Encryption in Delocalized Access Systems

Examensarbete utfört i Informationsteknologi
vid Linköpings tekniska högskola
av

Henrik Ahlström
Karl-Johan Skoglund

LITH-ISY-EX--07/4046--SE
Linköping 2007

Encryption in Delocalized Access Systems

Examensarbete utfört i Informationsteknologi
vid Linköpings tekniska högskola
av

Henrik Ahlström
Karl-Johan Skoglund

LITH-ISY-EX--07/4046--SE
Linköping 2007

Handledare: **Kent Axelsson**
ITN, Linköpings universitet

Viiveke Fåk
ISY, Linköpings universitet

Johan Andersson
Combitech AB

Joakim Pettersson
Combitech AB

Examinator: **Viiveke Fåk**
ISY, Linköpings universitet

Linköping, 6 December, 2007

Presentationsdatum
2007-11-23

Publiceringsdatum (elektronisk version)
2007-12-06

Institution och avdelning
Institutionen för systemteknik

Department of Electrical Engineering

Linköpings universitet

Språk

___ Svenska
_X_ Annat (ange nedan)

Engelska

Antal sidor
103

Typ av publikation

____Licentiatavhandling
_X_Examensarbete
____C-uppsats
____D-uppsats
____Rapport
____Annat (ange nedan)

ISBN --

ISRN LITH-ISY-EX--07/4046--SE

Serietitel (licentiatavhandling)

Serienummer/ISSN (licentiatavhandling)

URL för elektronisk version
http://www.ep.liu.se

Publikationens titel
Encryption in delocalized access systems

Författare
Karl-Johan Skoglund, Henrik Ahlström

Sammanfattning

The recent increase in performance of embedded processors has enabled the use of computationally heavy asymmetric cryptography in small and power efficient embedded systems. The goal of this thesis is to analyze whether it is possible to use this type of cryptography to enhance the security in access systems.

This report contains a literature study of the complications related to access systems and their functionality. Also a basic introduction to cryptography is included.

Several cryptographic algorithms were implemented using the public library LibTomCrypt and benchmarked on an ARM7-processor platform. The asymmetric coding schemes were ECC and RSA. The tested symmetric algorithms included AES, 3DES and Twofish among others. The benchmark considered both codesize and speed of the algorithms.

The two asymmetric algorithms, ECC and RSA, are possible to be used in an ARM7 based access system. Although, both technologies can be configured to finish the calculations within a reasonable time-frame of 10 Sec, ECC archives a higher security level for the same execution time. Therefore, an implementation of ECC would be preferable since it is faster and requires less resources. Some further suggestions of improvements to the implementation is discussed in the final chapters.

Nyckelord
Access systems, Cryptography, Public Key, Embedded system, Large integer arithmetic.

Abstract

The recent increase in performance of embedded processors has enabled the use of computationally heavy asymmetric cryptography in small and power efficient embedded systems. The goal of this thesis is to analyze whether it is possible to use this type of cryptography to enhance the security in access systems.

This report contains a literature study of the complications related to access systems and their functionality. Also a basic introduction to cryptography is included.

Several cryptographic algorithms were implemented using the public library LibTomCrypt and benchmarked on an ARM7-processor platform. The asymmetric coding schemes were ECC and RSA. The tested symmetric algorithms included AES, 3DES and Twofish among others. The benchmark considered both codesize and speed of the algorithms.

The two asymmetric algorithms, ECC and RSA, are possible to be used in an ARM7 based access system. Although, both technologies can be configured to finish the calculations within a reasonable time-frame of 10 Sec, ECC archives a higher security level for the same execution time. Therefore, an implementation of ECC would be preferable since it is faster and requires less resources. Some further suggestions of improvements to the implementation is discussed in the final chapters.

Acknowledgements

First of all we would like to thank Combitech AB for supporting our project idea and providing the resources for our project.

We would also like to thank Viiveke Fåk, our examiner and supervisor at ISY, for always bringing brightness, simplicity and correctness into this thesis project. We are also grateful for the support and patience when answering our novice questions regarding cryptography at the beginning of this project.

Further, thanks to supervisor Kent Axelsson at ITN for showing great interest in this thesis work.

We greatly appriciate the time and effort our supervisors at Combitech, Johan Andersson and Joakim Petterson, has put into this project guiding us with the organizational structure needed for a project of this kind and solving complexities when we got stuck.

We would also like to thank our friends and family for support and patience with our extra long workdays and occupied minds.

Contents

Glossary

1-wire A device communications bus system designed by Dallas Semiconductor.

3DES Triple Data Encryption Standard, see DES.

AES Advanced Encryption Standard.

ANSI American Standard Institute.

ARM7 A series of the ARM processor architecture.

CA Certificate authority.

CBC Cipher-block chaining mode.

CTR Counter mode.

DES Data Encryption Standard.

DLP The discrete logarithm problem.

DSA Digital Signature Algorithm.

DSP Digital signal processor.

ECB Electronic codebook mode.

ECDLP The elliptic curve discrete logarithm problem.

ECDSA Elliptic curve digital signature algorithm.

EFSL Embedded Filesystems Library.

FIPS Federal Information Processing Standard.

FLASH Non-volatile programmable memory.

FPGA Field-programmable gate array.

GCC GNU Compiler Collection.

IEEE Institute of Electrical and Electronics Engineers.

IV Initialization vector.

LTC LibTomCrypt library.

LTM LibTomMath library.

MAC Message authentication code.

MPI Multiple Precision Integer.

NIST National Institute of Standards and Technology.

Nonce Number used once.

PKI Public key infrastructure.

PRNG Pseudorandom number generator.

RAM Random access memory.

RISC Reduced instruction set computer.

RSA An asymmetric cryptography algorithm.

RTC Real Time Clock.

SD Secure Digital Memorycard.

SHA Secure Hash Algorithm.

SPI Serial Peripheral Interface Bus.

TFM TomsFastMath library.

TWI Two Wire Interface.

Twofish Twofish cipher.

VHDL Design-entry language for FPGA's.

CHAPTER 1

Introduction

This chapter gives an introduction to this master's thesis, "Encryption in delocalized access systems". A background is given and the application area, objectives, and limitations are described.

1.1 Background

When designing and implementing a multi-user access system there are several factors to consider, of which *key management* is one of the most important and most difficult to solve. Key management is the problem of making sure that each user has the correct key with the proper security level, at the right time. The access system has to distribute and keep track of the keys, making sure that no keys are lost or compromised.

The simplest and most common access system is the traditional mechanical lock cylinder, which is a simple and very reliable system. However, key management in a mechanical multi-user system is a momentous task and therefore several electronic solutions exist solving this problem. Even though the mechanical lock is often kept as the fallback system in case of a power-failure.

These electronic solutions are mostly based on *online* networked lock-terminals communicating with a centrally managed key server. The decision of who to grant access is made by the server.

This is practical and secure when all lock units are gathered in the same geographical area, such as a building or within company grounds.

For systems distributed over a larger geographic area, the options are fewer. Some solutions use wireless online links in the same way as a local system, others use the telephone network. However, wireless modems are power-consuming and expensive and phone lines are not available everywhere. So most systems of this type are based on the mechanical key.

The mechanical lock system lacks several important features compared to its electronic counterpart. The system is not suited for a multi-user environment since all authorized users have a copy of the same key (from the lockunit's viewpoint). Lost keys cannot be blocked and have unlimited lifespan. The keys can also easily be copied by corrupt users with physical access to the key. Or even within visual range of the key, according to a recent thesis work in Linköping University[1], which showed that it is possible to reproduce a key from a single photo.

1.2 Application area

As previously mentioned the mechanical system is still the type of access system which is the most frequently used, where the lockunits are geographically distributed.

Some companies deal with hundreds of keys daily, which results in a highly complex key management solution requiring large resources, both in additional work done by employees and in maintenance costs. This also often leads to security flaws due to the complexity of the system, mostly related to the human factor.

The system has to determine which employee to give which key, and often multiple employees have to access the same location. It also has to detect lost keys in the system and issue replacement cylinders whenever needed. If a key is knowingly compromised this results in a time consuming and expensive task changing the cylinder in the lock and distributing new keys to each user.

[1]The master's thesis done by Linus Fredriksson and Martin Gyllensten [1].

When there is a security breach due to a compromised key in this mechanical system there is no trace to which employee was responsible, since everyone has the same key and the keys can easily be copied.

Therefore the focus of this thesis work will be on the problems associated with access systems with geographically distributed objects and locations.

An example A company within the field of work of security. The company provides security for several hundreds of companies, and employs a number of guards. Each employee on guard detail will have to carry keys to all the companies which are to be visited during the shift. This gives an enormous administration of keys to ensure that each of the guards has the correct set of keys to all locations for each shift. The guard does not bring all keys for all companies every day since most of the companies are to be visited a couple of times a week.

This is positive for the security because if the guard is attacked there are a limited number of keys which are lost. If an alarm is set off in a company, which is excluded from the regular route of the guard. It is a time-consuming procedure to first return to the office to acquire the specific key and proceed to the corresponding company.

1.3 Objectives

It should be noted that even though a concept solution was found, it cannot be presented in the public version of this report due to a pending patent application. Therefore the objectives are separated into project objective and public thesis report objective.

Project objective The main objective with this thesis work is to construct a prototype system representing a concept which simplifies the task of key management and enables multi-user functionality in delocalized systems. It should act as a replacement to the mechanical lock, and has to retain as many of the desired properties found in the mechanical lock as possible.

The new solution has to comply to the demands which the mechanical system fulfill in the market today, which is a standalone lock unit not depending on additional network or power requirements. Also the new solution has to be a true multi-user system and provide most of the authorization, logging and key-integrity functionality which is common in its online counterpart.

Public thesis report objective The main objective of the report is to implement and benchmark a variety of asymmetric- and symmetric-cryptography algorithms on an embedded platform. Also an overview of the complications associated with access systems will be provided.

1.4 Problem description

By analyzing the objective, a list of questions are formulated into problems to be solved. These are the major tasks to be solved:

Security Is it possible to construct an access system with higher security by implementing more advanced cryptography? Could this system simplify the problem of key distribution in an access system?

Implementation Is it possible to implement the computationally advanced cryptography on an embedded system with small resources? Can this be done power-efficiently and at a reasonable cost? Could the calculations required be performed within a reasonable time?

1.5 Limitations

This report assumes the reader has a fundamental background in electronic design and construction of embedded systems, however no prior knowledge of cryptography is required. Further limitations of the project are divided into resource- and project-limitations.

Limitations of resources There are two project members. The project ranges over a period of 20 weeks with a contracted deadline on November 30 in 2007. The budget is approximately 10.000 SEK which is invested by Combitech AB.

The project has four mentors, two provided by Combitech and two from the University of Linköping. Combitech AB provides a workplace equipped with computers, stationery and lab equipment.

Limitations of the project A prototype of a concept using modern cryptography is to be developed within this project. The prototype is only for testing purposes and therefore a fully functional prototype is not included in this thesis work.

A secure implementation of the cryptography requires a cryptographically strong random number generator, however implementing this is not within the realm of this thesis.

The user interface is only to be developed for testing of the system, and will only perform a limited number of tasks.

1.6 Method

The work was done in several steps, beginning with theoretical studies of cryptographic algorithms. This was done in parallel with a market research of access systems, current solutions and requirements.

The next step was to design a concept solution, which was to be implemented and also to find suitable cryptographic algorithms usable in the implementation of the concept.

Thereafter a hardware implementation of the algorithms was done to measure and compare their performance, followed by a complete implementation of the concept on the platform.

The last step was to analyze the performance of the implementation and to summarize the conclusions, including suggested improvements which could be made to the system.

CHAPTER 2

Access systems

This chapter presents the description of general access systems and their functions. Furthermore are current access system solutions presented and how to improve their functionality. The suggestive improvements presented as *concept requirements* are based on the market investigation and personal experiences. The aim of this chapter is to present information of the requirements of a new system.

2.1 The principle of an access system

When discussing access systems, it is important to clarify their intended function and purpose. An access system is a system meant to protect locations within a domain against unauthorized access, while still granting access to authorized users. Sometimes, more than one access system is used in the domain, where one system is the primary with multiuser capability. The other system is a reliable backup-system which can be used by administrators if the first system fails.

2.1.1 Basic functionality

The primary function of an access system is, as mentioned, to keep unauthorized people out and to grant access to valid users. To make difference if the user is valid or not, the system must use some form of authentication. The authentication is done by analyzing information provided by the user. This in-

formation can be divided into three different categories, *identity codes*, which are presented as circles in *Figure* 2.1. These are often used in different combinations and the four most common are described below. Although the use of passport and driving licenses are common, they are not considered within this description of general access systems.

