Functional Specification Document

Objectives, Scope and Project Planning

For the Bachelor Thesis by Gabor Tanz and Patrick Hirt

Degree course:	Computer Science
Authors:	Gabor Tanz, Patrick Hirt
Tutor:	Dr. Annett Laube, Gerhard Hassenstein
Experts:	Dr. Andreas Spichiger
Date:	18th October 2019

Contents

1 Introduction

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

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

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

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

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

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

We will allow people to create electronic signatures, no matter where they are, or what device they're using, in a secure manner. Building on our previous work of Project 2 [14], we show how it is possible to securely integrate OpenID Connect (OIDC) authentication with remote digital signatures. We expand upon this previous work and show how it is possible to have a remote signing service with the capability of signing on the users' behalf without the need for completely trusting that service. Furthermore, we compare our solutions to those proposed by an industrial consortium led by Adobe Inc., and we show in which ways we believe our approach to be superior.

1.0.1 Purpose of this document

In this document we will outline the objectives, scope and methodologies for our thesis as well as provide a project timeline. The implementation will be documented separately.

1.1 A cryptographic primer

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

1.1.1 Hash Function

A hash function in cryptography is an one-way function which is able to map data of arbitrary length to fixed-size values [6]. One-way means that for a given hash value, it is infeasible to find the corresponding input data. Ideally, the only way for someone to invert such a hash function is to do an exhaustive brute-force search. This is called pre-image resistance. Furthermore, a cryptographic hash function needs to fulfil the following properties:

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

Hash functions fulfilling these properties are fundamental to our work (and to much of cryptography in general). Without them we would be completely powerless. An example for such a hash function is Secure Hash Algorithm 2 (SHA-2) [8].

1.1.2 Asymmetric cryptography

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

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

This might sound simple on the surface but opens up a world of possibilities. Only the private key has to be kept secret, the public key can be freely published [12].

The classical example for such an encryption system is the Rivest-Shamir-Adleman (RSA) scheme [11].

For a simplified example how public-key-based encrypted communication between two parties could work, [1] see figure 1.1.

1.1.3 Digital Signatures

Having briefly explained Hash Functions in 1.1.1 and Asymmetric Encryption in 1.1.2 we can now move on to introducing digital signatures. A digital signature is a way for verifying the integrity and authenticity of a message, that is, to know who the message author is and to guarantee that it wasn't tampered with [10].

Digital Signatures are not Electronic Signatures Please note that the Digital Signatures we describe here are distinct from Electronic Signatures. Electronic signatures provide the same legal standing as a hand-written signature on paper, and as such are defined in laws such as ZertES [9]. Digital signatures on the other hand merely refer to a mathematical scheme for providing message integrity and authenticity. Digital signatures are used to implement electronic signatures, but they're not equivalent.

