University of Twente

Master Thesis

Securing Patient Information in Medical Databases

Author:
Elmer Lastdrager

Supervisors:
Dr.ir. Ander de Keijzer
Dr. Svetla Nikova
Ghita Berrada MSc.
Dr. Qiang Tang

August 2011

Abstract

In hospitals, medical data is stored in databases. These *medical databases* store anything from diagnoses to *patient information*. Some of the data in a medical database is sensitive and access to this data should be limited to authorized persons. Furthermore, the integrity of the data should be protected to prevent unauthorized persons from making alterations. Currently, the medical database itself controls access to the data to prevent unauthorized disclosure of, and control alterations to, the data. However, this puts a lot of trust in the database. The database itself can access and alter the data and therefore the database administration can as well. If the database server is subject to a successful hacking attempt, all data stored in the database is visible to the attacker. We aim to reduce the risk of *information leakage* and we want to protect the *integrity of the data*, without trusting the database. Even if the database server is compromised, the data remains confidential and any alterations to the data can be detected easily.

We identified entities who have, or should not have, access to the database and discussed the security requirements of a medical database. We discussed several encryption schemes that can be used to provide confidentiality of the data, in particular *Type-Based Proxy Re-Encryption*, and signatures schemes, such as the *Bilinear Aggregate Signature Scheme*, to provide data integrity. A prototype of a secure medical database was implemented and run to compare the performance of a secure medical database against a non-secure medical database.

Our contributions are a theoretical discussion on the security of a medical database, the implementation of a prototype to simulate a secure medical database and the results of several experiments that we conducted. In this thesis, we show that the performance impact of providing confidentiality and integrity within a medical database is considerable. Even though our prototype is relatively slow, in practice the impact is probably less. If a doctor has to wait only one second to retrieve information of a patient, as opposed to waiting a few milliseconds with the non-secure medical database, the security benefits will outweigh the performance impact. Furthermore, by using a re-encryption scheme, a part of the decryption process, namely the re-encryption, can be offloaded to a proxy, thereby spreading the computational costs. Additionally, we have shown that our prototype scales linearly, which is an interesting property when large databases need to be secured. This leads to the conclusion that securely designing a medical database is possible without putting trust in the database itself.

Acknowledgements

First of all, I would like to thank my supervisors for their feedback during the development of this thesis. I would especially like to thank Ander for his guidance and for allowing me to work as teaching assistant in two courses. I greatly enjoyed the, sometimes hopeless, mission of attempting to teach technical medicine students how to write software in Java. I owe Qiang many thanks for his help in discussing and implementing the cryptographic theory, as well as his helpful feedback on the many draft versions of the thesis. Many thanks as well to Ghita for many corrections on database theory, the results and my English.

I had the ability to work on my thesis at a desk at the university, which allowed me to meet many other Master students, PhD students and employees. Especially my roommate Shaun, with which I had the best time, and former employee Michiel "the monkeyman". Together with Esther, Marleen, Mark, Benjamin, Joost, Kaj, Bas-Jan, Berkan and many others, they made me feel very welcome and they are the reason that I loved every single lunch and coffee break.

Furthermore, I would like everybody from the icehockey team The Slapping Studs for the good times. The Studs have given me every opportunity to get on the ice. I love playing icehockey and, during the writing of my thesis, I even became a referee. In particular, I would like to thank Marianne for all the "pepernotenpauzes" (this is not translatable or even an existing word) during the first few months of working on my thesis, and Elsemieke, Mathijs, Lex and Niek for coming over for a chat, every now and then.

Last, but certainly not least, I want to thank my brother Casper, sister Birgit, father Edwin and mother Rita for their support and understanding during the entire study.

Around computers it is difficult to find the correct unit of time to measure progress.
Some cathedrals took a century to complete.
Can you imagine the grandeur and scope of a program that would take as long ?

– Epigrams in Programming, ACM SIGPLAN (September 1982)

Contents

List of Figures

1 Introduction

A growing trend in hospitals is digitalisation, where documents containing sensitive patient information are stored digitally. This raises concerns about the security of the documents being stored, compared to storing the documents on paper, such as the *confidentiality* and *integrity* of the documents. Confidentiality of patient information was an issue even when it was still stored on paper. A malicious person could enter the hospital and steal the paper documents. Moreover, hospital staff can read paper documents they are not supposed to read, as long as they have physical access to the document. Confidentiality of paper documents is enforced by putting them behind locks. Basically, it comes down to enforcing physical access control to the document. When a document is stored digitally, similar issues arise. There is still the issue of access control, namely the question of who is allowed to read or change the document, and there is the issue of easy copying. It is much easier to copy a digital document than it is to copy a document on paper. An adversary who gains access to a workstation in the hospital, can copy documents from that workstation. This is a problem if those documents contain patient information. The main difference with paper documents is that an adversary does not need to be in the hospital itself. Hacking a workstation, located in the hospital, which is connected to the internet is sufficient to gain access to patient information. In the past, hackers have obtained access to servers with medical data on it by, for example, hacking a server [37, 18] or by stealing a laptop with patient records of millions of patients [13]. With regards to integrity, it is generally easier to change a digital document, than to change a paper document, as changes on paper are more noticeable and physical access is required to be able to make the change.

Objectives

In this thesis, we aim to reduce the information leakage and protect information integrity of patient information that is stored in a central location, so that the data can be accessed only by authorized persons and that the integrity of the data can be verified. We consider a medical database that contains patient information, measurement data and diagnoses related to the patients. Measurement data could be anything from brain scans to lab reports. We discuss methods that can be used to secure the information and protect the integrity of the data and the link between the patient information, the measurement data and the diagnosis. We develop a prototype that implements these security measures in a medical database and use the prototype to test the viability of the methods.

Research Questions

We identified several research questions that we would like to see answered. Our main research question is

How to secure a medical database?

We seek to investigate the answer to our main research question by answering the following questions.

How can patient information be encrypted, so that the database cannot decrypt it?

How can we protect the integrity of medical data within a database, if we do not trust the database?

What is the performance impact of providing confidentiality and integrity of the information?

Outline

The outline of this thesis is as follows. In Chapter 2, we discuss what medical data is, how it is being used and the context of our system compared to electronic health records. In Chapter 3, the topology of the system is discussed. Furthermore we establish a set of requirements to the system. Chapter 4 lists the cryptographic building blocks needed to build our system. We discuss which algorithms are used. Chapter 5 is about the experimental setup. We discuss how the building blocks are implemented in a system and how we will execute the experiments. The results of our experiments are discussed in Chapter 6. Finally, we discuss the conclusions of this thesis, and discuss future work, in Chapter 7.

2 Storing Electronic Medical Data

Nowadays, large amounts of medical data are collected and stored electronically. Storing medical data, like storing regular data such as holiday photos, can be done in several ways. One could store holiday photos as files in a certain directory structure, for instance in the "Pictures" folder of your home directory. However, this type of solution is not suitable for storing medical data, as medical data requires more meta data, other than the filename, and it needs to be searchable at low cost (without having to read all the files for each search). A commonly used method for storing the data is in a database. It is possible to retrieve, insert or delete data using a Database Management System (DBMS). The DBMS takes care of the consistency of the data. Using a database enables more efficient sharing of data between researchers and/or hospitals. We refer to a database that contains electronic medical data as a *medical database*.

In this chapter, we discuss medical data and all issues that come with processing it. In Section 2.1, we explain what is meant with medical data. Then, in Section 2.2, we describe the usage of electronic medical data, the context of our system and what usage we focus on in this thesis.

2.1 What is Medical Data?

Medical data can be anything from health records (such as patient information or diagnoses) to raw sensory data, such as samples of an EEG measurement. We distinguish two types of medical data, namely sensitive data, which contains patient information or can be linked back to a patient, and non-sensitive data. Sensory data, often referred to as measurement data, only contains samples of sensors and we consider this to be of the non-sensitive type. In the case of EEG, the sensory data consists of numbers representing voltages measured at the scalp of a person. In some cases, such as when an EEG measurement is stored in the EDF+ format [24], patient information is also stored in the datafile, effectively putting the entire datafile in the sensitive data category. To avoid this, medical data can be stored in a database. In the example of an EEG measurement, the sensory data will be stored in a datafile but the identifying information is stripped from that file. All identifying information is then stored in a medical database, together with a reference to the data file where the sensory data is stored. The data that is stored in the medical database is called meta data: it describes the actual data.

A medical database is, as stated in the introduction of this chapter, a database which stores medical data. In this thesis, we consider *relational databases*, as opposed to other types of databases, such as XML databases. However, the methods discussed in this thesis can be applied to non-relational databases as well. A relational database (from now on referred to as database) consists of tables and each table consists of rows and a fixed number of columns, forming a matrix where the rows, vertically, expand when data is added. Alternatively, one can view a table as a set of tuples. A visual representation of a table is given in Figure 2.1.

id	name	postal code
1	Alice Grotskiv	K96756
2	Bob Brummelbosch	M17391
3	Charlie Yinis	K59728
4	Diederik van Hamsteren	A87208
5	Erica VanMossel	F22679

Figure 2.1: A database table filled with some example data.

Medical databases vary in size, purpose and usage. They range from small databases located in a department of a hospital, where they store diagnoses, to national health record systems, and anything in between. Some databases contain patient information, such as a patient's name, others contain only anonymised information, used for statistical purposes or research.

2.2 Usage of Medical Data

We focus on a medical database management system with patient information, sensory data and meta data. The patient information is confidential and needs proper security. The sensory data and meta data are both not considered confidential, however, the association between them should be protected against corruption.

We first discuss Electronic Health Records in general, after which we discuss a specific medical database called MeDIA.

2.2.1 Electronic Health Records

Electronic Health Records (EHRs) are an example of a medical database. According to [21], EHR is

> "a repository of patient data in digital form, stored and exchanged securely, and accessible by multiple authorized users."

EHR systems provide access to health records to health care professionals and administrative staff. In [21], several types of EHRs are distinguished, such as the Electronic Medical Record (EMR) and Personal Health Record (PHR), while others do not make the explicit distinction between an EMR and EHR and consider them equal. An EMR contains information which is entered by a single hospital department, an entire hospital or parts thereof, or even information from multiple hospitals. Typically, only hospital staff will add information to the EMR.

A PHR, on the other hand, is controlled by the patient and contains information that is (at least partly) entered by the patient. It is defined in [14] as

> "an electronic application through which individuals can access, manage and share their health information, and that of others for whom they are authorized, in a private, secure and confidential environment."

A fundamental difference between EMR and PHR is that a PHR system needs to represent the health record in a way such that the patient can easily understand the information presented to him. The patient controls his health record and decides who to give access to his health record to. In an EMR system, only clinicians and administrative staff have access, so the presentation of the health record can be different, similarly to health records on paper.

In this thesis, we do not consider personal health records. Instead, we consider an electronic medical record system within a hospital. Such electronic medical records are widely used [16, 23]. However, connecting these systems to the internet could result in a security breach [22]. Protecting the servers that store the electronic medical records is therefore very important.

In order to discuss the security of electronic medical records, we focus on a medical database application called MeDIA.

2.2.2 MeDIA

MeDIA (short for Medical Data Integration Application [15]) is the name for the medical database management system that is currently being developed. The sensory data that MeDIA can store includes EEG, MRI or CT, together with its corresponding meta data, such as patient information and annotations. The main advantage of MeDIA over simply storing the binary data is that MeDIA handles *lineage*, *uncertainty*, *versioning* and is able to combine different types of data, for example, EEG and MRI. Furthermore, when someone wants to view EEG and MRI data from a single patient, he can query MeDIA, instead of having to query the different databases separately.

There are three features, lineage, uncertainty and versioning, that make MeDIA different from regular electronic medical record systems.

Lineage [5] is used to show where a particular data item is derived from. For example, if there are data items a, b, c and c is defined as $c = a + b$, c is being derived from a and b. This could be used in, for example, a diagnosis table. Every diagnosis lists the sensory data it is based upon. If, for some reason, a particular set of sensory data has to be rejected due to a wrong recording, it is easy to determine which diagnosis needs to be reconsidered.

Uncertainty [5] in a database means that items in the database have a certain probability attached to them. For example, storing weather predictions in a database involves probabilities. The prediction could say "80% chance of a sunny day, 15% chance of clouds and 5% chance of rain". The database will store all three possibilities (sun, clouds, rain) with the probability of occurrence.

Versioning is the ability to track previous versions of data items. For example, suppose the university stores all salaries of the employees in a database. Monica, an employee, gets her salary raised. Using versioning, Monica's old salary is archived and her "current" salary is updated. The database, thus, stores for each employee the current salary and all previous salaries. In a more medical setting, this could be used to save old diagnoses.

MeDIA consists of several components, such as the Application Programmer's Interface (API) and feature extractors. Figure 2.2 shows the design of MeDIA. User software communicates to the system through the API. MeDIA takes care of storing the sensory data, such as EEG values, extracting features and adding security (such as encryption of sensitive data). Feature extractors extract patterns or features that can, for example, allow artefact rejection, from the raw data. All extra information generated by components of MeDIA are stored in the Meta database. The MeDIA API transforms queries of the user application, to queries on the meta data and on the data files. Queries to the API are written in a language called MediQL. MeDIA transforms this query in subqueries to the different databases, which can be SQL, XQuery or, in the case when sensory data is stored in regular files, filesystem calls.

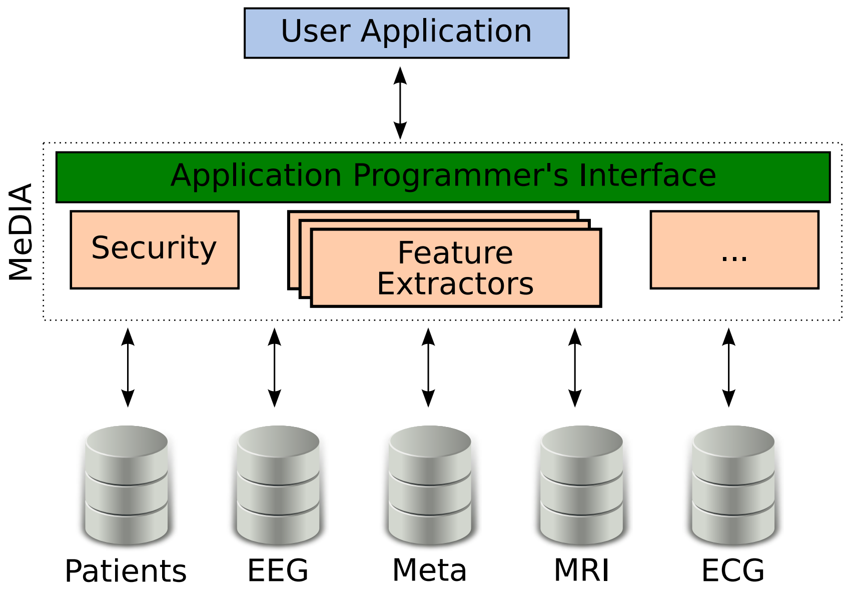


Figure 2.2: Components of MeDIA

3 System Design

In this chapter, we will discuss the topology of the system on which we base the security requirements. We introduce an alternative topology, where the medical database is hosted outside of the hospital network. Finally we define the security requirements to the system.

3.1 Topology

In order to implement security in a medical database, we first have to establish the topology that is currently being used and upon which we will implement the security. We assume that a medical database is being used by a hospital to store patient and treatment information. The topology of the database and the workstations querying the database is shown in figure 3.1.

The hospital has an internal network which is marked in the figure by a gray background. A firewall filters all traffic from and to the internet. The firewall is configured in such a way that it does not allow incoming connections from the internet to the database. Access to the database, therefore, is restricted to any workstation within the hospital network. This can be considered a good practice, as it may mitigate some attacks from hosts beyond the network, such as an attack on vulnerable software running on the hospital's servers. However, since there are many computers within the hospital network itself, there are still possibilities to attack the database. One may infect a workstation within the network with a virus, for example by using a phishing attack [17], thereby gaining access to the network and thus bypassing the firewall.

The medical data is stored as in the local hospital setting and it consists of EEG readings, annotations to these readings and patient information of each patient who has had an EEG recording. The database is only accessible from within the hospital network and, as we stated earlier, a firewall enforces this policy. The access control of the database is dealt with by using credentials. These credentials are given only to a selected group of employees, namely the employees who are allowed access to the data. In this situation, there is a system administrator who grants access to (other) employees, so obviously the administrator himself can also access the database.

Within the database, all data, including patient information, is stored without any form of encryption. The access control, based on credentials, only allows certain employees to access the data and is thus enforced by the database only. To the best of our knowledge, access is given to an entire table, and not to a specific set of records (rows), while this has been considered bad practice for a long time [1].

In Figure 3.1, each workstation represents a user that is connected to the network. A user can be a doctor, a nurse or some other entity, such as an administrator. The users connect to the database over the network and, if they are granted access with their credentials, execute queries.

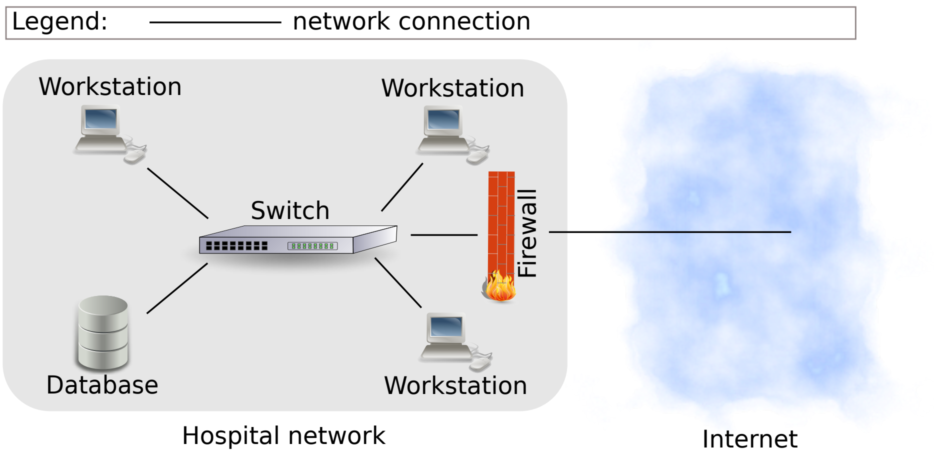


Figure 3.1: The topology upon which we base our analysis.

3.2 Alternative Topology

In the topology we previously discussed, the database was located within the hospital network. We now introduce an alternative topology in which the database is not hosted within the hospital's network, but externally. The alternative topology, as shown in Figure 3.2, looks very similar to the topology from Figure 3.1, except for the database. The database could be a server under control of the hospital, for example an owned server in a datacenter of a third party. It could also be a dedicated server for which the hospital has control over the software and where a third party provides the connection and hardware. A third possibility could be a hosted solution where both the server hardware and software is taken care of by a third party, and where the hospital can query the database without having to worry about configuration or server hardware.

The motivations for hosting the database server externally are flexibility, scalability and cost reduction. We will refer to an externally hosted database server as an *outsourced database*, and to externally hosted data as *outsourced data*. Mostly, hospitals want to focus on their core business, which is healthcare. Hosting a database server can be outsourced to a company with expertise in hosting. There are several advantages and disadvantages to the outsourcing of data, which we will discuss in Section 3.2.1.

The data stored in the database is accessible by a third party, namely the organisation which hosts the servers. In the case that a server (or resources on a server) is rented from a third party, the hospital would be putting data on a server owned by that third party. Even though there may be sufficient contractual agreements to make sure the third party will not access the data, it would be better to avoid this situation by technical means. We consider a database which is hosted by a third party to be untrusted. Basically, we assume there are no guarantees about integrity and confidentiality of the data. The connection between the clients and the database can be established over a private connection, such as a dedicated network connection, or a public connection, such as the internet. Especially when the traffic between the hospital and database travels over the internet, the risk of somebody eavesdropping should be taken into account, however, encrypting the connection between the hospital and the database could prevent eavesdropping.

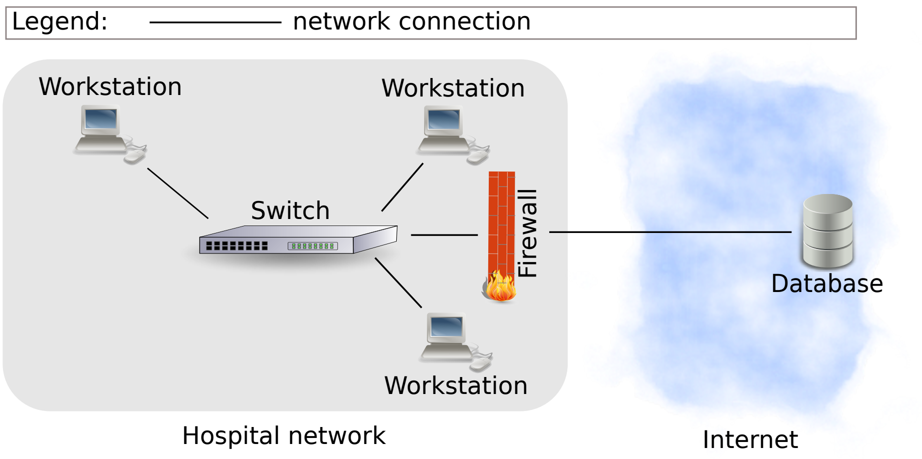


Figure 3.2: The alternative topology with the database outsourced.

3.2.1 Why Outsource Data?

The outsourcing of data is nowadays often related to Cloud Computing, which is in turn one of the building blocks for Utility Computing. The word “cloud” in Cloud Computing refers to the combination of datacenter hardware and software. According to [2], cloud Computing

> “refers to both the applications delivered as services over the Internet and the hardware and systems software in the datacenters that provide those services.”