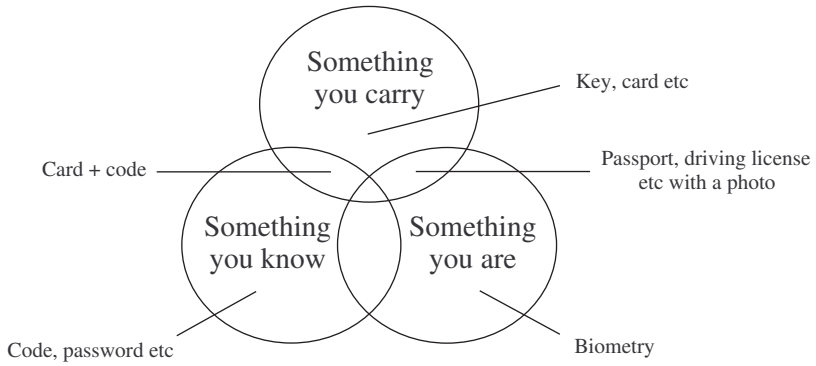

Figure 2.1: *The figure is describing the three different identity methods used in access systems. More information about these methods can be found in a Securitas document [2]*

Memory code In this case the user memorizes the code and only the code is required to get access. This type of identity control is weak and is only used in areas where there are many users who need to be able to delegate access to others, such as an entrance to an apartment building. It is cheap to install and maintain. However, the security is very low. The code can easily be copied, and the system has no log of whom has passed.

Carried code In this system the user brings a physical token, often a key card, although other solutions exists. The token contains information loaded by the access system to grant access. These kind of systems are slightly more secure than the previous alternative, although it is not difficult to copy the data from a token. The method of using just a token is common where a simple flexible multi-user system is required.

Usually the lock unit is connected to a central server which makes it possible to block a specific compromised token. The connection is either cable or wireless. This means the cost of setting up the infrastructure is high but the maintenance costs are low.

Carried code with memory code This method is a combination of the two previous methods. The token's information includes the memory code which the user has to provide each time an access is requested. This provides some more security, the ability to clone a token still exists but the adversary has to acquire the code. The code stored within the information is sometimes encrypted. When the lock unit is online, the code can be stored on the server instead. The advantage of this system compared to just using a card or code is that the code is easy to change and the token easy to block.

Biometric code The most common form of biometric codes is the visual inspection of the photograph on an identification card, such as a driving license. However the use of automated biometric readers is expanding as the technology advances and becomes more available. The basic different biometrics used within access systems are eyes, hands, voice and photo identifications. The most common automated biometric code is the fingerprint identification.

The disadvantage is that this biometric data is linked to a person. If it is stolen the persons biometric code is stolen for lifetime, it cannot be replaced or blocked.

The central area in the figure 2.1, where all three codes are combined, is considered to be the most secure combination of the different identity codes. However the implementation of such a system is unusual and often considered to be time-inefficient and expensive.

2.1.2 Additional functions

Some additional functions which are desirable in an access system include:

Timelock Limiting the validity period of the key. A user can be restricted to certain hours of the day, or the key lifespan could be limited.

Logging An important feature in access systems, which has authentication, is to keep a log-file of all access events by all users. This provides tracking functionality if a key and/or a user is compromised.

Authorization level This enables the system to have a hierarchy among the users. A certain security clearance can be required to access a certain location. This can also be implemented as a system where users can be mapped to which locations they have access to.

2.1.3 Security

The security is a measurement of how difficult it is to break into the object which the system protects. Due to the complexity of all the possible factors affecting the total security, it is only feasible to estimate the security of the specific access system.

The goal for an attacker of an access system is to get access. This could be done by going around the actual access system and through some other weaker point of the total defense, e.g. breaking a window or subverting a valid user. However, if only considering the access system's security it is still difficult to measure the security. Security systems are often broken in ways the system designer never could imagine.

A way of modeling the threats is to use an *attack tree*[1]. These attack trees are however very complex and difficult to apply on large systems such as access systems. The method is to first detect all possible attack goals, of which each goal forms an attack tree. All these trees might share several nodes or subtrees. After constructing these trees all the possible attacks of getting the goal is added to the tree. This is done in several steps and by several different persons to cover as many of the possible ways as possible. Regarding the complexity this is recommended to take months which is why it is not included in this thesis.

[1]The attack tree is mentioned by Bruce Schneier in his website [3].

2.2 Available solutions

Available solutions based on the four methods used for identification are presented in this chapter. The solutions can be categorized in three different groups of access systems; *unintelligent off-line systems*, *intelligent off-line systems* and *on-line systems*.

2.2.1 Unintelligent off-line systems

These systems are mostly based on the ordinary lock cylinder system widely used within many different areas. Another access system within this category is the key pad.

Mechanical lock cylinder The mechanical lock cylinder is as mentioned frequently used in geographical distributed areas. The primary advantage of such systems is the dependability. It is robust and well tested and it works in all kinds of weather in contrast to other systems. Other systems often need a backup system and they often require electricity. The mechanical lock cylinder is also very easy to use, since it is the most common system, generally everybody knows how to use it.

Disadvantages of this system is for example the maintenance of the system. For example: a broken key can be replaced with a new, however, this is often a time consuming procedure. A stolen or lost key creates greater problems, since each corresponding lock cylinder must be replaced. If this happens in a multi-user system, the expense and time loss can be very large. This makes the system usually inappropriate for a multi-user environment, since all users carry a copy of the same or a similar key.

A cylinder only relies on the key, it does not consider the user, which is another disadvantage of the system. It cannot really identify the user and it does not detect a copy. It is also impossible to block a certain key from the system without replacing all the cylinders. The system is however, suitable for simple and low security solutions such as an entrance door in a multi-apartment building.

Keypad The keypad is another unintelligent off-line system which is quite common. It has the same verification problem though it only controls the dialed code. This kind of system is common in staircase entrances. It is used where the security issue is not that important. It is a simple system often based on just one code although there are systems based on several codes. One code is for the lessees and another for the mail services. It is easy to change the code whenever wanted, though one must be in place for the operation.

2.2.2 Intelligent off-line systems

This kind of systems are based on the same functionality as an ordinary mechanical lock although it is common that this kind of systems have additional functions. These kinds of systems are based on embedded systems, and all verification of the user is done locally, often with some external additional devices. The system's external device is often a key set or a finger print reader.

It is possible to place a system like this in a location not supplied by power or a phone line. The power needed to make the operation could be based on battery or a scenario where each user brings some kind of device supplying the lock system with power during the operation.

2.2.3 On-line systems

These systems are connected to some central unit of intelligence. The most common use of such a system is the use of the entry cards. This system is more suited for multi-user environments than the other solutions. There are several solutions which are based on the same concept, e.g. radio-frequency identification (RFID) tags, although the entry cards is the most common and thereby used for this description. The reader is by wire or wireless connection connected to a central unit, often a computer, which controls the information of the card. The computer keeps a register within a database of authorized users of the system. A broken or lost card is easily blocked and removed from the register. This is a quick and inexpensive operation and some systems do not require the administrator to be in place for making the operation. Adding a new entry card or replace a lost card, is as easy as blocking one.

2.3 Brief conclusion

This study of access systems has involved many different areas such as functionality of existing access control systems, possible solutions and the requirements of a future product.

The study showed there are some flaws in the currently available access systems in the application area considered. Electronic solutions are power consuming and expensive, so the most commonly system used on the market today is still the mechanical system. It should be noted that the average person probably would consider the mechanical system to be more safe and reliable than an electronic solution, since it has been used for a long time.

2.4 Concept requirements

Designing a concept for use in an offline geographically distributed system places several challenges on the designer. Since this concept aims to be a replacement to the mechanical system, the concept should keep as many of the advantageous characteristics of the mechanical lock system as possible. These were found to be:

- **Reliability** This is a great advantage of the mechanical key. No electricity is required.
- **Robust** Works in all kinds of weather. The keys and the lock are made of steel and are tough. If the lock is frozen, apply heat and it works again.
- **Trusted technology** The general opinion is that it is secure. Keep track of the keys and the system is supposedly safe. This is of course incorrect, but it is the general belief.
- **Offline** No need for the lock to consult a user database.
- **Geographically independent** A lock can be placed anywhere.
- **User friendly** Easy to use, everyone knows how it works.

After some considerations it was determined which additional characteristics would be desirable in the new concept. In this thesis focus is on the security, therefore mostly security- and feasibility-related requirements will be considered[2]. The required improvements are found to be:

- **Copy proofing** There is no reliable integrity check of the key, a mechanical key can therefore be copied. All that is required is a photo of the key or an imprint of the pattern in a soft material.
- **Tamperproofing** An experienced lockpick, can pick almost any lock on the market.
- **Authentication** The mechanical system has no authentication of the individual holding the key.
- **Logging** There is no logging of who or when anyone has been allowed access. No one knows whose key was used when a breach happened.
- **Multi-user** A mechanical lock system with many users has many keys. Since there is no authentication, it is not possible to block a single key if it gets lost.
- **Limiting lifespan** A key will be valid until the lock (cylinder) is changed. Which makes lost keys an expensive problem.
- **Managing lost keys** As described above, there is no way to block a key. Losing a key results in changing the cylinder and redistributing keys to all the users.

2.5 Concept solution

Although a concept solution is developed within this thesis, it is classified due to the pending patent. This is as mentioned a public report thus the section covering the concept solution is omitted.

[2]Cost and implementation requirements are closely linked with feasibility requirements.

CHAPTER 3

Cryptography

Cryptography is a large subject which can be confusing at best sometimes. The term cryptography (or cryptology derived from Greek *kryptós* "hidden" and *gráfo* "write") is the study of message secrecy. The opposite is cryptanalysis which is the study of methods of how to reverse the encrypted message. This chapter aims to give some background on the encryption techniques and application areas considered during the design process of the system.

3.1 Basic cryptography

There is a tradition within the area of cryptography of using the names *Alice*, *Bob* and *Eve* to represent the different roles played by the communicating devices on a communication channel. By definition *Alice* sends messages to *Bob* and *Eve* is assumed to be eavesdropping on all messages sent on the communication channel.

3.1.1 Encryption

Encryption is used to communicate securely over an insecure communication channel. Consider *Alice* communicating with *Bob*. Any message from *Alice* to *Bob* is also received by *Eve*. To prevent *Eve* from understanding the message an encryption function $E(K_{enc}, m)$ is used to transform the so called *Plaintext*, m, into the unreadable *Ciphertext*, c, where K_{enc} represents the encryption key which is to be known only by the authorized communicants and not by *Eve*. In

order for *Bob* to be able to read the message, a decryption function $D(K_{enc}, c)$ is used to make the reverse transformation from *Ciphertext* into *Plaintext*, see figure 3.1.

Both these transformations require a cipher which is an algorithm used for performing encryption and decryption, see section 3.2. As shown in the formula, *Bob* needs to know two things to decrypt the *Ciphertext*; the algorithm D and key K_{enc}. The key is as mentioned to be kept secret although the algorithm can and should be public.

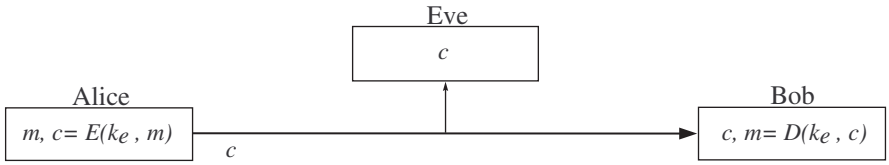

Figure 3.1: *The figure describes the relationship between* Alice, Bob *and* Eve *and the generic settings for encryption. These roles, representing the different parts affecting the communication, are common within the area of cryptography. More figures and description about the different roles can be found in "Practical Cryptography" by Bruce Schneier and Niels Ferguson [4].*

Algorithms are to be published for public testing. If the algorithm is kept secret, the chance to discover and provide a solution to a possible bug, which makes the algorithm weak, is smaller. The first person to find the exploit will likely be the attacker who was meant to be kept out.

3.1.2 Data integrity

Special care has to be taken to ensure that the message *Bob* receives is the same message which *Alice* sent. Since *Eve* is listening on the communication channel, she is also assumed to be able to change the messages sent on the channel. Encryption only prevents *Eve* from reading messages, not from changing them. Eve can still remove/append data to the message.

To combat this problem an algorithm which calculates a fingerprint of a message is used. This algorithm is called a *Hash* algorithm and computes a fixed sized so called *hash-value* or *hash-digest* from a message of arbitrary length. The

hash function is ideally a seemingly random mapping function from an input of any length to a fixed output of n bits, which can be explained as a fingerprint generator for the message. It is a one-way function which maps the message to the digest in such a way that it is impossible to find the message given only the hash-digest. Also a hash function should make it infeasible to find a message collision, which is finding more than one message which corresponds to the same hash-digest.

Applying this to the previous example; *Alice* computes the hash-digest of her message, and sends it along with the encrypted message. *Bob* decrypts the message, calculates the resulting hash-digest and compares it to the hash-digest which *Alice* sent along with the message.

3.1.3 Authentication

Using hash functions in communication enables the recipient to verify the integrity of the message against the hash-digest sent along with the message. However, how can *Bob* be sure the message really is from *Alice*? *Eve* could have changed the message and recalculated a new hash-digest.

A solution is to use *message authentication codes*, MAC, which are basically a hash function with an authentication key, K_{auth}. The fixed length MAC-digest is calculated using the MAC function $h(K_{auth}, m)$ and is sent together with the message.

