

UNIVERSITY OF TARTU
Institute of Computer Science
Computer Science Curriculum

Mart Simisker

Security of Health Information Databases

Master's Thesis (30 ECTS)

Supervisor: Jan Willemsen, PhD
Co-supervisor: Dominique Unruh, PhD

Tartu 2019

Security of Health Information Databases

Abstract:

Secure storage of sensitive data is a strong requirement in current times. Depending on the scenario it could prove more difficult than first expected. Data security on the database side is often overlooked or underestimated. Application side encryption can be used to avoid many of the common issues. In the thesis we aim to give an implementation of one scheme for secure data gathering and storage. The implementation consists of three applications to display the process of gathering data. We also attempt to integrate two low budget Hardware Security Modules (HSMs) into our scheme. The thesis shows the difficulties with the hope, that the process could be improved. The given example can be used to add specialised sensitive data collection methods to existing data management software.

Keywords:

Storage of sensitive data, database security, HSM, application side encryption

CERCS: P170: Computer science, numerical analysis, systems, control

Tervise infosüsteemide andmebaaside turvalisus

Lühikokkuvõte:

Tundlike andmete turvaline kogumine ja hoiustamine on väga vajalik. Olenevalt olukorrast võib see osutuda aga oodatust keerulisemaks. Andmebaasis olevate andmete turvalisus võib jääda tähelepanuta või seda võidakse ülehinnata. Rakenduse poolel andmete krüpteerimine on üks moodus laialdaselt esinevate probleemide ennetamiseks. Selle töö eesmärk on esitada näidisrakendus andmete turvalise kogumise kohta. See implementatsioon esitab andmete kogumise protsessi. Me katsetame kahte odavama hinnaklassi riistvaralisi turvamoodulit rakendusega siduda. Tulemustest on näha kaasnevaid raskusi, lootusega et protsessi saab parendada. Näidisrakendust saab kasutada tundlike andmete kogumise meetodite lisamisel olemasolevatesse andmehaldusrakendustes.

Võtmesõnad:

Tundlike andmete hoiustamine, andmebaaside turvalisus, riistvaraline turvamoodul, rakenduse poolne krüpteerimine

CERCS: P170: Arvutiteadus, arvutusmeetodid, süsteemid, juhtimine (automaatjuhtimisteooria)

Contents

| | | |
|----------|---|-----------|
| 1 | Introduction | 5 |
| 2 | Overview of security topics and use in software | 6 |
| 2.1 | Certificates | 6 |
| 2.2 | OpenSSL | 6 |
| 2.3 | Web of Trust and GnuPGP | 7 |
| 2.3.1 | Subkeys | 7 |
| 2.4 | Security in PostgreSQL | 7 |
| 2.5 | Java and cryptography | 8 |
| 2.5.1 | Sun PKCS#11 | 8 |
| 2.5.2 | Bouncy Castle | 9 |
| 2.6 | Hybrid encryption | 9 |
| 3 | Secure data storage | 10 |
| 3.1 | Potential attacks | 10 |
| 3.2 | Secure data storage structures | 11 |
| 3.3 | Comparison of different encryption levels | 13 |
| 4 | Secure key storage | 15 |
| 4.1 | Key storage schemes | 15 |
| 4.2 | Security comparison of the key storage schemes | 17 |
| 5 | Hardware Security Modules | 20 |
| 5.1 | Motivation | 20 |
| 5.2 | Comparison of different HSMs | 21 |
| 5.3 | About OpenPGP | 21 |
| 5.4 | Security of HSMs | 22 |
| 5.5 | Integration of HSMs | 22 |
| 5.5.1 | Integration of Nitrokey Start to the application | 22 |
| 5.5.2 | Integration of Nitrokey HSM to the application | 25 |
| 6 | The application | 29 |
| 6.1 | Architecture | 30 |
| 6.1.1 | Defining choices | 30 |
| 6.1.2 | Functionality description | 31 |
| 6.1.3 | Technical description of database | 33 |
| 6.1.4 | Technical description of chosen cryptographic functions | 34 |
| 6.1.5 | Technical description of Pre database application | 34 |
| 6.1.6 | Technical description of Data access application | 35 |

| | | |
|----------|--|-----------|
| 6.2 | Analysts certificate | 36 |
| 6.3 | Technologies | 36 |
| 6.4 | Validation and testing | 37 |
| 6.5 | Views | 37 |
| 6.6 | Performance and reliability | 39 |
| 7 | Conclusion | 42 |
| 7.1 | A look back at the choices | 42 |
| 7.1.1 | The HSMs | 42 |
| 7.1.2 | The application architecture | 43 |
| 7.2 | Future work | 43 |
| | References | 44 |
| | Appendix | 49 |
| | I. Licence | 49 |

1 Introduction

The operation of gathering data with the aim of analysis or for the necessity of later use has been around for a long time. In current times some forms of data are considered sensitive and the leakage of such data can lead to severe consequences. As such, it is important to have means of preventing unauthorised access to such data. Often the first level of access control is set up, such that the channel over which the data is moved is protected. Also the client entering the data is often times protected. On the other hand, what happens to the data on the server side and how it is stored over time can be easily overlooked, as people may have overconfidence in the security of the database or the access control of their system.

We consider a case, where the adversary might gain access to the database. The first solution is to encrypt data and thereby ensure the safety against all leakage. Such adversaries have been considered previously for example by Jingmin and Min in [JW01], where the database administrator was considered as an adversary. They also suggested encrypting sensitive data at the database level on insertion. The thesis provides an implementation of a different solution, where the encryption and decryption of the data are done on an application layer prior to insertion into the database.

The thesis is organised as follows. In section 2 we give an overview of some security software used in the process of data gathering. In section 3 we discuss different methods of secure data storage, more closely in section 3.1 we look at some potential attacks on the database side, then in section 3.2 we discuss secure data storage structures, and finally, in section 3.3 we compare different encryption levels. In section 4 we look at secure storage of cryptographic keys. Two main key storage formats which will be covered are HSM and plaintext keys. We name key storage schemes defined by the layer in the application structure on which the key is stored, additionally we take into consideration the format in which the key stored. We discuss the difference in security and accessibility of different key storage schemes.

In section 5 we look at HSMs generally, compare some models and look at a few security problems related to HSMs. We also describe the integration of the HSMs into a Java application. In section 6 we present the implementation – its requirements, architecture, the used technologies and how we validated the outcome.

We first look at different ways of database encryption and compare them. The main aim is to provide a guide for secure storage of sensitive data. We provide an example application to display one secure method of data gathering for later access. We also experiment with some low budget HSMs and try to integrate them into the application.

Through the work a role analyst will be mentioned. This role should be interpreted as a user that has access rights to all data.

The thesis was written as part of the project “Software Technology and Applications Competence Centre (STACC)” (Project number: EU48684) and was co-funded by European Regional Development Fund [STAb].

2 Overview of security topics and use in software

In the following section, we will look at some of the more established means or solutions of security. The solutions are related to public-key cryptography, an important ingredient in modern cryptosystems.

2.1 Certificates

According to [pub] a certificate is an electronic document, which includes information about the cryptographic key, information about the identity of its owner, and a digital signature from the issuer of the certificate. A certificate follows a format, the most common being X.509. The X.509 certificate standard is described in RFC 5280 [CSF⁺08]. The two types of X.509 certificates of interest to us are the *certificate authority* (CA) certificate and end-entity certificate. A CA certificate is used for issuing certificates, which is done by signing certificate signing requests. An end-entity certificate has not been authorised to issue other certificates.

In the context of this thesis, we will be using RSA public keys [KMKJR16], however the results can also be achieved with elliptic curve cryptography [LD00].

The process of constructing public key infrastructure (PKI) certification paths is guided by the following document [CDH⁺05]. The structure of a PKI scheme is heavily dependent on the application model and its design should be thoroughly considered. Although many complex structures can be constructed, we will be using a simple two level hierarchy with one CA certificate that signs all end-entity certificates.

The operation of verifying the validity of a certificate against the issuing CA certificate, also described in RFC 5280 [CSF⁺08, 3.2], is done by validating the contained digital signature (created by the issuer) using the issuer's certificate which is obtained from a different source. The operation can have many steps, as there could be a chain of certificates from the CA certificate to the final end-entity certificate containing intermediate certificates.

2.2 OpenSSL

OpenSSL [opeb] is a widely used open source general-purpose cryptography library, and Transport Layer Security and Secure Socket Layer protocol toolkit. The software can be used for generating key pairs, running cryptographic operations such as signing and encryption/decryption and more. Multiple online guides use this software.

One useful part of OpenSSL is the engine module. It is a cryptographic engine [opec], a collection of implementations of cryptographic commands. The engine can also be provided with a third-party module, in which case the engine can pass some parts of the commands to said module. The engines are typically used to support specialised hardware. One example where this is done, is a PKCS#11 engine with OpenSC_pkcs#11

module [Oped] – a combination for using some smart-cards and hardware cryptographic modules.

2.3 Web of Trust and GnuPGP

An alternative to PKI schemes is the web of trust scheme [web]. As opposed to PKI containing a hierarchy of trust authorities, in the web of trust, the trust scheme is decentralised. The aim of this is to provide a more flexible scheme.

According to Chapter 3 in The GNU Privacy Handbook [ACGW99, Chapter 3], in the web of trust model, the responsibility for validating public keys is delegated to people you trust. Trust for a key is categorized into 4 levels, these are *unknown*, *none*, *marginal* and *full*. A key is considered valid if it is signed by enough valid keys

- by the validating user
- by one fully trusted key
- by three marginally trusted keys,

and the path of signed keys leading from the key back to the validating key is five steps or shorter. Therefore the web of trust is a network of 'trusts' (signatures) between keys and validating a key is a path finding operation (on a directed graph with special rules).

2.3.1 Subkeys

In Chapter 4 The GNU Privacy Handbook [ACGW99, Chapter 4], a user would usually have a signing key with no expiration date and an encryption subkey, a key which has been signed by the master key. The compromise of an encryption key is seen as a high problem in this case, therefore they suggest using expiration date for an encryption key. Before the expiration, a new subkey is generated, signed and the public key is updated. The idea behind this is to limit the amount of encrypted documents which could be compromised in case of the compromise of an encryption key.

2.4 Security in PostgreSQL

The following will cover security options in one widely used relational database management system (DBMS) PostgreSQL [posc]. This DBMS was created in the 90s and has since been updated and improved. It is also open sourced enabling developers access to the source code to either verify the software or make improvements.

According to the guide [posb], PostgreSQL supports encryption on multiple levels.

- Encryption for specific columns – By providing a key with a request, an internal module called pgcrypto can be used to run encryption/decryption operations on the DBMS side.