Utility Computing refers to a service that uses Cloud Computing. There are several variations of Utility Computing, such as, among many others, Software as a Service (SaaS) or Platform as a Service (PaaS). An example of Utility Computing is service of Amazon called Relational Database Service (RDS). With Amazon RDS, a database runs on the servers of Amazon and, as customer, one can execute queries on it, while paying for the actual usage of the database. Under the umbrella of Web Services, Amazon provides many other services that qualify as Utility Computing, such as content distribution, database functionality or DNS hosting. Other companies, such as Google (AppEngine), Microsoft (Azure) and Apple (iCloud), provide similar services that qualify as Utility Computing.

Outsourcing the storage of data refers to delegating the storage of data to a third party. Potential benefits include the pay-as-you-go model, where you only pay for the actual usage, whereas normally you would have to buy storage (such as hard disks, or storage servers) in advance, and the possibility to mitigate risks, such as downtime caused by hardware failure, to the third party. There are drawbacks obviously, such as the lack of confidentiality of the data, as the data is stored on the hardware of somebody else, so there is no physical control over the data. Since the data is stored elsewhere, the data owner doesn’t have complete control in certain situations, such as when the storage provider cancels the contract or goes bankrupt.

Finally, there are two different kinds of “clouds”, namely a public cloud, such as the services provided by a third party, and a private cloud, which refers to the infrastructure in an internal datacenter of an organization. In the situation where a private cloud is used, the organization has full control over the hardware and network. The advantage of delegating certain risks is lost, compared to a public cloud, but there can still be savings in other areas such as energy consumption, by shutting down servers when they are inactive. When data containing sensitive information is to be stored, both a public and a private cloud can be considered. Even in a private cloud environment, sensitive data can still be accessed by administrators or other employees that have access to the infrastructure.

3.3 Security Requirements

As we consider storing (sensitive) patient information in a database, there is a need for security: an adversary should not be able to obtain the sensitive data. To be able to secure the system, we need to define a set of requirements that we can solve. All requirements in this section are valid for both topologies that were discussed in Sections 3.1 and 3.2. We first discuss the security properties. Then, we will define the entities that use our system, after which we will discuss who the adversary is. Finally, we will discuss the requirements to the security that we will take care of.

3.3.1 Definitions

We need our system to be secure and therefore we first need to establish some security definitions. According to Avižienis et al [3], security is a composite of several attributes, namely Confidentiality, Integrity and Availability (CIA). These are the primary attributes, whereas secondary attributes, such as accountability, authenticity and nonrepudiability refine the primary ones.

Confidentiality describes whether data (or information) is accessible by authorized users only. Confidentiality is usually enforced by means of encryption.

Integrity is the absence of improper alternations to the data, or system. Improper, in this case, can be substituted with “unauthorized” to conform to the usual definition. However, in [3], the definition is broadened to include alterations to the system that prevent correct functioning of a system. Note that this definition of integrity differs from the usage in the database world, where integrity means a consistent state. Integrity with respect to security is all about avoiding alternations, for example, changing a website to include malicious code.

Availability is the readiness of a system or service to serve clients. Consider, for example, a webserver, hosting several websites, that crashes after being exploited by hackers. This webserver is no longer processing requests and thus the websites are no longer served. Here, the availability of the webserver was targeted.

The secondary attributes are defined as follows.

Accountability refers to the level of availability and integrity of the identity of a person who performs an operation, such as storing an entry in a database. In other words, the identity should be known and requestable, if needed.

Authenticity refers to the integrity of a message, and possibly some metadata related to it, such as origin.

Nonrepudiability consists of two parts. First, nonrepudiation of message origin means the sender cannot deny having sent a message, e.g. availability and integrity of the identity of the message origin. Nonrepudiation of reception means the receiver cannot deny having received a message, e.g. availability and integrity of the identity of the receiver.

3.3.2 Entities

In a hospital, there are several entities dealing with confidential data or otherwise have access to (a part of) the system. There are entities who should have access to sensitive data, such as a doctor, and employees who should not have access to sensitive data, such as an administrator. Additionally, there may be a malicious outsider who wants to get access to the data. The entities in the topologies discussed earlier, are:

Doctor: has access to the data of his own patients, but not to the patients of another doctor.

Nurse: has access to the patient information of patients she is responsible for.

Secretary: has access to (for example) insurance information, or name and home address, of the patients of all doctors within the department.

System Administrator: responsible for taking care of the operation and/or maintenance of the system. The system administrator has access to all physical machines. He should not have access to any patient information.

Database Administrator: administers and maintains the database itself and therefore has access to the database. In a system where the database is in the hospital, this person may be the same as the system administrator. He should not have access to any patient information.

Other Employee: can be any employee other than the ones already discussed. This could be a person cleaning the rooms (who gets access to a logged-in computer), or a service engineer.

Hacker: tries to hack into the system in any way possible, either for a profit or “just for the fun of it”. We assume the hacker is not an employee (otherwise, he would be an “other employee”).

3.3.3 Adversary

In order to properly describe the requirements to the security of the system, we need to have knowledge of who the adversary is. Basically, any of the entities listed above can act as adversary. Most notably is the hacker, as an outside threat. The hacker should have no access to the system. Any access obtained, being either read or write access to the data, is considered harmful. Moreover, a hacker could target the availability of the system, for example, to blackmail a hospital.

Doctors, nurses and other employees, such as a secretary, can also be viewed as adversary if they want to obtain more privileges than they actually have. For example, a doctor could try to view information about a person who is not his patient. There have been several examples of such misuse. Similarly, nurses and secretaries could also try to view patient information from patients they are not allowed to view the information of. The category “other employees” are all hospital employees who have no rights to view patient information, but who have access to the hospitals internal network and have physical access to at least one department within the hospital.

Finally, there are the administrators that could be seen as adversary. Administrators would also fit in the category “other employees”, however they deserve a special note, as they often have full physical control over workstations and servers. As they administer the network and IT systems, they have more power than the regular employees. Our system has to be able to withstand a rogue administrator, trying to read, corrupt or delete patient information from the database. There is one subtype of administrator that we treat separately, namely the database administrator. If the data itself, including patient information, is outsourced, then the database administrator is likely to be an employee of the third party responsible for the storage of the database. Just as with the system administrators, the database administrators should not have any access to sensitive data.

3.3.4 Requirements

The security requirements to sensitive data that we take care of, are as follows.

1. Confidentiality: sensitive data (such as patient information) should only be accessible to a limited number of people that should have access to the data. This means a doctor treating a patient should get access to this patient's data, but a doctor from another department, who has nothing to do with that particular patient, should not be granted access, for example.

2. Integrity of data origin: it should be possible to verify that data is stored by an authorized person, and not by, for example, a database administrator, who happens to be able to write data to the database.

3. Integrity of data: retrieved data should be checked to ensure that it was not altered, either on purpose or due to data corruption.

We note that not all data may need the above security requirements. Sensory data, for example, could be protected for integrity only, if it does not contain sensitive information or information that could (in)directly identify a patient.

For the system itself, availability is an additional requirement. Doctors should be able to view the patient information of their patients. We delegate a part of this availability to the database storage provider, by assuming that the database is always available. The responsibility of availability for our system itself consists of the availability of the keys and methods for processing the decryption and verification functions.

4 Building Blocks

In the previous chapter, we described the system design without discussing security-specific details that are required to the system. In this chapter, we identify the building blocks for implementing the CIA properties (Confidentiality, Integrity and Availability).

But first, we introduce some notation that is used throughout the chapter.

4.1 Notations

We use $\mathbb{Z}$ to denote all integers, e.g. $\mathbb{Z} = \{0, \pm 1, \pm 2, \dots\}$. To denote the group $\{0, 1, 2, \dots, q-1\}$ under addition modulo q, we write $\mathbb{Z}_q$. Furthermore, $\mathbb{Z}^+$ are all positive integers: $\mathbb{Z}^+ = \{1, 2, 3, \dots\}$.

More generally, for a certain group $\mathbb{G}$ we define $\mathbb{G}^*$ to be the group $\mathbb{G}$ without its identity (or neutral) element e, e.g. $\mathbb{G}^* = \mathbb{G}\backslash\{e\}$.

The $\oplus$ is the bitwise exclusive or (XOR) operation and we use $||$ for denoting concatenation.

4.1.1 Bilinear Pairing

Many of the algorithms that we discuss that can be used to provide confidentiality and integrity use bilinear pairings to achieve their cryptographic strength. A bilinear pairing [10, 27] is a mapping of two groups onto a third group, each group being a multiplicative cyclic group of prime order, which we denote as $\hat{e} : \mathbb{G}_1 \times \mathbb{G}_2 \rightarrow \mathbb{G}_T$. Alternatively, it can be written as a function: $z = \hat{e}(x, y)$ for $x \in \mathbb{G}_1, y \in \mathbb{G}_2, z \in \mathbb{G}_T$. Every bilinear pairing has the following properties:

1. The pairing must be efficiently computable, e.g. there exists an algorithm that can efficiently compute the pairing.
2. The pairing must be bilinear: for all $u \in \mathbb{G}_1, v \in \mathbb{G}_2$ and $a, b \in \mathbb{Z}$ the equation $\hat{e}(u^a, v^b) = \hat{e}(u, v)^{ab}$ must hold.
3. Finally, the pairing must be non-degenerate. This means it cannot be a trivial map where everything is mapped to the identity element of $\mathbb{G}_T$. In order words, for a generator g_1 of $\mathbb{G}_1$ and a generator g_2 of $\mathbb{G}_2$ the equation $\hat{e}(g_1, g_2) \neq 1$ must hold.

It is possible that $\mathbb{G}_2 = \mathbb{G}_1$. In that case the pairing can be written as $\hat{e} : \mathbb{G}_1 \times \mathbb{G}_1 \rightarrow \mathbb{G}_T$.
For more information on pairings and the associated complexity assumptions, we refer to [27].

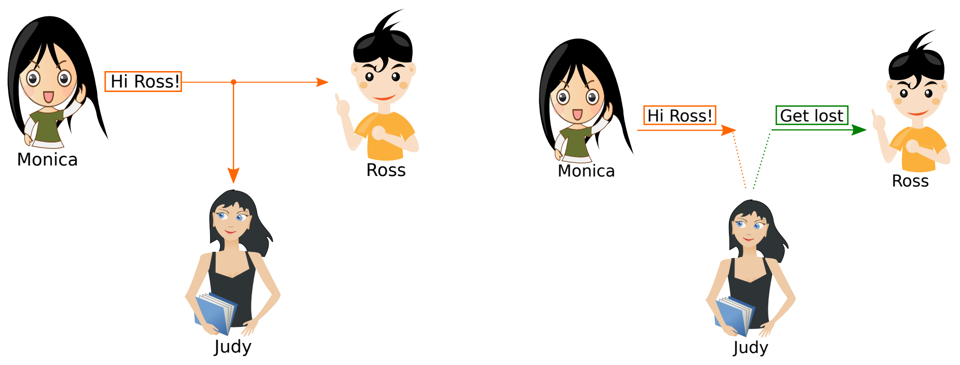
4.2 Confidentiality

In Section 3.3.1, we defined confidentiality as the property that describes whether data is accessible by authorized users only. As patient information can be stored in a medical database, and it generally is, confidentiality of the data is an important issue. The reason for this is that patient information contains data such as name, address or a social security number, which can be linked to diagnoses. Even if there is no direct link with a diagnosis, information about the health of a patient may leak. This is the case when, for example, the patient information of a person is found in a database with Epilepsy recordings. If this information is leaked, the patient may be rejected for insurance or jobs later on in his life, even if he had just a one-time epileptic seizure.

4.2.1 Attacking Confidentiality

Before we can discuss the different ways of ensuring confidentiality of a message, we first discuss which attacks on confidentiality are possible.

We distinguish two types of attackers. The first type is an passive attacker. A passive attacker can eavesdrop and thus read all communication. Wiretapping an internet connection and recording all activity is an example of a passive attack. Figure 4.1a shows an example of a passive attacker. The second type is an active attacker, who can not only eavesdrop, but also alter the communication. Clearly this attacker has much more power compared to a passive attacker. An example of an active attacker is a hacker who gets all traffic from a network switch to go through her computer. If Monica sends a message to her brother Ross, the active attacker Judy could change the message completely before relaying it to Ross. This is shown in Figure 4.1b.



(a) Passive attacker (b) Active attacker

Figure 4.1: Showing the difference between a passive attacker, who only eavesdrops, and an active attacker, who also alters messages. In the right picture, Judy changes the message of Monica such that Ross would think Monica wants him to leave.

An attacker can try to find the plaintext for a certain ciphertext by doing cryptanalysis, the science that tries to recover the plaintext from a ciphertext without having the decryption key. Schneier [32] defines several types of attacks that can be performed. We list some of them below.

Ciphertext-only attack: the attacker only knows several ciphertexts. The goal is to obtain as many plaintexts as possible or, if possible, the decryption key. If the cryptosystem is deterministic, e.g. two encryptions of the same plaintext lead to the same ciphertext, one may be able to find the plaintext by analysing the ciphertexts. This can be avoided by adding randomness during encryption, which we will discuss later.

Even when the encryption is random, one can sometimes obtain information from the plaintext by looking at the size of the ciphertext or by performing frequency analysis on the ciphertext. If we use a cryptosystem that preserves the length of the plaintext, or appends a fixed number of characters to the plaintext, one may be able to figure out whether ciphertexts are likely to be equal, by observing a set of encrypted plaintexts and taking note of the length of each ciphertext. This can be partly avoided by using a block cipher with a block size sufficiently large. A block cipher fits the plaintext in blocks and, if needed, appends dummy data at the end, so that all the blocks are filled with data to be encrypted.

The attacks on the GSM protocol by Barkan, Biham and Keller [4] are an example of ciphertext-only attacks.

Known-plaintext attack: the attacker knows several ciphertexts and the corresponding plaintexts. The goal of the attacker is to find the key, or to find a way to decrypt any future encryptions with this same key. A Caesar cipher can be attacked easily with a known-plaintext attack. A Caesar cipher works as follows. Encrypting a plaintext in a Caesar cipher is simple: shift each character three places to the right, rotating if required. This means the character ‘a’ is substituted with ‘d’, ‘b’ with ‘e’ and ‘z’ with ’c’. Decryption is performed the other way around, by shifting each character in the ciphertext three places to the left. More generally, the number of places shifted in Caesar cipher can be changed to any number between 1 and 25. The key k to the cipher would then be the number of places shifted. In our example above, we used $k = 3$. The known-plaintext attack on the Caesar cipher is easy: to find `k`, only 1 character of the plaintext and it’s corresponding character in the ciphertext is needed.

In [7], Biham and Kocher describe a known-plaintext attack to the encryption of PKZIP compressed archives.

Chosen-plaintext attack: this attack extends the known-plaintext attack. Not only does the attacker know ciphertexts and the corresponding plaintexts, also he can choose a plaintext to be encrypted. The goal is to obtain the key, information about the key or to find a way to decrypt any future encryptions. This attack is powerful, as the attacker can choose the plaintexts to be encrypted. One may send a message to an ambassador, which in turn sends it, in encrypted form, back to his country. In that case, the message, or at least a part of it, is chosen by the attacker. Alternatively, nowadays encryption is performed in devices such as gaming consoles or tablet computers. The encryption is performed in the software or hardware. If the attacker owns such a device, and thus has control over the hardware, he may be able to feed plaintexts into the encryption mechanism.

A variation on the chosen-plaintext attack is called the **adaptive-chosen-plaintext attack**, where the attacker can not only choose the plaintext, but he can modify his choice based on the resulting ciphertext. Basically, instead of being able to choose just one plaintext as with the chosen-plaintext attack, the attacker can choose many plaintexts.

Chosen-ciphertext attack: the opposite of the chosen-plaintext attack. Instead of choosing the plaintext to be encrypted, as with the chosen-plaintext attack, the attacker chooses the ciphertext to be decrypted. There exists a variation on this attack, the Adaptive-chosen-ciphertext attack, where the attacker can choose many ciphertexts to be decrypted.

Rubber-hose attack: try to obtain the key by any means. This means the attacker can threaten, kidnap, bribe, torture or blackmail the person who has possession of the key. Basically, this is one of the most effective ways of obtaining the key.

Having defined the attacks, we now proceed to explaining two types of cryptographic algorithms: symmetric and a-symmetric.

4.2.2 Symmetric Cryptography

Symmetric-key cryptography is a form of encryption where the decryption key `d` is equal to, or can be easily derived from, the encryption key `e` [28]. In most symmetric-key cryptography schemes `e` equals `d`. The key is often called a shared key as both the sender and the receiver of a message know the key.

Suppose Monica and Ross want to communicate with each other, but they do not want an outsider, Judy, to read their messages. They can go to a safe location and talk in person and make sure nobody is in the neighbourhood to overhear them. However, if both Monica and Ross want to communicate over the Internet, they need to use encryption to prevent Judy from reading their communication. Before they can start sending encrypted messages to each other, several steps have to be taken. Schneier [32] distinguishes five steps to transmit a confidential message.

1. Cryptosystem: both parties have to agree on a cryptosystem, e.g. an algorithm for encryption and decryption.
2. Key: both parties have to agree on a key.
3. Encryption: Monica will have to encrypt a message with the key, using the cryptographic algorithm, to obtain the ciphertext.
4. Transmission: Monica sends the ciphertext to Ross.
5. Decryption: Ross decrypts the ciphertext with the algorithm and key that were established in 1 and 2.

Suppose Judy, the evil mother of Monica and Ross, wants to read all the communication of her children. Judy can intercept all online communication between Ross and Monica, as if she sits in between both of them, and thus she will be able to read the ciphertext sent in step 4. Judy will not be able to decrypt the ciphertext and read the message if the algorithm (step 1) is sufficiently strong, and the key (step 2) is only known to Monica and Ross. The algorithm does not have to be a secret. If designed properly, the security depends on the key and not on the design of the algorithm. Therefore Monica and Ross should only keep the key (step 2) secret.

We use the following notation to encrypt a message `m` to get the ciphertext `c`, and then decrypt the ciphertext to obtain the message again, all using the key `k`.

$$E_k(m) = c$$

$$D_k(c) = m$$

Here, E and D are the encryption and decryption operation, respectively.

One of the disadvantages of symmetric cryptography is establishing the shared secret, the key. When an attacker gains possession of the key, he can decrypt all ciphertexts that were encrypted with that key. However, the main problem is to obtain a common key that only the participants have. A part of the solution to this problem is asymmetric cryptography.

4.2.3 Asymmetric Cryptography

Whereas symmetric cryptography uses an encryption and a decryption key that are equal, or can be easily derived from one another, asymmetric cryptography has different encryption and decryption keys, and computing the decryption key from the encryption key is infeasible [28]. The advantage of these systems is that the encryption key can be made public. Therefore, this system is also called *public-key cryptography.* Schneier [32] describes several steps to securely transmit a message.

1. Cryptosystem: both parties have to agree on a cryptosystem or algorithm to use. RSA and ElGamal are examples of such a public-key algorithms.
2. Key: the message's sender has to obtain the public key of the receiver
3. Encryption: the sender encrypts the message using the public key of the receiver.
4. Transmission: the ciphertext is transmitted to the receiver.
5. Decryption: the receiver decrypts the ciphertext using his private key.

To go back to the example of Monica and Ross, Ross can publish his encryption key, for example, by putting it on his website. Monica can then download the encryption key, or public key, of Ross and encrypt a message. Judy cannot retrieve the message from the sent ciphertext without the private key of Ross. However, since the encryption key is public, Judy can encrypt a message of her own and send it to Ross. The availability of public keys dismisses the need to establish a shared secret, given that the sender has the public key of the receiver.

We use the following notation to describe encryption and decryption using asymmetric keys.

$$E_{pk}(m) = c$$

$$D_{sk}(c) = m$$

Where E and D are encryption and decryption, respectively, `sk` is a private (secret) key, `pk` is the corresponding public key, `c` is the ciphertext and `m` is the message.

Encrypting and decrypting messages using asymmetric, or public key, encryption is much slower than symmetric encryption [32]. A solution to this problem is to use asymmetric encryption to exchange a symmetric key, called a session key. The session key is used to encrypt the message. Furthermore, since the encryption key is public, a chosen-plaintext attack is possible. Suppose, for example, that an adversary intercepts a certain encrypted (short) message. The adversary can then encrypt all possible plaintexts using the encryption key and check whether the resulting ciphertext equals the ciphertext he intercepted. This can be mitigated by either using a *non-deterministic encryption*, or by using session keys. Non-determinism, or sometimes called *randomisation*, adds randomness during the encryption. Basically, due to this randomness, the ciphertext is different after each encryption of the same plaintext. This is especially needed in asymmetric cryptography, since the encryption key is public and anybody can encrypt any message. With asymmetric cryptography, randomisation prevents linking ciphertexts if they contain the same plaintext, as encrypting the same plaintext twice results in two different ciphertexts. But, even in symmetric cryptography, randomisation can be useful to prevent the detection of duplicate messages.

To illustrate how asymmetric encryption and decryption works in practice, we will explain the ElGamal encryption scheme, since it is a relatively easy asymmetric encryption scheme to explain. ElGamal is an encryption scheme that uses randomisation. The generation of public keys in ElGamal [12, 32] is very similar to the Diffie-Hellman key exchange. However, Diffie-Hellman key exchange is used to establish a common session key. ElGamal, on the other hand, encrypts messages using a different random key for each message. ElGamal consists of the following algorithms [28]:

KeyGen: The user needs to generate a keypair before he can receive encrypted messages. First, we generate a prime number p and use the multiplicative cyclic group $\mathbb{Z}_p$ with a generator g. Note that ElGamal can be generalized to work in any multiplicative cyclic group, but we consider $\mathbb{Z}_p$ only. Then, we generate a random secret key a such that $1 \leq a \leq p-2$. We compute

$$A = g^a \bmod p$$

The algorithm outputs the public key $\langle p, g, A \rangle$

Encrypt: If we want to encrypt a message m to a user, we first have to obtain the (authentic) public key of that user: $\langle p, g, A \rangle$. First, we have to represent the message as an integer in $\{0, 1, \ldots, p-1\}$. As ElGamal is non-deterministic, we first have to generate a random r such that $1 \leq r \leq p-2$. Then, we compute the ciphertext in two parts:

$$c_1 = g^r \bmod p$$

$$c_2 = B^r m \bmod p$$

The output is $\langle c_1, c_2 \rangle$.

