Trustworthy Remote Signing

Accessible Electronic Signatures for Everybody

Bachelor Thesis by Gabor Tanz and Patrick Hirt

In this thesis, we propose a way to reduce the trust required when outsourcing electronic signatures to third party services. We compare our proposal to an existing standard by the Adobe-led Cloud Signature Consortium, and we implement it in a proof-of-concept.

Degree course:	Computer Science
Authors:	Gabor Tanz, Patrick Hirt
Tutors:	Prof. Dr. Annett Laube, Prof. Gerhard Hassenstein
Experts:	Dr. Andreas Spichiger
Date:	16th January 2020

Contents

Abstract

To enable a future digital single market, the European Union has proposed a standard for Remote Digital Signing. Remote Signing means that cryptographic operations are outsourced to a trust service provider, including the storage of the private keys. This is done to make digital signing more user-friendly, and to remove the burden of key management from the end-user. However, this means that effectively, end users have less control over their private keys. Remote signing services implementing the EU standard could sign arbitrary documents in their users' names, without their authorisation, and without them even noticing.

In this thesis, we expand upon the EU's standard by designing a remote signing service in such a way that it cannot sign any document without the users' direct authorisation, despite having control over the users' private keys. Contrary to proposals involving secret sharing schemes, with our design, no client-side software is necessary, except for a web browser.

To demonstrate our solution, we create a proof-of-concept implementation consisting of the signing service itself as well as a platform-independent verification program capable of both online and offline verification.

Acknowledgements

First and foremost, we would like to thank both of our tutors, Prof. Dr. Annett Laube and Prof. Gerhard Hassenstein, for their guidance and continuous support during our work.

We further thank Dr. Andreas Spichiger for supervising our thesis in the role of the expert.

Finally, we would like to thank the countless women and men all over this world who contribute to open source software. Our work would not have been possible without them.

Summary

The core problem of Remote Signing Services is that the Trust Service Provider (TSP) has control over the user's private key. The industry standard solution for key security is to employ a Hardware Security Module (HSM), which provides a secure enclave for the private keys. No one is supposed to be able to access the keys stored in such a HSM, not even the owners of the HSM.

But even if such a HSM is trustworthy, how is the signer supposed to securely access their own key stored in a remote HSM, to approve a specific signature operation? EIDAS [39] tries to solve this by requiring the implementation of a Signature Activation Protocol (SAP). The SAP should provide secure authorisation from the user's device through the signing service and to the HSM, activating the key in the HSM and generating a signature. EIDAS calls this authorisation information Signature Activation Data (SAD).

In practice, this works by the user providing authentication credentials (username/password, with an optional One Time Password (OTP)). These credentials are verified by the HSM before allowing use of the signing key, and it is required to only allow the signing of the documents the users subsequently submit to it, and no others.

The problem with this approach is that there is no accountability to this process. Users are expected to trust that the HSM properly validates their credentials and only then allows use of the signing key.

This is where we come in.

We propose a solution which makes remote signing services as standardised by the European Union (EU) more secure, and more trustworthy, by giving users back a part of the control they lost when they gave up ownership of their private keys.

With our solution, a TSP cannot sign a document in the user's name, despite being in possession of the user's private key. Users can now enjoy the usability advantages of Remote Signing, on any device, anywhere, and be freed of the burden of key management, without giving up control over what is signed in their name.

We have achieved this by separating authentication and signing, and placing these two concerns into the hands of separate organisations. The trust required is now distributed over two parties, and any one of them acting alone cannot create a valid signature.

We believe we have proposed the first real solution to the remote key activation problem, and our solution is user-verifiable, that is, the users of our solution are able to verify that the signing service indeed did use the key they entrusted it with to sign the documents they intended to sign, even after the signature's been created.

We have achieved this by incorporating the hashes of the documents-to-be-signed into the authentication process, by making them part of a nonce value used during the OpenID Connect (OIDC) authentication. The identity assertion subsequently issued by the Identity Provider (IDP) contains that same nonce value, but now protected by a digital signature issued by the IDP. This way, a secure link between the document the user intends to sign, the identification of the user, and the confirmed intent of the user to sign that document is established.

Then, based upon that identity assertion, the signing server issues the signature. By making the identity assertion - containing the document hashes - part of the signature file, anyone can verify that the signing server indeed signed the specific hashes intended by the user and nothing else.

If the signing server were to attempt to issue a signature without the IDP, it would be discovered during signature verification, because the document hashes wouldn't match the nonce value in the identity assertion. This way, we ensure that a rogue signing server cannot issue valid signatures, despite being in control of the users' private keys.

Part I.

Scope and Objectives

1. Introduction

Today, we use a number of computing devices interchangeably on a daily basis: a desktop workstation at the office, a laptop computer on the move, a tablet in the living room and of course, always by our side, the smartphone. In an increasingly cloudified and mobile world our expectation is to be able to do our work all the same, regardless of the computing device we use, or where we are.

We start editing a text document in Google Docs on our desktop workstation at the office, work on it a bit more on our laptop while travelling by train, and proofread it later on the smartphone. This device- and location-independent way of working has become the standard in recent years, and users have started to expect it from their IT sevices.

It's difficult to meet this expectation with the way electronic signatures are usually created today, using certificates stored on smartcards, plugged into a laptop, using a specialised card reader and accompanying software. It's annoying and inconvenient having to carry around cables and adaptors, and a lot can go wrong: a random operating system update breaking driver compatibility with the card reader, for example, leaving us dead in the water. If we want to make this easier on the user and to drive usage of electronic signatures and even make them mainstream, we have to do better.

At the root of this inconvenience is the requirement that the user keep their private key physically with them, stored in a manner making it difficult for anyone to steal it: on a smartcard. Any IT professional knows full well this demand isn't made from users in order to annoy them but because it is - more or less - the only practical *and* secure way to have users store their private key.

So-called Remote Signing Services aim to eliminate the need for people to carry their private key with them, and to locally create signatures, in the hope for improved ease of use, and eventually, greater adoption of digitally signing documents. However, allowing someone (the signing service, in this case) to be able to sign documents in place of the user introduces a number of serious security and confidentiality problems.

In this thesis, we analyse and address these problems, and we implement the proposed solutions in a fully functional Remote Signing Service, thereby showing that they work in the real world and not just on paper.

We will allow people to create electronic signatures, no matter where they are, or what device they're using, in a secure manner. Building on our previous work of Projekt 2 [42], we show how it is possible to securely integrate OIDC authentication with remote digital signatures. We expand upon this previous work and show how it is possible to have a remote signing service with the capability of signing on the users' behalf without the need for completely trusting that service. Furthermore, we compare our solutions to those proposed by an industrial consortium led by Adobe Inc., and we show in which ways we believe our approach to be superior.

2. Objectives

2.1. Main Objective

The main objective of this work is the finalisation of the concept we began in Projekt 2, and its implementation.

The implementation consists of the remote signing service itself in the form of a Representational State Transfer (REST) Application Programming Interface (API), and a cross-platform frontend authenticating the users through a trusted OIDC IDP. This frontend uses the REST API for signing the users' files. On top of that, it offers offline verification of existing signatures on desktop operating systems.

2.2. Previous Works

We build upon our previous work of Projekt 2 [42], where we specified the authentication process for qualified signatures, non-qualified batch signatures, the signature file format, as well as - to our knowledge - pioneering the secure integration of a digital signature with an OIDC ID token without requiring any change to the IDP. The main benefit of Projekt 2 for us is that it allowed us to familiarise ourselves with the topic of Remote Signing.

2.3. Backend

The signing server is the centrepiece of the service, where the actual signatures are being created. To that end, it generates signing keys, requests the Certificate Authority (CA) to sign them, builds the signature file, embeds certificate chain as well as revocation information, and adds at least one timestamp. It depends on the IDP for authenticating its users.

2.4. Frontend

The frontend must be cross-platform, where cross-platform means supporting the desktop operating systems GNU/Linux, Microsoft Windows and Apple MacOS as well as the mobile phone operating systems Google Android and Apple iOS. The frontend must support authentication through the IDP, creating signatures through the backend, as well as verifying them. Verification must be available online as well as offline, except for the mobile version, where offline verification is not required.

2.5. Comparison With Existing Solutions

The Cloud Signature Consortium standardised a remote signing service compliant with EIDAS regulation. Adobe has made an implementation of this standard. We will examine the Cloud Signature Consortium (CSC) specification and compare it with our solution, with a focus on security.

2.6. Evaluation of the Yubikey HSM for the Signing Service

We will evaluate the Yubikey HSM 2, and whether it would increase security if we were to integrate it into our solution.

3. Actors

Actors specify a role played by a user or a system for the purposes of a clearer definition. In this chapter, we will outline the actors in our system. For a graphical representation of the actors and their interactions, please see figure 3.1.

3.1. Big Picture

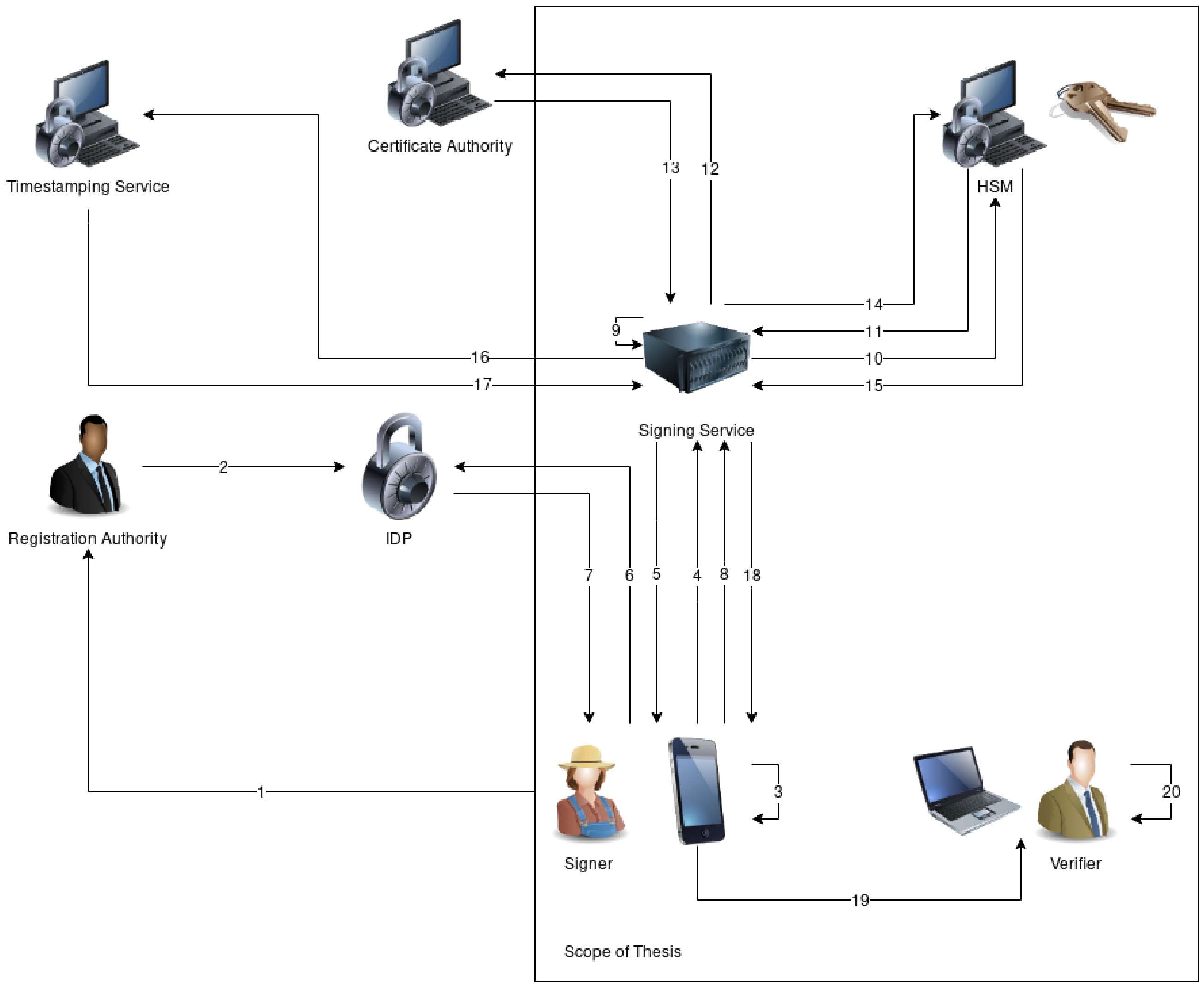


Figure 3.1.: The Actors in our system, and the scope of our thesis

The following subsections provide a description of the interactions as numbered in figure 3.1.

3.1.1. Registration

1. Registration of identity with the Registration Authority (RA) (Authenticator)

   A pre-requisite for using our signing service is that the user (Verifier) is registered with an IDP trusted by the Signing Service. The user registers at the RA by proving their identity to it. In most cases, this process needs to be completed only once per user.

2. Propagate identity to IDP (Identifier)

   After the identity has been confirmed, the RA propagates this fact to the IDP. From this point on the user is known to the IDP and the user may authenticate themselves to third parties with it. (The RA and the IDP may be the same entity.)

3.1.2. Identification

3. Generate document hash

   The user (Verifier) generates the hashes of the document(s) they wish to sign. This is needed before the identification step in order to prevent signing of arbitrary documents by securely binding the document hashes to the authentication request and subsequent identity assertion, as we've shown in our previous work [42].

4. Send hash to signing service

   The user sends the hashes of the documents they generated in the previous step to the signing service.

5. Receive OIDC redirect to IDP

   The signing service constructs OIDC redirects for all configured IDPs, and returns them to the user. These redirects contain a strong cryptographic link to the hashes submitted in the previous step.

6. Login to IDP

   The user follows the redirect they received in the previous step and authenticates with the IDP in order for it to confirm their identity.

7. Receive ID token

   After successful authentication, the user receives an authentication token from the IDP, signed by the IDP, as proof of their identity.

8. Send ID token to signing service

   The user sends the ID token from the previous step to the signing service.

9. Verify ID token

   The signing service validates the ID token (JSON Web Token (JWT)) and checks its contents, in order to confirm whether the secure linking of the user's intent to sign specific documents with the authentication is valid.

3.1.3. Signature Generation

10. Request signing key Certificate Signing Request (CSR) from HSM

    The signing service requests the HSM to generate a new signing key in the name of the user.

11. Receive CSR

    The HSM generates this new signing key and returns the Certificate Signing Request (CSR).

12. Send CSR to CA

    The signing service sends the CSR to the CA.

13. Receive signed certificate

    The CA signs the CSR and returns the signed certificate to the signing service.

14. Request signature from HSM

    The signing service sends the data to be signed, which includes the document hashes, to the HSM.

15. Receive signature

    The HSM signs the data, using the signing key generated in step 10, and returns the signature to the signing service.

16. Request timestamp from Timestamping Service (TSS)

    The signing service sends the hash value of the signature data to the TSS.

17. Receive timestamp

    The TSS adds a timestamp to the signature and signs it, and returns the timestamp along with its signature to the signing service.

18. Send signature to signer

    The signing service constructs the final signature file containing at least the signed hash, the signed timestamp, the ID token and the chains of all used certificates as well as revocation information to the user.

Steps 16 and 17 correspond exactly to Request For Comments (RFC) 3161 [24].

3.1.4. Signature Verification

19. Send document and signature to receiver (Verifier)

    The user sends the document and the signature to the intended receiver.

20. Verify document and signature

    The receiver uses the validation tool to validate the document and signature.

3.2. The Signer

The Signer is the person who wishes to have the signing server sign one or more documents in their name. For example, a medical professional issuing a prescription for medication to a patient.

3.3. The Verifier

The Verifier is the person who wishes to verify the integrity of a document and its signature. For example, the pharmacist whom the patient gives the prescription to (as previously signed by The Signer as specified in section 3.2) in order to purchase the medication prescribed by the medical professional.

3.4. The Authenticator

The Authenticator is the system who authenticates The Signer as specified in section 3.2. In National Institute of Standards and Technology (NIST) terminology [51], this is the entity establishing the Identity Assurance Level (IAL). In order for The Authenticator to be able to authenticate The Signer, they must have been registered with The Authenticator by The Identifier as specified in section 3.5.

3.5. The Identifier

The Identifier is the system or person who asserts the identity of The Signer. In order for the signing service to issue qualified signatures as defined by Swiss Federal Assembly legislation [50] and Swiss Federal Council regulations [56], the identity must be proven in-person using a government-issued photographic identification document such as a passport.

3.6. The Signing Service

The Signing Service is the system who actually creates the signatures on behalf of the user. It generates the signing keys and requests the CA (actor 3.7) to sign them.

3.7. The Certificate Authority

The CA is the system who signs the signing keys generated by the Signing Service (actor 3.6). The CA offers Certificate Revocation List (CRL) and Online Certificate Status Protocol (OCSP) responders.

4. Functional Requirements

4.1. Terminology

The usual modal verbs are to be interpreted as in RFC 2119 [53].

4.1.1. Practically Impossible

"Practically impossible" means the probability of it being possible is not zero, but so small for it not to matter in practice. An example for this would be finding the prime factors of the product of two carefully chosen 1024 bit numbers within 24 hours.

4.1.2. Being made difficult

"Made difficult" means something far from impossible for someone with near-unlimited resources like a state actor, but extremely difficult if not impossible even for a highly skilled single person with the resources expected for a single person. For example, stealing a smart card from someone and misusing the contained private key.

4.2. Signature Requirements

4.2.1. Authenticity

It must be practically impossible for anyone to forge a signature without it being detected upon signature verification.

4.2.2. Integrity

It must be practically impossible for anyone to modify a signed document without being detected upon signature verification. A secure hash algorithm must be used for hashing the document. Secure means the algorithm to be pre-image as well as collision-resistant as validated by the NIST Cryptographic Algorithm Validation Program [18].

4.2.3. Verifiability

Anyone must be able to verify the authenticity of a signature and the integrity of the signed document.

4.2.4. Non-repudiation

It must be practically impossible for anyone to deny having signed a document.

4.2.5. Long-Term Validation

Signatures must be suitable for Long-Term Validation (LTV) using RFC3161 [24] timestamps.

4.2.6. Secure Coupling of Authentication and Signature

It must be practically impossible for anyone to abuse a stolen OIDC ID token to sign a document other than intended by The Signer.

4.2.7. Authentication Protocol

Standard OIDC must be used for authenticating The Signer as specified in the standard [46].

4.2.8. Supported File Formats

It must be possible to sign any file, regardless of its format.

4.2.9. Bulk Signatures

For qualified signatures, it must be possible to sign more than one document at once.

For advanced signatures, it may be possible to sign several documents one after the other without requiring re-authentication.

4.2.10. Device-local Hashing of Documents

In order to ensure privacy and protection of information as required by 5.0.2, documents to be signed must not leave the users' device. For webinterfaces, this means that the document must be hashed in the browser itself.

4.3. Signature Server Requiremenents

4.3.1. Signing Key Security

Technical measures must be taken to make it difficult to steal the private keys generated on behalf of the users.

4.3.2. No unauthorised identity delegation

It must be practically impossible for the signing server to create a signature on its own.

4.3.3. Random Number Generation

The Random Number Generator (RNG) used for generating signatures must be cryptographically secure.

4.3.4. REST API

The Signature Server must offer a REST API that can be used by third parties to interface with the signing service, for example in order to implement custom frontends or to include it as part of their product, or for users that don't like Graphical User Interface (GUI)s.

5. Non-Functional Requirements

5.0.1. Efficient Signature File Format

The file format for the signature file shall be based on our previous work [42].

5.0.2. Protection of Information

Information not strictly required by the party in order to fulfil their function must not be disclosed to the aforementioned party. In particular, the document to be signed must not be disclosed to the signing server nor to the IDP. The IDP must not learn of the document hash. More generally, every actor must not have any more information disclosed to it than is necessary for them to perform their function.

5.0.3. Offline Validation

The Verifier must be able to verify signatures without an active internet connection using a desktop or laptop computer running GNU/Linux, MacOS or Windows.

5.0.4. Code Quality

The code produced should be, wherever possible:

- Readable
- Well-formatted according to the recommended community standards of the language
- Compileable without any errors nor warnings with the compiler at its strictest setting
- Covered by unit tests

Public APIs should be documented.

5.0.5. Ease of Use

In order to ensure usability conforms to a minimum standard, the following requirements should be fulfilled:

- Unneccessary steps or clicks should be minimised. The minimal amount of user interaction should be strived for for any given user-facing action or use case.
- The user interface should be so simple that non-IT people can use it. Specialised jargon should be avoided.

5.0.6. Reactive Design

The user interface should be useable both on mobile devices (smartphones) as well as desktop devices (laptops). Useable means that the user isn't required to zoom around on a mobile device because the User Interface (UI) is layouted with desktop operating systems in mind alone, nor should a desktop user be presented with a tiny rectangle because the UI was designed for smartphones only. This requirement isn't about the UI looking pretty but about it being useable without being annoying on both form factors.

5.1. IDP Requirements

Anything related to the IDP is out of scope for our thesis, except for specifying what we require of the same. We assume to be using an existing, OIDC-conforming IDP providing the required registration and authentication levels.

5.1.1. Support for OIDC

The IDP must support standard OIDC as specified in the standard [46].

5.1.2. Levels of Assurance

The IDP must support Authenticator Assurance Level (AAL) 2 authentication for advanced signatures, and AAL 3 authentication for qualified signatures as specified in the NIST publication [51].

5.2. Prioritisation of Requirements

Requirement	**Prioritisation**
Authenticity of signature (4.2.1)	Required
Integrity of document (4.2.2)	Required
Verifiability of signature (4.2.3)	Required
Non-repudiation (4.2.4)	Required
Secure coupling of authentication and signature (4.2.6)	Required
Authentication protocol (4.2.7)	Required
Supported file formats (4.2.8)	Required
No unauthorised identity delegation (4.3.2)	Required
Random number generation (4.3.3)	Required
REST API (4.3.4)	Required
Offline validation (5.0.3)	Required
Signing key security (4.3.1)	Optional
Protection of information (5.0.2)	Optional
Device-local hashing of documents (4.2.10)	Optional
Efficient signature file format (5.0.1)	Optional
Bulk signatures (4.2.6)	Optional
Long-term validation (4.2.5)	Optional
Code Quality (5.0.4)	Optional
Ease of Use (5.0.5)	Optional
Reactive Design (5.0.6)	Optional

Table 5.1.: Prioritisation of Requirements

6. Use Cases

6.1. Document Signing

6.1.1. Prerequisites

These prerequisites have to be fulfilled in order for the following use cases to work:

1. The user (actor 3.2) has registered with the RA (actor 3.5) and is known to the IDP (actor 3.4)
2. The user has created and readied a document file to be signed

6.1.2. Interactive Qualified Signatures

Steps

The user performs the following steps:

1. Opens the webinterface of the signing service (actor 3.6) on their device
2. Selects the document file to be hashed
3. Selects the preferred IDP out of a list of trusted IDPs, if multiple IDPs are configured
4. Gets redirected to the IDPs login page
5. Authenticates with the IDP
6. Gets redirected back to the signing service
7. Receives the signature as a file download
8. Saves the signature file to their device

Result

The user has received a valid signature file for the document file they wanted to sign.

6.1.3. Bulk Advanced Signatures

In some cases, users might wish to sign document files all day long without being required to authenticate with the IDP for every document. In this case the authentication will be cached for a certain duration without needing to re-authenticate for each document. For security reasons, in this mode, only advanced signatures can be created.

From the users' point of view, it works like this:

1. The user opens the webinterface of the signing service on their device
2. If implemented, they click the button for authentication for batch advanced signatures
3. Gets redirected to the IDP
4. Authenticates with the IDP
5. Gets redirected back to the signing service

6. For the duration of the authentication, the user can now submit document files to be signed, receiving the corresponding signature files, without the need to reauthenticate.

Result

The user receives valid advanced signature files for each of the document files they submit for signing for the duration of the authentication.

6.2. Signature Validation

6.2.1. Prerequisites

These prerequisites have to be fulfilled in order for the following use cases to work:

1. A signature file has been created beforehand as described in 6.1.2.

6.2.2. Offline Validation

The signatures can be verified offline with just the document file, the signature file and the verification program. This mode will only be supported on desktop operating systems (GNU/Linux, Windows, macOS), not on mobile devices (Android, iOS).

Steps

The user performs the following steps:

1. The user opens the verification program
2. The user selects the document file and corresponding signature file and submits it to the verification program
3. The program verifies the signature and displays the result

Result

The user knows whether the signature is genuine and whether the document integrity is guaranteed, and whom the document was signed by.

6.2.3. Online Validation

For mobile clients a website will be provided to validate the signature by providing the document and the signature.

Steps

The user performs the following steps:

1. The user opens the web browser and navigates to the online verification service web interface
2. The user selects the document file and corresponding signature file and submits it to the verification service
3. The verification service verifies the signature and displays the result

Result

The user knows whether the signature is genuine and whether the document integrity is guaranteed.

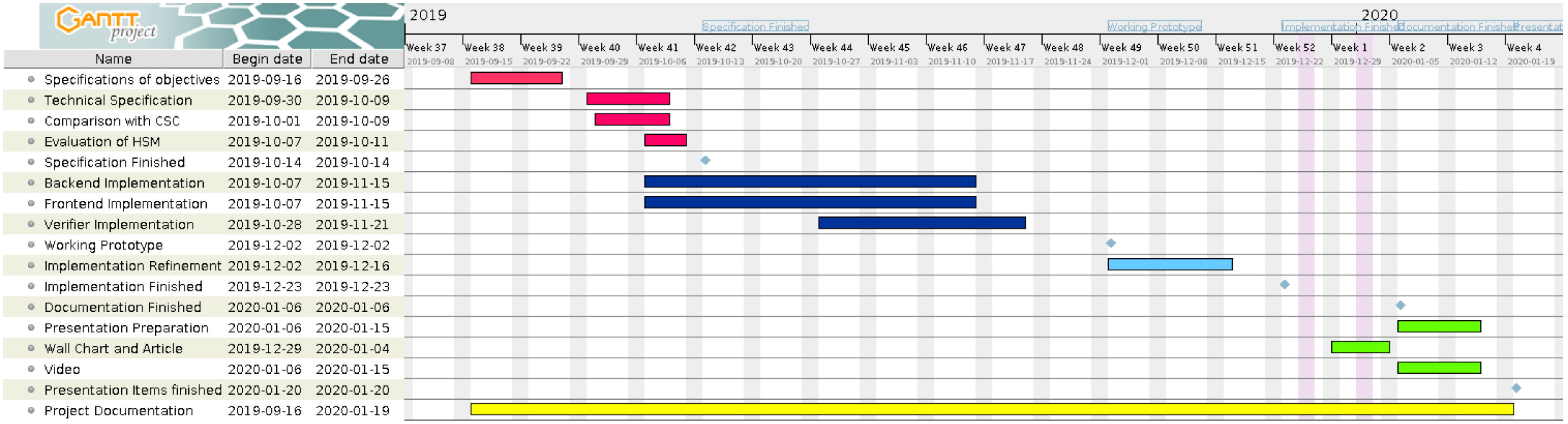
7. Project Management

7.1. Project Method

We will be using the SCRUM process for organising our work. This means splitting the work packages into stories and scheduling these stories for completion in sprints. Sprints shall last two weeks. At the end of every sprint the progress is reviewed with the advisors and the next sprint is planned.

7.2. Project Timeline

Since we know the timeframe in which we must complete our work, we created the following project timeline.



Name	Begin date	End date
Specifications of objectives	2019-09-16	2019-09-26
Technical Specification	2019-09-30	2019-10-09
Comparison with CSC	2019-10-01	2019-10-09
Evaluation of HSM	2019-10-07	2019-10-11
Specification Finished	2019-10-14	2019-10-14
Backend Implementation	2019-10-07	2019-11-15
Frontend Implementation	2019-10-07	2019-11-15
Verifier Implementation	2019-10-28	2019-11-21
Working Prototype	2019-12-02	2019-12-02
Implementation Refinement	2019-12-02	2019-12-16
Implementation Finished	2019-12-23	2019-12-23
Documentation Finished	2020-01-06	2020-01-06
Presentation Preparation	2020-01-06	2020-01-15
Wall Chart and Article	2019-12-29	2020-01-04
Video	2020-01-06	2020-01-15
Presentation Items finished	2020-01-20	2020-01-20
Project Documentation	2019-09-16	2020-01-19

Figure 7.1.: Project timeline

7.3. Work Packages

7.3.1. Specification of Objectives

The objectives of our thesis will be specified. This included functional and non-functional requirements.

7.3.2. Technical Specification

The requirements defined in the objectives will be mapped to concrete technology choices, and the reasons are given for the choice of languages and frameworks.

7.3.3. Comparison with CSC Implementation

We will compare our specification with the Remote Signing Standard specified by the CSC.

7.3.4. Evaluation of Yubikey HSM

We will evaluate how we can use the Yubikey HSM for our Signature Service.

7.3.5. Backend Implementation

The backend consisting of the Signing Service and the OIDC Coupling with the IDP will be implemented.

7.3.6. Frontend Implementation

The frontend consisting of the Hashing and UI will be implemented.

7.3.7. Standalone Verifier Implementation

The application for the offline verification of the signature will be implemented.

7.3.8. Implementation Refinement

After the working prototype, eventual bugs will be fixed and optional goals will be implemented.

7.3.9. Documentation

We will document the work.

7.3.10. Presentation

We will create the slides and prepare the presentation.

7.3.11. Wall Chart and Article

We will create a wall chart and short article describing our work.

7.3.12. Video

We will create a short video introducing our problem.

Part II.

Report

8. A Cryptographic Primer

In this chapter we well very briefly introduce the most important IT security and cryptography building blocks we use to make remote digital signing possible. Readers with a basic knowledge of IT security topics such as hash functions, X.509, Public Key Infrastructure (PKI) and Digital Signature Algorithm (DSA) can safely skip it. The descriptions given are as brief as possible in order to introduce the topics, they're not meant to be complete nor excruciatingly precise.

