UNIVERSITY OF TARTU
Institute of Computer Science
Software Engineering Curriculum

Toomas Aleksander Veromann

WYSIWYS Extensions to the Estonian ID Card Browser Signing Architecture

Master's Thesis (30 ECTS)

Supervisor: Arnis Paršovs, PhD

Tartu 2021

WYSIWYS Extensions to the Estonian ID Card Browser Signing Architecture

Abstract:

Since the first ID cards were issued in Estonia, hundreds of millions of electronic signatures have been created. As opposed to paper-based documents, where signatories have the option to inspect the documents before signing, the signatures that are given online, through browser extensions, are given by signers without being able to verify what is the actual data that is being signed. Instead of displaying the documents securely on the signer's device, service providers supply a hash value, which the signer must cryptographically sign. This so-called blind signing is convenient for service providers and signatories but does not protect signatories against service providers asking them to sign something that they may not be willing to sign. In this thesis, two What You See Is What You Sign (WYSIWYS) solutions were proposed to address this problem. The proposed solutions were implemented by modifying the existing ID card software and the results were subsequently analyzed. The proposed improvements to the existing browser signing solution enable users to inspect documents before signing, providing the possibility to sign documents in web environments with as much confidence as paper-based documents.

Keywords: Digital signing, WYSIWYS, Estonian ID card, chrome-token-signing, DigiDoc4

CERCS: P175 Informatics, systems theory

WYSIWYS edasiarendused Eesti ID-kaardi brauseris allkirjastamise arhitektuurile

Lühikokkuvõte:

Alates esimeste ID-kaartide väljastamisest Eestis on loodud sadu miljoneid digiallkirju. Erinevalt paberil esitavatest dokumentidega, kus allkirjastajatel on võimalus dokumentide sisuga eelnevalt tutvuda, antakse internetis allkirju ilma dokumentide tegelikku sisu nägemata. Allkirjastajatele näidatavate dokumentide asemel saadavad teenusepakkujad brauseris allkirjastamise laiendusele räsiväärtuse, mille pealt luuakse krüptograafiline allkiri. Sellist viisi allkirjastamine on mugav nii teenusepakkujatele kui ka allkirjastajatele, kuid ei kaitse allkirjastajaid selle eest, et teenusepakkujad paluvad neil allkirjastada midagi, millele kasutajad ei pruugi olla nõus alla kirjutama. Selle töö tulemusena valmis olemasolevale brauseris allkirjastamise lahendusele kaks "see, mida näed, seda allkirjastad" (ingl. *What You See Is What You Sign (WYSIWYS)*) täiendust. Väljapakutud täiendused implementeeriti kasutuselolevat ID-kaardi tarkvara modifitseerides ning leidusid analüüsiti. Need edasiarendused võimaldavad kasutajatel eelnevalt tutvuda veebikeskkondades allkirjastatavate dokumentidega ning dokumente allkirjastada sama enesekindlusega nagu paberdokumente.

Võtmesõnad: Digiallkirjastamine, WYSIWYS, Eesti ID-kaart, chrome-token-signing, DigiDoc4

CERCS: P175 Informaatika, süsteemiteooria

Contents

1 Introduction

The Estonian identity card is an Estonian government-issued identity document with digital capabilities, providing the possibility for digital authentication and signing. Along with identity cards, the Estonian government also issues other digital identity documents - residence permit cards, digital identity cards, e-resident's digital identity cards and diplomatic identity cards. All of these smart cards are referred to in this thesis under the general term "ID card".

The first Estonian electronic identity documents were issued in 2002, and the first digital signatures were given in the same year. Since then, thousands of e-services have been created and over 600 million digital signatures have been provided, solidifying Estonia's reputation as a digital state. Over the past 19 years, activities such as signing documents, making bank transfers and even casting votes in elections have been overwhelmingly changing from being paper-based to digital. [1, 2, 3]

Digital authentication and signing with the ID card is done with the help of ID card software. The Estonian ID card software bundle is provided and maintained by the Estonian Information System Authority (Riigi Infosüsteemi Amet - RIA) and comes with a desktop application named DigiDoc4 Client, as well as browser extensions for all major browsers. Over the years, several security flaws have been found with the Estonian ID card and its browser authentication and signing architecture, but generally, the Estonian ID card signing architecture has been deemed secure and remains widely regarded as the paragon, which other countries should seek to take as an example. [4, 5, 6]

Today, there are two main ways of how documents can be signed using the Estonian ID card:

1. signing documents in web environments using browser extensions;
2. signing documents using stand-alone applications such as the DigiDoc4 Client on desktop computers and laptops, or the RIA DigiDoc application on mobile devices.

The subsections below introduce the document signing process in web environments, and describe the problem related to the process that this work tries to address.

1.1 Signing Using Browser Extensions

In some cases, such as internet banking, there is a need for web applications to ask users to confirm certain actions by providing a digital signature. Signing in web environments is done through the use of browser extensions, which allow websites to use JavaScript calls to obtain a person's signature for a given hash [7]. Such signing processes are often started by users explicitly expressing intent to start signing, for example, by pressing a button on a service provider's website.

In order to compose a valid digital signature structure, service providers first ask for the user's public certificate through the browser extension. This public certificate must correspond to the private key which the document will be signed with. Figure 1 illustrates this certificate selection dialog as shown by the browser extension on the Linux platform. After the user has selected the relevant certificate, the browser extension returns this certificate back to the service provider.

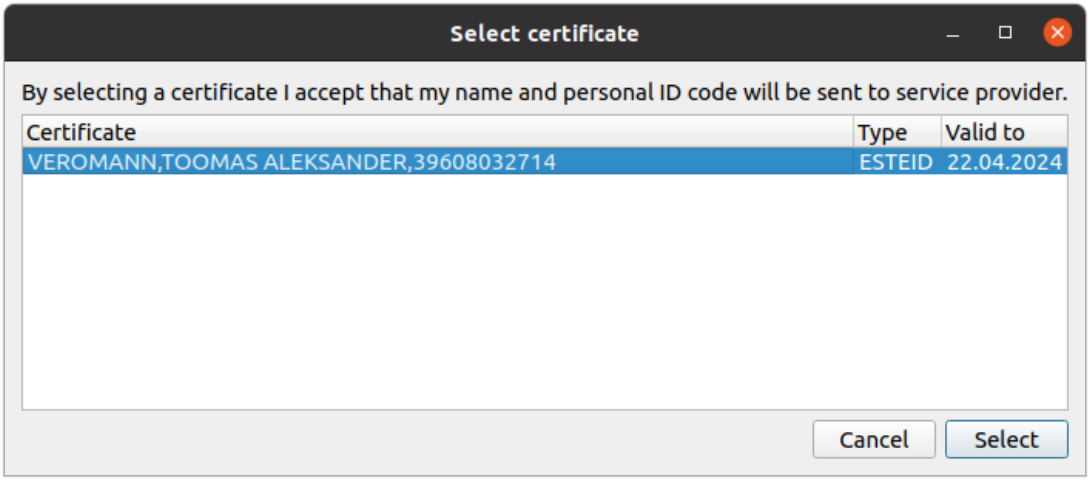


Figure 1. Certificate selection dialog in a web environment on the Linux platform

After the service provider has received the user's certificate, they can ask the user to provide their digital signature for a given hash. Figure 2 shows one such example - a bank asking for payment confirmation from a user by the user providing their digital signature. In the figure, the user is asked to enter their signing PIN (PIN2), whereas the user does not see what exactly are the contents of the document that is being signed. Once the user has entered their PIN2, a cryptographic signature is produced on the received hash

value. This raw cryptographic signature is then returned to the service provider through the browser extension. The service provider then validates the received signature value and composes a digital signature container.

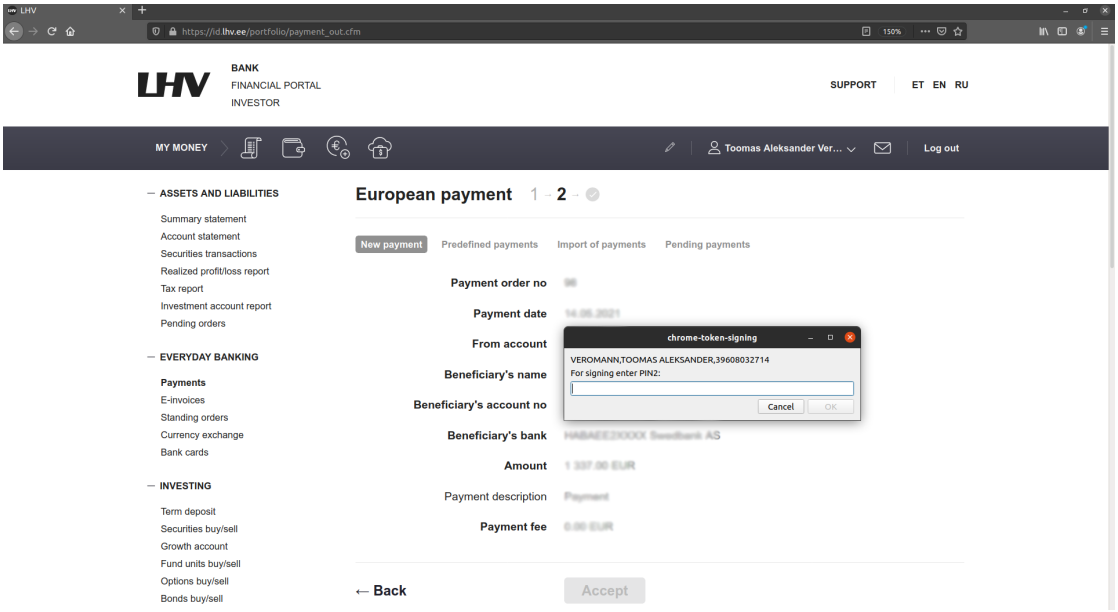


Figure 2. Digital signing process in a web environment using the browser extension

1.1.1 The Most Popular Use Cases

Digital signing in web environments is used by several service providers to allow users to conveniently state their intent, or provide their consent to certain actions. Below, we discuss the most popular use cases where digital signing is used in web environments.

Online Banking. Considering the number of daily transactions, it is likely that online banking is one of the most common use cases for signing using browser extensions [8]. Contracts and transactions are generally confirmed with digital signatures, with the exception of low value payments in certain banks such as LHV [9]. Digitally signed applications for loans such as home-, private- and student loans can also be submitted online.

Even after a user has digitally signed a bank transaction, banks often do not make it possible to download the container, which contains the document that has been signed.

Sometimes, banks allow invoice confirmations, digitally signed by the bank, to be downloaded instead [10].

