
```

Listing A.1: JSON Schema describing the lay-out of privacy filter configuration files

A.3 Example Privacy Filter: KBC Transaction Data

```
{
    "schemeName": "KBCTransactionData",
     "detector": {
         "contentRepresentation": "json",
         "mechanism": {
             "mechanismName": "containernameExact",
             "value": "kbc-bank"
         } },
     "transformations": [
         {
             "level": 1,
             "tactics": []
         }, {
        "level": 4,
        "tactics": [ {
                "field": "accountOwner",
                "fieldType": "string",
                "transformation": {
                    "transformationName": "pseudonym",
                    "pseudonym": "J.G."
                }
            }, {
                "field": "IBAN",
                "fieldType": "string",
                "transformation": {
                    "transformationName": "pseudonym",
                    "pseudonym": "BE42429824824354"
                }
            }, {
                "field": "saldo",
                "fieldType": "float",
                "transformation": {
                    "transformationName": "perturbation",
                    "perturbationFactor": 1.1
                }
            }, {
                "field": "$.history[*].from",
                "fieldType": "string",
                "transformation": {
                    "transformationName": "pseudonym",
                    "pseudonym": "BE42429824824354",
                    "equalsCondition": ["BE66123456783456"]
                }
```

```
            }, {
                "field": "$.history[*].to",
                "fieldType": "string",
                "transformation": {
                    "transformationName": "pseudonym",
                    "pseudonym": "BE42429824824354",
                    "equalsCondition": ["BE66123456783456"]
                }
            }, {
                "field": "$.history[*].from_name",
                "fieldType": "string",
                "transformation": {
                    "transformationName": "pseudonym",
                    "pseudonym": "J.G.",
                    "equalsCondition": ["Jesse Geens"]
                }
            }, {
                "field": "$.history[*].to_name",
                "fieldType": "string",
                "transformation": {
                    "transformationName": "pseudonym",
                    "pseudonym": "J.G.",
                    "equalsCondition": ["Jesse Geens"]
                }
            }, {
                "field": "$.history[*].description",
                "fieldType": "string",
                "transformation": {
                    "transformationName": "random",
                    "equalsCondition": ["Ontduiking belastingen"]
                }
            }, {
                "field": "$.history[*].amount",
                "fieldType": "float",
                "transformation": {
                    "transformationName": "perturbation",