8.1. Hash Function

A hash function in cryptography is an one-way function which is able to map data of arbitrary length to fixed-size values [55]. One-way means that for a given hash value, it is infeasible to find the corresponding input data. Ideally, the only way for someone to invert such a hash function is to do an exhaustive brute-force search. This is called pre-image resistance. Furthermore, a cryptographic hash function needs to fulfil the following properties:

1. For a given input value, it must always produce the same hash value (it must be deterministic)
2. It must be infeasible to find to different input values that produce the same hash value (this is called collision resistance)
3. For a given input value, it must be infeasible to find another input value that produces the same hash value (second pre-image resistance)
4. A minimal change in the input value must result in a completely different output value (avalanche effect)

Hash functions fulfilling these properties are fundamental to our work (and to much of cryptography in general). Without them we would be completely powerless. An example for such a hash function is Secure Hash Algorithm 2 (SHA-2) [47].

8.2. Asymmetric cryptography

Asymmetric cryptography, sometimes called public-key cryptography, is a type of encryption which uses pairs of keys. This is in contrast to symmetric encryption which uses only one key (for example, a passphrase encrypting a file).

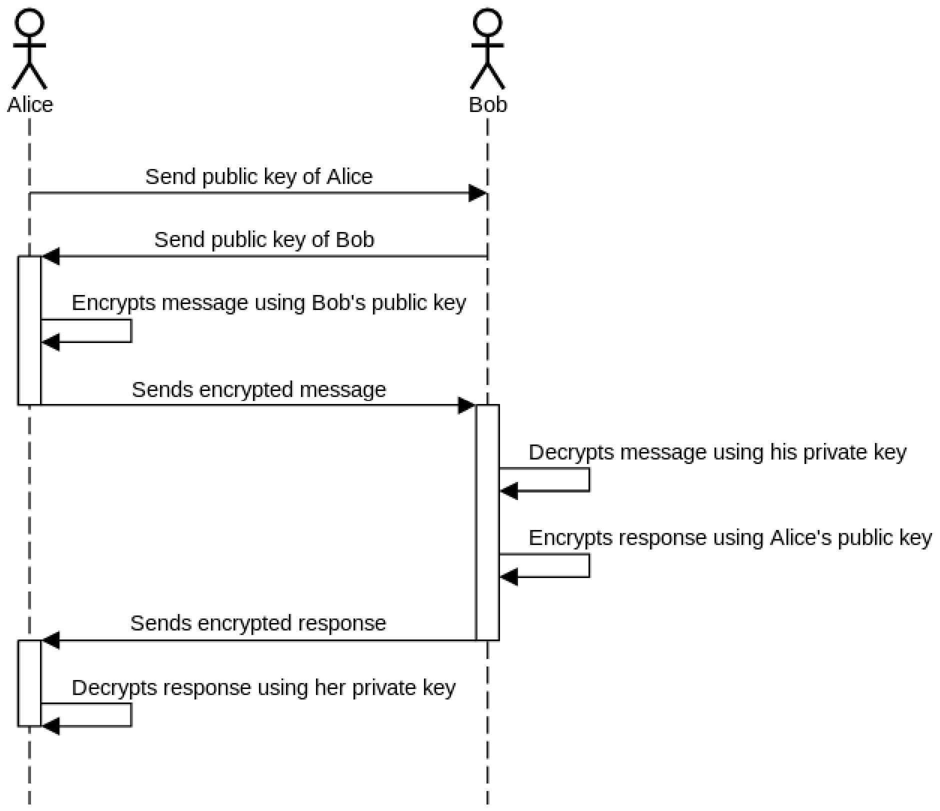
With symmetric encryption the passphrase must me known both to encrypt and to decrypt the message, but with public-key cryptography, the public key can be used to encrypt a message and the private key to decrypt it.

This might sound simple on the surface but opens up a world of possibilities. Only the private key has to be kept secret, the public key can be freely published [58].

The classical example for such an encryption system is the Rivest-Shamir-Adleman (RSA) scheme [52].

For a simplified example how public-key-based encrypted communication between two parties could work, [1] see figure 8.1.

[1] We're well aware of the major security problems in this example, like the fact that both the key exchange and the message exchange happen unauthenticated and without integrity protection, but we intentionally chose to keep the example as simple as possible in order to keep it easily comprehensible by a wide audience.



Alice
Bob
Send public key of Alice
Send public key of Bob
Encrypts message using Bob's public key
Sends encrypted message
Decrypts message using his private key
Encrypts response using Alice's public key
Sends encrypted response
Decrypts response using her private key

Figure 8.1.: Simplified example of two Actors, Alice and Bob, exchanging encrypted messages using public-key cryptography

8.3. Digital Signatures

Having briefly explained Hash Functions in 8.1 and Asymmetric Encryption in 8.2 we can now move on to introducing digital signatures. A digital signature is a way for verifying the integrity and authenticity of a message, that is, to know who the message author is and to guarantee that it wasn't tampered with [38].

Digital Signatures are not Electronic Signatures Please note that the Digital Signatures we describe here are distinct from Electronic Signatures. Electronic signatures provide the same legal standing as a hand-written signature on paper, and as such are defined in laws such as ZertES [50]. Digital signatures on the other hand merely refer to a mathematical scheme for providing message integrity and authenticity. Digital signatures are used to implement electronic signatures, but they're not equivalent.

If we want to create a digital signature on a message, we perform the following steps:

1. We take our message and run it through a cryptographic hash function, thus obtaining the hash value.
2. Then, we encrypt the hash value using our private key.
3. We transmit the message and the encrypted hash value to the recipient.

In order to verify the authenticity and integrity of the message, the recipient performs the following steps:

1. They run the message through the same cryptographic hash function we did and obtain its hash value.
2. They decrypt the encrypted hash value we sent them using our public key and compare it to the hash value they obtained themselves in step 1.
3. If the values match, the recipient can be confident that a) the message wasn't tampered with and b) we authored it.

In the message exchange shown in figure 8.1, there is a problem: anyone could encrypt messages for Bob and pretend to be Alice, since his public key is, well, public.

So by employing public-key cryptography, Bob is able to receive encrypted messages from Alice but they're of limited use to him, since he has no way of knowing who actually sent them. Fortunately, we can solve this problem by using digital signatures.

Before Alice encrypts her message to Bob using his public key, she creates an digital signature by using a hash function and her private key as described above. Then she encrypts both the message and the signature using Bobs public key and sends the two to him.

Bob then decrypts the message and verifies the digital signature as described above.

However, there is a serious problem still: an evil actor with the ability to intercept the communication between Alice and Bob could not only read their messages, but change them at will, effectively impersonating Bob as seen from Alice, and Alice as seen from Bob. For a solution to this problem please see section 8.3.

Figure 8.2 expands upon figure 8.1 to illustrate this attack.

8.4. Public-Key Infrastructure and Certificate Authorities

Public-key encrypted and authenticated communication as described in chapter 8.3 is vulnerable to man-in-the-middle attacks as illustrated in figure 8.2. This attack works because Malroy is able to mislead Bob and Alice to use his keys instead of theirs by intercepting and replacing their public keys in the initial key exchange.

This could be solved trivially if Alice and Bob exchanged their keys in a secure manner, for example by meeting face-to-face, thus ensuring Malroy can't sit in the middle. However, this negates the main advantage of using public-key cryptography: if they're forced to meet they could just as well exchange a symmetric key and use that for encrypting their messages.

This one of the problems a PKI solves. On an abstract level, a PKI is a mechanism that couples a public key with an identity [57]. What this means for the attack shown in figure 8.2 is that it provides Alice and Bob a way to make sure they're using each others' keys and not Malroys', thus preventing the attack. Because Alice and Bob now have a mechanism to verify which identity a public key refers to, they can detect Malroys attack because the public keys maliciously issued by him will not correspond to Alice nor Bob.

A well-known and widely-used example for such a PKI is X.509 [44]. In practice such PKIs are complex, and because this section's already become longer than we like we'll forego explaining how X.509 works.

8.5. Trusted Digital Timestamping

Trusted digital timestamping is a scheme for proving the existence of a piece of information at a certain point in time. There are several such schemes, such as X9.95 or ISO/IEC 18014. In this section we will focus on PKI-based timestamping as defined in RFC 3161 [24].

In RFC 3161, timestamps are issued by a trusted third party, the Time Stamping Authority (TSA).

Trusted timestamps are created by using digital signatures (see 8.3) and hash functions (see 8.1). In order to create a timestamp, the following steps are performed:

1. We feed the information to be timestamped to a hash function and obtain its the hash value
2. We send the hash value to the Time Stamping Authority (TSA)
3. The TSA concatenates the hash value with a timestamp
4. The TSA feeds the concatenation of our hash value with the timestamp to a hash function, in turn obtains the hash value of the concatenation
5. The TSA digitally signs the hash value from the previous step
6. The TSA sends the signed hash as well as the timestamp back to us

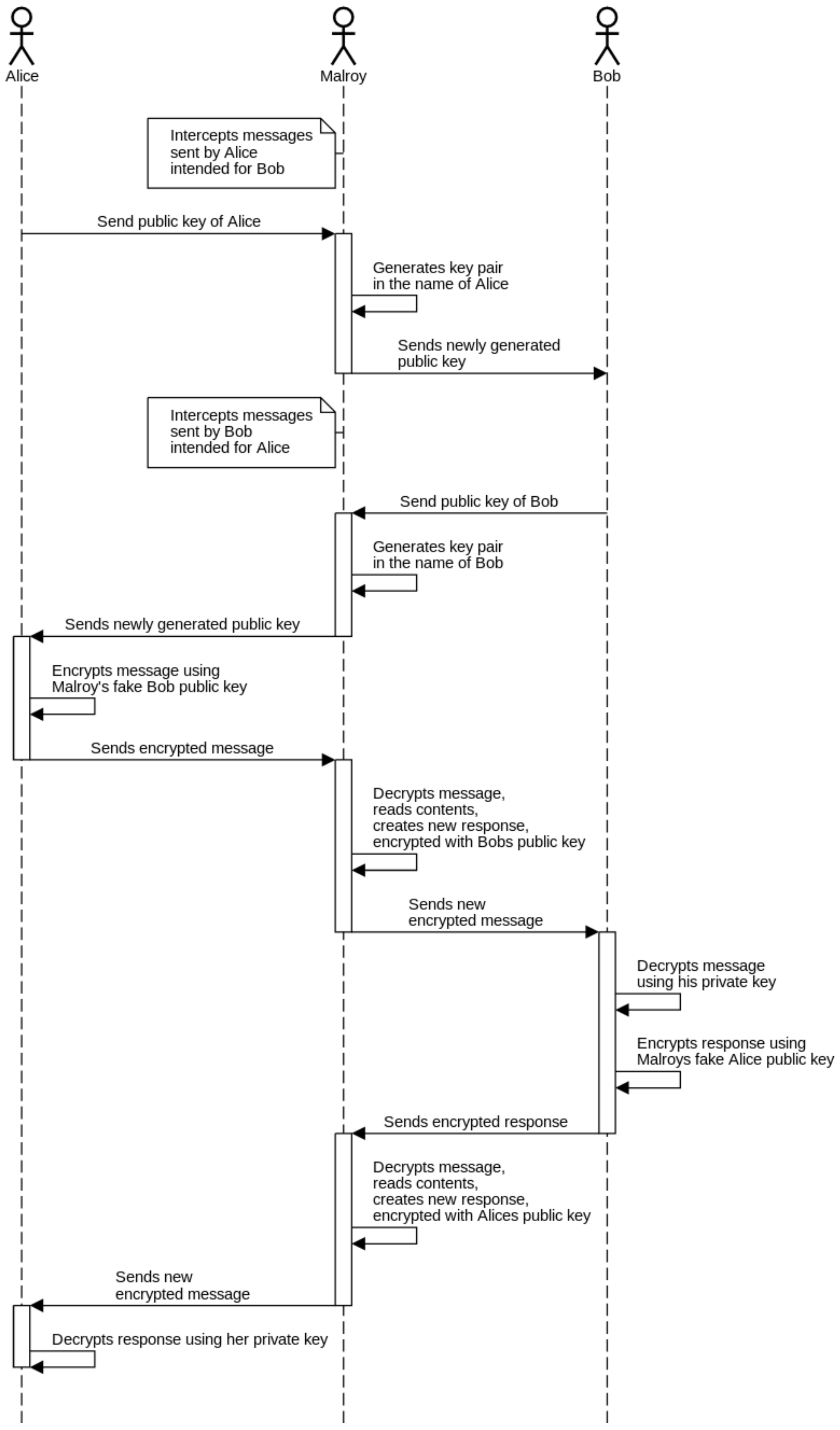

Alice
Malroy
Bob
Intercepts messages sent by Alice intended for Bob
Send public key of Alice
Generates key pair in the name of Alice
Sends newly generated public key
Intercepts messages sent by Bob intended for Alice
Send public key of Bob
Generates key pair in the name of Bob
Sends newly generated public key
Encrypts message using Malroy's fake Bob public key
Sends encrypted message
Decrypts message, reads contents, creates new response, encrypted with Bobs public key
Sends new encrypted message
Decrypts message using his private key
Encrypts response using Malroys fake Alice public key
Sends encrypted response
Decrypts message, reads contents, creates new response, encrypted with Alices public key
Sends new encrypted message
Decrypts response using her private key

Figure 8.2.: Man in the middle attack on unauthenticated public-key encrypted communication

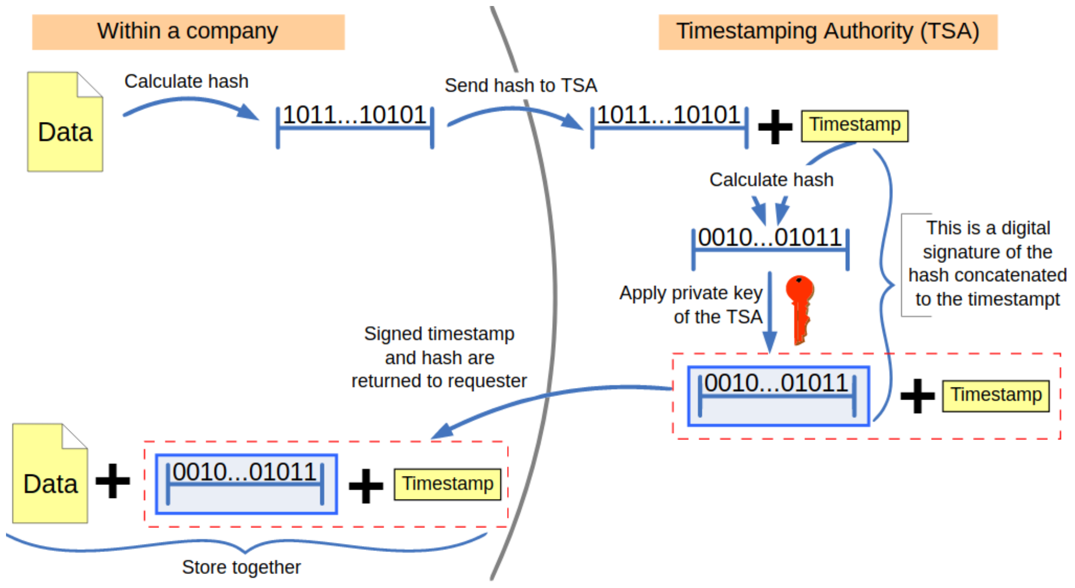

Within a company
Timestamping Authority (TSA)
Data
Calculate hash
1011...10101
Send hash to TSA
1011...10101
Timestamp
Calculate hash
0010...01011
This is a digital signature of the hash concatenated to the timestampt
Apply private key of the TSA
Signed timestamp and hash are returned to requester
0010...01011
Timestamp
Data
0010...01011
Timestamp
Store together

Figure 8.3.: Process of obtaining a timestamp from a Time Stamping Authority (TSA). Source: `https://en.wikipedia.org/wiki/File:Trusted_timestamping.svg`

7. We store the signed hash, the timestamp and the original information

For an illustration of this process, see figure 8.3

8.6. Summary

In a nutshell, the main ideas to take away from this chapter are:

- Hash functions are one-way functions, mapping data of arbitrary length to fixed-length values
- Asymmetric cryptography allows for advertising the public portion of the key, and can be used to encrypt messages
- Digital signatures provide a means of verifying the integrity and authorship of a message
- Public Key Infrastructures provide a way to pair a public key with an identity
- Trusted Digital Timestamping is a means to proving the existence of a piece of information at a given point in time

9. Technical Specification

9.1. Requirements for IDPs

The IDP is a critical component of the remote signing solution, since identification and authentication is delegated to it. Furthermore, if the IDP is not carefully selected and vetted, the most significant security advantage of our solution as compared to existing services, like Adobe Document Cloud or Swisscom All-In Signing Service, the distribution of trust, may be compromised.

This is why the IDP used with any instance of this signing service must be carefully selected, and it must fulfil at least the following requirements, in addition to the requirements specified by European Telecommunications Standards Institute (ETSI) [40].

All of the requirements specified must be met in order for qualified signatures to be emitted. For an explanation on what qualified signatures are, please see 10.3.

9.1.1. Independence of IDP and Signing Service

The IDP and the organisation responsible for it must be independent from the organisation responsible for the signing service and vice versa. It must not be the same company, or conglomerate, or even the same owner of two companies, that own or operate these services; because otherwise there may be a single stakeholder able to influence or even control both. This must never happen as it would compromise the distribution of trust.

9.1.2. Enrollment and Identity Proofing Requirements

The IDP is required to identify and register users at Identity Assurance Level 3 (IAL3) as defined by NIST [48]. In practice, this means that physical presence is required for identification, and two pieces of strong physical proof of identification has to be presented, such as a passport as well as a drivers' licence. For a complete list of requirements, please see the relevant NIST publication [48].

9.1.3. Authentication and Lifecycle Management Requirements

The IDP is required to authenticate users at Authenticator Assurance Level 3 (AAL3) as defined by NIST [49]. This level of authentication requires the use of a multi-factor hardware crypto device, such as a MobileID Subscriber Identity Module (SIM) card with a key protected by a Personal Identification Number (PIN) not shorter than six digits (something you have, plus something you know). The use of software multi-factor crypto devices is allowed as well, but only in combination with additional factors. For a complete list of requirements, please see the relevant NIST publication [49].

9.1.4. Requirements for Advanced Electronic Signatures

The requirements specified in 9.1.2 and 9.1.3 apply to qualified electronic signatures. For advanced electronic signatures, Authenticator Assurance Level 2 (AAL2) is sufficient. The AAL required by the signing service to emit the desired signature level can be specified to the IDP [54, Section 2], and is subsequently confirmed by the IDP in the field `acr` of the JWT `id_token`. If the IDP does not confirm the required AAL, the signing process is aborted. Since the `id_token` is included in the signature file, anyone can verify the AAL confirmation by the IDP.

9.1.5. Use of proper X.509 Certificates and Chains for Signing JWTs

The IDP is required to sign the JWT `id_tokens` using proper X.509 certificates signed by a trusted CA, and is required to publish the complete certificate chain in its JSON Web Key Store (JWKS) in the field `x5c` as specified in [45, RFC 7515, Section 4.1.6]. This JWKS along with optional revocation information for LTV can then be embedded into the signature file by the signing service upon signature creation, and checked by the verifier, even offline.

Reason for Requirement

The use of X.509 is permitted by the JSON Web Signature (JWS) standard [45, Section 5.1.6] but not required for use with OIDC, which is why we require it explicitly here. Since a JWT is a JWS [28, Section 1], and a JWS is used with a JSON Web Key (JWK) [27] stored in a JWKS, and a JWKS allows the use of keys other than X.509 certificates that are part of a certificate chain issued by a trusted CA, this presents us with a problem.

Upon signature verification, the verifier needs to check whether the `id_token` embedded into the signature really was issued by a trusted IDP. It verifies this by checking whom the JWT `id_token` was signed by. If the verifier didn't check this, any valid JWT could be placed into the signature and the link between the signer identity assertion and the data signed would be broken (or could be forged).

This verification is performed as described in [28, Section 7.2] by using the JWKS issued by the IDP. However, this JWKS might contain plain public keys, not signed by any CA [1]. If that JWKS is embedded into the signature, the verifier is able to confirm which key in the JWKS was used in signing the JWT, but there would be no way for the verifier to confirm that the JWKS embedded into the signature is authentic, that it really is the one used by the IDP. An attacker could embed an `id_token` and a matching JWKS into a signature file, and the verifier would accept it.

There's an obvious solution for this: the verifier simply fetches the JWKS from the IDP at verification time (over Transport Layer Security (TLS)) and uses the keys contained in it to verify the JWT `id_token`. As easy as this solution sounds, unfortunately it comes with two undesirable consequences:

1. The verifier can no longer function offline. Embedding the JWKS is useless as the verifier has no way of checking whom it was issued by.
2. When the IDP rotates the keys in the JWKS, which it will sooner or later, and no longer publishes the old keys used for signing the `id_token` embedded into the signature, verification of the `id_token`'s JWS signature will no longer be possible, and thus the signature will be invalidated.

This is why we require the use of proper X.509 certificates and chains for signing the JWT `id_token`.

9.2. Protocol

In this section, we provide an explanation of the protocol and the values used in it.

The protocol of our signing service consists of five main phases:

- Pre-Login (see 9.2.1)
- Login (see 9.2.2)
- Post-Login (see 9.2.3)
- Signature Generation (see 9.2.4)
- Signature Verification (see 9.2.5)

For a high-level overview of these phases, see figure 9.1.

[1]For an example of a JWKS using plain RSA keys not signed by a CA, see `https://www.googleapis.com/oauth2/v3/certs`.

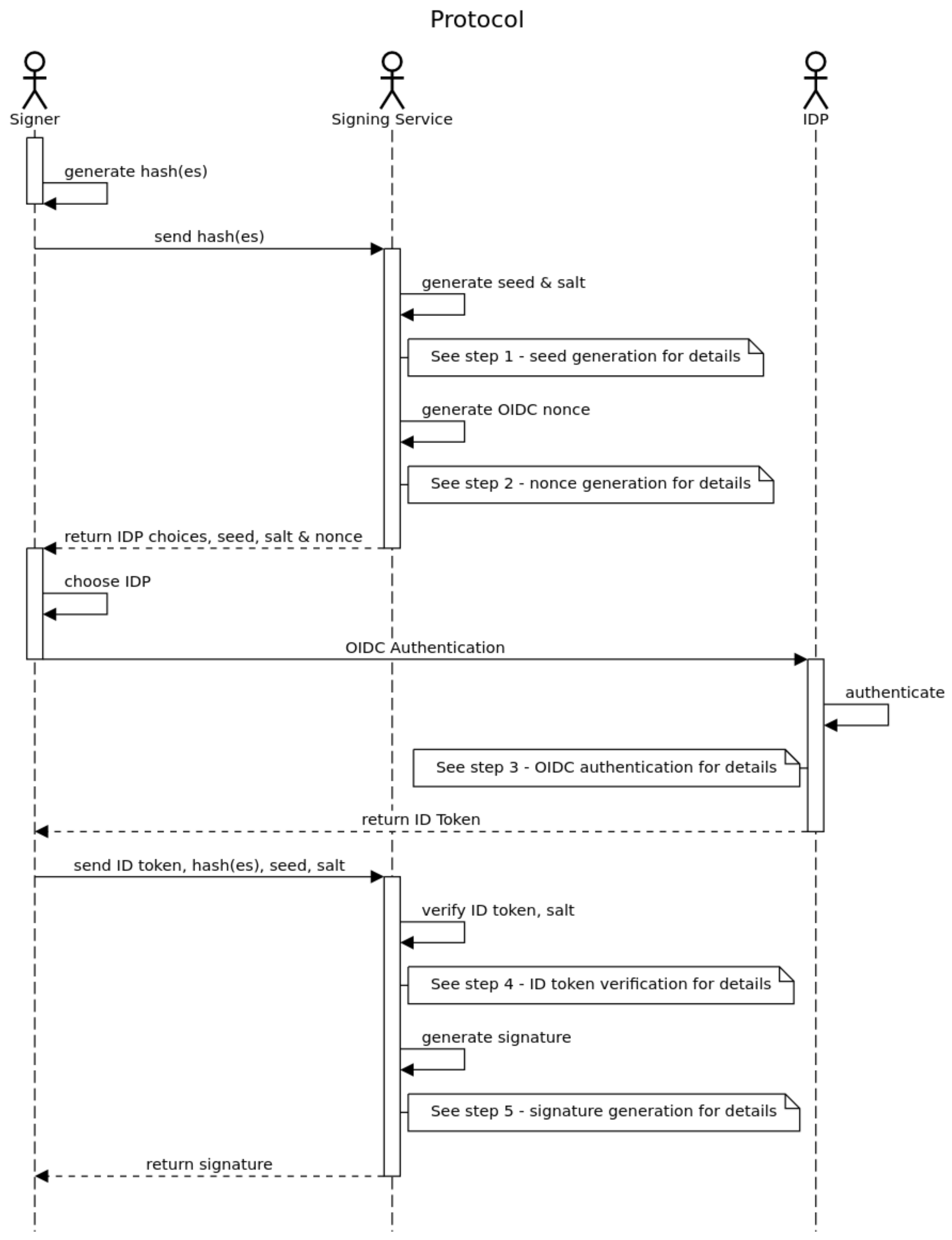

Protocol
Signer
Signing Service
IDP
generate hash(es)
send hash(es)
generate seed & salt
See step 1 - seed generation for details
generate OIDC nonce
See step 2 - nonce generation for details
return IDP choices, seed, salt & nonce
choose IDP
OIDC Authentication
authenticate
See step 3 - OIDC authentication for details
return ID Token
send ID token, hash(es), seed, salt
verify ID token, salt
See step 4 - ID token verification for details
generate signature
See step 5 - signature generation for details
return signature

Figure 9.1.: High-level protocol overview

9.2.1. Pre-Login

In this phase the signer provides a list of hashes to signing service (this list may consist of just one entry in case the signer wishes to sign a single document).

Then, the signing service generates a random seed.

$$seed = random()$$

This seed, together with a static secret known to the server only, is used to calculate the key that is used to generate a Keyed-Hash Message Authentication Code (HMAC) of the sorted list[2] of hashes. The key is calculated using a HMAC-based Key Derivation Function (HKDF) as specified in RFC 5869 [26].

$$hmacKey = HKDF(seed, staticServerSecret)$$

From the key obtained from the HKDF and the document hashes, a salt value is derived by using a HMAC as specified in RFC 2104 [23].

$$salt = HMAC(hmacKey, sorted(hashes))$$

Next, the nonce is calculated by masking each hash with the salt by using a HMAC, and hashing this list with a secure hash algorithm [19].

$$nonce = H(\{\ HMAC(salt, h)\ \ \forall\ h \in sorted(hashes)\ \})$$

The nonce needs to be derived from the list of salted hashes because only the salted hashes will be included in the signature file.

For a sequence diagram of this phase, see figure 9.3.

Purpose of the salt The salt is used to mask the document hashes in the OIDC nonce from the IDP. This way the IDP cannot learn whether two people sign the same document(s). If we wouldn't salt the hashes, the OIDC nonce would be the same for the same sorted list of hashes, and the IDP could detect when two people sign the same document(s).

While someone could argue that the IDP learning about different people signing the same document isn't much of a security problem, we want to ensure that each involved party is given the absolute minimum of information necessary for them to fulfil their role.

In addition to shielding the hashes from the IDP, the salt masks the document hashes from recipients of multi-file signatures. When multiple files are signed at once, all of the hashes are included in the resulting signature file. However, since someone could receive only a subset of the files that were signed together, they could learn the document hashes of the other files. (For example, a company signing a thousand invoices at once but sending each customer only their invoice.)

Again, this shouldn't be a significant problem but again, we don't want to allow this. So we include only the salted hashes in the signature file, and the salt itself.

Since the verifier is in possession of the documents, they can calculate their respective hashes, and then derive the salted hashes themselves since the salt is included in the signature file. Then they can check whether their document hash(es) are in the list of salted hashes. This way, they can verify their documents without learning anything about the other documents, not even their hash values.

[2]The list has to be sorted, otherwise the same hash values in a different order would produce different HMACs.

Purpose of the seed The seed is necessary in order to obtain a different HMAC key for every request despite using the same static secret on the server, and in order to strengthen the HKDF as recommended in RFC 5869 [26, Section 3.1]. Furthermore, the seed is used as a Cross Site Request Forgery (CSRF) protection mechanism without the need for the server to keep any state. If no such seed were used for generating the salt, the signing server would be forced to keep the CSRF token in memory and link it to the users session, in order to be able to check whether the hashes or the salt were replaced somewhere in the OIDC authentication implicit flow (and as such, establishing the secure link between the document hashes and the authentication). If the signing server didn't verify this, it would allow for skipping the Pre-Login step (see 9.2.1) and proceeding directly to the Post-Login step (9.2.3) as seen from the signing service.

The server then returns a list of IDP choices as well as the `seed` and the `salt` to the client.

For a sequence diagram of this phase, see figure 9.2.

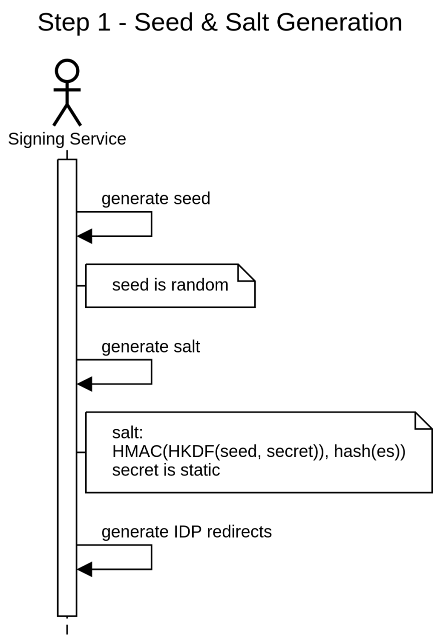


Figure 9.2.: Seed & salt generation step

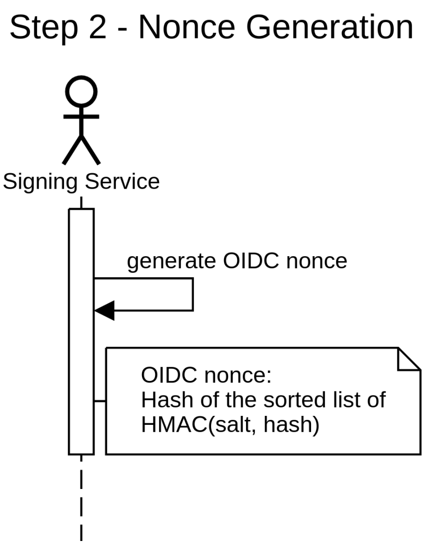


Figure 9.3.: Nonce generation step

9.2.2. Login

Having received a list of IDPs from the server, the client chooses an IDP and follows the link. The user then authenticates with the IDP and receives their ID token as seen in figure 9.4.