Rahvaalgatus. Permanent residents of Estonia, who are at least 16 years of age, have the opportunity to digitally sign proposals listed in the Citizen Initiative Portal, `rahvaalgatus.ee`[1], to be sent to the Estonian Parliament (Riigikogu). According to the Riigikogu Rules of Procedure and Internal Rules Act, appeals with at least 1 000 signatures must be processed by the Riigikogu [11]. Since 2014, 16 such proposals have been processed, two of which have been composed into laws.

After signing, a container containing the given digital signature can be downloaded by the signatory. Furthermore, the service provides signatories the option to "revoke" the given signature. This essentially means that the digital signature container, which contains the given signature, is deleted from the service's storage.

Dokobit. One of the most popular online signing services in Estonia is Dokobit[2] - a service that can be used for creating and validating digital signatures [12]. In Dokobit, identity documents used in several countries are supported, including the ID cards of Estonia, Latvia, Lithuania and Finland. Even though signatories have the option to view files directly in the browser before and after signing, signatories still do not have full certainty, whether or not the displayed files are the files that they are actually signing. After signing, signatories can download the relevant digital signature containers.

Webdesktop. Webdesktop by Webware[3] is proprietary software, which uses browser signing for a part of its functionality. Webdesktop's services include e-invoicing and document management and its clients range from Estonian ministries and governmental agencies to financial institutions, private companies and educational institutions. [13]

An example of using Webdesktop is the Document Management Infosystem used by the University of Tartu, where employees can create and sign digital signature containers, register employment contracts, and more [14]. It is likely that other document management software and systems exist that use this method of digital signing.

[1] `https://rahvaalgatus.ee/`
[2] `https://www.dokobit.com`
[3] `http://www.webware.ee/`

1.1.2 Security Issues

The current browser signing architecture requires users to trust service providers, since users are not able to inspect on their computers what they are about to sign. As users can not ascertain the contents of the documents they are in the process of signing, there are several possibilities where users could be deceived into signing something they did not intend to sign.

First of all, if a web service is malicious, it may request the signatory to sign a hash that corresponds to a document which the signatory did not mean to sign. This would be especially difficult for victims to detect later, as the timestamp and Online Certificate Status Protocol (OCSP) data, which are used to provide the long-term validity of signatures, may be obtained at any time after the cryptographic signature has been given [15].

Even if a service provider is not malicious by nature, the service provider's web services may get compromised by malicious entities. The most recent publicly known attack on known Estonian institutions was the Drupal[4] hack of late 2020, where the servers of the Estonian Health Board, Ministry of Economic Affairs and Communications, Ministry of Social Affairs, and Ministry of Foreign Affairs were breached. Even though much is still unknown about the motive of the attacks, it is known that the attackers had access to the Estonian Health and Welfare Information Systems Centre servers for around 8 hours, and much longer to the Ministry of Economic Affairs and Communications systems. If attackers have compromised a web service which uses digital signing, they may have the opportunity to modify this digital signing logic, for example by changing contents of the documents that will be signed. This can cause unsuspecting users to give legally binding digital signatures, signing documents that they did not intend to. [16, 17]

Another threat to users is website spoofing - creating a close resemblance of a reputable website in order to mislead visitors. Untrustworthy websites can, for example, pose as an online shop, requesting users to authenticate themselves or give their signature, while in the background forwarding the requests on the signer's behalf to a completely different, possibly legitimate web service. This is a serious threat, since according to a cyber security study, launched by SEB Estonia in 2020, only around 1 in 10 people could clearly detect a spoofed website [18].

[4] `https://www.drupal.org/`

Such incidents are not only a thing of the past, but have happened very recently and may happen again in the future. As such, there is a substantial need to shift security-critical user actions away from servers to the devices trusted by users whenever possible. Given the online-tendency of Estonian citizens, it is essential for users to be able to ascertain, what documents they are signing, with as much confidence as in real life.

The motivation behind this thesis is to design and implement a more secure browser signing solution, where users can inspect the documents that they are about to sign. Furthermore, this thesis aims to provide a basis for future analysis, discussion and development of WYSIWYS solutions in the browser signing architecture.

1.2 Thesis Structure

The thesis is structured as follows:

- Section 2 provides an overview of digital signature container formats currently used by Estonian ID card software and describes the existing browser signing architecture;
- Section 3 presents and discusses a proof-of-concept WYSIWYS solution in which service providers send plain text data that is displayed to the user before signing;
- Section 4 presents and discusses a proof-of-concept WYSIWYS solution in which service providers send ASICE containers to be signed by users;
- Section 5 presents the concluding statements for this thesis.

2 Background

This section intends to give a brief overview of digital signature container formats currently used by Estonian ID card software and the signature formats which they contain. This section also details the existing browser signing architecture.

2.1 Digital Signature Containers

Digitally signed documents are composed of signed files and corresponding signatures, usually in the form of containers. These containers are essentially ZIP files with another extension, representing the incorporated signature structure. In Estonia, historically, there have been multiple file formats for digital signature containers in use:

- **DDOC** – used until 2014, this format was based on the ETSI TS 101 903 Standard called XML Advanced Electronic Signatures (XAdES), but was not fully compliant with the standard. The DDOC format has since been discontinued [19];
- **BDOC** - also known as ASiC-E LT-TM or BDOC-TM, the BDOC format replaced DDOC, improving security and international compatibility. This format rose from cooperation between Baltic states and implements time marking through the OCSP (RFC 2560) protocol for ensuring the long-term evidential value of digital signatures [20, 21];
- **ASICE** - also known as ASiC-E LT or BDOC-TS, is the current standard for digital signatures used in the European Union. ASICE containers use the RFC 3161 timestamp standard for proving the long-term evidential value of digital signatures. All new digital signatures created using the DigiDoc4 Client use the ASICE format by default. [19]

2.1.1 ASICE Container Format

The most contemporary digital signature containers created by the Estonian ID card software are ASICE containers. The internal structure of an ASICE container, as specified by [15], is as follows:

1. `mimetype` file - a file used to identify the format of the container. This file must

contain the value `application/vnd.etsi.asic-e+zip`;

2. `manifest.xml` file - an XML file containing a list of all the directories and files present in the container, excluding mimetype and signature files. This file must be present in the `META-INF/` directory of the container, as specified by [22];
3. at least one data file (i.e., a file that is signed);
4. one or more `signatures*.xml` files - files containing exactly one XAdES signature each, with signed data files being referenced using `<ds:Reference>` elements. Signature files must be located in the `META-INF/` directory. Additionally, these `signatures*.xml` files must contain timestamp and OCSP data. [15]

An illustration of the ASICE container format and partial contents of the container's elements are given in Figure 3.

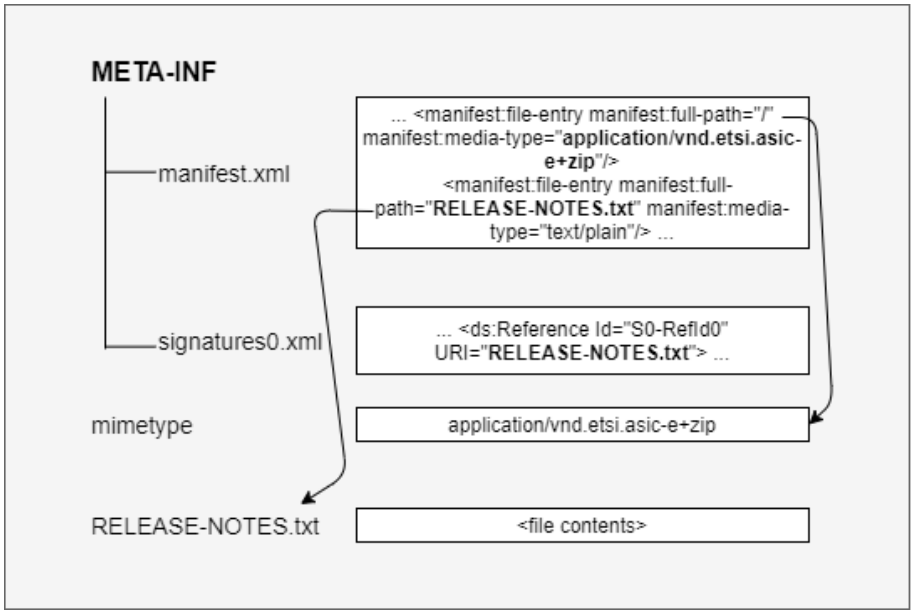


Figure 3. Contents of an ASICE container from [23]

2.1.2 XAdES Signatures

Digital signature containers created by Estonian ID card software contain digital signatures in the XML-based XAdES format. This section provides an overview of the most

important XAdES elements, as well as how XML signatures in general are created and validated. Figure 4 displays the main elements of a XAdES signature, with more detailed explanations of the elements according to [15] and [24] given below.

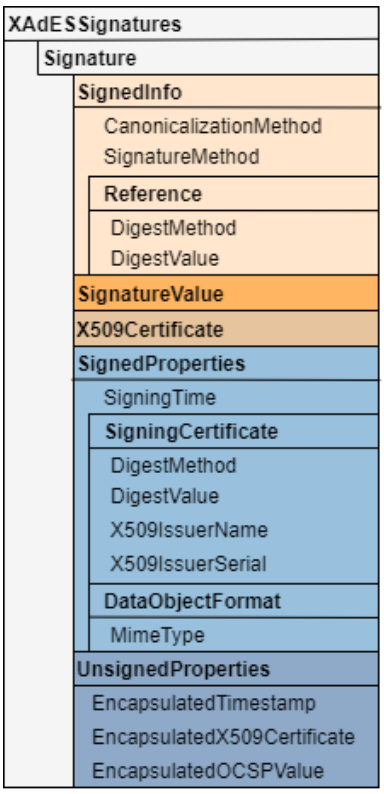


Figure 4. The main components of a XAdES signature. Some elements have been intentionally left out to improve readability.

The XAdES structure contains several elements that are cryptographically signed, as well as various elements that are not signed. The signatory's cryptographic signature value, produced by a signing token such as an ID card, is carried in the `<SignatureValue>` element. This cryptographic signature value must be Base64 encoded before being inserted into the XAdES structure. The signatory's public (X.509) certificate is also added to the XAdES structure into the `<X509Certificate>` element. This public key, contained in the certificate, can be used to validate the given signature.

The actual data that is signed by the signatory is the digest of the contents of the `<SignedInfo>` element. This `<SignedInfo>` element specifies the algorithms that are used in the signing process as `<SignatureMethod>` and `<CanonicalizationMethod>`

elements. The <SignatureMethod> element specifies the signing algorithm used to create the cryptographic signature, whereas the <CanonicalizationMethod> element specifies the algorithm that is used to canonicalize the <SignedInfo> element before signing.

