Remote Signing Service

Accessible Electronic Signatures for Everybody

Bachelor Thesis by Gabor Tanz and Patrick Hirt

TODO abstract TODO

Degree course:	Computer Science
Authors:	Gabor Tanz, Patrick Hirt
Tutor:	Dr. Annett Laube, Gerhard Hassenstein
Experts:	Dr. Andreas Spichiger
Date:	20th December 2019

Berner Fachhochschule | Haute école spécialisée bernoise | Bern University of Applied Sciences

Versions

Version	Date	Status	Remarks
0.1	01.08.2013	Draft	Lorem ipsum dolor sit amet
0.2	21.08.2013	Draft	Phasellus scelerisque
0.3	02.09.2013	Draft	Donec eget aliquam urna. Lorem ipsum dolor sit amet
1.0	26.01.2014	Final	Lorem ipsum dolor sit ametPhasellus scelerisque, leo sed iaculis ornare
1.1	31.01.2014	Correction	Layout changed
1.2	07.02.2014	Addition	Chapter 1.1 extended

Contents

1. Introduction

The trend for Signing Service Solutions goes in the direction of Remote Signing. Users no longer hold on to the cryptographic keys themselves, instead a remote service stores the users' signing keys or even generates the keys on-demand. This has a serious drawback: such a signing service is able to sign without user's knowledge or consent, thus it must offer complete trustworthiness. In this thesis, we provide a specification as well as a proof-of-concept where the signing process is tied to the identity of the user, such that the signing service by itself cannot sign a document.

The implementation consists of the remote signing service itself exposing a Representational State Transfer (REST) Application Programming Interface (API), a cross-platform verification program offering both online and offline verification, and a cross-platform frontend authenticating the users through a trusted OpenID Connect (OIDC) Identity Provider (IDP).

1.1. Previous Works

We build upon our previous work of Project 2 [33], where we specified the authentication process for qualified signatures, non-qualified batch signatures, a 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.

In this thesis, we expand upon our previous work substantially both functionally and conceptually. This thesis is not just an implementation of an existing concept.

1.2. Overview of Contents

TODO

2. Evolution of the Signing Protocol

2.1. Overview

The purpose of the signing protocol is to specify which steps have to be carried out by which actor in what order so as to create a signature.

2.2. Signature Format

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 [31], 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 [5], which, in essence, is a container format for the actual document along with the signature information. With their solution, arbitrary document formats can be used, but the downside is that the user needs to install a program that is able to extract and display the document contained in DigiDoc, even if they just wished to view the document without verifying the signature.

For this reason, 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).

2.2.1. Original Format

Our original specification for the signature format is based on our work in Projekt 2 [33] 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 (SHA)256, SHA3)
- Timestamp according to Request For Comments (RFC) 3161[1]
- Public key (Privacy-Enhanced Mail (PEM))
- Issuing Certificate Authority (CA) (PEM)
- Subject
- Validity

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

- 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)).

2.2.2. 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 [13].

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 simply consist 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 [13, 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 [28, 4.19].

Qualified Electronic Signature

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

1. It is created using a secured signature creation device [13, Art. 6]
2. It is based upon a qualified certificate [13, 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.

2.3. Flaws of the Original Protocol

The original protocol, which we specified in Projekt 2 [33] 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 could be abused to Denial of Service (DoS) the signing server, and it made this part of the signing server stateful.

Furthermore, 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.

Moreover, 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.

2.3.1. 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 hash(es) 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 protects against the IDP gaining knowledge of the signed hashes and to protect against the IDP learning about the document hashes.

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 Cross Site Request Forgery (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.

2.3.2. Draft 2: Improving signing of multiple documents

Even with the improvements in draft 1 (section 2.3.1), 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 Keyed-Hash Message

Authentication Code (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.