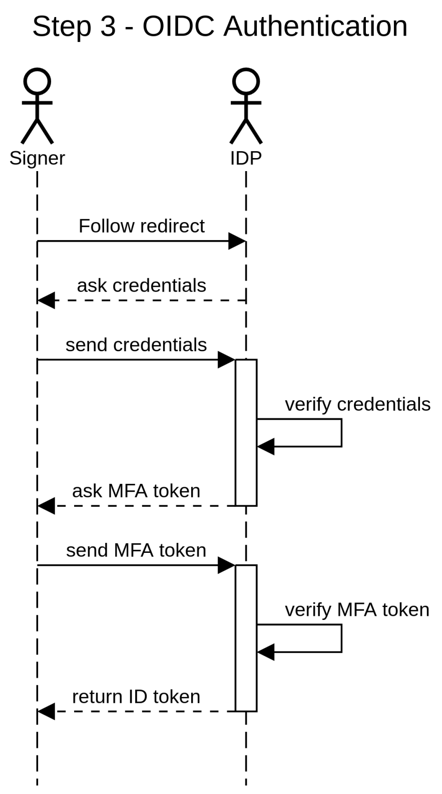


Figure 9.4.: OIDC authentication step

We discussed having the nonce and redirect links be generated by the client, but we decided against it for two reasons:

- It would mean relying on the client's RNG, instead of the server's. Obviously we still implicitly rely on the client's RNG since we depend on TLS, but still, it's one less potential weak point.
- We have to write less client-side code.

9.2.3. Post-Login

As shown in figure 9.5, the client sends the ID token, the list of hashes, the seed and the `salt` to the signing service. The signing service then verifies the `salt`, OIDC nonce and ID token as described in 9.2.3. After this step, the seed is not used anymore and is discarded.

Verification of hashes, seed and salt

Upon receiving a well-formed request in the format described in 9.3.1, the signing server performs no fewer than the following checks:

1. Verifying the id_token as described in [28, Section 7.2]
2. Checking the length of the seed and salt values
3. Checking that the salt and nonce match the calculation described in 9.2.1

These verifications are paramount to the safety of the system.

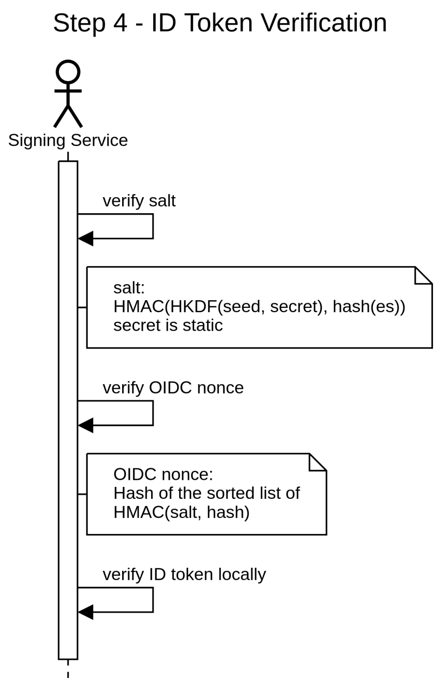


Figure 9.5.: Token verification step

9.2.4. Signature Generation

The signing server requests a new signing key from the HSM, which in turn generates a private key and returns a CSR. This CSR is sent to the CA where it is signed and the signed certificate returned.

The data to be signed is then submitted to the HSM to be signed using the signing key, resulting in a Cryptographic Message Syntax (CMS) with an embedded signature. The CMS consists of the certificate chains for the signing key and the JWT key of the IDP, OCSP and CRL for them, the JWT itself, and the masked document hashes and related data.

The hash of this CMS is sent to a TSA where a signed timestamp is created and returned.

Finally, the timestamp and the signature are combined in a protobuf message as specified in section 9.4.

See figure 9.6 for a sequence diagram of this process.

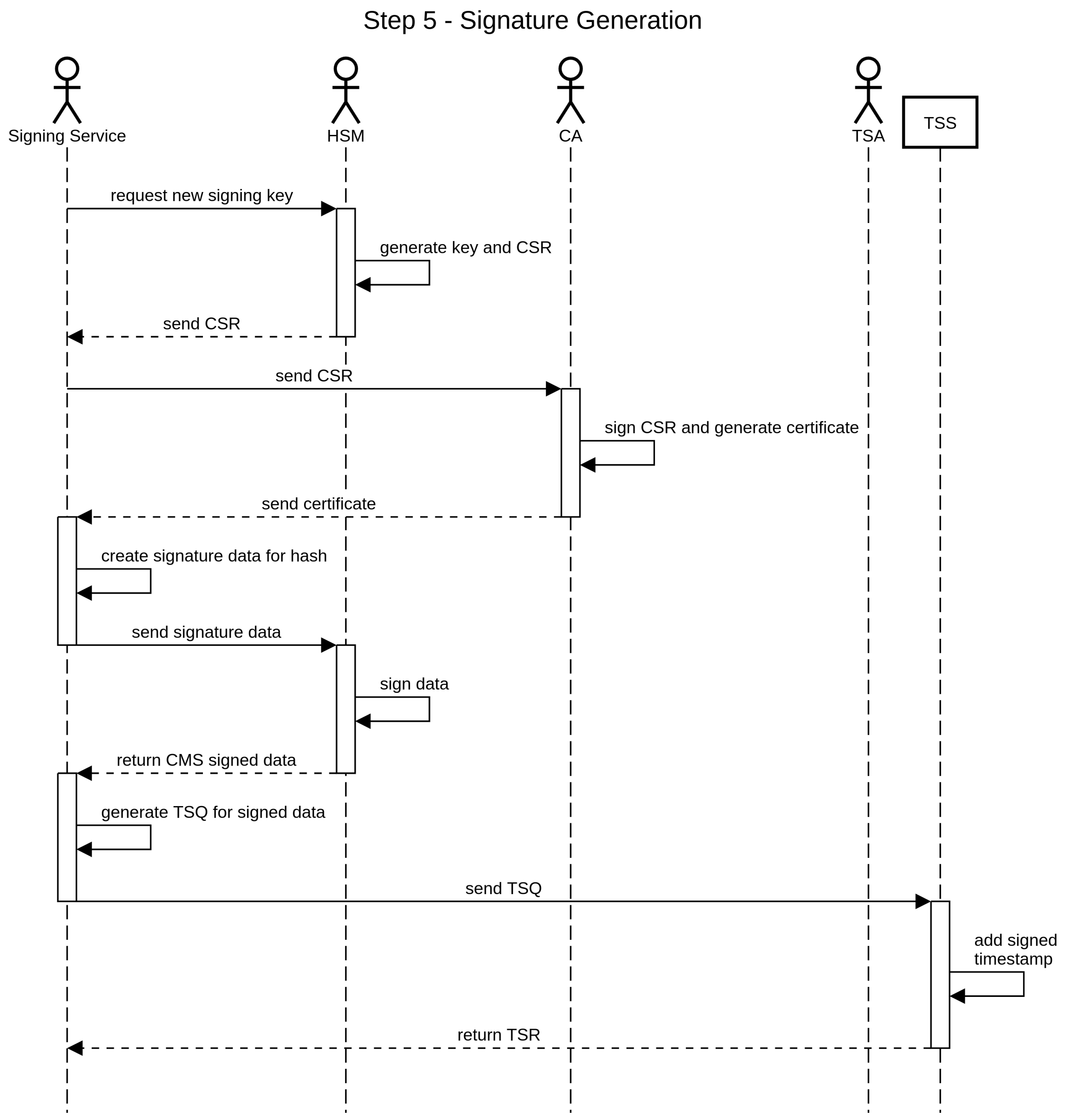
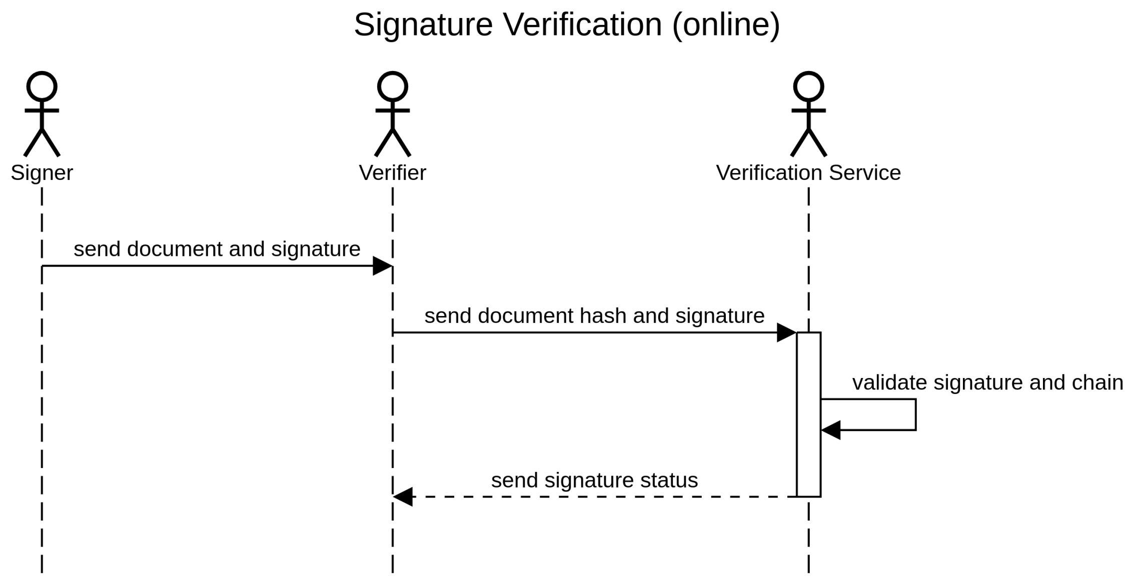


Figure 9.6.: Signature generation step

9.2.5. Signature Verification

The signature verification can either be online (figure 9.7) or offline (figure 9.8).

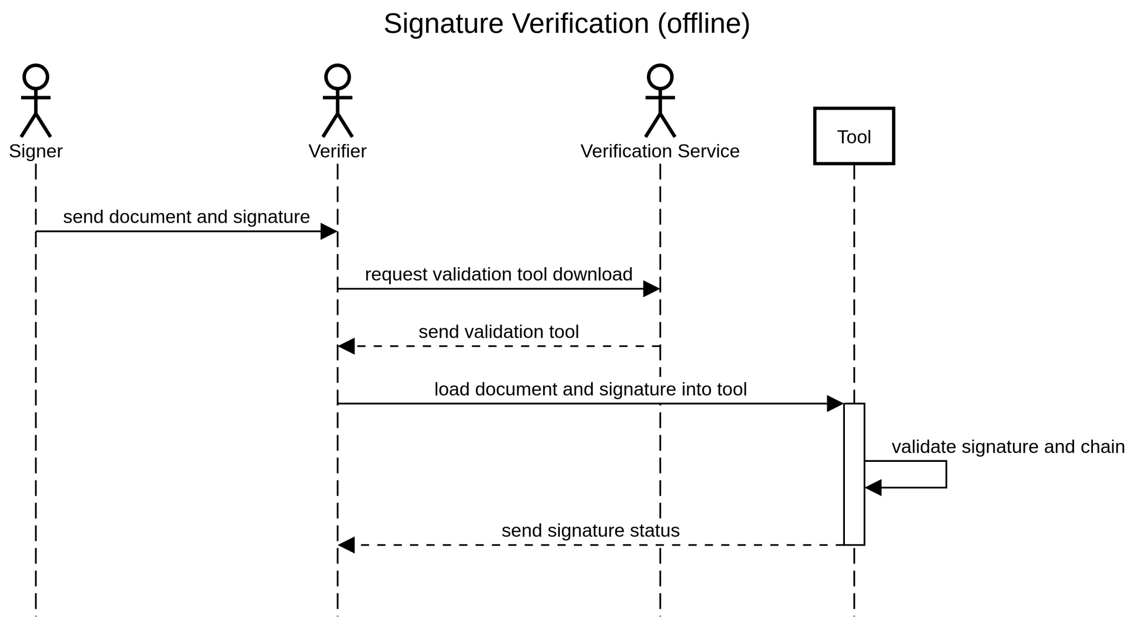
For online verification, the user simply uploads the list of hashes and the signature file to the verification service.



Signature Verification (online)
Signer
Verifier
Verification Service
send document and signature
send document hash and signature
validate signature and chain
send signature status

Figure 9.7.: Online Signature Verification Protocol

For the offline verification, the user downloads the verification programme and starts it. Then, they upload the list of hashes and signature file just as if they were using online verification, but to their own copy of the verification service running on their computer without needing any network connection instead of a remote verification service.



Signature Verification (offline)
Signer
Verifier
Verification Service
Tool
send document and signature
request validation tool download
send validation tool
load document and signature into tool
validate signature and chain
send signature status

Figure 9.8.: Offline Signature Verification Protocol

To verify a signature, the verifier first needs to verify the chain of signed timestamps and their respective CA chains (figure 9.9).

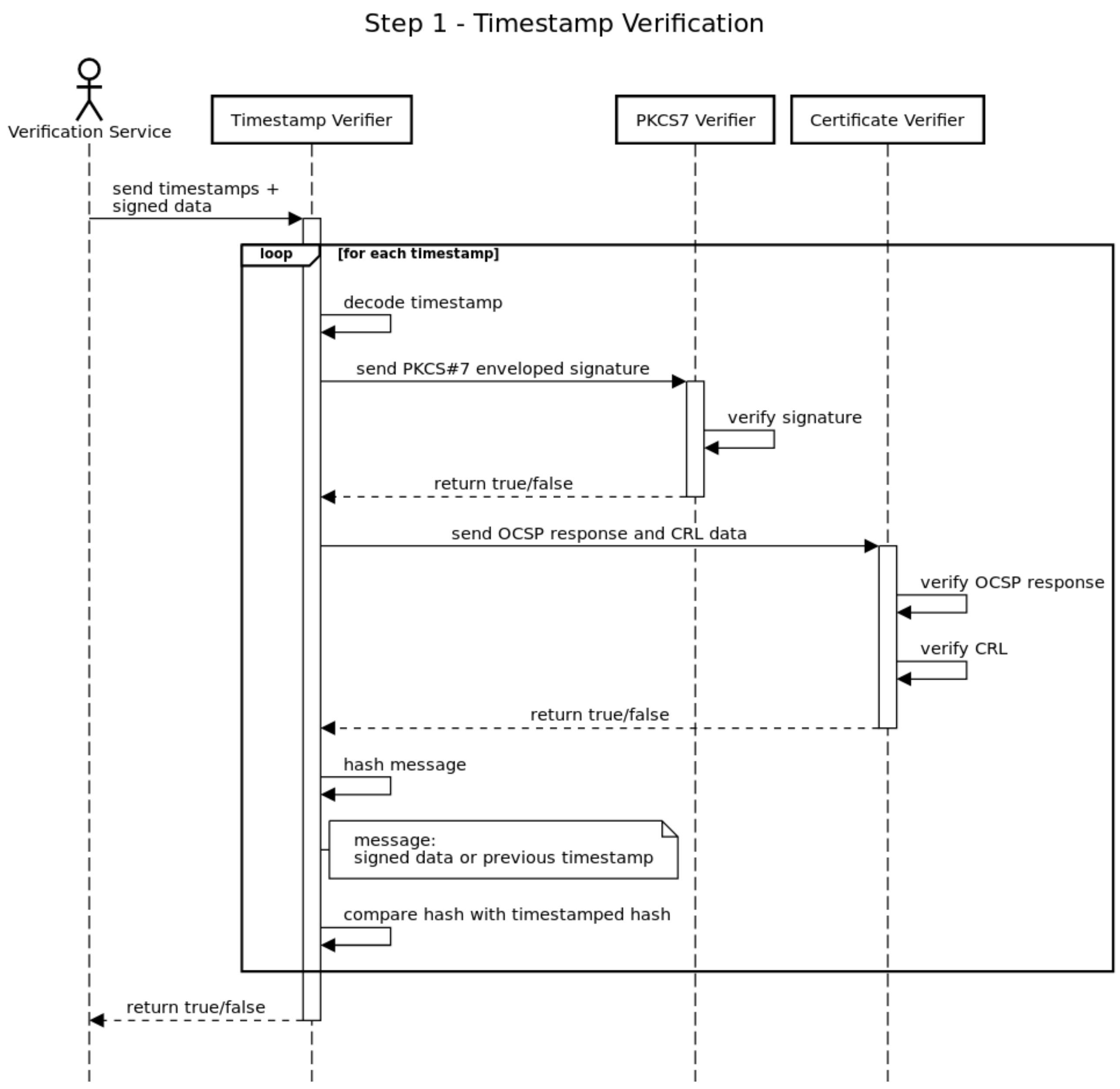


Figure 9.9.: Timestamp verification step

The signature itself can then be verified (figure 9.6), along with the certificate chain for the signing certificate.

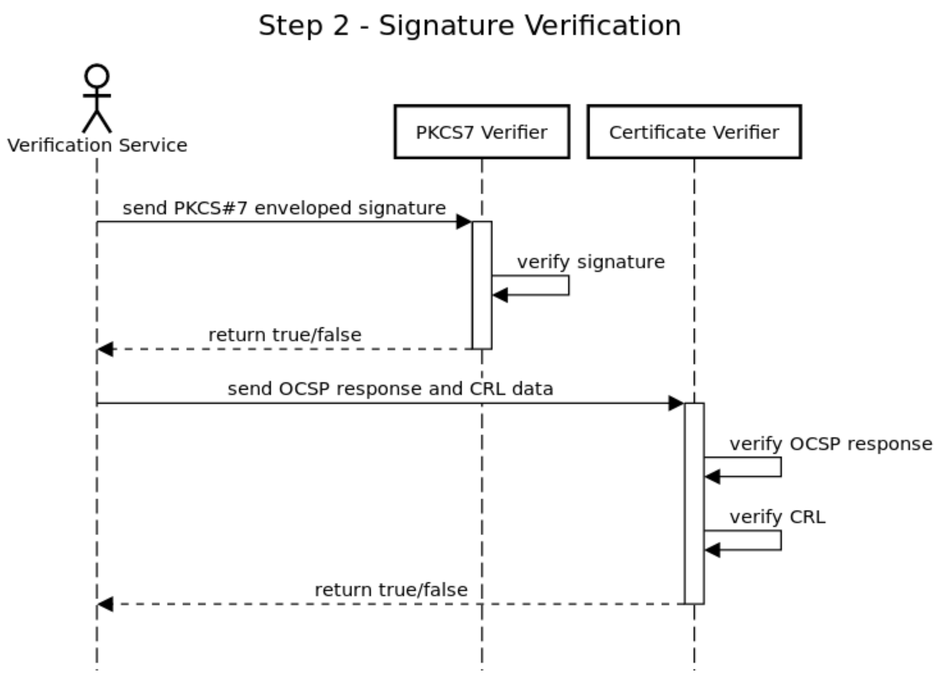


Figure 9.10.: Signature verification step

Then, the ID token itself needs to be verified, as well as the certificate chain of the key used to sign the ID token (figure 9.11).

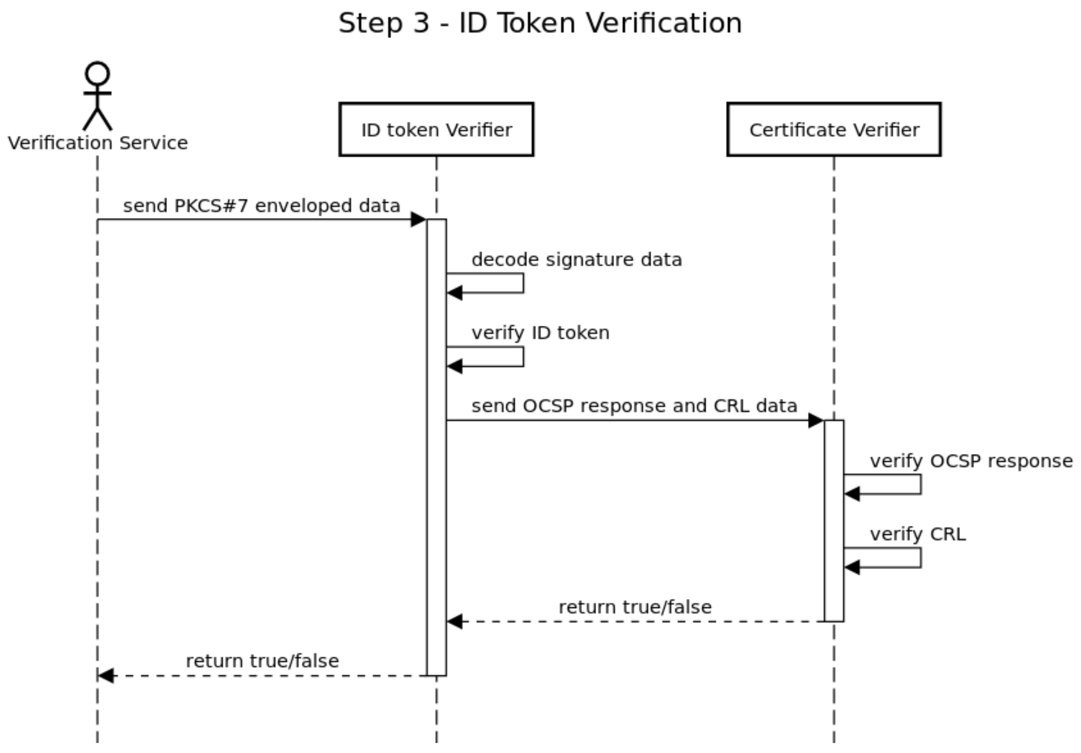


Figure 9.11.: ID token verification

To verify the link of the document hash with the ID token the following steps are performed:

- Salt the document hash
- Check whether the salted document hash is part of the list of salted hashes in the signed data. If not, abort with failure. If yes, proceed.
- Hash the sorted list of salted document hashes in the signed data.
- Check whether the resulting hash matches the nonce in the JWT. If not, abort with failure.

Figure 9.12 illustrates this process.

Multi-File Signature Verification For multiple files signed by the same signature file (multiple file hashes), the process is the same, except that the steps enumerated above are repeated for each hash.

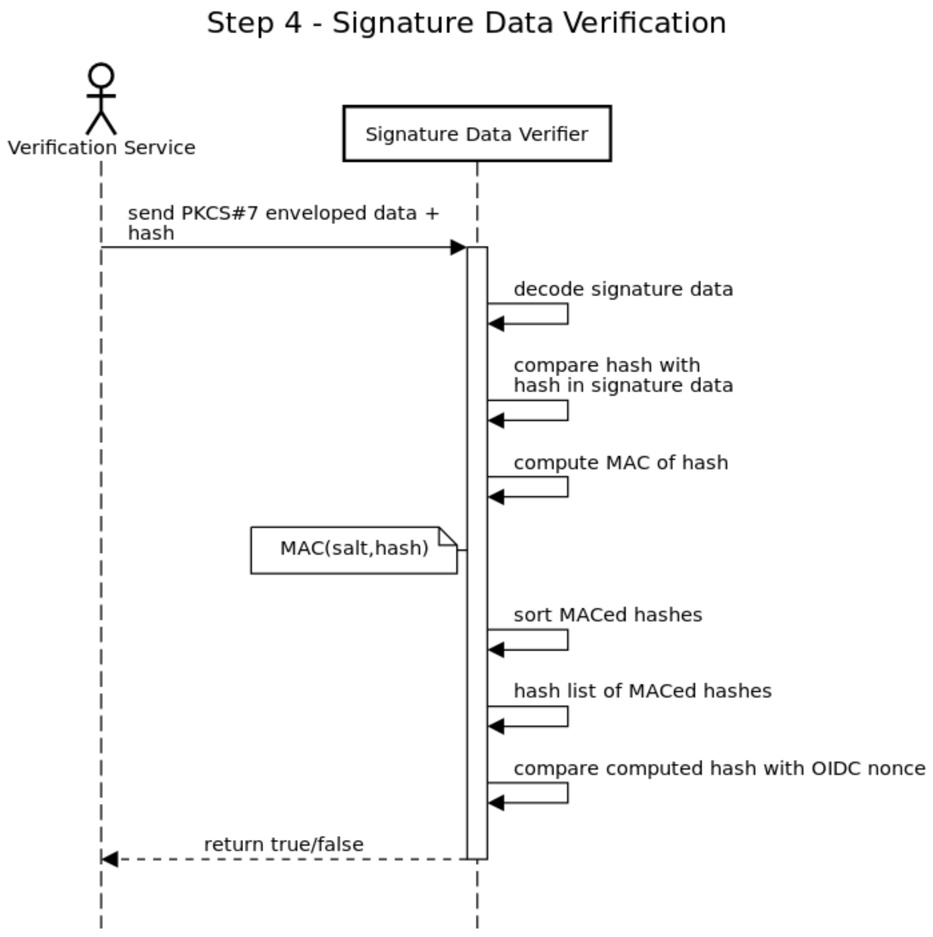

Step 4 - Signature Data Verification
Verification Service
Signature Data Verifier
send PKCS#7 enveloped data + hash
decode signature data
compare hash with hash in signature data
compute MAC of hash
MAC(salt,hash)
sort MACed hashes
hash list of MACed hashes
compare computed hash with OIDC nonce
return true/false

Figure 9.12.: Signature data verification

9.3. REST API

In this section the REST API is specified. Which endpoints are offered, what they expect as input data and what they respond with. The examples are given for illustration purposes and are not normative.

9.3.1. Signing Service

The signing service offers the following endpoints.

No.	Endpoint	Method	Description
1	/api/v1/login	POST	Send hashes to sign, receive list of OIDC providers
2	/api/v1/sign	POST	Send hashes to sign, receive signature URL
3	/api/v1/signatures/:id	GET	Retrieve signature file

Table 9.1.: Endpoints offered by the Signing Service

In the most common case, the client application will call these endpoints in the order as follows.

1. The user submits the hashes of the documents they wish to sign and receives a list of IDPs
2. After authentication with the IDP, the user submits the hashes again, this time the server will begin construction of the signature file
3. After the signature is created, the user downloads the file to their device

But, the order in which these endpoints are used is not enforced by the API except for what is required with the OIDC implicit flow, and the linkage of the id_token with the signing.

Client errors

If a malformed request is sent by the client, the signing server will respond with a 400 client error code, and a message indicating the cause of failure. This format of returning errors is the same for all endpoints of the signing server.

Parameter	Presence	Type	Description
message	MANDATORY	String	Reason for rejection of request

Table 9.2.: Output in case of invalid input

Sample Error Response:

```
HTTP/1.1 400 Bad Request
Content-Type: application/json

{
    "message": "Invalid hash length"
}
```

Listing 9.1: Error response

login

This endpoint is used for submitting the hashes the user wishes to sign.

Parameter	Presence	Type	Description
hashes	MANDATORY	List<String>	Hashes of the documents to be signed

Table 9.3.: Input to the login endpoint

Duplicates in the list of hashes are not allowed and are rejected by the API as described in 9.3.1. The length of each hash is checked, and if they don't match the hashing algorithm used the request is rejected as well. The encoding of the hashes is checked, and if they don't appear to be a string of sane hex numbers the request is rejected.

Output:

Parameter	Presence	Type	Description
providers	MANDATORY	Map<String, Url>	List of providers with the redirect url
seed	MANDATORY	String	Seed for generating the salt
salt	MANDATORY	String	Salt for generating the OIDC nonce

Table 9.4.: Output of the login endpoint

Sample Request:

```
POST /api/v1/hashes HTTP/1.1
Host: service.example.org
Content-Type: application/json

{
    "hashes": [
        "e8a96e6203b9c0df058ba862abc63d9c520157faef6d5d54e54e526b0a85b2be",
        "0b9a7fd3e612061a7fe6d834e102a143170f33d0e8c5a8eb79416aa3eb53c0d6"
    ]
}
```

Listing 9.2: login request

Sample Response:

```
HTTP/1.1 200 Ok
Content-Type: application/json

{
    "providers": {
        "SwissID": "https://...&nonce=6cd7ef99e5e79d68d...f26d06bd728ae49fa",
        "SecIDP": "https://...&nonce=6cd7ef99e5e79d68d...f26d06bd728ae49fa"
    },
    "seed": "84c97acc49335faa0266fb29b4228205e9400a85a10faa68ec30cf894e1730ed",
    "salt": "cfb663431af5e2d68be48867f93e86e477cd7eeefc10b16a51c238d2c810561b"
}
```

Listing 9.3: login response

Sign

After having authenticated with the IDP, the client application calls the signing endpoint. This is where the actual signature file is being assembled by the signing server.

Parameter	Presence	Type	Description
id_token	MANDATORY	String	OIDC ID token
seed	MANDATORY	String	Seed for generating the salt
salt	MANDATORY	String	Salt for generating the OIDC nonce
hashes	MANDATORY	List<String>	Hashes of the documents to be signed

Table 9.5.: Input to the signing endpoint

The signing server performs the checks described in 9.2.3, and if any of them fail to pass, the server rejects the request as described in 9.3.1.