When *Alice* wants to send a message she computes the MAC, $a = h(K_{auth}, m)$ and sends the complete message as $(m \| a)$. When *Bob* receives the message $(m_{rcv} \| a_{rcv})$ he calculates his own MAC $a_{bob} = h(K_{auth}, m_{rcv})$ and verifies $a_{bob} = a_{rcv}$. If the codes are different he discards the message. When *Bob* does this he verifies the integrity and the authenticity of the message.

Since K_{auth} is a shared secret, *Bob* can verify the authenticity of the message. This means he can verify the sender really is *Alice*, since only she has the other key.

3.1.4 Asymmetric encryption

Previous sections have described symmetric encryption, where *Alice* and *Bob* share the same secret key, K_{enc}. However, they can not send the key over the communication channel. Since *Eve* is listening in, they have to meet in person to synchronize keys. Keys have a limited lifespan and *Alice* may have many people to communicate with, so exchanging keys is a tedious task.

A solution to the problem of key distribution is asymmetric cryptography, commonly known as Public-Key cryptography[1]. In asymmetric encryption both *Alice* and *Bob* have two paired keys, a public key, P, with a corresponding secret or private key, S. Both publish their public keys somewhere for everyone to see, while they keep the private key secret. The encryption technique is basically a one-way function, so a message encrypted with a public key can only be decrypted with the corresponding private key. This technique simplifies the problem of key distribution somewhat, since there is only one key to distribute and it can be publicly published.

When *Alice* wants to send *Bob* a message she finds his public key, P_{Bob}, which is publicly available. She uses the key to encrypt her message into $c = E(P_{Bob}, m)$, and sends c to *Bob*. When *Bob* receives the message he decrypts the message using his private key into $m = D(S_{Bob}, c)$.

Asymmetric cryptography algorithms are based on complicated mathematical problems, this is what gives them the strength. However, it also makes them computationally slow and memory consuming. Therefore for most applications it is inefficient to encrypt the whole message using asymmetric encryption. A common solution is to perform a symmetric encryption of the message, m, with a random key, K_{sym}, into $c_m = E_{sym}(K_{sym}, m)$. Then encrypt K_{sym} using asymmetric encryption $c_{key} = E_{asym}(P, K_{sym})$, giving the encrypted message $(c_m \| c_{key})$ to transmit. To decrypt this message the procedure is done in the reversed order although the public key is replaced with the private key.

[1]The name Asymmetric cryptography is chosen in this report to minimize the confusion with the actual public key.

3.1.5 Digital signatures

Digital signatures is the way to check data integrity with asymmetric cryptography. It works in a similar way as *message authentication codes*, MAC. The signing process uses a hash function producing a fixed sized hash-digest. The signing function encrypts the hash-digest using the private key into a signature and the signature can be verified by anyone with the senders public key.

When *Alice* signs a message for *Bob*, she uses an appropriate hash function to generate the hash-digest, $h_d = HASH(m)$. She then signs h_d using her private key, S_{Alice}, giving $s = \sigma(S_{Alice}, h_d)$ and transmits the signature, s, together with the message.

Bob can then verify the signature by computing his own hash-digest of the received message, $h_{d,comp} = HASH(m_{rcv})$. The result should be the same as returned from the signature verification algorithm, $h_{d,rcv} = v(P_{Alice}, s_{rcv})$. If $h_{d,comp}$ equals $h_{d,rcv}$ the message integrity is verified, otherwise it has been tampered with.

Since the message is signed with a private key and a message is encrypted with a public key the same key-pair should never be used for both applications. Each user has to have at least two separate key-pairs, one for encrypting messages and one for signing them.

3.1.6 Public key infrastructure

The problem with asymmetric keys is authentication. How can *Alice* find *Bob*'s public key when she wants to send him a message? *Alice* may never have met *Bob*, but she wants to send him a secure message. How can she be sure that the public key she finds really belongs to *Bob* and not *Eve* posing as *Bob*?

They have to rely on a trusted third party, which both trust, to supply them with the correct keys. The trusted third party is called a *certificate authority*, CA, and maintains a *public key infrastructure*, PKI. The CA collects user information and the public key from a user it recognizes into a file which is signed by the CA. This signed file is called a *certificate*.

Certificates state that the CA recognizes the user and links the user to the pub-

lic key presented in the certificate. Anyone who trusts the CA can verify the signature on the certificate and use the public key to send a secure message to the user specified.

Often there is a hierarchy with multiple levels of CA's signing each others certificates, thus delegating authority. The top CA publishes a root certificate which contains the CA's public key, and is signed by the CA's private key.

Assuming both *Alice* and *Bob* have setup their keys with the CA. For *Alice* to send *Bob* a message she can find *Bob*'s certificate in the PKI database. She can verify the integrity and authenticity of his certificate, using the CA's public key. She encrypts her message with *Bob*'s public key, found in his certificate and appends her signature to the data. When *Bob* receives the data he can decrypt the message using his private key. He can verify *Alice*'s signature using *Alice*'s certificate found in the PKI database.

The PKI solves the problem of key distribution since every user only has to update a copy of the certificates of participating users and it seems like a perfect solution.

However, there is the issue of what happens when a user's, or worse the CA's private key is compromised. There are no obvious ways of revoking a certificate after it has been issued. Some implementations use revocation lists, others use short expiration times.

A second problem is finding a CA which is trusted by everyone. Within a company it might be a company server, in a bank the customers trusts the bank server but there is no global authority everyone trusts.

3.1.7 Attacks

When designing a secure system there are several attacks to the cryptography which have to be considered. Knowledge of the attacks enables the designer to analyze the weaknesses of the different algorithms and choose the proper ones to implement in the system. Also sometimes certain steps have to be taken to ensure the proper implementation of algorithms as not to make the system vulnerable to attack.

Here are some of the most common cryptanalysis attacks[2] explained briefly:

Brute force The brute force attack is the simplest attack method possible, it is an exhaustive search of all possible keys. The security level of a system can be defined by how many calculation steps it would require to do a brute force attack. To brute force attack a keysize of N bits would require 2^N calculated steps. The brute force method is the most fundamental form of attack and the following three attacks are modifications of this attack depending on how much information is given.

Ciphertext only This attack is the hardest attack to perform on an encryption, since the attacker has to guess the plaintext and the key, knowing only the ciphertext. This is often done by combining a language directory and a brute-force attack.

Known plaintext The attacker knows the ciphertext and the plaintext, from this he/she tries to decipher the encryption key. This is a common attack since messages often can be predictable and is done by brute force attack.

Chosen plaintext In this attack the attacker gets to choose the plaintext, and given the ciphertext result, tries to decipher the key. This can be done offline where the attacker chooses a list of messages to encrypt, or in the more effective online attack, where the attacker can choose the next message depending on the result of the previous message.

Collision The collision attack is based on the birthday paradox, which is that in a room with at least 23 persons, there is more than 50% chance that two persons have the same birthday. The result of this shows that in a system which uses a random session key of N bits, an attacker could expect a key to be used twice within $2^{N/2}$. This attack applies both to finding collisions in encryption keys and hash-digests. This in practise limits the workload required to do a

[2]The attack concepts are the same as is used in the book *Practical Cryptography* by Bruce Schneier and Niels Ferguson [4].

brute force attack on a system with a N bit key, from 2^N calculations, to a much smaller workload $2^{N/2}$ calculations.

Man in the middle In the man in the middle attack, Eve poses as the man in the middle and impersonates the other party in the communication. This attack is done by intervening, when *Alice* tries to set up a secure channel with *Bob*. Instead the channel is established with *Eve*, who poses as *Bob*. At the same time *Eve* sets up a secure channel with *Bob* where she poses as *Alice*. *Eve* can now read and/or modifying all messages while she forwards them to *Alice* and *Bob*.

3.1.8 Bits of security

To be able to compare the security level between different cryptography technologies, the concept of *work factor* is defined and is measured in *bits of security*. It describes the amount of work the fastest currently available attack would require on the algorithm with the specific key. The fastest attack on an algorithm with N bits of security would require 2^N calculated steps.

3.2 Symmetric cryptography

Symmetric key encryption was briefly mentioned in the previous section 3.1. This section describes the symmetric encryption, its algorithms and variations in more detail. Furthermore, the ciphers and their modes will be presented and data encryption standards will be described.

In symmetric key encryption the sender and the receiver use the same key, K_{enc} (or rarely different keys, but related in an easily computable way). Other names for symmetric key encryption are one-key, single-key and private-key encryption[3].

The modern study of symmetric key encryption is mostly studies of the stream cipher and the block cipher and their applications. A cipher (or cypher) is, as

[3]Use of the latter term can sometimes conflict with the actual *private key* in public-key cryptography and is therefore not further used in this report.

mentioned, an algorithm for performing encryption and decryption, a series of well defined steps taken to ensure that the ciphertext appears as random to anyone eavesdropping.

3.2.1 Stream ciphers

Stream ciphers are used for encrypting a continuing stream of data for transmission on a communication channel. In the stream cipher the bit stream of the *Plaintext* is ciphered using a stream of key bits. The output of the cipher depends on the internal state of the cipher algorithm. Therefore the same text string will result in different *Ciphertext* every time it's encrypted.

The advantages of this cipher are high speed and low hardware complexity. However there are some security issues when implementing the ciphers.

Stream ciphers are designed to encrypt a continuous stream of data. However, the task in the found concept solution, in this thesis, is to encrypt fixed sized data blocks. Therefore stream ciphers will not be considered further.

3.2.2 One-time-pad ciphers

In general a symmetric key cipher is considered secure if the most effective attack has approximately the same workload as a brute force attack. However, they can be broken since the same key is used multiple times. The one-time-pad ciphers solve this problem and give perfect secrecy to the messages sent. This is done by always encrypting the message using a fresh new random key.

For example a four letter message, encrypted using a one-time pad, is impossible for the attacker to decrypt since every possible four letter *Plaintext* could be the true message. The true message is just as likely to be "fast", "kiss" or "stop" from the attacker's viewpoint[4].

The keys that are used once can never be reused and have to be synchronized before communication begins. A key of the same length as the message is required for each message, and special care has to be taken to archive truly random data for the key values. If the key is not perfectly random the security is

[4]Further reading, *The Codebreakers* page 398-399 [5].

compromised.

In its basic form the ciphertext is produced by XOR-ing[5] the message bitwise with the key-data. In the resulting ciphertext the message would be completely indistinguishable from all the possible false answers.

However, this type of cipher requires extensive distribution of keys and is quite impractical in reality. Transferring user information and credentials of 1 KiB of data to the receiver would require 1 KiB of non-reusable key-data. This would require an enormous key distribution structure.

3.2.3 Block ciphers

Block ciphers encrypt the *Plaintext* by dividing the data into fixed sized blocks and processing each block at a time. Each block is processed with a block cipher encryption algorithm. The algorithm processes the fixed size *Plaintext* block of length n, together with a fixed size key and sometimes along with an initialization vector. The output is a fixed size *Ciphertext* block of the same length n.

The substitution box, s-box, is commonly used in block ciphers and are therefore briefly described. S-boxes are tables which present the specific substitute that is to be done to encrypt the input. A simple example is a 3×2 S-box table found in table 3.1. Given a 3 bit input, the 2 bit output is found by selecting the row using the outer bit, and the column by using the the inner 2 bits. Assume the input is 101 which gives the outer bit 1 and the inner bits 01. The output from the S-box will be 10. Note that this is a simple example, some block ciphers are using several S-boxes within the algorithm to enhance the security.

Since encryption is done block by block, block ciphers are vulnerable to known-plaintext attacks, which is as the name implies, an attacker trying to determine the key by comparing the *Ciphertext* to the known *Plaintext* for a specified block. This is countered by using block ciphers in modes, making each block's *Ciphertext* dependent on the previous block output data, Modes will be covered in the following section.

[5] XOR, *exclusive or*, is a boolean operator which returns true only if exactly one of its operands is true.

S-Box

		Inner 2 bits			
		00	**01**	**10**	**11**
Outer bit	**0**	10	11	01	00
	1	11	10	00	01

Table 3.1: *The table shows a simple example of an S-box. It takes a 3 bit input and substitute it to 2 bits which is the output.*

In this section some block ciphers which are relevant to this thesis will be described.

3.2.3.1 Data encryption standard

Data Encryption Standard, DES is a block cipher and was selected in United states in November 1976 to be a Federal Information Processing Standard (FIPS). DES uses a blocksize of 64 bits. These days DES is considered to be insecure for many applications. A DES keysize of 56 bit have been broken in less than 24 hours[6] and should therefore not be used for any new application.

One attempt to enhance the security, was to upgrade the algorithm to 3DES, TDES or TDEA (Triple Data Encryption Algorithm). 3DES is basically three subsequent rounds[7] of DES, each with a new key. This approach may be slower than other block ciphers but has the advantage of being backward compatible with old DES hardware/software.

3.2.3.2 Advanced encryption standard

The Advanced Encryption Standard, AES, was the new standard replacing DES. The AES was decided by a design-contest were several candidates where contributed. The contest was held by The National Institute of Standards and Technology, NIST, and resulted in the new AES standard on October 2, 2000[8].

[6] According to several websites, e.g. Keshava P. Subramanya at University of California, Santa Barbara [6].

[7] A *round* consist of several repetitions of a weak block cipher and several of these rounds in sequence can make the block cipher stronger.

[8] This is found by the Commerce Department's publication of the AES winner [7].