2.3.3. Draft 3: Simplifying and using CMS where possible

Draft 2 (section 2.3.2) resulted in a rather complicated schema that looked something like listing 2.1.

```
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 2.1: 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 2.3.2 we need to include the other maskes hashes: that's what `other_macs` is for.

`Timestamped` allows us to add certificate revocation information (Certificate Revocation List (CRL) and Online Certificate Status Protocol (OCSP)) for the certificates used in the RFC 3161 [18] 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.

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

When we examined RFC 5652 [19] more closely, we discovered that it's possible to add CRLs as well as OCSP responses to Cryptographic Message Syntax (CMS) messages (but not to Public Key Cryptography Standard 7 (PKCS7) [19, Section 10.2.1, RevocationInfoChoices and OtherRevocationInfoFormat]). Since both `SignatureData` and the RFC 3161 [18] timestamps are RFC 5652 CMS messages which do support including revocation information, we can simply put the revocation information in there 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 2.2: Simplified schema

Unfortunately we can't get rid of LTV completely because there is no way to add revocation information to a RFC 7517 [21] 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 2.2.

3. Technical Specification

3.1. 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) [32].

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 2.2.2.

3.1.1. 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.

3.1.2. Enrollment and Identity Proofing Requirements

The IDP is required to identify and register users at Identity Assurance Level 3 (IAL3) as defined by National Institute of Standards and Technology (NIST) [35]. 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 [35].

3.1.3. Authentication and Lifecycle Management Requirements

The IDP is required to authenticate users at Authenticator Assurance Level 3 (AAL3) as defined by NIST [36]. 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 [36].

3.1.4. Requirements for Advanced Electronic Signatures

The requirements specified in 3.1.2 and 3.1.3 apply to qualified electronic signatures. For advanced electronic signatures, Authenticator Assurance Level 2 (AAL2) is sufficient. The Authenticator Assurance Level (AAL) required by the signing service to emit the desired signature level can be specified to the IDP [29, Section 2], and is subsequently confirmed by the IDP in the field `acr` of the JSON Web Token (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.

3.1.5. 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 [34, RFC 7515, Section 4.1.6]. This JWKS along with optional revocation information for Long-Term Validation (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 [34, 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 [22, Section 1], and a JWS is used with a JSON Web Key (JWK) [21] 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 [22, 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`.

3.2. 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 3.2.1)
- Login (see 3.2.2)
- Post-Login (see 3.2.3)
- Signature Generation (see 3.2.4)
- Signature Verification (see 3.2.5)

For a high-level overview of these phases, see figure 3.1.

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

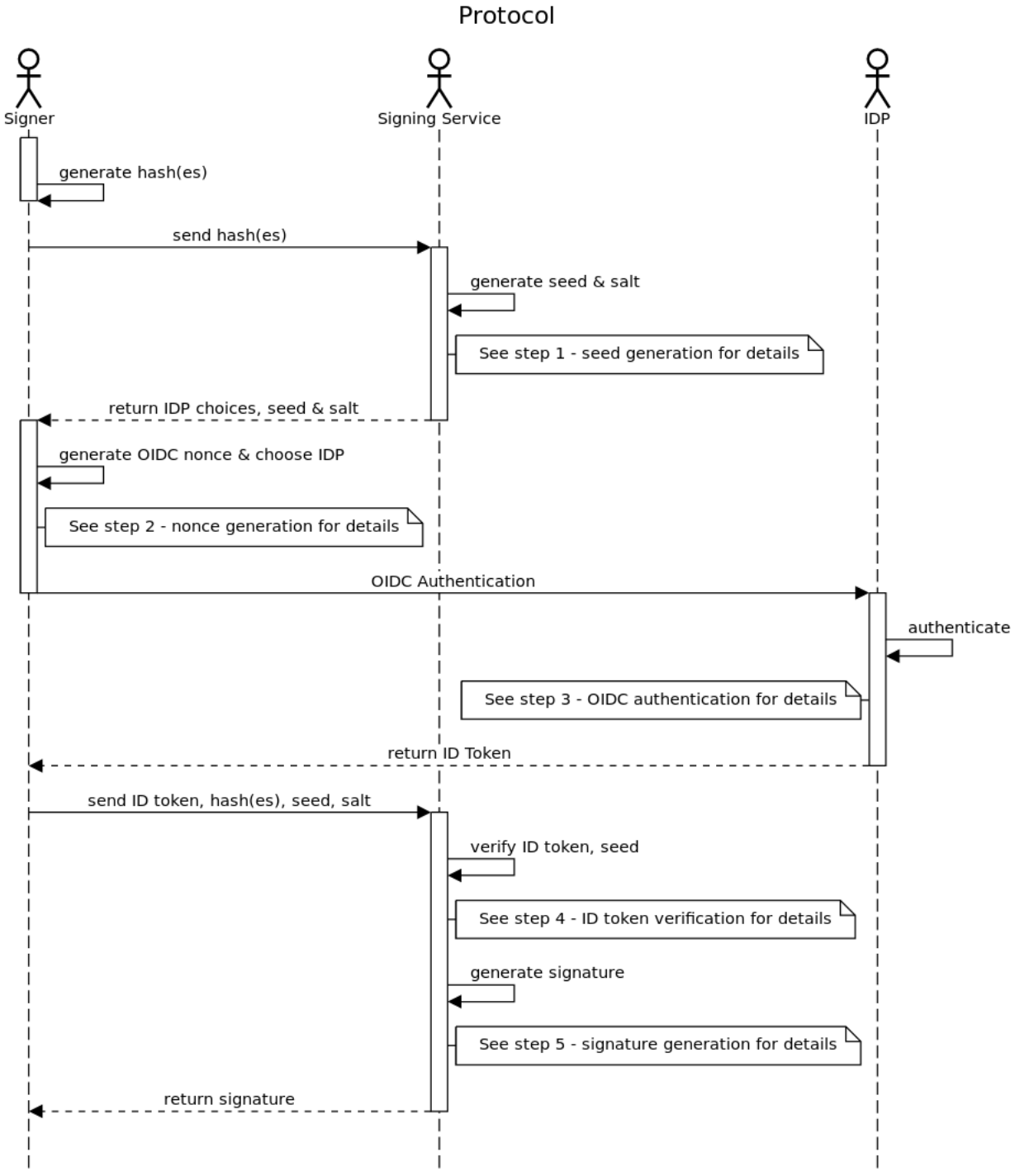

Figure 3.1.: High-level protocol overview

3.2.1. 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 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 [20].

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

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 [17].

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

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 [14].

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

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

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 server 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[3].

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 document(s), they can calculate their respective hashes, and then derive the salted hash(es) 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 document(s) 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.
[3] as specified in the non-functional requirements, "Protection of Information"