Output:

Parameter	Presence	Type	Description
signature	MANDATORY	Url	Url to the generated signature file

Table 9.6.: Output of the signing endpoint

Sample Request:

```
POST /api/v1/sign HTTP/1.1
Host: service.example.org
Content-Type: application/json

{
    "id_token": {...},
    "seed": "84c97acc49335faa0266fb29b4228205e9400a85a10faa68ec30cf894e1730ed",
    "salt": "cfb663431af5e2d68be48867f93e86e477cd7eeefc10b16a51c238d2c810561b",
    "hashes": [
        "e8a96e6203b9c0df058ba862abc63d9c520157faef6d5d54e54e526b0a85b2be",
        "0b9a7fd3e612061a7fe6d834e102a143170f33d0e8c5a8eb79416aa3eb53c0d6"
    ]
}
```

Listing 9.4: Sign request

Sample Response:

```
HTTP/1.1 201 Created
Content-Type: application/json

{
    "signature": "https://.../api/v1/signatures/0b1131ca0b68f3d55b8e32a55e8"
}
```

Listing 9.5: sign response

signatures/:id

Input:

Parameter	Presence	Type	Description
id	MANDATORY	String	id of the hash that was signed

Table 9.7.: Input to the signature retrieval endpoint

Output: Binary data, meant to be presented to the user as a file download

Sample Request:

```
GET /api/v1/signatures/e8a96e62034e526b0a85b2be HTTP/1.1
Host: service.example.org
```

Listing 9.6: signature request

Sample Response:

```
HTTP/1.1 200 OK
Content-Type: application/octet-stream
Content-Disposition: attachment; filename="signaturefile"

<binary data follows>
```

Listing 9.7: signature response

9.3.2. Verification Service

Endpoints:

Endpoint	Method	Description
/api/v1/verify	POST	Send hash signature file for verification

Table 9.8.: Overview of endpoints offered by the verification service

verify

Input:

Parameter	Presence	Type	Description
hash	MANDATORY	String	Hash of the signed document
signature	MANDATORY	String	base64 encoded signature file

Table 9.9.: Input to the signature verification endpoint

Output:

Parameter	Presence	Type	Description
valid	MANDATORY	Boolean	validity of signature
error	OPTIONAL	String	error message why the signature is invalid
id_token	MANDATORY	Object	id token + claims + cert chain
signature	MANDATORY	Object	signature data (hashes, salt, algorithms)
signing_cert	MANDATORY	Object	cert chain of signer cert + signer info
timestamp	MANDATORY	Object	signing time and cert chain

Table 9.10.: Output of the signature verification endpoint

Sample Request:

```
POST /verify HTTP/1.1
Host: service.example.org
Content-Type: application/json

{
    "hash": "e8a96e6203b9c0df058ba862abc63d9c520157faef6d5d54e54e526b0a85b2be",
    "signature":
        "IyMjIFJFU1QgQVBJIFNwZWNpZm...yMjIyMgRW5kcG9pbnQKYGBgUE9TVCAvYXBpL3YxL3NpZ25gYGAK"
}
```

Listing 9.8: sign request

Sample Response:

```
HTTP/1.1 200 OK
Content-Type: application/json

  {
    "valid":true,
    "id_token":{
      "Issuer":"https://keycloak.thesis.izolight.xyz/auth/realms/master",
      "Audience":["thesis"],
      "Subject":"2d76a06e-651d-4b96-9024-c81cbdbf6948",
      "Expiry":"2019-12-06T11:26:24+01:00",
      "IssuedAt":"2019-12-06T11:11:24+01:00",
      "Nonce":"106c85bbfd85a70dfd99408428c0d67163bce696a57ea97fd1e65d4be3304b41",
      "AccessTokenHash":"",
      "email":"test2@thesis.izolight.xyz",
      "email_verified":true,
      "cert_chain":[
        {
          "issuer":"CN=Thesis Intermediate
              CA,OU=CA,O=Thesis,L=Bern,ST=BE,C=CH",
          "subject":"CN=Thesis IdP,OU=CA,O=Thesis,L=Bern,ST=BE,C=CH",
          "not_before":"2019-11-27T22:58:00Z",
          "not_after":"2022-11-26T22:58:00Z"
        },
        ...
      ]
    },
    "signature":{
      "salted_hashes":[
        "b6b0b1064ed7dfdf351db7d7bd5b52123f3e0070fcef40860dfde1e57c8ad5bc",
        "6a7ec5219706cf7a9c373fe72de5dcdce2dcd1df5a2b97b5282699c31eb5513b"
      ],
      "hash_algorithm":"SHA2_256",
      "mac_key":"c400087d1da8c443988fbf12ea48e56164c5de5a69769bab2eccf93f40560849",
      "mac_algorithm":"HMAC_SHA2_256",
      "signature_level":"ADVANCED"
    },
    "signing_cert":{
      "signer":"CN=USER Test2,OU=Demo Signing Service",
      "signer_email":"test2@thesis.izolight.xyz",
      "cert_chain":[
        {
          "issuer":"CN=Thesis Root CA,OU=CA,O=Thesis,L=Bern,ST=BE,C=CH",
          "subject":"CN=Thesis Root CA,OU=CA,O=Thesis,L=Bern,ST=BE,C=CH",
          "not_before":"2019-11-23T05:28:00Z",
          "not_after":"2049-11-15T05:28:00Z",
          "ocsp_status": "Good",
          "ocsp_generation_time": "2019-11-23T05:27:00Z"
        },
        ...
      ]
    },
    "timestamp":{
      "SigningTime":"2019-12-06T10:11:26Z",
      "cert_chain":[
        {
          "issuer":"CN=SwissSign Platinum CA - G2,O=SwissSign AG,C=CH",
          "subject":"CN=SwissSign Platinum CA - G2,O=SwissSign AG,C=CH",
          "not_before":"2006-10-25T08:36:00Z",
          "not_after":"2036-10-25T08:36:00Z"
        },
        ...
      ]
    }
  }
```

Listing 9.9: Sign response

Invalid signature:

```
HTTP/1.1 200 OK
Content-Type: application/json

{
    "valid": false,
    "error": "signed hash and provided hash don't match"
}
```

Listing 9.10: sign response

9.4. Signature File Format

Digital signatures unfortunately aren't very simple. Several components have to be put together for them to work. In this section, we will outline the signature file format, its parts, and what they are for.

9.4.1. High-Level Overview

The signature file consists of the following parts, from innermost to outermost:

- The innermost part of the signature file is the information that is being signed.
- This information gets enveloped in a RFC 5652 [25] CMS message along with certificates and revocation information.
- This CMS is encoded with Distinguished Encoding Rules (DER)[3] and added to a Protobuf message.
- A number RFC 3161 [24] timestamps are added to the same Protobuf message.

For a graphical overview of the format see figure 9.13.

[3]We like to think of DER encoding as in German, "DER ENCODING!", the one and only encoding! We thought it was funny.

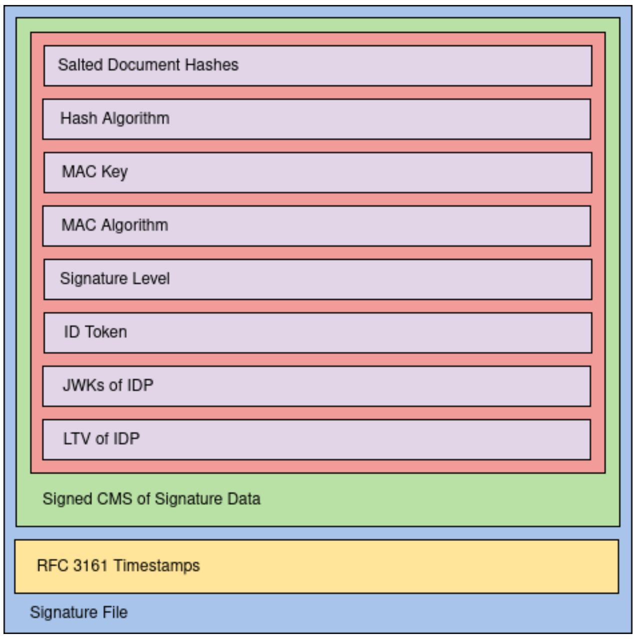


Figure 9.13.: Signature file overview

In the following subsections each of these parts is explained in more detail.

9.4.2. SignatureData: The information that is being signed

The information that is being signed is structured as a Protobuf message with the following schema:

```
message SignatureData {
    repeated bytes salted_document_hash = 1;
    HashAlgorithm hash_algorithm = 2;
    bytes mac_key = 3;
    MACAlgorithm mac_algorithm = 4;
    SignatureLevel signature_level = 5;
    bytes id_token = 6;
    bytes jwk_idp = 7;
    map<string, LTV> ltv_idp = 8;
}
```

Listing 9.11: SignatureData schema

salted_document_hash is a list of salted document hashes. They are salted as described in section 9.2.1. There must be at least one salted hash. There cannot be more than 100'000 hashes. We limit this so that no one tries encoding arbitrary data in there (like pictures of cats) just because they can.

hash_algorithm is the hash algorithm used for the values in `salted_document_hash` as well as the OIDC nonce.

mac_key is the key fed to the HMAC function specified by `mac_algorithm` for masking the hashes. For an explanation on that masquerade and why it's necessary see 9.2.1.

mac_algorithm is the specific HMAC function that's being used to masquerade the hashes. mac_key is the key that's fed to mac_algorithm.

signature_level is either qualified or advanced, depending on the Level of assurance (LoA) provided by the IDP. For an explanation on what advanced and qualified signatures are, see 10.3.

id_token is the OIDC token issued by the IDP. It must be included into the signature file in order for the verifier to be able to confirm that the user authenticated with the IDP, the linkage between the document hashes submitted to the IDP, the authentication and the signature file. If this token weren't included the signing server could issue signatures on its own, which we're making impossible.

jwk_idp is the JWK according to RFC 7517 [27] as published by the IDP. It is included for verification of the signature on the id_token.

ltv_idp envelops CRL and OCSP information for the certificates in the CA chain of the certificate used by the IDP when signing the OIDC id_token. This separate protobuf message is necessary because contrary to CMS, the JWK specification [27] doesn't allow for including revocation information. Because there may be a number of certificates in the JWK, this field is a map of the fingerprint of the certificate to its revocation information. The field is optional, it's only needed if LTV is desired. For more information on LTV see 9.4.4.

Putting it together These fields are put into the protobuf message SignatureData (for the schema, see listing 9.11). The message is serialised to its binary format. Then, a CMS message is constructed, enveloping and signing the binary-serialised SignatureData protobuf message. The signer certificate along with the CA certificate chain is added to the CMS message, and if LTV is desired, their respective CRL and OCSP responses.

Please note that RFC 5652 [25] CMS messages are required and no older standards because they don't allow for including OCSP responses, only CRLs. For more information on including OCSP in CMS see [25, Section 10.2.1, RevocationInfoChoices and OtherRevocationInfoFormat].

In summary, we now have serialised the core signature information, signed it, and combined it with certificate information. The next step is to timestamp the signature as described in 9.4.3.

9.4.3. SignatureFile: Combining Signed Data with Timestamps

All signatures issued by the signing server are augmented by at least one timestamp by at least one TSA independent of the signing server and the IDP.

Why timestamp?

Timestamping allows the verifier to assert the point in time the signature was created. This information is confirmed by a party independent of the signature service and the IDP, thus the verifier can be fairly confident about the authenticity of this information.

It's important for the verifier to know reliably when the signature was created, because that way the verifier is able to check whether the certificates used where valid at the time of signature creation, not at signature validation time.

Someone could ask, why not just put the date and time of creation into the signature itself? It would be much simpler and easier, instead of going through the trouble of obtaining one or more RFC 3161 [24] timestamps from independent trusted third parties. If the date and time were part of the signed data it would be protected by the digital signature, and no one would be able to alter it.

That's not good enough, because we need to guard against CA failure. Imagine what happened if the CA used by the signing server were compromised: all certificates issued by that CA would get revoked, and thus all signatures created with these certificates would become invalid. Potentially millions of contracts and other such documents would become effectively null and void in an instant, with widespread and quite possibly disastrous repercussions on its users (like losing a house, because from a legal perspective, the mortgage has just disappeared into thin air). All information authenticated by certificates signed by that CA would become untrustworthy, including the date and time of creation included in it.

Fortunately, in our solution, it is highly unlikely that someone is able to forge a signature even if they completely compromised the CA, because they'd need to compromise the IDP or its CA as well in order to be able to falsify a signature.

But even so, all issued signatures becoming void is unacceptable.

This is why the timestamping is so important. The timestamp confirms that the signatures were issued before the CA was compromised, thus protecting their validity. This is also the reason why the signing server can't just add the date and time of creation in the signature on its own, because that information is protected by a signature that was created with a certificate authenticated by a compromised CA, thus making that information unreliable.

By using at least one independent third party TSA the time of signature creation can be ascertained, because that information is authenticated by a different CA. The risk can be reduced further by combining TSAs. Then the signature will remain valid even if the signature server CA and the TSA CA were compromised.

Readers knowledgeable with JWT will observe that we could use the `iat` (Issued At) and `exp` (Expiration Time) claims from the `id_token` issued by the IDP instead of adding a RFC 3161 timestamp, since the `id_token` is signed by a party using a different CA than the signing server as well, but unfortunately these claims are optional and we can't rely on them being present [28, Sections 4.1.4 and 4.1.6].

There's another reason for using timestamps: Long-Term Validation; but this is out of scope for this chapter. For more information on that please refer to 9.4.4.

Combining the SignatureData CMS with the RFC 3161 Timestamps

For adding the timestamp(s) to the CMS message created in 9.4.2 we define another Protobuf message as follows:

```
message SignatureFile {
    bytes signature_data = 1;
    repeated bytes rfc3161 = 2;
}
```

Listing 9.12: SignatureFile message

signature_data is the DER-encoded CMS message created as described in 9.4.2.

rfc3161 is one or more DER-encoded timestamps according to RFC 3161 [24].

Any number of timestamps can be added. Timestamps either confirm `signature_data` or another timestamp earlier in the chain. The chains of timestamps formed that way can be used for LTV as described in 9.4.4.

The signature file returned to the user is the binary serialisation of `SignatureFile`.

9.4.4. Long-Term Validation

Long-Term Validation allows for the validation of signatures long after the document was signed [41].

We need LTV for two main reasons:

1. Imagine if the CA were revoked that was used for the signatures: all signatures created using the same CA would become invalid instantly, making countless documents, constracts and the like unverifiable.

2. Extending the validity of the signature beyond the lifetime of the CA used to sign it, for signatures that need to remain valid and verifiable for a very long time.

In order for us to achieve this, all required elements for signature validation must be embedded into the signature file. Without the addition of these elements, a signature can only be validated for a limited time. This limitation occurs because the CAs eventually expire, or get revoked. Once the CA certificate has expired, the issuing authority is no longer responsible for providing the revocation status information on that certificate. Without the confirmed revocation status information on the signing keys, the signature cannot be validated.

To overcome this limitation, the following information has to be embedded into the signature:

1. A timestamp on the signature
2. The signing certificate
3. An archive timestamp of the previous content

The archive timestamp establishes the date in which the information collected was issued. Provided the archive timestamp is valid, we can be sure that the revocation information was issued at that time, and check the validity of the signing certificate and the CA certificate chain. Thus we can be certain that it was not revoked at the point in time the document was signed. This allows us to extend the validity of the signature past the expiration time of the CA.

However, this does not extend the validity of the signature indefinitely, it merely extends the expiration until the expiration time of the timestamping authority's certificate.

For many cases this may be enough, but it doesn't quite allow for long-time archival yet. When the timestamping certificates' expiration is impending, the signature expiration time has to be extended by adding another timestamp signed by a CA not yet close to expiration. This re-stamping has to be repeated periodically in order to keep the signature valid and verifiable. This allows for near-indefinite archival.

10. Evolution of the Signing Protocol

10.1. Overview

It took us many iterations of gradual improvement until we arrived at the final version of the proposed solution. In this chapter we will briefly document and explain the main points of how our idea evolved, what problems we found and how we've overcome them, both for the signature file format as well as the protocol.

10.2. Signature Format

10.2.1. Support for Arbitrary File Formats

When using our signing service, people should be able to sign arbitrary files of any format. We don't want to place restrictions upon users such as "Portable Document Format (PDF) only" or "Microsoft Word documents only". Such restrictions are unnecessary and would only serve to constrain the number of people using the service (as they couldn't use their preferred formats).

This requirement presents us with a small challenge, since it is impossible to embed a digital signature into arbitrary file formats. Some formats support it out of the box, such as PDF [41], while with others existing metadata fields could be repurposed to contain signature data. Finally with some formats it's not possible to include additional data at all.

The government of Estonia faced the same problem when they developed their solution to digital signatures. They solved it by creating a new document format called DigiDoc [8], which is, in essence, a container format for the actual document along with the signature information. With their solution, arbitrary document formats can be used. However, the downside is that the user needs to install a program able to extract and display the document contained in the DigiDoc container, even if they just wished to view the document without verifying the signature.

We chose to have a detached signature file, that is, the signature data resides in a file separate from the document that was signed. In contrast to the Estonian solution, the advantage is that people don't need to install additional software if all they want is to view the signed document. The disadvantage is that users need to handle two files instead of one (the document and its signature).

10.2.2. Original Format

Our original specification for the signature format is based on our work in Projekt 2 [42] which contained the following fields:

- Signature (Base64)
- Signature format (RSA Probabilistic Signature Scheme (RSA-PSS), EdDSA with Curve25519 and SHA-512 (Ed25519))
- Signature hash algorithm (Secure Hash Algorithm 2 with 256-bit output length (SHA-256), Secure Hash Algorithm 3 (SHA-3))
- Timestamp according to RFC 3161[1]
- Public key (Privacy-Enhanced Mail (PEM))

[1] `https://tools.ietf.org/html/rfc3161`

- Issuing CA (PEM)
- Subject
- Validity
- Level

This format would be encoded as a protobuf message in order to not have an overly verbose file (as opposed to Extensible Markup Language (XML)), but still support having a schema (as opposed to JavaScript Object Notation (JSON)).

10.3. Difference Between Advanced And Qualified Signatures

The distinction between electronic and digital as well as advanced and qualified signatures is derived from Swiss Federal Law [50].

Electronic or Digital Signatures

An electronic signature is a purely technical, non-legal term. Put simply, the term denotes electronic information associated logically with other electronic information. Such information may be used by a signatory for creation of a signature. It may consist simply of a digitally scanned, handwritten paper signature.

In contrast, a digital signature is always based upon one or several cryptographic algorithms. A digital signature incorporates an unforgeable representation of the original data (guaranteed integrity) and, as such, enables proof of the origin of data.

Advanced Electronic Signature

As defined in Swiss Federal Law [50, Art. 2], an advanced electronic signature is an electronic signature which fulfills the following requirements:

1. It is exclusively associated with the holding person
2. It allows for identification of the holding person
3. It is created by means under sole control of the holding person
4. It is associated with personal information of the holding person in such a manner that retroactive modification of the data can be detected

Advanced electronic signatures have no direct legal significance, however, they may reinforce the cogency of proof in a court of law [43, 4.19].

Qualified Electronic Signature

A qualified electronic signature is an advanced electronic signature which meets the following additional conditions:

1. It is created using a secured signature creation device [50, Art. 6]
2. It is based upon a qualified certificate [50, Art. 7 and 8], whose subject is a natural person, and which was valid at the time of signature creation.

A qualified electronic signature is legally equivalent to a hand-written signature, that is, it is admissible in a court of law, it can be used to sign legally binding contracts, and so on.

10.4. Signature Protocol

The original protocol, which we specified in Projekt 2 [42] employed a server-side secret nonce to generate the nonce used in the OIDC authentication request, which needed to be kept in memory until the signer returned with the ID token from their trip to the IDP. This introduces two disadvantages for the signing server:

- It could be used to Denial of Service (DoS) the signing server, by forcing it to store immense amounts of such nonce values
- It makes the server stateful, since it is forced to store state between two requests in form of the nonce

Furthermore, with our original idea, for the verification of the signature all documents that were signed together needed to be present at the time of verification, since their hashes were incorporated in the OIDC nonce. (The nonce cannot be reconstructed and verified without knowing all of the document hashes that went into it in the first place.)

On top of that, multi-signatures, while technically possible, were made impractical for some applications: If a single person wishes to sign multiple documents at once that will be used together, (for example, an apartment rental contract, house rules, and a bank deposit confirmation) this won't be a problem. However, if multiple, independent documents are to be signed together (for example, a company sending 50 bills to 50 different customers), having to send each customer all the bills is just silly.

We wanted to do better than that.

10.4.1. Draft 1: Making the Protocol stateless

Since storing the secret nonce on the signing server is undesirable, we thought about changing the protocol to make this part stateless.

To achieve this, we introduce two more nonce-like values called `seed` and `salt`. The `seed` is a randomly generated value that is used to verify the id token when the signer returns from the IDP. The `salt` is the Message Authentication Code (MAC) of the document hashes concatenated with the `seed`, using a static server side secret as key.

The `salt` takes the role of the original nonce that was used to construct the OIDC nonce and to protects against the IDP gaining knowledge of the hashes to be signed.

The signing server returns both the `seed` and the `salt` to the client, which then constructs the OIDC nonce. The OIDC nonce is now the MAC of the list of hashes with the `salt` used as key.

When the signer returns to the signing server, it presents the `seed`, `salt`, the hashes and ID token. Using the `seed` and the static secret the server can reconstruct the `salt` and verify that the presented `salt` is the same.

This functions as a CSRF protection of a malicious IDP requesting signatures using past values, while also allowing us to keep the signing server stateless.

After this step the `seed` will not be used anymore and therefore doesn't need to be in the signature document. The OIDC token will then be verified with the `salt` and the hashes.

10.4.2. Draft 2: Improving signing of multiple documents

Even with the improvements in draft 1 (section 10.4.1), only one signature file will be generated for multiple documents, incorporating all document hashes irrevocably linked together. Verifying the signature would require having all documents present, which is impractical.

To solve this, our first idea was to include the hashes of the other documents, signed together, and then generate a signature for each file. The sorted list of hashes is fed to the MAC function in the verification step. This however would leak information about the other documents, as they would be just plain hashes. We put a lot of thought into minimising the amount of information all involved actors learn, such as masking the document hash from the IDP, and we're not satisfied with a solution where the other recipients learn about unrelated document hashes just because they were signed together.

Our solution for this is to generate a MAC of each hash with the `salt` as key and include that in the signature file, with the OIDC nonce just being the hash of the sorted MACs.

This way the verifiers' own MAC can be generated during verification with the other MACs just being used as additional input parameters without leaking the hashes of the documents. Assuming the HMAC function used is secure, the only information that the receiver of the signature file could learn is the number of the documents that were signed together.

10.4.3. Draft 3: Simplifying and using CMS where possible

Draft 2 (section 10.4.2) resulted in a rather complicated schema that looked something like listing 10.1.

```
message SignatureData {
    bytes document_hash = 1;
    HashAlgorithm hash_algorithm = 2;
    bytes mac_key = 3;
    MACAlgorithm mac_algorithm = 4;
    repeated bytes other_macs = 5;
    SignatureLevel signature_level = 6;
    bytes id_token = 7;
    repeated bytes jwk_idp = 8;
    map<string, LTV> ltv_idp = 9;
}

message Timestamped {
    bytes rfc3161_timestamp = 1;
    map<string, LTV> ltv_timestamp = 2;
}

message SignatureContainer {
    bytes enveloped_signature_data_pkcs7 = 1;
    map<string, LTV> ltv_signing = 2;
}

message LTV {
    bytes ocsp = 1;
    bytes crl = 2;
}

message SignatureFile {
    SignatureContainer signature_container = 1;
    repeated Timestamped timestamps = 2;
}
```

Listing 10.1: Draft 2 schema 1

`SignatureData` is the information that gets signed. It obviously contains the hash of the document in `document_hash`, but for the reasons explained in 10.4.2 we need to include the other maskes hashes: that's what `other_macs` is for.

`Timestamped` allows us to add certificate revocation information (CRL and OCSP) for the certificates used in the RFC 3161 [24] timestamps. The inclusion of these is necessary for proper offline verification, where the verifier is most likely not able to retrieve this information by itself, and to prove the signing key was valid at the time of signing.

`SignatureContainer` is used to add revocation information for the certificates used in `SignatureData`.

When we examined RFC 5652 [25] more closely, we discovered that it's possible to add CRLs as well as OCSP responses to CMS messages (but not to Public Key Cryptography Standard 7 (PKCS7) [25, Section 10.2.1, RevocationInfoChoices and OtherRevocationInfoFormat]). Since both SignatureData and the RFC 3161 [24] timestamps are RFC 5652 CMS messages which do support including revocation information, we can simply put the revocation information in the CMS and don't need our own message formats.

Then it occured to us that we don't need to have the hash of the current document separate from the masked hashes of the other documents. We can simply include a list of masked hashes.

This works, because during verification, the original documents are present. The verifier simply calculates their hash values, masks the hashes using mac_key, and checks whether they're present in the list of masked hashes. This slightly simplifies the verification process, but the main advantage of this change is that it speeds up issuance of multi-file signatures tremendously.

When before, we created a signature file per document hash, now we create one signature file for all documents signed together. Since creating a signature file entails a nontrivial amount of work this change represents a vast improvement in signature creation speed.

```
message SignatureData {
    repeated bytes salted_document_hash = 1;
    HashAlgorithm hash_algorithm = 2;
    bytes mac_key = 3;
    MACAlgorithm mac_algorithm = 4;
    SignatureLevel signature_level = 5;
    bytes id_token = 6;
    bytes jwk_idp = 7;
    map<string, LTV> ltv_idp = 8;
}

message LTV {
    bytes ocsp = 1;
    bytes crl = 2;
}

message SignatureFile {
    bytes signature_data = 1;
    repeated bytes rfc3161 = 2;
}
```

Listing 10.2: Simplified schema

Unfortunately we can't get rid of LTV completely because there is no way to add revocation information to a RFC 7517 [27] JWK, so we still need it for the IDP certificates. Still, these changes result in a significant simplification of the schema, as seen in listing 10.2.

11. Implementation

11.1. Choices in Technologies

In this chapter, we outline the technologies we choose, and the reasoning for choosing them.

11.1.1. Backend

The backend is divided into two parts, the signing service and the verification service. They are developed independently in different programming languages.

We chose to split the backend into two because signature verification has to be available both online and offline.

Using two different programming languages and sets of libraries, implementing common formats and protocols independently from specification alone, allows us to hedge against the risk of some flaws in the libraries: if a library used in the signing service produced flawed output, it would likely be discovered by the verification service since it uses a different implementation of the same concepts. The exact same flaw being present in two different libraries implemented in different programming languages is rather small.

On top of that, we can test ourselves how well we've specified the protocols and formats: if these two services can be used together without problems, the specification was precise enough. If not, we learn where we must improve it, which is a win-win situation.

Signing Service

The signing service is implemented in the Kotlin programming language [31] using the Ktor framework [13]. For the signing service we chose Kotlin because it is a modern and concise programming language, providing null safety, coroutines for asynchronous programming, support for functional programming with higher-order functions and closures, and it's genuinely nice to program in. Kotlin offers seamless Java interoperability, allowing us to make use of the excellent and extensive Java ecosystem.

We haven't picked Kotlin and Ktor at random, we've discussed which language and framework to use for the signing server at length. The table 11.1 lists a short overview which other choices we've discussed but chosen not to pursue.

Verifier

The verification service is implemented in the Go programming language [30]. For the verifier we chose Go, because it allows us to generate statically linked binaries for all major platforms, without worrying about dependencies. Since we need binaries able to run on GNU/Linux, Mac OSX and Windows, this is very useful to us. Furthermore, the Go standard library already includes a lot of cryptographic functionality, requiring fewer third party dependencies. Go's built-in concurrency primitives (goroutines and channels) make it easy implement the verification of the many parts of the signatures asynchronously. With only having a single API endpoint for verifying the signature it isn't necessary to use a complicated web framework as the `http` package of the Go standard library provide all needed functions.

As with the signing service implementation, we evaluated alternatives, which are summarised in table 11.2.

Language	Framework	Reason for non-consideration
Java	Spring MVC	Steep learning curve, vast amount of functionality to learn, too large for our needs, little previous knowledge on our part
Java	Vaadin	Generated frontend is slow to use because of network latency, too much uncertainty with regards to WebAssembly (WASM) integration: Is it possible? How? How long will it take us, how much will we have to implement ourselves?
C#	.NET Core	Zero previous knowledge on our part both for the language and the framework, aversion to Microsoft because of their attempts to undermine open-source software [7]
Scala	Play	Little previous knowledge of the language, uncertainty of the time required to learn it properly, but otherwise an interesting contender
Python	Django	Huge framework to learn, no type safety, slow runtime performance

Table 11.1.: Programming languages and frameworks considered for use in the signing service implementation

Language	Framework	Reason for non-consideration
Java	JavaFX	Requires the user to either have Java installed or will be bundled with the program, which makes it quite large.
Anything	Electron	Electron is ludicrously resource-hungry, having the Chromium engine underneath. It also requires more frontend (javascript) code than any of us is comfortable with.

Table 11.2.: Programming languages and frameworks considered for use in the verification service implementation

11.1.2. Frontend

Given that the frontend must support the three desktop operating systems Microsoft Windows, GNU/Linux as well as Apple MacOS, the technological choices available to us are limited. On the desktop, we could use the Java Virtual Machine (JVM) platform and the JavaFX GUI library, whereas on the phones we could use Flutter [9]. However, developing three applications on five platforms using two new-to-us frameworks and programming languages would take a lot more time and resources than what is available to us in the scope of this thesis.

In order to reduce complexity and enable code reuse, we decide to implement the frontend as a web application. Web frontents are capable of running in any modern web browser regardless of platform, be it mobile or desktop. We're not happy about this, as we would much rather use mature, strongly-typed and well-designed languages and frameworks, but we're forced to make this compromise in order to meet our objectives in the time available. In order to reduce the pain, we will use TypeScript, which is a typed superset of JavaScript [32]. We discussed implementing a Single Page Application (SPA) using Angular [1], but for our use case it's overkill as there isn't too much client-side logic. In the interest of a small, fast-loading site we chose to stick with plain Hypertext Markup Language (HTML) and Cascading Stylesheets (CSS) and only adding as much JavaScript as necessary.