The <SignedInfo> element also references the data that is signed as <Reference> elements. At least two <Reference> elements, specifying the signed resources by URI reference, are present in the <SignedInfo> block - one for each data file that is signed, and one referencing the <SignedProperties> element. These <Reference> elements must contain the following sub-elements:

- the <DigestMethod> element, which specifies the hash algorithm to be applied. The value of this element must be http://www.w3.org/2001/04/xmlenc#sha256;
- the <DigestValue> element contains the Base64 encoded result of applying the hash algorithm specified in <DigestMethod> to the resource specified in <Reference>'s URI.

The <SignedProperties> element also contains several sub-elements, most notably:

- the <SigningTime> element, containing the claimed signing time;
- the <CertDigest> element, specifying the signatory's X.509 certificate digest;
- the <IssuerSerial> element, reflecting the signatory's certificate issuer details;
- the <MimeType> element, containing an indication of the signed data object's MIME type.

Some additional elements, required for long-time signature validation, are not included in the signature itself, but rather in a separate <UnsignedProperties> element:

- the <EncapsulatedTimeStamp> element, containing a Base64 encoded encapsulated timestamp service timestamp response;
- at least two <EncapsulatedX509Certificate> elements, containing Base64 encoded encapsulated X.509 certificates. At least the timestamping service responder certificate and signatory's issuer Certificate Authority (CA) certificate must be

present;

- the <EncapsulatedOCSPValue> element, containing a Base64 encoded encapsulated OCSP service response.

2.1.3 Creating and Validating a XAdES Signature

Creating an XML signature consists of several steps. First, all resources to be signed must be determined and identified through a URI. For each resource to be signed, a digest is calculated. This digest value is inserted as a <DigestValue> child element to the <Reference>, along with a <DigestMethod> element specifying the digest method that was used to create the digest value. A reference to <SignedProperties> must exist in addition to any data file references.

Once all digests are calculated, all <Reference> elements are collected within a <SignedInfo> element along with <CanonicalizationMethod> and <SignatureMethod> values, reflecting the algorithms for canonizing the <SignedInfo> element and producing the signature value, respectively. The signature value is calculated on the digest of the <SignedInfo> element. To allow for signature validation, the signatory's X.509 certificate is embedded as a child of the <KeyInfo> element.

Some unsigned elements are also added to the XML. First, to prove that the signature was created when the signer's certificate was valid, timestamp data is added to <SignatureTimeStamp> element. Then, OCSP data is added to indicate whether the signatory's certificate was valid at the time of the OCSP request. The OCSP data is embedded in the <RevocationValues> element and may contain OCSP responder's certificate. If OCSP responder certificate is missing from the OCSP response, it must be added to the <CertificateValues> element. <CertificateValues> must also contain the signatory's CA certificate. All of these elements are child elements of <UnsignedProperties>.

Validating an XML signature follows a procedure known as core validation. Core validation consists of the following sequential steps:

1. **reference validation** - the digests of each <Reference> in the <SignedInfo> element are verified by retrieving the respective resource, applying the specified

<DigestMethod> to it and comparing the resulting digest value with the embedded <DigestValue>. If the values do not match, reference validation has failed;

2. **signature validation** - the <SignedInfo> element is canonized using the method specified in the <CanonicalizationMethod> element, serialized, and the <SignatureValue> element is verified using the public key extracted from the signatory's certificate specified in <KeyInfo> and the method specified in <SignatureMethod>.

2.2 Current Browser Signing Architecture

Digital signing in web environments through browser extensions requires the use of several components. Figure 5 shows the components used in digital signature creation. A web application, e.g. an internet bank, communicates with browser-specific signing modules through the `hwcrypto.js`[5] JavaScript library. The browser-specific modules, commonly referred to as browser extensions, generally consist of two subcomponents – the browser-side component and the native operating system (i.e., MacOS, Linux, or Windows) component. The browser-side component, often written in native JavaScript, is responsible for data exchange with the native component, which in turn interfaces with the ID card's driver for signature creation.

There are multiple browser-specific modules for different browsers, the most notable being `chrome-token-signing`[6], which handles signing both on the Chrome and Firefox browsers, on Windows, macOS and Linux. In this thesis, `chrome-token-signing` is described, analyzed, and extended. Below, a technical overview of the digital signing process in web environments is given. This process is illustrated by Figure 6.

Communication between web services and the browser extension is handled by the open-source `hwcrypto.js` library. `Hwcrypto.js` is the recommended JavaScript library for interfacing with Estonian browser signing extensions, as it provides several helpful functions used in digital signing. This library contains functions for fetching the software version of the browser extension installed on the signatory's device, retrieving the user's public certificate for signing, and digitally signing a given hash. `Hwcrypto.js`

[5] `https://github.com/hwcrypto/hwcrypto.js`
[6] `https://github.com/open-eid/chrome-token-signing`

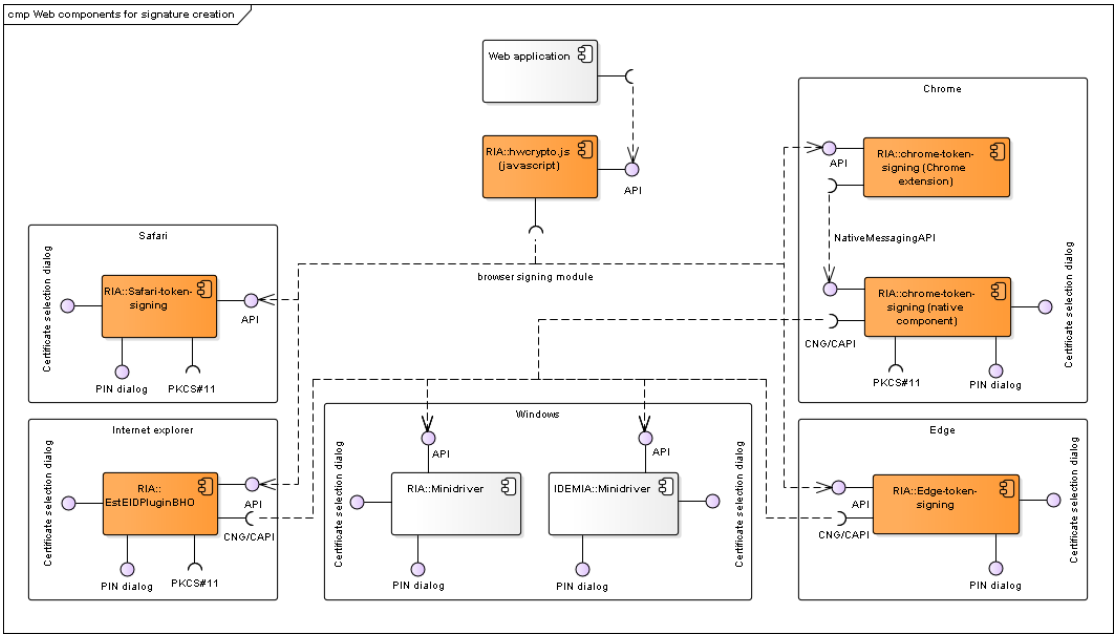


Figure 5. Components needed for creating signatures in web applications [25]

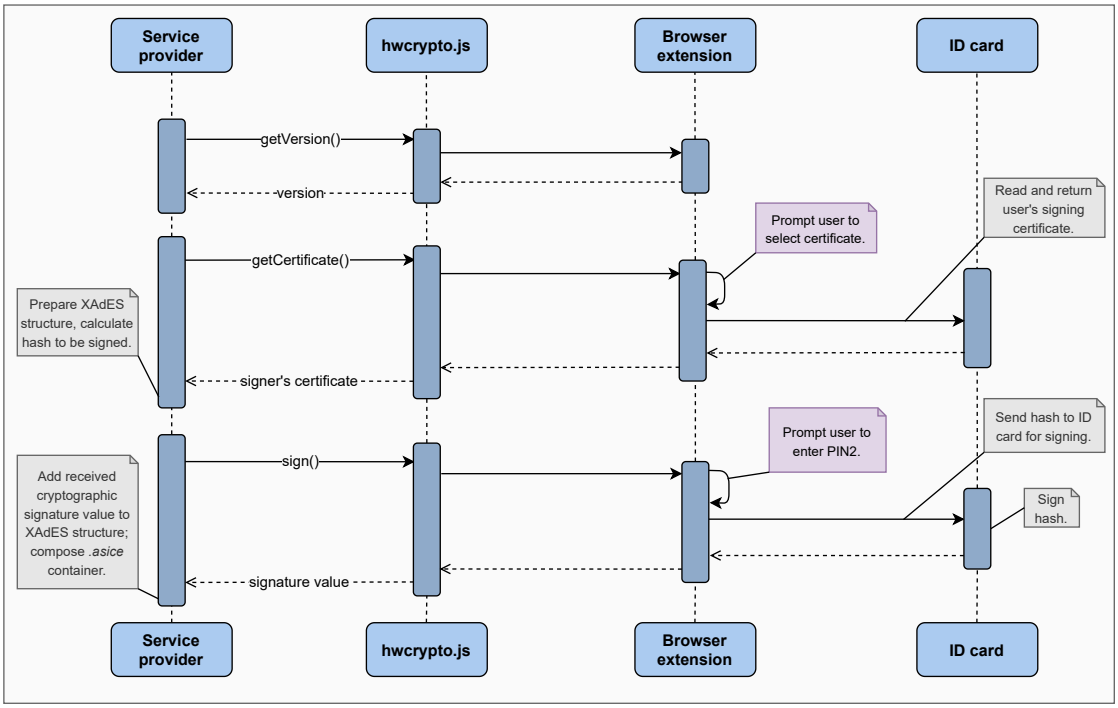


Figure 6. Current signing flow in web environments

API requests and responses are in the JSON data format. The request and response descriptions included in this thesis are based on documentation specified in [26] and were validated via source code given in [27, 28].

Starting the digital signing process. It is good practice for web services to first query the software version of the browser extension installed on the user's device. Over the years, there have been some major API changes for browser extensions. By retrieving the extension's version, service providers can determine whether any changes must be made to the upcoming requests, or whether signing should even be allowed with the given browser extension version. Sometimes, breaking API changes have been made to the browser extension, requiring users to update their ID card software before being able to digitally sign documents again [29]. Services can retrieve the user's browser extension version by using the `hwcrypto.getVersion()` function. This function does not take any parameters. The latest `chrome-token-signing` version as of writing this thesis is 1.1.4.543 [30]. In practice, this request is often not used by service providers.