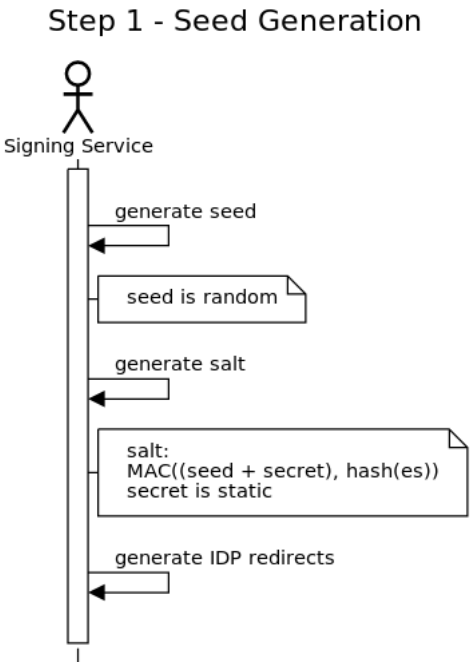

Figure 3.2.: Seed generation step

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 [20, Section 3.1]. Furthermore, the seed is used as a 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 3.2.1) and proceeding directly to the Post-Login step (3.2.3) 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 3.2.

3.2.2. Login

TODO update graphic: no client-side generating anything because no one likes to write javascript For a graphical representation of this, see figure 3.3.

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 3.4.

3.2.3. Post-Login

As shown in figure 3.5, 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 3.2.3. After this step, the seed is not used anymore and should be discarded.

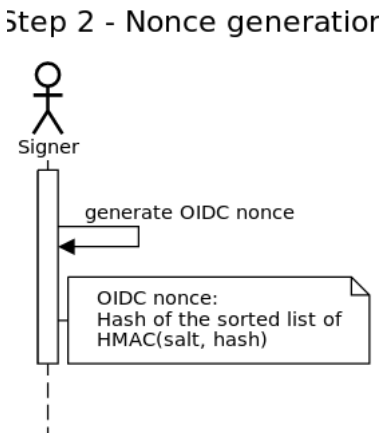

Figure 3.3.: Nonce generation step

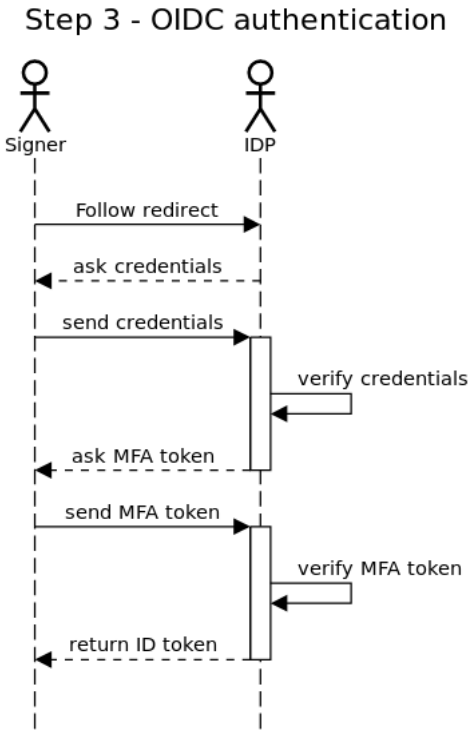

Figure 3.4.: OIDC authentication step

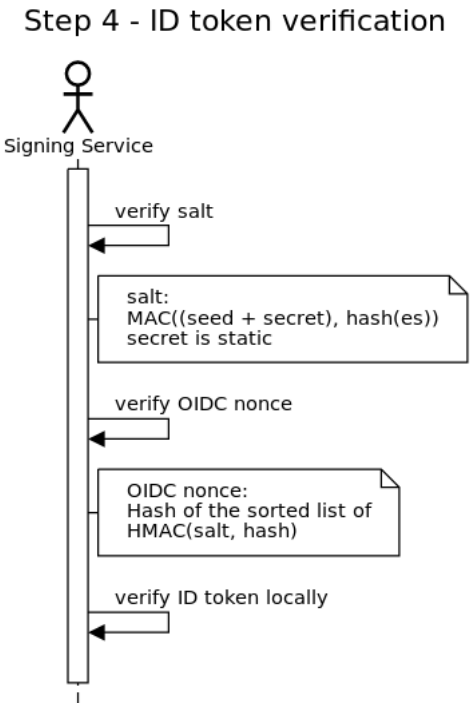

Figure 3.5.: Token verification step

Verification of hashes, seed and salt

Upon receiving a well-formed request in the format described in 3.3.1, the signing server performs no fewer than the following checks:

1. Verifying the id_token as described in [22, 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 3.2.1

These verifications are paramount to the safety of the system.

3.2.4. Signature Generation

The signing server requests a new signing key from the Hardware Security Module (HSM), which in turn generates a private key and returns a Certificate Signing Request (CSR). This CSR is sent to the CA where it is signed and the signed certificate returned.

Each hash is then submitted to the HSM to be signed by the signing key.

Then, for each hash an intermediate signature file consisting of the signed hash, the ID token, the `salt` and a sorted list of all other salted hashes is created. The hash of this file is sent to a Time Stamping Authority (TSA) where a signed timestamp is created and returned.

Finally, the signed timestamp, and all the certificate chains, OCSP responses and CRLs of the involved parties, is added to signature file, which is then returned to the user.

TODO update this figure since we no longer loop See figure 3.6 for a sequence diagram of this process.

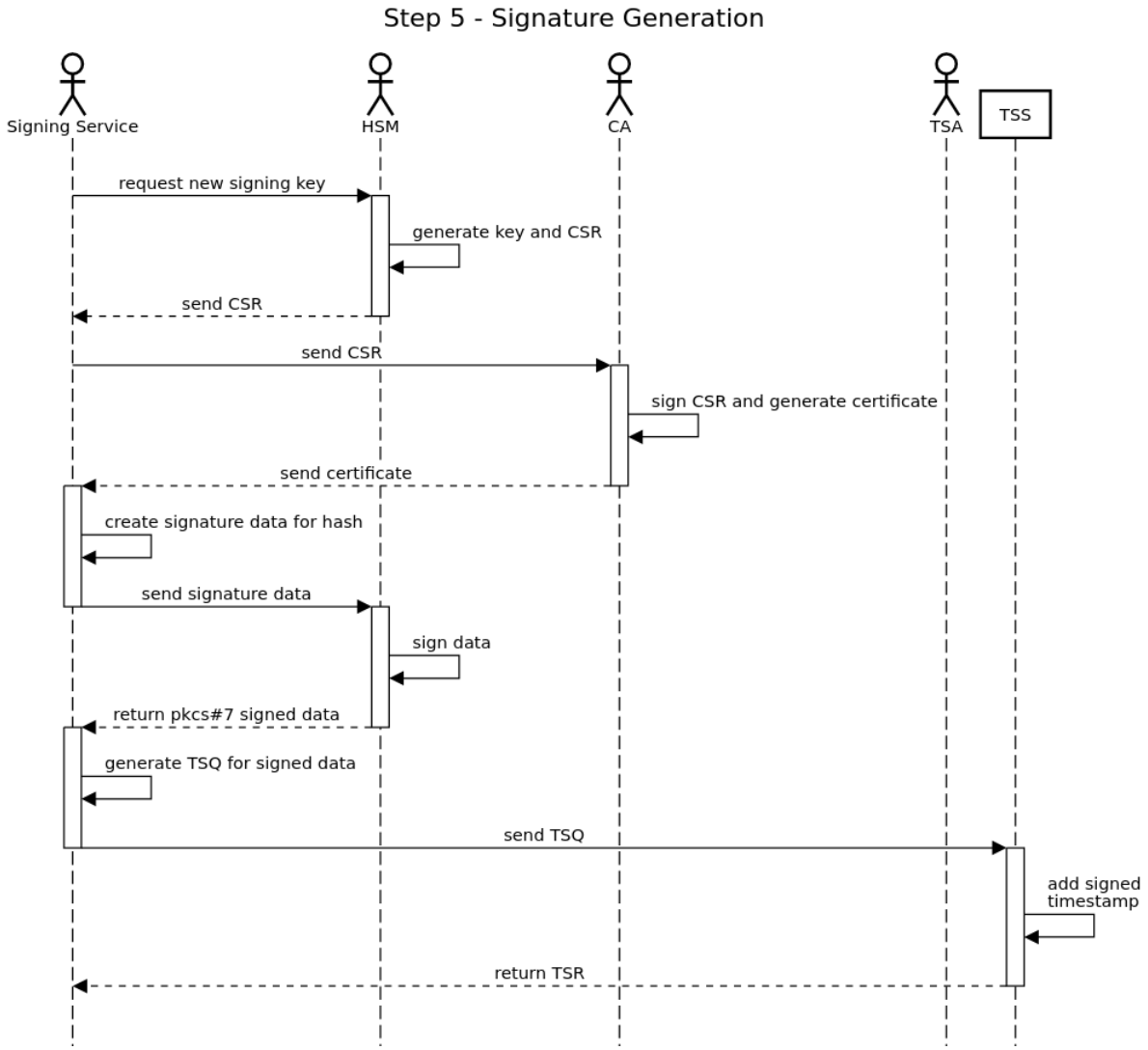

Figure 3.6.: Signature generation step

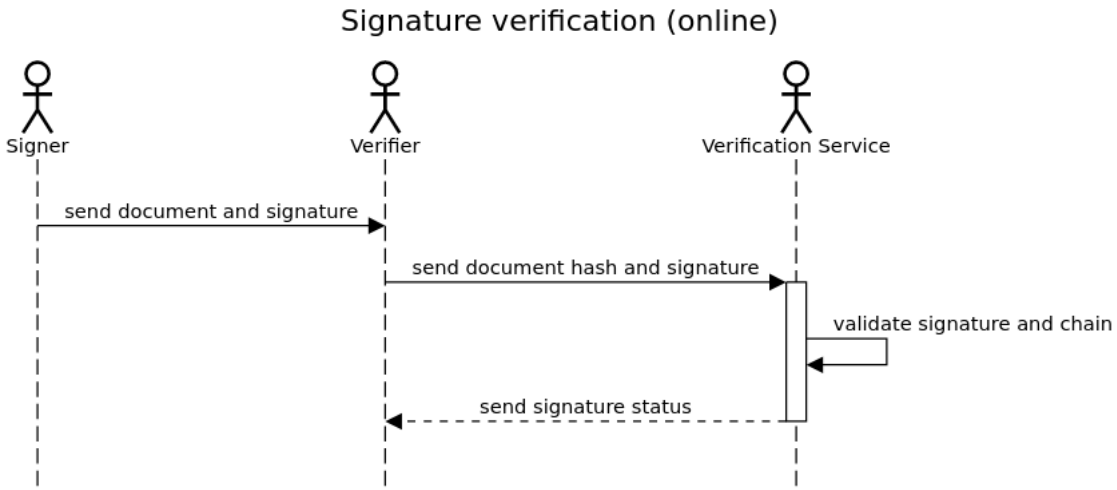

Figure 3.7.: Online Signature Verification Protocol

3.2.5. Signature Verification

The signature verification can either be online (figure 3.7) or offline (figure 3.8).

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

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.

To verify a signature, the verifier first needs to verify the chain of signed timestamps and their respective CA chains (figure 3.9).

The signature itself can then be verified (figure 3.6), along with the certificate chain for the signing certificate.

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 3.11).

In order to verify the link of the document hashes with the ID token, the document hash needs to be salted, inserted into the sorted list of the other salted hashes, and the resulting list hashed again. The result must be the same as the nonce in the ID token. At last, the hash of the document must match the included hash. Figure 3.12 illustrates this process.

3.3. 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.

3.3.1. 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 3.1.: Endpoints offered by the Signing Service

In the most common case, the client application will call these endpoints in the order listed:

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

message	MANDATORY	String	Reason for rejection of request

Table 3.2.: Output in case of invalid input

Sample Error Response:

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

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

Listing 3.1: 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 3.3.: Input to the login endpoint

Duplicates in the list of hashes are not allowed and are rejected by the API as described in 3.3.1. 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 3.4.: 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 3.2: login request

Sample Response:

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

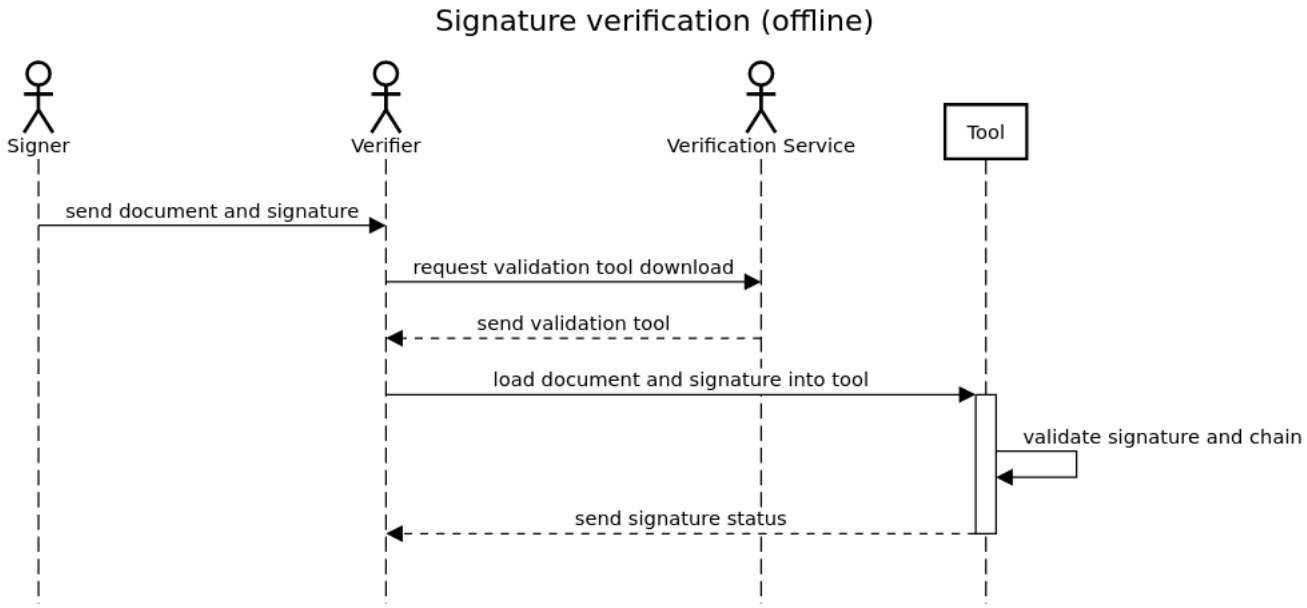

Figure 3.8.: Offline Signature Verification Protocol

