UNIVERSITY OF TARTU
Institute of Computer Science
Computer Science Curriculum

Artur Aralov

Database that stores data contained in digital signature file formats used in Estonia

Bachelor's Thesis (9 ECTS)

Supervisor: Arnis Paršovs, MSc

Tartu 2020

Andmebaas, millesse on salvestatud Eestis kasutatavates digitaalallkirja failivormingutes sisalduvad andmed

Lühikokkuvõte:

Bakalaureusetöös kirjeldatakse digitaalallkirja failivormingut ja sellega seotud termineid. Lisaks kirjeldatakse Eestis kasutatavate digitaalallkirja failivormingute spetsifikatsioone. Selle lõputöö eesmärk oli luua andmebaas, kuhu salvestatakse Eestis kasutatavates digitaalallkirja failivormingutes sisalduvad andmed. Seetõttu kirjeldab see töö ka nende digitaalallkirja failivormingute kogumise ja andmebaasi loomise protsesse.

Võtmesõnad:

Digitaalallkirja failivorming, failivorming, digitaalallkiri, avaliku võtme ser6t5tifikaat, veebisertifikaadi oleku protokoll, ajatempli protokoll

CERCS:

P175 Informaatika, süsteemiteooria

Database that stores data contained in digital signature file formats used in Estonia

Abstract:

The bachelor's thesis describes what digital signature file format is, and terms associated with it. In addition, specifications of digital signature file formats used in Estonia are described. The goal of this thesis was to create a database in which data contained in digital signature file formats used in Estonia will be stored. Therefore, this thesis also describes processes of collecting these digital signature file formats and creating a database.

Keywords:

Digital signature file format, file format, digital signature, public key certificate, Online Certificate Status Protocol, Time-Stamp Protocol

CERCS:

P175 Informatics, systems theory

Table of Contents

Introduction 5
1 Background information 7
1.1 Digital signature file format and related terms 7
1.1.1 Digital signature 7
1.1.2 Public key certificate 8
1.1.3 Online Certificate Status Protocol 10
1.1.4 Time-Stamp Protocol 11
1.2 Digital signature file formats specification 13
1.2.1 DDOC file format 13
1.2.2 BDOC file format 13
2 Data collection 16
2.1 Document register 16
2.2 Implementation details 17
2.2.1 Selenium WebDriver 17
2.2.2 Script 19
2.2.3 JUnit 23
2.2.4 Unit test case 24
2.3 Data collection 25
3 Database creation 28
3.1 Database management system 28
3.2 Database table structure 29
3.3 Technologies 31
3.3.1 DigiDoc4j 31
3.3.2 Java Database Connectivity 32
3.4 Implementation details 34
3.4.1 Program command line arguments 35
3.4.2 Static fields 35
3.4.3 Signatures handling 36
3.4.4 Signer's data extraction 36
3.4.5 Signer's certificate data extraction 38

3.4.6 OCSP data extraction 40
3.4.7 Timestamp data extraction 42
3.4.8 Interaction with the database 44
4 Outcome 47
5 Conclusion 48
References 49

Introduction

Today, the digital signing of documents is commonplace. In Estonia, to digitally sign a document, DigiDoc[1] software is mainly used. Using this software, it is possible to save a document in a digital signature file format, sign a document, encrypt a document, and verify the signature of the document [1]. There are also other ways to sign a document digitally. For example, it is possible to sign a document by using the Estonian State Portal[2] or Dokobit portal[3].

After the document is digitally signed, it is saved in a new format, which is a digital signature file format. Two digital signature file formats are used in Estonia, namely DDOC and BDOC. Each of these file formats has a unique structure. Even though the structure of these file formats is different, they contain similar data.

The goal of this thesis was to create a database in which data contained in DDOC and BDOC digital signature file formats will be stored. To do this, it was first necessary to collect these file formats. Then, it was necessary to extract data from collected file formats and insert it into a database.

The database is created for people who are interested in digital signatures or digital signature file formats used in Estonia. With the help of the created database, it is convenient to analyze the data contained in DDOC and BDOC file formats because it stores data in a structured format, using rows and columns. This makes it easy to locate and access specific data within the database. For example, database queries[4] can be used to reduce the time to search for data of interest. Also, new data can be inserted into the database, and the existing one can be modified. An indisputable fact is that most of the data contained in DDOC and BDOC file formats is encrypted, so it cannot be analyzed without a particular program. However, in the created database, all data is stored in decrypted form, so it is convenient to analyze it. In the

[1] https://installer.id.ee/

[2] https://www.eesti.ee/en/

[3] https://www.dokobit.com/et/

[4] https://study.com/academy/lesson/database-query-definition-tools.html

future, the database can be used to conduct statistical analysis or detect some digital signature file format non-compliances.

The work is divided into five chapters. The first chapter provides the reader with background information on which the rest of the work is based. The second chapter describes the process of collecting data (digital signature file formats used in Estonia). The third chapter describes the process of creating a database. The fourth chapter describes what has been done as an outcome of this thesis. The fifth chapter, which is the last, describes the conclusion of this thesis.

1 Background information

The purpose of this chapter is to provide the reader with background information on which the practical part of the work is based. It first describes what a digital signature file format is and related terms so that the reader can better understand the content of the thesis. At the end of this chapter, the specifications of DDOC and BDOC file formats are described. They are described in order to make it clear what data is stored in these file formats.

1.1 Digital signature file format and related terms

A file format defines a type of data stored in a file [2]. For example, every digital signature file format contains the following attributes that are linked to a digital signature:

- digital signature;
- public key certificate used for signing;
- Online Certificate Status Protocol response;
- timestamp of digital signature.

Besides, file format defines the structure of a file [2]. Many file formats contain a header, metadata, and end-of-file marker [2]. Based on the structure of the file format, the program or software displays its contents. For example, in order to display the contents of a BDOC digital signature file format, the DigiDoc software is used. Despite this, if the digital signature file format can be opened by using another program, the program may not have all the features needed to correctly display the data contained in this file format.

1.1.1 Digital signature

A digital signature is a mathematical technique used to verify the authenticity and integrity of digital data [3]. The first step in signing digital data is to hash it. The hashing process involves converting data of any size into the output of a specific form [3]. This is done by using a special kind of algorithm, also known as a hash function [3]. This means that digital data can vary significantly in size, but after hashing all hashes will have the same length. The data signing process is performed using public-key cryptography. Public-key cryptography is a system in which a pair of public and private keys is used [3]. The public key is available for

anyone to use, the private key is known only to the owner [3]. The private key is used to encrypt the generated hash [3]. After the hash is encrypted, the recipient can verify the authenticity of the digital data using the appropriate public key (provided by the signer).

It is worth noting that digital signatures are directly related to the content of each digital data. Thus, unlike handwritten signatures, which are usually the same regardless of the context of the document, each digital data with a digital signature will have a completely different digital signature value.

1.1.2 Public key certificate

To create a digital signature, a public key certificate is required. A public key certificate is an electronic document used to prove the ownership of a public key [4]. It contains a public key, as well as information about the owner of the key allowing it to be uniquely identified [4]. In addition, the certificate contains information about the certificate validity period, key assignment, certificate holder serial number, etc., depending on the format and version of the certificate [4].

The information contained in the public key certificate may be falsified. To resolve this issue, the Certificate Authority (CA) signs the public key certificate and confirms that the corresponding public key belongs to the person described in the certificate [5].

The public key certificate is based on the ITU-T X.509 standard (formerly CCITT X.509 or ISO/IEC 9594-8) that was published in 1988 [6]. Hence public key certificate is sometimes also referred to as X.509 certificate. There are several versions of the X.509 certificate. In the enterprise domain, the X.509 version 3 public key certificate is more often used, because it is more flexible (it contains numerous optional extensions thus it can be instantiated in many application-specific ways).

Informally, an X.509 version 3 certificate consists of two sections: certificate data and digital signature. The section related to certificate data contains the semantic part of the certificate. The main field there is the *Subject*. This field specifies the owner of the certificate. The section related to the digital signature contains a digital signature of all data from the section related to certificate data [6]. This digital signature is the result of a cryptographic hash

function of the certificate data, encrypted with the private key of the certification authority [7].

To describe the structure of the X.509 version 3 certificate the ASN.1 notation is often used. ASN.1 stands for Abstract Syntax Notation One and it describes the abstract data syntax in the field of telecommunications and computer networks [8]. Figure 1 shows the structure of the X.509 version 3 certificate.

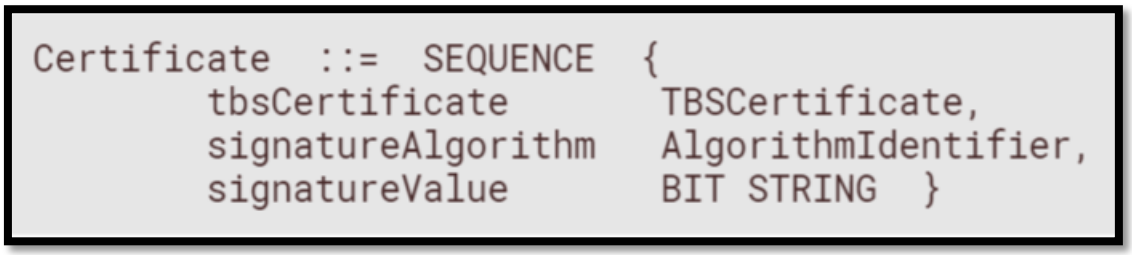

Figure 1. X.509 version 3 certificate structure [6].

As can be seen in Figure 1, the X.509 version 3 certificate structure consists of three fields: *tbsCertificate*, *signatureAlgorithm*, and *signatureValue*. The letters tbs in the *tbsCertificate* field name are deciphered as to be signed, that is, this field contains a block of data to be signed [6]. The *signatureAlgorithm* field specifies a digital signature algorithm that was used by the CA to sign the certificate [6]. The *signatureValue* field contains a digital signature computed upon the ASN.1 DER encoded *tbsCertificate* field [6].

As can be seen in Figure 1, each previously mentioned field is defined by various ASN.1 structures (*TBSCertificate*, *AlgorithmIdentifier*, *BIT STRING*). The author of this thesis did not consider it necessary to describe each ASN1.1 structure of the X.509 version 3 certificate since this does not interfere with understanding the essence of the work done. A detailed description of each ASN.1 structure of the X.509 version 3 certificate is posted on the following source [6].

1.1.3 Online Certificate Status Protocol

According to the EU eIDAS regulation Article 32[5], only signatures that have been issued with a valid certificate are valid. Therefore, before signing any document, it is necessary to verify that at the time the document was signed, the certificate was valid.

In Estonia, to verify that the certificate is valid, validity confirmation service is used [9]. The service is based on the Online Certificate Status Protocol (OCSP). In the case of OCSP, the client sends a certificate validity request to the server, and the server returns a signed response containing the status of that certificate and the time when the response was sent (timestamp) [9]. The status can have one of the following values: *valid*, *invalid*, *no information* [9].

To describe OCSP request and response structures, the ASN.1 notation can be used. Figure 2 shows the structure of OCSP request.

```
OCSPRequest     ::=     SEQUENCE {
        tbsRequest                      TBSRequest,
        optionalSignature   [0]     EXPLICIT Signature OPTIONAL }
```

Figure 2. OCSP request structure [10].

As can be seen in Figure 2, the OCSP request structure consists of two fields: *tbsRequest* and *optionalSignature*. The letters *tbs* in the *tbsRequest* field name are deciphered as *to be signed*, that is, this field contains signed OCSP request [10]. The *optionalSignature* field contains a digital signature algorithm, digital signature value after hashing and certificates required by the server to verify the signed response [10].

Figure 3 shows the structure of OCSP response.