Requests made through `hwcrypto.js` are first received by the `chrome-token-signing` extension's browser-side JavaScript component. In the browser-side component, three additional fields are set and attached to the request before forwarding the request to the `chrome-token-signing` native component:

1. **`type`** - used to specify the function that was called. Possible values are:
    - "VERSION" for retrieving the `chrome-token-signing` extension version;
    - "CERT" for retrieving the user's public certificates;
    - "SIGN" for signing a given hash.
2. **`nonce`** - an arbitrary alphanumeric string used by the browser-side component to prevent replay attacks by ensuring that previous communications are not reused.
3. **`origin`** - used to define where a given request originated from, and validate that each request within a given session originates from the same location. This `origin` value is saved in the `chrome-token-signing` extension's memory and is compared with the `origin` value from future requests of the same signing session. Essentially, this is the web service's hostname along with the protocol

(e.g., "https://rahvaalgatus.ee"). The protocol must be HTTPS, or else the request is rejected by the browser extension.

Obtaining the user's signing certificate. After fetching the browser extension's version, service providers obtain the signer's X.509 certificate, which must be embedded into the XAdES structure. For retrieving the X.509 certificate, services must use the `hwcrypto.getCertificate()` function. This function prompts the browser extension's native component to ask the user to choose a certificate from a list of smart card certificates currently available to the user's device (see Figure 1). The fields of the request body are detailed in Table 1. An example of the `hwcrypto.getCertificate()` request is given below:

```
{
    "options": {
        "lang": "et"
    }
}
```

Table 1. `Hwcrypto.js getCertificate()` request body fields

Name	Type	Description
`lang`	string	As per documentation, this must be a valid ISO 639-1 language code such as `et` or `en`. [26] In source code, ISO-639-2/B codes are also supported. By default, allowed values are: `et`, `en`, `ru`, `lt`, `lv`, `tr`, `est`, `eng`, `rus`, `lit`, `lat`, `tur`. [27]
`filter`	string	This field was previously used to filter out authentication certificates. Due to a critical bug in the ID card browser extension, this feature was disabled in the browser extension at the end of January 2021, but the field is still present in `hwcrypto.js`. When sending a request with this field containing the value "`AUTH`", the request fails due to the browser extension returning an `invalid_argument` error. [4, 27]

After the user has chosen a certificate, the browser extension returns it to the service provider. Fields of the returned object are specified in Table 2. An example of the `hwcrypto.getCertificate()` response is given below:

```
{
    "certificate": {
        "encoded": [84, 104, 105, 115, 32, 105, ...],
        "hex": "546865207175696636b206272e2..."
    }
}
```

Table 2. `Hwcrypto.js getCertificate()` response

Name	**Type**	**Description**
encoded	Uint8Array	DER-encoded certificate.
hex	string	Certificate encoded as a hex.

Obtaining the user's signature. Once the web service has received the user's X.509 certificate, it prepares a XAdES structure by including the user's certificate hash and data file to be signed under the `<SignedInfo>` element. Then, the service calculates the hash of the `<SignedInfo>` element - the hash, that must be signed by the user. To send the hash to the browser extension, service providers must use the `hwcrypto.sign()` function. This function prompts the browser extension's native component to ask the user to enter their PIN2 (as shown in Figure 2). The fields of the request are detailed in Table 3, and an example of the `hwcrypto.sign()` request is given below:

```
{
    "certificate": "308203E130820343A00...",
    "hash": {
        "type": "SHA-256",
        "value": [84, 104, 105, 115, 32, 105, ...],
        "hex": "40FB337C61E253CBB6B1..."
    }
    "options": {
        "lang": "et"
    }
}
```

Table 3. `Hwcrypto.js sign()` request body fields

Name	**Type**	**Description**
`certificate`	object	Certificate object, retrieved as `hwcrypto.js getCertificate()` response. The related private key must be used for signing.
`hash`	object	The hash to be signed. This object must contain the following values: • `type` - hash type from enum "`SHA-1`", "`SHA-224`", "`SHA-256`", "`SHA-384`" or "`SHA-512`"; • `value` - hash as byte array; • `hex` - hash encoded as hex.
`options`	object	Optional options for the signing process. Possible field values are: • `lang` - language hint, as introduced in Table 1; • `info` - informational message regarding the ongoing signing process. Though this value is present in `hwcrypto.js`, it is not used in the `chrome-token-signing` extension source code and thus, is ignored.

This hash must be signed by the user with the private key corresponding to the certificate that was returned by `hwcrypto.getCertificate()`. As a result of this `hwcrypto.sign()` function call, the browser extension returns an object containing the raw signature data to the service provider. The fields of this object are specified in Table 4, and an example of the `hwcrypto.sign()` response is given below:

```
{
    "signature": {
        "value": [84, 104, 105, 115, 32, 105, ...],
        "hex": "40FB337C61E253CBB6B1..."
    }
}
```

Table 4. `Hwcrypto.js sign()` response

Name	**Type**	**Description**
`value`	Uint8Array	Signature as a byte array.
`hex`	string	Signature encoded as hex.

Composing a digital signature container. After receiving the user's raw cryptographic signature, service providers validate the signature, before adding it to the XAdES structure. Then, service providers request timestamp and OCSP data and include it into the XAdES structure. Finally, service providers compose a digital signature container, containing the XAdES `signatures*.xml` file along with the signed data file(s) and additional metadata files.

As shown, it is apparent that signatories have no way to ascertain the actual contents of the documents that service providers ask them to sign. The next two sections of this work introduce and analyze two WYSIWYS solutions for the current web signing architecture.

3 Solution 1: Signing Plain Text

This section describes one WYSIWYS extension to the current Estonian browser signing architecture - a solution where service providers send plain text data that is displayed to the user before signing.

In the current browser signing solution, service providers must compose a XAdES structure and send the hash value of the `<SignedInfo>` element to the user's browser extension for signing. In the proposed WYSIWYS solution, service providers prepare and send a UTF-8 encoded plain text document instead. At the end of this process, service providers will be able to construct a digital signature container signing a single file containing the text that was displayed to the user. The updated browser signing flow for signing plain texts is illustrated in Figure 7.

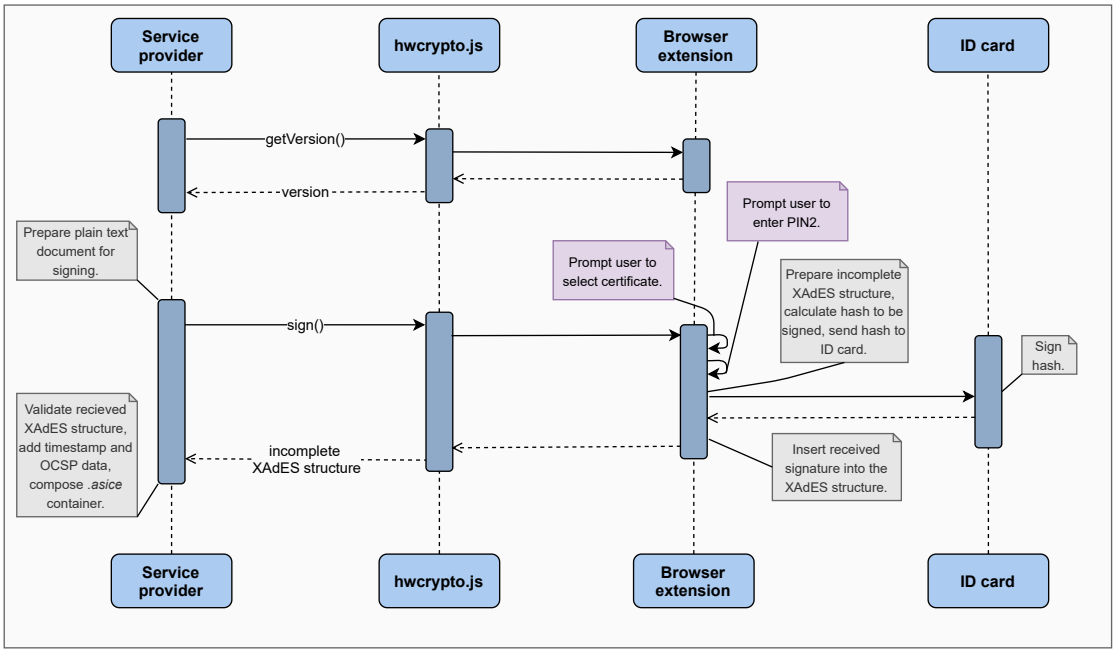


Figure 7. Proposed browser signing flow for signing plain text documents

Starting the digital signing process. As a precondition, it is recommended that service providers query the browser extension version on the user's device. Discrepancies between API versions can cause users to not be able to give digital signatures, especially in the case of breaking changes such as the ones described in this section.

Obtaining the user's signature. In the beginning of the digital signing process, instead of asking for the user's signing certificate, service providers prepare a plain text document that will be sent to the user for signing. This plain text document is Base64 encoded by the web service before being transferred to the browser extension. Transferring this plain text document is done by calling a modified version of the `hwcrypto.sign()` function. Next, the browser extension prompts the user to choose a certificate from a list of smart card certificates currently available to the user's device. This certificate is not sent to the service provider at this stage, but later, along with the signature value. Once the user has selected the certificate, the received plain text is Base64 decoded and displayed to the user by the `chrome-token-signing` native component in the PIN2 entry window (see Figure 8). The length of the text which is immediately visible to the user is around 500 characters, while the maximum length of the plain text document is over 8000 Unicode characters.

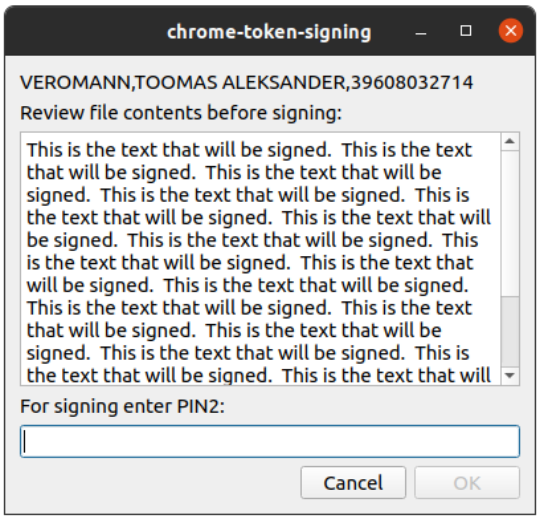


Figure 8. User interface of the proposed WYSIWYS solution

Here, the signer has the opportunity to inspect the document contents before choosing whether or not they want to proceed with signing. In the case of a longer document, the user has the ability to scroll through the entire file contents. If the user wishes to continue with signing, they must enter their signing PIN. Once they have entered their PIN2, the `chrome-token-signing` native component prepares an incomplete XAdES structure which contains a reference to the received plain text document. Once the

XAdES structure has been composed, the chrome-token-signing native component extracts the <SignedInfo> element from the structure, calculates the element's digest value and forwards the hash to the signer's ID card.