The final five condidates were; Rijndael, Twofish, RC6, Serpent and MARS. The final winner was the Rijndael cipher, invented by Joan Daemen and Vincent Rijmen. Rijndael can be specified by a key and blocksize of any multiple of 32 bits from 128 bits to 256 bits. However, AES is specified only to operate on a fixed blocksize of 128 bits using a keysize of 128, 192 or 256 bits[9].

3.2.3.3 Twofish

Twofish, one of the five finalists in the AES-contest, is a block cipher with a blocksize of 128 bits supporting keysizes up to 256 bits. It is related to the earlier Blowfish[10]. Instead of using fixed tables as S-boxes, values are generated dynamically using information from the key. One half of the key is used in encryption and the other is used generating the S-boxes. It is slower than Rijndael using a key of 128 bits, although faster when using a 256 bit key.

3.2.4 Modes of operation

Block ciphers operate on fixed sized blocks of *Plaintext* and generate *Ciphertext* blocks of the same length. A block cipher mode is an algorithm of how to use the cipher when encrypting more than one block, as explained in 3.2.4.1. Most block cipher modes require the size of the *Plaintext* to be an exact multiple of the blocksize, otherwise the *Plaintext* must be padded. There are many ways of padding a message. A simple example, which is reversible, is to split the message into blocks of the requested size. The last block is padded to reach the specific size by an appended single byte with value 128, followed by as many zero bytes as needed.

The mode appends an additional operation which operates on different data according to the mode. The data often consist of the output, the *Ciphertext*, from the previous block which can be combined with the *Plaintext* block in order to be encrypted. This is done by CBC mode covered in section 3.2.4.2.

Using a block cipher mode ensures that if encrypting a number of identical

[9]Found in the publication of the AES standard FIPS-197 [8].

[10]Blowfish is a 16-rounds cipher designed in 1993 by Bruce Schneier. It has a blocksize of 64 bit and a keysize of anywhere from 32 bits to 448 bits.

Plaintext blocks chained together, the resulting *Ciphertext* could be perceived as random. If the same *Plaintext* is encrypted using just a block cipher a *Ciphertext* containing repeating blocks would be produced. There are many different kinds of modes and some are presented below.

3.2.4.1 Electric codebook mode, ECB

ECB, *Electronic codebook*, is the simplest encryption mode which encrypts each block of the message separately. This mode is not recommended for use in any cryptography protocol at all[11]. The ECB mode is weak, because if two *Plaintext* blocks are the same, the *Ciphertext* blocks will be the same. This will leak information to the attacker depending on the structure of the message.

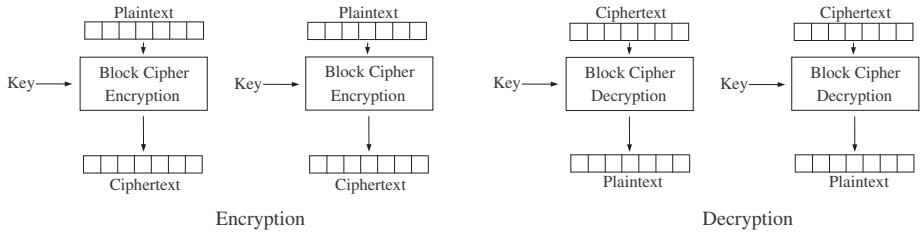

Figure 3.2: *The figure describes the encryption and decryption respectively, done by the easiest cipher mode called electronic codebook mode, ECB. The two ciphers to the left performs the encryption while the other two, the rightest ones, performs the decryption.*

3.2.4.2 Cipher block chaining mode, CBC

A common block cipher mode is CBC, *Cipher Block Chaining*. In contrast to ECB this mode is XORing each *Plaintext* block with the previous *Ciphertext* block and uses the result as input to the cipher. This avoids the problem with ECB and gives different *Ciphertext* blocks from the same *Plaintext* block. In the first cipher block there is no previous *Ciphertext* block, instead an initialization vector, IV, is used.

[11]This is mentioned in Bruce Schneier's and Neils Ferguson's book *Practical Cryptography* page 69 [4].

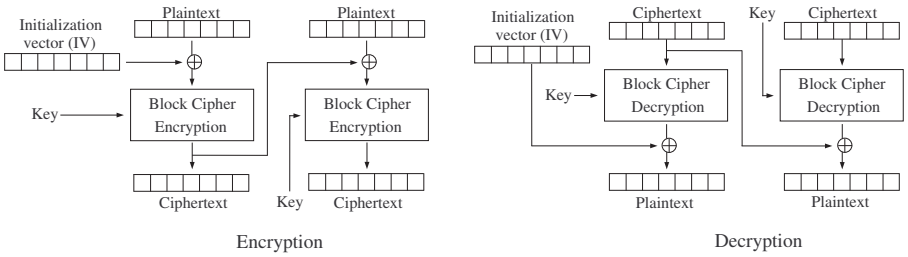

Figure 3.3: *This figure shows the encryption and decryption performed in cipher block chaining mode, CBC. Each plaintext block are XORed with previously ciphertext block to enhance the security of the cipher. The inversed procedure is done in decryption where each output block is XORed with previous ciphertext block to get the plaintext.*

Initialization vector An initialization vector, IV, is a block of text used to make the first input more randomized. This vector should not be fixed, because that will give the same problem as in ECB in the first block. Two messages with the same beginning will get the first *Ciphertext* block identical. Neither should a counter be used to vary IV due to the same problem. The problem is solved by using a *Nonce*[12] generated IV.

The recipient must know the IV to be able to decrypt the message. There is no requirement that the IV has to be kept secret so *Alice* can send the IV along with the message to *Bob*. Both *Alice* and *Bob* have to ensure the IV has not been used before, usually this is done by using the message number.

3.2.4.3 Counter mode, CTR

Counter mode turns the block cipher into a stream cipher. The CTR mode was standardized in 2001[13] although it has been around since the DES came in 1980. This stream cipher mode require some form of *Nonce* which is to be the input to the cipher together with the key. The *Nonce* is only to be used once with the same key and CTR often uses a counter to make sure the *Nonce* is only used once. To generate the key stream CTR concatenates the *Nonce* with the counter value, i, and encrypts it to form a single block, shown in equation 3.2.2. To fit the usual blocksize the typical setup is: 48 bits message number, 16 bits of

[12]The term *Nonce* is a contraction of number used once. The property of a *nonce* is critical due to the fact it must be unique.

[13]NIST standardization [9].

additional *Nonce* data and 64 bits of the counter

$$K_i := E(K, Nonce \ || \ i) \quad for \ i = 1, ..., k \tag{3.2.1}$$

$$C_i := P_i \oplus K_i \tag{3.2.2}$$

As shown in *figure* 3.4, the the output from the cipher, K_i within the equation 3.2.2, is XOR'ed with the *Plaintext* and thereby the *Plaintext* is encrypted into the *Ciphertext*.

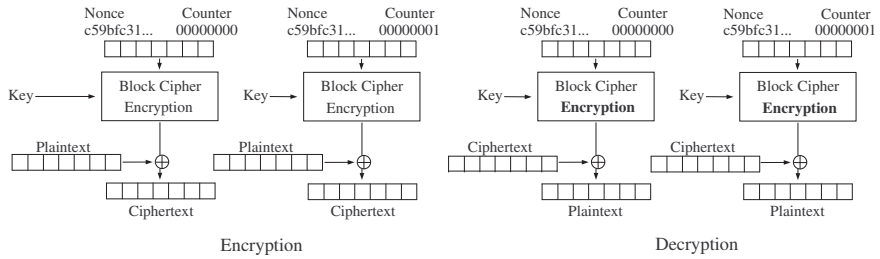

Figure 3.4: *The figure describes encryption and decryption in counter mode, CTR. Each cipher's input block in both encryption and decryption is a nonce. The plaintext in encryption is XORed with the output block to create the ciphertext. To decrypt the ciphertext, same procedure is made though the output block is XORed with the ciphertext to get the plaintext.*

There are several modes not described here, e.g. the cipher feedback mode (CFB) or the output feedback mode (OFB) etc. The modes described above are the most common and therefore considered and described within this thesis.

3.3 Hash functions

A hash function is a reproducible method that takes a character string of arbitrary length as input and produces a fixed-size result. This result can be relatively small and a typical use of a hash function is in digital signatures. These functions are sometimes called message digest functions, the result digest or fingerprint.

Hash functions are designed to be fast and to yield few hash collisions. This is impossible because it is usually a requirement that the hash value is stored in

fewer bits than the data being hashed. A lager hash value decreases the chance of a hash collision. A well designed hash function is a one-way operation.

A typical application is as follows; *Alice* sends a tough math problem to *Bob* and claims to have solved it. *Bob* would like to solve it himself. To be sure that *Alice* already solved it, she appends a random *Nonce* to the solution and computes its hash value, which is sent to *Bob*. The *Nonce* is kept secret until *Bob* has solved the math problem. He then appends the *Nonce*, given afterwards, and computes its hash value, which is compared to the previous value sent by *Alice*.

Other applications are the uses of hash values for message integrity, determining of whether or not any changes have been made to the message. The hash value is to be known by the receiver, though it must be protected during the communication.

3.3.1 Secure Hash Algorithm, SHA

The SHA functions are five hash-functions developed by the National Security Agency, NSA[14]: SHA-1, SHA-224, SHA-256, SHA-384 and SHA-512. The first function mentioned, SHA-1, uses a blocksixe of 512 bits and it produces a hash-digest of 160 bits from a message with maximum length of $(2^{64} - 1)$ bits.

The later SHA-224 and SHA-256 also works with a blocksize of 512 bits however they produce a hash-digest of 224 bits and 256 bits respectively. The maximum length of the message is still $(2^{64} - 1)$ bits.

The last functioned mentioned, SHA-384 and SHA-512, works with a blocksize of 1024 bits and the maximum message length is $(2^{128} - 1)$ bits. The two functions produce a hash-digest of 384 bits and 512 bits respectively.

3.4 Asymmetric cryptography

Asymmetric key encryption introduced in chapter *Basic Cryptography*, 3.1.4, will be further described in this section. The most common algorithms are RSA and

[14]These five algorithms for computing cryptographic hash functions are specified in *FIPS 180-2* [10]

ECC which are described described in detail below.

The two main branches of asymmetric cryptography are *Public key encryption*, introduced in section 3.1.4, and *Digital signatures*, section 3.1.5. The asymmetric algorithms are based on one-way functions, described in the article *The Principles of Science* [11] by William S. Jevons back in 1874. The first publication of asymmetric cryptography was in 1976 when W. Diffie 3.4.2.1 and Martin E. Hellman published *New Directions in Cryptography* [12], which is a cryptographic protocol of key exchange. However, the British signals intelligence agency, GCHQ, made the first invention, in the early 1970, although they kept it secret until 1997. NSA has also claimed to have invented the public-key cryptography, in the 1960:s, however there is little public evidence supporting this claim.

3.4.1 Mathematical description

There are several different asymmetric encryption schemes using different mathematical problems in order to make them irreversible. Algorithms described hereafter were considered being used within the project. To be able to understand the better part of the algorithms, most of the mathematics terms are being described, though some are assumed to be well known.

There are mainly three families of asymmetric cryptography. The most widely used are those based on the *integer factorization*, RSA in particular. The next mathematical problem providing use within cryptography is the *discrete logarithm problem*, DLP, used in digital signatures (DSA) and key agreement (Diffie-Hellman). The last family is based on arithmetic using elliptic curves. This problem is based on one-way function use of elliptic curves and is called the *elliptic curve discrete logarithm problem*, ECDLP.

Integer factorization problem The problem is to factorize a large integer constituting of two primes, e.g. finding a and b given 15. The solution could be 3 and 5. Every integer has a unique prime factorization. Although the multiplication of two prime numbers is easy, the factorization is much more difficult, even with currently availible algorithms[15].

[15]This is described by for example the RSA laboratories [13].

The discrete logarithm problem, DLP This problem is to compute k out of e.g $3^k \equiv 13 \ mod(17)$. In this example $k = 4$ thus $3^4 = 81$. 81 divided by 17 gives the remainder equal 13. There is no efficient algorithm for computing general discrete logarithms known today.

The elliptic curve discrete logarithm problem, ECDLP The ECDLP is similar to the one-way function on which DSA and Diffie-Hellman are based. The two different groups, additive cyclic group and the multiplicative group, can be considered similar and thereby the similar names on the problems. The ECDLP is the problem of finding the scalar k which is the value the point G is multiplied by into kG on an elliptic curve, given only the points G and kG.

3.4.2 Methods and algorithms

3.4.2.1 Diffie-Hellman

As previously mentioned the article *New directions in cryptography*[12] presents a key agreement scheme, which is a scheme describing how to jointly establish a secret key over an insecure communication channel. Merkle's work on public key distribution was an influence to Diffie and Hellman. Hellman suggested, in 2002, the algorithm to be called *Diffie-Hellman-Merkle key exchange* due to Merkle's contribution.