[5] https://www.eid.as/Regulation#article32

```
OCSPResponse ::= SEQUENCE {
      responseStatus         OCSPResponseStatus,
      responseBytes          [0] EXPLICIT ResponseBytes OPTIONAL }
```

Figure 3. OCSP response structure [10].

As can be seen in Figure 3, the OCSP response structure consists of two fields: *responseStatus* and *responseBytes*. The *responseStatus* field provides status for the response [10]. If the value of *responseStatus* field is *no information*, then the *responseBytes* field will be empty [10]. The *responseBytes* field contains the version of the response syntax, name or hash of the responder's public key, time at witch the response was given, signature computed across a hash of the response, signature algorithm and optional extensions [10].

As can be seen in Figure 2 and Figure 3, each previously mentioned field is defined by various ASN.1 structures (*TBSRequest*, *Signature*, *OCSPResponseStatus*, and *ResponseByes*). The author of this thesis did not consider it necessary to describe each ASN1.1 structure of OCSP request and response, since this does not interfere with understanding the essence of the work done. A detailed description of each ASN.1 structure of OCSP request and response is posted on the following source [10].

1.1.4 Time-Stamp Protocol

Time-Stamp Protocol (TSP) is a cryptographic protocol that allows to create evidence that digital data existed at or before a particular time [11]. The evidence that the digital data existed at or before a particular time is called a trusted timestamp and it is issued by a timestamping authority (TSA) [12].

The workflow of the TSP can be described as follows. First, using special cryptographic tools, a hash of the digital data is calculated [12]. Then, based on this hash, a request for a timestamp is generated. The request is sent to the TSA, which extracts the hash from the request, adds the present time to it, calculates the hash of this concatenation, and signs it all with its private key [12]. In the end, the signed hash is sent back to the requester of the timestamp [12].

To describe the time-stamping request and response structures, the ASN.1 notation can be used. Figure 4 shows the structure of the time-stamping request.

```
TimeStampReq ::= SEQUENCE  {
   version                      INTEGER  { v1(1) },
   messageImprint               MessageImprint,
     --a hash algorithm OID and the hash value of the data to be
```

Figure 4. Time-stamping request structure [12].

As can be seen in Figure 4, the time-stamping request structure consists of two fields: *version* and *messageImprint*. The *version* field specifies the version of the time-stamping request [12]. The *messageImprint* field contains a hash algorithm and a hash of the data to be time-stamped [12].

Figure 5 shows the structure of time-stamping response.

```
TimeStampResp ::= SEQUENCE  {
      status                  PKIStatusInfo,
      timeStampToken          TimeStampToken     OPTIONAL  }
```

Figure 5. Time-stamping response structure [12].

As can be seen in Figure 5, the time-stamping response structure consists of two fields: *status* and *timeStampToken*. The *status* field contains information about the status of response and can have one of the following values: *granted*, *grantedWithMods*, *rejection*, *waiting*, *revocationWarning*, *revocationNotification*, and *keyUpdateWarning*. The *timeStampToken* field contains the version of the time-stamp token, TSA's policy under which the response was produced, unique integer assigned by the TSA to each *timeStampToken* field, the time at which the time-stamp token has been created by the TSA and the name of TSA.

As can be seen in Figure 4 and Figure 5, each previously mentioned field is defined by various ASN.1 structures (*Integer*, *MessageImprint*, *PKIStatusInfo*, and *TimeStampToken*). The author of this thesis did not consider it necessary to describe each ASN1.1 structure of time-

stamping request and response since this does not interfere with understanding the essence of the work done. A detailed description of each ASN.1 structure of the time-stamping request and response is posted on the following source [12].

1.2 Digital signature file formats specification

First, a DDOC digital signature file format was created. Problems were using this file format. For example, there was problem sending DDOC file format by email because some mail servers have filtered DDOC mail attachments out [13]. Later a new digital signature file format was created, called BDOC. Today, if a person digitally signs a document using, for example, DigiDoc software, then by default BDOC digital signature file format is created.

The extension of DDOC file format is .ddoc and the extension of BDOC file format is .bdoc. Each of these file formats has a unique structure. Even though the structure of these file formats is different, they contain similar data.

1.2.1 DDOC file format

DDOC file format is based on the ETSI TS 101 903 Standard [14]. This standard is called XML Advanced Electronic Signatures and it describes the structure of digital signature file formats. DDOC is an XML document and its structure is determined by the following elements: signed files, signatures related to the files, signatory's digital certificate, validity confirmation, and the validity confirmer's certificate [15]. A detailed description of the structure of this file format is provided in Appendix I. Appendix I contains a table (Table 1) that presents the elements of the DDOC file format and their meaning. The meanings of the DDOC file format elements was obtained from the following source [16].

1.2.2 BDOC file format

BDOC file format was created later and was developed to replace the DDOC file format [15]. BDOC file format is fully compliant with the following standards: ETSI standard TS 101 903, ETSI standard TS 103 171, ETSI standard TS 102 918, and ETSI standard TS 103 174 [17]. This means that it meets all the requirements of these standards. The BDOC file format is a ZIP container with signed files, signatures, and protocol management information [18]. Since ZIP

is a compressed format, the size of the files stored in it may be smaller than that of the DDOC format [26]. The content of the BDOC file format is illustrated In Figure 6.

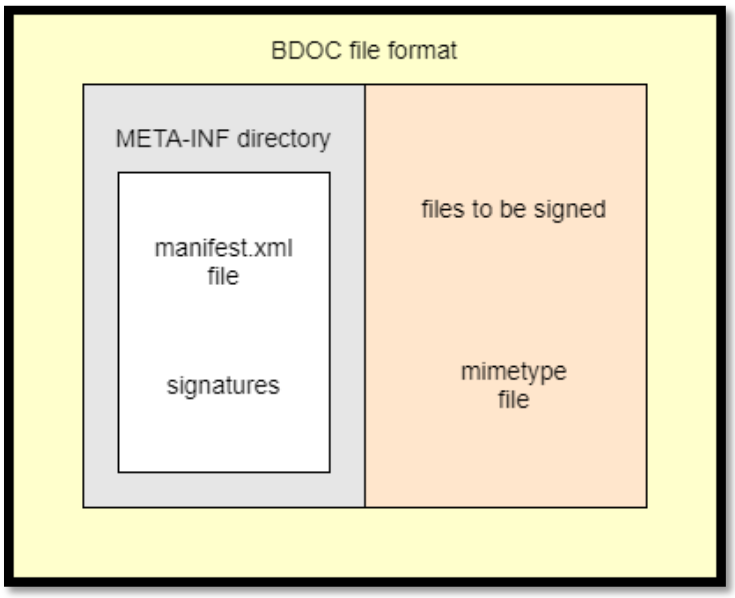

Figure 6. BDOC file format content.

From Figure 6 it can be seen that the BDOC file format contains a *META-INF* directory. The *META-INF* directory, in turn, contains a *manifest.xml* file. The root element of this file is *<manifest:manifest>.* This root element contains *<manifest:file-entry>* child elements, and their number depends on how many documents were signed [19]. The first *<manifest:file-entry> element* is different from the rest. It's *manifest:full-path* attribute value is always */* and it indicates to the root directory of the file format [19]. It's *manifest:media-type* attribute value specifies the type of a file format [19]. For example, its value can be *application/vnd.etsi.asic-e+zip* and this value is used to identify BDOC file format. Other *<manifest:file-entry>* elements contain the same attributes, but their meaning is different. The *manifest:full-path* attribute of these elements specifies document location within the file format and its value is a name of a document [19]. The *manifest:media-type* attribute of these elements specifies the MIME media type of a document [19]. MIME stands for Multi-Purpose Internet Mail Extensions [20]. It is a protocol that is used to exchange different types of data files on the Internet: application programs, videos, texts, images, audios, and other [20].

The *META-INF* directory also contains signatures. A directory may contain several signatures, their number depends on how many people signed the document. Signatures are XML Signature[6] files therefore their extension is .xml. Each file name contains a string *signatures* followed by the signature number. The numbering of XML Signature files starts from zero.

The structure of these XML files can be different. This is because there are two types of BDOC file formats: BDOC-TM and BDOC-TS. These file formats have different extensions. Extension of BDOC-TM file format is .bdoc and extension of BDOC-TS file format is .asice.

In BDOC-TM file format the long-term probative value of the signature is guaranteed by using RFC 2560 based time-mark protocol [21]. This protocol specifies OCSP request data that needs to be requested and OCSP response data that should be sent as a response to OCSP request [17]. The value of *producedAt* field in the OCSP response structure is considered as a time of signature creation [17].

In BDOC-TS additional timestamps are required in order to specify the time of certificate validity information. For this, additional element is included to the signature structure - *</xades:SignatureTimeStamp>* [17]. The value of this element is considered as a time of signature creation [17].

The structure of BDOC XML Signature file without validation data is provided in Appendix II. Appendix II contains a table (Table 2) that presents BDOC XML Signature file elements without validation data and the meanings of these elements. The meaning of elements presented in Table 2 were obtained from the following source [17].

Also in Figure 6, it is noted that the BDOC file format contains files to be signed. The fact is that after signing the files, copies are made from them. Then copies of the files are saved in a ZIP container (BDOC file format).

The BDOC file format also contains the *mimetype* file, as noted in Figure 6. The content of this file is the following string: *application/vnd.etsi.asic-e+zip*. This string is used to identify BDOC file format.

[6] https://www.xml.com/pub/a/2001/08/08/xmldsig.html

2 Data collection

In this thesis, data refers to digital signature file formats that are used in Estonia. In order to collect data, it was necessary to choose a document register from which data could be downloaded. Since it would take a lot of time to manually download data from the document register, a program was created. Given that, this chapter first describes a chosen document register. Then the implementation details of a program are described. At the end of this chapter, the collected data is described.

2.1 Document register

To collect data, it was necessary to choose a document register. The author of this thesis decided to choose for this purpose the Tartu town document register. This document register is a web application that contains documents related to Tartu City Government, Tartu City Council, and all authorities administered by the Tartu City Government [22].

Tartu town document register was chosen because it contains a document search engine using which it is possible to filter out unnecessary documents. For example, it is possible to filter out documents with restricted access. In addition, this document register contains documents of various types:

- legislation;
- permits, injunctions, and other decisions;
- agendas;
- protocols;
- agreements;
- incoming mail;
- outgoing mail.

This indicates that a variety of data can be stored in the database, and thus various questions can be answered based on stored data. For example, the following question can be answered: "What type of documents are signed most?". In addition to the previously presented arguments, this document register was also chosen because it contains all the

types of digital signature file formats that are used in Estonia (DDOC, BDOC-TM, and BDOC-TS).

2.2 Implementation details

The created program consists of two parts. The first part is a sequence of commands that need to be executed in order to download digital signature file formats. In other words, the first part is a script. To write a script, the Selenium WebDriver development tool was used. The second part is a unit test case, which was written in order to make sure that the script works as intended. To write a unit test case, JUnit testing framework was used. For the full source code of the program, see Appendix III.

2.2.1 Selenium WebDriver

To write a script, the Selenium WebDriver development tool was used. One of the reasons why Selenium WebDriver was chosen is that it is a publicly available automation framework and is free. Scripts written using Selenium WebDriver framework are compatible with many browsers and can be executed on various operating systems. Besides, it is possible to monitor what actions the machine performs during script execution. This is convenient because it makes it easy to notice if there are any errors in the written script.

Selenium WebDriver is a development tool for automating the actions of a web browser [23]. In most cases, this development tool is used to test web applications but is not limited to this. It can also be used to get data from various sources (websites).

Selenium WebDriver architecture consists of four components: Selenium Language Bindings, JSON Wire Protocol, Browsers, and Browser Drivers [24]. These components form the Selenium WebDriver API, which helps in data transferring between the program and the browser. Figure 7 illustrates the architecture of Selenium WebDriver.