Provided that the signatory entered a valid PIN2, a cryptographic signature is created and returned to the chrome-token-signing native component. First, the native component inserts this raw signature value into the <SignatureValue> element. Then, the native component Base64 encodes the incomplete XAdES structure and returns it to the service provider.

Validating the XAdES structure and composing a digital signature container. As creating the initial XAdES structure is now handled by the chrome-token-signing extension, service providers must validate the received incomplete XAdES Structure. In addition to XML Core Validation, service providers must also check whether the received XML includes any forbidden metadata, for example, a resolution stating that the user does not agree to what they have signed. Also, service providers must add timestamp and OCSP data to the XML structure. Once the service provider has validated the XAdES structure, and added timestamp and OCSP data to the structure, the service provider must compose an ASICE container with all the necessary files:

- the original data file, which was sent to the signatory's device;
- the signatures*.xml file, which contains the received XAdES structure and has been updated to include timestamp and OCSP data;
- the manifest.xml file;
- the mimetype file.

As the signature IDs inside the signatures*.xml files created by chrome-token-signing are unique, multiple signatures (from different signatories) that sign the same document can be composed into a single container.

3.1 Modifications to hwcrypto.js

This solution requires some changes to the `hwcrypto.js` API. The modified `hwcrypto.js` source code can be found on Github[7].

Firstly, service providers no longer need to retrieve the user's public certificate prior to signing. This means that the `hwcrypto.getCertificate()` function no longer has to be used. This is because the XAdES structure, returned by the `chrome-token-signing` browser extension after the user has signed the data, will contain the user's signing certificate.

Another modification is to the `hwcrypto.js sign()` function - instead of the `<SignedInfo>` digest, a Base64 encoded plain text file is sent to the user's device instead. An example of the updated `hwcrypto.sign()` request is given below:

```
{
    "file": {
        "base64": "VGhpcyBpcyBzYW1wbGUgcGxhaW4gdGV4dC..."
    },
    "options": {
        "lang": "et"
    }
}
```

The updated `hwcrypto.js sign()` request and response are described in Tables 5 and 6, respectively. An example of the updated `hwcrypto.sign()` response is given below:

```
{
    "signature": "
        PD94bWwgdmVyc2lvbj0iMS4wIiBlbmNvZGluZz0iVVRGLTgiIH..."
}
```

3.2 Modifications to chrome-token-signing

Before, the XAdES structure inside the `signatures*.xml` file had to be created by service providers, but with this solution, a large portion of the XAdES structure creation is

[7] `https://github.com/toomasveromann/hwcrypto.js/tree/thesis-plaintext`

Table 5. Updated `hwcrypto.js sign()` request body fields

Name	**Description**	**Comment**
`file`	The document to be signed. This object must contain the following value: • `contents` - UTF-8 plain text, encoded as Base64.	Added.
`options`	Optional options for the signing process. Possible values are: • `lang` - language hint, as introduced in Table 1; • `info` - informational message regarding the ongoing signing process.	Unchanged.

Table 6. Updated `hwcrypto.js sign()` response

Name	**Description**
`signature`	Incomplete XAdES structure containing the signatory's signature.

shifted to the chrome-token-signing extension. The modified chrome-token-signing source code can be found on Github[8].

With this solution, the chrome-token-signing native component must canonize and calculate the digests for all resources to be signed. Just like with the existing browser signing solution, chrome-token-signing is still responsible for interfacing with the ID card to retrieve a cryptographic signature. In addition to the cryptographic signature, the native component adds additional information such as the claimed signing time, signatory's certificate and certificate issuer information to the XML structure. Once chrome-token-signing has created the XML structure and inserted the cryptographic signature in it, chrome-token-signing serializes and Base64 encodes the structure before returning it to the service provider. An unminified, Base64 decoded version of the incomplete XAdES structure returned to service providers by the chrome-token-signing browser extension is shown in Appendix I. For the purposes of this proof-of-concept solution, the open-source XML parser RapidXML[9] was embedded in chrome-token-signing to create and serialize the XML.

Signed data files must be referenced by a unique URI containing the relative path of the file's location inside the digital signature container structure. As such, chrome-token-signing must specify the file name within the reference URI for each signed plain text document. As the signed document's name is covered by the given signature, a generic "document.txt" is chosen as a default name for plain text documents signed with the updated version of the chrome-token-signing browser extension.

As signature IDs within a container must be unique, random IDs are generated for every signature created with this extension. This allows for multiple signatures*.xml files that sign the same file, into a single container.

3.3 Discussion

This solution is most suitable for use cases, where short texts need to be signed. For example, this may be the case with bank transfers, where both parties' bank accounts, the amount to be transferred, and the date of transfer among other data must be documented

[8] https://github.com/toomasveromann/chrome-token-signing/tree/thesis-plaintext
[9] http://rapidxml.sourceforge.net/

in the signed file.

This approach mostly keeps the existing architecture between `hwcrypto.js` and `chrome-token-signing` intact. For signatories, the only noticeable difference is that the PIN2 entry window displays the text that has to be signed. This means that the signing process remains as convenient as with the existing solution. For service providers, this approach means that they are no longer responsible for generating XAdES structures. Instead, service providers must now validate the entire received XAdES structure. Also, service providers must still request timestamp and OCSP data and compose a valid digital signature container.

Two major limitations exist with this approach. First of all, even though service providers can compose containers with signatures by multiple signatories, only one data file can be signed by a signatory at once. This means that the proposed solution would not be a viable option when multiple files need to be signed at once.

Secondly, this solution only supports displaying plain text. Displaying other common document formats such as PDF or DOCX would require further development on the `chrome-token-signing` native component level, and displaying some unconventional file formats might not be supported at all. Implementing various document format parsers in the `chrome-token-signing` native component is not recommended, as it would bloat the native component.

As a large portion of documents signed online are not plain text files, but rather PDF files, an alternative would be to only embed a PDF parser into `chrome-token-signing`. Another possibility would be for the PIN2 entry dialog to show a link to the document on the user's device. Thus, the document to be signed could be opened by any relevant application the user has installed on their device. This option could be preferred, since it might be better to have separate tools that specialize in displaying different file formats rather than inflate the `chrome-token-signing` extension with this functionality.

4 Solution 2: Signing ASICE Containers

Without the current browser signing solution, users would be required to download digital signature containers from a web service, sign them using the DigiDoc4 Client, and subsequently upload the signed containers back to the web service. The second solution proposed in this work automates these steps, making the process more convenient for signers.

In this solution, the existing digital signing flow is modified, such that ASICE containers are transferred between the service provider and signatory's device. Instead of using the browser extension's native component to create a cryptographic signature, the DigiDoc4 Client is used to add signatures to the received container. The benefit of using the DigiDoc4 Client is that signatures can be added with the ID card, as well as other digital signing methods supported by the DigiDoc4 Client (such as Mobile-ID[10] and Smart-ID[11]). A technical overview of the proposed changes is given below. The updated browser signing flow for signing ASICE containers is illustrated in Figure 9.

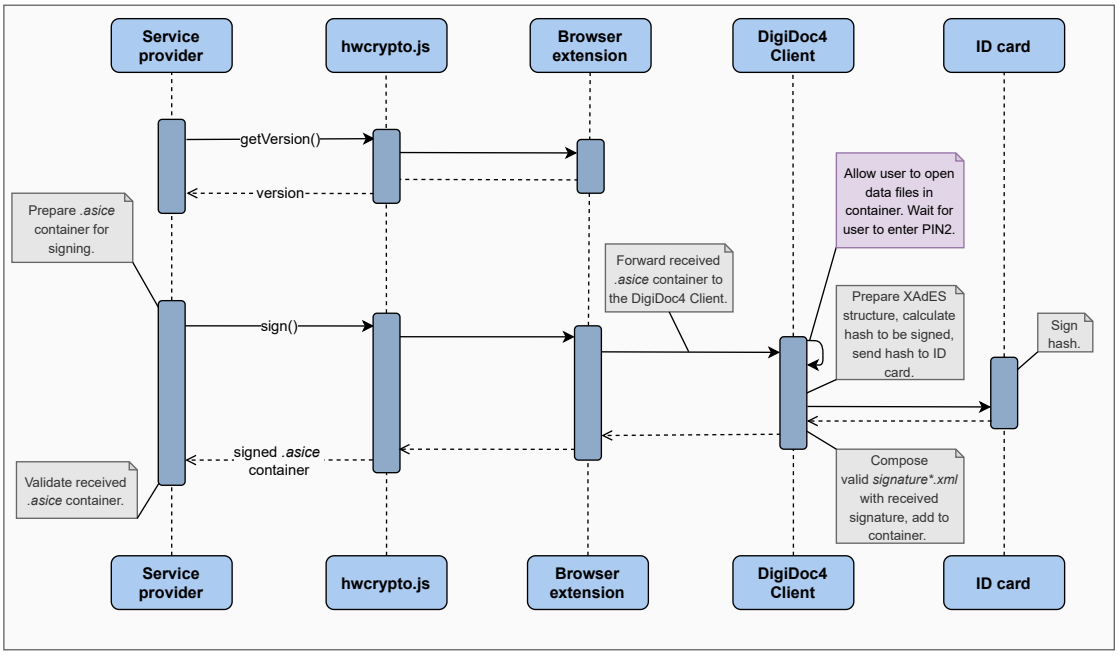


Figure 9. Proposed browser signing flow for signing ASICE containers

[10] https://www.id.ee/en/mobile-id/
[11] https://www.smart-id.com

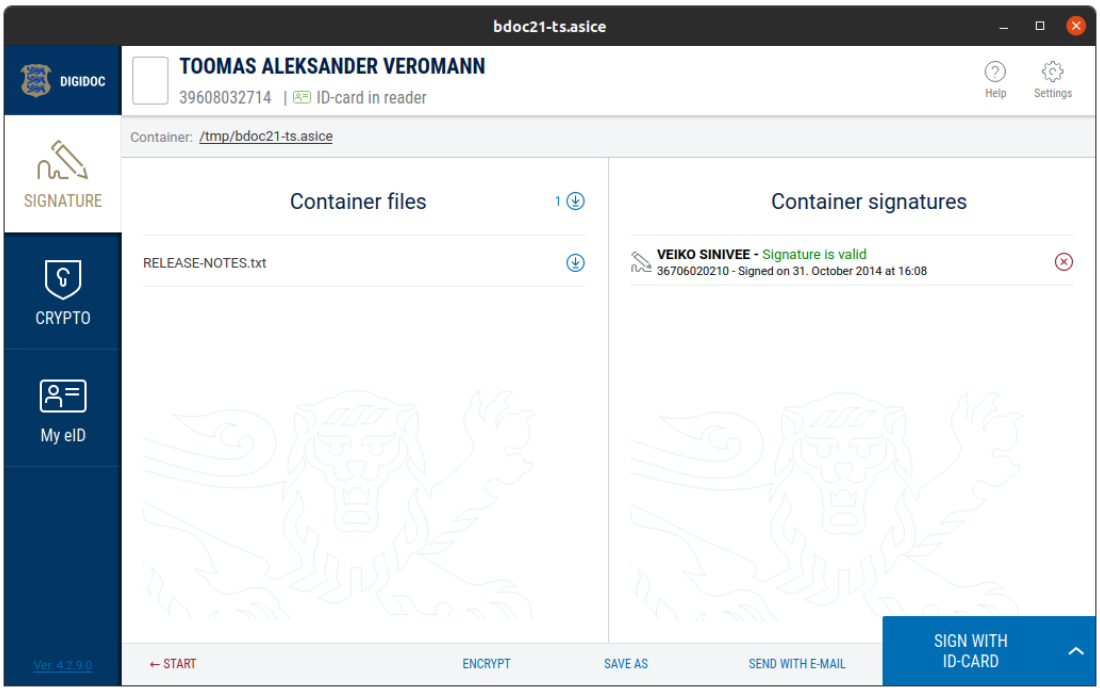
Starting the digital signing process. As with the first proposed solution, it is recommended that service providers first query the browser extension version on the user's device. Discrepancies between API versions can cause users to not be able to give digital signatures, especially in the case of breaking changes such as the ones described in this section.