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

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

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

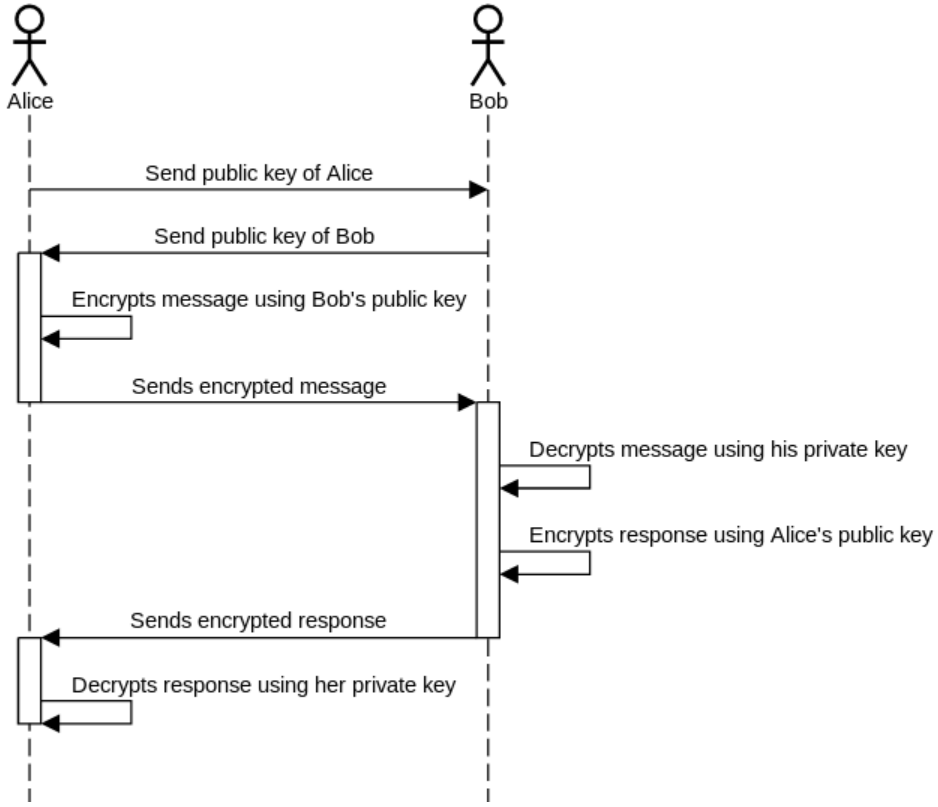

Figure 1.1: Simplified example of two Actors, Alice and Bob, exchanging encrypted messages using public-key cryptography

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

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

In the message exchange shown in figure 1.1, there is a problem: anyone could encrypt messages for Bob and pretend to be Alice, since his public key is, well, public.

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

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

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

However, there is a serious problem still: an evil actor with the ability to intercept the communication between Alice and Bob could not only read their messages, but change them at will, effectively impersonating Bob as seen from Alice, and Alice as seen from Bob. For a solution to this problem please see section 1.1.3.

Figure 1.2 expands upon figure 1.1 to illustrate this attack.