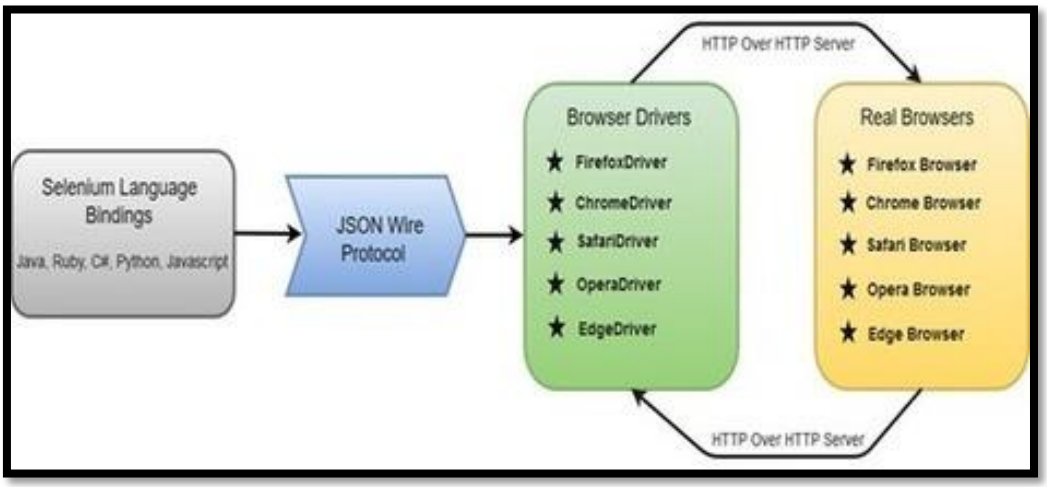

Figure 7. Selenium WebDriver architecture [25].

In Figure 7, the Selenium WebDriver architecture is presented in the form of a diagram. The components are presented in the diagram in order from left to right (see the direction of the arrows).

The first component of this diagram is the *Selenium Language Bindings*. This component is used to make it possible to write a script in several programming languages [24]. For example, it is possible to download the *Java Bindings* package in order to be able to write a script in Java programming language. *Selenium Java Bindings* component is a ZIP container that comprises of JAR[7] files.

The next component presented in Figure 7 is the *JSON Wire Protocol*. This component is used to transfer the data between the client (requesting program) and the server (responding program) [24]. By the name of the component, it is clear, that the data is transmitted in JSON[8] format.

The last two components, which are presented in Figure 7, form a cycle. They are presented as a cycle because browser driver repeatedly transfers data using HTTP server to a browser and vice versa. Browser is used to access data from various web sources. Using it, it is possible to send a request to a remote server. The remote server, in turn, sends a response to the

[7] https://fileinfo.com/extension/jar

[8] https://techterms.com/definition/json

browser. As shown in Figure 6, the browsers supported by Selenium WebDriver are Firefox, Chrome, Safari, Opera, and Edge. Other, less well-known browsers are also supported but not marked in this diagram. The Browser Drivers component presented in Figure 7 is used in order to establish a secure connection with the browser without revealing the internal logic of the browser's functionality. When the script is executed using Selenium WebDriver, the following operations are performed [26]:

1. an object is first created that represents the browser driver;
2. the browser driver uses an HTTP server for getting the HTTP requests;
3. after the execution of each Selenium command, an HTTP request is created and sent to the HTTP server;
4. the HTTP server determines the steps needed for implementing the Selenium command;
5. the implementation steps are sent to the browser;
6. after the execution status is returned from the browser, the HTTP server sends the execution status back to the script.

2.2.2 Script

The script is a method written in the Java programming language. This method takes no arguments. The body of the method contains commands that need to be executed in order to download digital signature file formats. All these commands are written inside the *try* statement so that if errors occur, the program would continue execution. The *catch* statement is used in order to save all error messages using the *StringBuffer* object. Using this approach, it was possible to immediately detect several errors and then correct them. In the final version of the script, there is no need to use *try* and *catch* statements since errors no longer occur when executing the script. They remained in the script to make it clear how the script was written.

At the beginning of the script, the following Selenium command is used to redirect to the Tartu town document register home page.

```
driver.get("https://www.tartu.ee/et/dokumendid");
```

The result of this command can be seen in Figure 8.

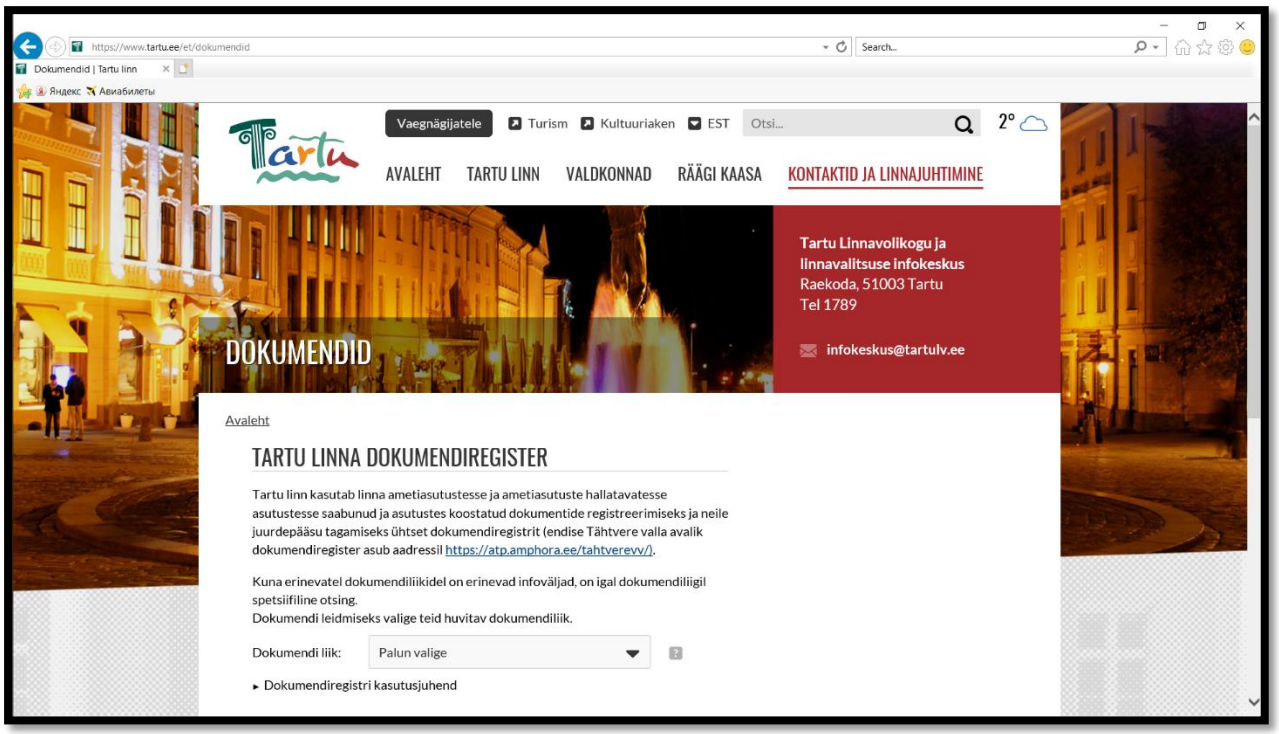

Figure 8. Tartu town document register home page.

Then, using a search engine contained in the document register, data about documents to be found is selected. In this example, all incoming letters that are public are selected in the interval from January 1, 2019, to December 31, 2019. After that, the *Otsi* button is pressed in order to send a request for documents to the server. The following Selenium commands were executed in order to complete these steps.

```
WebElement frame = driver.findElement(By.className("doc-iframe"));
driver.switchTo().frame(frame);
driver.findElement(By.name("WebView")).click();
new
Select(driver.findElement(By.name("WebView"))).selectByVisibleText("Väljami
nevad kirjad");
driver.findElement(By.name("WebView")).click();
driver.findElement(By.name("jpp")).click();
new
Select(driver.findElement(By.name("jpp"))).selectByVisibleText("Avalikud");
driver.findElement(By.name("jpp")).click();
driver.findElement(By.xpath("(.//*[normalize-space(text()) and normalize-
space(.)='Märksõna(d):'])[1]/following::div[4]")).click();
driver.findElement(By.id("edit-date-start")).click();
driver.findElement(By.id("edit-date-start")).clear();
driver.findElement(By.id("edit-date-start")).sendKeys("01.01.2019");
driver.findElement(By.xpath("(.//*[normalize-space(text()) and normalize-
```

```
space(.)='Märksõna(d):'])[1]/following::div[4]")).click();
driver.findElement(By.id("edit-date-end")).click();
driver.findElement(By.id("edit-date-end")).clear();
driver.findElement(By.id("edit-date-end")).sendKeys("31.12.2019");
driver.findElement(By.xpath("(.//*[normalize-space(text()) and normalize-
space(.)='Juurdepääs:'])[1]/following::button[1]")).click();
```

Figure 9 shows the result of these commands. In Figure 9 it can be seen, that data about documents is selected and that the *Otsi* button is pressed.

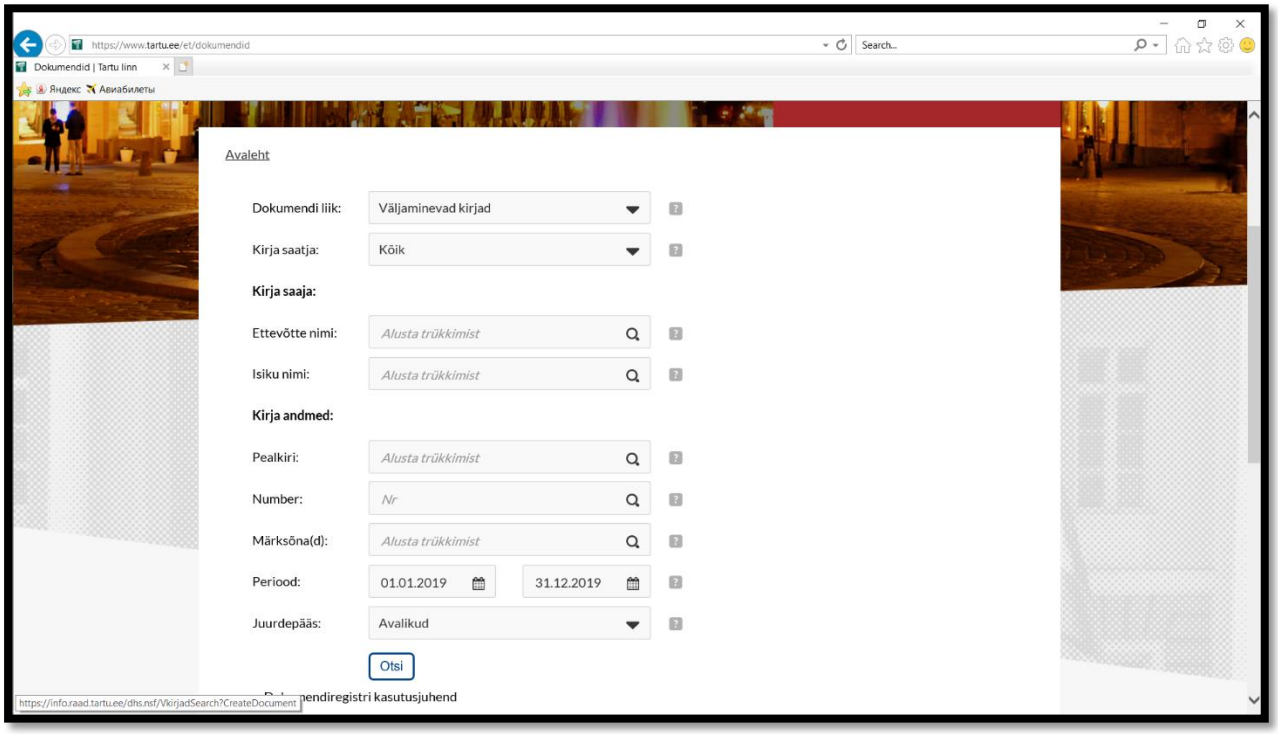

Figure 9. Request for documents.