```
{
"providers": {
"SwissID":
   "https://...&nonce=6cd7ef99e5e79d68d681e5d097b7f805381c4d013152fa3f26d06bd728ae
"Google":
   "https://...&nonce=6cd7ef99e5e79d68d681e5d097b7f805381c4d013152fa3f26d06bd728ae
},
"seed": "84c97acc49335faa0266fb29b4228205e9400a85a10faa68ec30cf894e1730ed",
"salt": "cfb663431af5e2d68be48867f93e86e477cd7eeefc10b16a51c238d2c810561b"
}
```

Listing 3.3: 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 3.5.: Input to the signing endpoint

The signing server performs the checks described in 3.2.3, and if any of them fail to pass, the server rejects the request as described in 3.3.1.

Output:

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

Table 3.6.: Output of the signing endpoint

Sample-Request:

```
POST /api/v1/sign HTTP/1.1
Host: service.example.org
```

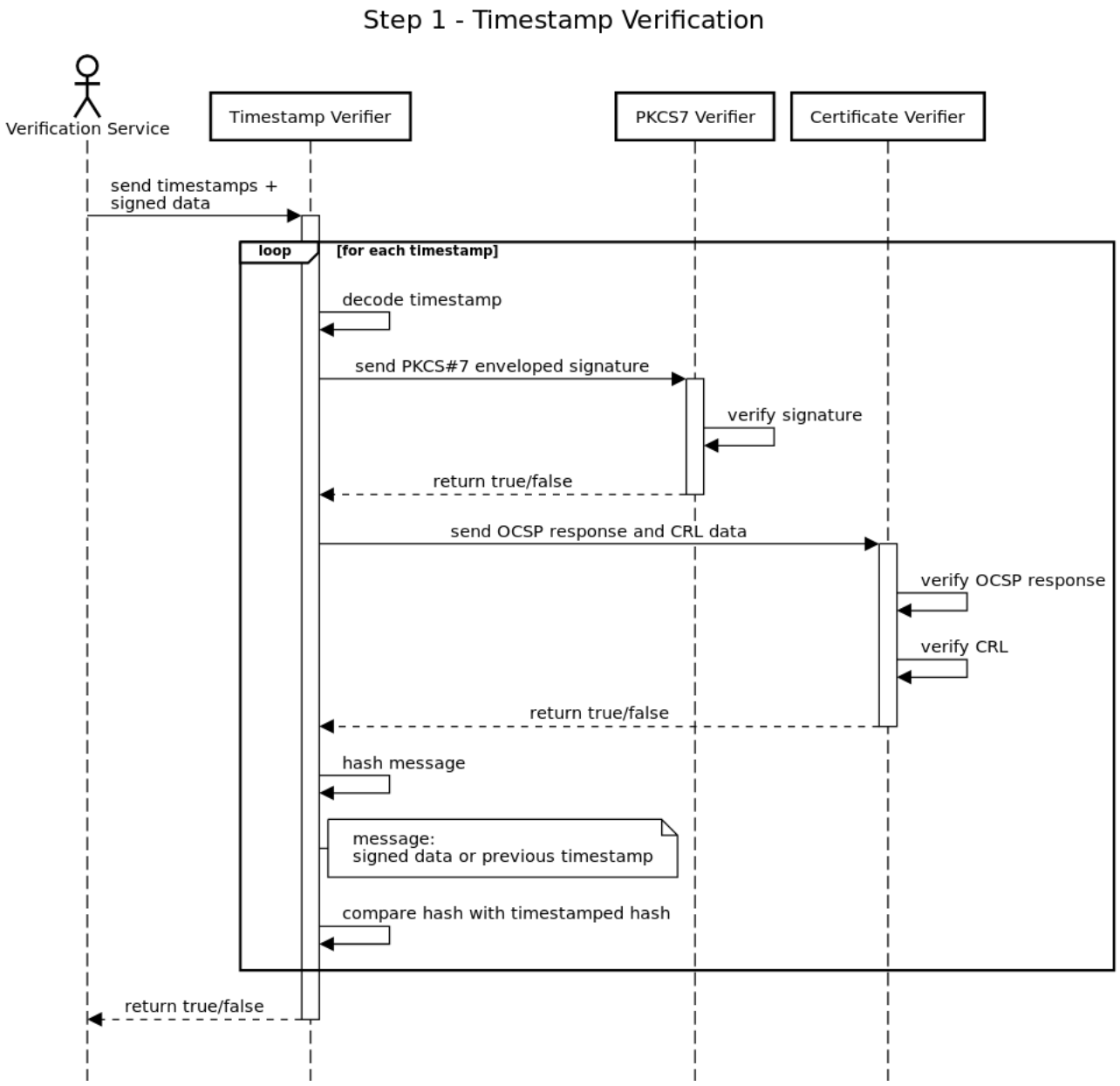

Figure 3.9.: Timestamp verification step

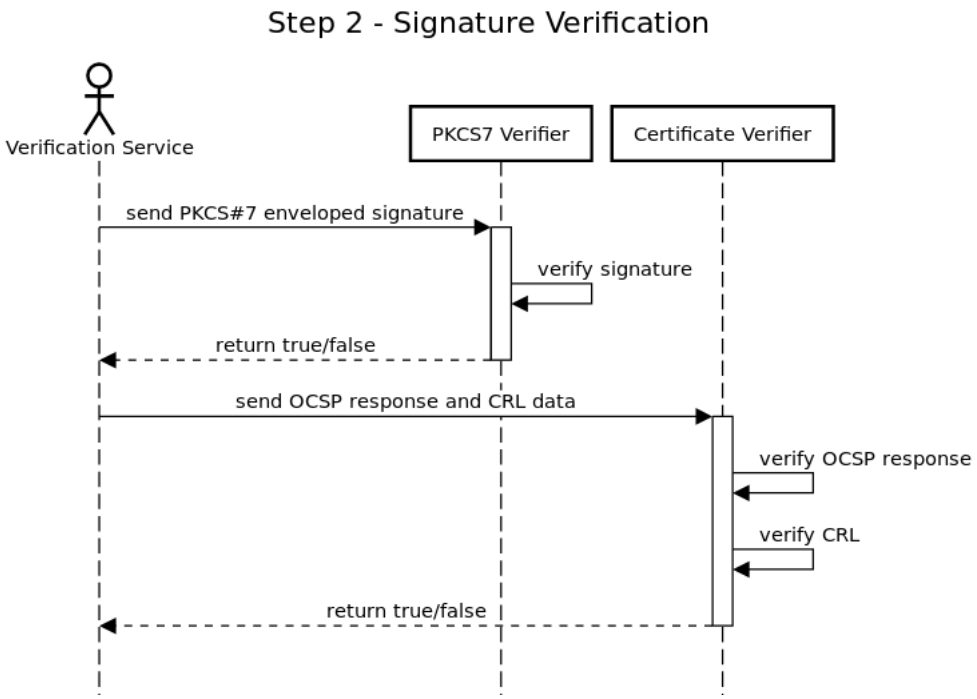

Figure 3.10.: Signature verification step

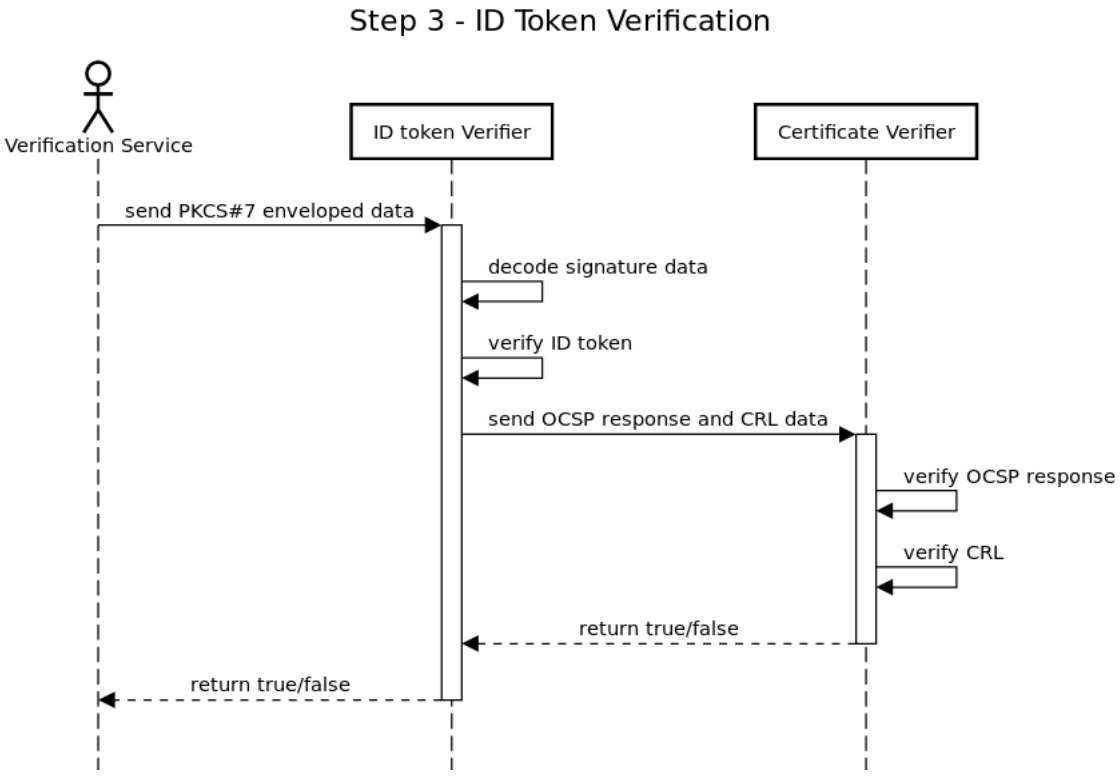

Figure 3.11.: ID token verification

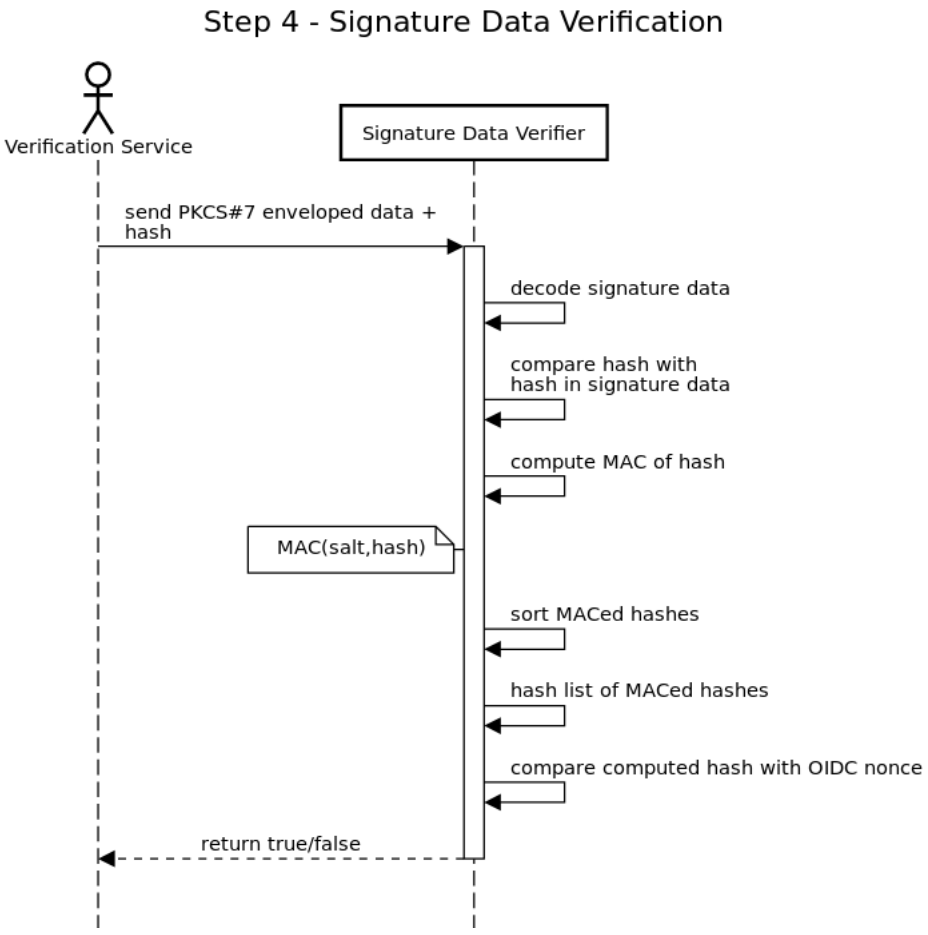

Figure 3.12.: Signature data verification

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

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

Listing 3.5: sign response

signatures/:id

Input:

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

Table 3.7.: 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 3.6: signature request

Sample Response:

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

<binary data follows>
```