Decrypt: The recipient can decrypt the ciphertext using his private key a by calculating $\frac{c_2}{c_1^a} \bmod p$, since

$$\frac{c_2}{c_1^a} \bmod p = \frac{A^r m}{(g^r)^a} \bmod p = \frac{(g^a)^r m}{(g^r)^a} \bmod p = \frac{g^{ar} m}{g^{ar}} \bmod p = m$$

We note that ElGamal, just like the Diffie-Hellman key exchange, is vulnerable to an active attacker. Suppose Monica wants to send a confidential message to Ross and she uses ElGamal to encrypt that message. An attacker, Judy, wants to read the message. Judy, being an active attacker as shown in Figure 4.1b, is located between Monica and Ross. If Monica sends her public key $\langle p, g, A = g^a \bmod p \rangle$ to Ross, Judy replaces the public key with her own public key $\langle p, g, J = g^j \bmod p \rangle$ and sends it to Ross instead of the public key of Monica. Now, Ross receives the public key from what he thinks is Monica, but he does not know that the public key is actually Judy's. Ross will encrypts his message to Monica by using $\langle p, g, J \rangle$ as public key and Judy can decrypt the message. For Judy to go unnoticed, she probably has to encrypt the message again and send it to Monica. This is a so-called *man-in-the-middle* attack. It can be avoided by authenticating the public keys using digital signatures.

Having discussed both symmetric and a-symmetric cryptography, we will now explain several algorithms which are relevant to protecting patient information and discuss whether we can use them. But before we continue doing that, we first describe an encryption scheme that is different from all others.

4.2.4 One-time-pad

You may wonder whether there exists an cryptographic algorithm that can create a ciphertext so that nobody can ever obtain the plaintext without knowledge of the key. An unbreakable algorithm, even with infinite computing power and an unlimited amount of time, exists and is called the one-time-pad [32]. A one-time-pad is a scheme with a key that is truly random and each character of the key is used only once to encrypt one character of a plaintext. The encryption of one character can be performed by adding the letters by their numerical value. The letter 'a' has value '0', 'b' has value '1', etcetera. Finally, 'z' has value '25'. All calculations are performed modulo 26, so `'z' + 'e' = 25 + 4 = 29 = 3 (mod 26) = 'd'`.

Suppose we want to encrypt the message "thesis". We generate a truly random key "pgyabz" and give this key to the recipient. We generate the resulting ciphertext as follows.

```
t  h  e  s  i  s  (plaintext)
p  g  y  a  b  z  (key)
19 7  4  18 8  18 (plaintext as numeric)
15 6  24 0  1  25 (key as numeric)
------------------------------------------- +
34 13 28 18 9  43 (result, plaintext+key)
8  13 2  18 9  17 (result mod 26)
i  n  c  s  j  r  (ciphertext)
```

Decryption is exactly the other way around, but instead of addition we use substraction.

```
i  n  c  s  j  r  (ciphertext)
p  g  y  a  b  z  (key)
8  13 2  18 9  17 (ciphertext as numeric)
15 6  24 0  1  25 (key as numeric)
------------------------------------------- -
19 7  4  18 8  18 (resulting plaintext mod 26)
t  h  e  s  i  s  (plaintext)
```

Since they key is as long as the plaintext, no information from the ciphertext can be deducted without knowing the key. By using a different key, it is possible to obtain any plaintext of the same length, as shown below.

```
i  n  c  s  j  r  (ciphertext)
b  j  r  g  f  y  (alternative key)
8  13 2  18 9  17 (ciphertext as numeric)
1  9  17 6  5  24 (key as numeric)
------------------------------------------- -
7  4  11 12 4  19 (resulting plaintext mod 26)
h  e  l  m  e  t  (plaintext)
```

There is no way to prove which key is the correct one, which is one of the advantages of a one-time-pad. Note that our example includes only the characters a to z. In practice, any set of characters can be chosen.

Shannon [34] introduced the notion of what he called *perfect secrecy*, which means that even an adversary with infinite computing power is unable to learn anything about the plaintext from the ciphertext, except maybe the length. Shannon proved that the one-time-pad is a perfect encryption scheme.

In practice, a one-time-pad does not work out. It is impractical due to the large keys, as large as the plaintext, and it suffers from the same key distribution problem as symmetric cryptography of how to establish a shared secret. Hence we need other algorithms to protect patient information, which we will discuss now.

4.2.5 Identity-Based Encryption

Identity-Based Encryption (IBE) is a form of asymmetric encryption where a users' public key is his identity, such as an email address, physical address or (unique) full name. The main advantage of IBE compared to regular asymmetric cryptography, is that, in IBE, there is no need to retrieve a users' public key. Suppose that, for example, Monica wants to send a secret message to Ross. With a regular asymmetric cryptosystem, Monica has to find Ross' public key first. She can either ask Ross to in person, in which case she could as well tell him the message right away, or, for example, download Ross' public key from his own website or from a keyserver. When an Identity-Based cryptosystem is being used, Monica can simply use Ross' identity as public key. For example, in the case the identity is based upon an e-mail address, Monica uses `ross@example.com` as public key.

Shamir [33] introduced the concept of IBE, but only gave an implementation proposal for identity-based signatures, thus it does not work for encryption. However Boneh and Frankliny [8] introduced a scheme that can be used for encrypting. They claim that their IBE scheme has a performance equal to ElGamal encryption. Furthermore, their scheme is secure against chosen ciphertext attacks. It consists of several algorithms:

Setup: during setup, we use a security parameter $k \in \mathbb{Z}^+$ to generate a prime number q, two groups named $\mathbb{G}_1$ and $\mathbb{G}_2$ of order q, a bilinear map $\hat{e} : \mathbb{G}_1 \times \mathbb{G}_1 \to \mathbb{G}_2$ and a random P, which should be a generator of $\mathbb{G}_1$. Then, we generate a random master-key $s \in \mathbb{Z}_q^*$, which is kept secret. We compute $P_{pub} = sP$. We define n to be the maximum number of bits we can encrypt. This may be regarded as a limitation to the algorithm, but it can be circumvented. If one uses this system to encrypt large messages, it is better in terms of speed to encrypt the message using a symmetric encryption scheme with a random session key. Then, the session key is encrypted using the IBE scheme. This way, n can be defined to be the maximum length of the session keys, which are typically shorter than the keys of asymmetric encryption. We discuss this possibility in our description of the `Encrypt` algorithm below. As final step in the setup phase, we choose four hash functions: $H_1 : \{0,1\}^* \to \mathbb{G}_1^*$, $H_2 : \mathbb{G}_2 \to \{0,1\}^n$, $H_3 : \{0,1\}^n \times \{0,1\}^n \to \mathbb{Z}_q^*$, $H_4 : \{0,1\}^n \to \{0,1\}^n$.

Obviously, the setup algorithm is executed only once, to set up the entire system. The result of this algorithm consists of all variables and functions we defined, except for the master-key s, thus:

$$params = \langle q, \mathbb{G}_1, \mathbb{G}_2, \hat{e}, n, P, P_{pub}, H_1, H_2, H_3, H_4 \rangle$$

The system parameters *params* are the input of all other algorithms and can be made public.

Extract: users need to obtain the private (decryption) key of their identity. They file a request to the keyserver, which has to validate whether the user should have the private key to the requested identity. If the request is valid, the keyserver uses this algorithm to extract the private key for the input identity ID. The identity ID is defined as $\{0,1\}^*$. First we compute the identity as element of $\mathbb{G}_1^*$ by using H_1 as $Q_{\mathrm{ID}} = H_1(\mathrm{ID})$. It is required to convert bits to a group to be able to perform calculations on it. Then we calculate the private key $d_{\mathrm{ID}} \in \mathbb{G}_1^*$ for this identity: $d_{\mathrm{ID}} = sQ_{\mathrm{ID}}$. Note that only the keyserver can perform this operation, as the master-key s is required. The keyserver can give d_{ID} to the user, who is then able to use the decryption algorithm to decrypt messages sent to him.

Encrypt: there are two parameters: a message $M \in \{0,1\}^n$ and an identity ID to which the message has to be encrypted. Just like with the extract algorithm, we have to compute the identity as an element in $\mathbb{G}_1^*$, thus we compute $Q_{\mathrm{ID}} = H_1(\mathrm{ID})$.

We choose a random $\sigma = \{0,1\}^n$, thereby making the encryption non-deterministic. Then, we compute $r = H_3(\sigma, M)$ with $r \in \mathbb{Z}_q^*$ and use r to compute

$$c_1 = rP$$

Then, we compute

$$c_2 = \sigma \oplus H_2(\hat{e}(Q_{\text{ID}}, P_{pub})^r)$$

The last component of our ciphertext is c_3, where the encrypted plaintext is stored, we compute c_3 as

$$c_3 = M \oplus H_4(\sigma)$$

The resulting ciphertext C is $\langle c_1, c_2, c_3\rangle$, where $c_1 \in \mathbb{G}_1^*$, $c_2 \in \{0,1\}^n$ and $c_3 \in \{0,1\}^n$.

As discussed before, it is possible to encrypt a message with a length that is bigger than n. To do so, we need a symmetric encryption scheme E to encrypt the message M. In that case, c_3 is computed as $c_3 = E_{H_4(\sigma)}(M)$, thereby encrypting M using the hash of σ as key to the encryption algorithm.

Decrypt: the input is the ciphertext $C = \langle c_1, c_2, c_3\rangle$, which is encrypted for a user with identity ID. To decrypt the ciphertext, we first compute $\sigma = c_2 \oplus H_2(\hat{e}(d_{\text{ID}}, c_1))$. Then, we compute the plaintext by computing $M = W \oplus H_4(\sigma)$. Finally, we have to verify whether the plaintext M is actually correct. We can do that by verifying the that $c_1 = rP$ with $r = H_3(\sigma, M)$.

If a message larger than n is encrypted, e.g. a separate symmetric encryption scheme is used in the encryption algorithm, we can obtain M by computing $M = D_{H_4(\sigma)}(c_3)$, where D is the decryption algorithm and $H_4(\sigma)$ the key.

Identity-based encryption could be used to secure patient information. The question remains what identity to use to encrypt the patient information. It seems reasonable to use some kind of patient identifier such as a social security number, or an internal hospital identification number, as identity of the patient. There is a problem though: we can grant somebody, for example a doctor, access to the patient information of a certain patient by giving the private key of the identity, but anybody with the private key can decrypt anything that was encrypted with the identity. This means it is not possible to grant access to only the full name and date of birth of a patient, without also giving access to, for example, the diagnosis and any other patient information. Instead, we would like to be able to have fine-grained access control, so that we can grant access to any chosen subset of patient information. Furthermore, it is not possible to revoke access, which we consider to be a deal-breaking limitation.

Boneh et al. [8] introduce solutions to the above problems. One option they call "managing user credentials" can be used to encrypt messages using more flexible access control. Normally, one would encrypt a message by using the identity, such as an email address, as public key. So Monica would encrypt a message to Ross using `ross@example.com` as identity. But Monica could also use the identity `ross@example.com || 2011`. Now, Ross has to obtain the private key to `ross@example.com || 2011`. This effectively enables expiration of the keys, as it forces Ross to obtain a new private key every year. Obviously, we could also encrypt a message by appending the year and month to Ross' email address. Alternatively, we could append certain other attributes, such as `patient129472 || dateofbirth`, to encrypt the date of birth for patient 129472 in a database. Anybody who wants to obtain access to the date of birth of this patient, has to request the private key from the keyserver. In that way, we use the keyserver to enforce access control, apart from any access control at the database itself. The attributes we can specify are rather simple. It is rather complex to specify an access policy like `patient12536 || requester=doctor || 2011` to enforce the person who requests access to be a doctor and to have the 2011-key of a particular patient. To do this, we would have to create some kind of protocol to which everybody has to conform, so that the keyserver can parse the identity.

We consider the suggestion of Boneh et al. that revoking public keys can be done by appending the date to the identity, to be a non-practical way of revocation. Instant revocation is not possible, since the private key of the identity will remain valid for all future encryptions using that same identity. Suppose some piece of very sensitive data is encrypted with the identity `secret || January 2011`. Now, a certain user Gavin has the private key to access this data. We want to revoke the access rights of Gavin to decrypt the data. For new ciphertexts, this is not a problem, since the keyserver can stop issuing the keys to Gavin, thereby preventing Gavin to decrypt ciphertexts that are encrypted later, for example to `secret || February 2011`. However, Gavin can still decrypt data with his private key (which is valid for January). Any existing ciphertext that Gavin can decrypt, has to be deleted in order to perform the revocation of Gavin's key. As we consider a ciphertext to be public, this is not possible. Therefore, revocation is not possible for IBE, as there is no way to invalidate a private key.

Eventually, the introduction of adding attributes to an identity in order to specify an access policy led to the introduction of a new type of encryption schemes, which we will discuss now.

4.2.6 Attribute-Based Encryption

Attribute-Based Encryption (ABE) was introduced by Sahai and Waters in [31] as a new type of IBE. Sahai and Waters wanted to support IBE by using biometric input as the identity. However, due to the nature of biometrics, some kind of fault tolerance is required in order to allow biometrics to be used with IBE. Regular IBE does not support fault tolerance, hence Sahai and Waters came up with Fuzzy Identity Based Encryption (FIBE) to support a certain error d. A second application of FIBE, apart from the biometric usage scenario, is ABE. This is useful when the sender of a message does not know the public key of the recipient, but does know a set of attributes of the recipient. Suppose, for example, that Janice wants to send an encrypted message to all teaching assistants of the course "Cryptography", that she teaches. Janice would encrypt the message under the attributes `teaching_assistant` and `cryptography`. Any person that has a private key corresponding to both attributes can then decrypt the ciphertext. This solution has a big advantage over IBE, since Janice does not need to know the public keys of the recipients. Furthermore, she has to encrypt and publish the ciphertext only once. All teaching assistants of the Cryptography course can then decrypt the ciphertext. In opposition, if Janice had used IBE, she would have had to encrypt the message for each recipient individually.

In the ABE scheme that was introduced in [31], there is one trusted party: the key generation center (KGC). This KGC sets up the system by defining the public parameters, generating a master-key and giving private keys to users. The fault tolerance d is defined in terms of sets. A user with the private key ω' can decrypt a ciphertext, which is encrypted with identity ω, only if the overlap of ω and ω' is greater than the fault tolerance, e.g. $|\omega \cap \omega'| \geq d$. To encrypt a message, the user first picks a set of attributes as public key and then encrypts the message. The public key is included as part of the ciphertext. Decrypting a ciphertext, which is encrypted under the identity ω, can be done by selecting a subset of d elements from $\omega \cap \omega'$ to be used as decryption key. A user can obtain a private key for a set of attributes by contacting the KGC. The KGC is responsible for verifying that this set of attributes is valid, e.g. one of the tasks of the KGC is to enforce access control on the attributes. The KGC itself is able to decrypt any message, as it is able to generate a private key valid for all attributes. There are two variants of ABE, namely Key-Policy ABE (KP-ABE) and Ciphertext-Policy ABE (CP-ABE).

In KP-ABE [20], the private key of a user contains the access structure for that user. The ciphertext is associated with a set of attributes ω. The recipient possesses the private key belonging to a certain access structure $\mathbb{A}$ and can only decrypt a ciphertext c if the set of attributes ω belonging to c satisfy $\mathbb{A}$. In the example of Janice, the ciphertext ω would be $\{teaching\_assistant, cryptography\}$ and a teaching assistant's private key's access structure could be

$$\mathbb{A} = (teaching\_assistant \ \wedge \ (cryptographI \ \vee \ discrete\_mathematics)$$

In KP-ABE, as the access structure is set in the private key, a message's sender cannot express the access policy of the ciphertext other than with some descriptive attributes [6]. This seriously limits the ability of the sender to define the access policy (sometimes called privacy) of the ciphertext. For example, a sender could define ω to be $\{teacher, cs, 2010\}$ in his intention to allow the teacher of the course computer science, given in 2010, to decrypt the ciphertext. However, this level of expressiveness is not possible with KP-ABE, as there exists an access policy $\mathbb{A}_2 = teacher \vee cs \vee 2010$ that allows decryption of the ciphertext. Obviously, one may consider this a problem of the trusted party, but the lack of control over the ciphertext by the sender is a major drawback in KP-ABE.

Ciphertext-Policy ABE was first introduced in [6]. In CP-ABE, the ciphertext contains the access structure $\mathbb{A}$ needed to decrypt the ciphertext. Basically this is the opposite of KP-ABE, as it switches the location of the access structure and the attributes. Each user in the system possesses a private key which contains a set of attributes ω that describe the particular user. Suppose, for example, that Sam has a private key with $\omega = (student, crsMathematics2010, crsSpanish2009)$, meaning Sam is a student who attended the 2009 Spanish course and the 2010 mathematics course. A teacher Spanish, called Fenny, wants to send an encrypted message to all students who attended the Spanish course in 2009. Fenny could encrypt her message under the access structure $\mathbb{A} = (student \ \wedge \ crsSpanish2009)$. Then Fenny sends the resulting ciphertext and $\mathbb{A}$ to the recipients. Decryption of the ciphertext is only possible when the attributes of the decrypting user match the access structure. CP-ABE allows the sender of a message to set the access structure for that message, resulting in flexibility in choosing who can decrypt the ciphertext. On the other hand, having a trusted party with control over the access control, such as with KP-ABE, allows for more consistent access control.

In [25], Li et al. proposed to use ABE to secure Patient Health Records (PHR) in cloud computing. Even though their solution uses PHR, where patients manage their own health records and decide who to grant access to, it is still interesting to discuss the application of ABE to ensure the security of patient information. The security model is a honest but curious database server. This means the database server, used for storing the data, tries to figure out the contents of the data stored, but will follow the protocol in general. This is especially important when dealing with update or delete requests that are required to achieve proper security. Li et al. introduce the concept of write keys which allows data owners to specify who exactly has permission to write data to the PHR. While this fits their security model, as it requires the database server to follow the protocol and check for write permissions, this does put a lot of trust in the database server. Another feature that is discussed is the break-glass method. The idea is that there are emergency situations where access to the PHR of somebody is immediately needed. Therefore, temporary access keys are kept apart to allow decryption is case of emergency. While we consider this a serious limitation to the security of the system, it is not relevant to our topology as we are not working with PHRs (where the patient controls access), but with an EMR (where an authority in the hospital, the keyserver, manages access). Finally, user revocation is carried out by removing the access key of the user from the keyserver and by updating (re-encrypting) all ciphertexts which the user had access to, for example by adding or removing an attribute. Furthermore, all keys of the other users have to be updated as well so that they are able to decrypt the new ciphertexts. Clearly, revoking consists of a lot of computational overhead and requires the database to honestly delete all old ciphertexts. If the database does not delete the ciphertexts, the user revocation is not possible.

While ABE offers more flexibility with specifying attributes during encryption, which could be really useful in our case, revocation in ABE requires updating many ciphertexts and keys. In our topology, we do not want to re-encrypt ciphertexts and issue new keys to the users who are (still) authorized, when the access rights of one user need to be revoked, as it are a very expensive operations. Therefore, we have to consider an alternative solution.

4.2.7 Type-Based Proxy Re-Encryption

Tang [36] introduced Type-Based Proxy Re-Encryption (TBPRE) as a method to enforce access policies on encrypted data. The concept of Proxy Re-Encryption (PRE) uses a proxy to transform a ciphertext encrypted with one key, to a ciphertext encrypted with another key, without performing decryption and thus without having any knowledge of the plaintext. The scheme of PRE consists of three entities: the delegator, the delegatee and the proxy. The delegator is the user who wants the proxy to re-encrypt ciphertexts so that they can be decrypted by the delegatee. The proxy can only perform the re-encryption operation if it has a so called Re-Encryption Key (REK), which can be generated by the delegator only. Figure 4.2 gives an overview of PRE.

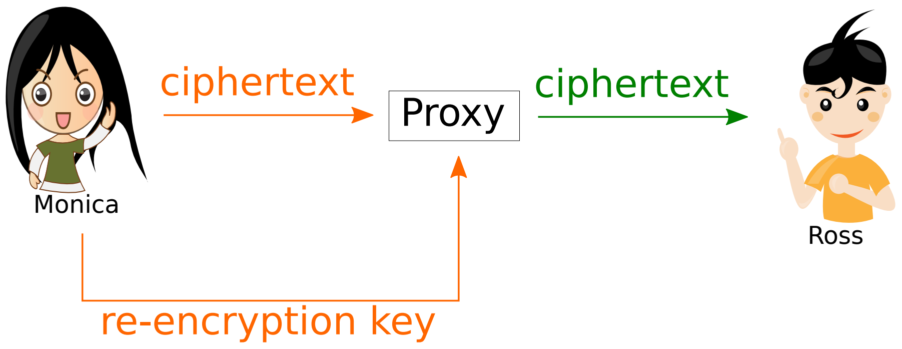


Figure 4.2: The proxy uses the re-encryption key to re-encrypt a ciphertext from the delegator (Monica) to the delegatee (Ross).

TBPRE extends PRE to use message types, which is basically an attribute, with the exception that a message must have exactly one type. A type can be an arbitrary number of bits $\{0,1\}^*$. When a message is encrypted with type t, the proxy can only re-encrypt the message if it possesses the re-encryption key with type t. The delegator can control which messages the proxy should be able to decrypt by only giving the proxy re-encryption keys to specific types. A re-encryption key depends on the delegator's public key, the delegatee's public key and the message type, and can only be generated using the delegator's private key. We use the notation $rk_{pk \xrightarrow{t} pk'}$ to show a re-encryption key from the delegator's public key pk, the delegatee's public key pk' and the message type t.