After requesting documents, information about the documents is displayed in the table, namely: date of publication, document number, a field of activity, recipient, and title of the document. The title of the document is presented in the form of a link, clicking on which a new tab will open. This new tab contains detailed information about the document and a link to download it. It is worth noting that the document may not be a digital signature file format, so before clicking on the link, in order to download the document, the document extension is checked. The link to download the document is clicked only if the document extension is DDOC, BDOC, or ASICE. The following commands were executed in order to download requested documents that correspond to digital signature file formats.

```
while (true) {
    List<WebElement> linksToDocuments =
driver.findElement(By.className("documents")).findElement(By.tagName("table
")).findElements(By.tagName("a"));
    for (WebElement linkToDocument: linksToDocuments) {
        linkToDocument.click();
        // Switch to another tab
        ArrayList<String> allTabs = new
ArrayList<String>(driver.getWindowHandles());
        allTabs.remove(oldTab);
        driver.switchTo().window(allTabs.get(0));
        try {
            List<WebElement> files =
driver.findElement(By.className("files")).findElements(By.tagName("a"));
            for (WebElement file: files) {
                if (file.getText().contains(".bdoc") ||
file.getText().contains(".ddoc") || file.getText().contains(".asice")) {
                    file.click();
                }
            }
        } catch (Exception exception) {
            // If there are no files related to this document, then just
change focus back to old tab (do nothing)
        }
        // Change focus back to old tab
        driver.close();
        driver.switchTo().window(oldTab);
        driver.switchTo().frame(frame);
    }
}
```

In order to demonstrate how one of the requested documents was downloaded, Figure 10 is presented. Figure 10 consists of two interconnected pictures. In the picture on the left, the table of requested documents is presented. After clicking on any link in a table, a new tab will open. In the picture on the right, the content of the opened tab can be seen after clicking on the first link in the table. The opened tab contains a link to download the document.

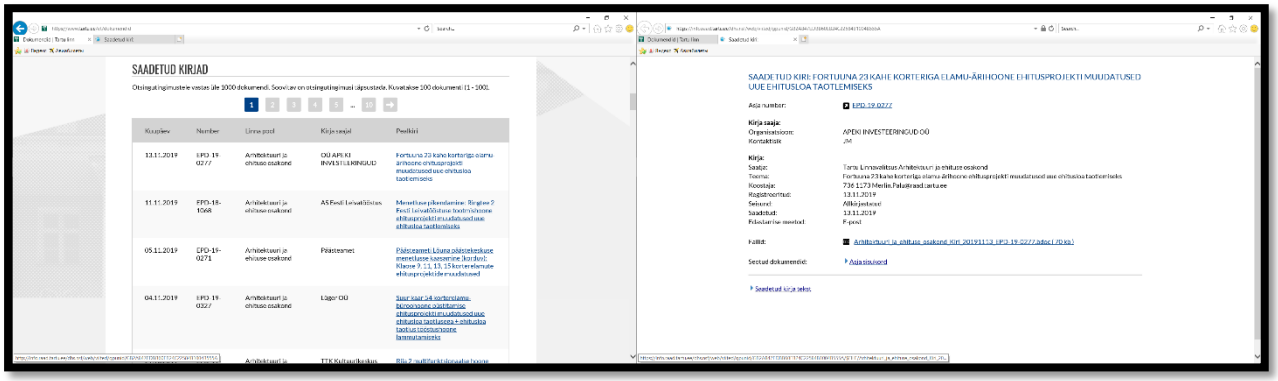

Figure 10. Downloading first document in the table.

Since information about the requested documents is presented on several pages, it was also necessary to write commands to navigate through the pages. In order to implement this functionality, the following commands were used.

```
// Go to next page, if exists
if (driver.findElements(By.className("next")).size() > 0) {
    driver.findElement(By.className("next")).click();
    Thread.sleep(5000);
}
else {
   break;
}
```

2.2.3 JUnit

Testing is the process of checking the functionality of a program in order to confirm that it works in accordance with certain requirements. Unit testing is testing where individual software components are tested. As a component, it is customary to consider a method or a class. In order to make sure that the previously described method (script) works as expected, the JUnit framework was used.

JUnit is a framework designed to test programs written using Java technology. JUnit belongs to the xUnit[9] family of frameworks [27]. This framework is used in Test-driven development (TDD). TDD is a software engineering technique in which the unit test case for a certain functional is written first, and then the implementation of this functional [28]. In order to write a unit test case, the framework provides the following features [29]:

- Fixtures;
- Test suites;
- JUnit runners;
- JUnit classes.

Fixtures are used to bring the environment in which the test runs to the required state so that the test can be run [29]. For example, prepare a specific set of data in order to load a database. Test suites, as the name suggests, are used to bundle a few unit test cases and test them together. A JUnit Runner is a class that extends JUnit abstract Runner class and it is

[9] https://martinfowler.com/bliki/Xunit.html

used for running test classes [29]. JUnit classes are used to write unit test cases. The most frequently used JUnit class is *Assert*. Without using the methods of this class, it is impossible to verify whether the component passed the test or not.

2.2.4 Unit test case

A unit test case is a part of the code that verifies that another part of the code (in particular, the method) works in accordance with certain requirements. The formally described unit test case is characterized by known input data and the expected output of the program, which is known before the test starts.

In this case, the unit test case was written primarily in order to bring the environment in which the test runs to the required state. Thus, a person who wants to run the previously described script can get the same result. In order to achieve this, the following *setUp* method of Fixture was used.

```
@Before
public void setUp() {
    driver = new ChromeDriver();
    driver.manage().timeouts().implicitlyWait(5, TimeUnit.SECONDS);
    Dimension dimension = new Dimension(1920,1080);
    //Resize current window to the set dimension
    driver.manage().window().setSize(dimension);
    errors = new StringBuffer();
}
```

The *@Before* annotation indicates that the method must be executed before the unit test case. Inside this method, a driver is configured that will be used to establish a connection with the browser. In this example, the *ChromeDriver* is used. *ChromeDriver* is a separate executable that is used to control Chrome. *implicitlyWait* method directs the Selenium WebDriver to wait for the Document Object Model[10] (DOM) element to appear a certain measure of time before throwing an exception. In this case, the wait time is five seconds. Then the browser window is resized. This is done so that all the necessary DOM elements are displayed in the browser window. Otherwise, it would not be possible to interact with all elements of the DOM. In the end, the *StringBuffer* object is created in which all error messages

[10] https://www.w3.org/TR/REC-DOM-Level-1/introduction.html

will be saved in case they arise. This object is used in the unit test case in order to verify that after running the script this object will not contain any error messages.

The unit test case is a method written in the Java programming language using the JUnit library. Below the unit test case can be seen that was written to ensure that the script works as required.

```
@Test
public void testDownloadFileFormats() {
    downloadFileFormats();
    assertEquals(errors.toString(), "");
}
```

The *@Test* annotation indicates that the method to which this annotation is attached can be run as a unit test case. Inside this unit test case, the method is called, which is a script. Then the *assertEquals* method of *Assert* class is called. By calling this method, it is checked whether the *StringBuffer* object contains error messages or not. If not, then the unit test case is considered passed.

At the end the following *tearDown* method of Fixture was used.

```
@After
public void tearDown() {
    driver.quit();
}
```

The *@After* annotation indicates that this method must be executed after the unit test case. Inside this method, a *quit* method is used. This method is used to close all tabs, shut down the browser, and free up all resources.

2.3 Data collection

The data was collected from Tartu Down Document Register. The data was collected by executing the previously mentioned program. For the collected data, see Appendix IV.

It is worth noting that the data is not collected in equal proportions. By this, the author of this thesis implies that the number of collected DDOC, BDOC-TM and BDOC-TS file formats is not the same. This is because in Estonia, at first, it was only possible to create DDOC digital signature file formats. BDOC-TM, and BDOC-TS digital signature file formats began to be used

in Estonia later. It became possible to create the BDOC-TS digital signature file format only since 2013 [30]. The BDOC-TS digital signature file format even later, in 2015.

In order to download data from Tartu town document register, the following documents were requested using the search engine located in the document register:

- legislations published from 2009 to 2019;
- permits, injunctions, and other decisions from 2009 to 2019;
- protocols published from 2009 to 2019;
- incoming mails published from 2009 to 2019;
- outgoing mails published from 2009 to 2019.

The interval from 2009 to 2019 was chosen for two reasons. The first reason is that until 2009 the documents in the document register were only in PDF format. The second reason is that at the time of writing the thesis, 2020 has just begun. Therefore, the author did not want to download the documents that were published in the register for the first few months of 2020.

Below, Chart 1 can be seen that shows the number of downloaded DDOC, BDOC-TM, and BDOC-TS digital signature file formats. As can be seen in Chart 1, the number of downloaded BDOC-TS digital signature file formats is the smallest, namely 443. This can be explained by the fact that this format has become the default format only recently, namely since 2019. Chart 1 shows that the number of downloaded DDOC file formats is 4,609, and this number is not the largest that can be seen in this chart. This can be explained by the fact that this file format was created earlier than others, and earlier, as might be assumed, people were less likely to sign documents. The number of downloaded BDOC-TM digital signature file formats is the biggest. One of the reasons why BDOC-TM digital signature file formats were downloaded the most is the fact that this format is used in Estonia as the default digital signature format from 2015 to the end of 2018, i.e. for 3 years. As can be seen in Chart 1, the number of downloaded BDOC-TM digital signature file formats is 5,266. From the foregoing, it can be concluded that in total about 10,000 digital signature file formats have been downloaded.

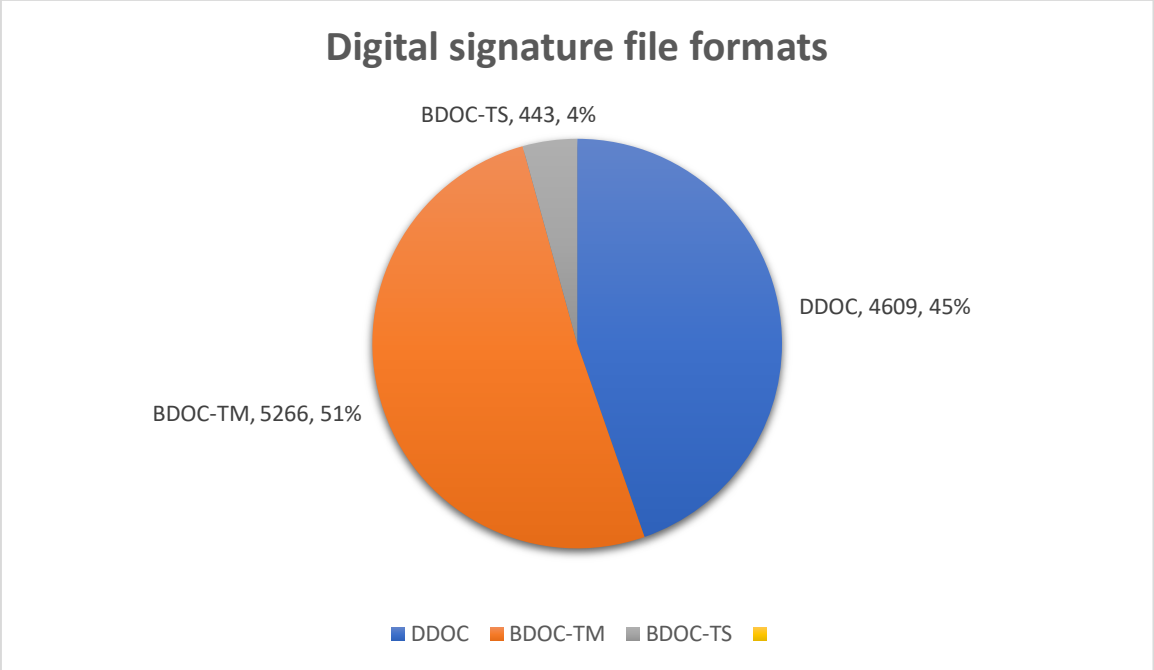

Chart 1. Collected digital signature file formats (sorted by format type).