                    "perturbationFactor": 10
                } }
        ] }
    ]
}
```

Listing A.2: Example configuration file for a privacy filter in JSON, for the KBC Transaction Data datascheme. Fields are specified in JSONPath.

A.4 Dependency injection configuration for CSS

```
{
"@context": [
  "https://linkedsoftwaredependencies.org/bundles/npm/pepsa-component/...",
  "https://linkedsoftwaredependencies.org/bundles/npm/@solid/community-server/...
  {
    "pepsa-component": "urn:pepsa-component:default:",
    "solid-server": "urn:solid-server:default:"
  }
],
"import": [
  "files-scs:config/app/main/default.json",
  "... (other imports here)"
],
"@graph": [
  {
    "comment": "The main entry point into the main Solid behaviour.",
    "@id": "urn:solid-server:default:LdpHandler",
    "@type": "ParsingHttpHandler",
    "args_requestParser": { "@id": "urn:solid-server:default:RequestParser" },
    "args_metadataCollector": { "@id": "urn:solid-server:default:OperationMetadataCollector" },
    "args_errorHandler": { "@id": "urn:solid-server:default:ErrorHandler" },
    "args_responseWriter": { "@id": "urn:solid-server:default:ResponseWriter" },
    "args_operationHandler": {
      "@type": "AnonymizingHttpHandler",
      "args_credentialsExtractor": { "@id": "urn:solid-server:default:CredentialsExtractor" },
      "args_modesExtractor": { "@id": "urn:solid-server:default:ModesExtractor" },
      "args_permissionReader": { "@id": "urn:solid-server:default:PermissionReader" },
      "args_authorizer": { "@id": "urn:solid-server:default:Authorizer" },
      "args_operationHandler": { "@id": "urn:solid-server:default:OperationHandler" },
      "args_dataTreatmentHandler": {
        "@id": "urn:pepsa-component:default:DataTreatmentHandler",
        "@type": "DataTreatmentHandler",
        "configMgr": { "@id": "urn:pepsa-component:default:ConfigurationManager" },
```

```
32          "tacticEncapsulator": {
33            "@id": "urn:pepsa-component:default:TacticEncapsulator",
34            "@type": "TacticEncapsulator",
35            "configMgr": { "@id":
              ↪  "urn:pepsa-component:default:ConfigurationManager" }
36          },
37          "parserSelector": {
38            "@id": "urn:pepsa-component:default:ParserSelector",
39            "@type": "ParserSelector",
40            "parsers": [
41              {
42                "@id": "urn:pepsa-component:default:JSONParser",
43                "@type": "JSONParser"
44              }, {
45                "@id": "urn:pepsa-component:default:XMLParser",
46                "@type": "XMLParser"
47              }, {
48                "@id": "urn:pepsa-component:default:TurtleParser",
49                "@type": "TurtleParser"
50              }
51            ]
52          }
53        },
54        "args_configurationManager": {
55          "@id": "urn:pepsa-component:default:ConfigurationManager",
56          "@type": "ConfigurationManager",
57          "rootScheme": "../../config/schema/root.json",