We continue with an example of the usage of TBPRE. Suppose there are a delegator Monica and a delegatee Ross, as shown in Figure 4.3. First of all, Monica and Ross each generate a keypair. Monica receives the answers to homework assignments from a friend of hers, who encrypts the answers under the type "homework". Monica wants to make the answers available to Ross and sets up automatic forwarding of all emails with "homework" in their title to Ross. Then, Monica generates a re-encryption key $rk_{pk_{\text{monica}} \xrightarrow{\text{homework}} pk_{\text{ross}}}$, which she gives to the proxy (1).

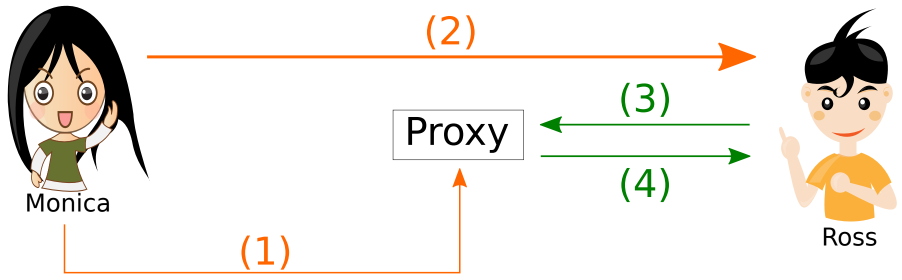


Figure 4.3: An alternative look at proxy re-encryption. The delegator (Monica) sends the re-encryption key to the proxy (1). Then, Monica sends a ciphertext to Ross (2), who can ask (3) the proxy to re-encrypt the ciphertext into a ciphertext that Ross can open using his private key (4).

Now, Ross will get encrypted emails from Monica (2) containing the answers to the homework assignments. However, the emails are encrypted for Monica, so Ross first has to provide the Proxy with the encrypted emails (3). The proxy will re-encrypt the emails using the re-encryption key that Monica provided, and send the result back to Ross. Now, Ross can decrypt the email and open the answers to the homework assignments.

Before discussing the advantages of TBPRE compared to Attribute-Based Encryption, we first describe TBPRE in more detail. The scheme consists of the algorithms listen below, as defined in [36].

Setup: this sets up the entire system. We generate a prime p, three multiplicative cyclic groups $\mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T$ of order p. Then we pick random generators $g_1 \in \mathbb{G}_1$ and $g_2 \in \mathbb{G}_2$ and a bilinear map $\hat{e} : \mathbb{G}_1 \times \mathbb{G}_2 \rightarrow \mathbb{G}_T$. We define l to be the maximum length of plaintexts we support. Thus, if the system can encrypt messages with length greater than l, we need to use a symmetric encryption scheme to encrypt the message, and then encrypt the symmetric key using TBPRE. In this case, l has to be sufficient to fit the entire symmetric encryption key. The last primitives are two hashing functions:

$$H_1 : \{0,1\}^* \rightarrow \mathbb{G}_2$$

$$H_2 : \{0,1\}^* \rightarrow \{0,1\}^l$$

Here, H_1 is used to map a type to an element in a group and H_2 is used to map a random string to a string suitable for usage as a key.

The public parameters are $\langle \mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, p, g_1, g_2, H_1, H_2, \hat{e}, l \rangle$

KeyGen$_1$: the keyserver will use this algorithm to generate a keypair for the delegator. It computes a private key sk randomly from $\mathbb{Z}_p$. The corresponding public key is computed as $pk = g_1^{sk}$.

KeyGen$_2$: this algorithm is used by the keyserver to generate a keypair for the delegatee. It differs slightly from KeyGen$_1$ in that the public key is computed using g_2 instead of g_1. Just as with KeyGen$_1$ the private key sk is chosen randomly from $\mathbb{Z}_p$. The public key is computed as $pk = g_2^{sk}$.

Encrypt$_1$: this algorithm can be used to encrypt a message given with a type to a user. The parameters are a message m, a type t of the message and the public key pk of the recipient.

We randomly choose $r \in \mathbb{Z}_p$ and $h \in \mathbb{G}_T$. We then compute the ciphertext in three parts:

$$c_1 = g_1^r$$
$$c_2 = m \oplus H_2(h)$$
$$c_3 = h \cdot \hat{e}(pk, H_1(0||t))^r$$

The ciphertext $C = \langle c_1, c_2, c_3 \rangle$.

Once a message is encrypted, an adversary will still be able to deduce the length of the message by looking at the length of c_2.

Encrypt$_2$: this algorithm differs from Encrypt$_1$ in that it does not use a message type. It can be used to encrypt a message which can then be decrypted by the corresponding keypair generated by KeyGen$_2$, e.g. when one wants to encrypt a message a delegatee as destination. The parameters are a message m and the public key pk of the recipient.

We randomly choose x and y from $\mathbb{Z}_p$ and w randomly from $\mathbb{G}_T$. We then compute the ciphertext in four parts:

$$c_0 = g_1^x$$
$$c_1 = pk^y = g_2^{v \cdot y}$$
$$c_2 = m \oplus H_2(w)$$
$$c_3 = w \cdot \hat{e}(g_1^x, g_2^y)$$

The ciphertext $C = \langle c_0, c_1, c_2, c_3 \rangle$.

Just as with Encrypt$_1$, an adversary is able to deduce the length of the plaintext.

Decrypt$_1$: this algorithm is used by the delegator to decrypt a ciphertext when the ciphertext is encrypted under a certain message type. This is the regular decryption algorithm, e.g. when no delegation is done. The parameters are the ciphertext $\langle c_1, c_2, c_3 \rangle$, the message type t and the private key sk. The delegator can decrypt the ciphertext to obtain the message m by computing:

$$m = c_2 \oplus H_2(\frac{c_3}{\hat{e}(c_1, H_1(0||t))^{sk}})$$

Decrypt$_2$: this algorithm can be used to decrypt ciphertext without a message type, such as ciphertexts computed by Encrypt$_2$ or re-encrypted ciphertexts computed by running Preenc (see below). The parameters are the ciphertext $\langle c_0, c_1, c_2, c_3 \rangle$ and the private key sk. The algorithm obtains the message m by computing:

$$m = c_2 \oplus H_2(\frac{c_3}{\hat{e}(c_0, c_1)^{sk^{-1}}})$$

Here, sk^{-1} is the inverse of sk.

Pextract: this algorithm generates (extracts) a re-encryption key that can be used by a proxy to re-encrypt ciphertexts. The algorithm is ran by the delegator and needs the public and secret key of the delegator himself (pk_A and sk_A), the public key pk_B of the delegatee and the message type t. It then generates a re-encryption key that only works for ciphertexts encrypted with pk_A as public key that have type t, and re-encrypts the ciphertext to pk_B. The re-encryption key is computed as follows.

First we pick a random s from $\mathbb{Z}_p$. Then, we compute:

$$rk_{pk_A \xrightarrow{t} pk_B} = \langle r_1, r_2 \rangle = \langle pk_B^s, g_2^s \cdot H_1(0||t)^{-sk_A} \rangle$$

Preenc: the proxy runs this algorithm to re-encrypt a ciphertext. The parameters are a ciphertext $c = \langle c_1, c_2, c_3 \rangle$ encrypted using a public key pk_A and with type t, and the re-encryption key $rk_{pk_A \xrightarrow{t} pk_B} = \langle r_1, r_2 \rangle$. We then compute the ciphertext in four parts as $c' = \langle c'_0, c'_1, c'_2, c'_3 \rangle$. First, we generate a random z from $\mathbb{Z}_p$ to randomize the new ciphertext. Then, we compute:

$$c'_0 = c_1$$

$$c'_1 = r_1 \cdot pk_B^z$$

$$c'_2 = c_2$$

$$c'_3 = c_3 \cdot \hat{e}(c'_0, r_2 \cdot g_2^z)$$

The result $\langle c_1, c_2, c_3 \rangle$ can be decrypted by the delegatee which has the private key corresponding to the public key pk_B using the algorithm Decrypt$_2$.

The scheme we listed above provides ciphertext privacy [36], which means that an adversary is unable to detect the difference between a ciphertext sent to the delegatee (using `Encrypt`$_2$) and a re-encrypted ciphertext (the result of `Preenc`). This means that even if an adversary is able to intercept the re-encrypted ciphertext, he is unable to see whether it is a ciphertext or a re-encrypted ciphertext, originally meant for somebody else. With ciphertext privacy, the delegator remains anonymous.

TBPRE uses an approach different from Attributed-Based Encryption (ABE) by the use of a proxy to re-encrypt. When using TBPRE, the database only stores ciphertexts and does not have any task in user revocation. TBPRE assumes the proxy to be curious but honest: it will perform a proper re-encryption, but will try to find information about the plaintexts. If user revocation is required, we require the proxy to delete re-encryption keys on request. The database only stores the ciphertexts, so if a delegator wants to revoke access to his data, he needs to ask the proxy to delete one or more re-encryption keys. Hence, revocation can be performed by removing the re-encryption keys from the proxy (or proxies) only, which is more efficient than ABE. Furthermore, by the use of a message type, TBPRE provides flexibility to specify which ciphertexts one should be able to decrypt, although it does not have the specificity of ABE in that matter.

Having discussed encryption schemes to use for providing confidentiality, we now have to provide integrity protection to the data, which we discuss in the following section.

4.3 Integrity

When a database server is untrusted, the integrity of the stored data is very important. Both confidential data, which is encrypted, and non-confidential data have to be integrity protected so that the database, or an unauthorized user, cannot modify data without being noticed. Especially the latter is important: we can not give any guarantee that the data stored in the database can only be edited by authorized users. Instead, we offer a method to detect unauthorized changes. Note that our definition of integrity, namely the ability to detect unauthorized changes, differs from the definition traditionally used within the database scene, where integrity means the consistency of the data within the database itself, e.g. the ability to detect changes due to storage failure.

4.3.1 Threats to Integrity

There are several threats to the integrity of the data after it has been stored in the database. But even before, data can be altered. When a doctor creates a health record with patient information on his own computer, a virus may change the contents before it is being saved on the harddisk. However, the security of a workstation is beyond the scope of this thesis. We consider the workstation of a doctor to be secure in the sense that all software is honest and does not steal or modify data without the intention of the doctor.

The first threat is when the data is in-transit after a doctor sends patient information to the database server. This is especially the case with asymmetric encryption, where an adversary can encrypt data and replace the doctor's encrypted data. The solution is to either integrity-protect the data itself (before encryption), or use a secured connection between the doctor and the database to deal with it, trusting the database to not allow unauthorized alterations.

Our goal is to put as little trust in the database server as possible. Therefore, we have to consider a database that attempts to alter the data. This could be the data itself or, in case of a relational database, the relation between the different data items. It could be that the database does not store data at all, or selectively stores the data. And when a request to update or delete the data is sent to the database, it may not, or only partially, be processed. Detecting a database server that does not, or only partially, process requests, is very complex and currently we are unaware of a good solution to be able to verify that the database stores the most recent version of the data, without having access to the data locally. Therefore, we consider the database to be "honest" in the sense that it will store the data. At any point in time, we should be able to verify integrity of the data in the database.

Even if the database is completely honest, there may still be other adversaries with access to the database who try to alter the data. One of such adversaries can be the database administrator, as mentioned in Section 3.3.3. But as we already discussed the case where the database tries to alter data, which basically covers the integrity of the entire database, a rogue administrator does not introduce new issues and is simply considered to be part of the database.

We now need to actually find a way to detect unauthorized changes to the data.

4.3.2 Protecting Integrity of Data

We will discuss data integrity by using a small hospital administration database as example. In this database, there are four tables: `Patients`, `Doctors`, `Diagnoses` and `Link`, as shown in Figure 4.4. The tables `Patients` and `Doctors` contain respectively the data of patients or doctors. The table `Diagnoses` contains all possible diagnoses of the patients. Finally, `Link` links the tables together, defining the correlation between a patient, a doctor and a diagnosis. We want to be able to store these tables in an untrusted database and to protect the data against unauthorized modifications.

At a first look, it would make sense to represent a row as a tuple of values, to preserve the order of the fields, and generate a signature over this tuple, so that changes can be detected. This approach has several disadvantages. First of all, if fields are encrypted and decryption rights are given per field, a user needs to decrypt all fields in order to be able to verify the integrity of the entire row. If the user does not have the decryption key for one of these fields, he is unable to verify the integrity. Another disadvantage is that if each row has one signature, it is impossible to verify only one field. Instead, a user has to retrieve all fields of that particular row and verify the entire row. For these reasons, we create a signature per field in the row. This, however, introduces several issues, such as the position of a field in the table. We now discuss our solutions to these issues.

Patients		
id	**name**	**gender**
0	Cully Barnaby	Female
1	Gavin Troy	Male
2	Deirdre McKenzy	Female

Doctors	
id	**name**
0	dr. Bush
1	dr. Friedlich
2	dr. Jansen

Diagnoses	
id	**diagnosis**
0	Healthy
1	Epilepsy
2	Brain dead

Link			
id	**patient**	**doctor**	**diagnosis**
0	0	2	0
1	2	0	1

Figure 4.4: Regular tables representing a hospital administration system, filled with example data.

Hashing or Signing

Protecting each row in the table is required in order to avoid unauthorized changes. To protect the column `name` in the patients table, we could create an extra column to the table named `name_hash`. For each `name`, we generate a `name_hash` that stores the hash of the name, using a secure hash function such as SHA256. However, an unauthorized user Eve who has access to the database could update any row, change the `name`, calculate the hash of the changed `name` and update the `name_hash` as well. This means the hash should be signed in such a way that Eve cannot create a valid signature on a new hash. One of the methods to create such a signature, is by using a message authentication code (MAC) [28]. A MAC needs a symmetric-key as input for both the creation and verification. Without knowledge of this key, one cannot create or verify the MAC, thereby preventing a change by an adversary. However, anybody who knows the key can generate a new MAC. Since the key is required to be able to verify, anybody who should be able to verify the MAC, can create a new one as well. Therefore a MAC is not feasible for us. An alternative to a MAC is a digital signature with asymmetric keys. The hash will be signed with a private key and can be verified by anybody, by using the public key. If we sign the hash of `name` and put the resulting signature in the `name_sig` column, the field `name` cannot be changed by anyone that does not possess the private signing key without going noticed. Throughout this section, we use one signing key for all signatures.

Bind to Row

There are still issues however. At this moment, we sign a hash of a certain field, such as a name. So we can verify whether the field is still correct. However, an adversary with access to the database can “copy” fields if he also copies the corresponding hash. Currently, we can only verify that a field contains the correct value by checking the hash, but we cannot verify that the position of the field in the table. To avoid this, we need to include the primary key `id` in the hash. This effectively ties the name column to the row. Now, it is no longer possible to take the value of a field and its hash and put it at another row, as the hash incorporates the primary key of the row and thus the hash will not verify.

Bind to Column

An adversary could still alter the database using another attack. Even though it is not possible to change the row of a value, it is still possible to change the column. Namely, we sign a hash on the primary key and the value itself. If two columns have the same type (e.g. a number), then they can be swapped by an adversary. If the hashes are also swapped, then there is no way to detect this. This can be avoided by including the name of the column in the hash as well.

Bind to Table

Signatures could, potentially, be swapped across different tables, so we also include the name of the table in the hash. Potentially, problems could arise on the database level as well (swapping signatures across databases, where the name of the table, column and the value of the primary key are equal). However, using only one signing key per database avoids this issue. Instead of including the table name in the hash, we can also use one signing key per table.

Delimiter

Finally, we need to use a delimiter between all the inputs of (the hash of) the signature. Otherwise, we may be unable to distinguish the concatenation of the inputs '1' and '23', which becomes '123', and '12' and '3', which becomes '123' as well. To avoid this, we use a delimiter that is different from the characters used as input. For example, we could use '||' as delimiter for numeric fields. The concatenation of '1' and '23' becomes '1||23'.

Alternatively it is possible to hash each individual input (namely the tablename, columnname, primary key and value), concatenate these hashes and hash the result. Hashes can be concatenated if they have a fixed length (which is the case for SHA256, which we use) or if they map to a set of characters that does not include the delimiter.

Input to the hashing function

Having discussed all issues with signing, we now summarize what inputs are required to the signing function in order to properly secure a field in the database.

Define inputs: The inputs are, first of all, the field itself. Also the primary key of the field's row, the columnname of the field's column and the name of the table are inputs.

Concatenate: Then, we concatenate all inputs using a delimiter '||'. For n inputs, the concatenated result is: 'input$_1$ || input$_2$ || ... || input$_n$'.

Hash: After the inputs are concatenated into one string of characters, we hash the inputs using a hashing function, which maps any input to a fixed-length output k: $hash : \{0,1\}^* \to \{0,1\}^k$.

Sign: Finally, we sign the hash of concatenated inputs using a signature scheme.

More formally, if we want to sign a field with value m in table T, where the column name is C and the primary key of the row is id, we can obtain the signature s:

$$s = sign_k(hash(T \text{ `||' } C \text{ `||' } id \text{ `||' } m))$$

Here, k refers to the private signing key and '||' is the delimiter. By signing the contents of the tables from Figure 4.4 as described above, we provide integrity protection. Figure 4.5 shows the integrity-protected tables.

We have to address one special case of integrity protection. If the data to be signed is confidential, we have to make the signature confidential as well. The signature can be made confidential by encrypting it, just like the confidential data is encrypted. If the data is confidential and the signature is public, an adversary could obtain the data without decrypting it by finding data for which the signature verifies. We discuss this situation in more detail in Section 5.1.2. The signature is encrypted using the tag of the column it is the signature of. For example, a signature of `name` is put in `name_sig`, and encrypted using the tag `name`. In this way, anybody with access to the field, is also able to ask for a re-encryption of the field's signature.

Patients				
id	**name**	**gender**	**name_sig**	**gender_sig**
0	Cully Barnaby	Female	*signature*	*signature*
1	Gavin Troy	Male	*signature*	*signature*
2	Deirdre McKenzy	Female	*signature*	*signature*

Doctors		
id	**name**	**name_sig**
0	dr. Bush	*signature*
1	dr. Friedlich	*signature*
2	dr. Jansen	*signature*

Diagnoses		
id	**diagnosis**	**diagnosis_sig**
0	Healthy	*signature*
1	Epilepsy	*signature*
2	Brain dead	*signature*

Link						
id	**patient**	**doctor**	**diagnosis**	**pat_sig**	**doc_sig**	**diag_sig**
0	0	2	0	*signature*	*signature*	*signature*
1	2	0	1	*signature*	*signature*	*signature*

Figure 4.5: A hospital administration system with integrity-protected tables.

We consider two signature schemes to use for creating signatures on data.

4.3.3 Boneh-Lynn-Shacham Signature Scheme

The Boneh-Lynn-Shacham (BLS) signature scheme [10] lets a signer produce a signature on a certain string $s \in \{0,1\}^*$. The private key of an asymmetric keypair is used to produce the signature. Anybody can verify the signature by using the public key to run the verification algorithm.

The scheme requires a Gap Diffie-Hellman group $\mathbb{G}$ of prime order p with generator g. Furthermore a hash function H that maps an arbitrary string to $\mathbb{G}$ is required: $H : \{0,1\}^* \to \mathbb{G}$.

There are three algorithms in this scheme:

KeyGen: this algorithm is used by the signer to create a keypair. First, we pick a random x from $\mathbb{Z}_p$, which is the private key, and set $sk = x$. Then, we compute the public key as $pk = g^{sk} \in \mathbb{G}$

Sign: this algorithm requires the private key sk and a message m. To sign an arbitrary string $m = \{0,1\}^*$, we first convert the message to an element in the group $\mathbb{G}$ by computing: $h = H(m)$. Then, we compute the signature $\sigma = h^{sk}$, which is an element in $\mathbb{G}$.

Verify: this algorithm requires the public key pk, a message m and the signature σ of which we want to verify that it is a correct signature on m. First, we compute $h_2 = H(m)$.

Then, check whether the type (g, pk, h_2, σ) is a so-called Diffie-Hellman tuple, which means it is of the form (q, q^a, w, w^a) for a certain q, w, a . If the signature is correct, the tuple (g, pk, h_2, σ) can be written as (g, g^{sk}, h_2, h^{sk}), provided that the hashes h_2 and h (from the Sign algorithm) are equal, in which case it is a valid Diffie-Hellman tuple.

Using the BLS scheme, signatures on data can be generated and verified. It provides a guarantee that the data is the same as when the person who possesses the private key signed it. The BLS scheme can be used to provide integrity of the data stored in a database. A patient can have another keypair, besides the keypair needed to provide confidentiality, with which the patient information is signed. Alternatively, the patient administration of a hospital can have a keypair that is used to sign the patient information.

We consider another signature scheme, which has the option of aggregating signatures so that they can be verified more efficiently.

4.3.4 Bilinear Aggregate Signature Scheme

The bilinear aggregate signature scheme (BGLS) [9, 10] supports, as the name implies, aggregation of signatures into one signature. An arbitrary party can aggregate any number of signatures without the need of any public or private key. Furthermore, the order of the signatures in the aggregation computation is irrelevant.

The BGLS scheme requires a bilinear group $\mathbb{G}$ of prime order p with a generator g. Similar to the BLS scheme, a hash function is required: $H : \{0, 1\}^* \to \mathbb{G}$.

There are five algorithms in the BGLS signature scheme:

KeyGen: this algorithm is equal to the KeyGen algorithm in the BLS signature scheme. The private key sk is randomly chosen from $\mathbb{Z}_p$ and the public key is computed as $pk = g^{sk}$.

Sign: to sign a message $m \in \{0, 1\}^*$ with a keypair (pk, sk), we compute a hash h of the public key and the message: $h = H(pk||m)$. The signature can be computed as $\sigma = h^{sk}$. Both h and σ are element of the group $\mathbb{G}$.

Verify: to verify a signature σ on a message m, we need the public key that corresponds to the private key that was used to sign the message. First, we compute $h = H(pk, m)$. The signature is valid if $\hat{e}(\sigma, g) = \hat{e}(h, pk)$.