3 Database creation

As mentioned earlier, the goal of this thesis was to create a database in which the data contained in the DDOC and BDOC digital signature file formats will be stored. Therefore, for data storage, it was necessary to choose a database management system (DBMS). In addition, it was necessary to write a program, which would extract data from collected digital signature file formats and insert this data into the database. Given that, this chapter first describes DBMS that was used for data storage. Then, the table structure of the created database is described. Following the table structure, the technologies that were used to create a program are described. At the end of this chapter, the implementation details of a program are described.

3.1 Database management system

The database management system is a software system for creating and managing databases [31]. By using a DBMS, it is possible to read, update, and delete data in a database. Exist several types of DBMS. In this case, to create a database, a relational database management system (RDBMS) was used.

An RDBMS is a DBMS that is used to create a relational database [32]. In a relational database, data is stored in tables [32]. A table is a data structure that consists of rows and columns. In each column of the table a specific data type is stored, in each cell - the attribute value. Rows of the table represent a set of related values that are related to a single object. Rows can contain a unique identifier called the primary key, and rows from several tables can be linked via foreign keys.

In order to create a database, a PostgreSQL RDBMS was used. This RDBMS was chosen because it is free and open-source RDBMS. This database management system is designed for extra-large database management [33]. Nowadays, it is widely used by many famous companies, like Uber, Spotify, and Instagram [34].

3.2 Database table structure

In order to create tables, PostgreSQL SQL statements were executed using the SQL query tool[11]. These statements consist of tokens. Each token can represent either a constant, keyword, or special symbol. For the SQL statements that were used for database table structure creation see Appendix V.

A database table structure is shown in Figure 11. As can be seen in Figure 11, a database structure consists of 4 tables. Between these tables, one-to-one relationship exists. In a one-to-one relationship, one record in a table is associated with only one record in another table [35]. In addition, Figure 11 shows data types of attributes in every table.

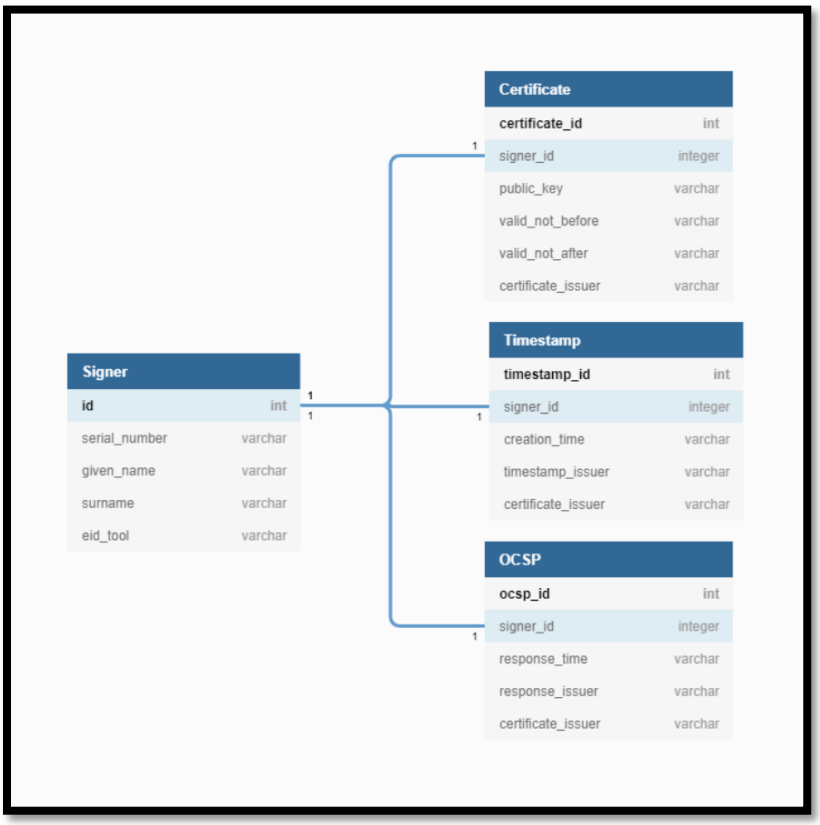

Figure 11. Database table structure.

The central table is the *Signer* table, which contains data describing a person, who signed the document. To be more precise, this table contains the following attributes:

[11] http://www.sql-query-tool.com/

- *Id*. This attribute is marked as a primary key and it is used to uniquely identify all table records;
- *serial_number*. This attribute specifies personal identity code;
- *given_name*. This attribute specifies signer's given name;
- *surname*. This attribute specifies signer's surname;
- *eid_tool*. This attribute specifies type of signer's certificate. For example, if a document is signed using Mobile-ID, then the value of this attribute will be *ESTEID (MOBIIL-ID)*.

The *Certificate* table contains data related to signer's certificate. This table is specified by the following attributes:

- *certificate_id.* This attribute is marked as a primary key and it is used to uniquely identify all table records;
- *signer_id*. This attribute is marked as a foreign key and it references the primary key of the *Signer* table, thereby establishing a link with this table;
- *public_key.* This attribute specifies type of public key certificate (RSA or ECC).
- *valid_not_before*. This attribute specifies the date then the client certificate becomes valid.
- *valid_not_after*. This attribute specifies the last day before certificate expiration.
- *certificate_issuer.* This attribute specifies an authority that issued certificate to the signer.

The *OCSP* table contains data related to certificate validity. This table is specified by the following attributes:

- *ocsp_id*. This attribute is marked as a primary key and it is used to uniquely identify all table records;
- *signer_id*. This attribute is marked as a foreign key and it references the primary key of the *Signer* table, thereby establishing a link with this table;
- *response_time*. This attribute specifies a time when OCSP response was issued.
- *response_issuer*. This attribute specifies an authority that issued the OCSP response;

- *certificate_issuer*. This attribute specifies an authority that issued certificate to the OCSP authority.

The *Timestamp* table contains data related to the timestamp that was given after signing a document. This table is specified by the following attributes:

- *timestamp_id*. This attribute is marked as a primary key and it is used to uniquely identify all table records.
- *signer_id*. This attribute is marked as a foreign key and it references the primary key of the *Signer* table, thereby establishing a link with this table;
- *creation_time*. This attribute specifies a time when a timestamp was given;
- *timestamp_issuer*. This attribute specifies an authority that issued the timestamp.
- *certificate_issuer*. This attribute specifies an authority that issued certificate to the Time Stamping Authority.

3.3 Technologies

In order to write a program with the help of which data contained in the DDOC and BDOC digital signature file formats would be inserted into the database, special technologies were used. Data from the collected digital signature file formats were extracted using the DigiDoc4 library. The extracted data was inserted into the database using the Java database connectivity API.

3.3.1 DigiDoc4j

DigiDoc4j is a Java library that is used to digitally sign documents. It can also be used to create and validate digital signature file formats [36]. To be more precise, then by using this library, it is possible to create BDOC-TM and BDOC-TS digital signature file formats and to validate BDOC-TM, BDOC-TS, and DDOC digital signature file formats. In this case, as mentioned earlier, it was used to extract data from BDOC and DDOC file formats.

The indisputable fact is that good documentation has been written for this library. For example, in order to get familiar with this library, it is possible to read the DigiDoc4j API

document[12]. By reading this document, the reader can familiarize with the following aspects of this library [37]:

- Supported functionality. In this section of a document, the main functional features of a library are described;
- Architecture. In this section of a document, a diagram of the components that make up the library architecture are presented, and each component is described;
- API overview. In this section of a document, examples of how to use this library are presented;
- Utility program overview. This section of a document describes what this tool is all about, how to install it and examples of its use are demonstrated.

In order to get acquainted with this library, the author of this thesis used this document. There are also other sources to familiarize yourself with the library. All the necessary information about the library and source code can be found on the following source [38].

3.3.2 Java Database Connectivity

Java Database Connectivity (JDBC) is a Java API that is used to establish a connection between the Java program and the database. When the connection to the database is established, it is possible to perform various tasks. JDBC library contains different APIs by using which it is possible to perform the following tasks related to database usage [39]:

- Making a connection to the database;
- Creating SQL statements. Their meaning has already been described in the thesis;
- Executing SQL queries. The SQL query is a request for data from one or more database tables and it is generated by using SQL statements;
- Viewing and modifying the resulting records.

In order to access a database, JDBC provides two types of processing models (architectures): two-tier and three-tier [40]. In this case, two-tier architecture was used. In two-tier architecture, only the JDBC driver is needed to establish a connection with the database [40].

[12] http://open-eid.github.io/digidoc4j/

It means that in this architecture there is no middle layer (application server), as in three-tier architecture. The two-tier architecture can be seen in Figure 12. Figure 12 shows that the user sends a request to the database via the application. The result of the request is sent back from the database system to the user, as can be seen in Figure 12.

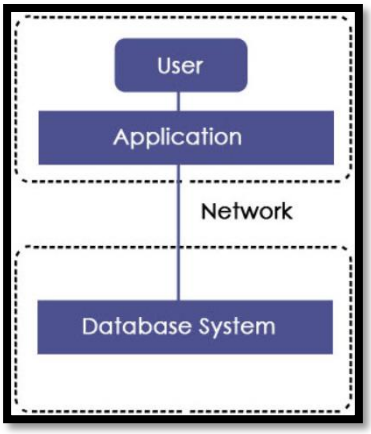

Figure 12. Two-tier architecture of JDBC API [40].

When Java architecture is mentioned, it will not be superfluous to mention its components. JDBC architecture consists of the following components [40]:

- *Driver*. This component is used to manage various database drivers. Via this component it is possible to create a *DriverManager* object;
- *Connection*. This component is used to establish a session between the java program and the database;
- *Statement*. This component is used to provide methods for the execution of the SQL queries;
- *PreparedStatement*. This component is used when there is a need to execute a SQL query several times;
- *CallableStatement*. This component is used when there is a need to access stored procedures;
- *ResultSet*. This component is used to hold data retrieved after the execution of the SQL query.

In Figure 13 the architecture of JDBC with the positioning of all its components can be seen. In Figure 13 arrows are shown. They are depicted to display how the components appeal to each other and in what order.

Also in Figure 13 different drivers are presented: PostgreSQL JDBC Driver, Oracle JDBC Driver, and Sybase JDBC Driver. Of course, other JDBC Drivers also exist. In this case, the PostgreSQL JDBC Driver was used. PostgreSQL JDBC Driver allows java programs to connect to a PostgreSQL database.