Listing 3.7: signature response

3.3.2. Verification Service

Endpoints:

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

Table 3.8.: 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 3.9.: 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 3.10.: Output of the signature verification endpoint

Sample Request:

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

{
"hash": "e8a96e6203b9c0df058ba862abc63d9c520157faef6d5d54e54e526b0a85b2be",
"signature":
   "IyMjIFJFU1QgQVBJIFNwZWNpZmljYXRpb24gU2NyYXRjaHBhZAoKIyMjIyBQcmUtQXV0aCBlbmRw...b2lud
}
```

Listing 3.8: 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"
        },
        ...
      ]
    },
    "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"
        },
        ...
      ]
    }
  }
```

[caption=sign response, captionpos=b, language=JavaScript, label=lst:verifyresponse]

Invalid signature:

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

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

Listing 3.9: sign response

3.4. Signature File Format

Digital signatures unfortunately aren't very simple. Several components have to be put together for them to work. TODO better introduction, explain why so many components are needed.

3.4.1. 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 [19] CMS message along with certificates and revocation information.
- This CMS is encoded with Distinguished Encoding Rules (DER)[4] and added to a Protobuf message.
- A number RFC 3161 [18] timestamps are added to the same Protobuf message.

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

3.4.2. 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 3.10: SignatureData schema

salted_document_hash is a list of salted document hashes. The are salted as described in section 3.2.1. 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 done see 3.2.1.

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 2.2.2.

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.

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

jwk_idp is the JWK according to RFC 7517 [21] 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 [21] 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 3.4.4.

Putting it together These fields are put into the protobuf message `SignatureData` (for the schema, see listing 3.10). 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 [19] 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 [19, 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 3.4.3.

3.4.3. 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 [18] 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 useless.

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 observer 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 [22, 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 3.4.4.

Combining the SignatureData CMS with the RFC 3161 timestamps

For adding the timestamp(s) to the CMS message created in 3.4.2 we define another Protobuf message as follows:

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

Listing 3.11: SignatureFile message

signature_data is the DER-encoded CMS message created as described in 3.4.2.

rfc3161 is one or more DER-encoded timestamps according to RFC 3161 [18].

Any number of timestamps can be added. Timestamps either confirm `signature_data` or another timestamp earlier in the chain. The chains of timestamps that are formed that way can be used for LTV as described in 3.4.4.

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

3.4.4. Long-Term Validation

Long-Term Validation allows for the validation of signatures long after the document was signed [31].

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.

4. Implementation

4.1. Choices in Technologies

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

4.1.1. 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 [9] using the Ktor framework [10]. For the signing service we chose Kotlin because it is a modern and concise programming language, providing guaranteed null safety, coroutines for asynchronous programming, higher-order functions and more, 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.

Verifier

The verification service is implemented in the Go programming language [7]. The verification service being separate allows us to distribute a smaller program, and since Go binaries are always statically linked we don't have to worry about shipping dependencies. TODO more reasoning about not using generics.

4.1.2. 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 Graphical User Interface (GUI) library, whereas on the phones we could use Flutter [6]. 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 [23]. We discussed implementing a Single Page Application (SPA) using Angular [2], 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.

4.1.3. 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` [24] API
2. Using the `CryptoJS` [8] JavaScript implementation
3. Using a WebAssembly (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 [24], which can be used to calculate Secure Hash Algorithm 2 (SHA-256) checksums. 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 4.1.

```
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 4.1: 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).

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

Using CryptoJS

CryptoJS does not have the limitation of SubtleCrypto and supports progressive hashing[2], as seen in listing 4.2.