Obtaining the user's signature. In the beginning of the digital signing process, instead of asking for the user's signing certificate, service providers compose and Base64 encode an ASICE container that will be sent to the user for signing. This container must contain the file(s) that must be signed by the user. The names of these files can depend on the exact use case of the service provider. This container is transferred to the `chrome-token-signing` browser extension by service providers calling a modified version of the `hwcrypto.sign()` function.

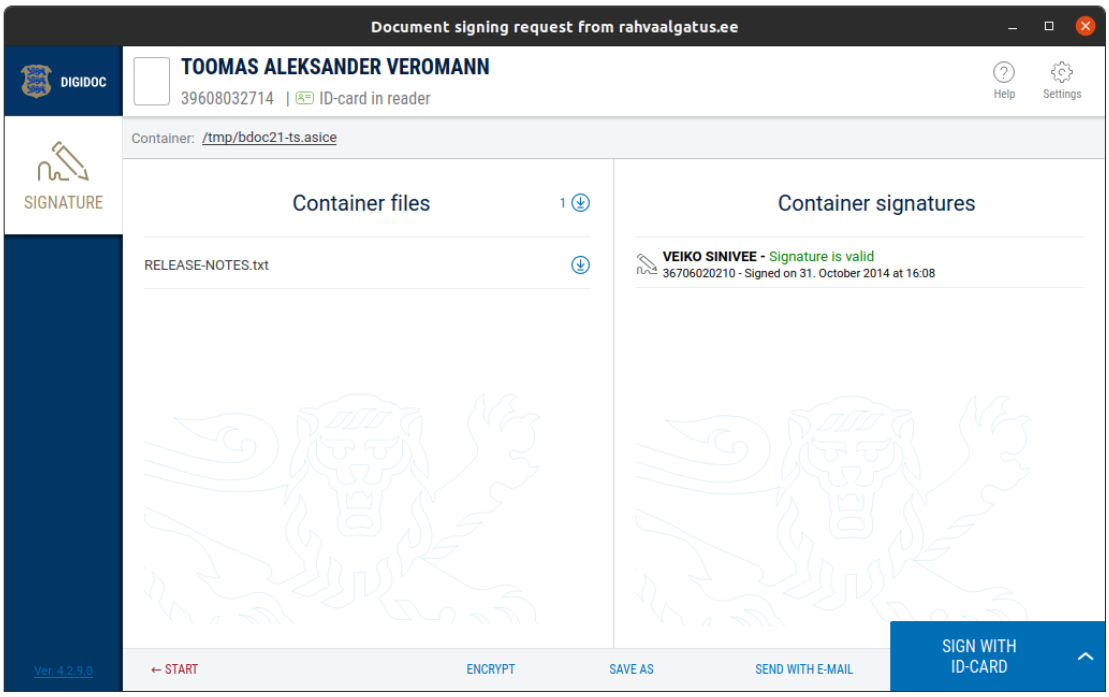
Instead of parsing the received container by itself, `chrome-token-signing` delegates opening the container to a modified version of the DigiDoc4 Client. Figure 10 shows the main windows of the standard and modified versions of the DigiDoc4 Client on the Linux platform before the signatory has added their signature. The key differences are as follows:

- The DigiDoc4 Client window title explicitly states the origin of the signing request (i.e., "Document signing request from `rahvaalgatus.ee`"). This shows signatories which website has initiated the signing request, and where the signed container will be submitted;
- The "Crypto" and "My eID" action menu items have been hidden to simplify the UI and help signatories differentiate between the browser signing-specific DigiDoc4 Client UI and the regular DigiDoc4 Client UI;
- The option to remove existing signatures from the container has been removed (i.e., no signature removal button next to the signatory's name).

At any time before or after giving their signature, signatories have the option to save the container to their device. Signatories also have the option to cancel the signing process by closing the DigiDoc4 Client through the control menu. Similarly to the first solution, once the user has decided to sign the documents, the DigiDoc4 Client prepares



(a) Current UI



(b) Proposed UI for the browser signing flow

Figure 10. The DigiDoc4 Client user interface before signing

a XAdES structure which contains references to the documents included in the received container. Once the XAdES structure has been composed, the DigiDoc4 Client extracts the `<SignedInfo>` element from the structure, calculates the element's digest value and forwards the hash to the signer's ID card.

Provided that the signatory entered a valid PIN2, a cryptographic signature is created and returned to the DigiDoc4 Client. The DigiDoc4 Client inserts this raw signature value into the `<SignatureValue>` element and subsequently requests timestamp and OCSP data before combining all the data into a `signatures*.xml` file.

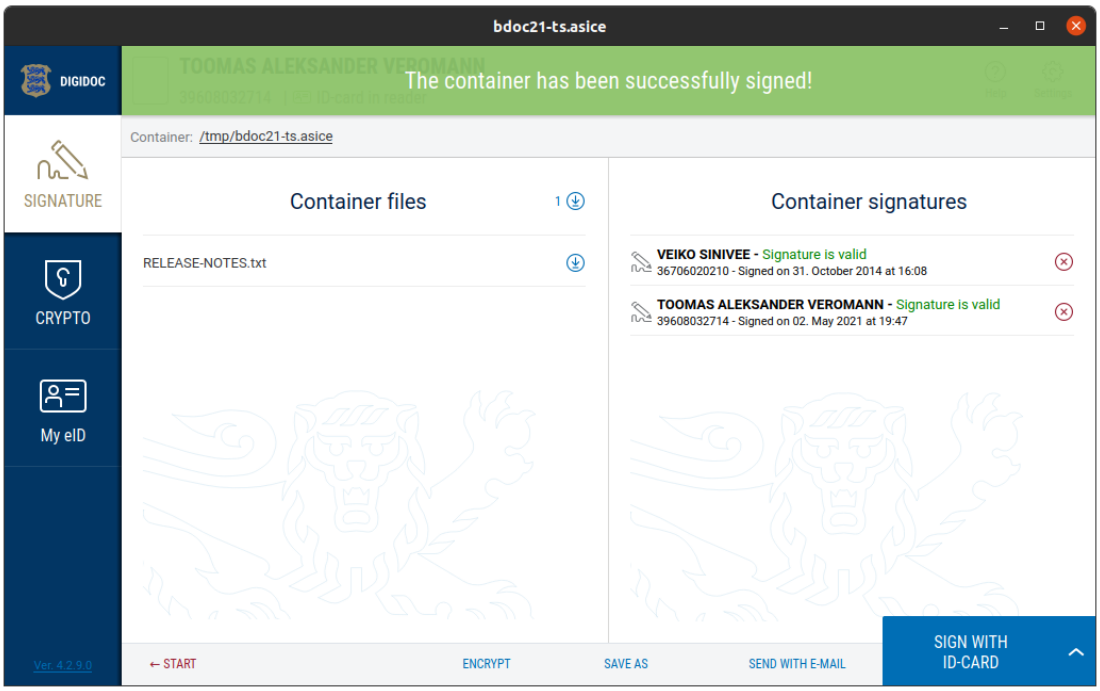
After signing, the signatory is further presented with the option to either cancel the signing process, or return the signed container to the service provider. Optionally, the user may add additional signatures to the container. Figure 11 shows the main windows of the standard and modified versions of the DigiDoc4 Client on the Linux platform after the signatory has added their signature. After the user has chosen to return the container to the service provider, the DigiDoc4 Client Base64 encodes the signed container and returns it to the service provider.

Validating the container. The received container must first be Base64 decoded by the service provider before it can be validated. As ASICE containers are transferred between the service provider and user, service providers need a way of automatically validating containers. Various container creation and validation tools such as the Signature Validation service[12] (SiVa) and software libraries such as libdigidocpp and DigiDoc4J exist, simplifying the validation pipeline for service providers. With this solution, service providers no longer need to handle timestamping and OCSP requests, as these requests are already handled by the DigiDoc4 Client on the client side.

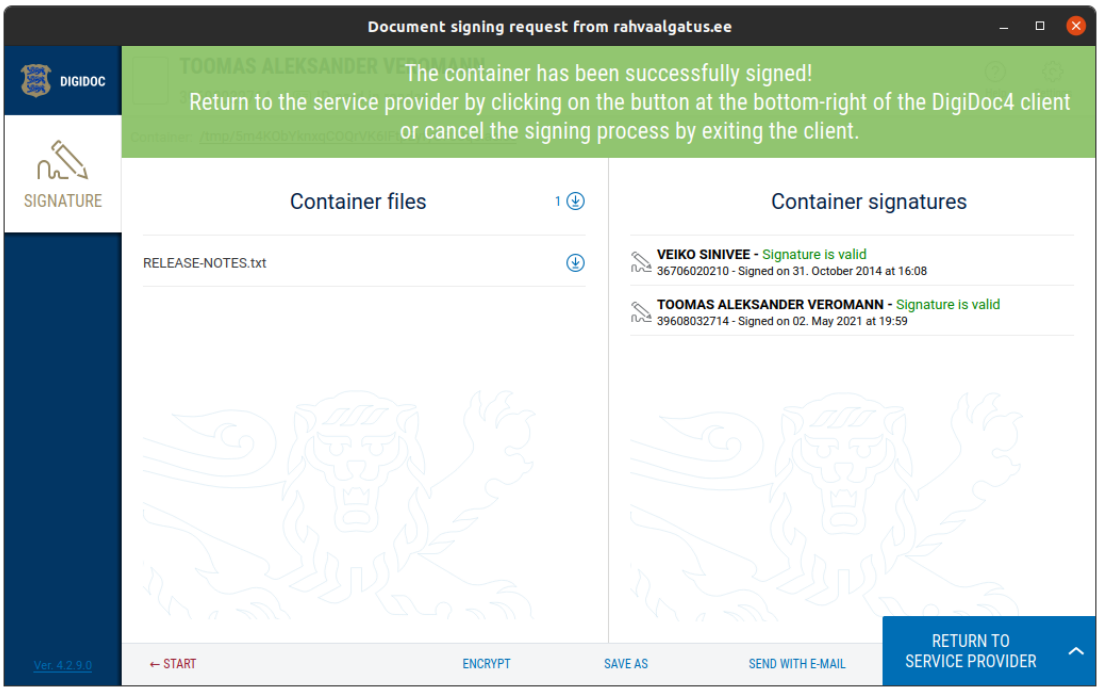
In addition to container validation, there are additional checks that service providers must make. First of all, service providers must verify that the received container contains a signature given by the expected signatory. This is more apparent in web services which require prior authentication, such as online banks, compared to services like Rahvaalgatus, which do not require previous authentication.