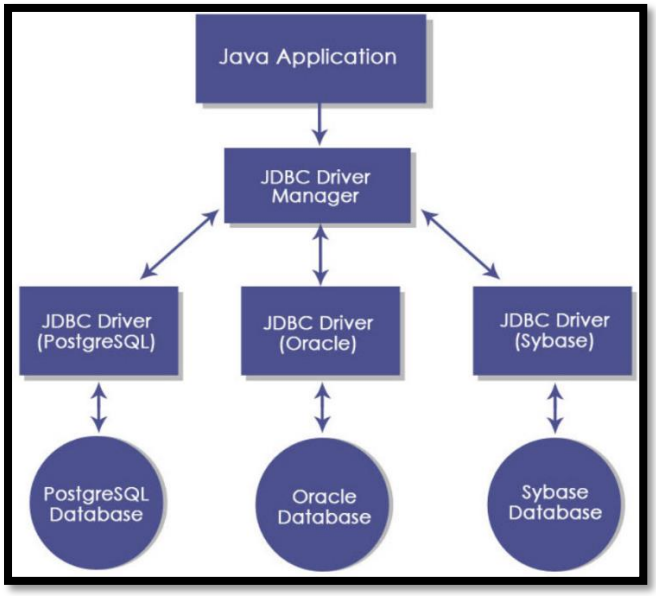

Figure 13. JDBC API architecture [40].

3.4 Implementation details

Based on the technologies that were used, it can be concluded that the program was written in the Java programming language. In this section, the author of this thesis considered it necessary to describe only the implementation of the most important parts of the program, namely:

- program command line arguments;
- static fields;
- signatures handling;

- signer's data extraction;
- signer's certificate data extraction;
- OCSP data extraction;
- digital signature file format timestamp extraction;
- interaction with the database.

Implementation related to object models[13] will not be described in the thesis. In addition, implementation related to iteration through an array with digital signature file formats and invocation of methods to insert various data to the database will not be described. For the full source code of a program, see Appendix II.

3.4.1 Program command line arguments

The program takes one command-line argument that is passed at the time of running the program. This argument specifies the path to the directory that contains digital signature file formats. It is assumed that the specified directory contains only digital signature file formats. Using this argument, an object of class *File* is initialized by passing the path of the directory to the constructor of this class. The following Java line of code was used to achieve this.

```
File directory = new File(args[0]);
```

Then, using this object that is created by using a command-line argument, all digital signature file formats are saved in an array. In the end, data is extracted from each digital signature file format.

3.4.2 Static fields

Two static fields are created at the beginning of the program execution. The first static field represents a *Logger* object provided by the *log4j* library. This object is used to store warnings as messages that occur during the execution of a program. In addition, this object is used to designate informational messages that highlight the progress of the application. The following line of code was used to create this object.

```
final static Logger logger = Logger.getLogger(FileFormatsProcessing.class);
```

[13] https://www.techopedia.com/definition/8635/object-model

The second static field represents a *PostgreSQL* object. This object is used to interact with the database. The following line of code was used to create this object.

```
final static PostgreSQL jdbc = new PostgreSQL();
```

Examples of the use of *PostgreSQL* object will be presented below. The following aspects will be presented: connection to the database, data insertion into the database, and disconnection from the database.

3.4.3 Signatures handling

One digital signature file format may contains several signatures (XML Signature files). First, in order to access each XML Signature file contained in a file format, it was necessary to create an object of type *Container* provided by DigiDoc4j library. Therefore, by using this object, it is possible to handle data files and signatures in a digital signature file format. To do this, the *fromExistingFile* method provided by DigiDoc4j library is called, which takes the *pathname* argument. This argument specifies the path to the directory that contains digital signature file formats. In order to get a list of all signatures in a digital signature file format, the *getSignatures* method of a *Container* interface provided by DigiDoc4j library is used. Then *for loop* is used in order to iterate through the list that contains signatures. The following lines of code were used to implement this functionality.

```
Container container =
ContainerBuilder.aContainer().fromExistingFile(pathname).build();

for (Signature signature : container.getSignatures()) { . . . }
```

As can be seen from these lines of code, the loop body is left blank (statements are missing). In the program inside this loop, various statements are executed, in order to extract various data from signatures.

3.4.4 Signer's data extraction

As mentioned earlier, the X.509 version 3 certificate consists of three fields: *tbsCertificate*, *signatureAlgorithm* and *signatureValue*. The data related to signer is contained in the *tbsCertificate* field. To be more precise, this data is contained in the *subject* field of *TBSCertificate* ASN.1 structure. The following data was extracted from the *subject* field:

- signer's serial number;
- signer's given name;
- signer's surname;
- eID tool that was used to sign a document.

In the beginning, in order to access the data contained in the *subject* field, a *X509Cert* object was created by calling a *getSigningCertificate* method of a *Signature* class. *Signature* class of DigiDoc4j library is used. This object provides a standard way to access all the attributes of an X.509 verison 3 certificate. Then, the *getSubjectName* method of *X509Cert* object is called. This method returns a *String* object that contains distinguished names composed of attributes and values. For example, given the following distinguished names: *SERIALNUMBER=37610280022,GIVENNAME=RAIT,SURNAME=KALLUS,CN="KALLUS,RAIT,37610280022",OU=digital signature,O=ESTEID (MOBIIL-ID),C=EE*. In this example *C=EE* is a distinguished name, *C* is an attribute type, and *EE* is an attribute value. From the above example, it can be concluded that distinguished names are composed of attributes and values separated by symbol =. The following lines of code were used to implement this functionality.

```
X509Cert signersCertificate = signature.getSigningCertificate();
String subjectName = signersCertificate.getSubjectName();
```

Only values of the data related to signer must be inserted into the database, there is no need for attributes. To get the values, the String object is split and unnecessary elements are filtered out. Then, each received value is stored in a separate variable, the value of which will be inserted into the database. Following is the example of how the signer's serial number value was extracted and stored in a seperate *String* object.

```
List<String> subjectTokens = Arrays.asList(subjectName.split(", "));
String serialNumber = "";

try {
    serialNumber = subjectTokens
            .stream()
            .filter(element -> element.contains("SERIALNUMBER=") == true)
            .collect(Collectors.toList()).get(0)
            .split("=")[1];
} catch (IndexOutOfBoundsException exception) {
    logger.warn("Serial number is not specified!");
}
```

The remaining values were extracted in the same way. From the above lines of code, it can be seen that the *warn* method of the *Logger* object is called. In this case, it is used to warn that the signer serial number was not indicated in the *subject* field.

3.4.5 Signer's certificate data extraction

The data related to the signer's certificate was extracted from different fields of *TBSCertificate* ASN.1 structure. Certificate validity data was extracted from the *validity* field of *TBSCertificate* ASN.1 structure. The *validity* field is defined by *Validity* ASN.1 structure. In Figure 14, the *Validity* ASN.1 structure can be seen.

```
Validity ::= SEQUENCE {
        notBefore      Time,
        notAfter       Time }
```

Figure 14. *Validity* ASN.1 structure [6].

As can be seen in Figure 14, the *Validity* ASN.1 structure consists of two fields: *notBefore* and *notAfter*. The *notBefore* field defines the date on which the certificate validity period begins. The *notAfter* field defines the date on which the certificate validity period ends. Both fields are encoded as UTC time[14]. Values that were obtained from these fields were inserted into the database.

Certificate issuer data was extracted from the *issuer* field of the *TBSCertificate* ASN.1 structure. The *issuer* field is defined by the *Name* ASN.1 structure. In Figure 15, the *Name* ASN.1 structure can be seen.

```
Name ::= CHOICE { -- only one possibility for now --
     rdnSequence  RDNSequence }
```

Figure 15. *Name* ASN.1 structure [6]

[14] https://www.obj-sys.com/asn1tutorial/node15.html

As can be seen in Figure 15, the *Name* ASN.1 structure consists of *rdnSequence* field. This field specifies distinguished names composed of attributes and values. Only values that were obtained from the *CN* attribute were inserted into the database.

Certificate public key data was extracted from the *subjectPublicKeyInfo* field of *TBSCertificate* ASN.1 structure. The *subjectPublicKeyInfo* field is defined by *SubjectPublicKeyInfo* ASN.1 structure. In Figure 16, the *SubjectPublicKeyInfo* ASN.1 structure can be seen.

Figure 16. *SubjectPublicKeyInfo* ASN.1 structure [6]

As can be seen from Figure 16, the *SubjectPublicKeyInfo* ASN.1 structure consists of two fields: *algorithm* and *subjectPublicKey*. The *algorithm* field specifies the algorithm with which the key is used. The *subjectPublicKey* field contains the signer's public key. Only values that were obtained from the *algorithm* field were inserted into the database.

The following lines of code were executed to extract the previously mentioned data.

```
X509Certificate certificate = signersCertificate.getX509Certificate();
String publicKey = certificate.getPublicKey().getAlgorithm();
java.sql.Timestamp validNotBefore = new
java.sql.Timestamp(certificate.getNotBefore().getTime());
java.sql.Timestamp validNotAfter = new
java.sql.Timestamp(certificate.getNotAfter().getTime());
String issuerName = signersCertificate.issuerName();
List<String> issuerTokens = Arrays.asList(issuerName.split(","));

String signersCertificateIssuer = "";

try {
    signersCertificateIssuer = issuerTokens
            .stream()
            .filter(element -> element.contains("CN=") == true)
            .collect(Collectors.toList()).get(0)
            .split("=")[1];
} catch (IndexOutOfBoundsException exception) {
    logger.warn("Signer's certificate issuer is not specified!");
}
```

As can be seen from the above lines of code, an *X509Certificate* object is first created by calling a *getX509Certificate* method provided by DigiDoc4j library. This object provides a standard way to access all the attributes of an X.509 version 3 certificate. Then, the *getAlgorithm* method of *Key* interface provided by *java.security* package is called. This method returns the standard algorithm name of the public key contained in the certificate. For example, *RSA* would indicate that the key is an RSA key. In addition, the data related to certificate validity is extracted by calling *getNotBefore* and *getNotAfter* methods of *X509Certificate* object. In the end, the certificate issuer data is extracted by calling the *issuerName* method provided by DigiDoc4j library. This method returns a *String* object that contains disinguished name of certificate issuer composed of attributes and values.

From the above lines of code, it can be seen that the *warn* method of the *Logger* object is called. In this case, it is used to warn that the signer's certificate issuer is not indicated in the *issuer* field.

3.4.6 OCSP data extraction

As mentioned earlier, the *OCSPResponse* ASN.1 structure consists of two fields: *responseStatus* and *responseBytes*. The data related to OCSP response creation time is contained in the *responseBytes* field. To be more precise, this data is contained in the *producedAt* field of *ResponseData* ASN.1 structure. In Figure 17, the *ResponseData* ASN.1 structure can be seen.

```
ResponseData ::= SEQUENCE {
   version             [0] EXPLICIT Version DEFAULT v1,
   responderID             ResponderID,
   producedAt              GeneralizedTime,
   responses               SEQUENCE OF SingleResponse,
   responseExtensions  [1] EXPLICIT Extensions OPTIONAL }
```

Figure 17. *ResponseData* ASN.1 structure [10].

As can be seen from Figure 17, the *ResponseData* ASN.1 structure consists of five fields. The *producedAt* field specifies the time at which this response was signed (OCSP response

creation time). Only values that were obtained from *producedAt* field were inserted into the database thus the meaning of other fields will not be given in this thesis.

The data related to the OCSP response issuer and OCSP certificate issuer is contained in the OCSP X.509 version 3 certificate. To be more precise, this data is contained in the *TBSCertificate* ASN.1 structure. The *subject* field of *TBSCertificate* ASN.1 structure contains an OCSP responder name that issued OCSP response. The *issuer* field of *TBSCertificate* ASN.1 structure contains a CA name that issued a certificate to OCSP responder.