- Data partition encryption – this is a method for securing data on either the file system level, or block level. This means that the files containing the data or the even the whole drive are encrypted. To start the DBMS, a key must be provided to access the data.
- Encrypting data across a network – by configuring the DBMS with a certificate, the connections to the DBMS will be encrypted using SSL. Alternatives to this are the use of tunnel or SSH.
- SSL host authentication – by configuring both the client and the server with a SSL certificate, additionally to providing encrypted channel, the authentication process becomes stronger.
- Client-side encryption – the method where the data is encrypted prior to sending it to the DBMS. The required support from PostgreSQL side is the potential of storing large bitstrings or other methods for data storage.

From the aforementioned possibilities we would recommend always using the SSL host authentication in case of sensitive data. Configuring has been described in [posa]. Certificate authentication assumes the presence of a PKI. A CA certificate has to be configured in the configuration file. When performing certificate authentication, the user's certificate is validated and finally, the Common Name attribute of the certificate is compared to the requested database name.

2.5 Java and cryptography

Java [Oraa] is a widely used platform and a programming language often used for writing enterprise software including web applications, mobile applications, machine learning and much more. It is object-oriented and strongly typed.

“The Java platform defines a set of programming interfaces for performing cryptographic operations. [jav]” Collectively known as the Java Cryptography Architecture (JCA) and Java Cryptography Extension (JCE), these interfaces are provider based. This means that different providers can be registered with the application, and then used for the cryptographic applications. One such provider is Sun PKCS#11 provider with the main aim of connecting native PKCS#11 cryptography API with the JCA.

Another example is Bouncy Castle Java cryptography API [otBCIa], a useful collection of tools for cryptographic operations.

2.5.1 Sun PKCS#11

The Sun PKCS#11 [jav] provider is meant for using cryptographic smartcards, hardware cryptographic accelerators, and high performance software implementations. The list

of supported algorithms [sun] contains various ciphers, signatures and secure random interface. We are interested in both the RSA keystore, the RSA decryption support and potentially Secure random generation. From the list of provided ciphers, RSA is only provided in electronic code book (ECB) mode with either PKCS1Padding or no padding. Because plain RSA is deterministic, it is not secure for practical use, if either the public key is public (enables brute force search for key) or if there are multiple encryptions with this key (potential collision). The padding makes collisions less likely, and helps against brute forcing. The problem is that PKCS1 padding is proven to be broken. First attack against PKCS#1 v. 1.5 padding was proposed in [Ble98] – a chosen ciphertext attack which required a decryption oracle. Two additional chosen plaintext attacks were proposed in [CJNP00]. This illustrates the risks of PKCS#1 v 1.5. A better choice would be to use Optimal Asymmetric Encryption Padding (OAEP) schemes, for example OAEP with sha-256 and mgf1 padding, an example provided in the Bouncy Castle implementation of RSA encryption.

The following guide [staa] shows how to work around missing paddings in the Sun PKCS#11 provider with the help of the Bouncy Castle provider. Using workarounds is a slippery road as there is a potential to make mistakes.

A note on PKCS#11 as described in [DKS08, Section 2], it is weak against malware running on the same machine. Such a scenario can reveal user PINs and allow the malware to use the PKCS#11 device on its own.

2.5.2 Bouncy Castle

The Bouncy Castle [otBCIa] cryptographic API is widely used and contains a wide range of supported algorithms [otBCIb]. Additionally there is a FIPS approved mode of operation which is useful for applications with strict requirements. There is no PKCS#11 support.

2.6 Hybrid encryption

A hybrid encryption scheme composes the speed and efficiency of a symmetric encryption scheme, with the possibility of using a shared key from public-key encryption scheme. The hybrid encryption approach was first formalized by Cramer and Shoup [Sho01, CS], and its components are called key encapsulation method *KEM* and data encapsulation method *DEM*. The *DEM* is the symmetric encryption scheme used to encrypt the message (of arbitrary length). The *KEM* is used to encrypt the key used by the *DEM*. The result of a hybrid encryption scheme consists of two ciphertexts, one containing the symmetric key and the other the message. A good explanation is also given in the introduction of [HHK06].

An example hybrid encryption scheme would be the following – the *KEM* part can be RSA with a padding scheme and the *DEM* part can be AES.

3 Secure data storage

This thesis is interested in encryption schemes, which provide secure storage for data. We will cover different adversary access levels. First we look at known attacks. Then we discuss data storage structures and compare different encryption levels while keeping in mind known attacks.

3.1 Potential attacks

We begin by describing our attacker model. We will also compare these attacks on database side in case of an application which encrypts all sensitive data on the application side.

Shmueli *et al.* [SVGE14] have categorised the attackers as intruders, insiders and administrators. We will use the same attackers, as they cover the potential dangers. As described by Shmueli *et al.* [SVGE14], the attacker can use direct and indirect storage attacks, memory attacks, passive observation resulting in leakage and active attacks of unauthorised modification of data.

Direct storage attacks involve physically stealing database drives. **Indirect storage attacks** give the attacker access to schema information and metadata, which the adversary can use to gather statistics and estimate distributions. The first attack is not relevant if the cryptographic methods are strong enough. In case of weak keys, and a scheme where only one key is used to encrypt everything, the attack would leak data. However, if different parts of the data are encrypted using different keys (satisfying the randomness requirements), then decrypting the whole data without the key (guessing) would take long enough to consider it impractical. A ciphertext does not reveal the used modes of operation. In case of block ciphers one can assume the common lengths but that still leaves multiple choices. An attacker should somehow guess the analysts private key (which we assume to be a strong RSA key and guessing it from ciphertexts would be difficult) and the used padding scheme. After which they would have to determine which symmetric encryption was used. If both of these can be broken then the rest of the data can be easily decrypted. Eventually the security relies on the strength of the analysts private key, because the padding schemes and encryption schemes used are not considered secret. The padding scheme exists to make outcomes of encrypting the same plaintext indistinguishable. Such an attack would be assisted by insider help, if there is at least one entry for which the correct plain-text is known – it would help knowing which output is correct. Indirect storage attack would be relevant in case of plain RSA or only symmetric encryption using the same key. However with correct hybrid encryption there are no patterns in the data. Only the lengths of ciphertexts vary – the only statistics is about which entries have longer ciphertexts.

In **memory attacks**, the adversary has access to the memory of the server running the database software. It is also said that the memory often contains the database cache

for optimisation reasons. Note that both the works of [AKSX04] and [SVGE14] have problems where they can not protect against memory attacks – in case of [SVGE14], the cache is encrypted but they state that there could be chances that values in plain-text are in the memory. When using the encryption on the application layer, the memory of the database does not contain plain-text values, however the application itself is still vulnerable to memory attacks.

Information leakage type attacks have been divided into **static leakage**, simply from observing snapshots of database at different times (for finding patterns), **linkage leakage**, links between database index and values, and **dynamic leakage**, the links between user actions/network activity and changes in the database. When considering these attacks against encrypted data, in case of hybrid encryption, the use of random symmetric keys with randomised padding schemes prevents static leakage by hiding patterns. Since the data is encrypted, index leakage does not leak the data. Dynamic leakage can at most reveal where the ciphertext originates from, but not its contents (assuming the steps prior to encrypting were performed over secure channels).

Unauthorised modification – active attacks are described as follows. **Spoofing**, which is replacing ciphertext values. If the public key used in creating the database entries is given, fake entries could be created given that the adversary has write access to the database. **Splicing**, the copying of ciphertext values from one place to another. Also **replay attack**, where old ciphertexts of values are reused. Both of these would work if pairs of values are copied to ensure the validity of hybrid encryption scheme. Therefore unauthorised modification must be prevented with database access control and partition encryption outside of use. A fix against splicing is using the encryption method suggested in [EWSG04, Section 3], where the cell coordinates (Table ID, Row ID, and Column ID) are also encrypted along side the value. Although, the statistics gathering problem has already been eliminated by the use of randomisation, the value substitution (splicing) was still possible and can therefore be avoided by this technique.

3.2 Secure data storage structures

Next, we will discuss different structures of secure data storage. With no security an adversary could access plaintext data, whenever an archived database dump might leak. This can be covered with transparent encryption – that means, the contents are encrypted during the backup process. Transparent encryption might also be used to protect the disk while the database is not operational by using procedures on operation system level. However, this only provides security when the system is not currently in use – it is weak against passive memory read or observation attacks.

We consider the case when an adversary has access to an in-use database environment. Encryption can be classified by granularity of logical data storage structure as follows:

- Table level (the whole) – one key per table

The whole table is encrypted as a whole. To add something, first the table must be decrypted.

- Column level (by attribute over all entries) – key per column
The table is encrypted by columns. When adding to a column, said column is first decrypted.
- Row level (by entry) – key per row
The table assumes the form of a list or an array, where elements are encrypted rows. Addition of an entry does not require decrypting the previous structure.
- Cell level (by attribute for specific entry) – key per cell in row
Every cell is encrypted separately.

Shmueli *et al.* [SVGE14, Section 3.2.] listed most common encryption granularities – these contained the cell, record and table, as well as page, but did not include the column level granularity. In [SVGE14, Section 3.3.], it was mentioned that column based encryption granularity is not common, because it would require column-oriented database architecture. An example of column based encryption granularity has been considered by Ge and Zdonik in [GZ07].

We can also classify encryption by granularity of key usage as key per table, key per column, key per row or key per cell in row.

On the application level, encryption granularities of logical structures can be constructed by using cell level encryption granularity and combining it with previous key usage granularity. This can alleviate the performance cost of adding to or modifying a large structure. In fact, the less granular encryption structures are usually managed by the DBMS (whole table encryption or column based encryption for example).

By constructing less granular structures using cell level granularity, we get:

- Table level (the whole) – one key per table
Every cell is encrypted using the same key. Provides granularity of cell level encryption without the overhead of multiple key management.
- Column level (by attribute over all entries) – key per column
Every attribute in an entry is encrypted with the attribute (column) key. Allows managing user access to a subset of attributes.
- Row level (by entry) – key per row
Every cell (attribute) in a row is encrypted using a new row key. Allows managing access control on row level.

The four structural levels provide protection against simple disk reading, with the assumption that keys are kept safe. We will write about the key management in section 4.

Shmueli *et al.* [SVGE14] have described encryption schemes, where the encryption/decryption is done on the operating system level. In one of the cases, the whole page is encrypted. In the scheme they proposed, the decryption is done upon accessing the cache. It seems however, that the data in the cache has been encrypted with a higher granularity than page level.