Aggregate: this algorithm can be run on a set of n signatures in any order: $\{\sigma_1, \sigma_2, \ldots, \sigma_{n-1}, \sigma_n\}$. If we define σ_i to be the i^{th} item within the set of signatures, the aggregate signature can be computed as $\sigma = \prod_{i=1}^{n} \sigma_i$. As no keys or original messages are required, this algorithm can be run by an arbitrary party that is able to perform the multiplications.

Aggregate verification: this algorithm is used by a verifier to verify the aggregated signature σ on a set of messages $\{m_1, m_2, \ldots, m_{n-1}, m_n\}$ of which σ is the aggregated signature. For each message m_i in the set of messages, we have to obtain the corresponding public key pk_i. Then, we compute the hash $h_i = H(pk_i||m_i)$ for $1 <= i <= n$. Finally, we accept the aggregated signature σ if $\hat{e}(\sigma, g) = \prod_{i=1}^{n} \hat{e}(h_i, pk_i)$.

The reason for including the public key in the signature when hashing a message is a requirement that the BGLS scheme poses to the signatures. Signatures can only be aggregated when they are distinct, to avoid an attack where an adversary signs messages as another user [9]. This can be mitigated by including the public key of the signer in the hash. Now, there are no duplicate hashes signed by two different users, provided that the hash algorithm is collision resistant.

To compare the BLS scheme with the BGLS scheme, we consider the expensive pairing operation. Suppose we want to verify n signatures The BLS verification algorithm requires $2n$ pairings, since verifying whether the tuple is a Diffie-Hellman tuple requires 2 pairings per verification [11]. If we use the BGLS regular verification algorithm, $2n$ pairings are needed. If, however, we use the aggregation method is used, only $n + 1$ pairings are needed. Recall that the Aggregate algorithm does not use the pairing operation (only multiplication) and the Aggregate verification needs 1 pairing to compute $\hat{e}(\sigma, g)$ and n pairings to compute $\prod_{i=1}^{n} \hat{e}(h_i, pk_i)$. Hence the cost of the verification of multiple signatures, with regards to the pairing operation, is the lowest for the BGLS scheme when the signatures are aggregated. The number of usages of the hash function, which maps to a string to a group and which is an expensive operation as well, is equal for both the BLS and BGLS scheme.

In [10] a sequential aggregate signature scheme is mentioned, but the scheme requires the aggregation to be performed as part of the signing process and is, in our topologies, not suitable. The main reason for this is that a client can request any set of records from the database. Pre-aggregated signatures are therefore not optimal, unless, for each possible query on the database, an aggregated signature is stored, but that is not feasible in terms of storage and computational cost.

The performance of BGLS is worse than the performance of an RSA-based or DSA-based signature scheme according to [29]. However, the alternative schemes do not allow aggregating signatures of distinct signers, which is the case in our topologies and thus we do not consider them to be a good alternative. The BGLS scheme can be implemented with the same cryptographic principles as TBPRE (used for confidentiality), which simplifies the implementation process. There may be other signature schemes that can be used within our topologies, but we consider that to be future work.

Now that we have discussed methods for adding confidentiality and integrity, we continue discussing another important issue with our topologies: availability.

4.4 Availability

Availability, the correct functioning of the system, is especially essential to medical data. Even if data is stored encrypted and integrity-protected, if doctors are unable to access the data at all times, they will refuse to use the system, favouring storage on less-secure digital systems or even on paper.

Availability is an issue that is present even without adding confidentiality and integrity to the system. Basically, the database needs to function properly at all times and the workstations, used by, for example, doctors, should be able to access the database. High-availability for databases can be achieved by running synchronized databases at several locations. However, as this was present in the non-secure system as well, we do not consider the availability of the database itself.

When confidentiality and integrity are added to the system, the availability of the data is the main issue. The data should be available whenever it is needed. In a non-secure system, the availability of the data depends on the availability of the services, such as the database and network. Within a secured system, parts of the data can be encrypted, so even if the database is available and a doctor is able to retrieve the encrypted data, he still needs to decrypt the data before it is of any use. Thus, the ability to decrypt data is a key issue regarding the availability. Within the Type-Based Proxy Re-Encryption scheme that we use, the decryption depends on the availability of the ciphertext, the re-encryption key, the proxy and the private key of the delegatee. We assume the ciphertext to be available, as we discussed before. Secondly, the re-encryption key has to be stored in the proxy, or generated. If the re-encryption key has not been generated, the delegator needs to generate it, given his private key, the public key of the delegatee and the message type. One issue in this is the public key of the delegatee: how does the delegator obtain this key? If the public keys are stored in a central location, such as a intranet website, the availability of this location is of influence of the availability of the system. In most cases, though, the re-encryption key has been generated previously and is stored in the proxy. Without a working proxy, no re-encryption will take place and thus the doctors are unable to view confidential data such as patient information. This is easily mitigated though. Type-based proxy re-encryption does not place any limit on the amount of proxies. In contrary, by design it is possible to have multiple proxies so that the delegator can choose which proxy can perform the re-encryption on his behalf. In our topology, multiple proxies, all with the same re-encryption keys, could be used to mitigate the risk to the system when a proxy is malfunctioning.

5 Experimental Setup

In this chapter, we use the identified building blocks for implementing confidentiality and integrity to set up an experiment. The goal of our experiment is to measure the performance impact of the security overhead, compared to a non-secure system. We first describe the environment of the experiment, the dataset that we use and a description of the specific queries we run. Then, we discuss the prototype that we made and what parameters we use to perform the experiments.

5.1 Database Setup

In our experiments, we simulate a hospital with a database that stores patient information. The topology of the hospital and the database was discussed in Section 3.1 (Figure 3.1 in particular). For our experiment, we assume a hospital with a database that contains health records of the hospital and a set of patients, namely patients that are suspected to have epilepsy. These patients have had EEG recordings to measure their brain activity over a certain period of time, typically 20 minutes. The database stores the patient information of these patients. Furthermore, the raw measurement data and the diagnoses are linked to the patient information in the database.

patients
`id`
`givenname`
`surname`
`gender`
`birthday`
`streetaddress`
`zipcode`
`city`
`country`
`bloodtype`
`nationalid`
`weight`
`height`

sensordata
`id`
`patientid`
`file`
`doctor`

diagnosis
`id`
`sensordataid`
`result`
`doctor`

Figure 5.1: Showing the layout of the database tables before adding security. If the name of the column is underlined, it is a primary key.

The database consists of three tables to store the patient information (`patients`), measurement data (`sensordata`) and diagnosis (`diagnosis`). The tables are shown in Figure 5.1. The `patients` table contains the patient information. Each row represents one patient. The `id` is unique and we consider it to be public (e.g. not sensitive). The other fields are confidential, as they contain information about the patient that can be considered sensitive, such as the name, date of birth or social security number. Furthermore, all fields need to be integrity protected to prevent unauthorized changes. The `sensordata` table contains no confidential fields, though the fields need to be integrity protected. The field `patientid` links a row to a single patient from `patients`, `file` contains the filename of the file where the sensory data of the measurement is stored and in `doctor` the doctor who is responsible for the measurement. The measurement is stored in a file to preserve compatibility with current systems. If, for example, the measurement is an EEG recording, the sensory data is stored in the EDF+ format, which many applications can support. The measurement files can be of arbitrary size, as only some meta data and interpretations of the measurement (such as the diagnosis) are stored in the database. The measurements themselves are in principle unrelated to each other, unless measurements were taken from the same patient. In reality, the `sensordata` table could contain more meta data of the measurement file than we did. Finally, the `diagnosis` table lists diagnoses. Each diagnosis (a row) is linked to a row in the `sensordata` table by the `sensordataid` field. The diagnosis itself, such as 'epilepsy', is stored in the field `result` and in `doctor` the name of the doctor who made the diagnosis is stored. The `id` fields of each table should be an index (or primary key in SQL), `diagnosis.sensordataid` a reference (or foreign key in SQL) to `sensordata.id` and `sensordata.patientid` a reference (or foreign key in SQL) to `patients.id`.

The database we use is a simplification in the sense that we do not have as many fields as there would be in practice. For example, we use the name of a doctor, whereas in a real system there would most likely be a table `doctors`. The layout of the database affects the performance of the queries. Our results are based on the database as listed in Figure 5.1.

For running experiments, we need to add confidentiality and integrity to this database to obtain a *secure database*.

5.1.1 Adding Confidentiality

First of all, as discussed in Section 4.2, Attribute-Based Encryption (ABE) extends Identity-Based Encryption (IBE), and since ABE provides much more fine-grained access control, we discuss ABE and Type-Based Proxy Re-Encryption (TBPRE) only. For use in our topologies, we consider TBPRE to be a better choice than ABE, especially when dealing with user revocation. With TBPRE, it is much more simple and faster to remove a re-encryption key from a proxy than to re-encrypt all related ciphertexts and re-compute and distribute keys for other users. In our topology, we want to be able to quickly revoke access to minimize the risks of information leakage. We use the message type to provide access control.

We apply TBPRE to our topology, which is an electronic medical database that is only accessible by the hospital staff. First of all, each doctor generates his own keypair. Patient information is encrypted with a keypair of that particular patient, so each patient has a *virtual keypair* as well. A virtual keypair means that a patient does not actually has possession of the keypair. Instead, the hospital stores the keypair in a highly secure trusted *keyserver*. This keyserver requires a high level of security, as it stores the private keys of all patients and guarantees the access control to the confidential information. By using this construction, the information of the patient can be encrypted with his public key. The keyserver, on behalf of the patient, acts as a delegator, as in the TBPRE scheme. Whenever a doctor, the delegatee, needs to decrypt the information of a patient for the first time, he needs to file a request to the keyserver. The keyserver then generates a re-encryption key that converts a ciphertext encrypted with the public key of the patient, to a ciphertext encrypted with the public key of the doctor. This allows us to perform access control specific to each patient, which is one row in the `patients` table. By using a message type during encrypting and in the re-encryption key, we can specify which column we grant access to. For example, “name” and “date of birth” could be column names. By including the message type, we can grant a doctor access to only a subset of the columns of one particular patient. From the database point of view, we can perform access control on each individual cell in a database table.

To be able to insert the information of a new patient, the keyserver needs to generate and store a keypair for this patient. Then, each field of input, such as the name or date of birth, needs to be encrypted separately using the public key of the client and the name of the field as type. The name of the field is the same as the name of the column in the database table. For example, to encrypt the name of “Sanne de Bruijn”, we use her public key to encrypt her name and use the tag “name”.

All fields are encrypted, except for the `id`, the `gender` and `bloodtype` of the patient, which we use as an example for fields that are required to be stored unencrypted, as they are needed by researchers who have access to the measurement data, but are unable to perform a decryption. In practice, any subset of fields can be encrypted.

Changes to Topology

As previously discussed, if we want to use TBPRE to provide confidentiality, two new entities are needed: a keyserver and a proxy. These entities are defined as follows.

Keyserver: the keyserver is a fully trusted entity that stores private keys of patients. For this reason, we require the keyserver to have a dedicated administrator. It is critical that the keyserver is not administered by the database administrator.

The task of the keyserver is to manage access control and generate re-encryption keys. When a re-encryption key is needed by the doctor, the keyserver distributes the re-encryption key, provided that the access control policy allows the disclosure of the particular data to the doctor. An alternative name for the keyserver could be a *registration* server.

Proxy: re-encrypts ciphertexts so that a doctor can decrypt them. The proxy is unable to view the decrypted data. We assume a honest proxy, that performs the re-encryption as it should. Within the system, multiple proxies can exist to balance the workload.

Figure 5.2 shows the topology with the proxy and a keyserver added. The proxy and keyserver are connected to the switch to allow connections from the workstations. To analyse the security of our system, we consider a doctor who uploads patient information and then retrieves it again. First, the keyserver generates a keypair for the new patient, stores the private key securely and sends the public key to the doctor. Note that, in order for the system to be safe, the public key has to be authenticated (by means of a signature), or the connection between the doctor and the keyserver needs to be authenticated. We assume the latter in our system. The doctor encrypts the patient information on his workstation, constructs an insert query and sends the query to the database. Since the patient information is being encrypted at the workstation, it remains confidential during transportation and when it is stored in the database. When the doctor retrieves the data, the database sends the encrypted patient information to the doctor, who forwards it to the proxy and gets it back. Again, at all moments the patient information is encrypted.

In principle, the keyserver can be offline, since there is no need to keep it available constantly. When a new patient is registered, the keyserver needs to generate a keypair for this patient. When a doctor needs to decrypt data and there is no re-encryption key available, the keyserver will

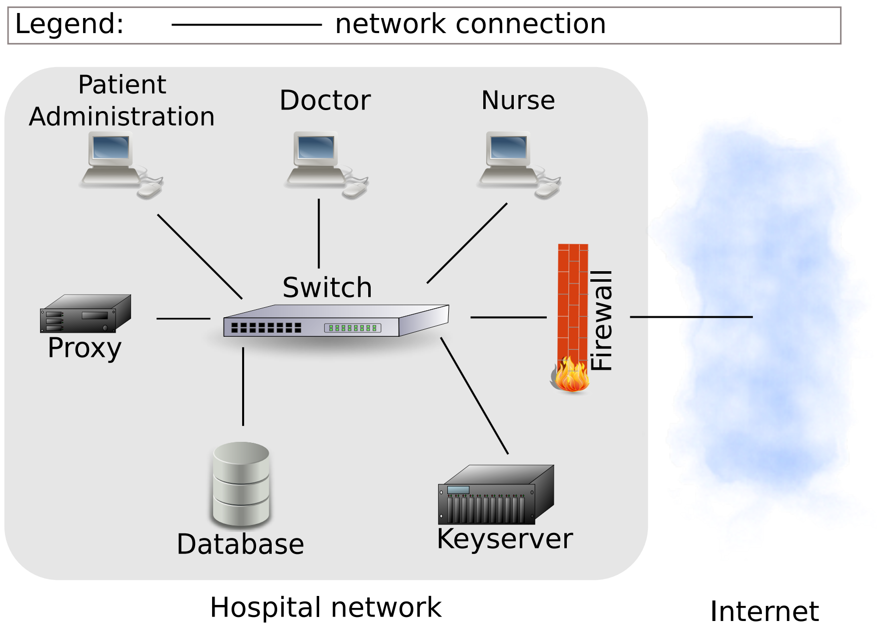


Figure 5.2: The topology with a proxy to support re-encryption, and a keyserver to store the patient keys.

have to generate one. These actions, however, are not frequently requested and do not require the keyserver to be available constantly. Furthermore, every time a certain public key is needed, for example when a user needs to verify a signature, the public key is needed and the keyserver can give it. However, public keys are required often and this would require a higher availability of the keyserver, compared to the situation where the keyserver is used for generating (re-encryption) keys only. Of these three tasks (generating keypairs, re-encryption keys and publishing public keys), the publishing of public keys can be delegated to another server, effectively reducing the need for a high-available keyserver. The keyserver can distribute authenticated lists of public keys to all users, or to another location instead, to overcome the dependency on the keyserver. The keyserver will remain a single point of failure. In theory it is possible to set up a second keyserver and synchronize it with the primary one, but then the private keys need to be synchronized as well, thus storing them in two locations. While this can solve the problem of a single point of failure, it is generally a a bad idea to store secret keys (such as the master-key) in different locations. Ideally, the keys need to be stored in as little places as possible, since they need to be kept really secret, as the confidentiality of the entire system depends on them.

The proxy plays a crucial role in the decryption phase and therefore it needs to be available at all times. Without a working proxy, doctors are unable to decrypt data. As mentioned, multiple proxies can be used to avoid this problem. Also, all proxies need to delete re-encryption keys when requested by the keyserver in order to support revocation. We will come back to this later, but first we discuss the retrieval of data.

Retrieving Data

A very frequent action within a health record system, including our system, is accessing the data. In Figure 5.3, the flow of the requests and exchanged messages is shown. We assume the doctor already has a keypair. First, the doctor has to send a request to the keyserver to ask for permission (a *decryption right*) for the patient he wants to decrypt the data of. The doctor has to specify of which patient he wants to see what fields. For example, the doctor could ask for permission to view the name of patient 42. The keyserver reviews this request using the access policy and determines whether the doctor should be granted a decryption right. If the doctor is allowed to see the name of patient 42, the keyserver will generate a re-encryption key using the private key of patient 42. Then, the keyserver sends the re-encryption key to the proxy. The doctor can then query the database, asking for the name of patient 42, and the database will send the ciphertext to the doctor. The ciphertext is encrypted using the public key of patient 42 and therefore the doctor is unable to decrypt it. Since the doctor wants to decrypt the ciphertext, he sends the ciphertext to the proxy. The proxy computes a new ciphertext using the re-encryption key and sends it back to the doctor. The doctor is able to decrypt the new ciphertext, thereby obtaining, in our example, the name of patient 42.

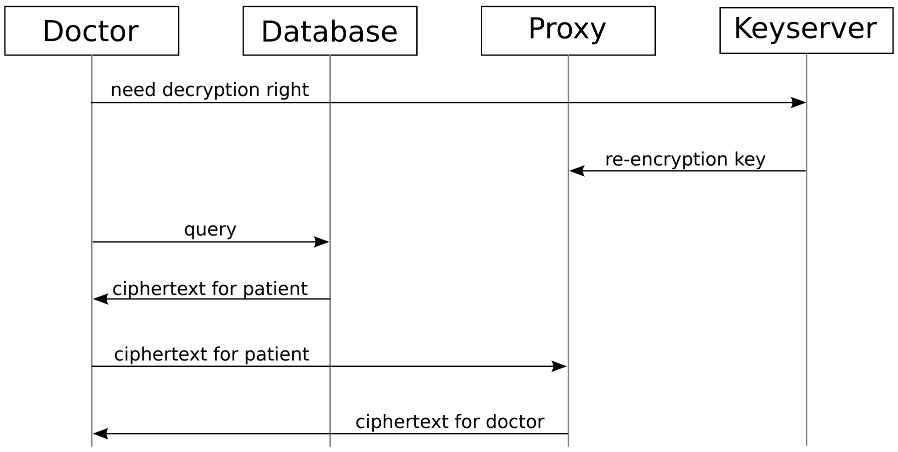


Figure 5.3: The message sequence diagram of data access within our system when a doctor wants to decrypt patient data for the first time. The doctor already possesses a keypair.

Revocation

For the confidentiality part of the security, the keypair of the doctor needs to be revocable in certain situations. Possibly, the doctor lost his private key, or he stored his private key on a tablet computer or laptop, that got stolen. To revoke the keypair of the doctor (or, more generally, the keypair of any delegatee), the keyserver needs ask the proxy to delete all re-encryption keys that re-encrypt to the doctor's keypair. Without re-encryption keys, the proxies are unable to re-encrypt ciphertexts and thus the doctor is unable to decrypt ciphertexts from the database. Effectively, this revokes the keypair of the doctor. We need to trust the proxy to delete re-encryption keys immediately for the revocation to work.

Revocation of a patient's keypair is not necessary, as only the trusted keyserver stores these keypairs. If, however, we need to revoke the keypair of a patient, we have to delete the secret key of the patient, delete the ciphertexts for the patient and create new ciphertexts.

5.1.2 Adding Integrity

In Section 4.3 we described two signature schemes that can be used to sign data and verify the signature of data and what fields need to be included in a signature in order to guarantee integrity. We concluded that the bilinear aggregate signature scheme (BGLS) is faster to use than the Boneh-Lynn-Shacham (BLS) signature scheme, as it required the least amount of pairings. Additionally, BGLS supports aggregation of signatures made by different signers, which can be used to compress an arbitrary amount of signatures to one.

Signatures are generated per field in the database. As discussed previously, for each field we need to integrity-protect, we create a hash on the name of the table, the columnname, the primary key and the value. Then, we can sign the hash using a private signing key and put the signature in the database. When we sign the hash of a field `field1`, we put the signature on the hash in a new field called `field1_sig`. Thus, for each field that needs to be signed, one column is added to the table.

Signing

The fields in all three tables have to be signed. Firstly, when the information of a patient is inserted in the `patients` table, it is first encrypted. When we want to protect the integrity of a field in the database, we have to sign the hash of the data as explained in Section 4.3.2. If, however, the data itself is encrypted, we have to encrypt the signature as well. Otherwise, an adversary is able to perform a *brute-force attack* by hashing all possible plaintexts and checking whether the signature validates for one of the hashes. Especially for fields such as the date of birth and other short or predictable fields, this is a threat. By encrypting the signature as well, an adversary is unable to find the plaintext by brute-force. However, encryped signatures cannot be aggregated, which is a limitation.

The actual signature is generated by using a private signing key. We decided to introduce the entity *patient administration* to the topology. In fact, the patient administration is just a special case of the Secretary entity that was discussed in Section 3.3.2. The patient administration is included in Figure 5.2. The patient administration has one keypair which is used for signing all patient information in the `patients` table. By signing the information, the patient administration states that it verified the validity of the information of the patients. The reason for this is that a doctor could sign the patient information, but in practice it will probably not be the doctor who inserts patient details, such as name and address, in the database. Alternatively, this can be changed to allow nurses or other staff, such as the secretary, to sign the patient information, but then some kind of list with signers need to be maintained. Another alternative is that each patient gets his own signing keypair. However, only the keyserver should have access to the private part of the signing key of each patient, just like with the keypair to ensure confidentiality. Then it is the question who will sign the data on behalf of the patient. Only the keyserver can sign data, as it has the private signing key of each patient. We do not want to give the keyserver the task of signing, which makes using a patient administration the best option.

The signatures on the tables `sensordata` and `diagnosis` are different, since we consider the fields in those tables not confidential. In `sensordata`, we sign the field `doctor` with the private key of the patient administration. The reason for this is that we assume that the patient administration controls which doctor treats which patient. The other fields (`patientid` and `file`) are signed using the signing key of the doctor that is mentioned in the `doctor` field. Similarly, in `diagnosis` the `doctor` fields are signed by the patient administration and the other fields are signed by that doctor.