The following lines of code were executed to extract the previously mentioned data.

```
java.sql.Timestamp OCSPResponseTime = new
java.sql.Timestamp(signature.getOCSPResponseCreationTime().getTime());
X509Cert OCSPCertificate = signature.getOCSPCertificate();
String OCSPSubjectName = OCSPCertificate.getSubjectName();
List<String> OCSPSubjectNameTokens = Arrays.asList(OCSPSubjectName.split(",
"));

String OCSPResponseIssuer = "";

try {
    OCSPResponseIssuer = OCSPSubjectNameTokens
            .stream()
            .filter(element -> element.contains("CN=") == true)
            .collect(Collectors.toList()).get(0)
            .split("=")[1];
} catch (IndexOutOfBoundsException exception) {
    logger.warn("OCSP response issuer is not specified!");
}

String OCSPCertificateIssuerName = OCSPCertificate.issuerName();
List<String> OCSPCertificateIssuerTokens =
Arrays.asList(OCSPCertificateIssuerName.split(","));

String OCSPCertificateIssuer = "";

try {
    OCSPCertificateIssuer = OCSPCertificateIssuerTokens
            .stream()
            .filter(element -> element.contains("CN=") == true)
            .collect(Collectors.toList()).get(0)
            .split("=")[1];
} catch (IndexOutOfBoundsException exception) {
    logger.warn("OCSP certificate issuer is not specified!");
}
```

As can be seen from the above lines of code, the OCSP response creation time is first extracted by calling a *getOCSPResponseCreationTime* method of *Signature* class. The *Signature* class of DigiDoc4j library is used. Then the *X509Cert* object is created by calling a

getOCSPCertificate method of the same class. This object provides a standard way to access all the attributes of an X.509 version 3 certificate. In order to get data related to the OCSP response issuer, a *getSubjectName* method is called of *X509Cert* class. This method returns a *String* object that contains the disinguished name of the OCSP response issuer composed of attributes and values. Only values of *CN* attribute type were extracted. In the end, data related to the OCSP certificate issuer is extracted by calling a *issuerName* method of *X509Cert* class. This method returns a *String* object that contains data related to OCSP certificate issuer composed of attributes and values (example): *C=EE,O=AS Sertifitseerimiskeskus,CN=EE Certification Centre Root CA,E=pki@sk.ee*. Only values of *CN* attribute type were extracted.

In addition, from the above lines of code, it can be seen, that two *warn* methods of the *Logger* object can be called. The first *warn* method will be called if the OCSP response issuer will not be specified in the *subject* field. The second *warn* method will be called if the OCSP certificate issuer will not be specified in the *issuer* field.

3.4.7 Timestamp data extraction

As mentioned earlier, the *TimeStampResp* ASN.1 structure consists of two fields: *status* and *timeStampToken*. The data related to timestamp creation time is contained in the *timeStampToken* field. To be more precise, this data is contained in the *genTime* field of *TSTInfo* ASN.1 structure. In Figure 18, the *TSTInfo* ASN.1 structure can be seen.

```
TSTInfo ::= SEQUENCE  {
   version                      INTEGER  { v1(1) },
   policy                       TSAPolicyId,
   messageImprint               MessageImprint,
     -- MUST have the same value as the similar field in
     -- TimeStampReq
   serialNumber                 INTEGER,
    -- Time-Stamping users MUST be ready to accommodate integers
    -- up to 160 bits.
   genTime                      GeneralizedTime,
   accuracy                     Accuracy                 OPTIONAL,
   ordering                     BOOLEAN              DEFAULT FALSE,
   nonce                        INTEGER                  OPTIONAL,
     -- MUST be present if the similar field was present
     -- in TimeStampReq.  In that case it MUST have the same value.
   tsa                          [0] GeneralName          OPTIONAL,
   extensions                   [1] IMPLICIT Extensions   OPTIONAL  }
```

Figure 18. *TSTInfo* ASN.1 structure [12].

As can be seen from Figure 18, the *TSTInfo* ASN.1 structure consists of ten fields. The *genTime* field specifies the time at which the time-stamp token has been created by the TSA. Only values that were obtained from *genTime* field were inserted into the database thus the meaning of other fields will not be given in this thesis.

The data related to the timestamp issuer and timestamp certificate issuer is contained in the timestamp X.509 version 3 certificate. To be more precise, this data is contained in the *TBSCertificate* ASN.1 structure. The *subject* field of *TBSCertificate* ASN.1 structure contains a TSA name that issued timestamp. The *issuer* field of *TBSCertificate* ASN.1 structure contains a CA name that issued a certificate to TSA.

The following lines of code were executed to extract the timestamp data.

```
java.sql.Timestamp timestampCreationTime = new
java.sql.Timestamp(signature.getTimeStampCreationTime().getTime());
X509Cert timestampTokenCertificate =
signature.getTimeStampTokenCertificate();
String timestampIssuer = "";

try {
    timestampIssuer =
Arrays.asList(timestampTokenCertificate.getSubjectName().split(", "))
            .stream()
            .filter(element -> element.contains("CN=") == true)
            .collect(Collectors.toList()).get(0)
```

```
            .split("=")[1];
} catch (IndexOutOfBoundsException exception) {
    logger.warn("Timestamp issuer is not specified!");
}

String timestampCertificateIssuer = "";

try {
    timestampCertificateIssuer =
Arrays.asList(timestampTokenCertificate.issuerName().split(","))
            .stream()
            .filter(element -> element.contains("CN=") == true)
            .collect(Collectors.toList()).get(0)
            .split("=")[1];
} catch (IndexOutOfBoundsException exception) {
    logger.warn("Timestamp certificate issuer is not specified!");
}
```

As can be seen from the above lines of code, the timestamp creation time is first extracted by calling a *getTimeStampCreationTime* method of *Signature* class. The *Signature* class of DigiDoc4j library was used. This method returns the signature timestamp generation time. Then the *X509Cert* object is created by calling a *getTimeStampTokenCertificate* method of the same class. This object provides a standard way to access all the attributes of an X.509 version 3 certificate. In order to get data related to the timestamp issuer, a *getSubjectName* method is called of *X509Cert* class. This method returns a *String* object that contains the disinguished name of the timestamp issuer composed of attributes and values. Only values of the *CN* attribute were extracted. In the end, data related to timestamp certificate issuer is extracted by calling an *issuerName* method of *X509Cert* class. This method returns a *String* object that contains the disinguished name of the timestamp certificate issuer composed of attributes and values. Only values of the *CN* attribute were extracted.

In addition, from the above lines of code, it can be seen, that two *warn* methods of the *Logger* object can be called. The first *warn* method will be called if the timestamp issuer will not be specified in the *subject* field. The second *warn* method will be called if the timestamp certificate issuer will not be specified in the *issuer* field.

3.4.8 Interaction with the database

First, in order to be able to insert data into the database, a connection should be established with it. The following *connect* method was used in order to establish a connection with the database.

```
public Connection connect() {
    connection = null;
    try {
        connection = DriverManager.getConnection(url, user, password);
    } catch (SQLException e) {
        System.out.println(e.getMessage());
    }

    return connection;
}
```

As can be seen, inside this method the *getConnection* method is called to establish a connection to the given database Uniform Resource Locator. *getConnection* method of *DriverManager* class provided by *java.sql* package was used. This method takes three arguments: *url*, *user*, and *password*. The *url* argument specifies a database *url.* The *user* argument specifies the database user on whose behalf the connection is being made. The *password* argument specifies the user's password.

After the connection with the database is established, the data can be inserted into it. In order to insert data related to signer's certificate the following *insertCertificate* method was used.

```
public void insertCertificate(Certificate certificate) throws SQLException
{
    String SQL = "insert into certificate (signer_id, public_key,
valid_not_before, valid_not_after, certificate_issuer)" + "values (?, ?, ?,
?,                                                                    ?)";

    PreparedStatement preparedStatement = connection.prepareStatement(SQL);
    preparedStatement.setInt(1,              certificate.getSignatureID());
    preparedStatement.setString(2,             certificate.getPublicKey());
    preparedStatement.setTimestamp(3,     certificate.getValidNotBefore());
    preparedStatement.setTimestamp(4,      certificate.getValidNotAfter());
    preparedStatement.setString(5,
certificate.getSignersCertificateIssuer());
    preparedStatement.executeUpdate();
}
```

As can be seen, inside this method a *String* object is created that represents a PostgreSQL *INSERT* statement. This type of statement allows inserting one or more rows into a table at a time. Then an object of *PreparedStatement* class provided by *java.sql* package is created that represents a precompiled SQL statement. This object can then be used to efficiently

execute SQL statements multiple times. *PreparedStatement* object was chosen over the *Statement* object because in case of *PreparedStatement* object query is rewritten and compiled by the database server. Thus, a query created by using *PreparedStatement* leads to faster execution and the ability to reuse the same SQL statement. In addition, the use of *PreparedStatement* object can prevent SQL injection, by escaping text for all the parameter values provided.

It is worth noting that the rest of the data that were extracted from the digital signature file formats were inserted into the database in the same way. Therefore, examples of how the rest of the data were inserted into the database will not be presented in this thesis. For the full source code of a program, see Appendix II.

After all the data is inserted into the database, the connection with the database should be closed and all the resources should be released (cursors, handles, etc). In order to achieve this, the *close* method of *Connection* class provided by *java.sql* package was us.

4 Outcome

As an outcome of this thesis, a database was created. The created database stores data that is contained in collected DDOC and BDOC file formats. It is worth noting that the database contains only data that, according to the author, is important when it comes to analyzing digital signature file formats. In more detail, the created database contains the following data:

- data describing a person, who signed the document;
- data related to signer's certificate;
- data related to certificate validity;
- data related to the timestamp that was given after signing a document.

The DDOC and BDOC file formats define the XML syntax for digital signatures and thus represent the XML Signature files. It can be seen from the above-presented list that the database stores only data that is associated with the XML Signature files. For example, the data that is contained in the *mimetype* file of the BDOC file format is not stored in the database, since it only indicates the type of file format and can hardly be of interest. For the same reason, the database does not store data that is contained in the *manifest.xml* file of the BDOC file format.

A situation may arise when a person wants to analyze data that is not in the database. In this case, small changes have to be made to the created program, and the table structure should be changed. However, this will not take much time, since other data can be extracted from DDOC and BDOC file formats in a similar way.

The database is created for people who are interested in digital signatures or digital signature file formats used in Estonia. The database can be used to analyze the data stored in it. Also, using the created program, it is possible to insert new data into the database. If there is a need to change the data contained in the database, database queries can be used.

5 Conclusion

The bachelor's thesis described what a digital signature file format is. The terms associated with digital signature file format have also been described: digital signature, public key certificate, Online Certificate Status Protocol, and Time-Stamp Protocol. In Estonia, two digital signature file formats are used: DDOC and BDOC. The thesis described the specification of these file formats.

The goal of this thesis was to create a database in which the data contained in the DDOC and BDOC digital signature file formats will be stored. To achieve this goal, two processes have been done: data collection and database creation.

The data collection process was done first. In order to collect the data, it was necessary to choose a document register from which data could be downloaded. The Tartu town document register was chosen. Since it would take a lot of time to manually download data from the document register, a program was created. The program was created using Selenium and Junit libraries.

The database creation process was done at the end. In order to create the database, it was necessary to choose a database management system. The PostgreSQL relational database management system was chosen for this purpose. In addition, it was necessary to write a program, which would extract data from collected digital signature file formats and insert this data into the database. The program was created using the DigiDoc4j library.

As an outcome of this thesis, a database was created. The created database stores data that is contained in collected DDOC and BDOC file formats. The database is created for people who are interested in digital signatures or digital signature file formats used in Estonia. In the created database, all data is stored in decrypted form, so it is convenient to analyze it. In the future, the database can be used to conduct statistical analysis or detect some digital signature file format non-compliances.