58          "detectionScheme": "../../config/schema/detection.json",
59          "schemeRules": "../../config/scheme_rules",
60          "ups": {
61            "@id": "urn:pepsa-component:default:UserPreferenceStore",
62            "@type": "UserPreferenceStore",
63            "rs": {
64              "@id": "urn:solid-server:default:ResourceStore_Backend"
65            }
66          }
67        }
68      }
69    }
70  ]}
```

Listing A.3: JSON-LD specifying dependency injection of MASS into CSS

A.5 Transformations performed in evaluation

```
{  "tactics": [ {
        "field":"$.history[*].amount",
        "fieldType":"float",
        "transformation":{
           "transformationName":"perturbation",
           "perturbationFactor":10 }
    }, {
        "field":"IBAN",
        "fieldType":"string",
        "transformation": {
           "transformationName":"pseudonym",
           "pseudonym":"BE42429824824354" }
    }, {
        "field":"$.history[*].timestamp",
        "fieldType":"float",
        "transformation":{ "transformationName":"remove" }
    }, {
        "field":"$.history[*].description",
        "fieldType":"string",
        "transformation":{ "transformationName":"random" }
    }, {
        "field":"$.history[*].from",
        "fieldType":"string",
        "transformation":{
           "transformationName":"pseudonym",
           "pseudonym":"NewName",
           "equalsCondition":"Jesse Geens" }
    }, {
        "field":"currency",
        "fieldType":"string",
        "transformation":{ "transformationName":"random" }
    }, {
        "field":"from",
        "fieldType":"string",
        "transformation":{
           "transformationName":"pseudonym",
           "pseudonym":"NewName",
           "equalsCondition":"Jesse Geens" }
    } ] }
```

Listing A.4: Transformations performed in evaluation

A.6 Community Solid Server architecture

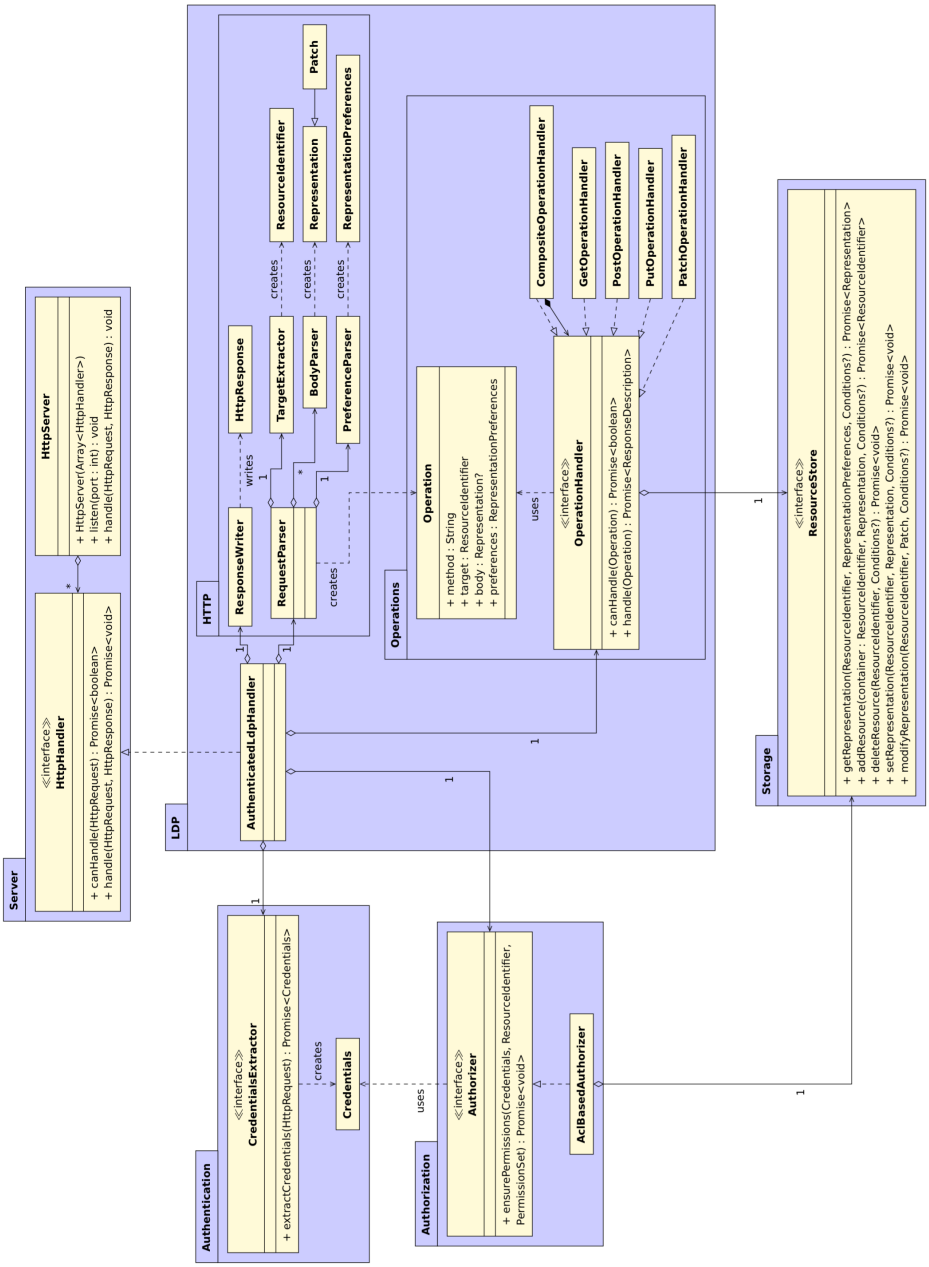


Figure A.1: Overview of the Community Solid Server architecture, by Verborgh (2020)

Bibliography

Erfan Aghasian, Saurabh Garg, and James Montgomery. An automated model to score the privacy of unstructured information-social media case. *Computers & Security*, 92:101778, 2020. ISSN 0167-4048. doi: https://doi.org/10.1016/j.cose.2020.101778. URL https://www.sciencedirect.com/science/article/pii/S0167404820300638.

Venky Anant, Lisa Donchak, James Kaplan, and Henning Soller. The consumer-data opportunity and the privacy imperative. *McKinsey*, apr 2020. URL https://www.mckinsey.com/business-functions/risk-and-resilience/our-insights/the-consumer-data-opportunity-and-the-privacy-imperative.

Tim Berners-Lee, James Hendler, and Ora Lassila. The semantic web. *Scientific American*, 284(5):34–43, 2001. ISSN 00368733, 19467087. URL http://www.jstor.org/stable/26059207.

John Bethencourt, Amit Sahai, and Brent Waters. Ciphertext-policy attribute-based encryption. In *2007 IEEE Symposium on Security and Privacy (SP '07)*, pages 321–334, 2007. doi: 10.1109/SP.2007.11.