11.1.3. Client-Side File Hashing in the Web Browser

However, the decision to implement the frontend as a web application presents us with a challenge: hashing the files to be signed client-side in the web browser itself. If we had implemented "proper" client applications this would've been easy, but in a web browser and using its JavaScript language not so much: it simply wasn't designed with performance in mind.

The easiest solution would be to upload the files to be signed to the server and hash them there, but this would be a clear violation of the least-information principle (the server doesn't need the file, only the hash) and a breach of user privacy. Nevermind the fact that signing large files could take a very long time over slow network connections, and turn out to be quite expensive for mobile users billed for data by volume.

Another solution would be to ask the user to enter the file hashes instead of selecting files, but this would be very user-unfriendly and most likely too much to ask from many users.

It is clear we must find a way to hash files in the web browser itself. In order to achieve this we have found the following options:

1. Using the browser-implemented `SubtleCrypto` [33] API
2. Using the `CryptoJS` [10] JavaScript implementation
3. Using a WASM-based implementation

Each of these options comes with a number of advantages and disadvantages, as discussed in more detail in the following sections.

Using SubtleCrypto

The `SubtleCrypto` class offers the `digest(algorithm, data)` method [33], which can be used to calculate SHA-256 checksums. This API is implemented in the browsers themselves, and no additional dependencies are necessary. The advantage of using this implementation is that it is available in all modern browsers[1], and since it's executed with native code, being able to take advantage of Advanced Vector Extensions 2 (AVX2) instructions, instead of JavaScript it should be quite fast. There's a major drawback though: hashing a large amount of data progressively is not supported, the data has to be passed to the function en bloc, as seen in listing 11.1.

```
crypto.subtle.digest("SHA-256", data).then(hash => {
    console.log(
        // convert ArrayBuffer to hex string
        Array.from(new Uint8Array(hash)).map(
            b => b.toString(16).padStart(2, '0')
        ).join('')
    );
});
```

Listing 11.1: Using SubtleCrypto for calculating SHA-256 checksums

Our testing showed that selecting files larger than 200MB crashes Firefox tabs when trying to read their contents into memory before we could pass it to the `digest` function. If we assume the users will only ever select small files this should not pose a problem, but unfortunately it's not safe to assume this. Furthermore, this limit is probably lower still on mobile devices such as smartphones (although we didn't test this).

Using CryptoJS

`CryptoJS`, a JavaScript library, does not have the limitation of `SubtleCrypto` and supports progressive hashing[2], as seen in listing 11.2.

```
const sha256 = CryptoJS.algo.SHA256.create();

sha256.update("Message Part 1");
sha256.update("Message Part 2");
sha256.update("Message Part 3");

const hash = sha256.finalize();
```

Listing 11.2: Progressive SHA-256 hashing using CryptoJS

[1]Where modern browsers means Mozilla Firefox, Google Chrome/Chromium, and Microsoft Edge, not older than the respective versions available in 2018

[2] By progressive hashing we mean the ability to pass to the hash function the data piece by piece in order to avoid holding all of it in memory at once.

The advantage of using `CryptoJS` over `SubtleCrypto` is, as mentioned, the ability to hash piece-wise.

The disadvantage is that we need to load a third-party JavaScript library, using built-in functionality would be preferable.

And since JavaScript is an interpreted language, using it to calculate the checksums results in performance figures everyone but web developers would laugh at. This is a problem especially on mobile devices limited in compute and memory resources as well as battery capacity. Since we want to support mobile devices properly, and don't want to limit users to small files, we must do better.

Using a WASM-based implementation

WASM provides a low-level virtual machine in the web browser itself, running machine-independent binary code, comparable to the JVM or the .NET Common Language Runtime (CLR), albeit much simpler and much less sophisticated. By using this virtual machine we should be able to run code at near-native speed written in a statically-typed, compiled language such as Rust, C/C++ or Go. Thus we expect significant performance gains over a JavaScript-based implementation. While developing the WASM-based hashing programmes, we encountered some interesting challenges, as described in the following paragraphs.

CORS Policy While JavaScript can be executed simply by pointing the browser at a local HTML file, the same doesn't work for WASM. The browser's security policy forbids it due to its Cross-Origin Resource Sharing (CORS) rule [6]. We solved this by starting the Hypertext Transfer Protocol (HTTP) server built in to Go's standard library and having the browser load the WASM binary through HTTP. For the Rust-based implementation we used the built-in web server of webpack [34].

JavaScript/WASM Compatibility The Golang project conveniently provides a file containing the necessary boilerplate code to load, start and interact with WASM programmes called `wasm_exec.js`. But there's a catch: for each version of Go, the version of the accompanying `wasm_exec.js` file used must match precisely. If it doesn't, the code will crash with a segmentation fault. It took us quite some time to figure out why the code we'd written only a few days prior would segfault now with no changes made to it.

Passing data Functions written in Go intended to be used from the JavaScript side of things need to have a very specific signature. As can be seen in listing 11.3, there is no typing: all arguments passed to the function are of type `js.Value` and the return value must be of type `interface{}`[3]. This posed us with the challenge of detecting the types and casting the data passed accordingly.

```go
func f(this js.Value, arg js.Value) interface{} {}
```

Listing 11.3: Golang WASM function signature

We've worked on this for hours, producing ugly reflection-based hacks, until we decided to just agree on the types of the arguments and return values beforehand despite the open function signature. Now all that's needed is a little boilerplate to convert a JavaScript `Uint8Array` to a Golang `[]byte`, as seen in listing 11.4.

```go
func progressiveHash(this js.Value, in []js.Value) interface{} {
  array := in[0]
  buf := make([]byte, array.Get("length").Int())
  js.CopyBytesToGo(buf, array)
  return this
}
```

Listing 11.4: Uint8Array to []byte

[3]`interface{}` is Go's equivalent of Java's `Object`, it could be anything.

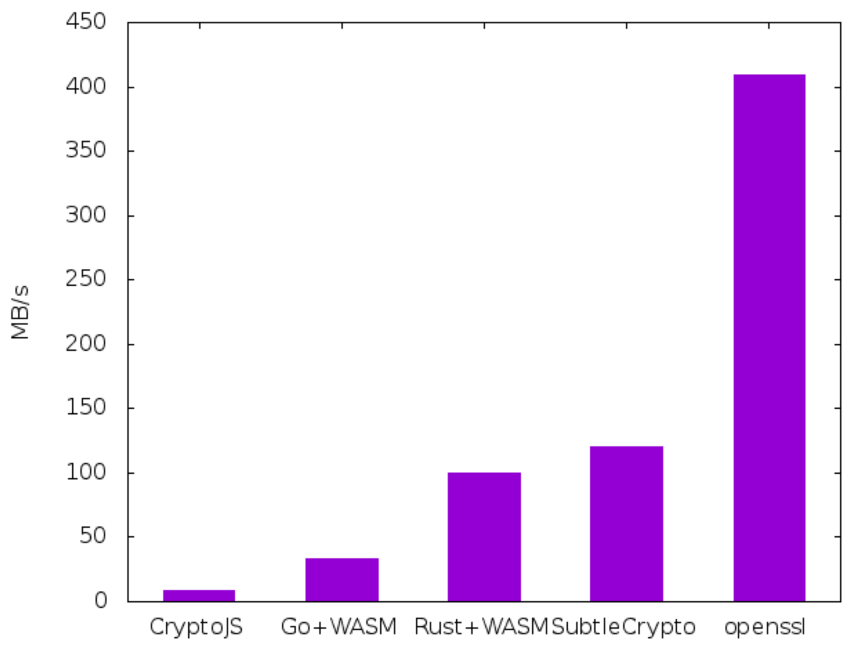


Figure 11.1.: Hashing speed in MB/s (higher is better)

Goroutines Go features its own concurrency primitive called Goroutines. From a programmers' perspective, they can be used like threads, but they carry much less overhead. Communication between goroutines is achieved by using so-called channels, which on a high level are comparable to queues. Unfortunately, the WASM specification wasn't drafted with this kind of concurrency in mind. Go is forced to unwind and restore the call stack when switching between goroutines, which is very expensive [21]. We rewrote the Go programme to work without them, and we've seen a small but significant performance improvement.

Rust based WASM As the Go implementation also includes the Go runtime, which makes the wasm file much larger, and starts a programme that will run continuously in the background it isn't the optimal choice for creating a WebAssembly implementation. As neither of us knows any other of the other languages that compile to WebAssembly well, we excluded them at first. However with the drawbacks of the Go based implementation we decided to try to implement a Rust-based version as well.

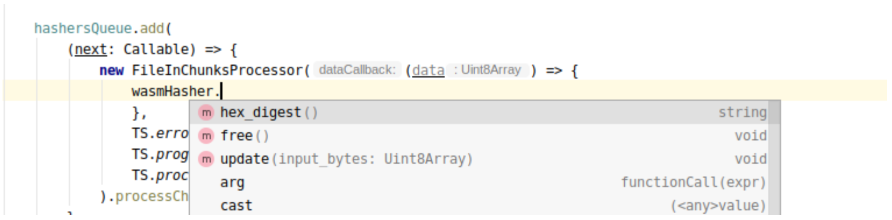
Performance Comparison

No one likes waiting for slow software to do its work, and neither do we. This is why we decided to compare the performance of each implementation in a simple test: we measure the time it takes for the browser to calculate the checksum of 1GB of random data using the aforementioned methods. The tests were run on Debian 10 using Firefox 69 on an Intel i7-8550U. The results can be seen in figure 11.1.

To have a baseline reference to compare the hashing speeds to, we include the performance of the `openssl` command-line programme. As expected, the in-browser `SubtleCrypto`-implementation is the fastest, followed by the Rust-based implementation. JavaScript is so ridiculously slow it's not even trying to compete.

11.1.4. Deciding On The In-Browser Hashing Implementation

It is clear from figure 11.1 that `SubtleCrypto` is the fastest of the options we tried. Unfortunately, since it doesn't support piece-wise hashing we're forced to pick the next-fastest option, the Rust-based implementation running in the WASM Virtual Machine (VM). Using Rust provides us with another significant advantage: the toolchain is highly developed. Upon compilation, the toolchain auto-generates TypeScript declaration files containing the



```
hashersQueue.add(
    (next: Callable) => {
        new FileInChunksProcessor( dataCallback: (data : Uint8Array ) => {
            wasmHasher.
            },      m hex_digest()                              string
            TS.erro m free()                                    void
            TS.prog m update(input_bytes: Uint8Array)           void
            TS.proc   arg                          functionCall(expr)
        ).processCh   cast                               (<any>value)
```

Figure 11.2.: Code Completion in Intellij

function signatures the WASM module exposes to the JavaScript world. This is very nice, since it allows for compile-time type checking and for smarter code completion in the Integrated Development Environment (IDE), as shown in figure 11.2.

11.1.5. Software Libraries

Table 11.3 summarises the software libraries used in the implementation of the signing service. All libraries we use are of high maturity, widely used, well tested and above all, free and open source software. That is, they are licenced with Free Software Foundation (FSF) approved open source licences.

11.1.6. Software Libraries - Verification Service

Table 11.3 summarises the software libraries used in the implementation of the verification service. As with the libraries for the signing service, they are all of high quality and free and open source software. In some cases, we needed to fork and extend functionality for which we created pull requests in the respective project repositories.

Enhancing Mozilla's PKCS7 library with OCSP support

Unfortunately, as useful as Mozilla's PKCS7 library is to us, it doesn't support verifying OCSP responses. Since we needed the functionality, we implemented this ourselves. We will submit a pull request to Mozilla, in the hope that they may find our contributions useful, and if so, that they can include our contributions into their library.

11.1.7. Dependency Management

For the signing service's dependency management, we use Apache Maven [2]. It allows us to specify project metadata, build configuration, and of course build- and runtime dependencies in a central location. Most Java developers are familiar with Maven and should be able to feel right at home. We chose Maven over alternatives such as Gradle and Ant because it is the most widely used dependency management and build system available in the Java world.

For the verifier, we use Go's built-in module system. A Go module is a set of Go packages in a file tree with a `go.mod` file at the root of the tree. This file defines the dependency requirements, which are pulled by the Go build system before compilation. A Go package is included with an import path, which specifies the location of the source code (the import path may be an Uniform Resource Locator (URL)). At this path the Go tool expects to find a source code repository (Git, SVN, Mercurial), with the tagging system of the Version Control System (VCS) used for determining the version to use, if specified.

Unfortunately, in the web world, there are no such well thought out and coherent solutions, and we are forced to stitch together multiple dependency management and build tools.

Name	Description
logback	An implementation of the Simple Logging Facade for Java (SLF4J) API. It's a replacement for log4j, but brings a large number of improvements. Also, it's Swiss, which we found nice.
Apache HttpClient	Ktor's built-in HTTP client isn't usable on its own, it requires a backing engine. We've selected the Apache HttpClient for this. Other alternatives were Coroutine-based I/O (CIO), Jetty, and OkHttp. Out of the available backing engines, it is the most configurable and complete HTTP client available at the time of writing. It supports both HTTP/1.1 and HTTP/2, and is the only one available that supports following redirects and allows for configuration of timeouts and proxies.
kotlinx.serialization	Kotlin Serialization is comparable to Jackson in that it provides serialisation of JSON, but unlike Jackson, it works without reflection. Instead it's implemented as a compiler plugin and a runtime library. The compiler plugin automatically generates the visitor code for the data classes, while the runtime library uses the generated code for serialisation without reflection. This massively improves runtime performance.
JUnit	JUnit is probably the most widely used unit testing framework in the Java ecosystem. We use it to write our unit tests with it, as it provides convenient integration into Ktor.
Bouncy Castle	Bouncy Castle is an extensive cryptography library for Java (and C#). We use it extensively in the signing service: to build CMS messages, to generate RSA keys, to work with X.509, to create OCSP and Time-stamp Protocol (TSP) requests and read the responses, and more. Without this library we couldn't have achieved what we did. Unfortunately, the documentation isn't very good (sometimes non-existent), and we've spent a lot of time figuring out how to use it correctly.
auth0 jwt/jwks	Java implementations of the JWKS and JWT standards, used for OIDC integration and id_token verification. We haven't looked for alternatives because this library works well, has on-point documentation, and is MIT-licenced.
jsoup	Jsoup is a library for working with HTML. It provides a well-designed API for extracting data from the Document Object Model (DOM), using jquery-like methods. We use it to submit the login form on the Keycloack IDP in the unit tests. We can't POST the form directly because there is a hidden CSRF token field in the form, whose value we have to extract. This is what jsoup is good at.
protobuf	This is the reference implementation of protocol buffers. We use it to build the signature file.
Koin	Insert-Koin is a pragmatic, light-weight dependency injection framework for Kotlin. What distinguishes it from other Dependency Injection (DI) frameworks, and in our opinion makes it better, is that it uses purely functional resolution. There is no reflection, no proxies, and not even code generation.

Table 11.3.: Software libraries used in the development of the signing service

Name	Description
github.com/sirupsen/logrus	A structured logger, that is API compatible with the Go standard library logger. Used so we don't have to do the log formatting all by ourselves.
go.mozilla.org/pkcs7	Library for parsing and verifying PKCS7/CMS and RFC3161 [24] timestamps.
gopkg.in/square/go-jose.v2	Implementation of the JavaScript Object Signing and Encryption (JOSE) set of standards like JWT and JWK. Used for extracting the X.509 certificates from the JWKs in the signature data and for verifying the ID token signature.
github.com/coreos/go-oidc	Used for verifying and parsing the OIDC ID tokens.
github.com/golang/protobuf	Reference implementation of protocol buffers, which are used in our signature format.
golang.org/x/crypto	Extended part of the Go standard library. Mainly used for parsing and verifying OCSP responses.
github.com/shurcooL/vfsgen	Generates go "code" from static files and makes them accessible via a virtual filesystem. This is used to work around Go's limitation of not being able to embed files in the compiled binary.

Table 11.4.: Software libraries used in the development of the verification service

npm The node package manager is used to install the TypeScript compiler `tsc`, which type-checks, then translates the TypeScript source code into JavaScript. We also use webpack, a tool which allows us to combine all of the compiled JavaScript files into a runnable single bundle file. The code in that bundle file is minified, a process in which the code is transformed in such a way as to be as small as possible. This reduces the amount of data that has to be transmitted over the network.

Cargo Cargo is the package manager for the Rust programming language. It downloads dependencies, compiles packages, and builds binaries ready for distribution. We use Cargo to build the WASM-based in-browser hashing binary, which in turn is used by the frontend of the signing server as well as the verifier.

11.1.8. Software Packaging

For the signing service, we use the shade plugin for Maven. This plugin allows us to package our artifact into a Java Archive (JAR), including all runtime dependencies and resources.

For the signing server, the resource files that have to be included into the production-ready JAR are the frontend files (HTML, CSS, JavaScript (JS) and WASM), as well as the Ktor configuration file. This allows for simple deployment: all that's needed on the target server is a Java Runtime Environment (JRE), and the rest is contained in the JAR. The admin doesn't even have to set up Tomcat, and running the server is as easy as executing `java -jar signingserver-shaded.jar`.

Go uses static linking by default, it even includes its own VM into the binaries it builds. This has the advantage of vastly simplified deployment: all anyone needs to do is to copy the binary and run it. The downside is that the binary is somewhat larger.

However, Go's build system doesn't support including additional files in that single binary. As a workaround, we've used the `vfsgen` library. For a given set of files and directories represented as the `http.FileSystem` interface, it generates Go code that statically implements that interface, which in turn is used just like any other Go package. The end result is one single executable file which contains everything it needs, even its own runtime (unlike the fat JAR which doesn't work without a JRE).

11.1.9. Custom Docker Image for Rust+Webpack Build

Gitlab Continuous Integration (CI) uses Docker images as the build environment in which to compile and package artifacts. For all but one of our components we were lucky enough to find pre-built images providing what we

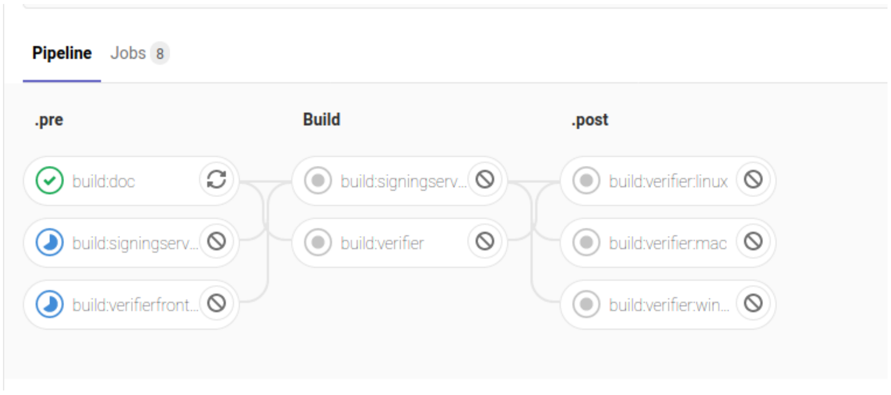


Figure 11.3.: Gitlab CI pipelines combining different build tools and the artifacts they produce

needed (for example, a Maven 3 environment). Unfortunately it seems with Rust + Webpack we're a bit on the forefront of things, and no one's published a usable Docker image for that build configuration to Dockerhub yet.

This is why we built our own image on top of a base Ubuntu+Rust image by adding the following:

1. Installing `curl`
2. Installing wasm-pack. This tool implements the workflow necessary for building Rust into WASM, generating the JS and TypeScript binding code, and generates npm packages ready to be included into our project.
3. Installing `cargo-generate`, a scaffolding tool which generates the boilerplate necessary for Cargo to work.
4. Installing npm, the node package manager.

We published this image onto Dockerhub as `izolight/rust-webpack`, hoping that others might find it useful.

11.1.10. Combined Build Process

Since we use so many languages, tools and frameworks, the build process gets a bit complicated. For this reason we document it here.

1. npm downloads and installs the webpack and TypeScript toolchains
2. wasm-pack calls Cargo, which downloads and installs the Rust dependencies, and compiles the WASM binaries
3. wasm-pack generates glue code allowing for WASM to JS bindings
4. webpack starts `tsc`, which typechecks, then translates the TypeScript source code into JavaScript. Then, it combines the generated JS from `tsc` as well as wasm-pack into a minified bundle file, and packages it together with the HTML and CSS files.
5. The verifier uses go `generate` to compile the frontend files it requires into a Go module, which were built in steps 1-4
6. The signing server copies its frontend files built in steps 1-4 into its resources directory, then calls Maven to build its fat JAR. Maven downloads the dependencies, compiles the Kotlin source code, generates the source code for marshalling using `kotlinx.serialization`, generates the source code for the protobuf marshalling classes, and packages everything together into a fat JAR, including the frontend files from the resources directory.
7. The verifier calls go `build` three times, once respectively for GNU/Linux, Mac OSX and Windows, resulting in the binaries for each platform.

Figure 11.3 illustrates the combined build process as implemented in Gitlab CI. Since there are quite a few steps to the build process and we have 8 Gitlab runners available, we have parallelised it as much as we could. This way we were able to reduce build times from 20 minutes to around 4 minutes.

11.2. Implementation Components

11.2.1. Design Principles

We've split the implementation of the Signing Service and the Verifier into small, replaceable modules with clearly defined interfaces. We've done this to achieve clear separation of concern and loose coupling, in the spirit of the Law of Demeter [14].

The Law of Demeter is a well-known heuristic that states that any module should not know about the internals of the objects it manipulates. It is a special case of Loose Coupling. Objects should hide their data, and expose operations. This makes it easy to add new types of objects without requiring changing existing behaviours. It also makes it hard to add new behaviours to existing objects. Data structures should expose data and have no significant behaviour.

Loose Coupling refers to the degree of knowledge that one component has of another. Structuring programs to consist of components that know little of one another results in easy-to-understand, easy-to-test code. Developers new to the project can start with work on a small module and don't need to understand the whole system. Refactoring (or replacing) the implementation of a component becomes easy, as there are clear boundaries, and changing the implementation of one component does not affect the others as long as the boundary contract remains satisfied.

Inversion of Control enables us to remove the few interdependencies remaining in the modules: knowing about each other. A component should not care about where, how and why another component is implemented, it should only care about being provided the behaviour it requires. Inversion of Control through Dependency Injection allows each component to declare its dependencies through interfaces, and having it supplied the implementation of that interface from the system it is part of, without knowing anything whatsoever about that implementation.

Applying these methodologies doesn't guarantee clean code. Nothing does. However, in our experience, modularisation, decoupling, and separation of concerns is the best single principle to apply to software development to achieve some level of code quality, and to keep the complexity of a growing system manageable.

11.2.2. Components of the Signing Service

There are three groups of components in the signing service:

- The views: They implement the REST interface to the outside world.
- The services: Replaceable components providing functionality needed by the views in order to be able to meet their purpose.
- Marshalling: Classes which provide type-safe serialisation and de-serialisation of input and output data, as well as validation.

Name	**Description**
SigningKeysService	Generates signing keys and associated Public Key Cryptography Standard 10 (PKCS10) certificate signing requests, signs CMS messages using the CA-signed certificate and its key. The signing key never leaves this service. It can be backed by a HSM.
CAService	Provides an interface to the PKI services offered by a CA. It submits PKCS10 certificate signing requests to the CA and returns the signed certificate.
TimestampingService	Provides an interface to the RFC 3161 trusted timestamping services. Given the data to timestamp, it constructs a timestamping request (including a nonce for increased security), submits it, and returns the timestamping response. The response is a CMS containing the signature of the hash of the data as well as the certificate chain of the signing certificate.
OIDCService	Implements OIDC-related functionality. An OIDC client is configured using a discovery document usually located under `.well-known/openid-configuration`. Upon instantiation, this service fetches the discovery document as configured and set itself up. It validates JWT id_tokens, and it constructs authorisation endpoint URLs for the frontend based on the discovery document.
NonceGenerator	Generates nonce values in a secure manner.
SecretService	Stores the static server secret used for authentication of the nonce values as described in 10.4.1, and operates on it. The server secret never leaves this service, and it could be backed by a HSM or other hardware crypto device for increased security. At the moment, the secret is generated randomly at instantiation time, and not configurable nor persisted, for security. We would like to apologise for the pun in the name.

Table 11.5.: Services and their responsibilities in the signing service

For a Unified Modeling Language (UML) component diagram showing a simplified overview of the components, see figure 11.4.

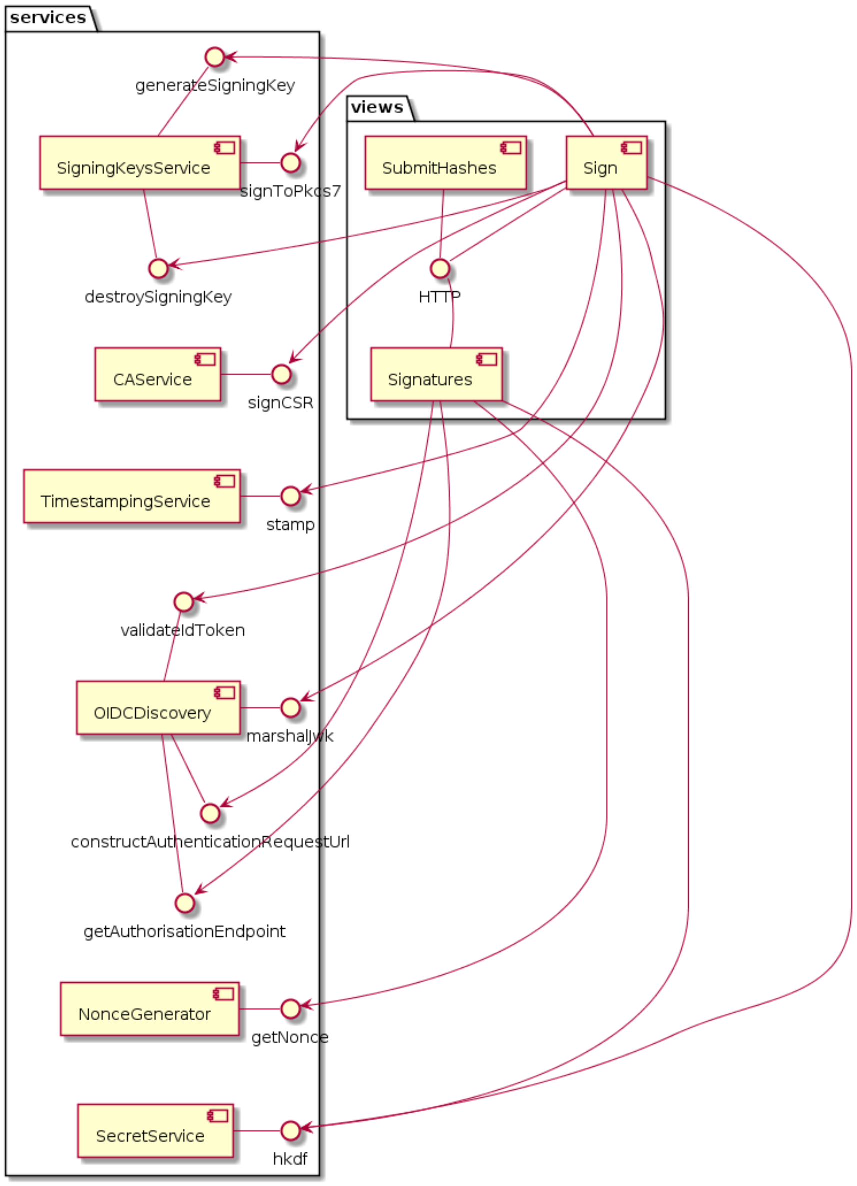


Figure 11.4.: Signing Service Components

11.2.3. Components of the Verifier

The verifier is divided into the following packages:

- verifier: Contains all the verification logic. A detailed list with the components is provided in table 11.6.
- verifier/api: The http router is set up here together with the handlers.
- verifier/config: Configuration files (for now just our root CA, which isn't in the system truststore), which will be embedded into the binary via vfsgen.
- verifier/static: The generated frontend files which will be embedded into the binary via vfsgen.
- verifier/pb: The generated protobuf serialisation and some helper functions for pretty printing the values.
- verifier/cmd: The entrypoint with the `main()` function, which wires together the other packages and starts the web server.

Name	**Description**
Verifier	Takes the outer deserialised protobuf file and the submitted hash and delegates the verifying of signature to the other verifier. All other verifiers will be started in go routines so they can start verifying as soon as possible. Some verifiers depend on output of other verifiers, which will be injected into them via calls from the main verifier that reads from or writes to channels in the other verifiers. It will finish either when one of the verifiers encounters an error or all of them have finished.
TimestampVerifier	Parses the RFC 3161 timestamps and verifies them. If multiple timestamps are included, the outermost (last) timestamp timestamps the previous one until the innermost (first) contains the timestamp for the signature container.
SignatureContainerVerifier	Verifies the PKCS7 which envelops our signature data.
SignatureVerifier	Checks if the submitted hash is included in the salted hash list and if the list matches the ID token nonce.
IDTokenVerifier	Verifies if the ID token is valid(just in a OIDC manner, our use of the nonce will be verified in the SignatureVerifier).
LTVVerifier	Is called by the other verifiers to check if the included LTV data (if any) verifies for the used certificates.

Table 11.6.: Services and their responsibilities in the verification service

For a UML component diagram showing a simplified overview of the components, see figure 11.5.