Often, security comes at the cost of performance. To give an example, decrypting a whole table encrypted with table level encryption takes less time than decrypting the whole table using cell level granularity. On the other hand, cell level encryption provides access control on cell level and enables creating schemes which require more keys to decrypt the whole table. In other words, cell level encryption could be used to create a more detailed access scheme with the loss of some performance. On the other hand, to access only one cell in the table, with cell level encryption, only one cell has to be decrypted as opposed to table level granularity encryption in which case the whole table would have to be decrypted.

3.3 Comparison of different encryption levels

The following is the ordered representation of different encryption levels, in the order of the number of keys per table ascending (as the number of rows exceeds the number of columns):

1. table level (fixed at one key per table)
2. column level (fixed at the number of columns in a table)
3. row level (grows by one for every entry)
4. cell level (grows by the number of columns per entry)

With each increase in the complexity, it is possible to construct more granular access control to stored data.

When using structures constructed from cell level granularity, with table level encryption, one can provide the access to the whole table. This means, that once the key is compromised or access is given, every row and column in the table become visible.

Column level encryption enables encrypting every column with a different key (or potentially grouping columns to be encrypted using the same key). This means, that providing access to one column does not compromise the data of another column thus enabling to create roles, which can access only some properties of the data.

Row level encryption means that every entry to the table is encrypted using a row key. Some rows can potentially be grouped together and encrypted with the same key. On first level access to data can be restricted on row level. Depending on the cryptosystem, this enables creation of access methods, where certain users have access to all rows and some only to their own data.

Finally, cell level encryption has the potential to provide the most granular access control. In this case, every property is encrypted independently. With hybrid encryption scheme, a potential use case would see a property encrypted for a few receivers. In the simple case, the table would contain one column, where the data is encrypted using the symmetric encryption scheme, and one column for every receiver containing the ciphertext of the symmetric key encrypted using the public key cryptosystem (or some other structure for the receivers).

Extension to multiple receivers Depending on the number of receivers, more complex encryption models can be generated, each having its benefits and negative effects. A few examples are:

- Receiver data is carried with the table (in the model)
Depending on the number of receivers, increases the number of columns (attributes) and can be seen as overhead for certain operations.
- Separating public key encryptions to a different table
Better handling for number of receivers.
Requires linking of entities and increases complexity
- Grouping users with similar roles to have access to a shared key
Every user in said group would have the shared private key encrypted for them with their public key, allowing them access to data encrypted for the shared key pair.
Less storage overhead and less key encryption operations during storage of entry. Unless the shared private key stays inside a secure environment, where access to said key is strictly controlled, there is a need to create a new shared key pair every time a user has their rights revoked from this key. Otherwise the revoked user could have stored the decrypted shared private key and upon access to new data, use the stored key to read the data.
Note: The reason that only new entries are mentioned is primarily because the user can use the stored key to decrypt entries that have already been made.

4 Secure key storage

In every cryptographic application which uses some secret keys an important question is where should the keys be stored. This question includes the location in the scheme as well as how the key should be stored. In this context, the key is the private key of an asymmetric key pair.

Depending on the threat model and enterprise policies, the solutions differ, with the use of *hardware security modules* (HSM) considered the most secure but often restrictive in the workflow and more difficult to integrate. On the other end are keys stored in plaintext files.

The following will discuss different potential solutions as to where encryption keys could be stored and the related potential consequences.

First we will compare two key storage formats – on a HSM and in a plaintext file.

We omit password-protected file-based keystores (JKS, PKCS#12 [Orab, Keytool implementation]) as they behave similarly to the plaintext file, in our use model. If, during a users normal use of the application, an adversary gains access to the file containing the users private key, due to the keystore being sent over the network, the adversary is likely to gain access to the PIN used to protect the contents as well.

We assume that the HSM is tamper resistant and has private keys generated on device with no backups and key export disallowed.

Table 1. Comparison of key storage formats

| Question | HSM | Plaintext key |
|---------------------------------|-------------------------------|-------------------------------|
| How is key access protected? | by PIN | file system access |
| Is key material protected? How? | yes (key never leaves HSM) | no only file system access |
| Can be stolen? | requires physical theft | copy (file system access) |

The Table 1 aims to compare the security of key storage methods by comparing how key access and key material are protected. It depicts how a HSM is more secure than plaintext keys which in turn affects the discussion on the security of key storage schemes.

4.1 Key storage schemes

In an application consisting of the three levels: database, intermediate application and the user (potentially behind a web client), there are three potential locations where the key could be stored. For each of these locations, the following questions need to be answered:

- How is the key access (requesting operations from the key) secured?
- How is the key material secured?

In all of the schemes we will refer to two key storage formats – the HSM and a plaintext file.

Scheme A – Key stored on the database side In this case a user would have to authenticate both to the application and to the database in order to access the data. The threat of key disclosure is reduced to the leakage of database, as the key is never sent over the network. On the other hand, the data is decrypted in the database and after that depends on the security of the channel. A problem, in which an adversary with access to the database has access to the data was also discussed by Jingmin and Ming in [JW01], where they proposed storing the key in encrypted form. Their idea also involved the use of security dictionary, for which objects stored in that area could not be modified by any other user (including the database administrator).

A.1 Plaintext key The main problem with this scheme is that if someone has access to the data, they may as well have access to the key, in which case encrypting the data becomes unnecessary. The adversary could make a copy of the database, which would include the keys, and they could just decrypt the data on another server. If the keys are in some encrypted format, then with the plaintext key setting, the key has to exist somewhere on the database machine, and with the assumption that the adversary has access to the machine, the key could leak.

A.2 Key on a dedicated device (HSM) When the key is stored on a HSM, the problem where an outsider with access to the database can instantly access the data is reduced. Depending on how much access the adversary has, they could find the stored PIN and make calls to the HSM to decrypt data while operating on the database machine. Note that this would probably leave traces. The previous attack of making a dump and decrypting elsewhere does not work in this case, as the HSM is required to decrypt the data.

A.3 Transparent operation system level encryption The other setting, when the key is stored on the database machine, is when transparent encryption on the operation system level is used. In this case all of the data is encrypted, however access is only controlled by access control rules, and not enforced by encryption. Also, on startup, the whole data is decrypted and an insider could have access to all of the data.

Scheme B – The key is stored on the application level Keeping the cryptographic key on the application level resolves the problem of keeping the key right next to the lock. The new problems which arise are user authentication and whether the key should be sent to the database when encrypting/decrypting data or should the data be encrypted/decrypted on the application level. The potential to encrypt and decrypt on the database level is provided in multiple database management systems (Postgres with pgcrypto module [posb], MySQL and Enterprise Encryption [Cor]). The problem with this is the fact that the key leaves the system, which creates the potential for key leakage. Note that the key can only leave the system, if it is stored in plaintext format. The other possibility is to run encryption/decryption procedures on the application level. This resolves the key leakage issue, however requires more computational resource from the application. The main issue with this is the requirement to implement cryptographic procedures in the application. Consequently this is prone to implementation errors (which are potentially less likely to be discovered due to the smaller user base compared to a database management system).

Scheme C – The key is stored on the user side The third option is to store the key on the user side. The three key storage formats can be used, HSMs can either be expensive or require installation of additional software. Files themselves can leak. Alternatives involve *password based key derivation function* (PBKDF). The access control of the key lies with the user. The security is also dependent on the users machine, the strength of used PINs and whether other users have access to the device. The decryption of data can take place on the user side or on the application server side. The first requires special software for the encryption/decryption process. Potential solutions involve dedicated software like plugins and extensions, the use of client side scripts which run in the clients browser. Potential weakness is malware. Sending the key to the application is only possible if the key is stored in the filesystem. This could also compromise the key, as it is sent away from the user.

Alternative: dedicated key server Alternative solution is to create a dedicated server which will store keys. Such a solution is described by Jingmin and Wang in [JW01] where key pairs are stored in a directory server, and for every query the database needs to request a key. Such a solution can create a performance bottleneck, however from the security viewpoint, it is possible to create stronger schemes. Potential benefits include integrity and security of keys.

4.2 Security comparison of the key storage schemes

Independent of the scheme, with access to the machine, plaintext keys can be stolen and keys inside a HSM can not. When using hybrid encryption to protect the data, the private

keys need to be accessed in order to decrypt the data.

Table 2. Comparison of key storage schemes in respect to localised attacks

| Threat | $A1$ | $A2$ | $A3$ | B_{DB} | B_F | B_H | C_L | C_S |
|---|------|------|------|----------|-------|-------|----------------|-------|
| Data leakage from insider attack on database | Y | Y | Y | maybe | N | N | N | N |
| Key leakage from insider attack on database | Y | N | - | maybe | N | N | N | N |
| Private key leakage due to application server vulnerability | N | N | N | Y | Y | N | N | Y |
| Symmetric key leakage due to application server vulnerability | N | N | N | maybe | Y | Y | Y | Y |
| Key leakage from due to client machine vulnerability | N | N | N | N | N | N | Y ¹ | Y |

Y – yes, the danger occurs.

N – no, the danger does not occur.

maybe – the occurrence of the danger depends on the exact implementation.

Scheme **B** has been divided into three cases – B_{DB} where encryption is done on the database side and the key is sent over the network, and the other two, B_F (short for file) and B_H (short for HSM) depending on the key storage format.

C_L is the case when the key is used locally in scheme **C**, and C_S is the case where the key is sent to the application side.

¹ – assumes that the key is stored in file format.

In table 2, key storage schemes are compared in regards to attacks or vulnerabilities on the three levels. With scheme **A.1** the adversary can access the key and the key material, granted they have access rights, and therefore can use database dump leakage attack. With scheme **A.2** the adversary could get key access but only on the database machine. The adversary can not access the key material, therefore only data available at a given time can be decrypted. With scheme **A.3**, there is no need for key access, as the data is decrypted on database startup.

With scheme **B**, if the key is sent to the database machine, the key material can leak. If all decryption is done on the application server, then key leakage depends on the key storage format.

With scheme **C**, if the key is sent to the application server, it might leak. Otherwise it depends on the storage format and the user machine's security. The symmetric key leakage of scheme **C**, caused by application side vulnerabilities, depends on the exact protocol of the encryption process. If the key is being sent in plaintext format, it could leak. If all encryption is done on the client side, then the symmetric key will not leak due to application side vulnerabilities.

The dedicated server scheme is assumed to never leak the key, however the key access might be broken in some authorisation attack.

We believe that a strong combination is to encrypt and decrypt data on the application server side, as this is more secure than scheme **A** however easier to implement than scheme **C** with user side decryption. The users key will be stored on the users side, as with scheme **C**. The best case would involve storing all keys on HSMs. The next level is to allow storage of user keys in filesystem, however the keys that can decrypt more data need to be stored on HSMs.