Arnar Birgisson, Joe Gibbs Politz, Ulfar Erlingsson, Ankur Taly, Michael Vrable, and Mark Lentczner. Macaroons: Cookies with contextual caveats for decentralized authorization in the cloud. In *Network and Distributed System Security Symposium*, 2014.

Dan Boneh, Kevin Lewi, and David Wu. Order-revealing encryption. Stanford University, 2016. URL https://crypto.stanford.edu/ore/. Accessed 2022-02-27.

John Borking, P. Verhaar, B.M.A. Eck, P. Siepel, G.W. Blarkom, R. Coolen, M. Uyl, J. Holleman, P. Bison, R. Veer, J. Giezen, Andrew Patrick, C. Holmes, J.C.A. Lubbe, Roy Lachman, S. Kenny, Randy Song, K. Cartrysse, J. Huizenga, and X. Zhou. *Handbook of Privacy and Privacy-Enhancing Technologies The case of Intelligent Software Agents*. CBP (Dutch Data Protection Authority), 11 2003. ISBN ISBN 90 74087 33 7. doi: 10.13140/2.1.4888.7688.

Ran Canetti and Amir Herzberg. Maintaining security in the presence of transient faults. volume 839, pages 425–438, 08 1994. ISBN 978-3-540-58333-2. doi: 10.1007/3-540-48658-5_38.

Sarven Capadisli. Web access control. W3C editor's draft, W3C, September 2021. URL https://solid.github.io/web-access-control-spec.

Sarven Capadisli, Tim Berners-Lee, Ruben Verborgh, Kjetil Kjernsmo, Justing Bingham, and Dmitri Zagidulin. Solid protocol. W3C editor's draft, W3C, October 2021. URL https://solidproject.org/TR/protocol.

Pierre-Antoine Champin, Gregg Kellogg, and Dave Longley. JSON-ld 1.1. W3C recommendation, W3C, July 2020. https://www.w3.org/TR/2020/REC-json-ld11-20200716/.

Cynthia Dwork. Differential privacy. pages 1–12, 06 2006. ISBN 978-3-540-35907-4. doi: 10.1007/11787006_1.

Taher ElGamal. A public key cryptosystem and a signature scheme based on discrete logarithms. *IEEE Transactions on Information Theory*, 31(4):469–472, 1985. doi: 10.1109/TIT.1985.1057074.

European Commission. EU GENERAL DATA PROTECTION REGULATION (GDPR): REGULATION (EU) 2016/679 OF THE EUROPEAN PARLIAMENT AND OF THE COUNCIL of 27 april 2016 on the protection of natural persons with regard to the processing of personal data and on the free movement of such data, and repealing directive 95/46/ec (general data protection regulation). *Official Journal of the European Union 2016*, L 119/1, 04 2016. ISSN 1977-0677. URL https://eur-lex.europa.eu/eli/reg/2016/679/oj.

David Evans, Vladimir Kolesnikov, and Mike Rosulek. *A Pragmatic Introduction to Secure Multi-Party Computation*, volume 2. Now Publishers Inc., Hanover, MA, USA, dec 2018. doi: 10.1561/3300000019. URL https://doi-org.kuleuven.e-bronnen.be/10.1561/3300000019.

Daniel Fett, Brian Campbell, John Bradley, Torsten Lodderstedt, Michael Jones, and David Waite. Oauth 2.0 demonstrating proof-of-possession at the application layer (dpop). Internet-Draft draft-ietf-oauth-dpop-03, IETF Secretariat, April 2021. URL https://www.ietf.org/archive/id/draft-ietf-oauth-dpop-03.txt. https://www.ietf.org/archive/id/draft-ietf-oauth-dpop-03.txt.

Benjamin Fuller, Mayank Varia, Arkady Yerukhimovich, Emily Shen, Ariel Hamlin, Vijay Gadepally, Richard Shay, John Darby Mitchell, and Robert K. Cunningham. Sok: Cryptographically protected database search. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 172–191, 2017. doi: 10.1109/SP.2017.10.

Craig Gentry. *A Fully Homomorphic Encryption Scheme*. PhD thesis, Stanford, CA, USA, 2009. AAI3382729.