Revocation

When using signing keys for the patient administration, used to sign patient information, and the doctors, used to sign diagnosis and measurements, the system needs to be able to revoke these keys. If a signing key needs to be revoked, a new signing key needs to be generated. Then, all users (such as doctors and nurses) within the topology need to be notified that the old signing key is no longer valid, and that it it replaced by the new signing key. From that moment onwards, even if an adversary has access to the private signing key, users of the system no longer accept signatures by that signing key. Furthermore, all signatures in the database, that are made by the signing key that needs to be revoked, need to be replaced by signatures that are created by the new signing key. This ensures that users are able to verify the integrity of the data once again.

Validation of Query Results

Special attention has to be given to the *validation* of the query results by the client (doctor). When the results of a query are validated, not only the integrity of the fields need to be verified, but also links between tables. If a client does not validate the results of a query, the database may produce fields with valid signatures, but the fields that are returned may not be what is requested. For example, if we ask the database to produce the result for the query '*Return the name of all patients that have epilepsy*', the database has to combine tables in order to link the diagnoses to the patients. The database could return the names of several patients, and we can verify the integrity of these fields. However, we also want to validate that the results are the answer to our query, e.g. that the returned patients indeed have the diagnosis epilepsy.

If, in a query, two tables are joined, additional fields may need to be requested in order to validate the integrity of the query results. This can make querying more complex. Suppose we have the tables in Figure 5.1. We could execute the following query:

```
SELECT s.file
FROM sensordata s, diagnosis d
WHERE s.id = d.sensordataid
AND d.id = 5;
```

This query selects the filename of the measurement file that is linked to a diagnosis with id '5'. To be able to validate the results of this query properly, we need to extend the query. The extended query becomes:

```
SELECT s.file, s.doctor, s.id, d.sensordataid, d.doctor
FROM sensordata s, diagnosis d
WHERE s.id = d.sensordataid
AND d.id = 5;
```

To validate this query, we first check whether `s.doctor` has a valid signature by the patient administration and obtain the public key of that doctor. Then, we verify the integrity of the `s.file`, which should be signed by `s.doctor`. Recall that a signature on a field contains of the tablename (in this case "sensordata"), the columnname ("file"), the primary key ("`s.id`") and the value of the field ("`s.file`"). Now, we have to validate the result of our query. We check the signature on `d.doctor` (by the patient administration) and obtain the public key of `d.doctor`. To check the link, we confirm that `d.sensordataid` has a valid signature of `d.doctor` and that it's value equals `s.id`. Then, we check whether the primary key of `d.sensordataid` is in fact "5".

Delayed Verification

Verifying the integrity of the query results, and validating the query results, may be slow. If no decryptions are required, a client can use the unvalidated and unverified results and do the verification and validation after the results have been displayed already. This method, which we call *delayed verification*, allows the client to use the query results even before they are verified and validated. Whether delayed verification can be used in practice depends on the type of query. For example, when a doctor wants to see the results of a large query (such as query 3), he can look at the results while they are not verified. Meanwhile, his computer verifies and validates the results of the query. This allows the doctor to start interpreting the data, assuming the data to be correct.

Even if a part of the results is encrypted, delayed verification can be used. After the sensitive data has been encrypted, the results are released to the doctor. Meanwhile, the signatures on the sensitive data are decrypted and the query results are verified and validated.

Key Distribution

So far, we assumed the keyserver to be responsible for publishing the public encryption keys of the patients, used to encrypt their patient information, and the public signing keys of the doctors and patient administration, used for verification of the data. The keyserver does not necessarily have to be responsible for this. The hospital may introduce a key distribution server that gets all authenticated public keys from the keyserver and takes care of the distribution of these public keys. In this way, the keyserver is only needed to generate and store patient keypairs, and to generate re-encryption keys.

5.2 Experiments

Our experiment consists of running several queries on a database and processing the results. We measure the time of completion of the queries. If a query returns encrypted or signed data, we decrypt or verify it. The database consists of the tables as shown in Figure 5.1. We create two databases: one with all data stored unsecured, and one with our security measures added. The databases are filled with dummy data that we generated. No real patient information is used in our experiments. Details on the dataset is discussed in Section 5.2.1. We run our experiments in four environments, which we will explain in Section 5.2.2. In Section 5.2.3 we describe the queries and the details on how to execute them. Finally, in Section 5.2.4 we describe the system specifications of the server that was used to run the experiments.

5.2.1 Dataset

We initialised the database with simulated data that we generated ourselves. The `patients` table was filled with 5000 entries of non-existing patients. We randomly generated three entries (measurements) in the `sensordata` table for each patient. Then, we randomly generated a diagnosis for one of these measurements. The diagnoses that are possible are “Normal”, “Epilepsy” and “Don’t know”. For each patient, one diagnosis was picked at random from the three possible diagnoses. Summarising, our database consists of 5000 entries in the `patients` table, 15000 entries in the `sensordata` table and 5000 entries in the `diagnosis` table. An example of the data that we stored can be found in Figure 5.4. The data is randomly generated and has no connection to real persons. In our database, we use a primary key on each `id` field, and a foreign key from

patients	
id	1
givenname	Shannon
surname	French
gender	female
birthday	3/24/1953
streetaddress	45 Iolaire Road
zipcode	NP4 2PT
city	NEW INN
country	GB
bloodtype	A+
nationalid	WT 67 44 84 A
weight	55.8
height	162

sensordata	
id	1
patientid	1
file	gfxdwmv.xml
doctor	jansen
id	2
patientid	1
file	bszwrsl.xml
doctor	jansen
id	3
patientid	1
file	gsjadfv.xml
doctor	jansen

diagnosis	
id	1
sensordataid	3
result	Epilepsy
doctor	jansen

Figure 5.4: An example of the dataset. All data of one (fictive) patient is shown. The tables are shown in a way different from the usual representation, to make them fit on the page.

`diagnosis.sensordataid` to `sensordata.id`, and a foreign key from `sensordata.patientid` to `patients.id`.

5.2.2 Query Environments

In order to compare the performance of queries in a non-secure database to running queries in a secure database, we use query environments. We distinguish four query environments:

Environment A: a non-secure database, with no confidentiality and no integrity.

Environment B: only confidentiality is provided. The data is not integrity protected.

Environment C: only integrity is provided. Confidential data is stored unencrypted.

Environment D: both confidentiality and integrity are provided. Confidential data is encrypted and all data is integrity-protected.

We run several queries on each of these environments. This allows us to measure differences in performance between environments. Furthermore, in environments C and D, the verification is measured both with and without aggregation to test the influence of aggregation on the execution time.

5.2.3 Queries

There are four queries that are executed during the experiment. The queries show differences between using the non-secure database (environment A), adding confidentiality (environment B), adding integrity (environment C) and adding both confidentiality and integrity (environment D). We execute the following queries in all environments:

1. *Display all patient information of patients 51, 99, 1287, 2841 and 4389.*

   This query requests for several patients all columns of the `patients` table.

2. *Display the name, gender, date of birth and most recent diagnosis (if any) of patient 3852.*

   In this query, some sensitive information that needs to be re-encrypted, is requested. The database is asked to link all three tables in order to find the most recent diagnosis (e.g. find the primary key with the highest value) belonging to patient 3852. To do this, the database has to join `patients` on `sensordata`, and `sensordata` on `diagnosis`. If the environment requires us to verify the query results (e.g. environments C and D), we have to validate the joining of the tables as well.

3. *Display the id's of all patients who have epilepsy.*

   No sensitive data is requested. This query is similar to query 2, except that instead of only one result, this query returns the id's of all patients with epilepsy. The database has to join the `sensordata` and `diagnosis` table and return `sensordata.patientid` when a diagnosis is epilepsy. Now, similar to query 2, we need to verify the joining in environments C and D. This query stresses the schemas that provides integrity.

4. *Display the name of all patients.*

   This query is computationally most expensive in environments B and D, where confidentiality is provided. Of all patients, the name is shown. To display one name, a re-encryption and a decryption are required. This query is meant to measure the extreme case where a part of the records of all patients have to be decrypted, hereby stressing the schemas that provides confidentiality.

The SQL queries for all environments and how the integrity of these queries is verified are shown in Appendix A.

5.2.4 System Specifications

For running the experiments, we used a dedicated server with the following specifications:

Operating system	OpenSUSE 11.3 (Linux 2.6.34) 64-bit
CPU	Quad-core Intel Xeon E5420 (clockspeed 2.50 GHz)
RAM	32 GB
Harddisk	4 TB raid-0 array

During the experiments we could use all resources, as there were no other users active on the system. The server was running a PostgreSQL server (version 8.4.4) that we used as database software.

5.3 Prototype

In order to run the experiments, we implemented a prototype. The prototype has been implemented in the C programming language. Although we tried to avoid platform-specific code, we have only tested and run the prototype on the Linux operating system. In order to run the prototype, several libraries are required.

For the cryptographic basics, such as mathematics with elliptic curves, we use the pairing-based cryptography (PBC) library version 0.5.12 by Ben Lynn [26]. The PBC library provides us with the primitives to implement BGLS (integrity) and TBPRE (confidentiality). Recall that TBPRE requires a hashing function $H2 : \{0, 1\} \rightarrow \{0, 1\}^l$ for some l. We implemented this by using SHA256 provided by libgcrypt [19]. Whereas the model uses a keyserver, we simplified this in our prototype by storing all keys in XML files. For reading and writing XML files, we used the Mini-XML library [35]. Finally, we make a connection to PostgreSQL through the libpq library [30]. Even though we used PostgreSQL as database server and use a library specific to PostgreSQL to connect to the database, the queries are not specific to any database.

The PBC library supports several groups of elliptic curves. On the website of PBC [26], the groups are benchmarked by the average time for a pairing to be computed. The conclusion of the PBC library website is that type a pairing is the fastest. However, during implementation, we noticed that for a type a pairing the pairing speed is indeed fast, but turning a bit-string into a group element (called *mapping*, e.g. $\{0, 1\}^* \rightarrow \mathbb{G}$) is very slow. Also, we discovered that some groups are slow with the pairing operation, but very fast in the mapping operation. Hence, we wrote a benchmark tool ourselves and tested the curves for both the pairing and mapping operation, but also all algorithms that TBPRE and BGLS use. Our conclusion is that the type a curve is the fastest for our scenario. The full results of our benchmark can be found in Appendix B.

5.3.1 Running the Queries

Our implementation runs the entities (such as clients and keyserver) locally and not over a network. This avoids many socket-related issues and makes the timing of the encryption overhead more accurate. The database also runs at the same machine, and we use a localhost-only network connection to send queries and retrieve the data.

To run the queries, we execute each query 100 times on each environment to get the average execution time and standard deviation. Executing a query involves sending a request to the database, fetching the result and to put the result on the screen. Depending on the environment that is used, the results have to be decrypted and/or verified before they can be put on the screen. To avoid performance issues due to putting large amounts of text on the screen (by calling the `printf` method in C), we filter the output so that only timing statistics are shown.

In order to determine how our solution scales, we split our dataset (of 5000 patients) in three parts with each one third of the patients. We conduct the experiments with 1666 patients, with 3333 patients and with the entire dataset of 5000 patients. As we defined queries with fixed identifiers, we cannot use just any subset of the dataset of 5000 patients. Instead, we note that our queries use identifiers that are 0 mod 3. First, we run the experiments on the full dataset of 5000 patients. Then, we delete all patients and the corresponding sensordata and diagnosis rows, where the `patientid` is 2 mod 3. This deletes a third of the dataset. Similarly, when we want to create a dataset of 1666 persons, we delete all rows belonging to the patients with id ≥ 1 mod 3.

6 Results

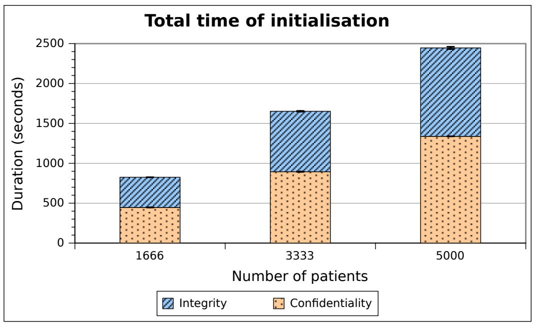
In this chapter, we discuss the results of our experiments. First, we mention the results of comparing elliptic curves in Section 6.1. Then, in Section 6.2 we discuss the results on initialising a database. Finally, we discuss the results of executing the four queries in Section 6.3 (query 1), Section 6.5 (query 2), Section 6.5 (query 3) and Section 6.6 (query 4).

6.1 Selecting an Elliptic Curve

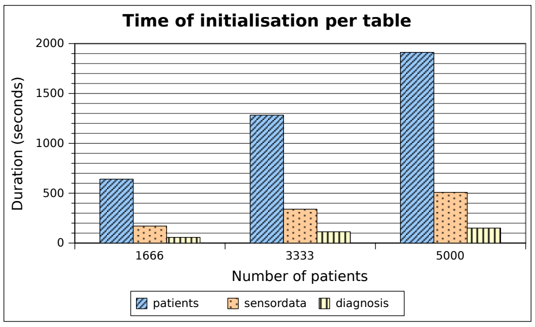
In the prototype section in the previous chapter, we already referred to the results in Appendix B, where we discuss the differences in performance between curves. However, the figures in Appendix B also show the relative difference in performance between various operations, such as encryption and decryption, or encryption and signing, for all curves. The results in the next sections are based on a type *a* curve.

6.2 Database Initialisation

First we discuss the results of the initialisation of a secure database with non-secure data. We created a non-secure database with the dataset, as described in 5.2.1. Then, we initialised the secure database with the dataset, which requires generating patient keys, encrypting and signing the data and storing the data in the secure database.



(a) Categorised by type of security



(b) Categorised by table

Figure 6.1: The execution time of initialising a secure database with data.

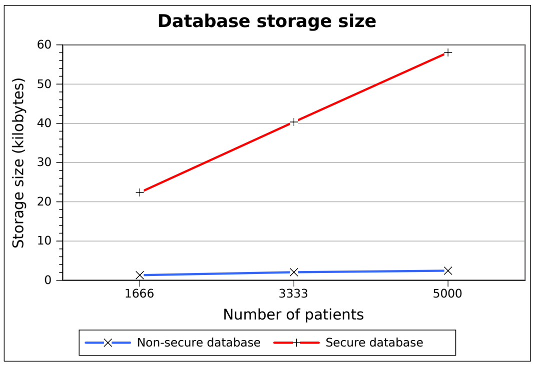


Figure 6.2: The storage costs for a non-secure and secure database compared.

In Figure 6.1 the total time of initialisation is shown for a secure database with 1666, 3333 and 5000 patients. Figure 6.1a shows the time to initialise the database for both integrity and confidentiality, which scales linearly when the dataset is increased. This is expected, as, in our dataset, each patient and the corresponding measurement and diagnosis contains the same amount of data and is independent of the other patients. Therefore, the computational cost of converting the data is equal for each patient.

In Figure 6.1b, we again show the initialisation time, but now categorised per table. The patients table takes the longest time, which makes sense, as it contains many fields that require requires both encryption and integrity protection. Note that the sensordata and diagnosis table contain the same amount of fields, which all have to be integrity protected, but it takes considerably longer to initialise the sensordata table, compared to the diagnosis table. This can be explained by looking at our dataset. Each patient has three rows in the sensordata table, and only one in the diagnosis table. The figure supports this and shows that the sensordata bar is three times higher than the diagnosis bar.

Figure 6.2 shows the storage costs for encryption of a regular, non-secure database, and secure databases. The storage costs are calculated by the database and represent the total cost of storing the data in the database, including a small overhead for the indexing of primary keys (about 2.1 MB for 5000 patients, which is about 3.5% of the total storage costs). After encryption, the storage size increases due to overhead that is added by Type-Based Proxy Re-Encryption, so that the ciphertext can be decrypted or re-encrypted. For signing, the original data is not touched. Instead, a fixed-size field that contains the signature is added to the table. Therefore, the scaling increases, linearly, depending on the number of fields. The figure confirms this. Note that the same databases are used for all query environments.

6.3 Query 1

We recall the description of the first query from Section 5.2.3: *Display all patient information of patients 51, 99, 1287, 2841 and 4389.* Figure 6.3 shows the execution time of this query, for all environments with a dataset of 5000 patients. For environment A (the non-secure environment), the execution time is not shown in the graph, since it is very low compared to the other environments. Running the query in environment A on a database with 5000 patients takes 0.48 ms (± 0.13 ms). In the figure, the bar for environment B (with only confidentiality) is split in two parts: the client/server part and the proxy part. For the doctor (or client) to decrypt the

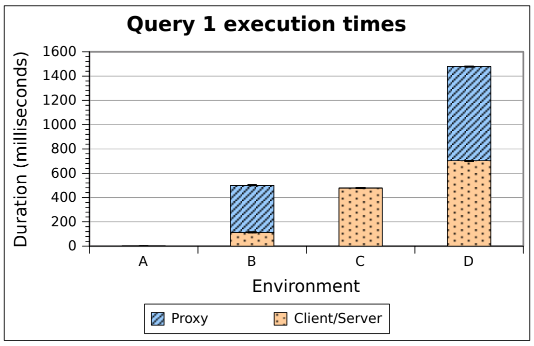


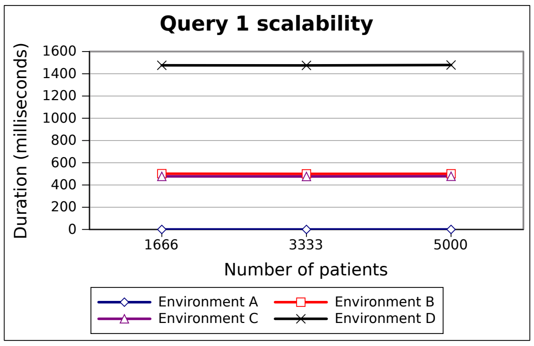
Figure 6.3: Time of execution of query 1 for all environments split in the processing time of the proxy and the rest, such as the database server for answering to the query and the client for performing a decryption.

result of the query, a re-encryption is required by the proxy. We observe that for environment B, the computational cost for the proxy is higher than the computational cost for the database server and the doctor combined. Environment C (only integrity) does not use a proxy, as no confidentiality is provided and thus no re-encryption is needed. Finally, executing the query in environment D (confidentiality and integrity) takes the most time. The cost for the proxy is higher than for environment B, while it may be expected to be equal. The reason for this is that signatures on sensitive data have to be encrypted as well. Also, before verification, the signatures have to be decrypted again.

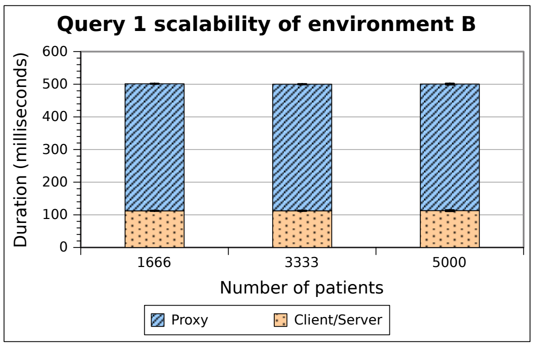
When looking at the non-secure environment (A) versus the secure environment (D), the difference in performance is considerable. In environment A and a database of 5000 patients, it takes about 0.48 ms (± 0.13 ms) to display the information of five patients, while displaying the same information in environment D takes 1478.80 ms (±1.04 ms), which is about 3080 times slower. However, we consider it to be workable if a doctor has to wait an additional 1.5 seconds before the information is displayed. The total execution time of the query depends mostly on the re-encryption and decryption algorithms, so the decision whether the query is workable in practice should take the number of fields to be decrypted into account, as well as the absolute execution time.

In Figure 6.4, the scalability of our system is shown. For all environments, the execution time for query 1 on different amounts of patients is roughly the same, as can be seen in Figure 6.4a. As query 1 requests a fixed amount of patients, the execution time should not be influenced by the size of the database, except for the time the database takes to find the records. In Figure 6.4b, we show the the scalability of the proxy and the client and server operations. Again, as a fixed amount of records is requested, the execution time remains the same. The same applies for Figure 6.4c, where the scalability of the proxy and client/server operations are shown for environment D, which ensures both confidentiality and integrity.

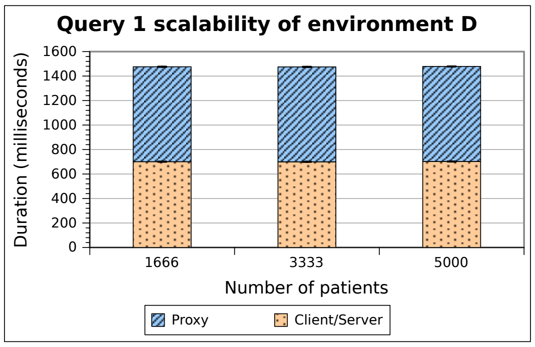
The BGLS signature scheme provides the ability to aggregate signatures. We executed the query with and without aggregation on signatures, to see whether there is indeed a performance increase as described in theory. In Figure 6.5, the increase in performance by using aggregation is shown. Indeed, the increase in performance is nearly 20% in environment C, where integrity verification is a big part of the computational costs of the entire query. In environment D, the increase in percentage of the total execution time of the query is less, since aggregation does not affect the confidentiality algorithms, but it is still about 7.5% faster than regular verification.



(a) Scalability per environment



(b) Scalability of the proxy



(c) Scalability of the proxy

Figure 6.4: Scalability of query 1 for the environments and, in the relevant environments, for the proxy.

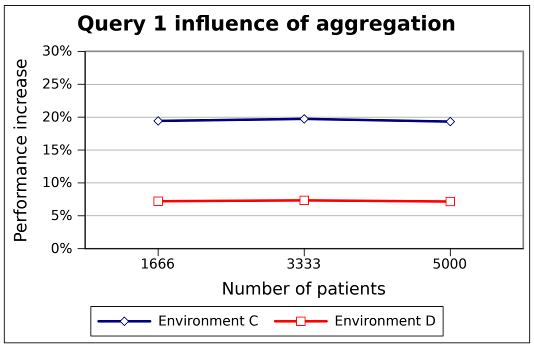


Figure 6.5: The increase in performance when using aggregating signatures during verification, compared to sequential verification of each signature.