5 Hardware Security Modules

A hardware security module is a dedicated device for storing secrets and only enabling limited access to said secrets. Additionally, depending on the design, a HSM can provide different levels of tamper resistance or detection. The aim of this is to prevent the leakage of the stored secrets against any sort of attacks. Due to high risks and standards the prices for said devices vary. Yubikey by Yubico and Nitrokey Start by Nitrokey are some of the cheaper versions. For this reason a copy of Nitrokey Start and Nitrokey HSM were ordered and used in this project. The main use case in the project was to store sensitive keys on a HSM (the CA signature creation key and user's/analyst's decryption key), and to use them when signing CSR-s or when decrypting data.

In a project with multiple users a cheap solution would be most useful. It should be mentioned that Nitrokey Start does not contain a tamper resistant smart card but the logic is stored in a microprocessor as opposed to other Nitrokeys products Pro, HSM and Storage.

The need for a HSM in this service architecture comes from the high risk which the leakage of a key can cause. The first two keys that must be protected are the CA private key since this is used for access control, and the analyst private key which holds access to all of the encrypted data. The leakage of the CA could be used to issue fake certificates and used to either inject data or try to create a new analyst certificate which has to be somehow injected to the application but if done right, could be used to cause data leakage and also cause denial of service to the organisation. The CA certificate can also be used to sign certificate revocation lists (CRLs) and therefore could also cause denial of service. The leakage of an analyst certificate would mean that the attacker could decrypt the whole database causing a major data leakage.

5.1 Motivation

In [PWHN18, Chapter 2], attack vectors on cryptographic keys are described – the aim is to show how keys could be extracted from computer-systems. Most of the vulnerabilities lead to attacks on main memory to enable access or extraction of private keys. These include Unified Extensible Firmware Interface (UEFI) manipulation to compromise an operating system (OS) before boot. Another attack uses microchips, which have been implemented for the use of Hardware-Enabled Remote Access. In [PWHN18, Figure 3] extraction attacks against private keys stored in the Main-Memory are presented. As they suggested in section IV, these dangers can be lessened by storing the cryptographic keys in a secure environment, such as a HSM, from where they cannot be extracted.

5.2 Comparison of different HSMs

Nitrokey Start According to the published factsheet [Nitc], the Nitrokey Start can hold 3 RSA or ECC key pairs and one certificate. Suggested RSA key length is 2048 bits. 4096 bit keys are supported but have long operation times. Supports 256 bit ECC keys. At the time of purchase, the device costs around 29€. Supports OpenPGP/GnuPG email encryption. Does not contain a tamper resistant card but rather emulates the way a HSM stores keys and provides access. The emulation is done in a micro processor.

Nitrokey HSM According to the published factsheet [Nitb], the Nitrokey HSM 2 can hold 38 RSA key pairs of length 2048bits or 19 RSA key pairs of 4096 bits, 300 ECC key pairs of length 256bit or 150 ECC key pairs of 521 bits. At the time of purchase, the device costs around 59€. Does not support OpenPGP/GnuPG email encryption. Performance from the specifications for plain RSA is given for 4096 bit keys 4 seconds and for 2048 bit keys 250 milliseconds. This means that the decryption of a batch of data can take quite a while as opposed to some other models.

Gemalto SafeNet Luna PCIe HSM According to specifications [Gem17b], PCIe and Network attached devices can (depending on the model) do 1000 to 10000 RSA 2048 bit key transactions per second.

Gemalto SafeNet Luna USB HSM Gemalto also has a USB HSM [Gem17a], which has lower performance compared to PCIe and network attached HSMs from Gemalto, reaching 63 transactions per seconds of 2048 bit RSA keys.

When picking devices to be distributed to users, cost effectiveness is often considered. From the security viewpoint, tamper resistance is important. On the other hand, the user might not be running large numbers of decryption operations. Unless real-time decryption as a service is the intended use, a slower device can also achieve the objective. This leads to picking the cheaper models. The scale of the project influences the process of picking the right HSM. For an example, the Nitrokey USB keys guarantee a life expectancy of at least 100000 PIN entries for Nitrokey Start, 500000 PIN entries for Nitrokey Pro and Nitrokey HSM. These numbers set an upper bound on the amount of collectable data. For a large scale project, a faster device with a longer lifetime expectancy should be picked. For comparison, the Safenet Luna USB HSM [Gem17a] has mean time between failures given in 858,824 hours.

5.3 About OpenPGP

OpenPGP message protocol standardised in [CDF⁺07] is of interest to us since it also explains the subkeys. In 5.5.1.2 it is said that a “Public-Subkey packet (...) has exactly the same format as a Public-Key packet, but denotes a subkey. One or more subkeys may be associated with a top-level key. By convention, the top-level key provides signature services, and the subkeys provide encryption services.” Since RSA itself does

not define subkeys this definition was required to provide some insight into the OpenPGP cards [Opef].

5.4 Security of HSMs

Although many HSMs claim tamper resistance, there can sometimes be exploits in their PKCS#11 API. Matteo *et al.* in [BCFS10] created an automated tool called Tookan, which was used to analyse and attack commercially available tamper resistant cryptographic security tokens. Their aim was to extract sensitive cryptographic keys. The results showed that 9 out of 17 tested devices had a vulnerability. Their results appear in [BCFS10, Table 3]. Out of the listed devices 1 was a Gemalto USB device, on which no vulnerabilities were found, however 2 out of 3 tested Gemalto smart card models contained vulnerabilities. This shows the importance of research prior to selecting a device.

In 2018 Benadjila, Renard, Tebuchet *et al.* mentioned Nitrokey in [BRT⁺18, Section 1.3]. When comparing secure USB devices, they mention the open sourced and open hardware facts of Nitrokey. Also that since the firmware of these devices can not be updated over USB, they are resistant to a BadUSB host to device attack (although later there is a note, that there is a not secure firmware update – a controlled but not cryptographically sound, as they say). They also point out the lack of some crucial features – an integrated user input system for more secure PIN entry. Also the firmware does not make use of dedicated kernel isolation and in-depth defence techniques therefore any software vulnerability would compromise the platform.

5.5 Integration of HSMs

The following will describe our process of preparing Nitrokey device with the aim of integrating them to a Java application. Our aim is to be able to generate the keys on the secure area of a HSM, then certify them and finally test access to both the key and the certificate through a Java application.

5.5.1 Integration of Nitrokey Start to the application

First, we will describe the attempt of integrating Nitrokey Start with the application.

Device initialisation Prior to the integration steps the device had to be initialised. We began by following the Nitrokey Start installation guide for Windows [Nita]. As suggested, we used the tool from Gpg4win [gpg] toolset – the GnuPG tools for Windows.

During the first connection attempts there were issues with software. If the computer had a built-in card reader GnuPG attempted using it since it occupied slot 0 and said there was no card. The device had to be ejected from the computer.

Key generation Next we used the GPA software from the Gpg4win suite to generate RSA keys on the card (inside the Nitrokey Start). This process generated 3 RSA key-pairs.

Certification The next step was to generate a *certificate signing request* (CSR) to link the token with our PKI. We required an X.509 certificate. The problem occurred while generating the CSR, as there was need to sign the CSR with the key for which the request was being generated for, but the device did not allow signing with a key that was meant for encryption operations. There are online tutorials for creating CSRs for the authentication key (key slot 3). Note that at this point, we had strayed from the official guide, as the official guide did not explain certification.

To see if the Nitrokey Start device can even be used with a Java application, we attempted an access test before completing the certification. Our first attempt using the SunPKCS#11 provider resulted in no positive results besides the application discovering the device. It soon came apparent, that a certificate-private key pair had to exist in order for Java to enable access to it.

We then executed the following commands:

1. `gpgsm --gen-key`
2. `openssl x509 -req -in <exported_csr>.csr -CA rootCA.crt -CAkey rootCA.key -CAcreateserial -out <signed_crt_to_card>.crt -days <integer> -sha256`
3. `gpgsm --import`

The first command created a CSR, using the third key which is the authentication key, and enabling actions sign and encrypt. The second command signed the CSR. The third command was used to import the certificate to the card (the HSM). The first and third command required additional interactive input. The *<exported_csr>.csr* is the output of the first command. The *<signed_crt_to_card>.crt* of command two will be imported in command three.

The next attempt was connecting the Nitrokey with a simple Java application. The first plan was to follow Java Sun providers online guides [jav]. Most of the connections succeeded up to the point where the aliases of keys stored in the keystore were supposed to be enumerated, however the list was empty. The connection with the token was verified by providing wrong PIN and getting an error.

Second try at importing the certificate Since the GPG tool seemed to import the certificate in a way, which did not tie the certificate to a key, other tools were tried. The OpenSC library contained PKCS#11 and PKCS#15 tools. The first attempts to import a certificate failed. The *pkcs11-tool* gave a *wrong PIN len* error. The *pkcs15-init* tool simply did not import it.

A hypothesis was that the import failed due to missing space. Therefore using the *pkcs15-init* tool, the public key of the third certificate was removed from the token. After this, the import worked. However the token had also deleted the private part of the key.

After trial and error of trying to generate a new key-pair using *pkcs11-tool* and *pkcs15-init* tool, and all attempts failing, it was discovered at [Opef, 3. Generating keys], which stated that PKCS#11 support for key generation for GNUK was missing in the OpenSC.

After generating the key pair using OpenPGP commands, the device was in the same status, as prior to attempting to import the certificates.

Conclusion At this point we gave up on using the Nitrokey Start for our project. It came apparent that initialising the device using GnuPGP methods does not work with X.509. The lesson learned is that the device was chosen without enough research. For S/MIME/X.509 there is a Nitrokey guide [Nitd]. This assumes that the user has a key-certificate pair in .p12 file format, which will be imported to the device. The downside is that in this way, the key is not generated on device and could leak prior to import.

5.5.2 Integration of Nitrokey HSM to the application

Because of the difficulties in using the GNUK with PKCS#11 and getting access to the keys, a Nitrokey HSM was ordered with the aim of using it for CA operations and the analysis certificate. Next, we will give an overview of setting up and connecting the Nitrokey HSM to our application.

The main issue was getting a PKCS#11 engine to work. OpenSSL guides for generating certificates assumed the existence of a shared library but the process of acquisition was omitted. The second problem was that the engine PKCS#11 [Oped] project by OpenSC had been merged to another project libp11 [Opee] but it was not clear from the start what should be used in place of the late engine PKCS#11.