O. Goldreich, S. Micali, and A. Wigderson. How to play any mental game. In *Proceedings of the Nineteenth Annual ACM Symposium on Theory of Computing*, STOC '87, pages 218–229, New York, NY, USA, 1987. Association for Computing

Machinery. ISBN 0897912217. doi: 10.1145/28395.28420. URL https://doi-org.kuleuven.e-bronnen.be/10.1145/28395.28420.

Vipul Goyal, Omkant Pandey, Amit Sahai, and Brent Waters. Attribute-based encryption for fine-grained access control of encrypted data. In *Proceedings of the 13th ACM Conference on Computer and Communications Security*, CCS '06, pages 89–98, New York, NY, USA, 2006. Association for Computing Machinery. ISBN 1595935185. doi: 10.1145/1180405.1180418. URL https://doi-org.kuleuven.e-bronnen.be/10.1145/1180405.1180418.

D. Hardt. The oauth 2.0 authorization framework. RFC 6749, RFC Editor, October 2012. URL http://www.rfc-editor.org/rfc/rfc6749.txt. http://www.rfc-editor.org/rfc/rfc6749.txt.

Emad Heydari Beni, Bert Lagaisse, Wouter Joosen, Abdelrahaman Aly, and Michael Brackx. Datablinder: A distributed data protection middleware supporting search and computation on encrypted data. In *Proceedings of the 20th International Middleware Conference Industrial Track*, Middleware '19, pages 50–57, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450370417. doi: 10.1145/3366626.3368132. URL https://doi-org.kuleuven.e-bronnen.be/10.1145/3366626.3368132.

Jaap-Henk Hoepman. Privacy design strategies. 10 2012. ISBN 978-3-642-55414-8. doi: 10.1007/978-3-642-55415-5_38.

Jan Holvast. History of privacy. volume 298, pages 13–42, 07 2009. ISBN 9780444516084. doi: 10.1007/978-3-642-03315-5_2.

Marc Jarsulic. Using antitrust law to address the market power of platform monopolies. *Center for American Progress*, jul 2020. URL https://www.americanprogress.org/article/using-antitrust-law-address-market-power-platform-monopolies/.

M. Jones, J. Bradley, and N. Sakimura. Json web token (jwt). RFC 7519, RFC Editor, May 2015. URL http://www.rfc-editor.org/rfc/rfc7519.txt. http://www.rfc-editor.org/rfc/rfc7519.txt.

Roland Legrand. Groen licht voor vlaamse 'intelligente verkeerswisselaar van data'. *De Tijd*, nov 2021a. URL https://www.tijd.be/de-tijd-vooruit/tech/Groen-licht-voor-Vlaams-Datanutsbedrijf-en-zelfbeheerde-identiteit/10349559. In Dutch.

Roland Legrand. Digita exporteert vlaamse technologie naar zweden. *De Tijd*, nov 2021b. URL https://www.tijd.be/de-tijd-vooruit/tech/digita-exporteert-vlaamse-technologie-naar-zweden/10349255.html. In Dutch.

Yehuda Lindell. Secure multiparty computation. *Commun. ACM*, 64(1):86–96, December 2020. ISSN 0001-0782. doi: 10.1145/3387108. URL https://doi-org.kuleuven.e-bronnen.be/10.1145/3387108.

Brijesh Mehta, Udai Pratap Rao, Ruchika Gupta, and Mauro Conti. Towards privacy preserving unstructured big data publishing. *Journal of Intelligent & Fuzzy Systems*, 36:3471–3482, 2019. ISSN 1875-8967. doi: 10.3233/JIFS-181231. URL https://doi.org/10.3233/JIFS-181231.

Nandana Mihindukulasooriya and Roger Menday. Linked data platform 1.0 primer. W3C note, W3C, April 2015. URL https://www.w3.org/TR/2015/NOTE-ldp-primer-20150423/.

Jackson Morgan, Aaron Coburn, and Matthieu Bosquet. Solid oidc primer. W3C editor's draft, W3C, January 2022. URL https://solid.github.io/solid-oidc/primer/.