```
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 4.2: Progressive SHA-256 hashing using CryptoJS

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 [4]. 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. The code is in appendix A. For the Rust-based implementation we used the built-in web server of webpack [25].

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 4.3, 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.

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

Listing 4.3: Golang WASM function signature

[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.

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

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 4.4.

```
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 4.4: Uint8Array to []byte

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 [16]. 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, in order to see how they compare both in performance and ease of development.

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 the aforementioned options 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 code used for each example is in appendix B. The tests were run on Debian 10 using Firefox 69 on an Intel i7-8550U. The results can be seen in figure 4.1.

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. In order to provide a reference to compare the hashing speeds to we include the performance of the `openssl` command-line programme.

4.1.4. Deciding On The In-Browser Hashing Implementation

It is clear from figure 4.1 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 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 4.2.

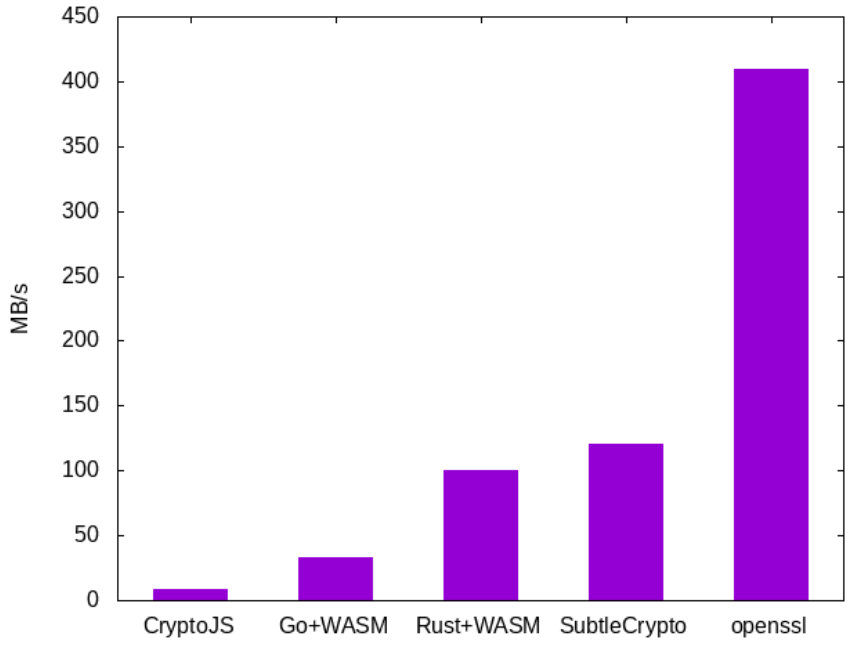

Figure 4.1.: Hashing speed in MB/s (higher is better)

Figure 4.2.: Code Completion in Intellij

4.2. Implementation Components

4.2.1. 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 [11].

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.

4.2.2. Components of the Signing Service

There are two 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.

For a Unified Modeling Language (UML) component diagram showing a simplified overview of the components, see figure 4.3.

4.2.3. Components of the Verifier

The verifier has an embedded webserver with a single API endpoint. The webserver handles the HTTP side of things. It interfaces with the verifier modules, which are split by responsability (e.g. `id_token` verification). For a UML component diagram showing a simplified overview of the components, see figure 4.4.

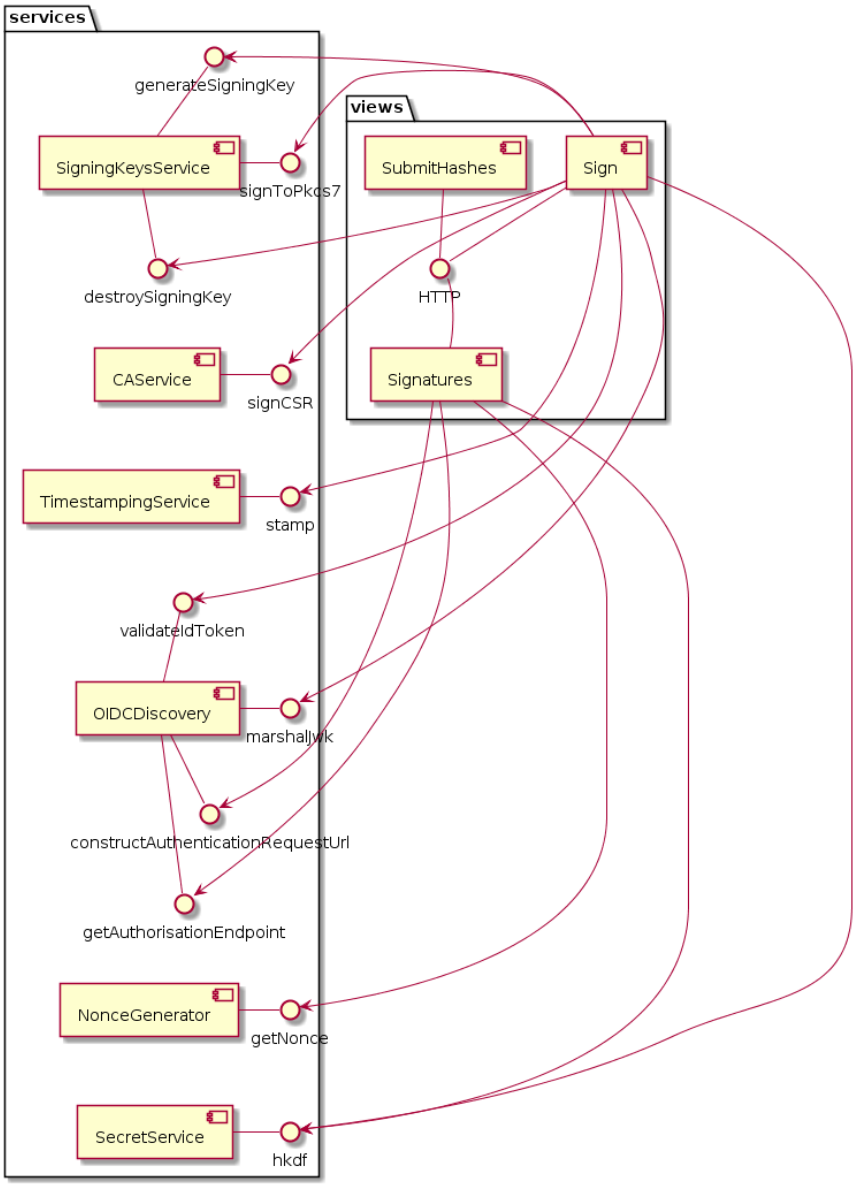

Figure 4.3.: Signing Service Components

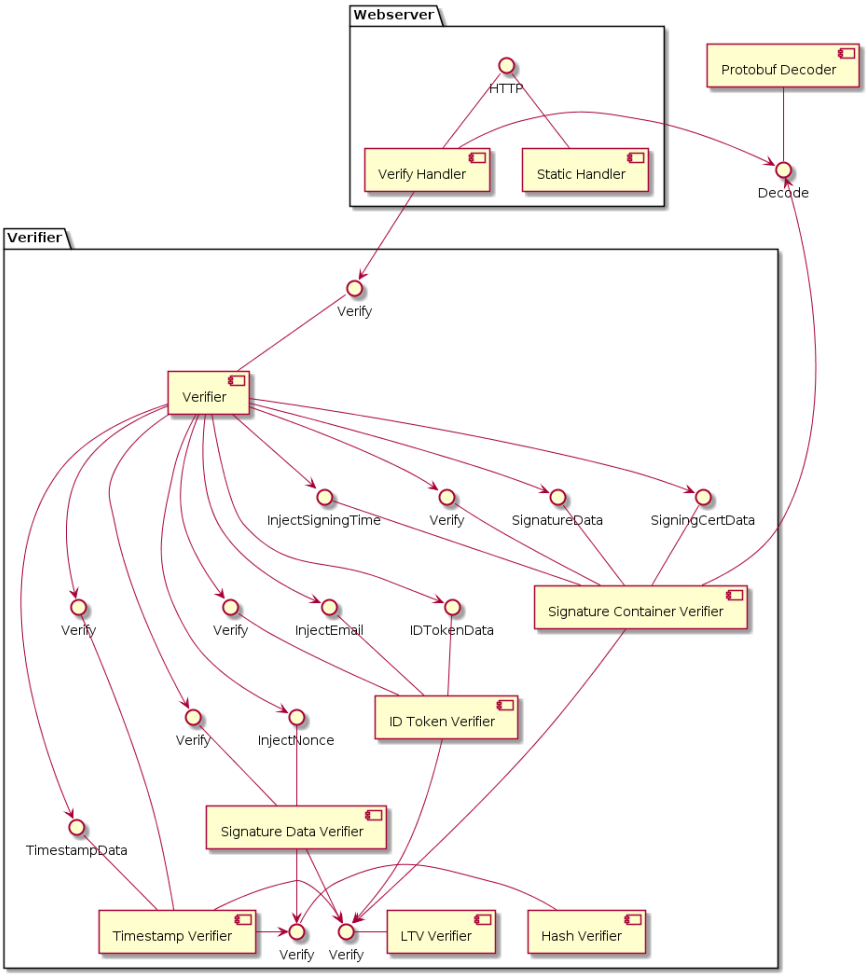

Figure 4.4.: Verifier Components

5. CSC Standard

5.1. CSC Specification

The Cloud Signature Consortium (CSC) has been formed to standardise cloud-based digital signatures, while meeting the European Union (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].

5.1.1. Comparison

One main difference is that in the Cloud Signature Consortium (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. Furthermore it allows for the identity provider and the signing service to be the same system and doesn't use standard OIDC with support for HTTP Basic or Digest authentication. This means that standard OIDC IDPs can't be used with the signing service, which is a disadvantage: before the IDP can be used, it has to implement the extensions specified by the CSC standard. Another difference is that a Signature activation data (SAD) is returned, 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.

5.2. Adobe Remote Signing

Adobe allows remote signing through its document cloud. To this end Adobe allows users to be authenticated by its own Adobe Approved Trust List (AATL) and the European Union Trust List (EUTL), which contain over 200 providers [1]. The solution claims to comply with eIDAS to provides Advanced Electronic Signatures (AES) and Qualified Electronic Signatures (QES) with LTV.

6. Yubikey HSM2

6.1. 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 [26]. We were offered one from G. Hassenstein to investigate whether it could be used in our work.

6.2. 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 (PKCS)#11 interface, it is possible to connect to it over the network as well, which would enable us to really have 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 [26].

6.3. SDK

Yubico offers a Software Development Kit (SDK) [26] 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 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.

6.3.1. 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 [26].

6.3.2. Shell

The yubihsm-shell is used for configuration of the the device. The full command reference can be found on the yubico website [27].

6.3.3. 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 [12] 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.

6.3.4. 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.

6.3.5. 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 [15].

6.4. Conclusion

TODO insert programming language. Using a HSM would definitively make our application more secure. Unfortunately the SDK only provides libraries for C and Python and not for #insertprogramminglanguage.

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 #insertprogramminglanguage, 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). However, we could make use of the PKCS#11 module and access the YubiHSM that way.

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

7. 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.

7.1. 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 TODO link to section where this is explained in more detail. 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, inspired by blockchain technologies. 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, everyone could see this by inspecting the data structure, because the signature file would not be published in it.

7.2. 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.

7.2.1. Nested Signatures

One possibility is that the next 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.

7.2.2. Pairing-based Signatures

With pairing-based cryptography like Boneh-Lynn-Shacham (BLS) [30] 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.

7.3. CADES Signatures