References

[1] DigiDoc tarkvara. 2019. https://www.ria.ee/et/riigi-infosusteem/eid/digidoc tarkvara.html (30.11.2019).

[2] File Format. 2011. https://techterms.com/definition/file_format (30.11.2019).

[3] Rouse M. digital signature. 2019. https://searchsecurity.techtarget.com/ definition/digital-signature (30.11.2019).

[4] Rouse M. public key certificate. 2007. https://searchsecurity.techtarget.com/ definition/public-key-certificate (30.11.2019).

[5] Russell A. What is a Certificate Authority (CA)? 2019. https://www.ssl.com/faqs/ what-is-a-certificate-authority/ (30.11.2019).

[6] Cooper D., Santesson S., Farell S., et al. Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile. 2008, no. 5280, pp. 10-26. https://tools.ietf.org/html/rfc5280 (17.12.2019).

[7] Digital Certificates Explained. https://sites.google.com/site/amitsciscozone/home/ security/digital-certificates-explained (17.12.2019).

[8] Steedman D. E.1 What is ASN.1? https://archive.bgbm.org/TDWG/acc/Documents/ asn1gloss.htm (17.12.2019).

[9] Sertifitseerimiskeskuse OCSP-teenus SK-OCSP. https://www.sk.ee/upload/files/ kehtivuskinnituse_tech_kirjeldus.pdf (26.12.2019).

[10] Santesson S., Myers M., Ankney R., et al. X.509 Internet Public Key Infrastructure Online Certificate Status Protocol – OCSP. 2013, no. 6960, pp. 11-18. https://tools.ietf.org/html/rfc6960 (26.12.2019).

[11] Trusted timestamping. http://www.ssl4net.com/technology/trusted-timestamping (04.01.2020).

[12] Adams C., Cain P., Pinkas D., Zuccherato R. Internet X.509 Public Key Infrastructure Time-Stamp Protocol (TSP). 2001, no. 3161, pp. 1-9. https://tools.ietf.org/html/rfc3161 (04.01.2020).

[13] Kleiner F.S., Mamiya C.J. Gardner's Art through the Ages: The Western Perspective. *Mesopotamian Seals*, 2004, vol. 1, p. 37.

[14] DigiDoc formaadi kirjeldus. 2004, no. 1.3.2. https://id.ee/public/ DigiDoci_vorming_1.3.2.pdf (28.01.2020).

[15] DigiDoc file formats: DDOC, BDOC, CDOC, 2014, no. 35780. https://www.id.ee/ index.php?id=35780 (28.01.2020).

[16] Eastlake D., Reagle J., Solo D. (Extensible Markup Language) XML-Signature Syntax and Processing. 2002, no. 3275, pp. 15-44. https://www.ietf.org/rfc/rfc3275.txt (28.01.2020).

[17] BDOC – FORMAT FOR DIGITAL SIGNATURES. 2014, no. 2.1.2, pp. 2-17. https://www.id.ee/public/bdoc-spec212-eng.pdf (28.01.2020).

[18] BDOC file format, what is it, when will it replace DDOC format and whats needed for transition?2015, no. 34336. https://www.id.ee/?id=34336 (28.01.2020).

[19] Durusau P., Hamilton D., Brauer M., Oracle Corporations, editors. Open Document Format for Office Applications (OpenDocument). 2011, no. 1.2. http://docs.oasis-open.org/office/v1.2/os/OpenDocument-v1.2-os part3.html# RefHeading 752849_826425813 (28.01.2020).

[20] What is MIME (Multi-Purpose Internet Mail Extensions). https://www.interserver.net/ tips/kb/mime-multi-purpose-internet-mail-extensions/ (28.01.2020).

[21] What's the difference between the digital signature formats .ddoc, .bdoc and .asice?. 2019, no. 37370. https://www.id.ee/index.php?id=37370 (28.01.2020).

[22] TARTU LINNA DOKUMENDIREGISTER. https://www.tartu.ee/ru/dokumendid (25.02.2020).

[23] The Selenium Browser Automation Project. https://www.selenium.dev/ documentation/en/ (25.02.2020).

[24] Neha V. All You Need to Know About Selenium WebDriver Architecture. 2019, no. 8. https://www.edureka.co/blog/selenium-webdriver-architecture/ (25.02.2020).

[25] Selenium WebDriver. https://www.javatpoint.com/selenium-webdriver (25.02.2020).

[26] Siminiuc A. How Does Selenium WebDriver work. 2015. https://www.linkedin.com/pulse/how-does-selenium-webdriver-work-alex-siminiuc (25.02.2020).

[27] JUnit 4. 2020, no. 4.13. https://junit.org/junit4/ (25.02.2020).

[28] Guernsey M. Test-Driven Database Development: Unlocking Agility (Net Objectives Lean-Agile Series) (1st ed.). *What is TDD?* 2013.

[29] JUnit - Test Framework. https://www.tutorialspoint.com/junit/junit_test_framework.htm (25.02.2020).

[30] BDOC failiformaat, mis see on, kuna sellele üle minnakse ning mida selleks tegema peab. 2015, no. 34070. https://www.id.ee/?id=34070 (03.03.2020).

[31] DBMS. https://techterms.com/definition/dbms (07.03.2020).

[32] RDBMS. https://techterms.com/definition/rdbms (07.03.2020).

[33] Gurgel F. Managing PostgreSQL backup and replication for very large databases. 2018. https://medium.com/leboncoin-engineering-blog/managing-postgresql-backup-and-replication-for-very-large-databases-61fb36e815a0 (07.03.2020).

[34] What is PostgreSQL? https://stackshare.io/postgresql#description (07.03.2020).

[35] Chapple M. One-to-One Relationships. 2020. https://www.lifewire.com/one-to-one-relationships-1019757 (07.03.2020).

[36] DigiDoc teegid - Java teek - digidoc4j (ver 3.3.0). 2019, no. 36870. https://www.id.ee/?id=36870 (14.03.2020).

[37] DigiDoc4j 4.0.0 API. http://open-eid.github.io/digidoc4j/ (14.03.2020).

[38] DigiDoc4j. https://github.com/open-eid/digidoc4j (14.03.2020).

[39] JDBC - Sample, Example Code. https://www.tutorialspoint.com/jdbc/jdbc-sample-code.htm (14.03.2020).

[40] JDBC Architecture. https://www.educba.com/jdbc-architecture/ (14.03.2020).

Appendix

I. DDOC file format structure

Tabel 1. DDOC file format structure.

DDOC element	Element meaning
</SignedDoc>	Root element of DDOC file format. It contains attributes related to file format name, file format version and namespace used for providing uniquely named elements and attributes in an XML document.
</DataFile>	This element contains one or more original data files or references to external files. Moreover, it has unique file identifier which begin with the character *D*, file name, document encapsulation method, data type of original data, size of original data file in bytes and other data.
</Signature>	This element contains any number of signatures.
</SignedInfo>	This element contains the data to be signed.
</CanonicalizationMethod>	This element specifies the canonicalization algorithm and this algorithm is applied to the *</SignedInfo>* element before performing signature calculations.
</SignatureMethod>	This element specifies the algorithm used for signature generation and validation.
</Reference>	This element specifies a digest algorithm and digest value, and optionally an identifier of the object being signed, the type of the object, and a list of transforms to be applied prior to digesting.

</DigestMethod>	This element specifies the digest algorithm to be applied to the signed object.
</DigestValue>	This element specifies the encoded value of the digest. The digest is encoded using base64 binary-to-text encoding.
</SignatureValue>	This element specifies the actual value of the digital signature. It is encoded using base64 binary-to-text encoding.
</KeyInfo>	This element specifies the certificate used to give the signature and its RSA public key.
</X509Data>	This element specifies one or more identifiers of keys or X509 certificates.
</X509Certificate>	This element specifies base64-encoded certificate.
</Object>	This element contains *</QualifyingProperties>* element.
</QualifyingProperties>	This element contains </*SignedProperties*> and *</UnsignedProperties> elements*.
</SignedProperties>	This element specifies additional data to be included in the signature: signing time, info about certificate used to give the signature, signature policy, place of signing and signer role.
</UnsignedProperties>	This element specifies unsigned data: validity conformation server certificate info, validity conformation info, validity confirmation server (OCSP responder) certificate and actual validity confirmation.

II. BDOC XML Signature file structure without validation data

Table 2. BDOC XML Signature file structure without validation data.

BDOC element	Element meaning
</ds:Signature>	Root element of BDOC file format. It contains attribute related to signature id. For example, *Id="S0".*
</ds:SignedInfo>	This element contains two or more *</ds:Reference>* elements. These *</ds:Reference>* elements contain one reference to every document in the file format to be signed and one reference to *</xades:SignedProperties>* element.
</ds:CanonicalizationMethod>	This element specifies the canonicalization algorithm and this algorithm is applied to the *</SignedInfo>* element before performing signature calculations.
</ds:SignatureMethod>	This element specifies the algorithm used for signature generation and validation.
</ds:Reference>	This element specifies a digest algorithm and digest value. It contains *</DigestMethod>* and *</DigestValue>* elements.
</ds:SignatureValue>	This element specifies the actual value of the digital signature. It is encoded using base64 binary-to-text encoding.
</ds:KeyInfo>	This element specifies the certificate used to give the signature and its RSA public key.
</X509Data>	This element specifies one or more identifiers of keys or X509 certificates.
</X509Certificate>	This element specifies base64-encoded certificate.

III. Source code of the created programs

https://github.com/AralovArtur/thesis

IV. Collected data

https://drive.google.com/open?id=1RG0KNxvp0xFhlMtkRqFu4ydKv_zxbnXY

V. SQL queries for database table structure creation

```
CREATE TABLE public.Signer
(
    id serial,
    serial_number character varying NOT NULL,
    given_name character varying NOT NULL,
    surname character varying NOT NULL,
    eID_tool character varying NOT NULL,
    CONSTRAINT signer_pkey PRIMARY KEY (id)
)

CREATE TABLE public.Certificate
(
    id serial,
    signer_id integer NOT NULL,
    public_key character varying NOT NULL,
    valid_not_before character varying NOT NULL,
    valid_not_after character varying NOT NULL,
    certificate_issuer character varying NOT NULL,
    CONSTRAINT certificate_pkey PRIMARY KEY (id),
    CONSTRAINT certificate_signer_id_fkey FOREIGN KEY (signer_id)
      REFERENCES signer (id) MATCH SIMPLE
      ON UPDATE NO ACTION ON DELETE NO ACTION
)

CREATE TABLE public.OCSP
(
    ocsp_id serial,
    signer_id integer NOT NULL,
    response_time character varying NOT NULL,
    response_issuer character varying NOT NULL,
    certificate_issuer character varying NOT NULL,
    CONSTRAINT ocsp_pkey PRIMARY KEY (ocsp_id),
    CONSTRAINT ocsp_signer_id_fkey FOREIGN KEY (signer_id)
      REFERENCES signer (id) MATCH SIMPLE
      ON UPDATE NO ACTION ON DELETE NO ACTION
)

CREATE TABLE public.Timestamp
(
    timestamp_id serial,
    signer_id integer NOT NULL,
    creation_time character varying NOT NULL,
    timestamp_issuer character varying NOT NULL,
    certificate_issuer character varying NOT NULL,
    CONSTRAINT timestamp_pkey PRIMARY KEY (timestamp_id),
    CONSTRAINT timestamp_signer_id_fkey FOREIGN KEY (signer_id)
      REFERENCES signer (id) MATCH SIMPLE
      ON UPDATE NO ACTION ON DELETE NO ACTION
```

VI. Licence

Non-exclusive licence to reproduce thesis and make thesis public

I, Artur Aralov,

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,

   Database of digital signature file formats used in Estonia,
   supervised by Arnis Paršovs

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.

Artur Aralov

08/05/2020