The key exchange procedure is as following

1. *Alice* and *Bob* agree to use a prime number, such as $p = 23$ and a base $g = 5$.
2. *Alice* chooses a secret integer $a = 6$ and sends *Bob* $(g^a \ mod(p))$
   - $5^6 \ mod(23) = 8$
3. *Bob* chooses a secret integer $b = 15$ and sends *Alice* $(g^b \ mod(p))$
   - $5^{15} \ mod(23) = 19$
4. *Alice* computes $(g^b \ mod(p))^a \ mod(p)$
   - $19^6 \ mod(23) = 2$

5. *Bob* computes $(g^a \ mod(p))^b \ mod(p)$

   - $8^{15} \ mod(23) = 2$

The result, 2, in the last equation is the shared secret. The information *Eve* got was the prime $p = 23$, the base $g = 5$ and she also knows $5^a \ mod(23) = 8$ and $5^b \ mod(23) = 19$. The last two equations give *Eve* no information due to the *discrete logarithm problem*. The only thing to be kept secret are the two numbers $a = 6$ and $b = 15$ also the shared secret $g^{ab} = g^{ba}$. Note that the numbers a and b should be large primes in a real system. This key exchange and the use of the *discrete logarithm problem* seems to be the take off for asymmetric cryptography.

3.4.2.2 Rivest, Shamir and Adleman, RSA

Diffie-Hellman's work was followed by the asymmetric algorithm RSA which is the first algorithm suitable for encryption as well as signing. It was invented by Ronald Rivest, Adi Shamir and Leonard Adleman in 1977.[16] The letters RSA are the initials of their surnames. RSA was one of the first great advances in asymmetric cryptography and the algorithm is based on the integer factorization problem, see paragraph 3.4.1. This section describes how to generate a pair of keys and use them in encryption and decryption respectively.

Generating the keys This task can be divided into five steps. The first step, generate large prime numbers, requires a cryptographically secure random number generator. These prime numbers for RSA are in the order of thousands of bits, the smallest recommended keysize by NIST is 1024 bits. A simple example is given with small numbers to show the principle of the key generation.

1. Choose (generate) two[17] different large prime numbers, $\boldsymbol{p}$ and $\boldsymbol{q}$

   $p = 7$

   $q = 19$

[16] More history of the RSA algorithm is found at the RSA laboratories [14].

[17] in the RSA cryptography standard PKCS #1 v2.1 [15] a so-called multi-prime is mentioned where it is possible to choose u distinct primes.

2. Let n, the modulus for both the public and the private key, be defined by:

$$n = p \cdot q \tag{3.4.1}$$

$$\begin{aligned} n &= 7 \cdot 19 \\ &= 133 \end{aligned}$$

3. Compute the totient[18]

$$\Phi(n) = (p-1)(q-1) \tag{3.4.2}$$

$$\begin{aligned} \Phi(n) &= (7-1)(19-1) \\ &= 6 \cdot 18 \\ &= 108 \end{aligned}$$

4. Choose a small integer e which is a coprime to $\Phi(n)$ such that $1 < e < \Phi(n)$

e	GCD(e, $\Phi(n)$)	Status
$e = 2$	$GCD(2, 108) = 2$	incorrect
$e = 3$	$GCD(3, 108) = 3$	incorrect
$e = 4$	$GCD(4, 108) = 4$	incorrect
$e = 5$	$GCD(5, 108) = 1$	correct

Table 3.2: *The table shows the different attempts of finding e out of $\Phi(n)$. In this example $e = 5$ is a small number and a coprime to $\Phi(n)$ and thereby a solution.*

5. Compute d to satisfy the equation

$$de \equiv 1 \ \ mod(\Phi(n)) \tag{3.4.3}$$

Equation 3.4.3 can be written as $de = 1 + k\Phi(n)$ or $d = \frac{(1+k\Phi(n))}{e}$ where k is any integer. The last equation gives following table.

[18]The totient, Φ, of a positive integer, n, is defined to be the number of positive integers less than or equal to n which are *coprime* numbers to n, e.g. $\Phi(9) = 6$, since there are six numbers; 1, 2, 4, 5, 7 and 8 that are *coprime* numbers to 9.

k	$d = (1 + k\Phi(n))/e$	Status
$k = 0$	$d = (1/5) = 0.2$	incorrect
$k = 1$	$d = (109/5) = 21.8$	incorrect
$k = 2$	$d = (217/5) = 5.4$	incorrect
$k = 3$	$d = (325/5) = 65$	correct

Table 3.3: *The table describes the procedure of finding d by letting the integer k increase from* 0 *until d becomes an integer. Note that this is a simple example and to achieve a high security level the smallest d may not be the optimal solution.*

This was a basic example. The calculations are becoming more time consuming as the integers are expanding. A more efficient way of finding d, is to use *Euclid's algorithm*[19]. However, these are the basics of key generation in RSA. The integers e and d, chosen in step four respectively five, are the private and the public key. It is possible to choose which integer is to be public respectively private. The different sizes of e and d makes them suitable for different calculations. The integer e is smaller than d, and by choosing e as the private key, the decryption process will be faster than the encryption process. When choosing e to be the public key, the encryption will be faster than the decryption. However, The integer d is hereafter described as the private key and e is the public key.

RSA encryption This computation requires the integers n and e and the specific *Plaintext*, m. The size of the message is limited by the size of n. This means that in RSA encryption the message must be divided into, k, blocks $m = m_0, m_1 ... m_k$ where each $m_i < n$. In the example the message '6' is used which is less than $n = 133$. The encryption algorithm is done by:

$$c = m^e \ \ mod(n) \tag{3.4.4}$$

[19]The Euclidean algorithm also known as Euclid's algorithm, is an algorithm to determine the *greatest common divisor* (GCD) of two elements. It is dated back to the ancient Greeks and is one of the oldest known algorithm [16].

Encrypting the message '6' using the public key $e = 5$ results in the *Ciphertext*:

$$\begin{aligned} c &= 6^5 \ \ mod(133) \\ &= 7776 \ \ mod(133) \\ &= 62 \end{aligned}$$

Because of the expensive computations of RSA it is not recommended to split the message into several blocks. The solution is to encrypt the message with a symmetric encryption and then encrypt the secret key with asymmetric encryption. The decryption in RSA is however not as easy as the RSA encryption.

RSA decryption The decryption is similar to the encryption but it involves a larger exponent. The decryption requires the *Ciphertext*, c, the integer n and the private integer d and is calculated by:

$$m = c^d \ \ mod(n) \tag{3.4.5}$$

Decrypting the *Ciphertext* '62' using the private $d = 65$ will hopefully result in message $m = 6$. The calculations are done by:

$$\begin{aligned} m &= 62^{65} \ \ mod(133) \\ &= 62 \cdot 62^{64} \ \ mod(133) \\ &= 62 \cdot (62^2)^{32} \ \ mod(133) \\ &= 62 \cdot 3844^{32} \ \ mod(133) \\ &= 62 \cdot (3844 \ \ mod(133))^{32} \ \ mod(133) \\ &= 62 \cdot 120^{32} \ \ mod(133) \end{aligned}$$

The sequence of operations which reduced 62^{65} to 120^{32} is now repeated to reduce the exponent to 1.

$$\begin{aligned} m &= 62 \cdot 36^{16} \ \ mod(133) \\ &= 62 \cdot 99^8 \ \ mod(133) \\ &= 62 \cdot 85^2 \ \ mod(133) \\ &= 62 \cdot 43 \ \ mod(133) \\ &= 2666 \ \ mod(133) \\ &= 6 \end{aligned}$$

As shown in decryption there are a great number of operations, however this example is using small numbers. Another short example is $p = 61$ and $q = 53$. These prime numbers give $n = 3233$ and the totient $\Phi(n) = 3120$. Then chose $e = 17$ and compute $d = 2753$. The use of the public and the private key will be:

The **public key** is $(n = 3120, e = 17)$

$$\begin{aligned} c &= m^e \ \ mod(n) \\ &= m^{17} \ \ mod(3120) \end{aligned}$$

The **private key** is $(n = 3120, d = 2753)$

$$\begin{aligned} m &= c^d \ \ mod(n) \\ &= c^{2753} \ \ mod(3120) \end{aligned}$$

3.4.2.3 Digital signature algorithm, DSA

This algorithm is within the Digital Signature Standard, DSS, issued by NIST[20]. The algorithm is used to compute a digital signature of a message and provides both signature generation and verification. The signature generation process, uses the private key and the verification process uses the public key.

DSA parameters The parameters within DSA are similar to RSA's parameters and are declared as follows[21]:

1. $p =$ a prime modulus, where $2^{L-1} < p < 2^L$ for $512 \leq L \leq 1024$ and L is a multiple of 64
2. $q =$ a prime divisor of $p - 1$, where $2^{159} < q < 2^{160}$
3. $g = h^{(p-1)/q} \ \ mod(p)$, where h is any integer where $1 < h < p - 1$ such that
   $h^{(p-1)/q} \ \ mod(p) > 1$ (g has order $q \ \ mod(p)$)

[20]This Federal Information Processing Standard Publication (FIPS) prescribes three algorithms suitable for digital signature generation and verification: Digital Siganture Algorithm (DSA), RSA ds algorithm and Elliptic Curve Digital Signature Algorithm (ECDSA) [17].

[21]The example is found in the NIST - publication FIPS PUB 186-2 [17].

4. $x =$ a randomly generated integer where $0 < x < q$

5. $y = g^x \ \ mod(p)$

6. $k =$ a randomly generated integer where $0 < k < q$

The integers p, q and g can be public as well as the public key y. However, x is to be kept secret as it is the private key. The public and the private key respectively are normally constant while the parameter k is generated for each signature, and thereby to be kept secret.

DSA signature generation The signature generation, according to DSS[22], makes use of the hash function SHA-1. However, several different hash functions can be used. The signature of message m is the two numbers r and s which are computed according to the equations

$$r = (g^k \ \ mod(p)) \ \ mod(q)$$
$$s = (k^{-1}(SHA\text{-}1(m) + x \ \ r)) \ \ mod(q)$$

DSA signature verification Let m', r' and s' correspond to m, r and s to represent the received parameters and y is the public key to the signature. There are two conditions the verifier checks first, $0 < r' < q$ then $0 < s' < q$. If the received parameters do not fulfill the conditions the signature shall be rejected, otherwise the following procedure is done:

$$\begin{aligned} w &= (s')^{-1} \ \ mod(q) \\ u1 &= ((SHA-1(m'))w) \ \ mod(q) \\ u2 &= (r'w) \ \ mod(q) \\ v &= (((g)^{u1}(y)^{u2}) \ \ mod(p)) \ \ mod(q) \end{aligned}$$

If $v = r'$, the signature is verified and the receiver knows from whom it was sent. Otherwise the message may have been changed, the generation of the signature may have been incorrect or it may have been posted by somebody else.

[22]The Digital Signature Standard are recommending SHA-1 in the FIPS 186-2 [17].

3.4.2.4 Elliptic curve cryptography, ECC

Elliptic curve cryptography was discovered by Victor Miller (IBM) and Neil Koblitz (University of Washington) in 1985[23]. ECC is, unlike RSA, based on the (elliptic curve) discrete logarithm problem. Although the use of elliptic curves within cryptography is relatively new, the mathematics of elliptic curves is not. There are two types of elliptic curves. The elliptic curve can either be **odd characteristic**, also called *modulo p* or *prime characteristic*, based on prime numbers or **even characteristic**, also known as *over finite field with* 2^m *elements*, based on binary numbers. An implementation of ECC requires a specific curve which is to be known by both *Alice* and *Bob*. The choice of the elliptic curve parameters is important due to the security of the algorithm since it depends on the curve. NIST presents a list of 15 standardized elliptic curves of different groups and keysizes [24]. The equation describing the elliptic curve is:

$$y^2 = x^3 + a\,x + b \tag{3.4.6}$$

where x, y, a and b are real numbers and with the condition

$$4a^3 + 27b^2 \neq 0 \ \ mod\ p \tag{3.4.7}$$

All points, (x, y), satisfying the condition along with a finite point O and the addition operation, form a *group*. In order to present the group and the equations done by the algorithm, the following simple example[25] is used to show the procedure of generating an elliptic curve group, which is required to accomplish the encryption and decryption respectively .

Example of group generation

First an elliptic group is defined which could be either of the two characteristic groups, prime or binary numbers. The prime characteristic, E_p, is chosen within this example. Let the prime number $p = 23$ and let the constants $a = 1$ and $b = 1$. This makes the elliptic curve equation $y^2 = x^3 + x + 1$. The condition gives $8 \neq 0$, which verifies the group. The following step is to determine the quadratic residues Q_{23} which is done by calculating $1^2\ mod\ 23$, $2^2\ mod\ 23$, ...,

[23] According to Certicom [18].

[24] The standardized curves are found at NIST's homepage [19].

[25] The example is described in more detailed by Jean-Yves Chouinard [20].