Michael Naehrig, Kristin Lauter, and Vinod Vaikuntanathan. Can homomorphic encryption be practical? In *Proceedings of the 3rd ACM Workshop on Cloud Computing Security Workshop*, CCSW '11, pages 113–124, New York, NY, USA, 2011. Association for Computing Machinery. ISBN 9781450310048. doi: 10.1145/2046660.2046682. URL https://doi-org.kuleuven.e-bronnen.be/10.1145/2046660.2046682.

Shota Nakatani, Sachio Saiki, Masahide Nakamura, and Kiyoshi Yasuda. Generating personalized virtual agent in speech dialogue system for people with dementia. In *HCI*, 2018. URL http://www27.cs.kobe-u.ac.jp/achieve/data/pdf/1285.pdf.

Gregory Nelson. Practical implications of sharing data: A primer on data privacy, anonymization, and de-identification. 04 2015.

Claudio Orlandi. Is multiparty computation any good in practice? In *2011 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 5848–5851, 2011. doi: 10.1109/ICASSP.2011.5947691.

Pascal Paillier. Public-key cryptosystems based on composite degree residuosity classes. In Jacques Stern, editor, *Advances in Cryptology — EUROCRYPT '99*, pages 223–238, Berlin, Heidelberg, 1999. Springer Berlin Heidelberg. ISBN 978-3-540-48910-8.

Pedro Pisa, Michel Abdalla, and Otto Carlos M. B. Duarte. Somewhat homomorphic encryption scheme for arithmetic operations on large integers. pages 1–8, 12 2012. ISBN 978-1-4673-5217-8. doi: 10.1109/GIIS.2012.6466769.

Eric Prud'hommeaux and Gavin Carothers. RDF 1.1 turtle. W3C recommendation, W3C, February 2014. URL https://www.w3.org/TR/2014/REC-turtle-20140225/.

Ansar Rafique, Dimitri Van Landuyt, Emad Heydari Beni, Bert Lagaisse, and Wouter Joosen. Cryptdice: Distributed data protection system for secure cloud data storage and computation. *Information Systems*, 96:101671, 2021. ISSN

0306-4379. doi: https://doi.org/10.1016/j.is.2020.101671. URL https://www.sciencedirect.com/science/article/pii/S0306437920301289.

Tom Relihan. Will regulating big tech stifle innovation? *MIT Sloan*, sep 2018. URL https://mitsloan.mit.edu/ideas-made-to-matter/will-regulating-big-tech-stifle-innovation.

Amit Sahai and Brent Waters. Fuzzy identity-based encryption. In Ronald Cramer, editor, *Advances in Cryptology – EUROCRYPT 2005*, pages 457–473, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg. ISBN 978-3-540-32055-5.

N. Sakimura, J. Bradley, M. Jones, B. de Medeiros, and C. Mortimore. Openid connect core 1.0 incorporating errata set 1. Technical report, OpenID Foundation, nov 2014. URL http://openid.net/specs/openid-connect-core-1_0.html.

N. Sakimura, J. Bradley, and N. Agarwal. Proof key for code exchange by oauth public clients. RFC 7636, RFC Editor, September 2015.

Andrei Sambra, Henry Story, and Tim Berners-Lee. Webid 1.0. W3C editor's draft, W3C, March 2014. URL https://www.w3.org/2005/Incubator/webid/spec/identity/.

Sandvine. The mobile internet phenomena report. https://www.sandvine.com/hubfs/Sandvine_Redesign_2019/Downloads/2021/Phenomena/MIPR%20Q1%202021%2020210510.pdf, May 2021.

Manish Singh. Web creator tim berners-lee's startup inrupt raises $30 million. *TechCrunch*, dec 2021. URL https://techcrunch.com/2021/12/09/tim-berners-lee-inrupt-fundraise/.

Steve Speicher, Ashok Malhotra, and John Arwe. Linked data platform 1.0. W3C recommendation, W3C, February 2015. URL https://www.w3.org/TR/2015/REC-ldp-20150226/.

Latanya Sweeney. Simple demographics often identify people uniquely. *Health*, 671, 01 2000.