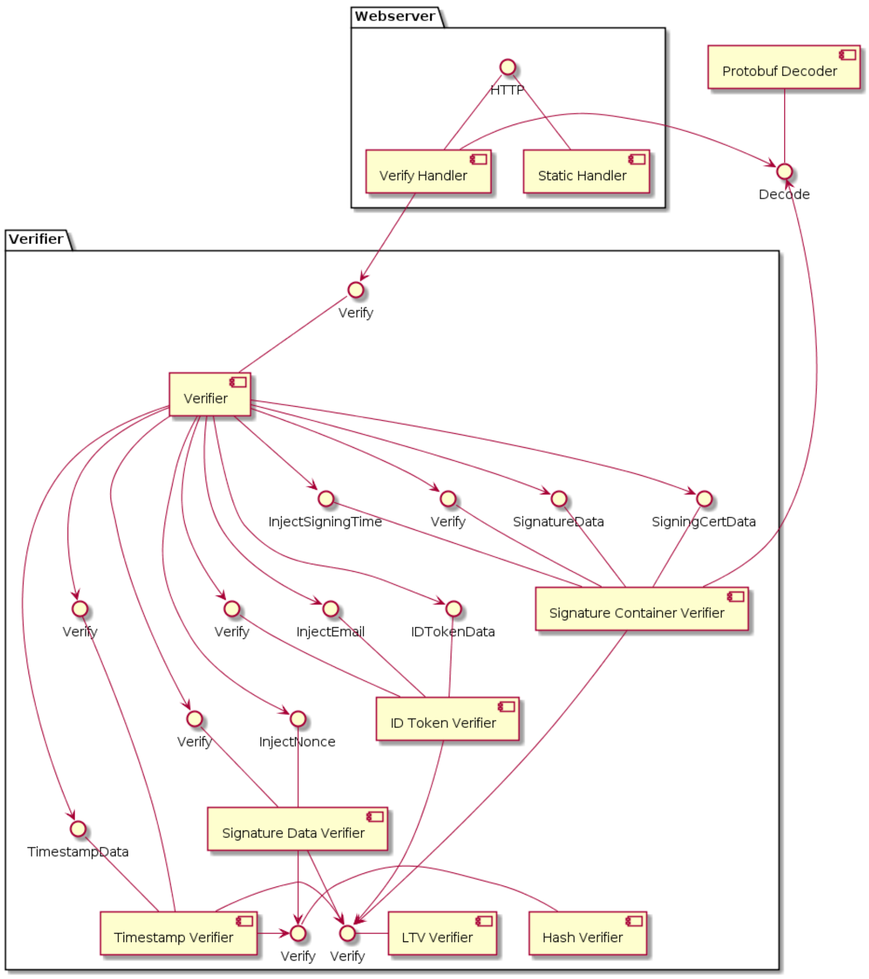


Figure 11.5.: Verifier Components

11.3. Finding services fitting our requirements

For our remote signing service to work, we need an IDP satisfying the requirements documented in section 9.1, a TSA willing to timestamp our signatures, and a CA with an OCSP responder that will sign our certificates.

11.3.1. Finding a TSA

Searching for publicly available TSAs, we've found a list as part of the software-library `python3-rfc3161ng` [22]:

- `http://freetsa.org/tsr`
- `http://time.certum.pl`
- `http://timestamp.comodoca.com/rfc3161`
- `http://timestamp.geotrust.com/tsa`
- `http://timestamp.globalsign.com/scripts/timstamp.dll`

However we went with the SwissSign TSA, as it seemed the least untrustworthy to us.

It worked as expected the first time we tried requesting a timestamp from it. We used the `openssl ts` command line program. Furthermore, to our knowledge, it is available free of charge, and during our development and testing they didn't restrict us as to the number of timestamps we requested, which was good enough for our Proof Of Concept (POC).

It can be found at `http://tsa.swisssign.net`.

We construct the timestamping requests using the `TimeStampRequestGenerator` class from BouncyCastle [5], which works well.

11.3.2. Finding an IDP

Finding an IDP for the Swiss population providing such strong guarantees as required was not easy, we were able to find only one: SwissID. We searched online for documentation from them on how we could integrate their IDP into our project, but we could find none. So we sent them an e-mail and asked, and got a phone call from one of their key account managers, who seemed to understand little of our actual request but wanted to send us a whitepaper. That turned out to be marketing material with very little technical information and quite useless to us. Fearing the sudden appearance of a very large bill we stopped communicating with them.

For the purposes of our POC and in the interest of having a working demo soon, we don't really need an IDP with actual guarantees, we just need any working OIDC IDP.

That led us to search for publicly available IDPs that we could use free of charge, with self-onboarding and readily-available online documentation.

We found the following:

- Github: Doesn't support OIDC, only OAuth
- Yahoo, Paypal, Salesforce, Phantauth, Okta, Google: Don't use X.509
- Microsoft: Uses X.509, but the intermediate certificate is self-signed. Why they're doing that is unclear to us, it seems futile.

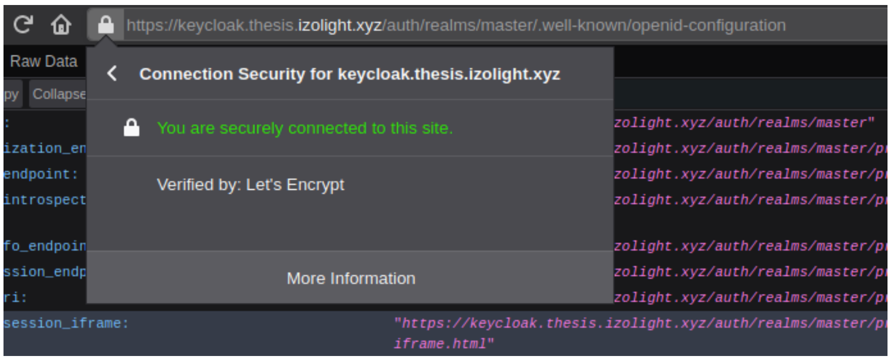
So, in the end, unfortunately we couldn't find any free-of-charge IDP that we could use. Thus we were forced to set up our own. For this purpose we selected Keycloak, a powerful and open source Identity and Access Management (IDM), developed by Red Hat [12]. Keycloak is highly configurable and allowed us to run exactly the IDP we needed. For the duration of our thesis, it can be found at `https://keycloak.thesis.izolight.xyz/auth/realms/master/.well-known/openid-configuration`.

Infrastructure setup for the IDP

It would've been easy to run Keycloak on our local machines, and for the demo this would've been enough, but we wanted to do it properly, with an internet-accessible IDP. To achieve this, we needed a server, a certificate for HTTP over TLS (HTTPS), and Domain Name System (DNS) entries.

For the server, we chose Scaleway's development server simply because it costs as little as 3 EUR per month [29]. For certificates, we used the excellent Let's Encrypt CA [15], which is a trusted CA issuing domain validation certificates free of charge.

This only left us with DNS. Fortunately Gabor Tanz already runs his own DNS server for other purposes (and also owns a domain), so we simply added the appropriate subdomain entries and we were ready to proceed.



https://keycloak.thesis.izolight.xyz/auth/realms/master/.well-known/openid-configuration

Connection Security for keycloak.thesis.izolight.xyz

You are securely connected to this site.

Verified by: Let's Encrypt

```
:                 zolight.xyz/auth/realms/master"
ization_en        zolight.xyz/auth/realms/master/p
endpoint:         zolight.xyz/auth/realms/master/p
introspect        zolight.xyz/auth/realms/master/p

fo_endpoin        zolight.xyz/auth/realms/master/p
ssion_endp        zolight.xyz/auth/realms/master/p
ri:               zolight.xyz/auth/realms/master/p
session_iframe:               "https://keycloak.thesis.izolight.xyz/auth/realms/master/p
                              iframe.html"
```

Figure 11.6.: Firefox trusting the certificate issued by the free and open CA Let's Encrypt

11.3.3. Finding a CA

Unfortunately, Let's Encrypt doesn't issue signing certificates [16], only domain validation certificates. We didn't spend much time searching for other CAs since we expected to find none that issues signing certificates in an automated way and free of charge. Consequently we proceeded to set up our own CA, as we did with the IDP. This meant more of an up-front investment of time, but better predictability of efforts since everything would be under our control, if we'd run into any issues.

To set up our own CA, we selected Cloudflare's `cfssl` PKI toolkit [11]. There was no evaluation of which software to use, since at this point we were behind schedule (because we spent more time on the concept part than we planned in order to get multi-signatures right). We weren't keen on spending much more time on evaluating the CA toolkit to use because it'd be relevant for the POC only anyway, and it didn't matter too much as long as it fulfilled its purpose. On top of that we already had some knowledge of `cfssl`.

On the same server we'd already spun up to host our `Keycloak` IDP, we installed `cfssl`.

11.3.4. Real-time Updating of OCSP Responses for Certificates Issued

`cfssl` keeps a database of the certificates it issues, which mainly consists of the subject name hash and the certificate's serial number. Periodically, the OCSP responder software checks the certificates issued, and generates OCSP responses for these certificates, which it in turn stores in its own database. This is done for perfomance reasons, as generating an OCSP response would be too costly.

Ordinarily this wouldn't be a problem, but for our purposes we need the OCSP responses to be available immediately, since we generate, sign and destroy the signing key in a manner of seconds. The fastest `cfssl` is able do update its OCSP response database is once per minute, which is way too slow for us. Interactive users can't wait for 60+ seconds for signature generation.

So we expanded `cfssl`'s code to generate the OCSP responses real-time, that is, as soon as a certificate is issued and stored in the database, the associated OCSP response is generated and stored in the responders' database.

We have found many people on the internet asking for the same feature, hitting the same problem we did. For this reason we will submit a pull request to the `cfssl` project with our enhancements, hoping that others might find it helpful.

Separation of services using Docker containers

In order to have some separation between the different pieces of software we decided to use Docker. Docker makes the deployment of the software easier as we don't have to install software and their dependencies, which may conflict on the host system.

Table 11.7 contains a complete list of Docker containers running on our server. For a graphical representation, please see figure 11.7.

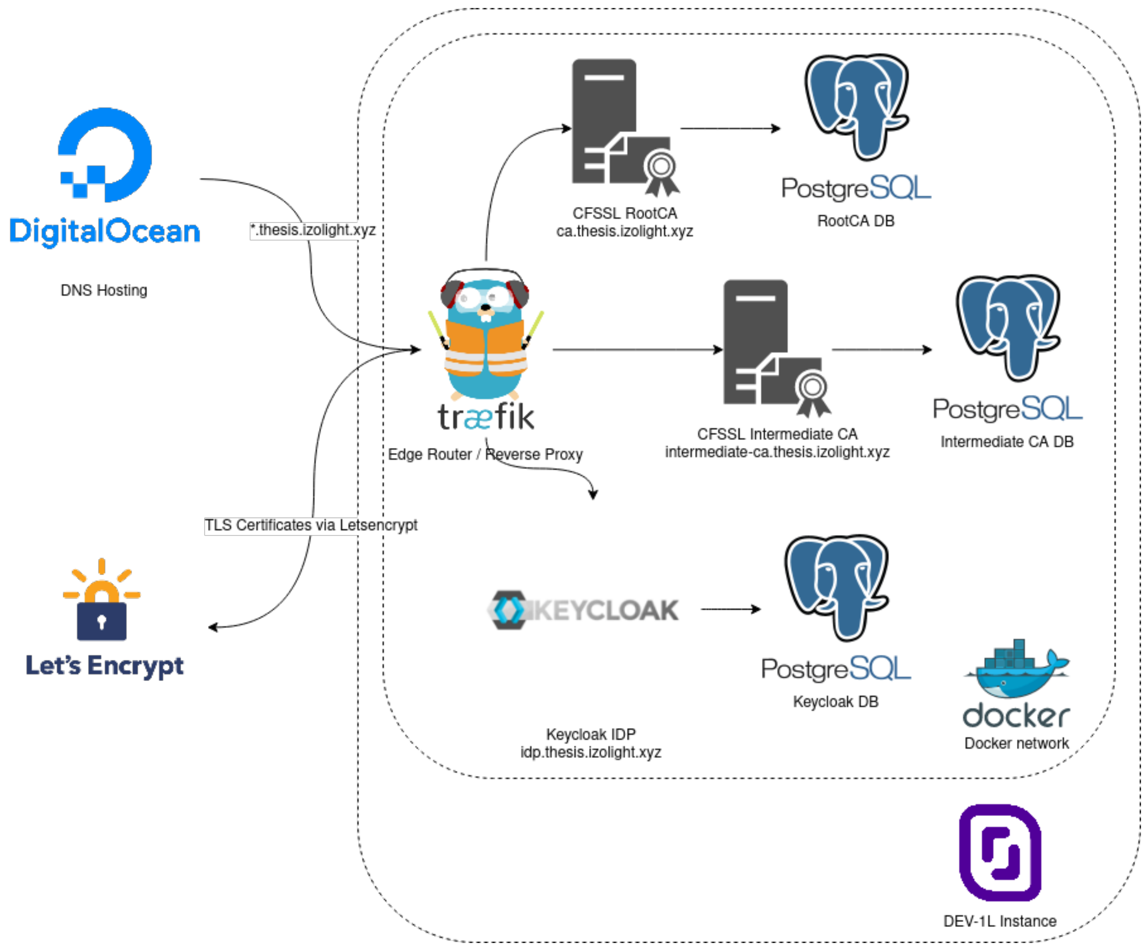


Figure 11.7.: Infrastructure Big Picture

traefik is an open-source edge router implementation. It receives outside requests on behalf of the actual system, finds out which service is responsible to handle them, and then passes them on. It is able to automatically discover configuration and provide its routing accordingly, and it can act as a HTTPS termination point. This helps lessen the load on backend components. In our case, it takes HTTPS requests, figures out which container is responsible for handling it, and passes it on as HTTP (thus terminating TLS).

Keycloak is the IDP, and keycloak-db contains its PostgreSQL database.

cfssl-root and `cfssl-intermediate` contain the respective root and intermediate certificates, and serve certificate signing requests. The containers with the -db suffixes run their PostgreSQL databases, same as for keycloak. The intermediate CA exposes a REST API used by the signing server.

cfssl-root-ocsp and `cfssl-intermediate-ocsp` contain the OCSP responders, which handle OCSP requests for certificates issued by `cfssl-root` and `cfssl-intermediate`.

11.4. Method of Work

Two people hacking away at a shared project each on their own rarely works out well. If we're lucky the code produced might even work, but it's probably going to be of low quality, and there's risk of duplication of effort, coordination and communication difficulties, as well as missing deadlines.

Name	**Command**	**Ports**
traefik	/entrypoint.sh –docker...	0.0.0.0:80->80/tcp, 0.0.0.0:443->443/tcp
keycloak	/opt/jboss/tools/docker-entrypoint...	127.0.0.1:8080->8080/tcp, 8443/tcp
keycloak-db	docker-entrypoint.sh postgres	5432/tcp
cfssl-root	cfssl serve -db-config=/data/cfssl/...	8888/tcp
cfssl-root-db	docker-entrypoint.sh postgres	127.0.0.1:5432->5432/tcp
cfssl-intermediate	cfssl serve -db-config=...	8888/tcp
cfssl-intermediate-db	docker-entrypoint.sh postgres	127.0.0.1:5433->5432/tcp
cfssl-root-ocsp	cfssl ocspserve -address=0.0.0.0...	8888/tcp
cfssl-intermediate-ocsp	cfssl ocspserve -address=...	8888/tcp

Table 11.7.: Docker containers running on our POC server

This is why before starting work, we agreed upon a minimum of methodology.

11.4.1. Version Control for Source Code

We put all source code under version control, in a shared repository. This way each student can see what the other person is working on, and the thesis advisors can keep an eye on what we're doing. The commit log comes in handy for documenting the work timeline after the fact as well, and it's a way of backing up our work. Thankfully Bern University of Applied Sciences (BFH) offers its students a Gitlab instance [4], so we will be using that.

11.4.2. Standardised Build Tools and Environment

Few things are more annoying in a developer's life than troubleshooting build failures. One developer adds a dependency, installs it on their machine, and forgets to tell the others. Another developer updates their system and now uses a later version of a library, causing incompatibilities.

Furthermore, our advisors most likely aren't very keen on installing the toolchains necessary for building our artefacts on their own machines, they'd rather examine the results directly.

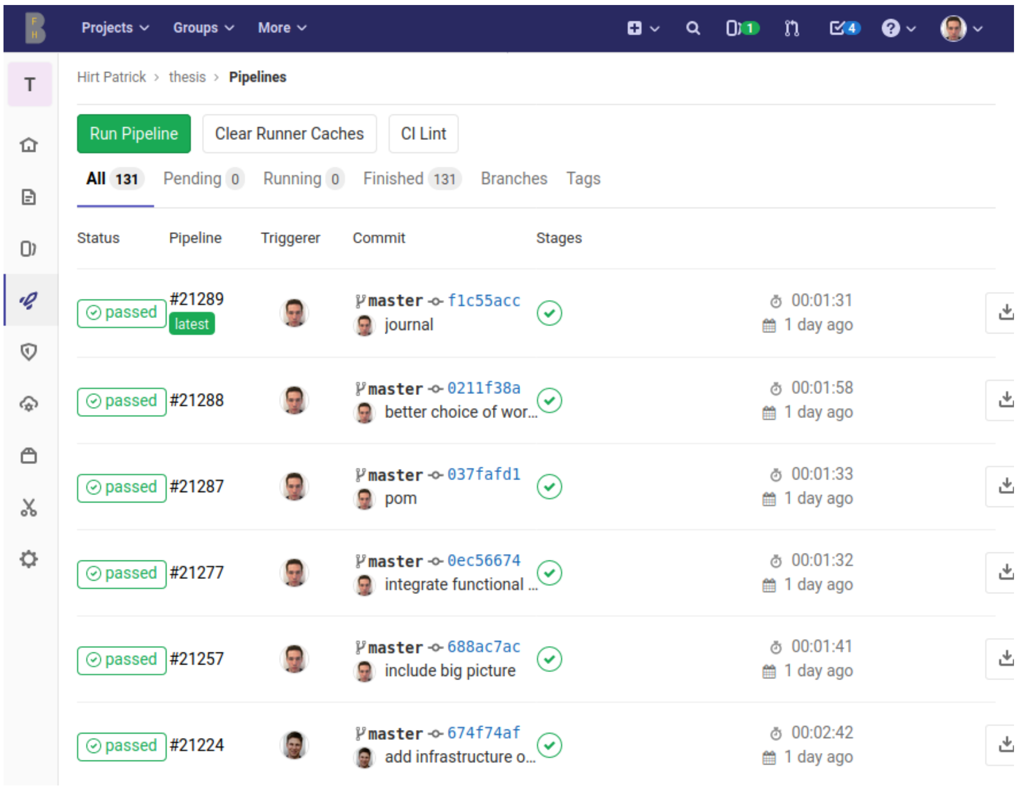
This is why a standardised, centralised build environment is a good thing to have. Since BFH is so kind as to offer us Gitlab runners toghether with their Gitlab instance [4], we will use that. We've set up build pipelines based on Docker images to build our components as well as the documentation each time new commits are pushed to the master branch. Figure 11.8 contains a screenshot of what this looks like in Gitlab's web interface.

11.4.3. Subdividing Work Packages

To be able to plan and schedule our work efficiently, we divided the work packages into user stories, as is customary in SCRUM. We again used Gitlab for filing, planning and tracking the user stories. Gitlab's issue system isn't exactly meant for use with SCRUM, but we've managed to adapt it as follows:

- Issues are used for User Stories
- Labels are used for tracking progress ("todo", "doing", "done")
- Milestones are used for Sprint boundaries

Together with the issue board view used as a Storyboard view, this works quite well for smaller projects such as ours. Figure 11.9 contains a screenshot of what this looks like in action.



Projects Groups More

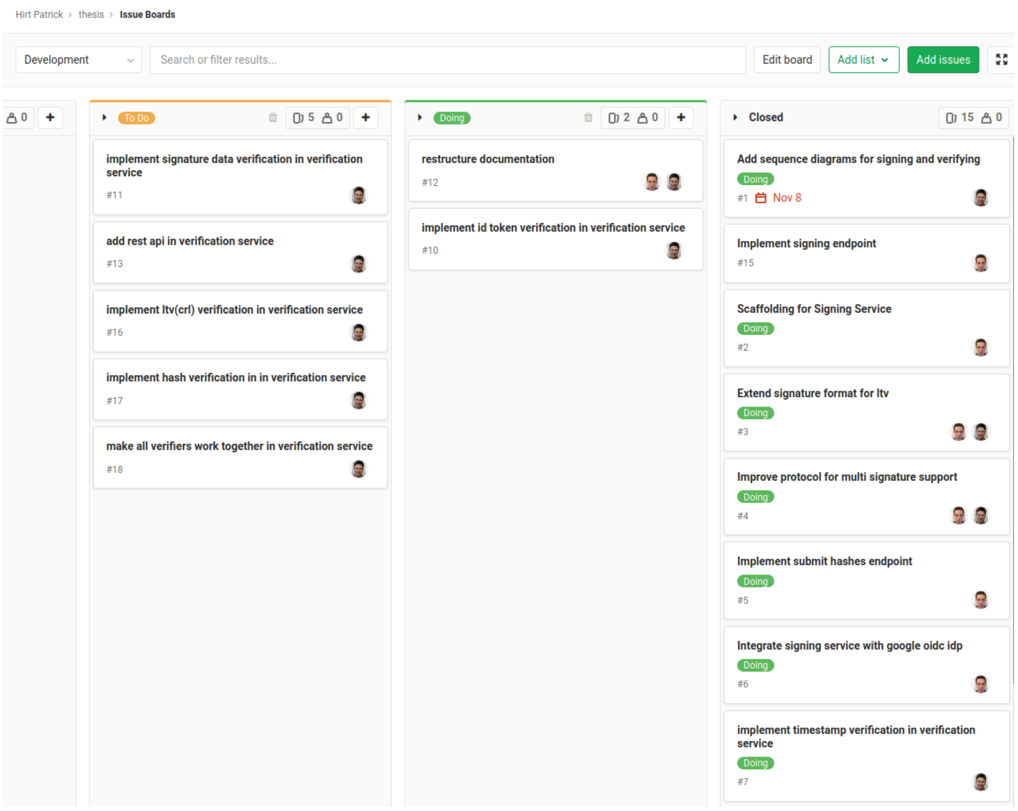
Hirt Patrick › thesis › **Pipelines**

Run Pipeline | Clear Runner Caches | CI Lint

All 131 | Pending 0 | Running 0 | Finished 131 | Branches | Tags

Status	Pipeline	Triggerer	Commit	Stages		
passed	#21289 latest		master f1c55acc journal	✓	00:01:31 1 day ago	
passed	#21288		master 0211f38a better choice of wor...	✓	00:01:58 1 day ago	
passed	#21287		master 037fafd1 pom	✓	00:01:33 1 day ago	
passed	#21277		master 0ec56674 integrate functional ...	✓	00:01:32 1 day ago	
passed	#21257		master 688ac7ac include big picture	✓	00:01:41 1 day ago	
passed	#21224		master 674f74af add infrastructure o...	✓	00:02:42 1 day ago	

Figure 11.8.: Gitlab CI Jobs



Hirt Patrick › thesis › **Issue Boards**

Development | Search or filter results... | Edit board | Add list | Add issues

To Do (5, 0)
- implement signature data verification in verification service #11
- add rest api in verification service #13
- implement ltv(crl) verification in verification service #16
- implement hash verification in in verification service #17
- make all verifiers work together in verification service #18

Doing (2, 0)
- restructure documentation #12
- implement id token verification in verification service #10

Closed (15, 0)
- Add sequence diagrams for signing and verifying — Doing — #1 Nov 8
- Implement signing endpoint #15
- Scaffolding for Signing Service — Doing — #2
- Extend signature format for ltv — Doing — #3
- Improve protocol for multi signature support — Doing — #4
- Implement submit hashes endpoint — Doing — #5
- Integrate signing service with google oidc idp — Doing — #6
- implement timestamp verification in verification service — Doing — #7

Figure 11.9.: SCRUM Storyboard

11.4.4. Tracking Progress

In addition to the commit log and the Storyboard, we kept an informal work log in the repository so that our advisors can check up on our progress effortlessly any time they want. On top of that we met with them regularly during the whole work, discussing the progress, potential issues and of course to receive regular feedback.

11.4.5. Test Coverage

For the signing server, we achieve a test coverage of 86.4%, as measured by the Intellij coverage runner. This value, while respectable, isn't that as high as we'd like. We inspected the coverage report and found that this is mainly due to deserialisation code in data classes that must take fields that are never used. Excluding these classes, we achieve a coverage of 94.9%.

For the verification server, we achieve a test coverage of 85.3%, as measured by the Go test tool. Most of the uncovered parts are due to the same reasons as in the signing server tests.

12. CSC Standard

12.1. CSC Specification

The Cloud Signature Consortium (CSC) has been formed to standardise cloud-based digital signatures, while meeting the EU's regulation for signatures (electronic IDentification, Authentication and trust Services (eIDAS)). The consortium consists of of all kind of members from different industries: Software companies like Adobe, the German Bundesdruckerei, Certificate Authorities (CA) like QuoVadis, but also academic institutions like the Technische Universität Graz.

The result of this consortium is an API specification for remote electronic signatures and remote electronic seals. The specification is published as a PDF document, as well as an Open Application Programming Interface (OpenAPI) specification, and a JSON schema [3].

12.2. Comparison

We have examined the standard and compared it to our solution. We document the findings in this section.

12.2.1. Information Leakage to IDP

One main difference is that in the CSC specification, the IDP gets to know the hash that will be signed, and how many documents will be signed through the `numSignatures` and hash parameters in the credential authorisation, which has a slight impact on privacy and is a violation of the least information principle. In our solution, this problem doesn't exist.

12.2.2. Information Leakage to Signing Service

The CSC standard allows for the document to be signed to be transmitted to the signing service. We see absolutely no reason at all for this to be allowed, since in any case, only a hash of the document is signed. There is no scenario - be it in our solution, or Adobe's - where the signing service is required to recieve the full document. This is a violation of the least information principle and a break of privacy. In our solution, this problem doesn't exist.

12.2.3. Missing Separation of Concern

The CSC standard allows for the IDP and the signing service to be the same system, controlled by the same organisation. We find this highly problematic, as this means a single entity is able control all the parts needed to create a signature. To be fair, the EU allows this as well. We strongly disagree with this, and explicitly forbid it in our specification. There should never be a single organisation in complete control, especially not one with a profit motive.

12.2.4. Weak Authentication

The CSC standard allows for HTTP Basic or Digest authentication. Saying this is wholly inadequate for creating legally binding signatures would be an understatement.

12.2.5. No Use of Standard Protocols

The CSC standard doesn't use standard protocols like OIDC or even Security Assertion Markup Language (SAML), which is a disadvantage: Before any IDP can be used, it has to implement the extensions specified by the CSC standard.

Another difference is that SAD is returned to the client, which has a defined validity period and allows for further signatures to be created without re-authorisation, which means that an attacker who is able to steal the SAD could sign arbitrary documents. In our solution, this isn't possible.

12.2.6. No User Controlled Signing Process

With the CSC standard, there is nothing stopping the TSP creating signatures without the user's knowledge or consent. The TSP controls all parts necessary. The SAD looks good on paper, as it suggests the user is in control of their key, but in reality there is no security whatsoever to it. In our solution, the signing service cannot create a signature without the direct authorisation of the user through the IDP, which we forbid to be controlled by the same organisation.

12.3. Conclusion

We find the CSC standard to be significantly less secure than our proposal. There are significant privacy and security issues. Personally, we would not use any solution based on this standard.

13. Yubikey HSM2

13.1. Introduction

As we want to make our service as secure as possible we want to eliminate saving of the signing keys on disk. To this end commonly a HSM is used. Unfortunately most commercial HSM are financially out of reach for use in this thesis. Thankfully, Yubico offers a relatively inexpensive ($650) Universal Serial Bus (USB) powered solution: the YubiHSM-2 [35]. We were offered one from G. Hassenstein to investigate whether it could be used in our work.

13.2. Main Features

The yubihsm-2 allows us to generate and store the signing keys on the device and perform the cryptographic operations there without the keys ever leaving the device. Another capability of the yubihsm-2 is remote management and operation. In addition of using the device on the host where it is attached via a standard Public Key Cryptography Standard 11 (PKCS11) interface, it is possible to connect to it over the network as well, which would enable us to realise a dedicated signing server.

It supports modern standards like SHA-256 for hashing, up to 4096 Bit RSA in Probabilistic Signature Scheme (PSS) mode for signing, and Elliptic Curve Cryptography (ECC)-based signatures in Elliptic Curve Digital Signature Algorithm (ECDSA) with many different curves and Edwards-curved Digital Signature Algorithm (EdDSA) using curve25519 as well. The full specifications can be found on the website [35].

13.3. SDK

Yubico offers a Software Development Kit (SDK) [35] for Linux (Fedora, Debian, CentOS, Ubuntu), macOS and Windows. The SDK consists of a C and Python library, a shell for configuring the HSM, a Public Key Cryptography Standard (PKCS)#11 module, a connector for accessing it over the network as well as a setup tool and code examples with documentation.

In the Windows version a key storage provider is also included.

13.3.1. Connector

The yubihsm-connector provides an interface to the yubikey via HTTP as the transport protocol. Upon inspecting it, we found that the protocol isn't RESTful, and the payload seems to be binary. The connector needs to have access to the USB device, but incoming connections to the connector don't need to originate from the same host. The sessions between the application (not the connector) and the YubiHSM 2 are using symmetric, authenticated encryption [35].

13.3.2. Shell

The yubihsm-shell is used for configuration of the the device. The full command reference can be found on the yubico website [36].

13.3.3. libyubihsm

`libyubihsm` is the C library used for communication with the HSM. It's possible to communicate with the device using a network or directly over USB. The device only allows one application to access it directly as exclusive access [17] is required.

This means, that even if we want to have the signing application run on the same server as the YubiHSM is attached to, it is probably better to use the HTTP connector as this enables multiple instances of the application to access it concurrently.

13.3.4. python-yubihsm

The Python library either needs to have a connector already running or direct access via USB. Otherwise it seems to offer the same features as the C library, but we haven't verified this exhaustively.

13.3.5. PKCS#11 module

With the PKCS#11 module yubico provides a standardised interface to the HSM. The module needs a running connector and doesn't allow USB access. Not everything in the standard directly translates to the capabilities of the HSM, so some values are fixed [20].

13.4. Conclusion

Using a HSM would definitively make our application more secure. Unfortunately the SDK only provides libraries for C and Python and not for Kotlin.