22^2 *mod* 23. All the answers will be Q_{23}= $\{1,2,3,4,6,8,9,12,13,16,18\}$. Now, for $0 \leq x < p$, compute $y^2 = x^3 + x + 1 \ \ mod\ 23$ and determine if y^2 is in the set of quadratic residues Q_{23}. This gives all the points, excluding the finite point O, for the group E_p as:

$$E_{23} = \left\{ \begin{array}{ccccccc} (0,1) & (0,22) & (1,7) & (1,16) & (3,10) & (3,13) & (4,0) \\ (5,4) & (5,19) & (6,4) & (6,19) & (7,11) & (7,12) & (9,7) \\ (9,16) & (11,3) & (11,20) & (12,4) & (12,19) & (13,7) & (13,16) \\ (17,3) & (17,20) & (18,3) & (18,20) & (19,5) & (19,18) & \end{array} \right\}$$

The point $(4,0)$ is added as the finite point O to make the group complete. To continue the operations on the group, several parameters are to be defined. Let $P = (x_1, y_1)$ be a point on the curve and define $-P = (x_1, -y_1)$. If $Q = (x_2, y_2)$ is defined as a similar point although $P \neq -Q$, then the addition, $P(x_1, y_1) + Q(x_2, y_2) = -R(x_3, -y_3)$, is defined by:

$$x_3 = \lambda^2 - x_1 - x_2 \ \ mod\ p \tag{3.4.8}$$

$$-y_3 = \lambda(x_1 - x_3) - y_1 \ \ mod\ p \tag{3.4.9}$$

where

$$\lambda = \begin{cases} \frac{y_2 - y_1}{x_2 - x_1} & \text{if } P \neq Q \\ \frac{3{x_1}^2 + a}{2y_1} & \text{if } P = Q \end{cases}$$

In excess of addition, multiplication is used over an elliptic curve group $E_p(a,b)$ which is the equivalent operation as the modular exponentiation in RSA. Let $P = (3,10) \in E_{23}(1,1)$. Then $2P = (x_3, y_3)$ is equal to: $2P = P + P = (x_1, y_1) + (x_1, y_1)$.

Since $P = P$, the multiplication procedure is a special case of addition and the values of λ, x_3 and y_3 are given by:

$$\lambda = \frac{3{x_1}^2 + a}{2y_1} \ mod(p) \quad \Rightarrow 6 \tag{3.4.10}$$

$$x_3 = \lambda^2 - x_1 - x_2 \ mod(p) \quad \Rightarrow 7 \tag{3.4.11}$$

$$y_3 = \lambda(x_1 - x_3) - y_1 \ mod(p) \quad \Rightarrow 12 \tag{3.4.12}$$

Equation 3.4.11 gives: $x_3 = 6^2 - 3 - 3 \ mod(23) = 7$ and equation 3.4.12 gives: $y_3 = 6 \cdot (3 - 7) - 10 \ mod(23) = 12$. Therefore $2P = (x_3, y_3) = (7, 12)$. The multiplication kP is obtained by repeating the elliptic curve addition k times.

Elliptic curve encryption and decryption For *Alice* to encrypt a message m, the message must be encoded into a point, $P_{message}$, from the finite set of points in the elliptic group $E_p(a, b)$. This can be done in several ways, however it is not included in this example. In order to encrypt, the next step after generating the group, is to choose a generating point $P \in E_p(a, b)$, such that the smallest value of n for which $nP = O$ is a large prime number. Both the group E and the generating point P can be public.

An example of ECC encryption Consider the elliptic curve $y^2 = x^3 - x + 188 \ mod \, 751$, $(a = -1, b = 188$ and $p = 751)$. The elliptic curve group is $E_{751}(-1, 188)$. Let the generator point be $G = (0, 376)$, then the multiples of kG are within $1 \leq k \leq 751$.

Alice's message to *Bob* is encrypted with *Bob*'s public key. Suppose his randomly chosen private key is $n_B = 85$. Then the public key, P_B is given by:

$$P_B = n_B G \tag{3.4.13}$$

which is $85(0, 376) = (671, 558)$. Suppose the message is encoded to the point $P_{message} = (443, 253) \in E_{751}(-1, 188)$. By using *Bob*'s public key, P_B, and the message point, $P_{message}$, together with a randomly chosen number $k = 113$, *Alice* encrypts the message point into the ciphertext pair of points as following:

$$
\begin{aligned}
P_C &= [(kG), (P_{message} + kP_B)] \\
&= [(113 * (0, 376)), ((443, 253) + 113 * (671, 558))] \\
&= [(34, 663), ((443, 253) + (47, 416))] \\
&= [(34, 633), (217, 606)]
\end{aligned}
$$

The resulting ciphertext, the pair of points, is $P_C = [(34, 633), (217, 606)]$. *Alice* simply moves the generation point and *Bob*'s public key point with a certain number k, then adds the message point to the public point. The number k is not to be known be anybody, not even *Bob*.

An example of ECC decryption *Bob* now have the pair of points:
$P_C = [(34, 633), (217, 606)]$. In order to decrypt this ciphertext *Bob* first multiplies the first point in P_C with his own private key, $n_B = 85$. The resulting point $n_B(kG)$ is than subtracted from the second point of P_C, $(P_{message} + kP_B)$. The last calculation gives the message point $P_{message}$. The same algorithm used to encode the message into the message point is now used in reverse to get to the message.

$$
\begin{aligned}
P_C &= [(kG), (P_{message} + kP_B)] \\
P_{message} &= (P_{message} + kP_B) - n_B(kG) \\
P_{message} &= P_{message} + k \cdot (P_B - n_B G) \\
P_{message} &= P_{message} = (443, 253)
\end{aligned}
$$

Since *Bob*'s public key P_B is calculated by his private key n_b multiplied by the generation point G, $P_B = n_B G$, the message is found. *Bob* simply moves the generation point, already moved k times by *Alice*. The resulting point is than subtracted from his public point, also already moved by *Alice*, and thereby getting the message point. In ECC the subtraction $A - B$, is done by changing sign on y_B from $B = (x_b, y_B)$ to $B = (x_B, -y_B)$ which is the definition of the reflection point.

Elliptic curve digital signature, ECDSA This is another process of computing a digital signature compared to DSA mentioned in section 3.4.2.3. ECDSA is a variant of DSA although it is based on elliptic curves. The advantage of using ECDSA is the same as the general with elliptic curve cryptography, the keysize.

A security level of 80 bits, meaning the attacker needs 2^{80} signature generations to find the key is equivalent with a DSA keysize of 1024, however the size of an ECDSA key is only 160 bits long.

3.5 Pseudorandom number generators

Keys are generated from a source of random data. The keys are the most important part of the cryptography session. It is important that the keys are constructed from truly random and unpredictable data. In a computer system, which is a logical machine which behaves in a predictable way, it can be hard to find a good source of randomness. Randomness is measured in entropy, which is a definition of how many bits of acquired random data are truly random.

Since real random data is hard to come by in a computer, a pseudorandom number generator , PRNG, is used. The pseudorandom generator provides random data calculated from an internal state and then updates the internal state. The generator is seeded with the real random data periodically to make it hard to determine the internal state.

The choice of a proper PRNG is an important design consideration, it is important to find a cryptographically strong PRNG, with an uncorrelated and uniform spectrum. Equally important is finding a good system specific source of entropy. However, this is a design consideration outside the scope of this thesis work, where we simply aim to demonstrate the concept.

3.6 Summary

The choices of which algorithms and keysizes to be further tested and benchmarked in this thesis, are made based on research in books and recommendations by standardization institutes. The decision is also influenced on which algorithms are availible in the cryptographic libraries, these libraries will be further explained in section 4.3.3.

Decisions regarding the symmetric ciphers are mainly based on the book *Practical Cryptography* by Bruce Schneier and Niels Ferguson. The ciphers which are

to be implemented and benchmarked are DES, AES and Twofish. The keysizes to be tested are if possible in the range of 128-256 bits.

For the choices of asymmetric algorithms the recommendations were mostly from NIST, which recommends both RSA- and ECC-technology. Recommended keysizes of the asymmetric algorithms are ranging from 1024 bits to 4096 bits for RSA and 160 bits to 521 bits for ECC.

CHAPTER 4

Technology considerations

The goal of the testplatform implementation is to study the requirements needed by an embedded system when performing asymmetric cryptography operations. In the testplatform-implementation the focus will be on benchmarking ECC and RSA operations in the embedded system. Moreover, a speed comparison on different implementations of the symmetric algorithms will be done.

The testplatform will further be used to implement and test a concept of a complete lockunit, however this is outside the scope of this report.

4.1 Approximating the requirements

The complications associated with this type of cryptography is that it relies on computationally heavy math operations performed on large integers. Computations on large integers require a large amount of memory and processor instruction cycles.

4.1.1 Related work

A short websearch provided some reports of previous studies implementing asymmetric cryptography on embedded platforms. A report [1] describing the implemention of ECC and RSA on 8 bit platforms with keysizes of up to 1024 bits RSA and 160 bits ECC, showed that it is possible to implement RSA and ECC with very small resources.

[1] Efficient Implementation of Public Key Cryptosystems on Mote Sensors (Short Paper).

Furthermore, another thesis report[2] described implementing RSA algorithms on an embedded platform gave some performance figures of RSA (up to 1024 bits) on an 32 bit ARM7 platform.

Studying both reports gave an estimate of what execution time to expect for the asymmetric operations.

4.1.2 Code design approach

Doing a completely new design would require a considerable amount of time and resources from the project. The result would be an efficient and fast implementation allowing a small and power-efficient processor with limited resources to be used. However, this would probably require more time than what is available for this thesis work, and there is a great probability that the resulting application would have unexpected security flaws.

Instead the decision was to go with a free open-source library for the implementations of the cryptographic algorithms. This likely results in a slower implementation requiring more resources, but the advantage is that a public library is reviewed by more people and therefore security flaws can be discovered faster. The free public library is a platform independent and flexible way to go. It can easily be tweaked to suit a new processor platform when a new product version is required.

The primary memory requirement is that the processor has to be able to process all the data and keys in RAM memory. The largest datablock considered in this evaluation will be the RSA 4096 bit (512 Byte) key. Therefore the approximated RAM memory requirement will be at least double the key-size, 1 KiB, to perform the arithmetic calculations. Additionally another 2 KiB of RAM is required, to store input and output variables without the need for slow write operations to flash memory. Therefore the minimum RAM memory requirement is 3 KiB, although more would greatly simplify the development process.

There is a certain tradeoff between computation time and required memory

[2] A master's thesis *Implementing the Transport Layer security Protocol for embedded systems*, by Bengt Werstén [21] done at Linköping University.

resources. It is sometimes possible to speed up an algorithm considerably by using pre-calculated lookup-tables of common calculations.

4.2 Choosing processors

The processor performance has on average been doubled every two years during the last decades, according to Moore's law [22]. This makes it feasible to use more complex cryptography in smaller embedded systems than was possible just a few years back.

Some processors have builtin modules to help perform the cryptographic routines, enabling much faster calculation times and freeing up processor time for other tasks. The devices on the market today are mostly modules for AES- and 3DES-operations. Also several processors have builtin RSA accelerating modules enabling fast routines for performing operations on the smaller RSA keys (< 1024 bits).

Using a public library instead of writing processor-specific cryptographic algorithms is a less efficient way to go, therefore the processor has to be more powerful. The consensus from the related work study was that at least a 16 bit architecture has to be used. However, since the processor has to process large datablocks and most 16 bit processors generally have limited amounts of RAM, a 32 bit architecture seemed to be a more reasonable choice.

The considered processors were narrowed down to some candidates from the largest manufacturers:

- **MSP430-Series, Texas Instruments** The MSP430-series is a low-power processor family, with a 16 bit RISC core running at up to 16 MHz. The onchip memory is up to 10 KiB RAM and up to 60 KiB Flash. The more advanced devices in the series features specific modules for hardware multiplication and automated data transfer, DMA. The MSP430-series is designed for low power consumption, it averages at $330\mu A/MIPS$ and has a power-down mode only requiring $0.2\mu A$. This makes it extremely suitable for battery powered applications.

- **AT91SAM7XC-Series, Atmel** The AT91SAM7X-series processor family is built on the 32 bit ARM7 RISC core which operates at 60 MIPS, with a memory of up to 128 KiB RAM and 512 KiB Flash. The processor has several on-chip peripheral modules to simplify communication interfaces, such as USB, Ethernet, SPI and RS232. Furthermore, the processor has two specialized cryptography modules supporting AES- and 3DES-operations.

- **LPC2000-Series, NXP** The LPC2000-Series is built on the ARM7 RISC Core and is very similar to the Atmel processor series listed above. Except that it has slightly less memory and does not contain the cryptography modules.

In general, the processor choice for benchmarking the cryptographic algorithms was to start with abundance of performance, rather than to start with the limited system the 16 bit MSP430 would provide. However, the MSP430 would be a good platform for the final product due to the extremely low power-consumption.

The remaining two candidates were based on the ARM7 core, so while the final choice was the AT91SAM7XC-series from Atmel, this was mostly due to the symmetric cryptography module and the amount of available demo code.

4.3 Development tools

4.3.1 Evaluation board

The AT91SAM7X-EK evaluation board from Atmel provides a simple start with the AT91SAM7XC256 processor. The board has several implemented interfaces and is delivered with plenty of example code, enabling easy access to most of the features. Further specifications and a complete block diagram can be found in the *AT91SAM7X-EX User Guide*[23].

4.3.2 Compiler and code environment

Three major development tools were considered for the platform chosen. In the end the decision was based on the purchase price of the tool.