Also, since with this solution, signatures can be given with ID cards as well as Mobile-

[12] `https://open-eid.github.io/SiVa/`



(a) Current UI



(b) Proposed UI for browser signing

Figure 11. The DigiDoc4 Client user interface after signing

ID and Smart-ID, service providers might have to differentiate between the electronic identity tools used to give signatures. Some web services might require the highest level of evidentiary value for digital signatures, however some tokens such as Smart-ID Basic do not offer this [31]. As with the first proposed solution, service providers must also ensure that no additional metadata was included in the XAdES signature.

4.1 Modifications to hwcrypto.js

This container signing solution requires some changes to the `hwcrypto.js` API. The modified source code for `hwcrypto.js` can be found on Github[13].

As the signatory's X.509 certificate is embedded in the signature which is included in the container, there is no need to ask for the signatory's certificate beforehand. This means that `hwcrypto.js getCertificate()` is no longer required to be used.

Another modification is to the `hwcrypto.js sign()` function - now, ASICE containers are transferred between the web service and the browser extension. An example of the updated `hwcrypto.sign()` request is given below:

```
{
    "container": {
        "base64": "UEsDBBQAAAgAAFtwX0WKIflFHwAAAB8AAAAIAAAAb..."
    },
    "options": {
        "lang": "et"
    }
}
```

Tables 7 and 8 highlight changes to the `hwcrypto.js sign()` request and response, respectively. An example of the updated `hwcrypto.sign()` response is given below:

```
{
    "container": "UEsDBBQAAAgAAFtwX0WKIflFHwAAAB8AAAAIAAAAb..."
}
```

[13] `https://github.com/toomasveromann/hwcrypto.js/tree/thesis-asice`

Table 7. Updated `hwcrypto.sign()` request body fields

Name	Description	Comment
`container`	The container, whose contents are to be signed. This JSON object must contain the following value: • `base64` - Base64 encoded container bytes.	Added.
`options`	Optional options for the signing process. Possible values are: • `lang` - language hint, as introduced in Table 1; • `info` - informational message regarding the ongoing signing process.	Unchanged.

Table 8. Updated `hwcrypto.js sign()` response

Name	Description
`container`	ASiC-E container with added signatory's signature

4.2 Modifications to chrome-token-signing

In contrast to the existing browser signing solution, where `chrome-token-signing` is responsible for most of the signing-related actions, in this solution, the responsibility of signing is shifted from the `chrome-token-signing` native component to the DigiDoc4 Client. Here, `chrome-token-signing` acts as a proxy between the service provider and the DigiDoc4 Client on the signatory's device. The modified source code for `chrome-token-signing` can be found on Github[14]

Once `chrome-token-signing` receives the signing request and finishes request validations, the browser extension decodes and saves the received container under a randomly generated name (e.g., "`container-atFLetThGcRE7RqLhR8H07VP.asice`") to the signatory's device's temporary data folder (e.g., "`/tmp/`" on Linux). Once the container has been saved, `chrome-token-signing` opens the modified DigiDoc4 Client in the browser signing-specific mode and stays on stand-by, listening for an exit code of the DigiDoc4 Client. Depending on the exit code, `chrome-token-signing` determines whether the

[14] `https://github.com/toomasveromann/chrome-token-signing/tree/thesis-asice`

signing process was completed successfully or was cancelled by the signatory, and returns either the signed container or the respective error response back to the web service.

4.3 Modifications to the DigiDoc4 Client

To allow the DigiDoc4 Client to function in a way necessary for the browser signing flow, support for additional command line parameters was added. Also, functionality irrelevant to signing in web environments was hidden in the browser signing-specific UI of the DigiDoc4 Client. The modified source code for the DigiDoc4 Client can be found on Github[15].

The DigiDoc4 Client allows the path to a container's location to be passed to the application. In addition to this, changes were made to allow opening the DigiDoc4 Client in the browser signing-specific UI mode, as well as allowing to specify the signature request origin. Likewise, additional exit codes were defined for communicating user intent between the DigiDoc4 Client and `chrome-token-signing`.

The complete syntax for executing the DigiDoc4 Client in the browser signing-specific mode is:

```
qdigidoc4 -sign-only <path-to-container> -source <origin>
```

where:

- the `-sign-only` flag indicates, that the DigiDoc4 Client should open in the modified browser signing view;
- `<path-to-container>` is a valid path to a container on the signatory's device's file system;
- `-source <origin>` indicates the origin URL of the incoming signature request.

4.4 Discussion

The DigiDoc4 Client is a well-known, trusted and stable application, with a familiar user interface. Using the DigiDoc4 client to digitally sign documents in web environments

[15] `https://github.com/toomasveromann/DigiDoc4-Client/tree/thesis-asice`

has several benefits. Firstly, incorporating the DigiDoc4 Client into signing in web environments means that migrating from the current browser signing solution to this proposed one would not be difficult, as the DigiDoc4 Client user interface is already familiar for most users. Another major benefit to using the DigiDoc4 Client is that it allows users to save containers on their device, enabling signatories to retain possession over the containers which they have signed.

A further benefit of this solution is the possibility to integrate it with other nations' electronic signature software. It might not require extensive development, as changes to `chrome-token-signing` will be limited to validating that given software exists and opening the software on the user's device. From the software side, only a couple of modifications to these clients are necessary:

- additional command-line arguments for specifying the container to be signed;
- optional UI changes to differentiate between browser signing views and offline signing views;
- a means of communicating successful and unsuccessful signing completion to the browser extension.

It must be noted, that as the DigiDoc4 Client doesn't automatically render documents included in the opened container, there is still the possibility that users do not inspect the container contents before giving their signature. A possible solution may be for the DigiDoc4 Client to automatically try to render the data files inside the container. One such example, where this has been implemented is eParakstītājs[16], the official Latvian software for creating and signing electronic documents (see Figure 12). However, an unwanted side effect may arise when attempting to display documents with proprietary file formats (e.g., `.doc`) - the content of the documents may be shown incorrectly. This issue is less apparent with open-source formats such as `.pdf`.

With this solution, the complexity regarding the creation of XAdES signatures and ASICE containers is shifted away from the service providers to the DigiDoc4 Client, which already contains all the necessary logic. This increases the ease of adoption for service providers, because now service providers must only handle validating the

[16] `https://www.eparaksts.lv`

Figure 12. The eParakstītājs 3.0 UI with a rendered document [32]

received containers and the signatures which they contain.

An important note to emphasize is that using the DigiDoc4 Client for signing also solves the problem of blind signing that is present when using Mobile-ID and Smart-ID to creating digital signatures in web environments. This is because now, instead of service providers, the DigiDoc4 Client would be responsible for creating the hash sent to the end user's device.

For web services, migrating to this solution would mean that service agreements with timestamping authorities would no longer be required due to timestamp and OCSP requests being done in the DigiDoc4 Client. On the other hand, this may cause problems for users signing large amounts of documents, since so-called regular users of the DigiDoc4 Client are allocated an IP-based fixed number of free timestamp requests every month. The exact number is not reflected in the DigiDoc4 Client documentation. Further analysis is needed to determine, whether this issue is significant and whether these monthly limits must be rethought by RIA.

Of course, it must also be noted, that with this approach, entire containers (and not just

short hash values) are transferred back and forth. This might be an issue if the signatory has a slow internet connection and the files to be signed are huge. Designing a secure, yet user friendly WYSIWYS solution is an intricate task, requiring meaningful design choices. What must be kept in mind, is that good user experience is achieved not by controlling what users can or can not do, but rather by helping and influencing them with good UI choices.

5 Conclusions

There is a security issue within the current Estonian browser signing architecture - users can not be certain of the contents of the documents that they are asked to sign by web services. This thesis provided proof-of-concept implementations of two WYSIWYS solutions to address this issue and propose future improvements.

The first solution of plain text signing is most suitable when signing shorter texts that can be fully displayed to the signer. Compared to the existing browser signing solution, the user experience is essentially the same, with the only noticeable change being the plain text document shown in the PIN2 entry dialog. This solution is best suited when users need to confirm simpler actions, such as bank transactions.

The second solution of ASICE container signing automates the process of manually downloading and uploading containers to service providers. The benefit to this approach is that any files using any electronic identity tool (including Mobile-ID and Smart-ID) can be signed using a signing process that is familiar and well known to users. However, to fully benefit from the WYSIWYS feature, users have to open and inspect the files before adding their signature. Hence, this solution is more suited for signing standard contracts that require additional precaution from the signatory. This signing solution provides more flexibility for both service providers and signatories, and therefore can cover more use cases than the proposed plain text signing solution. Furthermore, this solution also has the potential to solve the problem of blind signatures that are given using Mobile-ID and Smart-ID in web environments.

Both of these proposals were demonstrated to representatives of RIA on April 29, 2021. We hope that in one way or another, the problem of users having to give blind signatures will eventually be solved.

References

[1] e-estonia, "Estonia introduced a new ID card." https://e-estonia.com/estonia-introduced-a-new-id-card/. (2021.04.22).

[2] Forbes, "The Tiny European Country That Became A Global Leader In Digital Government." https://www.forbes.com/sites/delltechnologies/2016/06/14/the-tiny-european-country-that-became-a-global-leader-in-digital-government. (2021.04.22).

[3] Financial Times, "Estonia leads world in making digital voting a reality." https://www.ft.com/content/b4425338-6207-49a0-bbfb-6ae5460fc1c1. (2021.04.22).