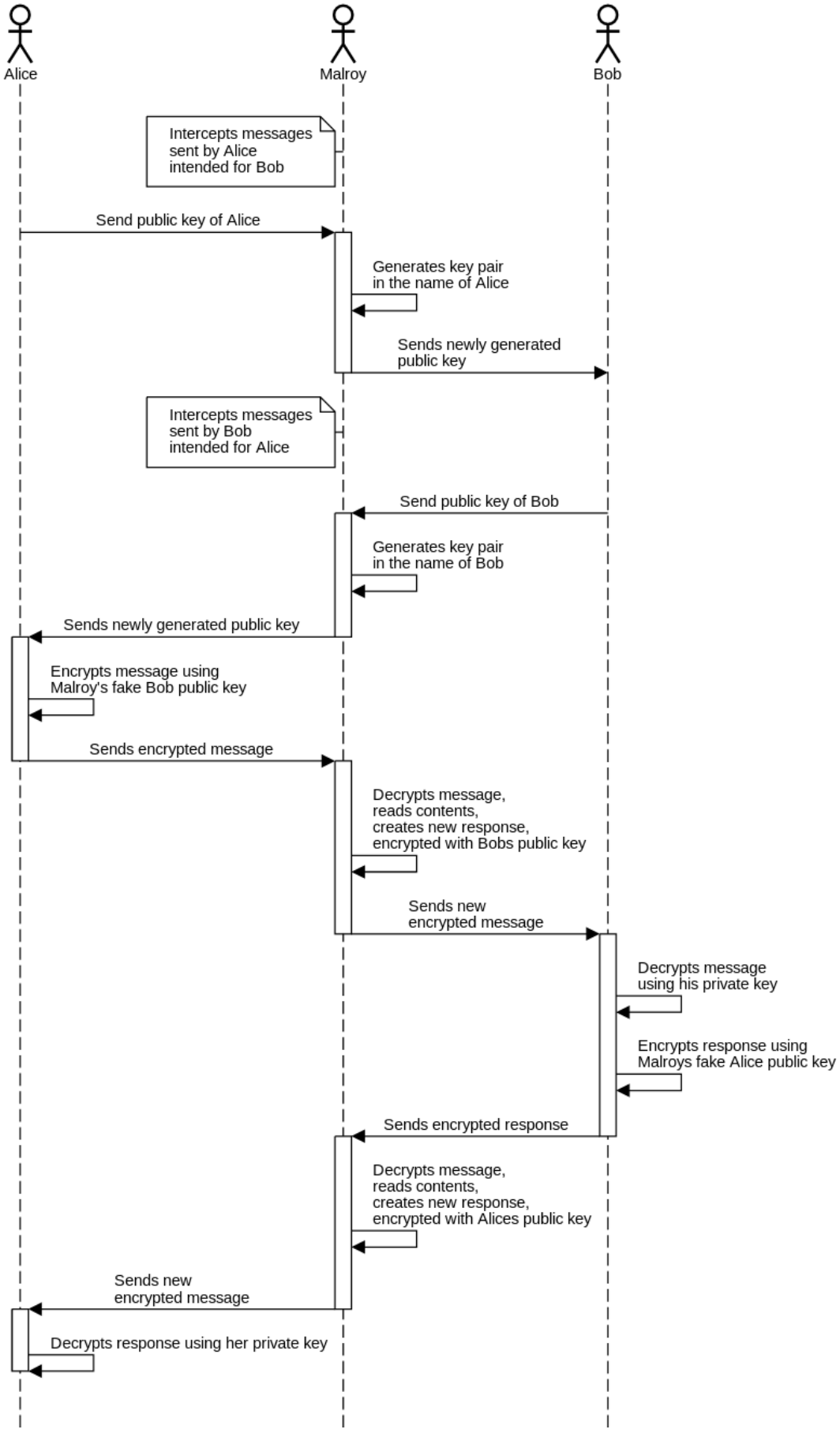

Figure 1.2: Man in the middle attack on unauthenticated public-key encrypted communication

1.1.4 Public-Key Infrastructure and Certificate Authorities

Public-key encrypted and authenticated communication as described in chapter 1.1.3 is vulnerable to man-in-the-middle attacks as illustrated in figure 1.2. This attack works because Malroy is able to mislead Bob and Alice to use his keys instead of theirs by intercepting their public keys in the initial key exchange.

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

This one of the problems a PKI solves. On an abstract level, a PKI is a mechanism that couples a public key with an identity [15]. What this means for the attack shown in figure 1.2 is that it provides Alice and Bob a way to make sure they're using each others' keys and not Malroys', thus preventing the attack. Because Alice and Bob now have a mechanism to verify which identity a public key refers to, they can detect Malroys attack because the public keys maliciously issued by him will not correspond to Alice nor Bob.

A well-known and widely-used example for such a PKI is X.509 [7]. In practice such PKIs are complex, and because this section's already become longer than we like we'll forego explaining how X.509 works.

1.1.5 Trusted Digital Timestamping

Trusted digital timestamping is a scheme for proving the existence of a piece of information at a certain point in time. There are several such schemes, such as X9.95 or ISO/IEC 18014. In this section we will focus on PKI-based timestamping as defined in RFC 3161 [1].

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

Trusted timestamps are created by using digital signatures (see 1.1.3) and hash functions (see 1.1.1). In order to create a timestamp, the following steps are performed:

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

For an illustration of this process, see figure 1.3

1.1.6 Summary

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

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

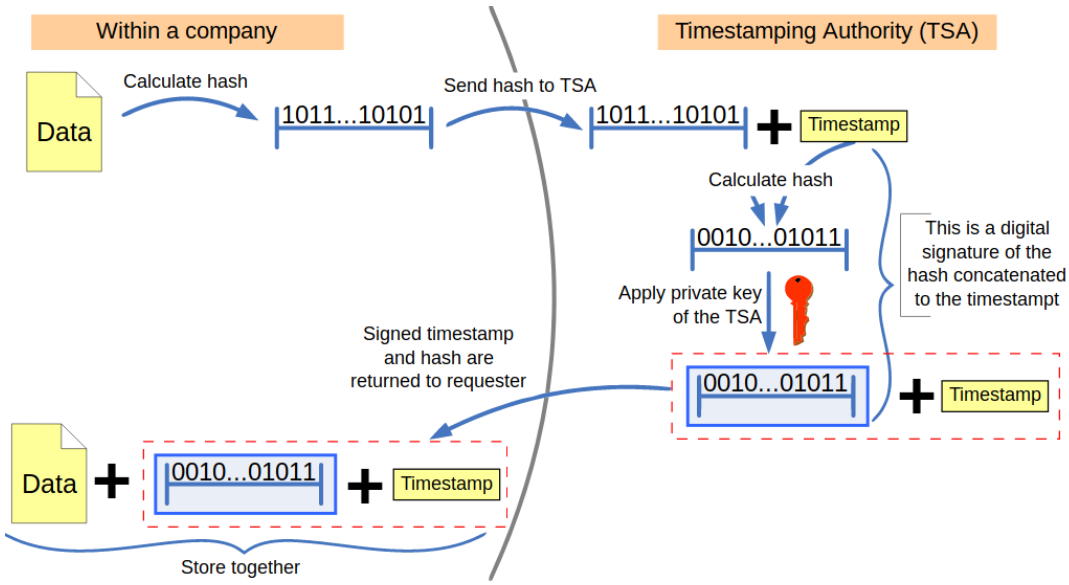

Figure 1.3: Process of obtaining a timestamp from a Time Stamping Authority (TSA). Source: `https://en.wikipedia.org/wiki/File:Trusted_timestamping.svg`

2 Objectives

2.0.1 Main Objective

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

2.1 Previous Works

We build upon our previous work of Project 2 [14], where we specified the authentication process for qualified signatures, non-qualified batch signatures, the signature file format, as well as - to our knowledge - pioneering the secure integration of a digital signature with an OIDC ID token without requiring any change to the IDP.

2.2 Backend

The server is the centrepiece of the service, where the actual signatures are being created. It depends on the IDP for authenticating its users. In our implementation, we will aim to protect the private keys by using a Hardware Security Module (HSM).

2.3 Frontend

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

2.4 Comparison With Existing Solutions

The Cloud Signature Consortium standardised a remote signing service with OIDC/oAuth. Adobe has made an implementation of this standard. The goal is to learn how this implementation works and compare it with our solution, with a focus on security.

2.5 Evaluation of the Yubikey HSM for Signing Service

In order to provide a secure solution for the signing keys, we will evaluate the Yubikey HSM 2. This would allow us to avoid having the signing keys on the filesystem, thus strongly improving the security of our solution.

3 Actors

Actors specify a role played by a user or a system for the purposes of a clearer definition. In this chapter, we will outline the actors in our system.