Embedded workbench for ARM, IAR systems The Embedded Workbench suite provides a collection of all development tools required for ARM7 development, including compiler, debugger and libraries. The version considered and tried in this project was the kickstart version limited to 32 KiB of program code.

Embedded Workbench includes several great examples which makes it easy to get started. However, the code size limitation made this tool unusable and there was no budget for purchasing a compiler to the project.

Yagarto, open-source Yagarto[3] is an open-source toolchain based on the GCC ARM package. It is compiled for windows and contains a complete package of compiler, assembler, linker and debugger for ARM7-core devices. The package also includes the Eclipse Environment with the Zylin CDT, (C/C++)-plugin.

CrossWorks for ARM, Rowley associates The Crossworks development system is a commercial integrated environment based on the GNU Compiler toolchain. Since it is based on the GNU toolchain it was not considered as a real alternative to Yagarto.

The IAR Embedded workbench was used when getting started with the evaluation board, since it was delivered with demonstration code for this environment. When the code size increased beyond the size limit of the kickstart version the switch was made to the Yagarto toolchain since it was free. However, it should be noted that the Yagarto toolchain requires quite a bit of time and tweaking before it is properly configured and development begins.

4.3.3 Cryptography libraries

The first considered implementation was to program new routines for cryptographic algorithms. However, after some research on the cryptographic routines to be tested, it was quickly realized that this would be an enormous task. Taking up to much time from the project and thus limiting the number of algorithms to be tested.

[3]Yet Another GNU ARM Toolchain.

Instead it was decided not to reinvent the wheel and implement one of the numerous public domain cryptographic libraries available. Some of the primary ones considered were:

Crypto++ The Crypto++ library is a free[4] C++ library which implements most cryptographic routines. It contains support for both ECC and RSA with both encryption and signatures. It contains projects and builds out of the box on most compilers on both Windows and Linux. The documentation is in the form of a class-reference indexed web page.

LibTomCrypt The LibTomCrypt, LTC, library is a public domain library with extensive documentation. It supports several symmetric ciphers and hash functions. There is support for the most common asymmetric algorithms, such as RSA, DSA, ECC and Diffie-Hellman. LibTomCrypt is written completely in standard C and features support for several platforms, including embedded platforms.

Botan The Botan cryptographic library is written entirely in C++ and supports a wide variety of algorithms and standards. There is support for several symmetric algorithms including the five AES-contest finalists. Botan also supports several of the asymmetric cryptographic algorithms and schemes. However, there is no support for ECC and is thereby no longer interesting for this project.

The decision was to use the extensively documented LibTomCrypt library, mostly because it was written entirely in portable C and contained support for all the required cryptographic algorithms.

LibTomCrypt contains no built-in support for the large integer arithmetic needed by the asymmetric cryptography. This is to make it portable and to allow the developer to use the math-library of his/her preference. This requires a decision of which large integer library to use.

[4]Licence is public domain.

4.3.4 Large integer library

A large integer library is required since the processor only supports arithmetic calculations of up to 32 bit integers. The large integer library enables the support of operations performed on the large integers required by the asymmetric cryptography ($\approx$ 4000 bits). As an example multiplying a key of size n, this requires a 2n precision of the result. The largest RSA key considered in this implementation is 4096 bits, requiring the software to handle 8192 bit integers.

The documentation of LibTomCrypt refers to two other math libraries, developed by the same author[5], for use with LibTomCrypt. Both libraries listed below will be implemented and compared on the test-platform.

LibTomMath LibTomMath, LTM, is an efficient and well documented code library for manipulating large integers. It is written in C-language and is highly platform independent. The code is well commented, and although there are faster libraries available, LibTomMath is easy to configure and optimize.

LibTomMath is written as a multiple precision integer library, MPI, which supports manipulating integers of variable precision. This means, if the resulting precision of an arithmetic operation is larger than the output variable's current precision, the output variable is expanded to fit the extra precision. However, this type of big integer requires the most computational overhead of any style of arithmetic[6].

The library implements several specialized reduction functions such as Toom-Cook, Karatsuba, Comba or baseline multiplication. LTM automatically decides which reduction algorithm to use based on the magnitude of the inputs.

The library was written mostly for educational purposes, and features the accompanying textbook *Implementing Multiple Precision Arithmetic*[24] and also a user guide written by the author of the project.

TomsFastMath TomsFastMath, TFM, is a fast and efficient large integer arithmetic library. It is optimized for fast modular exponentiations which is used in asymmetric cryptography implementations such as; RSA, ECC and DH. The

[5]Tom St Denis.

[6]This mentioned in the book *BigNum Math* By Tom St Denis [24].

code is written mostly in ISO C but also contains some inline-assembler operations to speed up processor specific operations. It is highly configurable with predefined macros for ECC and also contains ports to several processors, including the ARM7 processor core.

TomsFastMath is a fixed precision large integer arithmetic library, this means it handles all integers with the maximum precision. If the result of an operation requires more precision than the maximum, the extra precision is lost. The maximum precision is defined at compile-time and cannot be changed.

4.4 Key device

A study of suitable devices which could replace the traditional key in an access system was done. The purpose of this study was to determine which key device would be suitable when designing a new access system.

For this specific system the iButton product series from Dallas Semiconductor was chosen and implemented. The iButton device is a small robust container which communicates by touching it to a reader. Communication is done via a 1-wire protocol where the device is powered from the same line it communicates on, thus requiring only two electric connections.

Some of the built-in security functions in the iButton are; each device contains a unique 64 bit on-chip laser written serial number, some models have a built-in challenge response algorithm and several devices contain password protected memory.

CHAPTER 5

Implementation

The implementation is meant as a study of whether it is possible to perform asymmetric cryptography operations on an embedded platform or not, and to which extent.

This chapter describes the implementation of the free open-source library LibTomCrypt on the development board AT91SAM7XC-EK. It also features a basic overview of the hardware used and test benchmarking procedure. Some additional design features are implemented on the board to enable a smooth transition to the implementation of the complete access system concept.

5.1 Hardware

The hardware of the platform uses the AT91SAM7XC-EK development board as a base, which is described in section 4.3.1.

5.1.1 Processor

The development board features an AT91SAM7XC256[1] processor which is based on a 32 bit ARM7 RISC[2] core. The AT91SAM7XC256 processor executes at 60 MIPS with 256 KiB FLASH- and 64 KiB RAM-memory. It features additional cryptographic support modules, specialized for AES and 3DES cryptography.

[1] Further reading and specifications of the processor at ATMEL's website [25].
[2] Further reading and specification of the core at the ARM's website [26].

Also several communication modules such as USB, USART[3] and Ethernet are included on the board.

The programming of the processor is done using the JTAG-interface available on the development board. Communication with the processor is mainly done using the debug serial interface and portable memory mediums.

5.1.2 Implemented perhiperals

A number of interfaces were implemented on the development board in order for the system to be able to communicate properly. Here is a quick summary of the perhiperals implemented on the board:

Real time clock A RTC, was determined useful for a concept implementation of the access system. It was implemented using the MT41T00 external clock circuit, from STMicroelectronics, connected to a 32 kHz crystal oscillator. The clock circuit connects to the TWI port on the processor and is interfaced using a modified version of drivers provided by Atmel.

Secure digital flash card The SD-card provides an easy way of updating the large amounts of data used in the system. It is mostly used to load key-, plaintext- and ciphertext-data into the program. The results of the cryptographic operations and system logs can also be stored on the card. Since the SD-card is portable the data can be imported and analyzed on a PC. The hardware implementation is on a built in card-connector on the development board, the connector interfaces to the SPI interface on the processor.

Dallas 1-wire Support for Dallas 1-wire devices was included since the concept solution is based on the Dallas iButton device. The interface implementation was done by connecting the Dallas DS9097U serial 1-wire adapter to the serial port mounted on the development board.

[3]Universal Asynchronous Receiver/Transmitter.

Door motor control A simple motor driver is implemented by I/O control, which can be used to drive a standard electronic lock cylinder. This is useful for demonstration purposes of the concept implementation.

5.2 Software

The code is written in standard C language and is compiled using the arm-elf (GCC) toolchain, version 4.1.1. The program processes simple tasks and is constructed using a simple round-robin scheduler which uses the interrupt on the timer-module to generate a system clock tick variable, incrementing at 991 Hz.

5.2.1 Cryptographic algorithms

The LibTomCrypt library was used to implement support for the cryptographic algorithms required. The library is highly portable, written completely in standard C.

The library is easy to understand and configure, it provides a simple configuration header file where the programmer can enable/disable support for each algorithm separately. Thus including only support for the specific desired algorithms when compiling the code, reducing memory usage. This is useful when benchmarking the algorithms, since there is limited memory available on the processor.

The main difficulty with implementing the cryptographic routines was the limitations in memory. LibTomCrypt requires about 75 KiB for ECC and 40 KiB for RSA.

5.2.2 Large integer arithmetic

LibTomCrypt requires a large arithmetic library when performing asymmetric cryptography. There are two libraries which are implemented to test the performance differences in the large integer arithmetic. The two libraries, LibTomMath, LTM, and TomsFastMath, TFM, have different approaches of how to

implement the support for the large integers and performing the calculations. Which is shown by analyzing the codesize of the libraries.

The codesize of the LTM library was fairly constant at about 57 KiB, input size only limited by available RAM. The TFM library was harder to implement since the codesize varies depending on the maximum size of the inputs, therefore the library was only successfully implemented for ECC.

5.2.3 Filesystem library

The *Embedded Filesystems Library*, EFSL, which is an open-source library with support for Microsoft filesystems is used, enabling easy support for reading and writing to files on Secure Digital memory cards. Thus the data and keys to be tested on the test-platform can be easily edited on a PC and then imported into the embedded system. The library implementation requires about 1.5 KiB RAM- and 35 KiB FLASH-memory.

5.2.4 Memory overview

The code implementation of the routines required on the testplatform, excluding the cryptography library is shown in table 5.1. It uses a total of 102 KiB of flash memory. Subtracting this from the 256 KiB of available flash memory leaves a maximum of 154 KiB code to be used when testing the cryptography- and math-libraries.

5.3 Benchmarking goals

The goal of the test-platform is to provide benchmarked data on the various cryptographic algorithms suitable for implementation in an access system. A background of which algorithms were desired to implement is found in section 3.6. Some of the desired algorithms had to be skipped since they are not supported by LibTomCrypt.

Code sizes of different implementations

Task	Code size [B]
Basic Program	18952
Real Time Clock	8676
iButton	4056
SD-card	33524
Libc	31609
LibGCC	4928
Total	**101745**
Availible memory	**256000**
Free memory	**154255**

Table 5.1: *The table shows the memory required by the different applications implemented. These figures are exclusive the cryptographic library. The available memory is 256 KiB and with the listed applications there are approximately 154 KiB remaining.*

Symmetric algorithms The table 5.2 describes which symmetric algorithms will be tested. The comparison is between LibTomCrypt and where applicable the processor specific cryptographic hardware modules.

Testchart for symmetric cryptographic algorithms

Algorithm	Block size [bits]	Key size [bits]	LibTomCrypt	Hardware Module
AES	128	128	x	x
AES	128	256	x	-
DES	64	56	x	x
3DES	64	168	x	x
Twofish	128	128	x	-
Twofish	128	256	x	-

Table 5.2: *The table shows which symmetric algorithms are to be tested and if they are going to be implemented in the hardware accelerator.*

Asymmetric algorithms When comparing the asymmetric algorithms, the results are dependent on a wide variety of configuration parameters. There are dozens of parameters to configure and to find the optimal configuration would require a large amount of time and testing. However, the purpose of the test is to compare how the two technologies ECC and RSA perform in relation to each other. To do this accurately, the test of each technology will be performed using both available large integer arithmetic libraries with as similar configuration as possible. The tests to be performed are presented in table 5.3.

Testchart for asymmetric cryptographic algorithms

Algorithm	Key size [bit]	LibTomMath	TomsFastMath
ECC	112 - 521	x	x
RSA	1024 - 4096	x	x

Table 5.3: *The table shows the symmetric algorithms and libraries that are going to be implemented.*

5.4 Measurements

Several measurements will be done when benchmarking the algorithms. The implementations will be analyzed regarding two factors; execution time and memory consumption.

Execution times The execution time will be measured in two different ways. For the asymmetric cryptography the timing will be done using a count variable, which is incremented by the 991 Hz clock timer interrupt. The result will be obtained by reading and rescaling the value of the variable. Therefore the accuracy of the result is $t_{measured} \pm 1.01mS$.

For the considerably faster symmetric cryptography operations, the start and stop operations were indicated on an external I/O-pin, and measured using an oscilloscope. The readout of the oscilloscope has a precision of three digits, therefore the accuracy is $t_{measured} \pm 0.05mS$.

Memory analysis The memory consumption of the algorithms is analyzed from the linker-file produced by the development environment. This linker-file provides information of the required memory for each source file. This analysis can be done rather accurately since LibTomCrypt has a separate source file for each algorithm.

CHAPTER 6

Results

The results of benchmarking different algorithms on the test-platform, in both symmetric and asymmetric cryptography, are presented in this chapter.