As the HTTP interface isn't documented and most likely not intended to be used directly, we would be forced to reverse engineer it for use with Kotlin, which would probably take too much time for use in this thesis. The long-term stability of such an approach would be questionable, as yubico could change the protocol without warning (and they probably will, as they won't expect people to be using it directly).

A possible workaround for this would be to use the PKCS11 API and bind it to the Java Cryptography Architecture (JCA).

In conclusion, using the YubiHSM would improve security, but due to time constraints we will make it an optional goal. We will however aim to make the signing part of our application pluggable (standardised interface allowing for differing implementations, for example by using a factory pattern, or DI) so that anyone can easily add support for a HSM later, be it Yubico's or another manufacturer's.

14. Evaluation

In this chapter, we will review the objectives specified in the requirements, examine whether they've been achieved, and to which degree.

For a detailed specification of objectives, please see chapter 2. In the interest of brevity, the full specification won't be repeated here.

14.1. Functional and Non-Functional Requirements

Requirement	**Prio.**	**Done**	**Comment**
Authenticity of signature (4.2.1)	Required	Yes	Protected by at least three distinct hashes and signatures, once by signature on JWT, once by signature of document hash, and once by signature of timestamp
Integrity of document (4.2.2)	Required	Yes	Protected by hash and signature, same as 4.2.1
Verifiability of signature (4.2.3)	Required	Yes	Verifiable by standard technologies X.509 and JOSE, open specification and implementation, easily replicated
Non-repudiation (4.2.4)	Required	Yes	Strong authentication provided by IDP, and this identity assertion is part of the signature
Coupling of authentication & signing (4.2.6)	Required	Yes	Linkage with HMAC functions based on secure hash functions
Authentication protocol (4.2.7)	Required	Yes	Standard OIDC is used, no alterations or extensions necessary
Supported file formats (4.2.8)	Required	Yes	Detached signature ensures compatibility with any file format, past, present, or future
No unauthorised id. delegation (4.3.2)	Required	Yes	Without a valid JWT issued by the authorised IDP, the verifier won't accept the signature as valid. Thus, without the active help of the IDP, the signing server can't create a valid signature. It cannot even steal and abuse a valid JWT, since each token is bound to a specific document hash. All the IDP could do with a stolen JWT is sign a document the user wanted to sign anyways, since otherwise there would be no such JWT.
Random number generation (4.3.3)	Required	Yes	We use the operating system provided Cryptographically Secure Random Number Generator (CSRNG) exclusively.
REST API (4.3.4)	Required	Yes	The REST API is implemented fully as specified in 9.3.
Offline validation (5.0.3)	Required	Yes	Three standalone builds of the verifier program are provided, capable of offline verification.
Signing key security (4.3.1)	Optional	Yes*	The signing key is only held in memory, and at that for the shortest duration possible (typically less than two seconds), before being destroyed. It would be exceedingly difficult for an attacker to steal it. However, the use of a dedicated HSM could increase security further, which we didn't.
Protection of information (5.0.2)	Optional	Yes	No party receives more than the information they need. We explicitly forbid transmitting the document to be hashed to the signing server, unlike the CSC. Furthermore, we salt the hashes to shield them both from the IDP as well as from other recipients of multi-file signatures.

Device-local hashing (4.2.10)	Optional	Yes	Implemented through in-browser hashing using WASM.
Efficient signature file format (5.0.1)	Optional	Yes	Protobuf is one of the most efficient serialisation formats in existence, in most cases as dense as DER-encoded Abstract Syntax Notation One (ASN.1).
Bulk signatures (4.2.6)	Optional	Yes	Bulk signing or multi-signatures are accounted for in the concept and implemented.
Long-term validation (4.2.5)	Optional	Yes	LTV is accounted for in the concept and implemented by embedding all information necessary into the signature file, and by allowing for infinite lengths of TSP chains.
Code Quality (5.0.4)	Optional	Yes	Hard to measure objectively, but we think we've achieved a good level of code quality by strictly separating concerns, using mature libraries, naming classes, methods, arguments and variables well, and by ensuring we have excellent test coverage.
Ease of Use (5.0.5)	Optional	Yes	We don't force the user to jump through hoops. Signing and verifying is simple and straight-forward.
Reactive Design (5.0.6)	Optional	Yes	The frontend scales down so that it is usable on mobile just as easily as it is on desktop.

Table 14.2.: Achievement of Objectives

14.2. Use Cases

In section 6, we specified the use cases whose implementation we review here. For brevity, we won't repeat the full specification, instead focusing on the result.

Use Case	**Impl.**	**Comment**
Interactive Qualified Signatures (6.1.2)	Yes	Interactive Qualified Signatures work as specified.
Bulk Advanced Signatures (6.1.3)	No	Multi-file signatures are possible, but not with the reduced-security one-time login method.
Offline Validation (6.2.2)	Yes	Offline validation is implemented with the three standalone, per-platform builds of the verifier.
Online Validation (6.2.3)	Yes	Online validation is implemented with the verifier running on a server.

Table 14.4.: Coverage of Use Cases

14.3. Project Management

At the beginning of this work we allocated the time available to us to the work packages, and created a project timeline. There were some deviations from the plan, which we outline here.

14.3.1. More Time Invested Into Concept

We planned to have the specification complete to a degree it'd be ready for implementation at 14th October. We were ready by that date to begin implementation, but we weren't satisfied with our solution. The solution we had by then would've worked, and it would've worked rather well, but we wanted to improve it further (see chapter 10), which we did. We have achieved an efficient and secure solution, which we are proud of. Deviating from the project plan to achieve this is a trade-off well worth it.

14.3.2. Implementation Completed Later Than Planned

Because we invested more time that planned into the concept phase, the implementation was completed later than planned as well. With the original plan it was to be complete by 23rd December, in reality it was finished by 4th January.

14.3.3. Division of Work

As is defined in the SCRUM project management method, project members aren't assigned tasks head of time. Instead, before each Sprints starts, the whole team commits to achieving the Sprint goals together. During the Sprint, developers take responsibility for User Stories autonomously. This is why we didn't pre-assign them in the planning phase.

Since the project's finished now, we document which work package was completed by whom here.

Work Package	**Completed by**	**Comment**
Specification of Objectives (7.3.1)	Both	
Technical Specification (7.3.2)	Both	
Comparison with CSC Implementation (7.3.3)	Gabor Tanz	
Evaluation of Yubikey HSM (7.3.4)	Gabor Tanz	
Backend Implementation (7.3.5)	Both	Signing Service predominantly Patrick Hirt, Verification Service predominantly Gabor Tanz
Frontend Implementation (7.3.6)	Both	Frontend for Signing Service initially developed by Patrick Hirt, subsequently adapted for use with Verification Service by Gabor Tanz
Standalone Verifier Program (7.3.7)	Gabor Tanz	
Implementation Refinement (7.3.8)	Both	
Documentation (7.3.9)	Both	
Presentation (7.3.10)	Patrick Hirt	
Wall Chart and Article (7.3.11)	Both	Wall Chart Gabor Tanz, Booklet Patrick Hirt
Video (7.3.12)	Both	

Table 14.6.: Division of Work

All in all, the division of work was fair. No one person worked more than the other in any significant way.

15. Further Work

Due to time constraints given by the timeboxed bachelor thesis, we weren't able to explore all aspects of the remote signing service. We document these aspects and our thoughts on them here for future works.

15.1. Public Append-Only Data Structure

The main defence against malicious signature services - signing document files without the users' consent - is the integration of the authentication token signed by the IDP into the signature file (as described in 9.2.1). If the signing server were to create a signature file on their own they'd be unable to get such a token from the IDP, and this would be detected upon signature verification.

However, if the IDP were under the control of the same organisation as the signing service, or if the IDP were compromised as well, or if the user were to be tricked into authenticating with the IDP not knowing what they were doing, a malicious signing service could still create a valid signature not authorised by the user.

In order to defend against this, as an additional safety mechanism, we propose using a public append-only data structure (for example, a Merkle hash tree).

The signature service would be required to publish all signatures it creates by appending them to this data structure. This would allow everybody and anybody to see the signatures the signing server issues.

If the signing service were to create a signature without the users' consent, the signer could see this by inspecting the data structure, as there would be an entry for a signature there the signer doesn't remember creating.

If the signing service were to create a signature without publishing it into the data structure, any verifier could see this by inspecting the data structure, because the signature file would not be published in it.

15.2. Multi-Party Signatures

In order to facilitate signatures with multiple parties (for example, a standard apartent rental contract) we need to design a mechanism for generating and validating such signature schemes. There are many possibilities to implement this.

15.2.1. Nested Signatures

One possibility is that the subsequent signer signs the previously created signature file of the document instead of the document itself. The signing service will then generate another signature for the previous signature. The new signature would replace the original one, as it embeds it. This can be repeated as many times as necessary, creating a chain of signatures. This method would allow not only for an arbitrary number of signatures on the same document, but it would also embed ordering of the signatures. This could be useful, as some organisation's processes may require their documents to be signed in a specific order.

For the validation process just the final signature is needed (as it embeds all previous signatures) and the document itself, and then the whole signature chain can be validated recursively, with the innermost signature validating the document integrity.

15.2.2. Pairing-based Signatures

With pairing-based cryptography like Boneh-Lynn-Shacham (BLS) [37] we could implement n-of-n or m-of-n multi signatures. This wouldn't provide nor require any ordering in the signing process, and while much more elegant, would complicate the cryptographic aspect[1] and could introduce errors as we don't have a lot experience with pairing based cryptography.

[1]Saying we fully understand the mathematics behind it would be a lie.

Declaration of primary authorship

We hereby confirm that we have written this thesis independently and without using other sources and resources than those specified in the bibliography. All text passages which were not written by us are marked as quotations and provided with the exact indication of its origin.

Place, Date:	Bern, 16th January 2020	
Last Name/s, First Name/s:	Patrick Hirt	Gabor Tanz
Signature/s:	....	....

Bibliography

[1] “Angular: One framework. Mobile & desktop.” [Online]. Available: https://angular.io/

[2] “Apache Maven.” [Online]. Available: https://maven.apache.org/

[3] “Architectures and protocols for remote signature applications.” [Online]. Available: https://cloudsignatureconsortium.org/resources/download-api-specifications/

[4] “BFH Gitlab.” [Online]. Available: https://gitlab.ti.bfh.ch

[5] “Bouncycastle API: TimeStampRequestGenerator.” [Online]. Available: https://www.bouncycastle.org/docs/pkixdocs1.5on/org/bouncycastle/tsp/TimeStampRequestGenerator.html

[6] “CORS: Cross-origin Resource Sharing.” [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/HTTP/CORS/Errors/CORSRequestNotHttp

[7] “Embrace, extend and exterminate.” [Online]. Available: https://en.wikipedia.org/wiki/Embrace%2C_extend%2C_and_extinguish

[8] “EVS 821:2014: Format for Digital Signatures.” [Online]. Available: https://www.evs.ee/products/evs-821-2014

[9] “Flutter Mobile Application Development Framework.” [Online]. Available: https://flutter.dev/

[10] “Google CryptoJS: A JavaScript implementation of standard and secure cryptographic algorithms.” [Online]. Available: https://cryptojs.gitbook.io/docs/

[11] “Introducing CFSSL: CloudFlare’s PKI toolkit.” [Online]. Available: https://blog.cloudflare.com/introducing-cfssl/

[12] “Keycloack Identity and Access Management Solution.” [Online]. Available: https://www.keycloak.org/about.html

[13] “Ktor: asynchronous Web framework for Kotlin.” [Online]. Available: https://ktor.io/

[14] “Law of Demeter.” [Online]. Available: https://en.wikipedia.org/wiki/Law_of_Demeter

[15] “Let’s Encrypt.” [Online]. Available: https://letsencrypt.org/

[16] “Let’s Encrypt Frequently Asked Questions.” [Online]. Available: https://letsencrypt.org/docs/faq/#technical

[17] “Libyubihsm.” [Online]. Available: https://developers.yubico.com/YubiHSM2/Component_Reference/yubihsm-shell/

[18] “National Institute of Standards and Technology Cryptographic Algorithm Validation Program .” [Online]. Available: https://csrc.nist.gov/projects/cryptographic-algorithm-validation-program

[19] “NIST FIPS PUB 180-4: Secure Hash Standard.” [Online]. Available: https://web.archive.org/web/20161126003357/http://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.180-4.pdf

[20] “PKCS#11 with YubiHSM 2.” [Online]. Available: https://developers.yubico.com/YubiHSM2/Component_Reference/PKCS_11/

[21] “Post on Google Groups, WASM specification impeding Goroutines.” [Online]. Available: https://groups.google.com/d/msg/golang-nuts/YJefPwnzpzQ/Lm5NznW3DQAJ

[22] “Python 3 RFC 3161ng library.” [Online]. Available: https://github.com/trbs/rfc3161ng/blob/master/README.rst

[23] “RFC 2104: HMAC: Keyed-Hashing for Message Authentication, url = https://tools.ietf.org/html/rfc2104.”

[24] “RFC 3161: Internet X.509 Public Key Infrastructure Time-Stamp Protocol (TSP).” [Online]. Available: https://tools.ietf.org/html/rfc3161

[25] “RFC 5652: Cryptographic Message Syntax (CMS).” [Online]. Available: https://tools.ietf.org/html/rfc5652

[26] “RFC 5869: HMAC-based Extract-and-Expand Key Derivation Function (HKDF).” [Online]. Available: https://tools.ietf.org/html/rfc5869

[27] “RFC 7517: JSON Web Key (JWK).” [Online]. Available: https://tools.ietf.org/html/rfc7517

[28] “RFC 7519: JSON Web Tokens.” [Online]. Available: https://tools.ietf.org/html/rfc7519

[29] “Scaleway Elements Development Instances.” [Online]. Available: https://www.scaleway.com/en/virtual-instances/development/

[30] “The Go Programming Language.” [Online]. Available: https://golang.org/

[31] “The Kotlin Programming Language.” [Online]. Available: https://kotlinlang.org

[32] “TypeScript: A typed superset of JavaScript that compiles to plain JavaScript.” [Online]. Available: https://www.typescriptlang.org/

[33] “Web Crypto APIs: The SubtleCrypto Class.” [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/API/SubtleCrypto

[34] “Webpack Module Bundler.” [Online]. Available: https://webpack.js.org/

[35] “YubiHSM.” [Online]. Available: https://www.yubico.com/product/yubihsm-2/

[36] “YubiHSM Shell.” [Online]. Available: https://developers.yubico.com/YubiHSM2/Component_Reference/yubihsm-shell/

[37] D. Boneh and C. Gentry and H. Shacham and B. Lynn, “Aggregate and Verifiably Encrypted Signatures from Bilinear Maps,” Eurocrypt 2003, LNCS 2656, Tech. Rep., 2003.

[38] Eliza Paul, “What is Digital Signature - How it works, Benefits, Objectives, Concept,” Tech. Rep., 2019. [Online]. Available: https://www.emptrust.com/blog/benefits-of-using-digital-signatures

[39] European Parliament, “Regulation (EU) No 910/2014 of the European Parliament and of the Council of 23 July 2014 on electronic identification and trust services for electronic transactions in the internal market and repealing Directive 1999/93/EC,” EU, Tech. Rep., 2014.

[40] European Telecommunications Standards Institute, “ETSI TS 319 411: Policy and security requirements for Trust Service Providers issuing certificates,” ETSI, Tech. Rep., 2018.

[41] European Telecommunications Standards Institute ETSI, “ETSI TS 102 778-1 V1.1.1: Electronic Signatures and Infrastructures (ESI): PDF Advanced Electronic Signature Profiles,” ETSI, Tech. Rep., 2009.

[42] Gabor Tanz, Patrick Hirt, “Projekt 2: Aufbau einer DSS Infrastruktur,” Bern University of Applied Sciences, Tech. Rep., 2018.

[43] Hassenstein Gerhard, “Digital Signature - Advanced Topics .” [Online]. Available: https://drive.google.com/file/d/1tLZJ3OeDTtPL6-I6t6LhBd3LLlkRW4w4/view

[44] International Telecommunication Union Telecommunication Standardisation Sector ITU-T, “X.509: Public-key and attribute ertificate frameworks,” 2016. [Online]. Available: https://www.itu.int/rec/T-REC-X.509

[45] Jones M., Bradley J., Sakimura N., “RFC 7515: JSON Web Signature (JWS),” IETF, Tech. Rep., 2015.

[46] N. Sakimura and J. Bradley and M. Jones and B. de Medeiros and C. Mortimore , “OpenID Connect 1.0: An identity layer on top of the OAuth 2.0 [RFC6749] protocol .” [Online]. Available: https://openid.net/specs/openid-connect-core-1_0.html

[47] National Institute of Standards and Technology, “USA Patent No. 6829355: Secure Hash Algorithm 2.” [Online]. Available: https://worldwide.espacenet.com/textdoc?DB=EPODOC&IDX=US6829355

[48] ——, “NIST Special Publication 800-63A: Digital Identity Guidelines: Enrollment and Identity Proofing Requirements,” NIST, Tech. Rep., 2017.

[49] ——, “NIST Special Publication 800-63B: Digital Identity Guidelines: Authentication and Lifecycle Management,” NIST, Tech. Rep., 2017.

[50] Office of the Swiss Federal Chancellery , “Loi fÃl’dÃl’rale sur les services de certification dans le domaine de la signature Ãl’lectronique et des autres applications des certificats numÃl’riques .” [Online]. Available: https://www.admin.ch/opc/fr/classified-compilation/20131913/index.html

[51] Paul A. Grassi and Michael E. Garcia and James L. Fenton, “NIST Special Publication 800-63-3: Digital Identity Guidelines,” NIST, Tech. Rep., 2017.

[52] R. Rivest and A. Shamir and L. Adleman, “A Method for Obtaining Digital Signatures and Public-Key Cryptosystems,” Tech. Rep., 1978.

[53] S. Bradner , “Key words for use in RFCs to Indicate Requirement Levels .” [Online]. Available: https://tools.ietf.org/html/rfc2119

[54] Sakimura N., Bradley J., Jones M., de Medeiros B., Mortimore C. , “OpenID Connect Core 1.0 .” [Online]. Available: https://openid.net/specs/openid-connect-core-1_0.html

[55] Shai Halevi and Hugo Krawczyk, “Randomized Hashing and Digital Signatures.” [Online]. Available: http://www.ee.technion.ac.il/~hugo/rhash/

[56] Swiss Federal Council, “Ordonnance sur les services de certification dans le domaine de la signature Ãl’lectronique et des autres applications des certificats numÃl’riques .” [Online]. Available: https://www.admin.ch/opc/fr/classified-compilation/20162168/index.html

[57] Techtopia, “An Overview of Public Key Infrastructures (PKI),” 2015. [Online]. Available: https://www.techotopia.com/index.php/An_Overview_of_Public_Key_Infrastructures_(PKI)

[58] William Stallings, *Cryptography and Network Security: Principles and Practice*. Prentice Hall, 1999.

Acronyms

AAL Authenticator Assurance Level.

AAL2 Authenticator Assurance Level 2.

AAL3 Authenticator Assurance Level 3.

API Application Programming Interface.

ASN.1 Abstract Syntax Notation One.

AVX2 Advanced Vector Extensions 2.

BFH Bern University of Applied Sciences.

BLS Boneh-Lynn-Shacham.

CA Certificate Authority.

CI Continuous Integration.

CIO Coroutine-based I/O.

CLR .NET Common Language Runtime.

CMS Cryptographic Message Syntax.

CORS Cross-Origin Resource Sharing.

CRL Certificate Revocation List.

CSC Cloud Signature Consortium.

CSR Certificate Signing Request.

CSRF Cross Site Request Forgery.

CSRNG Cryptographically Secure Random Number Generator.

CSS Cascading Stylesheets.

DER Distinguished Encoding Rules.

DI Dependency Injection.

DNS Domain Name System.

DOM Document Object Model.

DoS Denial of Service.

DSA Digital Signature Algorithm.

ECC Elliptic Curve Cryptography.

ECDSA Elliptic Curve Digital Signature Algorithm.

Ed25519 EdDSA with Curve25519 and SHA-512.

EdDSA Edwards-curved Digital Signature Algorithm.

eIDAS electronic IDentification, Authentication and trust Services.

ETSI	European Telecommunications Standards Institute.
EU	European Union.
FSF	Free Software Foundation.
GUI	Graphical User Interface.
HKDF	HMAC-based Key Derivation Function.
HMAC	Keyed-Hash Message Authentication Code.
HSM	Hardware Security Module.
HTML	Hypertext Markup Language.
HTTP	Hypertext Transfer Protocol.
HTTPS	HTTP over TLS.
IAL	Identity Assurance Level.
IAL3	Identity Assurance Level 3.
IDE	Integrated Development Environment.
IDM	Identity and Access Management.
IDP	Identity Provider.
JAR	Java Archive.
JCA	Java Cryptography Architecture.
JOSE	JavaScript Object Signing and Encryption.
JRE	Java Runtime Environment.
JS	JavaScript.
JSON	JavaScript Object Notation.
JVM	Java Virtual Machine.
JWK	JSON Web Key.
JWKS	JSON Web Key Store.
JWS	JSON Web Signature.
JWT	JSON Web Token.
LoA	Level of assurance.
LTV	Long-Term Validation.
MAC	Message Authentication Code.
NIST	National Institute of Standards and Technology.
OCSP	Online Certificate Status Protocol.
OIDC	OpenID Connect.
OpenAPI	Open Application Programming Interface.

OTP One Time Password.

PDF Portable Document Format.

PEM Privacy-Enhanced Mail.

PIN Personal Identification Number.

PKCS Public Key Cryptography Standard.

PKCS10 Public Key Cryptography Standard 10.

PKCS11 Public Key Cryptography Standard 11.

PKCS7 Public Key Cryptography Standard 7.

PKI Public Key Infrastructure.

POC Proof Of Concept.

PSS Probabilistic Signature Scheme.

RA Registration Authority.

REST Representational State Transfer.

RFC Request For Comments.

RNG Random Number Generator.

RSA Rivest-Shamir-Adleman.

RSA-PSS RSA Probabilistic Signature Scheme.

SAD Signature Activation Data.

SAML Security Assertion Markup Language.

SAP Signature Activation Protocol.

SDK Software Development Kit.

SHA-2 Secure Hash Algorithm 2.

SHA-256 Secure Hash Algorithm 2 with 256-bit output length.

SHA-3 Secure Hash Algorithm 3.

SIM Subscriber Identity Module.

SLF4J Simple Logging Facade for Java.

SPA Single Page Application.

TLS Transport Layer Security.

TSA Time Stamping Authority.

TSP Time-stamp Protocol.

TSP Trust Service Provider.

TSS Timestamping Service.

UI User Interface.

UML Unified Modeling Language.

URL Uniform Resource Locator.

USB Universal Serial Bus.

VCS	Version Control System.
VM	Virtual Machine.
WASM	WebAssembly.
XML	Extensible Markup Language.

List of Figures

List of Tables

Listings

A. User Guide

In this short user guide, we show how to use our proof-of-concept implementation step by step.

A.1. Signing a Document

This section contains a step by step guide on how to obtain a signature for a document.

A.1.1. Ready the Document

The first step is to make sure the document is ready.

We **strongly** recommend using a read-only format, such as PDF, as the slightest change to the source document will invalidate its signature. Some editors even modify the document if it is simply opened for viewing, such as Microsoft Word.

If the document is ready for signing, navigate to the signing server using the preferred web browser. Whether a desktop, a laptop, a mobile device such as a smartphone or a tablet is used doesn't matter.

The frontend will present itself as shown in figure A.1.

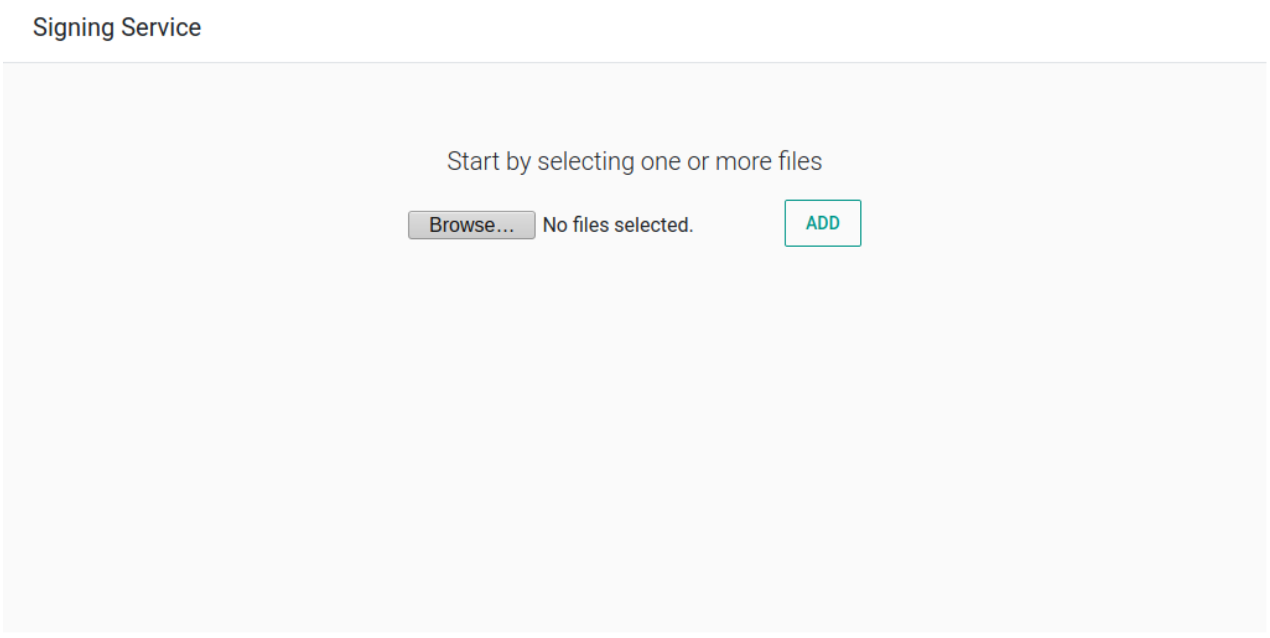


Figure A.1.: The frontend of the signing server upon initial access

A.1.2. Submit the Document for Hashing

The next step is to submit the document for hashing. Click the button labelled "Browse" and select the document to be signed.

The signing of multiple documents at once is supported as well. If multiple documents are to be signed together, simply select them all.

The frontend will look like as shown in figure A.2.

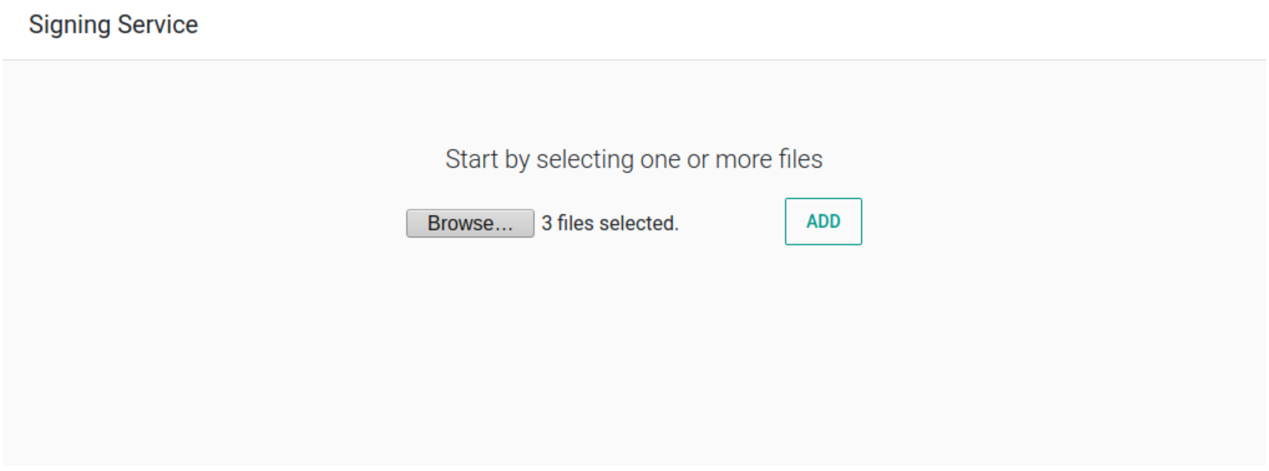
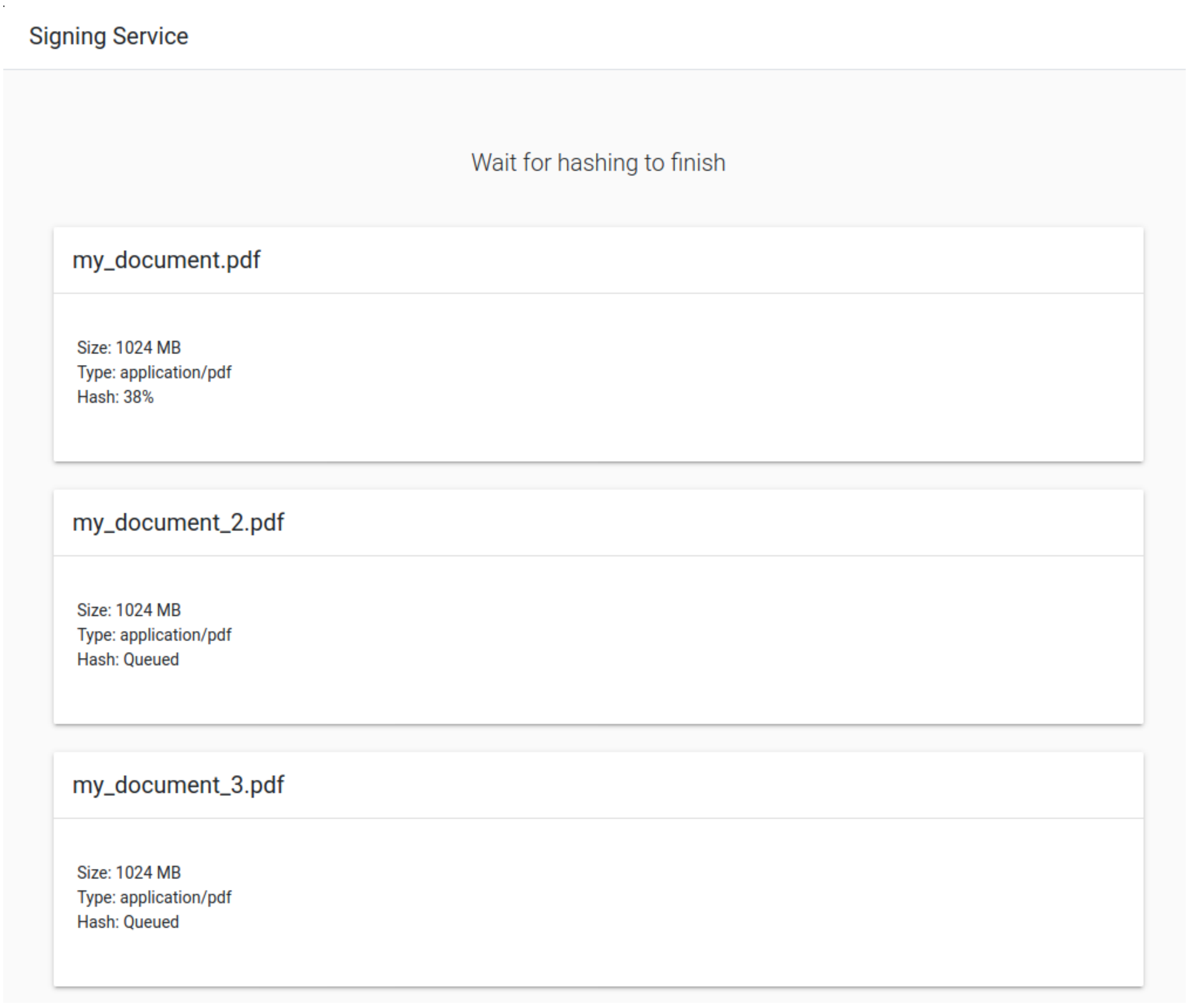


Figure A.2.: The frontend of the signing server after selecting the documents to sign

Then click the green button labelled "Add". The frontend will immediately begin hashing them. It will show the progress for each file as it is processing it, then display the hash value, for double-checking by the user, if so desired.