Setup and key generation Following the guide [Opeg] on OpenSC, we began by initialising the Nitrokey HSM. This was done using *pkcs11-tool* from the OpenSC project. We confirmed that the PIN change commands worked. To create a certification scheme, we needed a CA keypair. For key pair generation we tested with a key length of 4096 bits and assigned it id 10 (the same as in the example). The key pair was generated successfully in a bit over 2 minutes. We later switched to a 2048 bit key.

Missing engine PKCS#11 Next step in the guide was using OpenSSL to generate a certificate signing request. The main difficulty was that most of the guides assumed that `engine_pkcs11` was already present. There were no explanations to be found where to get it. After some research it came apparent it had to be compiled on the system. As there existed `engine_pkcs11` project in the OpenSC git, the assumption was that that would be the correct project. However the project had been merged to `libp11` project. The `libp11` git contained a detailed installation guide, which we attempted to follow. Three options were described for building the library on Windows – MSVC (Microsoft Visual Studio Command Prompt), MSYS2, Cygwin and MinGW / MSYS. The first attempts were done with MinGW / MSYS as these tools had been previously installed on the device. Something in the compile process went wrong. Another few attempts were made with MSYS2. In that case the build seemed to succeed, however the build process used OpenSSL from wrong locations, and when trying to use there were errors. Eventually we built the library with MSVC which was also first on the list of possible installation methods. Eventually the build seemed to succeed.

We wish to note that the build results with two dll-s located in the *src* folder of the project. The one to be used in place of *engine_pkcs11* is the *pkcs11.dll* file.

On Windows 10 we built the `libp11` application with Microsoft Visual Studio 2017 Community edition and referenced OpenSSL for windows version 1.1.1b and later also with 1.1.0. On Windows 7 we built the `libp11` application with Microsoft Visual Studio 2019 Community edition and referenced OpenSSL for windows version 1.1.1b. In case of Windows 10, the engine load step in OpenSSL succeeded. On Windows 7 it

did not – this could have been to do with the location of OpenSSL dll-s. In the libp11 release notes it was also noted that Openssl 1.1.1 beta is supported from libp11 version 0.4.8. The tests were carried out with libp11 version 0.4.10 release.

The second command on OpenSSL was to access a certificate. When running it on the Windows 10 PC it informed that the slot number was invalid. There was no change when using another notation to provide the slot number. The slot number had been confirmed with the *pkcs11-tool*.

It is important to mention that whenever there was a built in smart card reader either the OpenSSL or in case of Nitrokey Start the GPA and the *openpgp-tool* had problems. The problems were induced by assuming the use of slot 0 which was occupied by the empty card reader. This was confirmed on one Windows 10 PC where, after ejecting the reader, the software started to function. This is not always possible as was displayed on a Lenovo ThinkPad running Windows 10, as the eject device option was not displayed in the UI. This should be considered because in order to deploy a solution to masses, the connection of a device should be as simple as possible, and ejecting a built-in device seemed risky.

Attempts on virtualised Ubuntu As the certification step failed on Windows, the next idea was to attempt a Linux based operating system. For that we chose Ubuntu 18. The problem with a virtualised operating system was that in this case, the HSM was not connected to the *pkcs11-tool*.

Success on Ubuntu This time we prepared a clean install of Ubuntu 18 on a low performance HP Compaq. Using `apt-get install` we installed *openssl* and *libengine-pkcs11-openssl*. Then we built and installed OpenSC according to the build guide on Unix flavours [opea]. After installation, the Nitrokey HSM was detected through *pkcs11-tool*.

Creating the CA certificate This is the example for key id 1, device in slot 0. Disclaimer – the commands contain default locations to so-s and the default user PIN of a Nitrokey HSM.

Let `engine_path` be `/usr/lib/x86+_64-linux-gnu/engines-1.1/pkcs11.so`
and `module_path` be `/usr/lib/opensc-pkcs11.so`.

When replicating, these variables have to be entered to the command.

Open terminal and execute the following commands:

1. Run *openssl*
2. `engine dynamic -t -pre SO_PATH:engine_path -pre ID:pkcs11 -pre LIST_ADD:1 -pre LOAD -pre MODULE_PATH:module_path`
3. `req -engine pkcs11 -new -key 0:1 -keyform engine -out cert.pem -text -x509 -days 3640`

4. Quit *openssl*
5. `openssl x509 -in cert.pem -out cert.der -outform der`
6. `pkcs11-tool --module module_path -l --pin 648219 --write-object cert.der --type cert --id 1`

Step 2 prepares a cryptography engine for communicating with the PKCS#11 device. During step 3, a self-signed certificate is created – the certificates details have to be entered. Step 5 is simply converting the certificate from pem format to der format, to be written to the token. The result is a self signed certificate.

Issuing certificates First, we generate a key pair:

```
pkcs11-tool --module module_path -l --pin 648219 --keypairgen --key-type
rsa:2048 --id 2
```

If the key resides in a HSM as in this case, it is important not to disable signing operations – otherwise we can not generate a CSR.

Open terminal and execute the following commands:

1. Run *openssl*
2. `engine dynamic -t -pre SO_PATH:engine_path -pre ID:pkcs11 -pre LIST_ADD:1 -pre LOAD -pre MODULE_PATH:module_path`
3. `req -engine pkcs11 -new -key 0:2 -keyform engine -out analysis.csr`
4. `x509 -engine pkcs11 -req -days 3640 -CAform=der -CA cacert.crt -CAkeyform engine -CAkey 0:1 -CAcreateserial -in analysis.csr -out analysis.crt`
- alt `x509 -engine pkcs11 -req -days 3640 -CAform=der -CA cacert.crt -CAkeyform engine -CAkey 0:1 -CAserial cacert.srl -in analysis.csr -out analysis.crt`
5. Quit *openssl*
6. `openssl x509 -in analysis.pem -out analysis.der -outform der`
7. `pkcs11-tool --module module_path -l --pin 648219 --write-object analysis.der --type cert --id 2`

In this example we create another certificate-key pair on the same token. We sign it with our CA certificate from previous step. In step 3 we create a CSR – certificate details have to be entered. When generating the analysts certificate, we entered 'analyst' in the

organisational unit (OU) field – the reason will be explained in section 6. In step 4 we signing the CSR using our CA. In a real world example, step 4 would use a different device. The difference between step 4 and alternative step ‘alt’, is that in the alternative step the CA already has a serial number file.

Conclusion We managed to initialise the Nitrokey HSM, generate RSA key pairs, create a CA certificate and issue a user and analysts certificate. Testing with a simple Java application resulted in successfully accessing the key store, accessing the certificates and also private keys. This part concludes with having reached the ability to decrypt ciphertexts inside a Java application.

6 The application

We describe the goal and the requirements of this application.

The goal of the application is to collect sensitive health data and store it in a secure manner for later analysis in a project centred setting.

The collection of data might take a few years but less than 20.

There are two main roles:

Role 1 – users who provide data

Role 2 – analyst with access to all of the data

Functional requirements:

1. A User can access their data
2. A User can access only their data
3. The Analyst must be able to access the users' submitted data
4. The data is stored in a database.
5. The application can distinguish between a user and the analyst when collecting data.

Non-functional requirements:

1. The application must support a userbase of size m .
2. The original data remains secure even if the database is compromised (for example, someone downloads the whole database).

Adversary The security model defines an adversary, who might try to make copies of database rows or entire tables. The adversary can have access to the database server. The adversary is interested in leaking the data but not in denial of service or destruction of the database. (The last part is important so that the adversary would not simply drop the tables and cause data loss.)

A typical data collection application would also state many requirements about who can edit the data or access the application. This is a prototype application with limited requirements – another application encapsulating or implementing this would have to implement user control, access and other business logic.

6.1 Architecture

Based on the requirements, we define the following architecture.

6.1.1 Defining choices

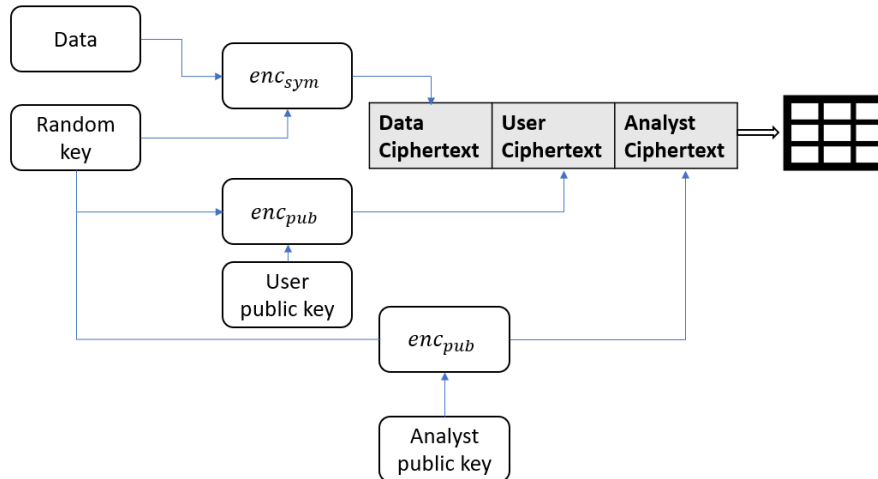


Figure 1. Storing an Entry

Data storage scheme We will be using row level encryption to separate every entry. The data will be encrypted and decrypted on the application level using hybrid encryption, which allows separating the users who can create entries from those that can access entries, while containing the efficiency of block ciphers. This satisfies functional requirements 1 through 3 (the necessary users have access to their data). The private keys will be stored using key storage scheme C – the key is stored with the user who can access the data. Every entry will be encrypted using a randomly generated symmetric key, which will in turn be encrypted for the user who provides the data and for the analyst using public key encryption schemes (depicted on figure 1). In such a scheme, the sensitive data is stored in the database in only encrypted format. This protects against simple data leakage. Using row level encryption allows reading and writing by entry, which is necessary for the user to decrypt their data. Using encryption on the application level is to enforce our encryption requirements. Decryption will also be done on the application level due to ease of implementation. We can divide the application into two here – if the key is stored in plaintext file, the application can run in a different machine. For a more secure solution the application should run in the same machine with the user. This also allows the use of HSMs.