6.4 Query 2

The second query of our experiments was: *Display the name, gender, date of birth and most recent diagnosis (if any) of patient 3852.* The execution times of running this query, for a dataset of 5000 patients, on all four environments are shown in Figure 6.6. For this query, verification of the integrity of the results, as well as validation of the query takes the most time. This can be seen by looking at the differences between environment B (confidentiality) and environment C (integrity).

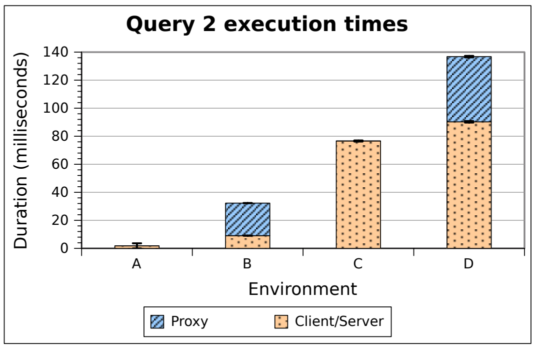


Figure 6.6: Time of execution of query 2 for all environments split in the processing time of the proxy and the rest, such as the database server for answering to the query and the client for performing a decryption.

The total execution time of environment D is 136.80 ms (±0.59 ms), which is 76 times slower than the 1.79 ms (±1.83 ms) it takes to execute the query in environment A, for a dataset of 5000 patients. The execution times for smaller datasets is similar, since the query requests information of one patient regardless on the size of the dataset.

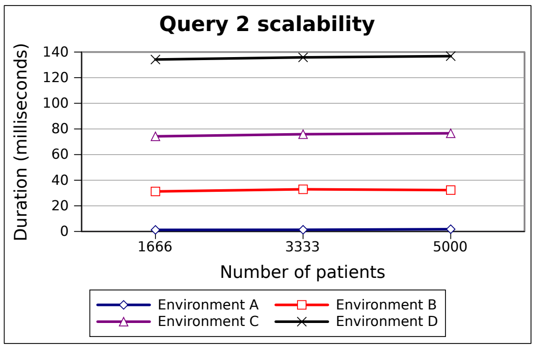
The scalability of the second query is shown in Figure 6.7. Since the result of the query does not depend on the size of the dataset, no shocking results were found.

In Figure 6.8, the influence of aggregation on the verification of signatures in the BGLS signature scheme is shown. Compared to the first query, query 2 uses relatively more verifications, since only two fields (name, date of birth) require decryption. All four requested fields require verification, as well as some other fields that are used to validate the query result. This is best shown in environment C, where aggregation makes the entire query execute around 25% faster. When fields and signatures have to be decrypted, in environment D, aggregation causes an improvement in performance of 11-12%.

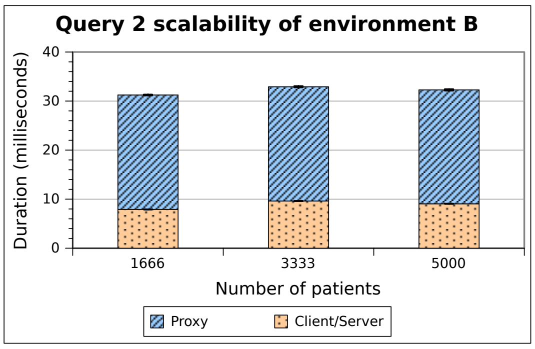
6.5 Query 3

The third query is defined as: *Display the id's of all patients who have epilepsy.* This query is interesting from a theoretical perspective, since it requests the id's of roughly a third of patients (note that there are three diagnoses, one of which is epilepsy, and each patient is assigned one diagnosis at random, thus the expected number of patients with epilepsy is 33%). It offers a way to see the scalability of performing verifications on a large subset of the data.

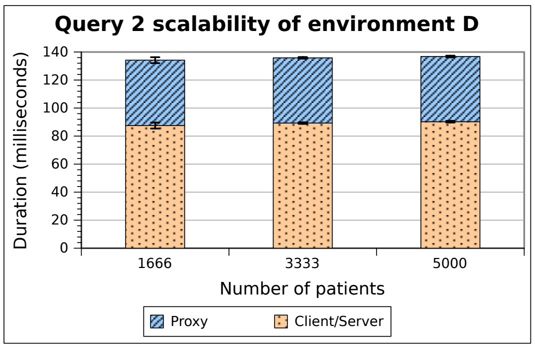
The execution times for a dataset of 5000 patients are shown in Figure 6.9. There are no results for environments B (confidentiality) and D (confidentiality and integrity), since there are no



(a) Scalability per environment



(b) Scalability of the proxy



(c) Scalability of the proxy

Figure 6.7: Scalability of query 2 for the environments and, in the relevant environments, for the proxy.

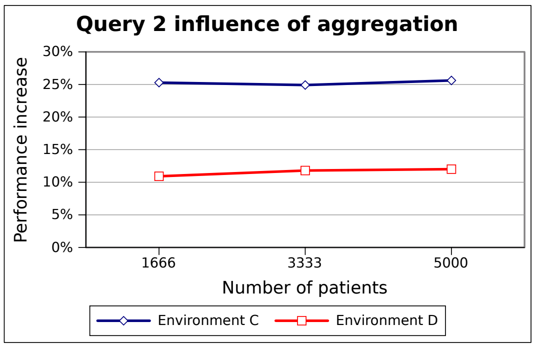


Figure 6.8: The increase in performance when using aggregating signatures during verification, compared to sequential verification of each signature.

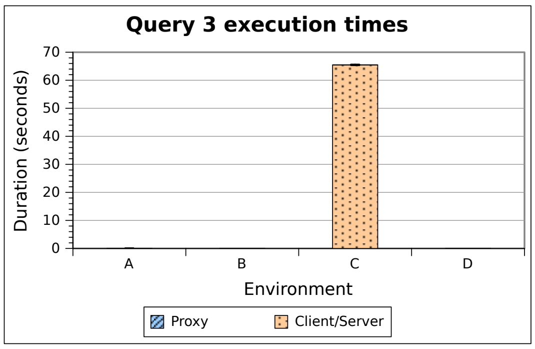
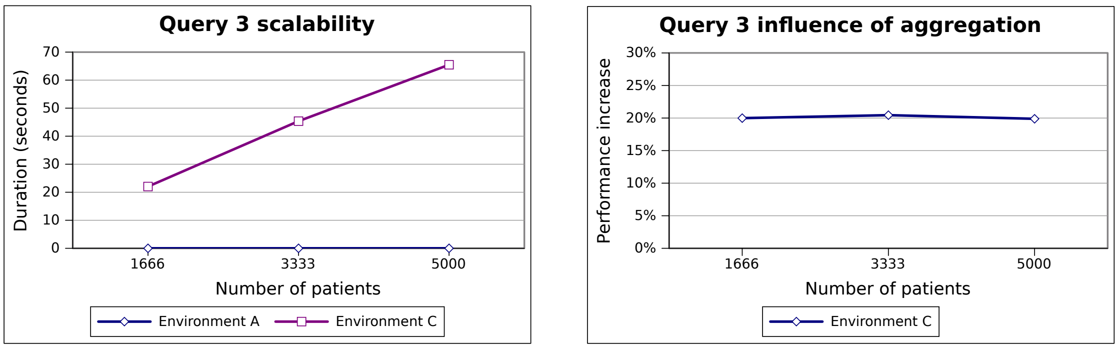


Figure 6.9: Time of execution of query 3 for the environment split in the processing time of the proxy and the rest, such as the database server for answering to the query and the client for performing a decryption.



(a) Scalability per environment

(b) Influence of aggregation

Figure 6.10: Scalability of query 3 for both relevant environments and the influence of aggregation.

decryptions involved and therefore the results for environment B are the same as environment A, and the results for environment D are the same as the results for environment C. For environment A (the non-secure environment), it takes 5.93 ms (±5.42 ms) to execute the query for a dataset of 5000 patients. When the query is executed in environment C (integrity), the execution time of the query is very long: 65.49 s (±87.73 ms) for a dataset of 5000 patients. The execution time of the query for environment D is over 11000 times slower than the execution time for environment A.

In Figure 6.10a, the scalability of the system when running the third query is shown to be linear. For this query, the number of returned results, and therefore the number of verifications to perform, grows linearly as the dataset grows. We conclude that, for environment C, our system scales linearly. The influence of aggregation on the execution time of the third query is roughly 20% for each dataset, as shown in Figure 6.10b. This leads to the conclusion that aggregation is interesting for any size of the dataset.

6.6 Query 4

The fourth and last query was defined as: *Display the name of all patients.* This query requires the first name and surname of all patients to be decrypted and, if the environment requires it, to be verified. Data from only one table is requested, which limits the verification to only these two fields and not any validation on the database's joining of tables, such as in query 3. Figure 6.11 shows the execution times of the query in all environments but environment C (integrity), for a dataset of 5000 patients. Environment C is not included since the query requires us to decrypt data. As the signatures are encrypted as well, we are unable to run the query in environment C. We expect the results of environment C to be lower than the execution time of the client/server for environment D, as environment D requires a decryption of the signatures, which environment C does not have. Environment A (no security) is included in the figure, but the duration is too short to be visible in the graph: 14.82 ms (± 2.43 ms) for a dataset of 5000 patients. Compared to environment A, the duration of executing the query in environment D (confidentiality and integrity) is massive: 280.45 s (± 256.40 ms) for a dataset of 5000 patients. Environment D is almost 19000 times slower than environment A.

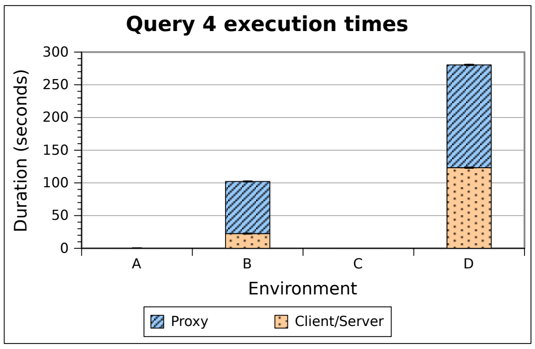
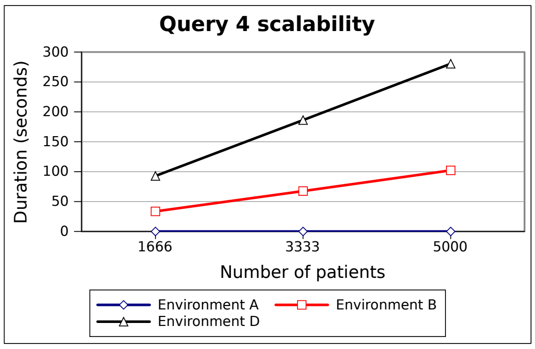


Figure 6.11: Time of execution of query 4 for all environments split in the processing time of the proxy and the rest, such as the database server for answering to the query and the client for performing a decryption.

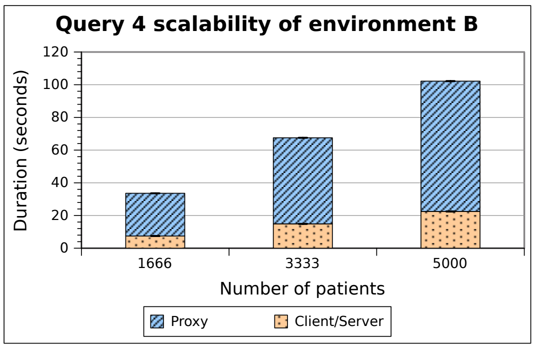
In practice, doctors will not request the names of all patients regularly. Possibly, there are moments that the patient administration wants a list of all patients, but as the figure shows, this is a very expensive query to run on a secure database (environment D in particular).

In Figure 6.12 the scalability of the system is shown. Figure 6.12a shows the linearity of the system. Environment D scales with a different scaling factor than Environment B (confidentiality), since each signature that has to be verified, needs to be decrypted first. In environment B, the proxy requires the most time for the tested sizes of our dataset, as shown in Figure 6.12b. The other computations, such as the decryption that the doctor's workstation has to perform, require less time. For environment D, the proxy requires relatively less time, since the client (the doctor) needs to verify the results of the query.

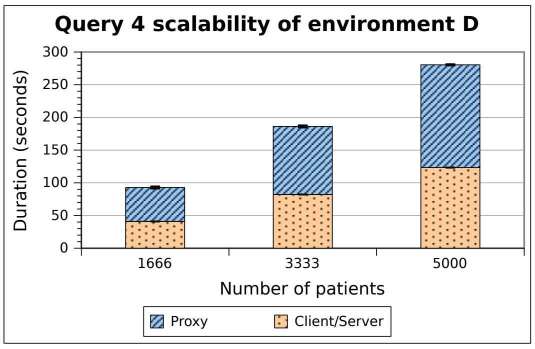
Finally, Figure 6.13 shows the influence of aggregation on the verification of signatures. Clearly, the improvement of performance is a constant percentage of the query time. For query 4, aggregation makes the query roughly 6.5% faster.



(a) Scalability per environment



(b) Scalability of the proxy



(c) Scalability of the proxy

Figure 6.12: Scalability of query 4 for the environments and, in the relevant environments, for the proxy.

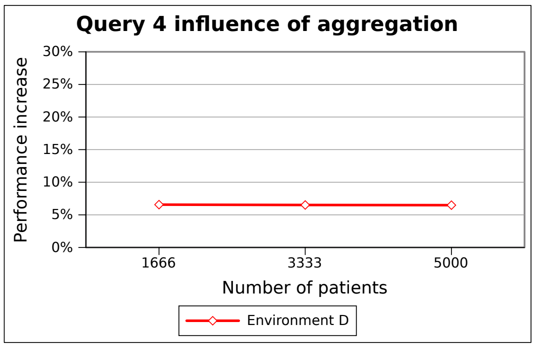


Figure 6.13: The increase in performance when using aggregating signatures during verification, compared to sequential verification of each signature.

7
Conclusions

In this thesis, we explored the security of medical databases and identified security requirements of medical databases. We discussed encryption schemes that can be used to prevent information leakage and signature schemes to protect information integrity. A prototype was made to implement security for a medical database and to test the performance penalty of the security. The results show that there is a big performance penalty. When the records of a few patients are requested, the delay caused by security is high. For example, requesting all patient information of five patients takes no more than 1.5 seconds, as opposed to 0.5 milliseconds without security. However, we need to consider the delay of adding security as absolute number, not the relative delay compared to the non-secure situation. We consider 1.5 seconds to load the patient information of five patients acceptable. When a large set of data needs is requested from a secure database, the execution time of the request grows linearly with the number of requested patients.

In the Introduction, we listed several research questions that needed to be answered. The first question was *How can patient information be encrypted, so that the database cannot decrypt it?* In Chapter 4, we identified the building blocks for providing confidentiality. We concluded that Type-Based Proxy Re-Encryption is the best solution to provide confidentiality, while allowing the revocation of access when needed. We implemented Type-Based Proxy Re-Encryption and identified the performance of the various algorithms. Details on this comparison can be found in Appendix B.

The second question was *How can we protect the integrity of medical data within a database, if we do not trust the database?* In Chapter 4, we discuss how the integrity of the data can be verified and how the result of a query can be validated. We advise the use of the Bilinear Aggregate Signature Scheme to make signatures of the data, and discuss how the integrity can be verified by the clients. We tested the performance impact of aggregation on the signatures and conclude that aggregation may speed up the verification by up to 25%.

The third research question was *What is the performance impact of providing confidentiality and integrity of the information?* We created an experimental setup and implemented a prototype to test the performance of the added security. The experiment and prototype are discussed in Chapter 5. In Chapter 6, we discuss the results of the experiments and conclude that the interpretation of the performance impact depends on the type of usage of the database. For a doctor who needs to decrypt patient records during patient care, the performance penalty is a second at maximum. If a doctor, or somebody from the patient administration, wants to request data of many patients, the performance is significantly lower. For example, decrypting and verifying the full name of 5000 patients takes up to 5 minutes. Whether this is acceptable, depends completely on the situation in which the system is used.

Finally, our main research question was *How to secure a medical database?*. In Chapter 2, we discussed the usage of a medical database and introduced an example of a medical database to protect. We discussed the security requirements to a medical database in Chapter 3, that were used to determine which building blocks to choose and define how a medical database should be secured. We discussed how confidentiality can be implemented to prevent information leakage and how to implement signatures to protect information integrity. This thesis discusses the required building blocks, that others have proven to be secure, for designing a secure medical database. The solution we present allows the outsourcing of a medical database, since our methods prevent the database from viewing the data that is being stored in the database. But even if the database is located within a hospital, it should be considered to implement the security features discussed in this thesis to prevent information leakage or unauthorized changes of information by hackers or malicious employees.

7.1 Future Work

There are several ways of dealing with medical data in a database. Patient information can be inserted in a medical database. To provide proper security, we add signatures and encrypt the information to protect the integrity and confidentiality, respectively. From that moment onwards, it can be retrieved from the database, or not, if the database failed to retrieve it. If the data should be deleted, one can request the database to delete the data, but even if the database refuses to delete the data, the confidentiality is ensured if the secret key and all re-encryption keys are deleted. There is, however, one other scenario: an update to the data. At the moment a part of the data is updated, signatures are created on the new data and, if the data is sensitive, the data is encrypted. The question is, however, how to detect whether the retrieved data is in fact the most recent data, and not an older version? When the database is initialised with data, the database is in a consistent state regarding the integrity of the data in a security setting. After an update, two consistent states exist: one that existed before the update, and a new state with the new data and new signatures on that data. This is particularly relevant when the database uses *versioning*, as discussed in Section 2.2.2. The question of how to determine, as a client, which state the database uses to retrieve data, is considered future work.

Another open problem is the disability to execute certain queries. Due to the encryption of data, the database is unable to execute a query that returns certain patients based on a field that is encrypted. The query '*Give all patients who have Nikita as first name*' is an example of such a query. A solution to this would be to return all first names and let the client decrypt each of them. However, this requires the client to obtain decryption rights to all first names, a right the keyserver is unlikely to grant. Possibly the notion of *searchable encryption* provides a solution, but the combination of searchable encryption and type-based proxy re-encryption has yet to be explored.

So eine Arbeit wird eigentlich nie fertig, man muss sie für fertig erklären, wenn man nach Zeit und Umständen das mögliche getan hat.

– Johann-Wolfgang von Goethe, Italienische Reise (1787)

Bibliography

[1] R. Anderson. Clinical system security: interim guidelines. *British Medical Journal*, 312(7023):109–111, 1996.

[2] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. H. Katz, A. Konwinski, G. Lee, D. A. Patterson, A. Rabkin, I. Stoica, and M. Zaharia. Above the clouds: A berkeley view of cloud computing. Technical Report UCB/EECS-2009-28, EECS Department, University of California, Berkeley, Feb 2009.

[3] A. Avižienis, J.-C. Laprie, B. Randell, and C. Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE Transactions on Dependable and Secure Computing*, 1(1):11 – 33, Jan.-March 2004.

[4] E. Barkan, E. Biham, and N. Keller. Instant ciphertext-only cryptanalysis of GSM encrypted communication. *Journal of Cryptology*, 21:392–429, 2008.

[5] O. Benjelloun, A. Das Sarma, A. Halevy, M. Theobald, and J. Widom. Databases with uncertainty and lineage. *The VLDB Journal*, 17:243–264, 2008.

[6] J. Bethencourt, A. Sahai, and B. Waters. Ciphertext-Policy Attribute-Based Encryption. *2007 IEEE Symposium on Security and Privacy*, pages 321–334, May 2007.

[7] E. Biham and P. Kocher. A known plaintext attack on the PKZIP stream cipher. In B. Preneel, editor, *Fast Software Encryption*, volume 1008 of *Lecture Notes in Computer Science*, pages 144–153. Springer Berlin / Heidelberg, 1995.

[8] D. Boneh and M. Franklin. Identity-based encryption from the Weil pairing. *SIAM J. of Computing*, 32(3):586–615, 2003.

[9] D. Boneh, C. Gentry, B. Lynn, and H. Shacham. Aggregate and verifiably encrypted signatures from bilinear maps. In E. Biham, editor, *Advances in Cryptology EUROCRYPT 2003*, volume 2656 of *Lecture Notes in Computer Science*, pages 641–641. Springer Berlin / Heidelberg, 2003.

[10] D. Boneh, C. Gentry, B. Lynn, and H. Shacham. A survey of two signature aggregation techniques. *RSA Cryptobytes*, 6(2):1–9, Summer 2003.

[11] D. Boneh, B. Lynn, and H. Shacham. Short signatures from the weil pairing. In C. Boyd, editor, *Advances in Cryptology ASIACRYPT 2001*, volume 2248 of *Lecture Notes in Computer Science*, pages 514–532. Springer Berlin / Heidelberg, 2001.

[12] J. Buchmann. *Introduction to Cryptography.* Springer, 2004. ISBN 0-387-20756-2.

[13] I. Burrell. Millions of medical records lost by the NHS. *The Independent*, July 1, 2011.

[14] Connecting for Health. The personal health working group final report. Technical report, June 2003. Markle Foundation.

[15] A. de Keijzer. MeDIA: Medical data management. Technical Report TR-CTIT-11-17, Centre for Telematics and Information Technology, University of Twente, Enschede, The Netherlands, 2011. ISSN 1381-3625.

[16] C. M. DesRoches, E. G. Campbell, S. R. Rao, K. Donelan, T. G. Ferris, A. Jha, R. Kaushal, D. E. Levy, S. Rosenbaum, A. E. Shields, and D. Blumenthal. Electronic health records in ambulatory care a national survey of physicians. *New England Journal of Medicine*, 359(1):50–60, 2008.

[17] R. Dhamija, J. D. Tygar, and M. Hearst. Why phishing works. In *Proceedings of the SIGCHI conference on Human Factors in computing systems*, CHI '06, pages 581–590, New York, NY, USA, 2006. ACM.

[18] M. Dupler. Computer servers hacked. *Tri-City Herald*, January 13, 2011.