Please note that no data whatsoever leaves the device at this point. All operations are performed purely in-browser.

Figure A.3 illustrates what the frontend looks like during document hashing.


Signing Service

Wait for hashing to finish

my_document.pdf

Size: 1024 MB
Type: application/pdf
Hash: 38%

my_document_2.pdf

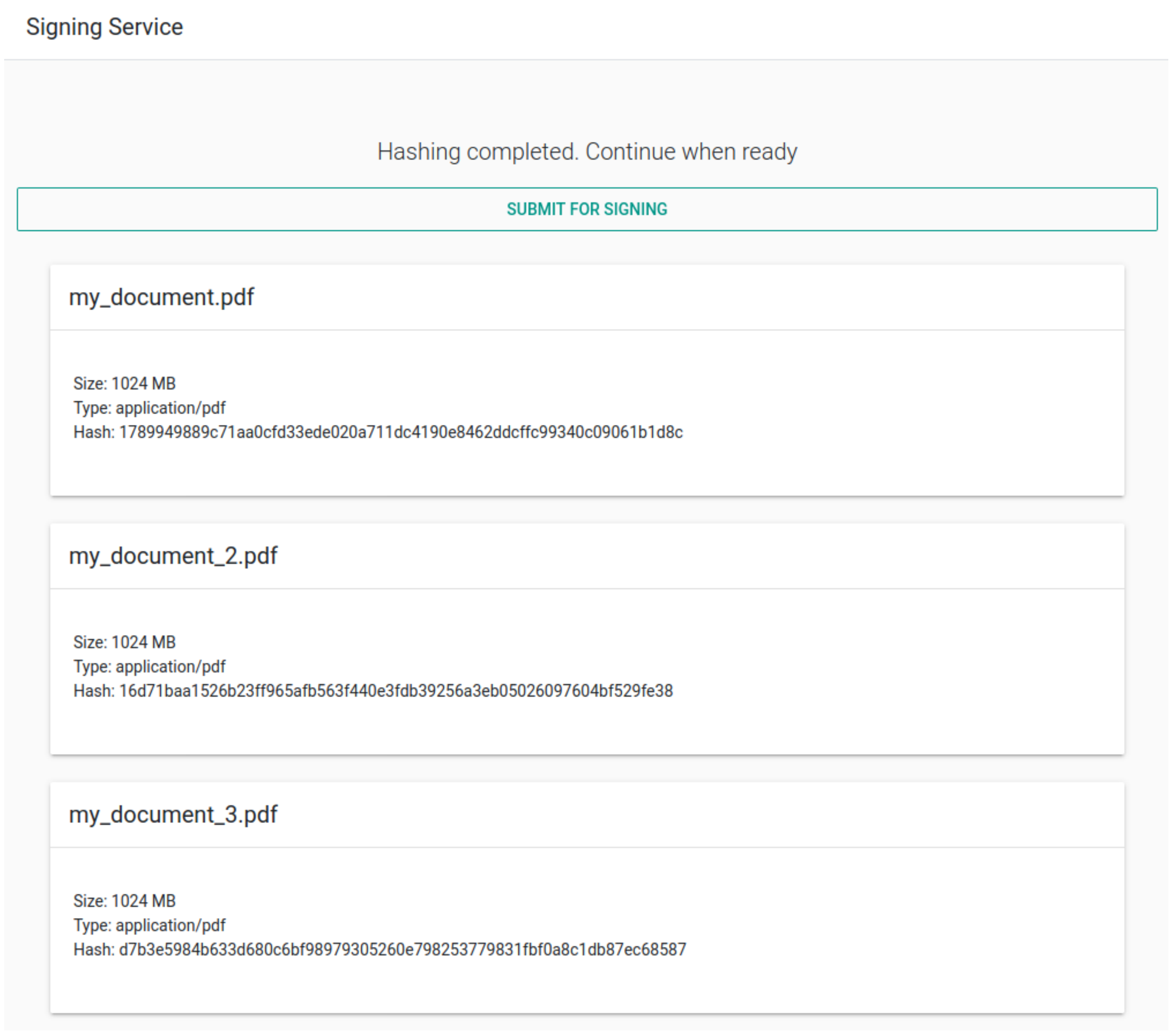
Size: 1024 MB
Type: application/pdf
Hash: Queued

my_document_3.pdf

Size: 1024 MB
Type: application/pdf
Hash: Queued

Figure A.3.: The frontend of the signing server when hashing

When hashing is completed, the frontend will offer to submit the hashes to the signing server. Figure A.4 illustrates what this looks like.


Signing Service

Hashing completed. Continue when ready

my_document.pdf

Size: 1024 MB
Type: application/pdf
Hash: 1789949889c71aa0cfd33ede020a711dc4190e8462ddcffc99340c09061b1d8c

my_document_2.pdf

Size: 1024 MB
Type: application/pdf
Hash: 16d71baa1526b23ff965afb563f440e3fdb39256a3eb05026097604bf529fe38

my_document_3.pdf

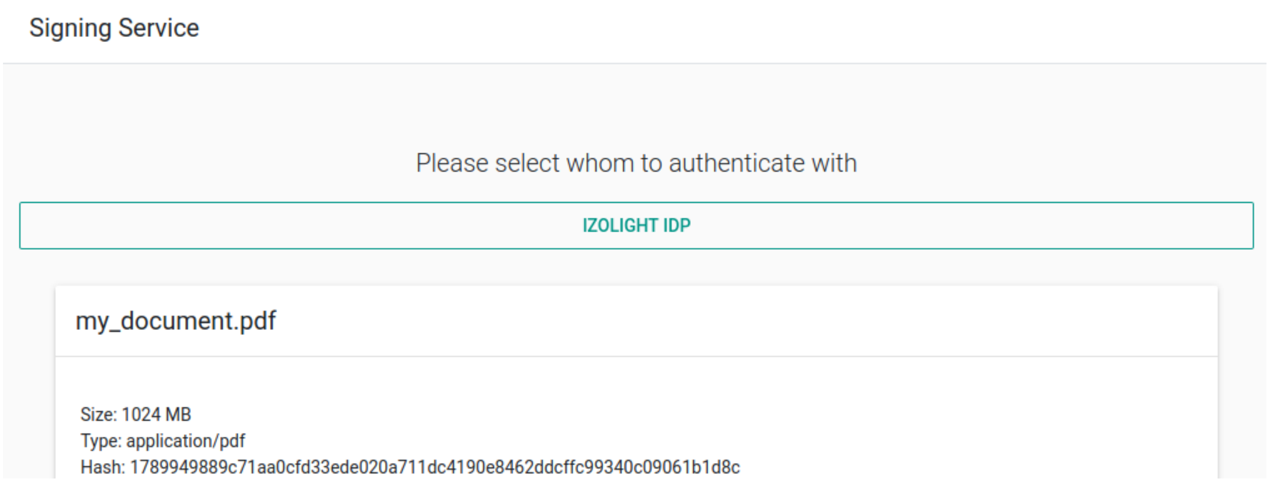
Size: 1024 MB
Type: application/pdf
Hash: d7b3e5984b633d680c6bf98979305260e798253779831fbf0a8c1db87ec68587

Figure A.4.: The frontend of the signing server upon completion of hashing

When ready, press the green button labelled "Submit for Signing". The hashes will be submitted to the signing server.

A.1.3. Selecting an IDP and Authenticating

The signing server will respond with a list of the IDP it trusts, which are displayed as buttons, as shown in figure A.5.


Signing Service

Please select whom to authenticate with

my_document.pdf

Size: 1024 MB
Type: application/pdf
Hash: 1789949889c71aa0cfd33ede020a711dc4190e8462ddcffc99340c09061b1d8c

Figure A.5.: The frontend of the signing server offering the choices of IDPs

If there's only one IDP, only one button is displayed. Click on the button representing the preferred IDP. This will start the authentication process.

What the next steps look like depends on the IDP. In our demo IDP, the login screen looks like shown in figure A.6.

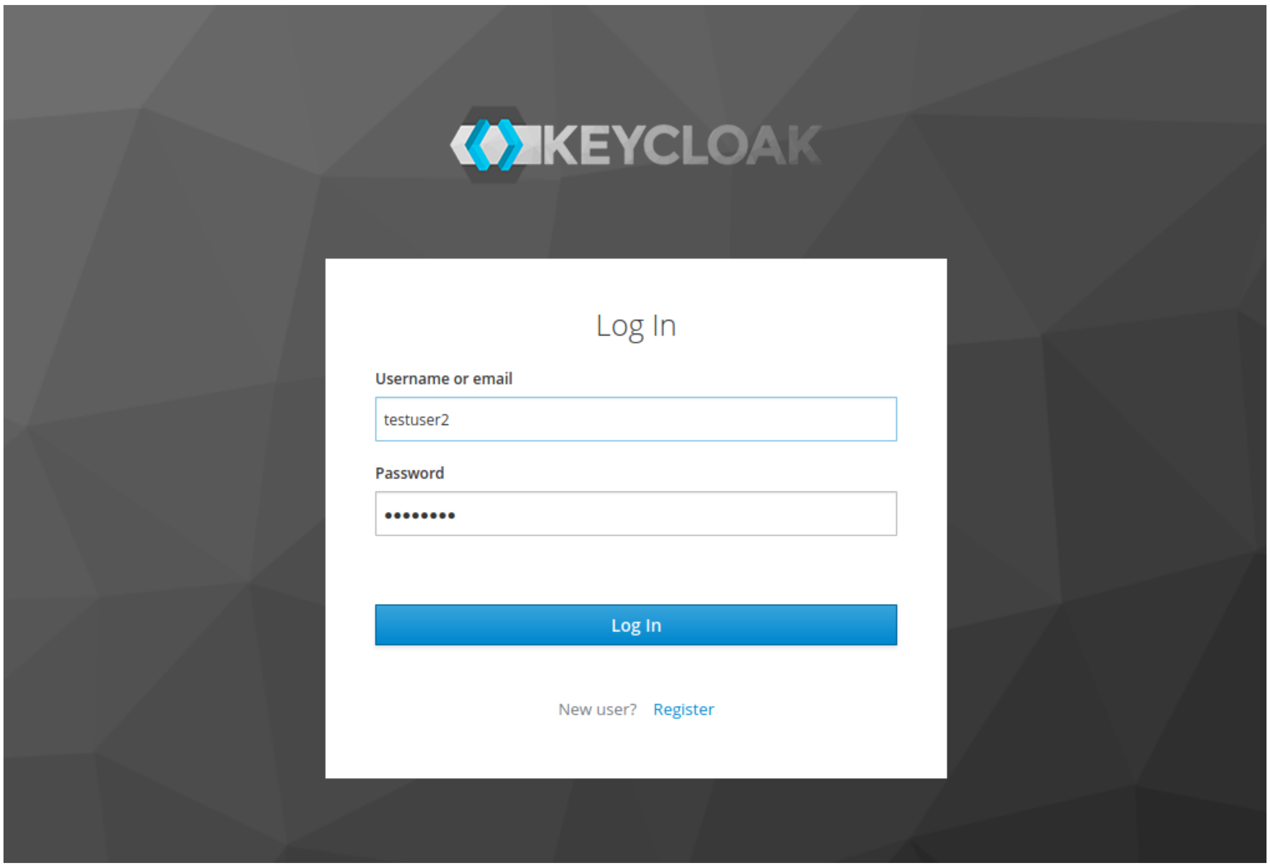


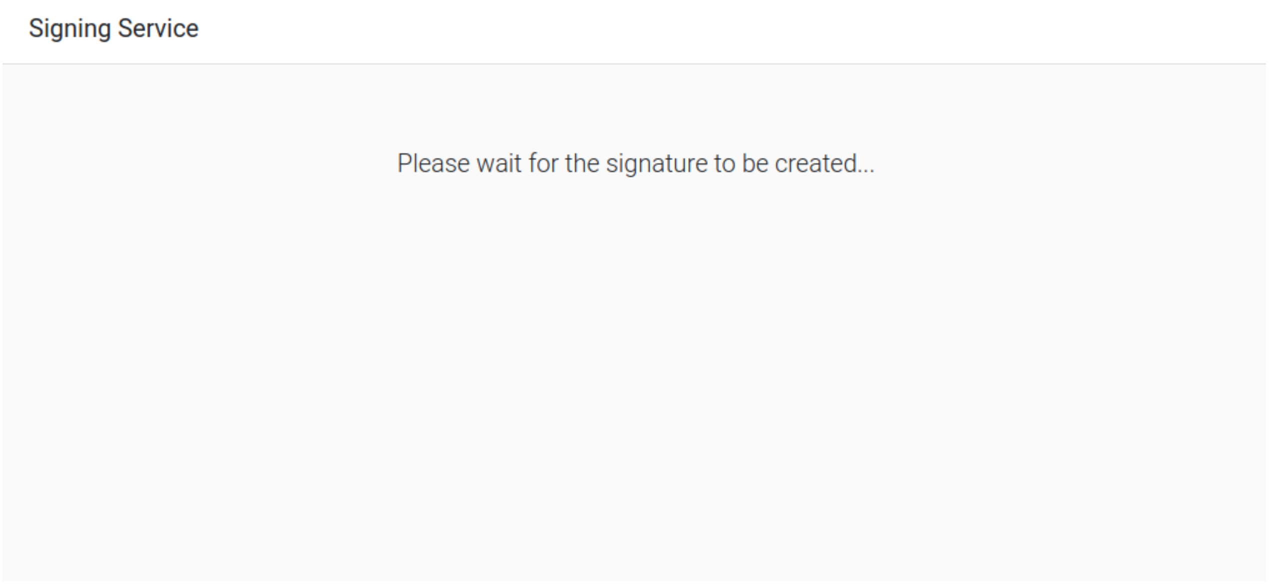
Figure A.6.: The IDP asking for credentials

Provide the credentials and multi-factor tokens as required by the IDP. Upon successful authentication, the signing server will begin creating the signature.

A.1.4. Retrieving the Created Signature

This may take a few seconds, depending on server load.

During signature creation, the frontend looks like shown in figure A.7.

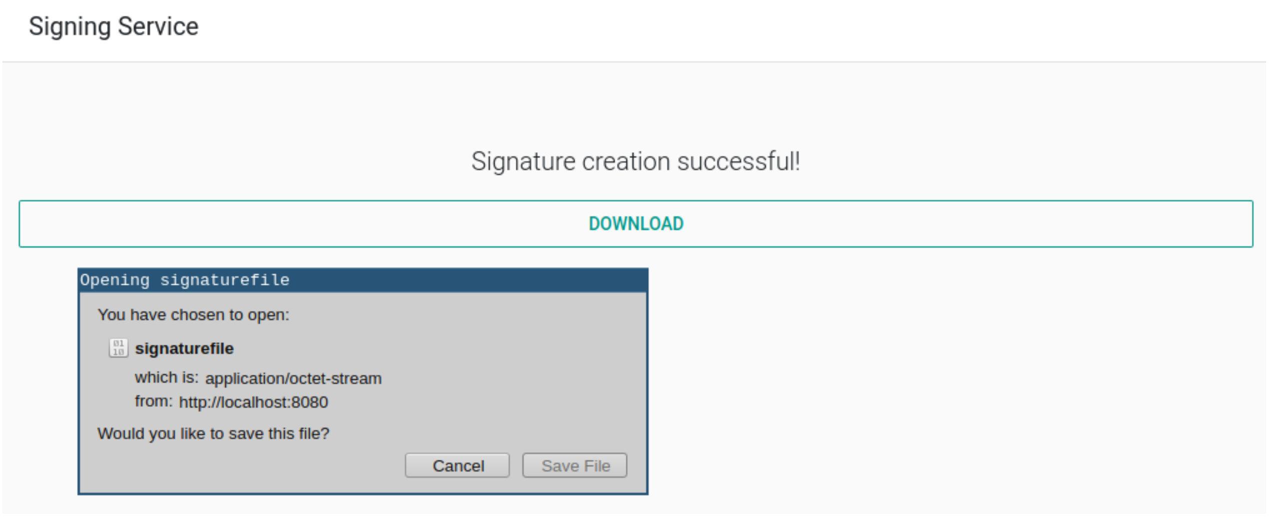


Signing Service

Please wait for the signature to be created...

Figure A.7.: Signature creation in progress

When signature creation is completed, a file download button will appear. Click the button, and a file download will be launched, as shown in figure A.7.



Signing Service

Signature creation successful!

Opening signaturefile

You have chosen to open:

signaturefile

which is: application/octet-stream

from: http://localhost:8080

Would you like to save this file?

Figure A.8.: Signature file download

Make sure to save the signature file, for it is not persisted on the server indefinitely.

Congratulations! The signature has been created successfully.

A.2. Verifying a Signature

This section contains a step by step guide on how to verify a signature that was previously created.

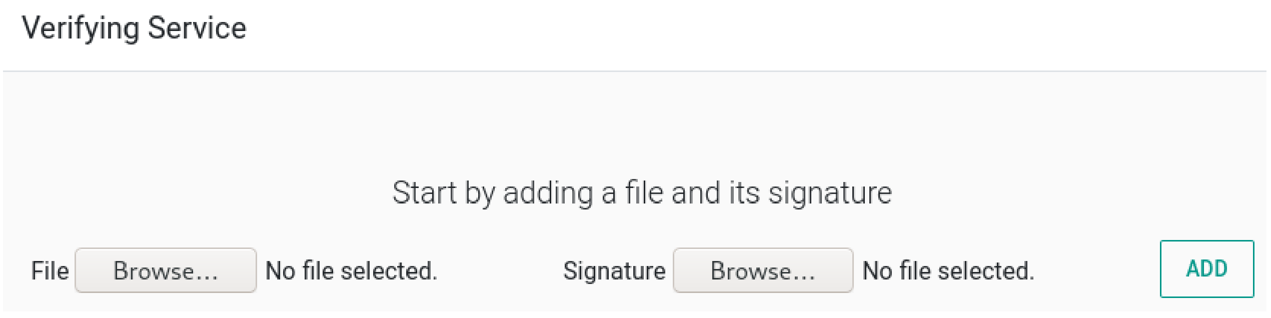
A.2.1. Ready the Document and Its Signature File

The first step is to make the document and its signature file is at hand.

If the document is ready for verification, navigate to the verification service using the preferred web browser. The verification service may be running either in an internet-accessible location, or on the local machine. For devices such as tablets and smartphones running Android or iOS, only online verification is supported. For devices such as laptops and desktops running GNU/Linux, Mac OS X or Windows, offline verification is supported by launching the appropriate build of the verifier program on the local machine.

Whether the verification service is accessed locally or not doesn't matter for the steps that follow, the procedure remains the same.

Upon accessing the verification service, its frontend will present itself as shown in figure A.9.


Verifying Service

Start by adding a file and its signature

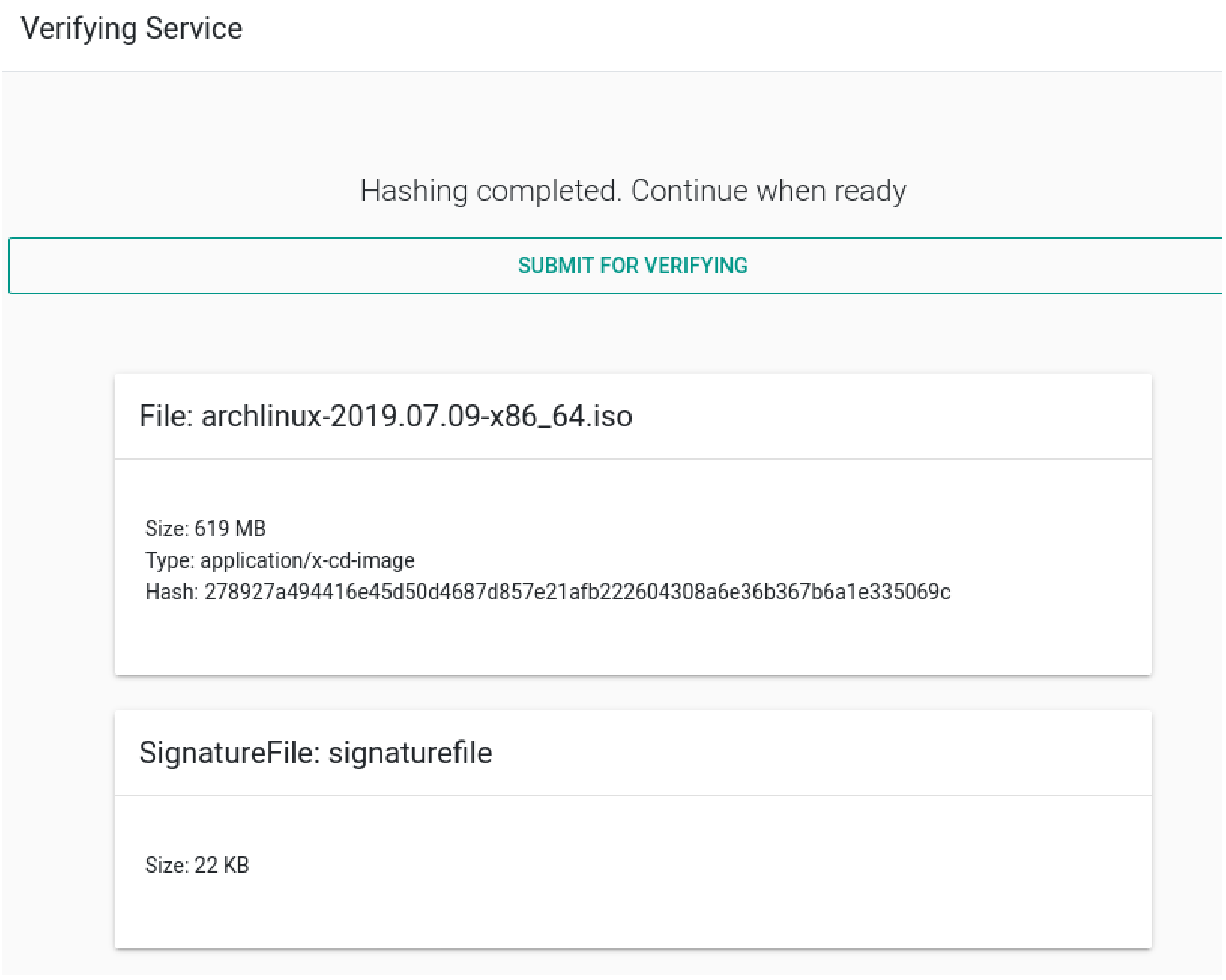
File Browse... No file selected. Signature Browse... No file selected. ADD

Figure A.9.: The frontend of the verification service upon initial access

Proceed by selecting first the document, and then its signature to begin the verification process.

After selecting the files, press the green "Add" button. The frontend will start hashing the document. When hashing is completed, the document's hash and the signature file are ready for transmission to the verification service, as shown in figure A.10.

Please note that no matter how the verification service is used, remote or local, mobile device or not, the document never leaves the device.


Verifying Service

Hashing completed. Continue when ready

File: archlinux-2019.07.09-x86_64.iso

Size: 619 MB
Type: application/x-cd-image
Hash: 278927a494416e45d50d4687d857e21afb222604308a6e36b367b6a1e335069c

SignatureFile: signaturefile

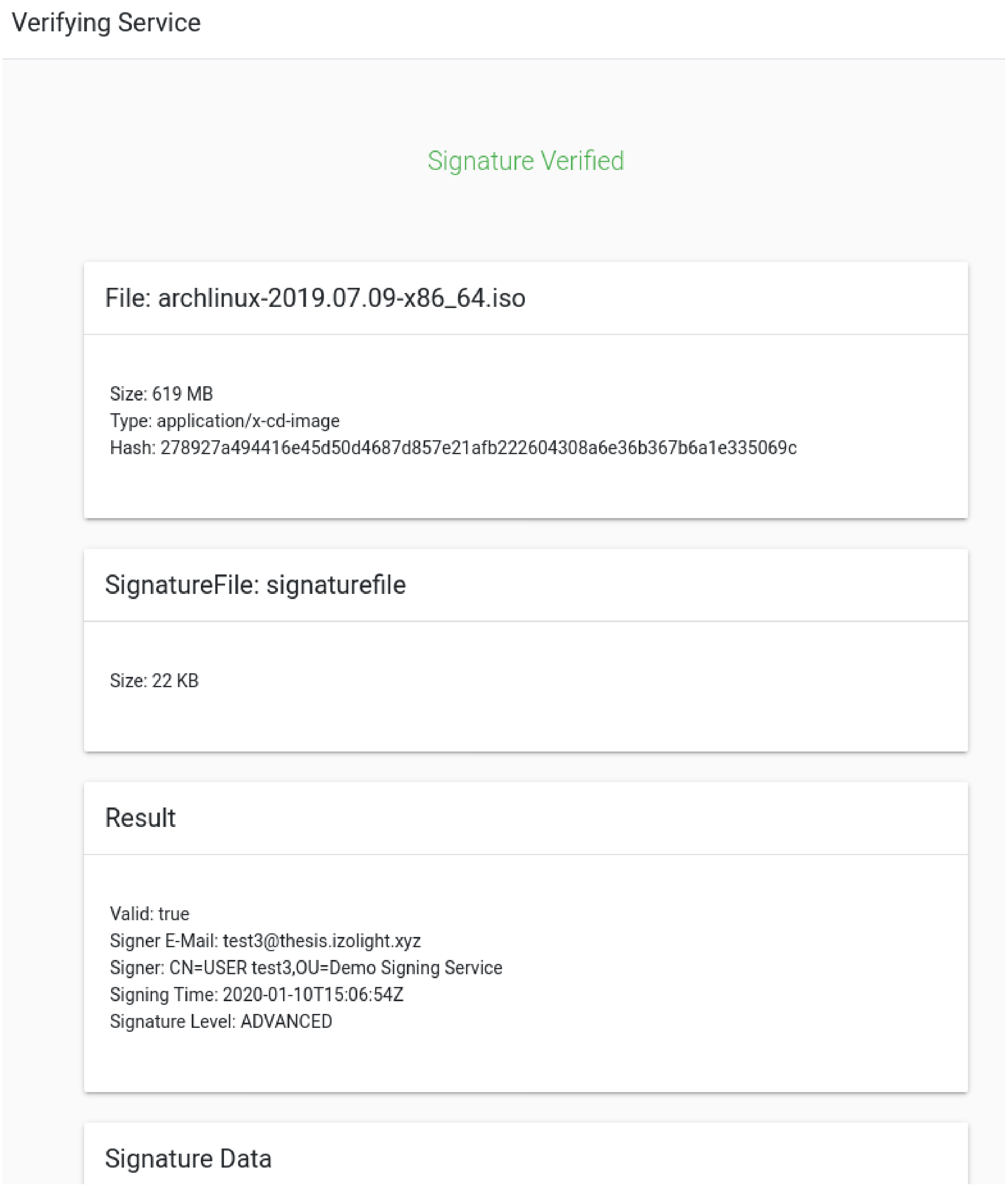
Size: 22 KB

Figure A.10.: Verification service ready for verification

A.2.2. Submit Document and Signature File for Verification

When ready for submission, press the green button labelled ”submit for verifying”.

The verification service will then attempt to verify the signature. If signature verification was successful, the verification service will display a result as shown in figure A.11.


Verifying Service

Signature Verified

File: archlinux-2019.07.09-x86_64.iso

Size: 619 MB
Type: application/x-cd-image
Hash: 278927a494416e45d50d4687d857e21afb222604308a6e36b367b6a1e335069c

SignatureFile: signaturefile

Size: 22 KB

Result

Valid: true
Signer E-Mail: test3@thesis.izolight.xyz
Signer: CN=USER test3,OU=Demo Signing Service
Signing Time: 2020-01-10T15:06:54Z
Signature Level: ADVANCED

Signature Data

Figure A.11.: Verification service has completed verification

For verification of another document as part of the same signature, the procedure is exactly the same, simply select the other document together with the same signature file as above.

Congratulations! The signature has been verified successfully.