3.1 Big Picture

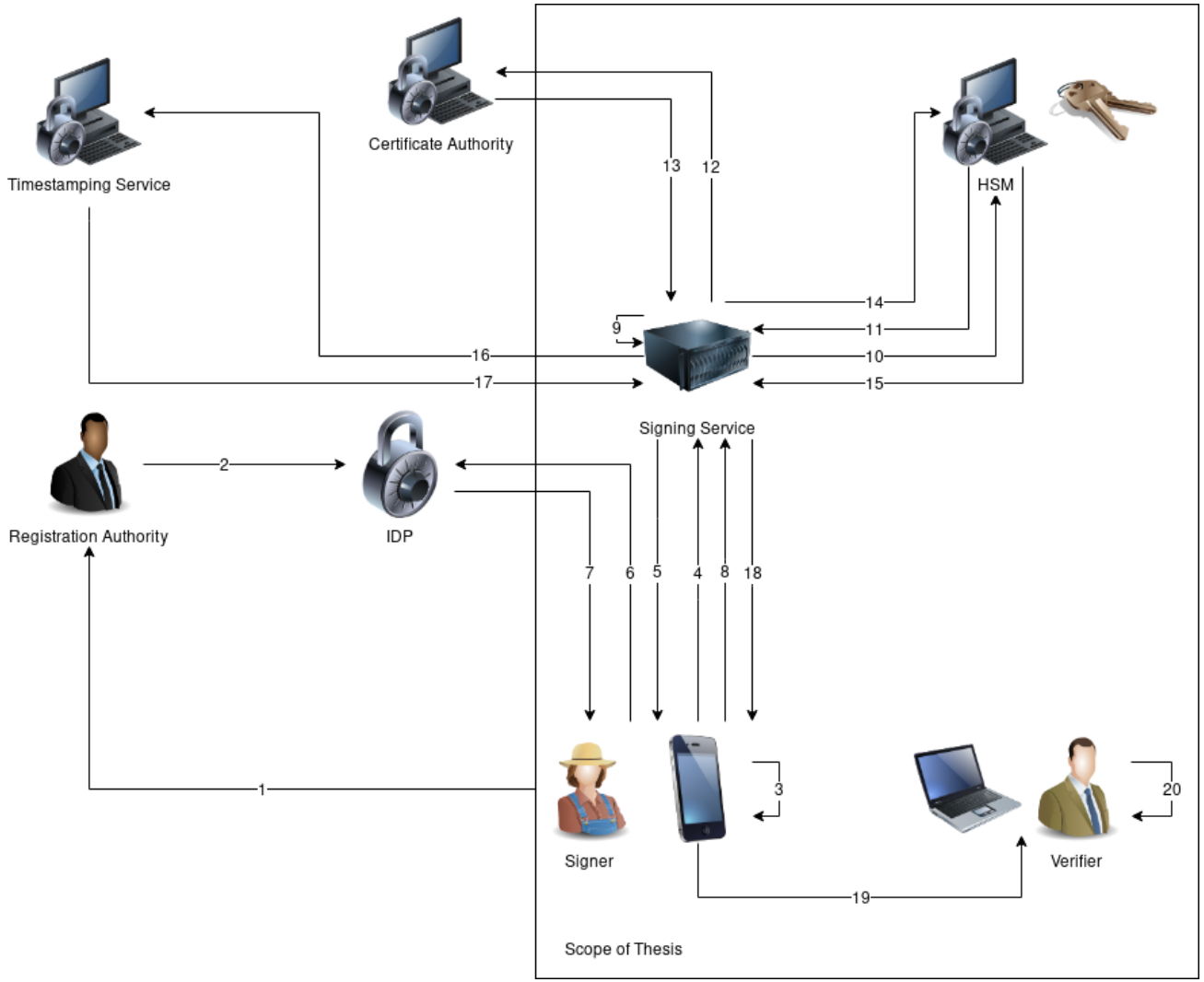

Figure 3.1: Big Picture

3.1.1 Registration

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

   A pre-requisite for using our signing service is that the user (Verifier) is registered with an IDP supported by the Signing Service. The user registers at the RA by proving their identity to it. This process to be completed only once.

2. Propagate identity to IDP (Identifier)

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

3.1.2 Identification

3. Generate document hash

   The user (Verifier) generates the hashes of the document(s) they wish to sign. This is needed before the identification step in order to prevent signing of arbitrary documents by securely binding the document hashes to the authentication request and subsequent identity assertion, as we've shown in our previous work [14].

4. Send hash to signing service

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

5. Receive OIDC redirect to IDP

   The signing service computes a nonce from the hashes and a random value. This nonce will be used in the OIDC request. Then it sends the redirect to the IDP to the user.

6. Login to IDP

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

7. Receive ID token

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

8. Send ID token to signing service

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

9. Verify ID token

   The signing service validates the ID token (JSON Web Token (JWT)) and checks if the nonce matches, thus completing the secure coupling of intent to sign specific documents with the authentication process.