[19] Free Software Foundation. The libgcrypt library. `http://www.gnupg.org/documentation/manuals/gcrypt/`, July 2010. Version 1.4.6.

[20] V. Goyal, O. Pandey, A. Sahai, and B. Waters. Attribute-based encryption for fine-grained access control of encrypted data. In *Proceedings of the 13th ACM conference on Computer and communications security*, pages 89–98. ACM, 2006.

[21] K. Häyrinen, K. Saranto, and P. Nykänen. Definition, structure, content, use and impacts of electronic health records: A review of the research literature. *International Journal of Medical Informatics*, 77(5):291, 2008.

[22] J. Herrin and B. J. Dempsey. Web-enabled medical databases: a threat to security? *Methods Inf Med*, 39(4-5):298–302, 2000.

[23] A. K. Jha, T. G. Ferris, K. Donelan, C. DesRoches, A. Shields, S. Rosenbaum, and D. Blumenthal. How common are electronic health records in the United States? A summary of the evidence. *Health Affairs*, 25(6):w496–w507, 2006.

[24] B. Kemp and J. Olivan. European data format 'plus' (EDF+), an EDF alike standard format for the exchange of physiological data. *Clinical Neurophysiology*, 114(9):1755 – 1761, 2003.

[25] M. Li, S. Yu, K. Ren, and W. Lou. Securing personal health records in cloud computing: Patient-centric and fine-grained data access control in multi-owner settings. In *Security and Privacy in Communication Networks*, volume 50 of *Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering*, pages 89–106. Springer Berlin Heidelberg, 2010.

[26] B. Lynn. The pairing-based cryptography library. `http://crypto.stanford.edu/pbc/`, June 2011. Version 0.5.12.

[27] D. Meffert. Bilinear pairings in cryptography. Master's thesis, Radboud Universiteit Nijmegen, 2009.

[28] A. Menezes, P. V. Oorschot, and S. Vanstone. *Handbook of Applied Cryptography.* CRC Press, 1996. ISBN 0-849-38523-7.

[29] E. Mykletun, M. Narasimha, and G. Tsudik. Authentication and integrity in outsourced databases. *Trans. Storage*, 2:107–138, May 2006.

[30] PostgreSQL Global Development Group. libpq: PostgreSQL API. `http://www.postgresql.org`, May 2010. Version 8.4.4.

[31] A. Sahai and B. Waters. Fuzzy identity-based encryption. *Advances in Cryptology–EUROCRYPT 2005*, pages 457–473, 2005.

[32] B. Schneier. *Applied cryptography: protocols, algorithms, and source code in C*. Wiley, 1996. ISBN 0-471-11709-9.

[33] A. Shamir. Identity-based cryptosystems and signature schemes. In G. Blakley and D. Chaum, editors, *Advances in Cryptology*, volume 196 of *Lecture Notes in Computer Science*, pages 47–53. Springer Berlin / Heidelberg, 1985.

[34] C. Shannon. Communication theory of secrecy systems. *Bell System Technical Journal*, 28, October 1949.

[35] M. Sweet. Lightweight XML library. `http://www.minixml.org`, May 2009. Version 2.6.

[36] Q. Tang. Type-based proxy re-encryption and its construction. In *Progress in Cryptology - INDOCRYPT 2008, 9th International Conference on Cryptology in India, Kharagpur, India*, volume 5365 of *Lecture Notes in Computer Science*, pages 130–144, Berlin, December 2008. Springer Verlag.

[37] Hackers from abroad obtain data on Washington patients. *The NY Times*, December 8, 2000.

A SQL Queries

A.1 Environments A and B

In environment A, the database is not secured. Thus, the queries are straightforward. In environment B, confidentiality is provided, but this does not require us to query additional fields, as all ciphertexts are stored instead of the plaintexts. The queries are as follows.

Query 1

```
SELECT   *
FROM     patients
WHERE    id = 51 OR id = 99 OR id = 1287
         OR id = 2841 OR id = 4389;
```

Query 2

```
SELECT   p.givenname, p.surname, p.gender, p.birthday,
                   d.diagnosis
FROM     patients p
                   LEFT JOIN sensordata s
                             ON s.patientid = p.id
                   RIGHT JOIN diagnosis d
                             ON d.sensordataid = s.id
WHERE    p.id = 3852
ORDER BY d.sensordataid DESC
LIMIT    1;
```

Query 3

```
SELECT   s.patientid
FROM     sensordata s, diagnosis d
WHERE    d.sensordataid = s.id
         AND d.result = 'Epilepsy';
```

Query 4

```
SELECT   givenname, surname
FROM     patients;
```

A.2 Environments C and D

In environment C, only integrity is provided. In environment D, both confidentiality and integrity are provided. For both environments, we need to query additional fields in order to be able to validate the result obtained from the database. The queries are listed below, as well as a brief description of what fields have to be requested additionally, compared to the queries in environments A and B, and how the query can be verified by the client.

Query 1

```sql
SELECT  *
FROM    patients
WHERE   id = 51 OR id = 99 OR id = 1287
        OR id = 2841 OR id = 4389;
```

For query 1, no additional fields are required. This query is exactly the same as for environments A and B. Since we use the '*' to request all fields, the signed hashes, used to validate the result, are returned as well.

Query 2

```sql
SELECT  p.givenname, p.surname, p.gender, p.birthday,
        d.result, p.givenname_sig, p.surname_sig,
        p.gender_sig, p.birthday_sig, d.result_sig,
        p.id, s.id, d.id, s.patientid, s.patientid_sig,
        d.sensordataid, d.sensordataid_sig, d.doctor,
        d.doctor_sig, s.doctor, s.doctor_sig
FROM    encrypted.patients p
        LEFT JOIN encrypted.sensordata s
                ON s.patientid = p.id
        RIGHT JOIN encrypted.diagnosis d
                ON d.sensordataid = s.id
WHERE   p.id = 3852
ORDER BY d.sensordataid DESC
LIMIT 1;
```

As we need to verify the fields of query 2, we need to request the signatures as well. Furthermore, we need to validate the query itself, since one of the fields `s.patientid` and `d.sensordataid` could have been altered. In order to be able to verify the signatures on these fields, we have to select `s.doctor` and `d.doctor` and their signature. Moreover, we select the primary keys `p.id`, `s.id` and `d.id`, which are used to verify the integrity of the fields (recall that each signature consists of a hash in which the primary key is included).

In order to validate this query, we need to verify that `s.patientid` equals `p.id` and that the signature on `s.patientid` is valid. Similarly, we need to check that `d.sensordataid` equals `s.id` and that `d.sensordataid` has a valid signature. If these conditions are met, we need to verify the signature of `p.name`, `p.surname`, `p.gender` and `p.dateofbirth` and then we know the result is valid. Note that we do not have to check the primary keys (`p.id` and `s.id`) themselves, as they are implicitly checked for validity by checking the signature of the other fields, such as `p.name`. Recall that the signature of `p.name` includes the primary key, which ties `p.name` to a certain primary key, thus rendering any modification to `p.id` useless.

Query 3

```
SELECT  s.patientid, s.patientid_sig, d.sensordataid,
        d.sensordataid_sig, d.result, d.result_sig,
        s.id
FROM    sensordata s, diagnosis d
WHERE   d.sensordataid = s.id
        AND d.result = 'Epilepsy';
```

For query 3, we need to verify the signatures on `s.patientid` and `d.diagnosis`, and verify that `d.sensordataid` equals `s.id` and that `s.patientid` equals `p.id`.

Query 4

```
SELECT  id, givenname, surname, givenname_sig,
        surname_sig
FROM    patients;
```

Finally, to verify the result of query 4, we need to request the signatures on the hashes of the `givenname` and `surname` fields. Additionally, the field `id` has to be requested in order to be able to calculate the hash of the the givenname and surname.

B Comparing Elliptic Curves

During the implementation of our prototype, we discovered that chosing a type of curve with fast pairings is not a guarantee for fast algorithms. The *mapping* operation, where an arbitrary string is mapped to an element of the group, is very slow for certain types of curves. Figure B.1 shows the costs for computing a pairing and mapping for each type of curve. We chose the standard curves that come with the PBC library [26] and have a discrete log security of at least 1024 bits, which is considered good enough [26].

In Type-Based Proxy Re-Encryption (TBPRE, Section 4.2.7) and the Bilinear Aggregate Signature Scheme (BGLS, Section 4.3.4), not only the pairing operation, but also the mapping operation is used frequently.

First of all, we identify the number of pairings and mappings for each algorithm in TBPRE. We only consider the H_1 hashing function in account, as it maps an arbitrary string to a group element, whereas the H_2 hashing function maps an arbitrary string to a fixed-size string. The results below are the number of pairings and mappings for the operation on n items (such as performing n encryptions, or n decryptions).

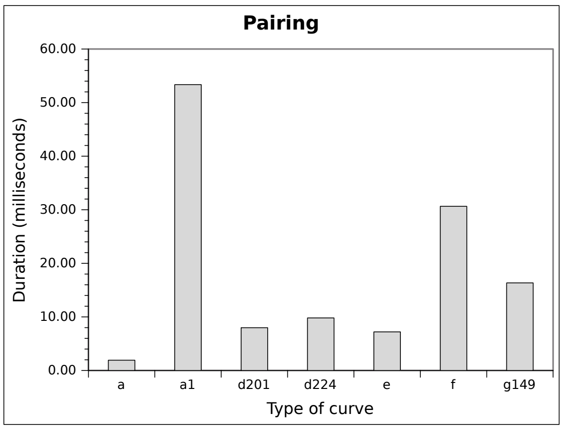
Algorithm	**Pairings**	**Mappings**
Encrypt_1	n	n
Encrypt_2	n	0
Decrypt_1	n	n
Decrypt_2	n	0
Pextract	0	n
Preenc	n	0

In Figures B.2, B.4 and B.3 the computational cost per type of curve for each algorithm of TBPRE is shown. Overall, a type a curve is the fastest for the most important operations (encrypt, decrypt, re-encrypt) and performs well on the other operations. The differences between curves can not entirely be explained by only looking at pairings and mappings. Other operations (multiplication, division, etc) are also of influence.

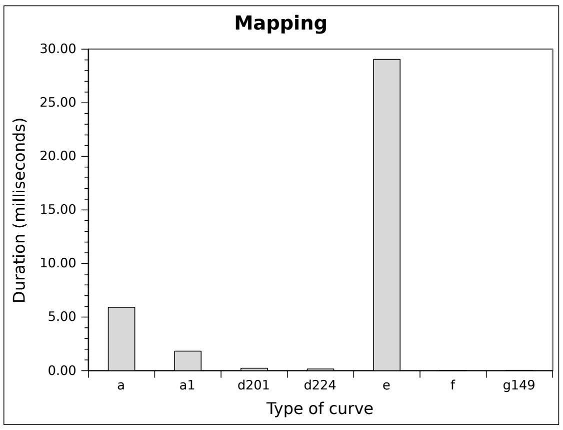
For the algorithms in BGLS, the number of pairings and mappings for n operations are below.

Algorithm	**Pairings**	**Mappings**
Sign	0	n
Verify	$2n$	n
Aggregate	0	0
Aggregate verification	$n+1$	n

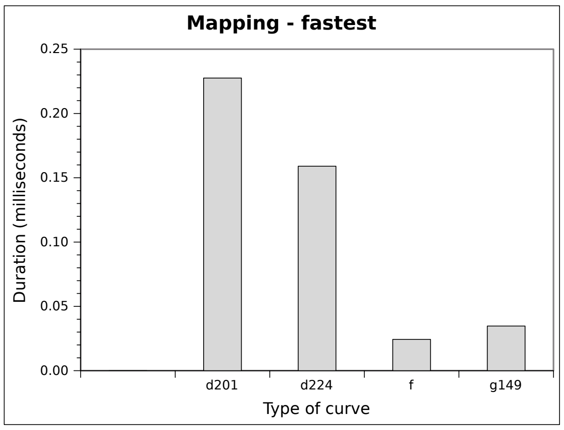
For the aggregate verification algorithm, the difference between pairings and mappings is minimal. Figure B.5 shows the performance of the algorithms on different curves. Even though the mapping operation is slow in a type a curve, for the verification algorithms (both regular and aggregated) it is the fastest, by a small margin. This can be explained by faster pairings and possible differences in other mathematical operations, such as multiplication. For the signing algorithm, type *a* is one of the slowest. However, since data is signed only once and verified multiple times, we consider verification to be more important than signing. Regarding aggregation, the type *a* curve is over twice as slow than the fastest curve (*g149*), but the execution time of the aggregation algorithm is relatively small compared to the execution time of the aggregate verification algorithm.



(a) Average time for a single pairing



(b) Average time for a mapping



(c) Average time for a mapping of five fastest types of curves

Figure B.1: A comparison of primitive operations between types of curves.

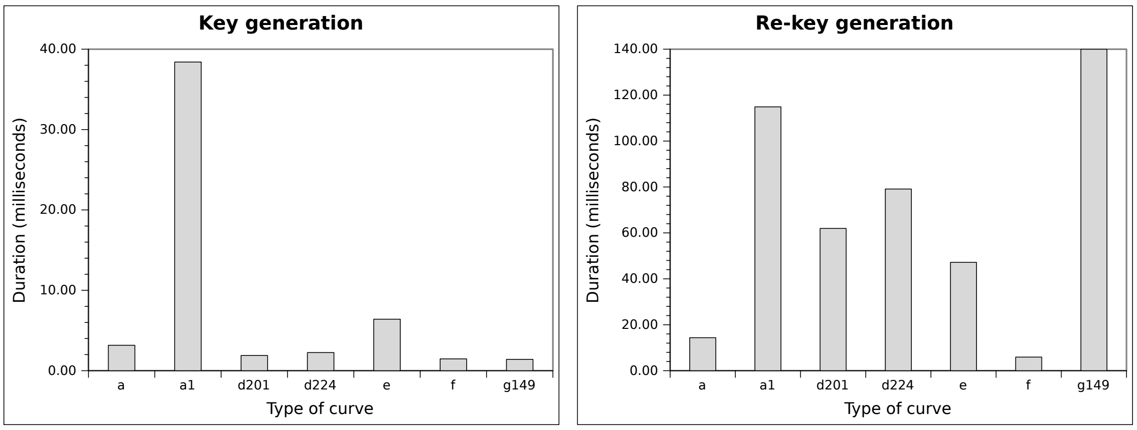


Figure B.2: Comparing curves for the TBPRE key generation algorithms.

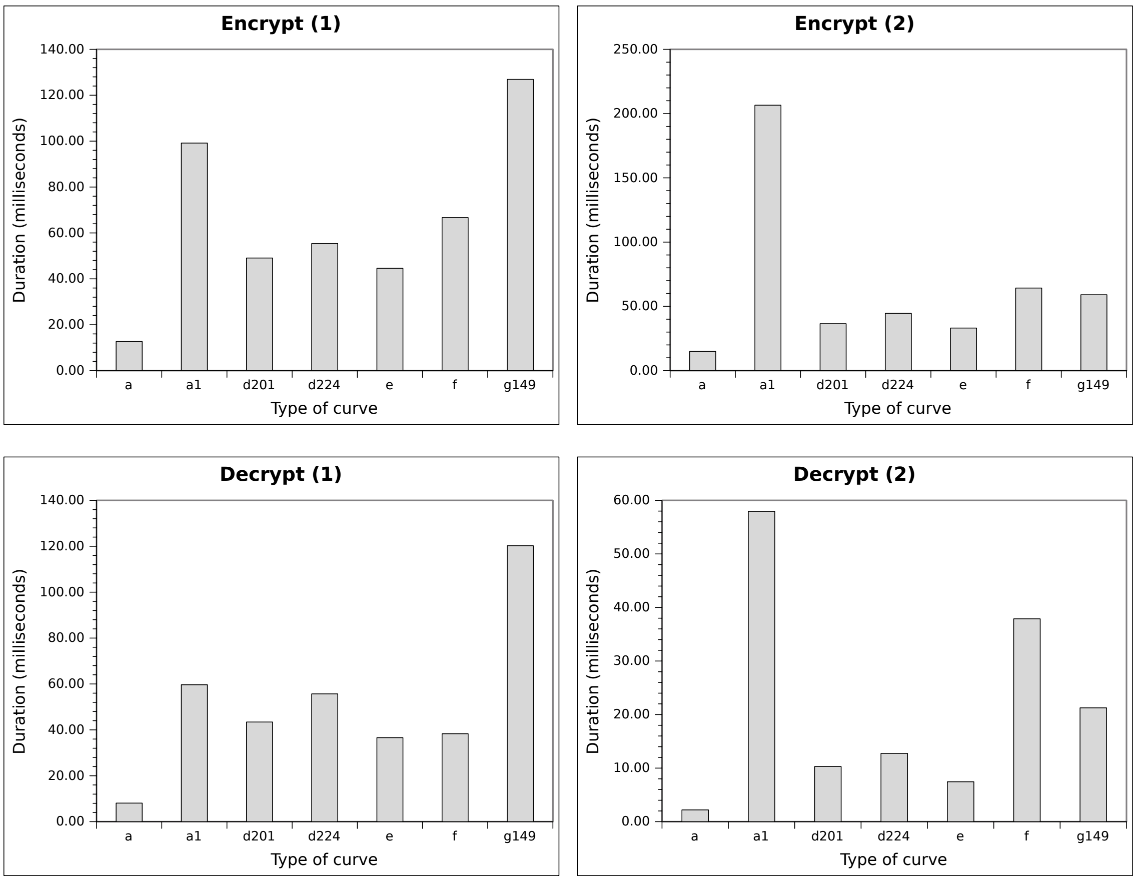


Figure B.3: Comparing curves for the TBPRE encryption and decryption algorithms. Each figure refers to the corresponding algorithm. For example, Encrypt (1) refers to the Encrypt_1 algorithm.

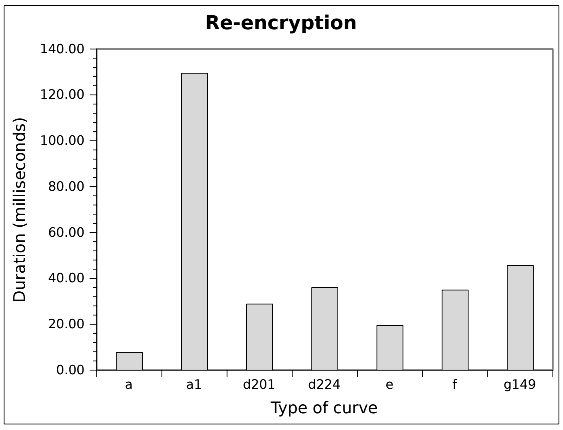


Figure B.4: Comparing curves for the TBPRE re-encryption algorithm, which is used by the proxy.

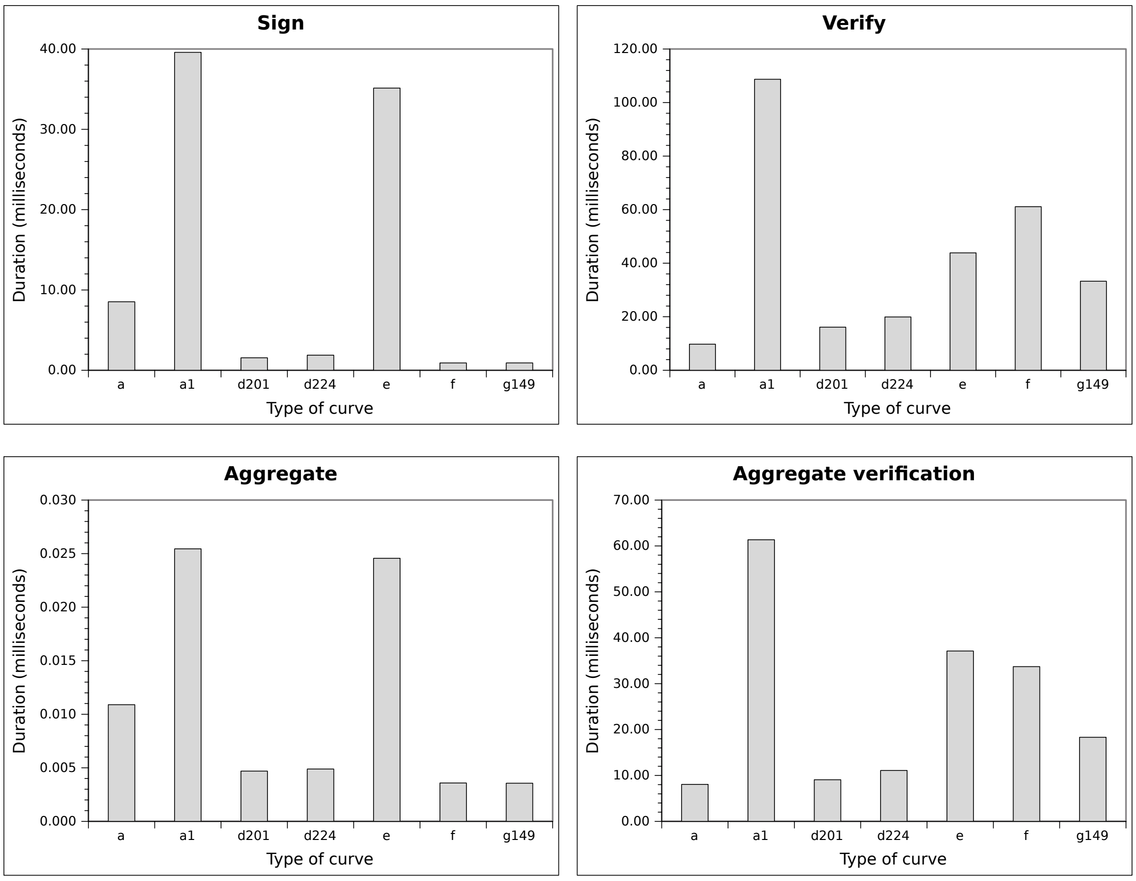


Figure B.5: Comparing curves for the BLGS algorithms.