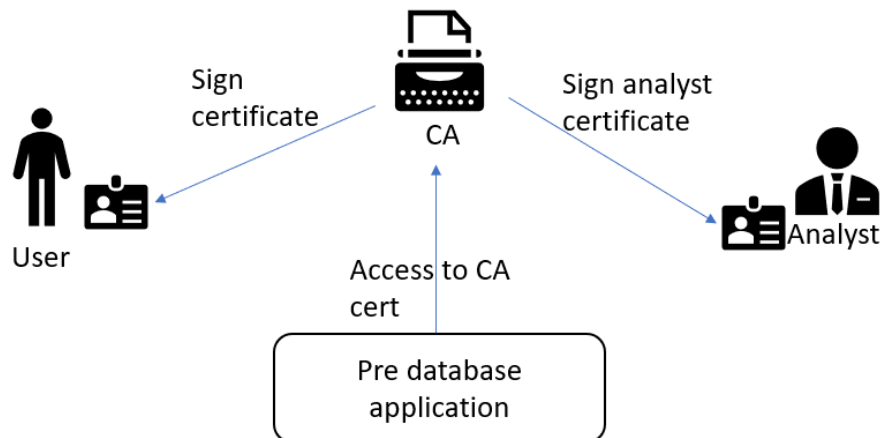


Figure 2. PKI

Access control We place role based restrictions on who can provide the data and who can access readable data. To enforce access control, this scheme will use certificates. We first set up an application/enterprise based *public key infrastructure* (PKI), which consists of a *certificate authority* (CA). This CA will sign certificates for users of the application, as well as provide the analyst with a signed certificate containing an extra field, which proves that this is indeed the analysts certificate. We use a centralised certification scheme as opposed to a peer-to-peer scheme. This is why we require X.509 instead of OpenPGP. The PKI is visualised in figure 2. We place the restriction that data can only be encrypted for valid certificates provided by the CA. We place the second requirement that the analysts public key comes from the analysts certificate. The readable data access is enforced by the ability to decrypt the data – only the owner of the certificate is supposed to be able to decrypt the data. (In a public application, additional access restrictions can be enforced as to who can request the ciphertext).

We do not define over-writing inserted data, as this would require verifying that the writer also wrote the previous entry. This could be done by encrypting cell coordinates (the technique of [EWSG04, Section 3] mentioned in subsection 3.1) – the decrypting party (modifier) proves to the encrypting party (inserter) that they know exactly where this key came from. For further authenticity of the scheme, a token could be added, for example a signature on the cell coordinates, which would be unknown until decrypted.

6.1.2 Functionality description

Functionality We can split the process of data gathering and re-accessing into different modules. These modules can be combined into one single application, however we will separate them to better visualise at which step certain functionality is applied. Next

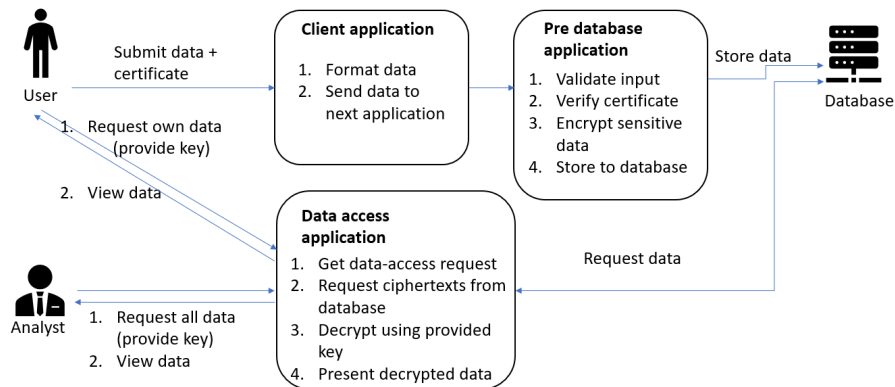


Figure 3. Application architecture.

we will describe the two main applications – first one for gathering data, second for accessing stored data . There will be an additional third application to assist in creating input for the first application. The three applications will be named as follows:

1. Pre database application
2. Data access application
3. Client application

Pre database application To enforce the scheme, we require a cryptographic application configured with the analyst’s certificate. This application will take as input plaintext data, a certificate (from the user) and some identifier (personal code for example, or some other numerical identifier if common identifiability of a person is not allowed). The application then verifies that the certificate is signed by the CA. After this, the data is encrypted using a hybrid encryption scheme, where first a symmetric key is randomly generated. Then the data is encrypted using the symmetric key, the key itself is encrypted for both, the user and the analyst using their public keys (obtained from validated certificates) and an asymmetric encryption scheme (RSA with some padding in this example). Finally, the three ciphertexts (data ciphertext, and both key ciphertexts) alongside with the personal identifier are sent to the database to be stored.

Note that all of these operations are done in an application, which is separate from the database. This allows the use of any DBMS, for which long enough data entries are allowed (the ciphertext can be long).

The first application should not contain any of the private keys, as it only creates entries.

Data access application For the data retrieval part, we require another cryptographic application, which will request ciphertexts from the database. Then it will perform decryption as described for hybrid encryption scheme. This application has access to the private key, which is kept safe by the user or the analyst.

Client application This web application visualises the classical form for gathering data from a user. In a production environment it is supposed to deal with authorisation, gathering the data, formatting and structuring according to some rules, and finally passing the data along side the users identifier and certificate to the first application for storage.

Service model The service model consists of three layers and is shown in figure 3. The three layers enumerated from the user side. On the first layer there's an application used to submit data and applications used to review data. On the second layer or intermediate layer, there is the intermediate application, which gets encryption requests, verifies the correctness of provided certificates, encrypts the data to be stored, and finally sends the insert command to the third layer. Finally, the third layer consists of a database where sensitive data is stored in encrypted format.

In the initial model, the data access application can make direct queries to the database, and then uses the clients private key to decrypt the entries. In a production environment it might be better to have an intermediate service which limits users access to the data according to the users clearance.

Such a model assumes that the PKI has been set up, and certificates have been issued.

Additionally, there could be an additional service to automate the issuing of users certificates. This could be done by having the users sent login credentials. The application would then authenticate the user and have the CA sign a simple users CSR. This would reduce the workload of setting up the users certificates.

A production case scenario would have authentication steps set in front of the applications. These are omitted in the prototypes.

6.1.3 Technical description of database

In our example, we use Postgres 10. The applications require a database with two users – one for reading and one for writing entries. There is one table called entry (See figure 4), with 4 fields + the id field. The `_ciphertext` fields contain base64 encoded ciphertexts. The `pno` field is the user identifier, which can be used for example re-accessing the data.

The base64 encoding is not really necessary on the database side and could actually be omitted for storage efficiency.

| entry | |
|-------|---------------------|
| 123 | id |
| ABC | pno |
| ABC | data_ciphertext |
| ABC | user_ciphertext |
| ABC | analysis_ciphertext |

Figure 4. Table entry

6.1.4 Technical description of chosen cryptographic functions

For the public key scheme we chose RSA with key lengths of 2048 bits with OAEP-WithSHA256AndMGF1Padding padding, as it is one of the supported schemes in the BouncyCastle API. For symmetric encryption we chose AES GCM with no padding. The AES key length is 256 bits. The GCM parameters are 12 bytes of IV and a 16 byte TAG. The parameters were chosen with consideration of [uMB] which in terms referred to [Dwo].

For encrypting we used Bouncy Castle Java API [otBCIa]. For decrypting, the applications either used Bouncy Castle Java API if the key was stored in file. For PKCS#11 support the SunPKCS#11 [jav] provider must also be used. Since the SunPKCS#11 provider does not support OAEPWithSHA256AndMGF1Padding, the decryption process is separated into two steps – the on token RSA/ECB/None (RSA cipher with no paddings), and then the padding is removed in the application (this two step solution was given in [staa]).

6.1.5 Technical description of Pre database application

The implementation is called DB-pre-layer-crypto as it is a cryptographic application on a layer right before the database.

It uses Spring Boot [Piv] and serves one path – /insert.

Expected format of input is data in JSON format described by schema shown in figure 5. The application verifies the existence of the fields in the structure, whether all fields can be converted to strings, and finally verifies the certificate. The verification of the certificate is handled by Java, and requires also the CA certificate to validate the

signature. If the verifications succeed, the data is encrypted and inserted into the database (as was described in figure 1). The current application has limitations as the methods of sending data, encrypting and inserting are constrained by the memory accessible to the application. A more improved application would use streams.

```
{
  "$schema": "http://json-schema.org/draft-07/schema#",
  "title": "Entry",
  "description": "Entry for DB pre layer crypto",
  "type": "object",

  "properties": {

    "pno": {
      "description": "The identifier of a person",
      "type": "string"
    },

    "data": {
      "description": "Sensitive data to be stored",
      "type": ["string", "object"],
      "$comment": "The contents of an object will be taken as a
        string."
    },

    "cert": {
      "description": "Users certificate in PEM format",
      "type": "string"
    }
  },

  "required": ["pno", "data", "cert"]
}
```

Figure 5. Entry insert schema

6.1.6 Technical description of Data access application

The second application serves a web interface (as a page) for accessing stored data – the user (any role) can configure the request they wish to make. During configuring, the user selects a role, whether to access only rows with a certain identifier (pno), or all rows. When accessing a certain pno, the user acts in the USER role and user_ciphertext is requested. When accessing all rows, the user acts as the ANALYST role, and the analyst_ciphertext is requested. The user also picks the decryption mode – either FILE or TOKEN (PKCS#11 token), depending on which format their key is stored in. In case

of TOKEN, the application must also be configured to enable access to the token. Upon submitting the form, data is requested, decrypted and displayed either in file format or in the browser, depending on the users choice.

This application contains many web components, which are used to configure the choice. It also has two classes which extend the `DecryptionService` – a class which decrypts the stored data into its original form. These services are:

- `FileBasedDecryptionService`
- `Pkcs11DecryptionService`

6.2 Analysts certificate

The application uses two main roles – a user and an analyst. The analyst is allowed access to all of the data. Other users can only access their own data. To distinguish between a simple user's certificate and an analyst's certificate, we add the special role inside the certificate. The absence of a role can be interpreted as simple user. This presents the question, in which field of the certificate should the role be stored. Since the X509 v3 has extension fields, we could try to find a place in there. The first structure is the distinguished name (DN) of the owner. Since the serial number (S), given name (G), surname (SN), common name (CN) and country (C) fields are often used, we could use the organisational unit (OU), title (T) or (TITLE) fields to store a special value. Such a format allows storing the subjects personal information with additional role based properties in the certificate. For this example, we choose the organisational unit field.

6.3 Technologies

Next we will give an overview of the technologies which were chosen to implement the applications.

Most of the testing and development was done on the Windows platform with either a 64bit Windows 10 or Windows 7. There are no known restrictions for building and running the software on a Linux system.

For certification we use OpenSSL [opeb]. A windows build of OpenSSL was provided at [Pro]. When experimenting with HSMs we used OpenSC project libraries OpenSC [Opeh] and libp11 [Opee].

All three web applications are written in Java and use Maven [Apab] for dependency management. For the database we use PostgreSQL which is a widely used DBMS. Spring Boot was used to create the database pre layer application. It contains necessary interfaces for setting up a small REST controller application. This application will process POST requests.