TODO supporting CMS/CADES signatures as defined in `https://www.etsi.org/deliver/etsi_ts/101700_101799/101733/02.02.01_60/ts_101733v020201p.pdf` for inclusion in pdfs

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

8. Declaration of primary authorship

I / We hereby confirm that I / 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 me are marked as quotations and provided with the exact indication of its origin.

Place, Date:	[Biel/Burgdorf], 07.02.2014	
Last Name/s, First Name/s:	[Test Peter]	[Müster Rösä]
Signature/s:	.....	.....

Bibliography

[1] "Adobe digital signatures faq." [Online]. Available: https://acrobat.adobe.com/us/en/sign/capabilities/digital-signatures-faq.html

[2] "Angular: One framework. mobile & desktop." [Online]. Available: https://angular.io/

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

[4] "Cors: Cross-origin resource sharing." [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/HTTP/CORS/Errors/CORSRequestNotHttp

[5] "Evs 821:2014: Format for digital signatures." [Online]. Available: https://www.evs.ee/products/evs-821-2014

[6] "Flutter mobile application development framework." [Online]. Available: https://flutter.dev/

[7] "The go programming language." [Online]. Available: https://golang.org/

[8] "Google cryptojs: A javascript implementation of standard and secure cryptographic algorithms." [Online]. Available: https://cryptojs.gitbook.io/docs/

[9] "The kotlin programming language." [Online]. Available: https://kotlinlang.org

[10] "Ktor: asynchronous web framework for kotlin." [Online]. Available: https://ktor.io/

[11] "Law of demeter." [Online]. Available: https://en.wikipedia.org/wiki/Law_of_Demeter

[12] "Libyubihsm." [Online]. Available: https://developers.yubico.com/YubiHSM2/Component_Reference/yubihsm-shell/

[13] "Loi fédérale sur les services de certification dans le domaine de la signature électronique et des autres applications des certificats numériques." [Online]. Available: https://www.admin.ch/opc/fr/classified-compilation/20131913/index.html

[14] "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

[15] "Pkcs#11 with yubihsm 2." [Online]. Available: https://developers.yubico.com/YubiHSM2/Component_Reference/PKCS_11/

[16] "Post on google groups, wasm specification impeding goroutines." [Online]. Available: https://groups.google.com/d/msg/golang-nuts/YJefPwnzpzQ/Lm5NznW3DQAJ

[17] "Rfc 2104: Hmac: Keyed-hashing for message authentication." [Online]. Available: https://tools.ietf.org/html/rfc2104

[18] "Rfc 3161: Internet x.509 public key infrastructure time-stamp protocol (tsp)." [Online]. Available: https://tools.ietf.org/html/rfc3161

[19] "Rfc 5652: Cryptographic message syntax (cms)." [Online]. Available: https://tools.ietf.org/html/rfc5652

[20] "Rfc 5869: Hmac-based extract-and-expand key derivation function (hkdf)." [Online]. Available: https://tools.ietf.org/html/rfc5869

[21] "Rfc 7517: Json web key (jwk)." [Online]. Available: https://tools.ietf.org/html/rfc7517

[22] "Rfc 7519: Json web tokens." [Online]. Available: https://tools.ietf.org/html/rfc7519

[23] "Typescript: A typed superset of javascript that compiles to plain javascript." [Online]. Available: https://www.typescriptlang.org/

[24] "Web crypto apis: The subtlecrypto class." [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/API/SubtleCrypto

[25] "Webpack module bundler." [Online]. Available: https://webpack.js.org/

[26] "Yubihsm." [Online]. Available: https://www.yubico.com/product/yubihsm-2/

[27] "Yubihsm shell." [Online]. Available: https://developers.yubico.com/YubiHSM2/Component_Reference/yubihsm-shell/

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

[29] 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

[30] D. Boneh, C. Gentry, H. Shacham, and B. Lynn, "Aggregate and verifiably encrypted signatures from bilinear maps," Eurocrypt 2003, LNCS 2656, Tech. Rep., 2003.

[31] E. T. S. I. ETSI, "Etsi ts 102 778-1 v1.1.1: Electronic signatures and infrastructures (esi): Pdf advanced electronic signature profiles," ETSI, Tech. Rep., 2009.

[32] ——, "Etsi ts 319 411: Policy and security requirements for trust service providers issuing certificates," ETSI, Tech. Rep., 2018.

[33] P. H. Gabor Tanz, "Projekt 2: Aufbau einer dss infrastruktur," Bern University of Applied Sciences, Tech. Rep., 2018.

[34] S. N. Jones M., Bradley J., "Rfc 7515: Json web signature (jws)," IETF, Tech. Rep., 2015.

[35] N. I. of Standards and Technology, "Nist special publication 800-63a: Digital identity guidelines: Enrollment and identity proofing requirements," NIST, Tech. Rep., 2017.

[36] ——, "Nist special publication 800-63b: Digital identity guidelines: Authentication and lifecycle management," NIST, Tech. Rep., 2017.

Acronyms

AAL Authenticator Assurance Level.

AAL2 Authenticator Assurance Level 2.

AAL3 Authenticator Assurance Level 3.

AATL Adobe Approved Trust List.

AES Advanced Electronic Signatures.

API Application Programming Interface.

AVX2 Advanced Vector Extensions 2.

BLS Boneh-Lynn-Shacham.

CA Certificate Authority.

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.

CSS Cascading Stylesheets.

DER Distinguished Encoding Rules.

DoS Denial of Service.

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.

EUTL European Union Trust List.

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.

IAL3 Identity Assurance Level 3.

IDE Integrated Development Environment.

IDP Identity Provider.

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.

PDF Portable Document Format.

PEM Privacy-Enhanced Mail.

PIN Personal Identification Number.

PKCS Public Key Cryptography Standard.

PKCS7 Public Key Cryptography Standard 7.

PSS Probabilistic Signature Scheme.

QES Qualified Electronic Signatures.

REST Representational State Transfer.

RFC Request For Comments.

RSA Rivest Shamir Adleman.

RSA-PSS RSA Probabilistic Signature Scheme.

SAD Signature activation data.

SDK Software Development Kit.

SHA Secure Hash Algorithm.

SHA-256 Secure Hash Algorithm 2.

SIM Subscriber Identity Module.

SPA Single Page Application.

TLS Transport Layer Security.

TSA Time Stamping Authority.

UML Unified Modeling Language.

USB Universal Serial Bus.

VM Virtual Machine.

WASM WebAssembly.

XML Extensible Markup Language.

List of Figures

List of Tables

Listings

A. Web Server from Go's Standard Library