3.1.3 Signature Generation

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

    The signing service requests a new signing key to be generated by the HSM in the name of the user.

11. Receive CSR

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

12. Send CSR to Certificate Authority (CA)

    The signing service sends the CSR to the CA in order for it to be signed.

13. Receive signed certificate

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

14. Request signature from HSM

    The signing service sends the hashes of the documents to be signed to the HSM.

15. Receive signature

    The HSM signs the hashes using the newly created signing key and returns the signature to the signing service.

16. Request timestamp from Timestamping Service (TSS)

    The signing service sends the signature value to the TSS.

17. Receive timestamp

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

18. Send signature to signer

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

3.1.4 Signature Verification

19. Send document and signature to receiver (Verifier)

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

20. Verify document and signature

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

3.2 The Signer

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

3.3 The Verifier

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

3.4 The Authenticator

The Authenticator is the system who authenticates The Signer as specified in section 3.2. In National Institute of Standards and Technology (NIST) terminology [5], this is the entity establishing the Identity Assurance Level (IAL). In order for The Authenticator to be able to authenticate The Signer, they must have been registered with The Authenticator by The Identifier as specified in section 3.5.

3.5 The Identifier

The Identifier is the system or person who asserts the identity of The Signer. In order for the signing service to issue qualified signatures as defined by relevant Swiss legislation [9] and Swiss Federal Council regulations [13], the identity must be proven in-person using a government-issued photographic identification document such as a passport.

3.6 The Signing Service

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

3.7 The Certificate Authority

The CA is the system who signs the signing keys generated by the Signing Service 3.6.

4 Functional Requirements

4.1 Terminology

As always, the usual modal verbs are to be interpreted as in RFC 2119 [4].

4.1.1 Practically Impossible

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

4.1.2 Being made difficult

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

4.2 Signature Requirements

4.2.1 Authenticity

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

4.2.2 Integrity

It must be practically impossible for anyone to modify a signed document without being detected upon signature verification. A secure hash algorithm must be used for hashing the document. Secure means the algorithm to be pre-image as well as collision-resistant as validated by the NIST Cryptographic Algorithm Validation Program [2].

4.2.3 Verifiability

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

4.2.4 Non-repudiation

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

4.2.5 Long-Term Validation

Signatures must be suitable for Long-Term Validation (LTV) using an RFC3161 [1] timestamp.

4.2.6 Secure Coupling of Authentication and Signature

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

4.2.7 Authentication Protocol

Standard OIDC must be used for authenticating The Signer as specified in the standard [3].

4.2.8 Supported File Formats

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

4.2.9 Bulk Signatures

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

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

4.2.10 Device-local Hashing of Documents

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

4.3 Signature Server Requiremenents

4.3.1 CA Key Security

Technical measures must be taken to make it difficult for the private keys of the signing CA to be stolen.

4.3.2 Signing Key Security

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

4.3.3 No unauthorised identity delegation

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

4.3.4 Random Number Generation

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

4.3.5 REST API

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

5 Non-Functional Requirements

5.0.1 Efficient Signature File Format

The file format for the signature file shall be based on our previous work [14].

5.0.2 Protection of Information

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

5.0.3 Offline Validation

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

5.1 IDP Requirements

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

5.1.1 Support for OIDC

The IDP must support standard OIDC as specified in the standard [3].

5.1.2 Levels of Assurance

The IDP must support Authenticator Assurance Level (AAL) 2 authentication for advanced signatures, and AAL 3 authentication for qualified signatures as specified in the NIST publication [5].

5.1.3 Code Quality

The code produced should be, wherever possible:

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

Public APIs should be documented.