For the cryptographic side of these applications we have used Bouncy Castle Java API [otBCIa]. For PKCS#11 support the SunPKCS11 [jav] provider must also be used.

The two user interfaced applications use Wicket Quickstart [Apa], a Java web framework provided by Apache. This can be used to create client side applications with ease.

6.4 Validation and testing

All three applications are created as maven projects, meaning they can be built by command `mvn package`. Specific build, configuration and run guides are present in the README files of the applications.

The applications presented in the thesis have been manually tested to verify positive workflow scenarios – the application works with correct inputs.

To validate the correctness of storing the data, we used unit testing. The purpose of unit testing is to validate that a component works as designed [uni] – the component is tested with some inputs, the output is compared to the expected output. We used JUnit 4 [JUn] as the testing framework. In the first application we created tests for `CryptRequest` class, which handles validating inputs and performs encryption. Tests were created as seen fit by the developer.

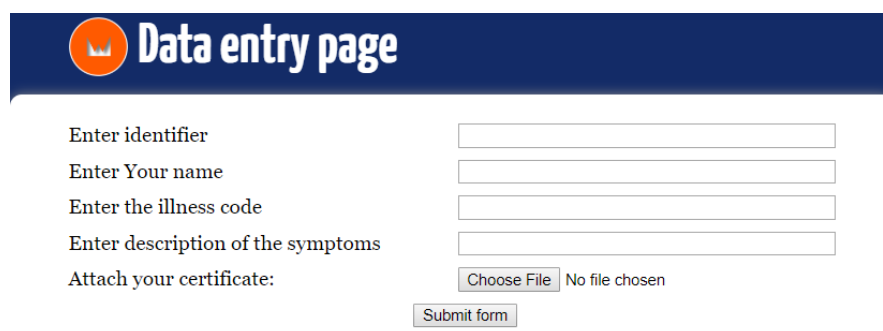
Application 2 was validated manually to see if the process of using a HSM was possible.

Application 3 was primarily created to assist testing Application 1. No other tests were performed besides verifying that the form was submitted, and message sent.

6.5 Views

The following displays the views of the second and third application.

For collecting data, the third application provides a form displayed on figure 6. It collects the identifier, certificate and the rest of the data, and sends it to the first application.



The image shows a web form titled "Data entry page" with a dark blue header containing a logo and the title. Below the header, there are five input fields with labels: "Enter identifier", "Enter Your name", "Enter the illness code", "Enter description of the symptoms", and "Attach your certificate:". The "Attach your certificate:" field includes a "Choose File" button and the text "No file chosen". At the bottom of the form is a "Submit form" button.

Figure 6. Form page of third application

Data download application

This is the main page of the data download application. Use it to decrypt and download stored data.

Role choice:

USER ANALYST

Enter user identifier (pno):

Decryption method:

FILE TOKEN

Upload the file containing the cryptographic key here. Expected in PEM format.

No file chosen

Download data View data

Figure 7. Home page of second application

The second applications home page, displayed on figures 7 and 8 is meant for configuring the user role and key storage format. In figure 8, the user has to enter their PIN, after which a connection to the PKCS#11 device is tested and then a dropdown selection of available keys is shown to the user, as displayed in figure 9. The user must select the correct key (based on the CN of the certificate). Finally, in figure 10 the CN of the chosen keys certificate is displayed.

If the user clicks on Submit next to Download data, figure 11 is shown. The data download page of second application For View data choice the view depicted on figure 12 is shown.

Role choice:

USER ANALYST

Decryption method:

FILE TOKEN

Enter user pin for your HSM:

Download data View data

Figure 8. Different choices of second application

6.6 Performance and reliability

If the application uses Nitrokey HSM as the analysts key storage format, then maximum number of users $m < 500000$.

The application is not performance oriented. The measurable processes are:

1. Application 1 – processing an insert request
2. Application 2 – accessing the data (from submit on home page to data access)

Both steps depend on the parameters of asymmetric encryption scheme used and the size of the data. For example in the RSA scheme, a longer modulus results in a longer ciphertext and a longer (encryption/decryption) function runtime. The runtime of step 2 also depends on whether the private key is stored in a file or on a HSM. With a Nitrokey HSM it has been specified that decryption operations with 4096 bit keys will take 4 seconds.

One could also measure the time it takes to start the applications, prepare the pages or validate data at different steps.

Another characteristic is reliability. The cryptographic functions used are reliable. However, when performed on a HSM, the reliability of the said device also factors into the reliability of the whole. When accessing data, it was noticed that there were a few connectivity issues, in which case the operation had to be repeated.

The second application has a known bug with form submit, where the first submit clears the form – this does not affect the internal functionality of the application (the

Decryption method:

FILE TOKEN

Choose the decrypting keys certificate:

CN=UserA, O=Internet Widgits Pty Ltd, ST=Some-State, C=EE ▾

Figure 9. Second application private key selection

Decryption method:

FILE TOKEN

CN=Jerry, OU=analysis, O=Example, ST=Some-State, C=EE

Figure 10. Second application after private key has been chosen

decryption), but is a usability problem of displaying the functionality (the user has to refill the form).

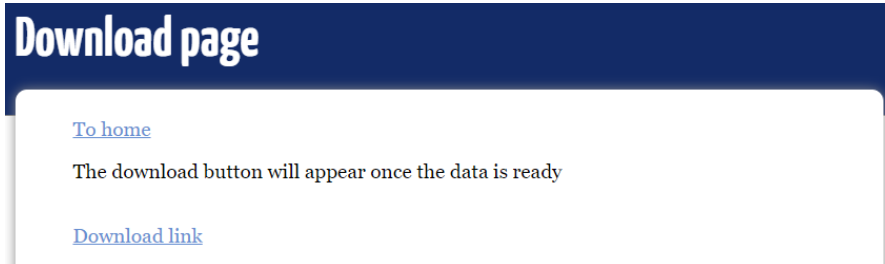


Figure 11. Second application download data choice



Figure 12. Second application display data choice

7 Conclusion

In the thesis, we discussed on the topic of secure storage of sensitive data. We looked at different approaches to secure data storage, and discussed their benefits and disadvantages. We also discussed different key storage schemes and compared two key storage formats. Based on these discussions, we chose the most fitting combination to implement.

To support hardware security modules in the implementation, we did some research on available models and documented our process of preparing the HSMs for the application.

We presented an example implementation of a secure data gathering software consisting of three applications, which only gathers data from trusted sources and where the data is secure against passive adversaries on the database side. The solution supports the use of hardware security modules which also ensure the safety of the keys. The example can be used when implementing similar functionality in larger existing data gathering software.

The software gives the first look at the methodology, although implementers should always pick the strongest available ciphers and key lengths. The performance results of such a software depend on given computational power and used devices. For an example, when using the Nitrokey HSM with 4096bit RSA keys, one decryption operation takes at least 4 seconds. The decryption implementation should also be optimised to use streams with the aim of reducing memory cost of the application.

7.1 A look back at the choices

7.1.1 The HSMs

Looking back at the choices made, the first device was selected with a lack of research. Some of the issues with Nitrokey Start could have been avoided with more previous research. On the other hand, the device was ordered to avoid waiting for the arrival. There was also a lack of understanding that the OpenPGP device would be complicated in a classical PKI scheme. The fact that the Nitrokey Start can only contain one certificate with its three keys was not visible from the start but discovered after problems with internally assigning a certificate to a key arose.

A possible way to still use the Nitrokey Start would be to make some steps manual or with the use of shell scripts. As we were aiming at an automatic Java based solution some of the possibilities were not really looked into, at least not early enough to be implemented or tested. The use of shell script invocations from inside a Java application seemed like a grey area and was left untouched, but this could have provided the functionality. On the other hand, there are multiple security considerations with shell scripts, and due to lack of experience in the field, we chose not to implement it into the example.

7.1.2 The application architecture

Looking critically at the applications, the data access application could have been designed better. When accessing one or a few rows of data the time it takes is not such an issue, but when the analyst wants to access all the rows, it will require a lot of time. The web interface can avoid page lock with the use of AJAX and lazy-loading of components. In case of a server-like implementation of this, the decryption should be done on another thread/as a sub process, and should leave a flag, that the work is done. If the key is presented in a file, the decryption should happen fairly fast depending on the computational resource present for the application. However, if the key is on a HSM, then the minimal time is given by the device. This can be further expanded – if the solution is implemented in a web application with on demand access, adding another 4 seconds could be unacceptable. This is where high-power HSMs come in, for example the Gemalto devices, which complete their tasks in milliseconds.

Additionally, the use of HSMs restricts running the application to the application user's machine. This could be avoided with the use of browser plug-ins to provide access to the PKCS#11 device. This however requires additional implementation and can create new vulnerabilities.

7.2 Future work

Alongside the contributions, we have identified the following opportunities for future work.

Improving the process of connecting a HSM The main aim of this is to improve the process of connecting a user HSMs for easier deployment. Having users compile their own PKCS#11 engines would be too difficult for this to be deployed to masses.

Extending PKCS#11 APIs We noticed that the widely used Bouncy Castle API supports RSA/OAEP but the Java PKCS#11 API does not. On the other hand, the Bouncy Castle API does not support PKCS#11. The easier method would be to extend the list of Java PKCS#11 APIs currently available paddings.

Application side encryption with alternative devices In this example we tested with two Nitrokey devices. It would be interesting to find other devices which could be used for application side encryption. One such example would be attempting to connect Yubikey from Yubico with this application.