Latanya Sweeney. K-anonymity: A model for protecting privacy. *Int. J. Uncertain. Fuzziness Knowl.-Based Syst.*, 10(5):557–570, oct 2002. ISSN 0218-4885. doi: 10.1142/S0218488502001648. URL https://doi-org.kuleuven.e-bronnen.be/10.1142/S0218488502001648.

Dimitri Van Landuyt, Vicky Vanluyten, Oleksandr Tomashchuk, and Wouter Joosen. A survey of architectural tactics for de-identification of personally identifiable information. 2021.

Ruben Verborgh. Solid server - selected architectural diagrams v1.3.0. https://rubenverborgh.github.io/solid-server-architecture/solid-architecture-v1-3-0.pdf, aug 2020.

Sophia Yakoubov. A gentle introduction to yao's garbled circuits. *Preprint on webpage at https://web.mit.edu/sonka89/www/papers/2017ygc.pdf*, 2017.

Andrew Chi-Chih Yao. How to generate and exchange secrets. In *27th Annual Symposium on Foundations of Computer Science (sfcs 1986)*, pages 162–167, 1986. doi: 10.1109/SFCS.1986.25.

Fiche masterproef

Student: Jesse Geens

Titel: Secure and protected data aggregation in the decentralized web

Nederlandse titel: Veilige en beschermde gegevensaggregatie in het gedecentraliseerde web

UDC: 621.3

Korte inhoud:

Solid is een W3C specificatie die tracht het web te decentraliseren met behulp van datakluizen. Datakluizen zijn gedecentraliseerde opslagplaatsen waar gegevens voor verschillende toepassingen worden bijgehouden. Een uitdaging hierbij is veilige en beschermde gegevensaggregatie over verschillende datakluizen heen. Dergelijke aggregaties brengen privacyrisico's met zich mee, net als schaalbaarheidsproblemen. Deze thesis introduceert een middleware die tracht deze problemen gedeeltelijk (op het niveau van de server) op te lossen, dankzij het introduceren van privacy filters en een nieuw mechanisme voor toegangstokens dat gedecentraliseerde tokenafvaardiging ondersteunt. Privacy filters staan toe om meer granulariteit te bereiken in de privacy van gedeelde bronnen. De middleware selecteer hierbij automatisch transformaties die uitgevoerd worden wanneer een bron verzocht wordt, op basis van een aantal contextuele parameters. Om vervolgens een tokenmechanisme te bekomen dat gedecentraliseerde tokenafvaardiging ondersteunt onderzoekt deze thesis het gebruik van macaroons binnen Solid. Macaroons ondersteunen niet alleen gedecentraliseerde tokenafvaardiging, maar zijn ook efficiënter om te genereren en verifiëren. Bovendien laten ze attesten van derde partijen toe, wat een nuttige eigenschap is voor datakluizen die gedeeld worden door een groep gebruikers. De middleware die deze thesis voorstelt werd geëvalueerd op grond van drie gebruiksscenario's. Voor privacy filters hebben experimenten aangetoond dat de overhead van de middleware toelaatbaar is voor kleinere bronnen met een overhead van ongeveer 50%. Echter, grotere bronnen (rond de 3MB) hebben een overhead die een vijfvoud wordt van de oorspronkelijke duur van het verzoek. Prestatietests die werden uitgevoerd op het genereren en verifiëren van macaroons hebben aangetoond dat de doorvoer hiervan bij macaroons respectievelijk zeven en elf maal groter is dan het DPoP (ES256) systeem. Ten slotte zijn er ook theoretische winsten voor gedecentraliseerde tokenafvaardiging.

Thesis voorgedragen tot het behalen van de graad van Master of Science in de ingenieurswetenschappen: computerwetenschappen, hoofdoptie Gedistribueerde systemen

Promotoren: Prof. dr. ir. Wouter Joosen
Dr. Bert Lagaisse
Prof. dr. Vincent Naessens

Assessoren: Prof. dr. Bart Jacobs
Dr. Emad Heydari Beni

Begeleiders: Dr. Emad Heydari Beni
Ir. Kristof Jannes