5.1.4 Ease of Use

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

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

5.1.5 Reactive Design

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

5.2 Prioritisation of Requirements

Requirement	**Prioritisation**
Authenticity of signature (4.2.1)	Must
Integrity of document (4.2.2)	Must
Verifiability of signature (4.2.3)	Must
Non-repudiation (4.2.4)	Must
Secure coupling of authentication and signature (4.2.6)	Must
Authentication protocol (4.2.7)	Must
Supported file formats (4.2.8)	Must
No unauthorised identity delegation (4.3.3)	Must
Random number generation (4.3.4)	Must
REST API (4.3.5)	Must
Offline validation (5.0.3)	Must
Signing key security (4.3.2)	Optional
Protection of information (5.0.2)	Optional
Device-local hashing of documents (4.2.10)	Optional
Efficient signature file format (5.0.1)	Optional
CA key security (4.3.1)	Optional
Bulk signatures (4.2.6)	Optional
Long-term validation (4.2.5)	Optional
Code Quality (5.1.3)	Optional
Ease of Use (5.1.4)	Optional
Reactive Design (5.1.5)	Optional

Table 5.1: Prioritisation of Requirements

6 Use-Cases

6.1 Document Signing

6.1.1 Prerequisites

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

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

6.1.2 Interactive Qualified Signatures

Steps

The user performs the following steps:

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

Result

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

6.1.3 Bulk Advanced Signatures

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

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

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

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

Result

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

6.2 Signature Validation

6.2.1 Prerequisites

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

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

6.2.2 Offline Validation

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

Steps

The user performs the following steps:

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

Result

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

6.2.3 Semi-Online Validation

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

Prerequisites

In addition to the prerequisites specified in subsection 6.2.1, it is necessary for the user to have opened the signature services' verification web page in their browser beforehand so that it is available to the user without connecting to the server again. This is why we call it semi-online. For example, the user opens the web browser on their mobile phone and loads the verification page. Then they board an aeroplane and turn on aeroplane mode. After takeoff, they decide to validate a signature. Then they proceed as follows:

Steps

The user performs the following steps:

1. The user opens the web browser where the verification page is still available
2. The user selects the document file and corresponding signature file and submits it to the verification website
3. The website verifies the signature in-browser (without needing to connect to any additional servers) and displays the result

Result

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

7 Project Management

7.1 Project Method

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

7.2 Project Timeline

Since we know in advance the timeframes within we must complete our work, we created the following project timeline.