```
package main

import (
    "log"
    "net/http"
)

func main() {
    log.Print("listening on :8080")
    log.Fatal(http.ListenAndServe(":8080", http.FileServer(http.Dir("."))))
}
```

B. Source Code for In-Browser Hashing

B.1. Golang

```
package main

import (
    "crypto/sha256"
    "fmt"
    "syscall/js"
    "time"
)

var hasher = sha256.New()
var start time.Time

func registerCallbacks() {
    js.Global().Set("progressiveHash", js.FuncOf(progressiveHash))
    js.Global().Set("startHash", js.FuncOf(startHash))
    js.Global().Set("getHash", js.FuncOf(getHash))
}

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

func startHash(this js.Value, in []js.Value) interface{} {
    start = time.Now()
    fmt.Printf("Start: %s\n", start.Format(time.RFC3339Nano))
    hasher = sha256.New()
    return this
}

func getHash(this js.Value, in []js.Value) interface{} {
    hash := hasher.Sum(nil)
    hashStr := fmt.Sprintf("%x", hash)
    fmt.Printf("Hash: %s\n", hashStr)
    end := time.Now()
    fmt.Printf("End: %s\n",end.Format(time.RFC3339Nano))
    fmt.Printf("Took %s\n", end.Sub(start))

    return js.ValueOf(hashStr)
}

func waitForever() {
    c := make(chan struct{}, 0)
```

```
    <-c
}

func main() {
    fmt.Println("WASM Go Initialized")
    registerCallbacks()
    waitForever()
}
```

B.2. Rust

```
extern crate sha2;
extern crate wasm_bindgen;

use std::cell::Cell;
use std::string::String;

use sha2::Digest;
use sha2::Sha256;
use wasm_bindgen::prelude::*;

#[wasm_bindgen]
pub struct Sha256hasher {
    hasher: Cell<Sha256>,
}

#[wasm_bindgen]
impl Sha256hasher {
    #[wasm_bindgen(constructor)]
    pub fn new() -> Sha256hasher {
        Sha256hasher {
            hasher: Cell::new(Sha256::default()),
        }
    }

    pub fn update(&mut self, input_bytes: &[u8]) {
        let hasher = self.hasher.get_mut();
        hasher.input(input_bytes)
    }

    pub fn hex_digest(&mut self) -> String {
        let hasher = self.hasher.take();
        let output = hasher.result();
        self.hasher = Cell::new(Sha256::default());
        return format!("{:x}", output);
    }
}
```

B.3. Reading a file piece-wise in TypeScript

```
interface FileChunkDataCallback {
    (data: Uint8Array): void
}

interface ErrorCallback {
    (message: string): void
}

interface FileReaderOnLoadCallback {
    (event: ProgressEvent): void
}

interface ProgressCallback {
    (percentCompleted: number): void
}

interface ProcessingCompletedCallback {
    (startTime: Date, endTime: Date, fileSize: number): void
}


class FileInChunksProcessor {
    public readonly CHUNK_SIZE_IN_BYTES: number = 1024 * 1000 * 20;
    private readonly fileReader: FileReader;
    private readonly dataCallback: FileChunkDataCallback;
    private readonly errorCallback: ErrorCallback;
    private readonly processingCompletedCallback: ProcessingCompletedCallback;
    private readonly progressCallback: ProgressCallback;
    private start: number = 0;
    private end: number = this.start + this.CHUNK_SIZE_IN_BYTES;
    private startTime?: Date;
    private numChunks: number = 0;
    private chunkCounter: number = 0;
    private inputFile: File | null = null;

    constructor(dataCallback: FileChunkDataCallback,
        errorCallback: ErrorCallback,
        progressCallback: ProgressCallback,
        processingCompletedCallback: ProcessingCompletedCallback) {
            this.fileReader = new FileReader();
            this.fileReader.onload = this.getFileReadOnLoadHandler();
            this.dataCallback = dataCallback;
            this.errorCallback = errorCallback;
            this.processingCompletedCallback = processingCompletedCallback;
            this.progressCallback = progressCallback;
    }

    public processChunks(inputFile: File) {
        this.inputFile = inputFile;
        this.startTime = new Date();
        this.numChunks = Math.round(this.inputFile.size /
           this.CHUNK_SIZE_IN_BYTES);
        this.read(this.start, this.end);
    }

    public getFileFromElement(elementId: string): File | undefined {
```

```
        const filesElement = q(elementId) as HTMLInputElement;

        if (Validate.notNull(filesElement.files)) {
            if (filesElement.files.length < 0) {
                this.errorCallback("Too few files specified. Please select one
                    file.")
            } else if (filesElement.files.length > 1) {
                this.errorCallback("Too many files specified. Please select
                    one file.")
            } else {
                return filesElement.files[0];
            }
        }
    }

    private getFileReadOnLoadHandler(): FileReaderOnLoadCallback {
        return () => {
            if (Validate.notNull(this.inputFile)) {
                this.dataCallback(new Uint8Array((this.fileReader.result as
                    ArrayBuffer)));

                this.start = this.end;
                if (this.end < this.inputFile.size) {
                    this.chunkCounter++;
                    this.end = this.start + this.CHUNK_SIZE_IN_BYTES;
                    this.progressCallback(
                    Math.round((this.chunkCounter / this.numChunks) * 100));
                    this.read(this.start, this.end);
                } else {
                    if (Validate.notNullNotUndefined(this.startTime)) {
                        this.processingCompletedCallback(
                            this.startTime, new Date(), this.inputFile.size);
                        }
                    this.progressCallback(100);
                }
        }
    }
}

private read(start: number, end: number) {
    if (Validate.notNull(this.inputFile)) {
        this.fileReader.readAsArrayBuffer(this.inputFile.slice(start, end));
    }
}

function errorHandlingCallback(message: string) {
    const errorElement = q("error");
    errorElement.innerHTML = `${errorElement.innerHTML} <p>${message}</p>`;
}

function progressCallback(percentCompleted: number) {
    if (percentCompleted == 100) {
        q("progress").innerText = `Completed!`
    } else {
        q("progress").innerText = `Hashing: ${percentCompleted}%`
    }
}
```

```
export function processFileButtonHandler(wasmHasher: Sha256hasher) {
    const startElement = q("start");
    const startTime = new Date();
    startElement.innerText = "started at " + dateObjectToTimeString(startTime);

    const processor = new FileInChunksProcessor((data) => {
            wasmHasher.update(new Uint8Array((data)));
        },
        errorHandlingCallback,
        progressCallback,
        (startTime, endTime, sizeInBytes) => {
            const hashStr = wasmHasher.hex_digest();
            wasmHasher.free();
        }
    );
    const file = processor.getFileFromElement("file");
    if (Validate.notNullNotUndefined(file)) {
        processor.processChunks(file);
    }
}
```