References

- [ACGW99] Mike Ashley, Matthew Copeland, Joergen Grahn, and David A. Wheeler. The gnu privacy handbook. <https://www.gnupg.org/gph/en/manual/book1.html>, 1999. Last accessed 2019.07.20.
- [AKSX04] Rakesh Agrawal, Jerry Kiernan, Ramakrishnan Srikant, and Yirong Xu. Order preserving encryption for numeric data. In *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data*, SIGMOD '04, pages 563–574, New York, NY, USA, 2004. ACM.
- [Apa] The Apache Software Foundation. Create a wicket quickstart. <https://wicket.apache.org/start/quickstart.html>. Last accessed 2019.08.12.
- [Apab] The Apache Software Foundation. Maven. <https://maven.apache.org/>. Last accessed 2019.05.13.
- [BCFS10] Matteo Bortolozzo, Matteo Centenaro, Riccardo Focardi, and Graham Steel. Attacking and fixing pkcs#11 security tokens. In *Proceedings of the 17th ACM Conference on Computer and Communications Security, CCS '10*, pages 260–269, New York, NY, USA, 2010. ACM.
- [Ble98] Daniel Bleichenbacher. Chosen ciphertext attacks against protocols based on the rsa encryption standard pkcs #1. In Hugo Krawczyk, editor, *Advances in Cryptology — CRYPTO '98*, pages 1–12, Berlin, Heidelberg, 1998. Springer Berlin Heidelberg.
- [BRT⁺18] Ryad Benadjila, Mathieu Renard, Philippe Trebuchet, Philippe Thierry, Arnaud Michelizza, and Jérémy Lefaure. Wookey: Usb devices strike back. In *Symposium sur la sécurité des technologies de l'information et des communications*, 2018.
- [CDF⁺07] J. Callas, L. Donnerhacke, H. Finney, D. Shaw, and R. Thayer. Openpgp message format. <https://tools.ietf.org/html/rfc4880#section-5>. 5, November 2007. Last accessed 2019.05.13.
- [CDH⁺05] M. Cooper, Y. Dzambasow, P. Hesse, S. Joseph, and R. Nicholas. Internet x.509 public key infrastructure: Certification path building. <https://tools.ietf.org/html/rfc5280>, September 2005. Last accessed 2019.05.11.
- [CJNP00] Jean-Sébastien Coron, Marc Joye, David Naccache, and Pascal Paillier. New attacks on pkcs# 1 v1. 5 encryption. In *International Conference on*

the Theory and Applications of Cryptographic Techniques, pages 369–381. Springer, 2000.

- [Cor] Oracle Corporation. Mysql enterprise encryption. <https://www.mysql.com/products/enterprise/encryption.html>. Last accessed 2019.08.02.
- [CS] R Cramer and V Shoup. Design and analysis of practical public-key encryption schemes secure against adaptive chosen ciphertext attack. *SIAM Journal on Computing*, 33(1):167-226, 2003. 1, 2, 3, 4, 5, 9, 11, 13, 20.
- [CSF⁺08] D. Cooper, S. Santesson, S. Farrell, S. Boeyen, R. Housley, and W. Polk. Internet x.509 public key infrastructure certificate and certificate revocation list (crl) profile. <https://tools.ietf.org/html/rfc5280>, May 2008. Last accessed 2019.05.11.
- [DKS08] S. Delaune, S. Kremer, and G. Steel. Formal analysis of pkcs#11. In *2008 21st IEEE Computer Security Foundations Symposium*, pages 331–344, June 2008.
- [Dwo] Morris Dworkin (NIST). Recommendation for block cipher modes of operation: Galois/counter mode (gcm) and gmac. <https://doi.org/10.6028/NIST.SP.800-38D>. Last accessed 2019.08.06.
- [EWSG04] Yuval Elovici, Ronen Waisenberg, Erez Shmueli, and Ehud Gudes. A structure preserving database encryption scheme. In Willem Jonker and Milan Petković, editors, *Secure Data Management*, pages 28–40, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.
- [Gem17a] Gemalto. Product brief safenet luna usb hsm versions 5.x and 6.x. <https://safenet.gemalto.com/resources/data-protection/safenet-usb-hsm-product-brief>, 2017. Last accessed 2019.04.22.
- [Gem17b] Gemalto. Safenet luna pcie hsm 7 specifications. <https://safenet.gemalto.com/data-encryption/hardware-security-modules-hsms/pcie-hsm/>, 2017. Last accessed 2019.04.22.
- [gpg] About gpg4win. <https://www.gpg4win.org/about.html>. Last accessed 2019.05.13.
- [GZ07] Tingjian Ge and Stan Zdonik. Fast, secure encryption for indexing in a column-oriented dbms. pages 676–685, 05 2007.

- [HHK06] J Herranz, D Hofheinz, and E Kiltz. KEM/DEM: Necessary and Sufficient Conditions for Secure Hybrid Encryption, 2006.
- [jav] Jdk 8 pkcs#11 reference guide. <https://docs.oracle.com/javase/8/docs/technotes/guides/security/p11guide.html>. Last accessed 2019.05.12.
- [JUn] JUnit. Junit. <https://junit.org/junit4/>. Last accessed 2019.08.05.
- [JW01] He Jingmin and Min Wang. Cryptography and relational database management systems. In *Proceedings of the International Database Engineering & Applications Symposium, IDEAS '01*, pages 273–284, Washington, DC, USA, 2001. IEEE Computer Society.
- [KMKJR16] Ed. K. Moriarty, B. Kaliski, J. Jonsson, and A. Rusch. Pkcs #1: Rsa cryptography specifications version 2.2. <https://tools.ietf.org/html/rfc8017>, November 2016. Last accessed 2019.07.20.
- [LD00] Julio López and Ricardo Dahab. An overview of elliptic curve cryptography. Technical report, 2000.
- [Nita] Nitrokey. Installation nitrokey start windows. <https://www.nitrokey.com/documentation/installation#p:nitrokey-start&os:windows>. Last accessed 2019.08.03.
- [Nitb] Nitrokey. Nitrokey hsm 2 factsheet. https://www.nitrokey.com/files/doc/Nitrokey_HSM_factsheet.pdf. Last accessed 2019.04.22.
- [Nitc] Nitrokey. Nitrokey start factsheet. https://www.nitrokey.com/files/doc/Nitrokey_Start_factsheet.pdf. Last accessed 2019.04.22.
- [Nitd] Nitrokey. S/mime email encryption. <https://www.nitrokey.com/documentation/smime-email-encryption>. Last accessed 2019.08.03.
- [opea] Compiling and installing on unix flavors. <https://github.com/OpenSC/OpenSC/wiki/Compiling-and-Installing-on-Unix-flavors>. Last accessed 2019.08.04.
- [opeb] openssl. <https://www.openssl.org/>. Last accessed 2019.05.11.
- [opec] openssl engine. <https://www.openssl.org/docs/man1.0.2/man3/engine.html>. Last accessed 2019.05.12.
- [Oped] OpenSC. engine_pkcs11. https://github.com/OpenSC/engine_pkcs11. Last accessed 2019.05.13.

- [Opee] OpenSC. libp11. <https://github.com/OpenSC/libp11>. Last accessed 2019.05.13.
- [Opef] OpenSC. Openpgp-card. <https://github.com/OpenSC/OpenSC/wiki/OpenPGP-card>. Last accessed 2019.05.13.
- [Opeg] OpenSC. Opensc. <https://github.com/OpenSC/OpenSC/wiki/SmartCardHSM>. Last accessed 2019.05.13.
- [Opeh] OpenSC. Opensc. <https://github.com/OpenSC/OpenSC>. Last accessed 2019.05.13.
- [Oraa] Oracle. Java. <https://www.java.com/en/>. Last accessed 2019.05.12.
- [Orab] Oracle. keytool. <https://docs.oracle.com/javase/8/docs/technotes/tools/unix/keytool.html>. Last accessed 2019.08.11.
- [otBCIa] Legion of the Bouncy Castle Inc. Bouncy castle. <https://www.bouncycastle.org/java.html>. Last accessed 2019.05.12.
- [otBCIb] Legion of the Bouncy Castle Inc. Bouncy castle specifications. <https://www.bouncycastle.org/specifications.html>. Last accessed 2019.05.12.
- [Piv] Pivotal Software Inc. Spring boot. <https://spring.io/projects/spring-boot>. Last accessed 2019.08.12.
- [posa] Certificate authentication. <https://www.postgresql.org/docs/11/auth-cert.html>. Last accessed 2019.05.12.
- [posb] Encryption options. <https://www.postgresql.org/docs/11/encryption-options.html>. Last accessed 2019.05.12.
- [posc] PostgreSQL. <https://www.postgresql.org>. Last accessed 2019.05.12.
- [Pro] Shining Light Productions. Win32 openssl. <https://slproweb.com/products/Win32OpenSSL.html>. Last accessed 2019.05.13.
- [pub] Public key certificate. https://en.wikipedia.org/wiki/Public_key_certificate. Last accessed 2019.05.11.
- [PWHN18] Sven Plaga, Norbert Wiedermann, Gerhard Hansch, and Thomas Newe. Secure your ssh keys! motivation and practical implementation of a hsm-based approach securing private ssh-keys. 06 2018.

- [Sho01] V Shoup. A proposal for an ISO standard for public key encryption. <http://shoup.net/papers/>, 2001.
- [staa] Example workaround sunpkcs#11 providers missing support of oaep padding. <https://stackoverflow.com/questions/23844694/bad-padding-exception-rsa-ecb-oaepwithsha-256andmgf1padding-in-pkcs11>. Answer by user divanov on 2014.05.25. Last accessed 2019.05.12.
- [STAb] STACC Ltd. Research: Grants received. <https://www.stacc.ee/research/grants-received/>. Last accessed 2019.08.11.
- [sun] Jdk 8 pkcs#11 reference guide appendix a: Sun pkcs#11 provider's supported algorithms. <https://docs.oracle.com/javase/8/docs/technotes/guides/security/p11guide.html#ALG>. Last accessed 2019.05.12.
- [SVGE14] Erez Shmueli, Ronen Vaisenberg, Ehud Gudes, and Yuval Elovici. Implementing a database encryption solution, design and implementation issues. *Computers & Security*, 44:33 – 50, 2014.
- [uMB] user Maarten Bodewes. Ciphertext and tag size and iv transmission with aes in gcm mode. <https://crypto.stackexchange.com/questions/26783/ciphertext-and-tag-size-and-iv-transmission-with-aes-in-gcm-mode>. Last accessed 2019.08.06.
- [uni] Unit testing. <http://softwaretestingfundamentals.com/unit-testing/>. Last accessed 2019.08.05.
- [web] Web of trust. https://en.wikipedia.org/wiki/Web_of_trust. Last accessed 2019.05.11.

Appendix

I. Licence

Non-exclusive licence to reproduce thesis and make thesis public

I, **Mart Simisker**,
(author's name)

1. herewith grant the University of Tartu a free permit (non-exclusive licence) to reproduce, for the purpose of preservation, including for adding to the DSpace digital archives until the expiry of the term of copyright,

Security of Health Information Databases,
(title of thesis)

supervised by Jan Willemson and Dominique Unruh.
(supervisor's name)

2. I grant the University of Tartu a permit to make the work specified in p. 1 available to the public via the web environment of the University of Tartu, including via the DSpace digital archives, under the Creative Commons licence CC BY NC ND 3.0, which allows, by giving appropriate credit to the author, to reproduce, distribute the work and communicate it to the public, and prohibits the creation of derivative works and any commercial use of the work until the expiry of the term of copyright.
3. I am aware of the fact that the author retains the rights specified in p. 1 and 2.
4. I certify that granting the non-exclusive licence does not infringe other persons' intellectual property rights or rights arising from the personal data protection legislation.

Mart Simisker
14/08/2019