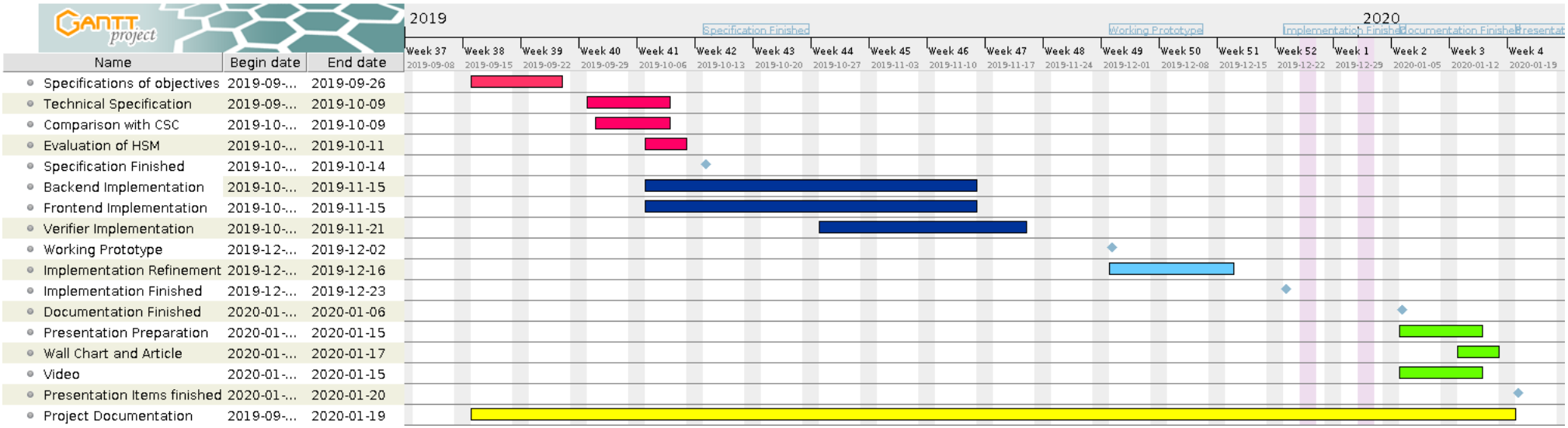

Figure 7.1: Project timeline

7.3 Work Packages

7.3.1 Specification of Objectives

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

7.3.2 Technical Specification

The requirements defined in the objectives will be mapped to concrete technology choices. Also the choice of languages and frameworks will be explained.

7.3.3 Comparison with CSC Implementation

We will compare our Specification with the Remote Signing Standard specified by the Cloud Signature Consortium (CSC).

7.3.4 Evaluation of Yubikey HSM

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

7.3.5 Backend Implementation

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

7.3.6 Frontend Implementation

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

7.3.7 Standalone Verifier Implementation

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

7.3.8 Implementation Refinement

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

7.3.9 Source Code Documentation

We will document the source code.

7.3.10 Presentation

We will create the slides and prepare the presentation.

7.3.11 Wall Chart and Article

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

7.3.12 Video

We will create a short video introducing our problem.

Bibliography

[1] "Internet X.509 Public Key Infrastructure: Time-Stamp Protocol (TSP)." [Online]. Available: https://tools.ietf.org/html/rfc3161

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

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

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

[5] P. A. Grassi, M. E. Garcia, and J. L. Fenton, "Nist special publication 800-63-3: Digital identity guidelines," National Institute of Standards and Technology, Tech. Rep., 2017.

[6] S. Halevi and H. Krawczyk, "Randomized hashing and digital signatures." [Online]. Available: http://www.ee.technion.ac.il/~hugo/rhash/

[7] I. T. U. T. S. S. ITU-T, "X.509: Public-key and attribute ertificate frameworks," 2016. [Online]. Available: https://www.itu.int/rec/T-REC-X.509

[8] N. I. of Standards and Technology, "Usa patent no. 6829355: Secure hash algorithm 2." [Online]. Available: https://worldwide.espacenet.com/textdoc?DB=EPODOC&IDX=US6829355

[9] Office of the Swiss Federal Chancellery, "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

[10] E. Paul, "What is digital signature - how it works, benefits, objectives, concept," Tech. Rep., 2019. [Online]. Available: https://www.emptrust.com/blog/benefits-of-using-digital-signatures

[11] R. Rivest, A. Shamir, and L. Adleman, "A method for obtaining digital signatures and public-key cryptosystems," Tech. Rep., 1978.

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

[13] Swiss Federal Council, "Ordonnance 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/20162168/index.html

[14] G. Tanz and P. Hirt, "Projekt 2: Aufbau einer DSS Infrastruktur," Bern University of Applied Sciences, Tech. Rep., 2018.

[15] Techtopia, "An overview of public key infrastructures (pki)," 2015. [Online]. Available: https://www.techotopia.com/index.php/An_Overview_of_Public_Key_Infrastructures_(PKI)

Acronyms

AAL Authenticator Assurance Level.

API Application Programming Interface.

CA Certificate Authority.

CSC Cloud Signature Consortium.

CSR Certificate Signing Request.

DSA Digital Signature Algorithm.

GUI Graphical User Interface.

HSM Hardware Security Module.

IAL Identity Assurance Level.

IDP Identity Provider.

JWT JSON Web Token.

LTV Long-Term Validation.

NIST National Institute of Standards and Technology.

OIDC OpenID Connect.

PKI Public Key Infrastructure.

RA Registration Authority.

REST Representational State Transfer.

RNG Random Number Generator.

RSA Rivest-Shamir-Adleman.

SHA-2 Secure Hash Algorithm 2.

TSA Time Stamping Authority.

TSS Timestamping Service.

UI User Interface.

List of Figures

List of Tables

Listings