[4] Estonian Information System Authority, "The Information System Authority (RIA) and its partners fixed a critical bug in the ID-card browser extension." https://www.ria.ee/en/news/information-system-authority-ria-and-its-partners-fixed-critical-bug-id-card-browser-extension.html. (2021.04.22).

[5] BBC, "Security flaw forces Estonia ID 'lockdown'." https://www.bbc.com/news/technology-41858583. (2021.04.22).

[6] The Economist, "Estonia takes the plunge." https://www.economist.com/international/2014/06/28/estonia-takes-the-plunge. (2021.04.22).

[7] Open Electronic Identity, "Architecture of ID-software. Signing in web browser. ." https://open-eid.github.io/#signing-in-web-browser. (2020.10.18).

[8] European Commission, "Digital Economy and Society Index (DESI) 2020 - Estonia." https://digital-strategy.ec.europa.eu/en/policies/desi-estonia. (2021.05.14).

[9] LHV, "LHV's low value payment saves you the trouble of entering your PIN." https://www.lhv.ee/en/news/2018/28. (2021.02.21).

[10] LHV, "How does one send a bank statement?." https://www.lhv.ee/en/faq/bank-statement. (2021.05.07).

[11] Riigi Teataja, "Riigikogu Rules of Procedure and Internal Rules Act." https://www.riigiteataja.ee/en/eli/518112014003/consolide. (2021.02.07).

[12] Estonian Information System Authority, "Developer-friendly electronic authentication and signing solutions by Dokobit." https://www.id.ee/en/article/developer-friendly-electronic-authentication-and-signing-solutions-by-dokobit/. (2021.02.07).

[13] Webware, "Clients | Webdesktop information management." https://webware.ee/en/clients. (2021.04.08).

[14] University of Tartu, "Document Management (in Estonian)." https://wiki.ut.ee/display/DHIS/Dokumendihaldus. (2021.04.10).

[15] Estonian Information System Authority, "BDOC – FORMAT FOR DIGITAL SIGNATURES." https://www.id.ee/public/bdoc-spec212-eng.pdf. (2021.02.07).

[16] Drupal, "Security advisories." https://www.drupal.org/security/. (2021.02.12).

[17] Geenius, "Estonian Information System Authority: Estonian Ministry of Economic Affairs and Communications did not use the best practices of information security, monitoring procedure initiated (in Estonian)." https://digi.geenius.ee/rubriik/uudis/ria-mkm-ei-kasutanud-infoturbe-osas-parimaid-praktikaid-algatasime-jarelevalvemenetluse/. (2021.02.12).

[18] SEB, "Vaid umbes 10% eestimaalastest tunneb ära võltsitud veebilehe." https://www.seb.ee/uudised/2020-11-23/vaid-umbes-10-eestimaalastest-tunneb-ara-voltsitud-veebilehe. (2021.02.12).

[19] Estonian Information System Authority, "What's the difference between the digital signature formats .ddoc, .bdoc and .asice?." https://www.id.ee/en/article/whats-the-difference-between-the-digital-signature-formats-ddoc-bdoc-and-asice/. (2020.11.22).

[20] Estonian Information System Authority, "Estonian Information System Authority blog. The king is dead, long live the king!* (in Estonian)." https://blog.ria.ee/elagukuningas/. (2020.10.18).

[21] Estonian Information System Authority, "BDOC, CDOC and ASICE DigiDoc file formats." https://www.id.ee/en/article/bdoc-cdoc-and-asice-digidoc-file-formats/. (2020.12.16).

[22] OASIS Open, "Open Document Format for Office Applications (OpenDocument) Version 1.3. Part 2: Packages." https://docs.oasis-open.org/office/OpenDocument/v1.3/cs02/part2-packages/OpenDocument-v1.3-cs02-part2-packages.pdf. (2021.02.07).

[23] Estonian Information System Authority, "DigiDoc container format life cycle." https://www.id.ee/en/article/digidoc-container-format-life-cycle-2/. (2021.02.07).

[24] W3C, "XML Advanced Electronic Signatures (XAdES)." https://www.w3.org/TR/XAdES/. (2021.05.02).

[25] Open Electronic Identity, "Architecture of ID-software. Web components.." https://open-eid.github.io/#web-components. (2020.10.18).

[26] hwcrypto, "API." https://github.com/hwcrypto/hwcrypto.js/wiki/API. (2021.02.15).

[27] Open-EID, "open-eid/chrome-token-signing : Chrome and Firefox extension for signing with your eID on the web - READ WIKI!." https://github.com/open-eid/chrome-token-signing. (2021.02.15).

[28] hwcrypto, "hwcrypto/hwcrypto.js : Browser JavaScript library for working with hardware tokens." https://github.com/hwcrypto/hwcrypto.js. (2021.02.15).

[29] Estonian Information System Authority, "Why is it necessary to update the ID-software?." https://www.id.ee/en/article/why-is-it-necessary-to-update-the-digidoc4-software/. (2021.05.13).

[30] Open EID, "Releases." https://github.com/open-eid/chrome-token-signing/releases. (2021.05.08).

[31] SK ID Solutions, “Upgrading your Smart-ID Basic account.” https://www.smart-id.com/help/faq/update-your-smart-id/upgrading-your-smart-id-basic-account/. (2021.05.13).

[32] FileInfo, “.EDOC File Extension.” https://fileinfo.com/extension/edoc. (2021.04.30).

Appendix

I. Incomplete XAdES Structure

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<asic:XAdESSignatures xmlns:asic="http://uri.etsi.org/02918/v1.2.1#" xmlns:ds="http
    ://www.w3.org/2000/09/xmldsig#" xmlns:xades="http://uri.etsi.org/01903/v1.3.2#">
  <ds:Signature Id="yy4cBWDxS22JjzhMaiRrV41m">
    <ds:SignedInfo>
      <ds:CanonicalizationMethod Algorithm="http://www.w3.org/2006/12/xml-c14n11"></
          ds:CanonicalizationMethod>
      <ds:SignatureMethod Algorithm="http://www.w3.org/2001/04/xmldsig-more#ecdsa-
          sha256"></ds:SignatureMethod>
      <ds:Reference Id="yy4cBWDxS22JjzhMaiRrV41m-RefId0" URI="document.txt">
        <ds:DigestMethod Algorithm="http://www.w3.org/2001/04/xmlenc#sha256"></ds:
            DigestMethod>
        <ds:DigestValue>V+pNwbCPfp0hNiyho0Pnbxn5A+sEFh7DbOX51r+l3bM=</ds:DigestValue
            >
      </ds:Reference>
      <ds:Reference Id="yy4cBWDxS22JjzhMaiRrV41m-RefId1" Type="http://uri.etsi.org
          /01903#SignedProperties" URI="#yy4cBWDxS22JjzhMaiRrV41m-SignedProperties">
        <ds:DigestMethod Algorithm="http://www.w3.org/2001/04/xmlenc#sha256"></ds:
            DigestMethod>
        <ds:DigestValue>XJ1t4ZkMSlpAwhbhHCNu4MHEb/6nUfGacXoM5irG0wM=</ds:DigestValue
            >
      </ds:Reference>
    </ds:SignedInfo>
    <ds:SignatureValue Id="yy4cBWDxS22JjzhMaiRrV41m-SIG">
      1IczLmWSkawx+tj+NJSNNxQpKG96TGQ5gAz5GrKLT3ywnR
      MGfGE67CzicIfB1JgoQtR0UJ+Nnexpl7yxSukT
      HEWNp57odNts9s2R3jZ6QjiQgOD6aH2BaY2qMxXBRFa9</ds:SignatureValue>
    <ds:KeyInfo>
      <ds:X509Data>
        <ds:X509Certificate>
          MIID4TCCA0OgAwIBAgIQQ2ewGXQm3/ZcwD6dAz7uEzAKBgg...
        </ds:X509Certificate>
      </ds:X509Data>
    </ds:KeyInfo>
    <ds:Object>
      <xades:QualifyingProperties Target="#yy4cBWDxS22JjzhMaiRrV41m">
        <xades:SignedProperties Id="yy4cBWDxS22JjzhMaiRrV41m-SignedProperties">
          <xades:SignedSignatureProperties>
            <xades:SigningTime>2021-05-08T13:00:54Z</xades:SigningTime>
            <xades:SigningCertificate>
              <xades:Cert>
                <xades:CertDigest>
                  <ds:DigestMethod Algorithm="http://www.w3.org/2001/04/xmlenc#
                      sha256"></ds:DigestMethod>
                  <ds:DigestValue>t8XmJFB7yzK0grbu9h99AfFqdNoCRB5JBRbrUOr3pRc=</ds:
                      DigestValue>
                </xades:CertDigest>
```

```
38                <xades:IssuerSerial>
39                  <ds:X509IssuerName>CN=ESTEID2018,organizationIdentifier=NTREE
                        -10747013,O=SK ID Solutions AS,C=EE</ds:X509IssuerName>
40                  <ds:X509SerialNumber>89596654014760014233626285350766702099</ds:
                        X509SerialNumber>
41                </xades:IssuerSerial>
42              </xades:Cert>
43            </xades:SigningCertificate>
44          </xades:SignedSignatureProperties>
45          <xades:SignedDataObjectProperties>
46            <xades:DataObjectFormat ObjectReference="#yy4cBWDxS22JjzhMaiRrV41m-
                  RefId0">
47              <xades:MimeType>text/plain</xades:MimeType>
48            </xades:DataObjectFormat>
49          </xades:SignedDataObjectProperties>
50        </xades:SignedProperties>
51      </xades:QualifyingProperties>
52    </ds:Object>
53  </ds:Signature>
54 </asic:XAdESSignatures>
```

II. Licence

Non-exclusive licence to reproduce thesis and make thesis public

I, **Toomas Aleksander Veromann**,
(author's name)

1. herewith grant the University of Tartu a free permit (non-exclusive licence) to

   reproduce, for the purpose of preservation, including for adding to the DSpace digital archives until the expiry of the term of copyright,

   WYSIWYS Extensions to the Estonian ID Card Browser Signing Architecture,
   (title of thesis)

   supervised by Arnis Paršovs.
   (supervisor's name)

2. I grant the University of Tartu a permit to make the work specified in p. 1 available to the public via the web environment of the University of Tartu, including via the DSpace digital archives, under the Creative Commons licence CC BY NC ND 3.0, which allows, by giving appropriate credit to the author, to reproduce, distribute the work and communicate it to the public, and prohibits the creation of derivative works and any commercial use of the work until the expiry of the term of copyright.

3. I am aware of the fact that the author retains the rights specified in p. 1 and 2.

4. I certify that granting the non-exclusive licence does not infringe other persons' intellectual property rights or rights arising from the personal data protection legislation.

Toomas Aleksander Veromann
14/05/2021