6.1 Symmetric algorithms

The symmetric algorithms tested on the test platform were AES, DES, 3DES and Twofish. The measurements of the different algorithms done using 10 KiB data and will be presented in the unit $[ms/KiB]$.

6.1.1 Symmetric encryption and decryption

Both encryption and decryption routines were implemented on the board for the measurements on the symmetric algorithms. The test was performed using several keysizes for the algorithms, wherever this was possible. The software algorithm test, was based on the cryptographic library LibTomCrypt and did not require any mathematical library. The library is configured for size rather than speed wherever possible.

For AES, DES and 3DES it was possible to use the hardware acceleration modules on the processor. This was also tested and the results are listed as a comparison in table 6.1.

Note that by using the hardware implementation in all three algorithms, there was a large improvement in speed. The AES cipher was 16.1 times faster than the software implementation, and 3DES was 42.7 times faster. The hardware

Symmetric encryption and decryption

			LibTomCrypt		Hardware module	
Algorithm	Block size [bits]	Key size [bits]	Encryption [ms/KiB]	Decryption [ms/KiB]	Encryption [ms/KiB]	Decryption [ms/KiB]
AES	128	128	9.2	9.2	0.57	0.57
AES	128	256	11.5	11.5		
DES	64	56	15.0	15.0	0.55	0.55
3DES	64	168	37.6	37.6	0.88	0.88
Twofish	128	128	9.1	9.1		
Twofish	128	256	9.1	9.1		

Table 6.1: *The table shows the encryption- and decryption time taken, for the listed symmetric ciphers, to compute 1 KiB.*

implementation improved the 3DES cipher significantly more. In the software implementation, AES was 4.1 times faster than 3DES, this can be compared to the hardware module, where AES only was 1.5 times faster.

Further results when comparing the symmetric ciphers, are the memory requirements of the different algorithms. These results are presented in table 6.2. It should be noted that to simplify portability within the library all ciphers requires the same 3.3 KiB of RAM.

Symmetric algorithm's different memory requirements

Symmetric cipher	Memory requirements of the algorithm's code [Byte]	Memory requirements of the algorithm's tables [Byte]	Total FLASH required [Byte]
AES	7304	4436	11740
DES/3DES	6028	2588	8616
Twofish	9888	392	10280

Table 6.2: *The table show symmetric algorithm's different memory requirements.*

The amount of memory each cipher requires is a measurement of the efficiency of the implementation. The DES cipher require the same memory size as the 3DES which is expected since the 3DES algorithm consists of running the DES

cipher three times. The small amount of predefined tables of the Twofish cipher compared to AES is because it uses an algorithm to compute the S-box values, this is slower but reduces the memory required.

6.2 Asymmetric algorithms

In the test of the asymmetric algorithms, RSA was compared to ECC. Due to the large amount of memory required by the asymmetric cryptography there was insufficient memory left to implement a PRNG, which is required to perform encrypt- and signature-operations. Therefore the implemented operations were limited to decryption and verification. The two different algorithms were benchmarked with the previously determined key sizes. ECC was also implemented with the two different mathematical libraries, LibTomMath and TomsFastMath. The TomsFastMath library could not be used in combination with RSA due to lack of memory.

6.2.1 Memory requirements

The results of the memory requirements of the different configurations used for the asymmetric cryptography tests, are shown in table 6.3. The LibTomCrypt library requries about 73 KiB for ECC operations and about 39 KiB for RSA operations, this includes support for a single symmetric algorithm (AES) and a hash function (SHA-256) required by the library for asymmetric operations. As mentioned previously, the large integer libraries also require a large amount of memory. The LTM library require 56.2 KiB of memory and the TFM library size depends on the maximum input size it is configured for. An input size of 521 bits for ECC results in 57.5 KiB and increases to 162.5 KiB for 1024 bits RSA. Since the remaining code memory on the processor is 154 KiB, the latter implementation is infeasible.

6.2.2 Asymmetric decryption

The two following tables, 6.4 and 6.5, shows the different decryption times for ECC and RSA respectively using their private keys. The data used in this test

Memory requirements

Configuration	Library	Flash memory [Byte]
ECC + LTM	LibTomCrypt	73140
	LibTomMath	56148
	Total	**129288**
ECC + TFM	LibTomCrypt	70288
	TomFastMath	57504
	Total	**127792**
RSA + LTM	LibTomCrypt	38460
	LibTomMath	56148
	Total	**94608**
RSA (max 1024) + TFM	LibTomCrypt	38460
	TomFastMath	162492
	Total	**200952**

Table 6.3: *The table shows the amount of Flash memory required by the different configurations used for the asymmetric cryptography. The amount of available memory is approximately 154 KiB.*

was a symmetric key of 128 bits, encrypted by the corresponding public key. In the first table, describing ECC decryption, both LibTomMath and TomsFastMath libraries are used and compared to each other.

The improvement factor of implementing the TomsFastMath library is increasing with the security level. The largest improvement is therefore with the 521 bit key, resulting in an improvement factor of nearly 4 times faster decryption.

Decryption in ECC

Key size [bits]	LibTomMath Decryption time [Sec]	TomsFastMath Decryption time [Sec]	Improved by factor
ECC 112	0.808	0.765	1.06
ECC 128	0.980	0.834	1.18
ECC 160	1.233	0.938	1.31
ECC 192	1.570	1.064	1.48
ECC 224	1.972	1.187	1.66
ECC 256	2.829	1.375	2.06
ECC 384	6.329	2.277	2.78
ECC 521	12.940	3.248	3.98

Table 6.4: *The table shows the decryption time taken for different keysizes in both LibTomMath and TomsFastMath.*

For the RSA decryption only keysizes of up to 3072 bits was possible, 4096 bit key failed due to insufficient RAM. It was impossible to use TomsFastMath or any other software acceleration for RSA decryption due to the limited Flash memory.

6.2.3 Verification

The table 6.6, show the results of verification with ECDSA. In this test the results are presented from testing the mathematical libraries TomsFastMath and LibTomMath. Further it presents the improvements by adding Shamir's trick, which is described in the paragraph below. The implementation of both Shamir's trick and TomsFastMath was only possible up to the ECC keysize of

Decryption in RSA

Key size [bits]	Decryption time [Sec]
RSA 1024	2.694
RSA 2048	14.734
RSA 3072	44.274

Table 6.5: *The table shows the decryption time in RSA. Only three different keysizes were possible to implement.*

128 bits, this is due to the fact that Shamir's trick stores precomputed large integers, and in TFM all integers are represented with the same precision. Since the LTM library only stores the required precision, the same number of integers require less RAM memory.

Shamir's trick Multiplication of points on the elliptic curve is done by additions and point doubling. Addition arithmetic is much more complex to perform than point doubling. Therefore, this trick is based on precomputing addition results, to be used when needed. Using this trick the performance can be increased by using more memory. This only affects the verify operation since it requires two multiplications.

With the limited amount of memory it was not possible to accelerate the RSA verification with any configuration except the most basic. The results are shown in table 6.7 and this shows that the verification is much faster than the corresponding times for decryption with RSA.

Verification in ECC

Key size	With LibTomMath		With TomFastMath	
Shamir's Trick	**No**	**Yes**	**No**	**Yes**
[bits]	[Sec]	[Sec]	[Sec]	[Sec]
ECC 112	0.809	0.588	0.727	0.535
ECC 128	1.103	0.782	0.827	0.595
ECC 160	1.612	1.147	1.042	Failed
ECC 192	2.300	1.603	1.309	Failed
ECC 224	3.246	2.206	1.626	Failed
ECC 256	4.761	3.310	1.905	Failed
ECC 384	11.665	8.056	3.428	Failed
ECC 521	24.948	17.226	5.646	Failed

Table 6.6: *The table shows the verification time taken for different keysizes in ECC in both mathematical library together with Shamir's trick wherever possible.*

Verification in RSA

Key size	Verification time
[bits]	[Sec]
RSA 1024	0.191
RSA 2048	0.665
RSA 3042	1.378

Table 6.7: *The table shows the verification time in RSA.*

CHAPTER 7

Discussion

Within this section a discussion of the thesis project is presented. The decisions and research will be analyzed, concluding with a discussion of the archived results.

7.1 Access systems

In this thesis work a study of access systems was done. Existing systems were analyzed to determine functionality, weaknesses and possible improvements. From the results of this study a concept solution for a new access system was designed. However, due to the pending patent application, the results regarding the study of access systems will not be further discussed in this public report.

7.2 Security

As mentioned in section 2.1.3, analyzing the security of an access system is a very difficult task. There are a great number of factors affecting the security of the whole system. This includes a large number of possible attack angles such as; physical- , electrical- , hardware- and cryptography-security. This task is even more complicated when considering the human factor, the actual users and administrators of the system. Therefore, from the entire system point of view, the choices of cryptographic algorithms and keysizes play a small role in

the security of the whole system.

However, in this thesis the focus is on the cryptography and how it can be implemented to solve the problem of key distribution. Therefore, only the cryptographic part of the access system will be considered in the scope of this report.

7.3 Cryptography choices

The choices of which algorithms to be implemented in this thesis work were mostly based on recommendations from the National Institute of Standards and Technologies, NIST. They publish US Federal Information Processing Standards, FIPS, containing recommendations of which cryptographic techniques to use and how to implement them. The recommendations apply to encrypted information within the US Government. When analyzing the security level between different cryptography technologies, the comparison is based on the work factor for the algorithm with the corresponding keysize. A comparison table is found in table 7.1, and includes data from the NIST recommendation documents.

Work factor for different technologies

Bits of security [bits]	Symmetric key size [bits]	RSA key size [bits]	ECC key size [bits]
80	80	1024	160
112	112	2048	224
128	128	3072	256
192	192	7680	384
256	256	15360	512

Table 7.1: *Work factor based (Security bits) comparison of keysizes with different cryptographic technologies.*

Recommended keysizes The lifespan of a cryptographic algorithm can be related to its work factor, and is determined by making estimates of future

computational progress. The estimates are produced using methods such as Moore's law, and therefore they are vulnerable to unexpected large cryptanalysis breakthroughs. It is therefore important to consider them only as estimates and guidelines, which could turn out to be quite unreliable. The currently recommended keysizes as predicted by NIST are presented in table 7.2.

Expected Key Lifetime

Bits of security [bits]	Algorithm security lifetime	RSA [bits]	ECC [bits]
min. 80	Trough 2010	1024	160
min. 122	Trough 2030	2048	224
min. 128	Beyond 2030	3072	256

Table 7.2: *The table show the keysizes recommended by NIST.*

7.3.1 Symmetric algorithms

The ciphers which were evaluated where chosen based on both recommendations from standardization institutes and from what was available in the library used for the implementation. AES and 3DES are recommended by NIST for use in new designs and was therefore chosen. Further, as a comparison, the Twofish cipher was included.

For symmetric algorithms the *work load* mostly defined as the resources required to do a bruteforce-attack on the algorithm, with compensation for *birthday attacks*. This is valid for most the unbroken ciphers. Most standards recommended a minimum keysize of 128 bits for all new implementations of symmetric ciphers. Although, it might seem unnecessary with 128 bits for most applications, it is reasoned that bits are cheap to implement in symmetric ciphers. The Rijndael cipher was standardized into AES by winning the AES Standardization competition. However, the main reason it won was that it requires few resources and can be implemented on small platforms. The strongest attack on AES, breaks an encrypted message, using 7 rounds of the specified 10 rounds

for the keysize of 128 bits. This is considered by some to be a rather small security margin, regarding future cryptanalysis breakthroughs. There were several other candidates which were considered to be more secure, by the AES jury, such as Twofish [1]. This is why Twofish was included in the benchmark.

The 3DES cipher was proposed as a solution after the previous DES cipher was broken, and has been recommended since 1991. It comprises of 3 rounds of DES and uses a 168 bit key. 3DES has become less common in new designs since the AES cipher is considerably faster.

Modes of operation NIST recommends several different modes of operation for block ciphers. Moreover, Bruce Schneier and Niels Ferguson provides a great summary of the modes in their book *Practical Cryptography* on page 77 [4]. The two finalists in their comparison was CBC- and CTR-mode, the main difference between them is how much information of the plaintext is leaked if the *Nonce* is used twice. Finally, they recommend CTR-mode as it is simpler to implement and does not require padding.

7.3.2 Asymmetric algorithms

In asymmetric cryptography there are two major different types of algorithms, RSA or ECC. They are quite different regarding the mathematics behind them, although the applications are the same. The main difference is the keysize. In table 7.3 showing the different recommendations based on security level, shows that the ECC keysizes are comparably much smaller.

RSA The RSA algorithm is still recommended by NIST although the large keysize makes the use of the algorithm computationally slow and memory consuming. It should be noted that RSA has been used for a longer time within asymmetric cryptography, and therefore more cryptanalysis research had been done on the algorithms and math. Therefore, RSA is considered the safe conservative choice compared to the newer ECC algorithms.

[1] Mentioned in Bruce Schneier's book *Practical Cryptography* page 65 [4].

Elliptic curve cryptography Certicom is one of the major advocators of the elliptic curve cryptography. They claim that, although the use of ECC is rather new, the mathematical research behind it, is 150 years old. However, according to Bruce Schneier, this is a modified truth because the research behind the mathematics are not from a cryptanalysis- but mathematical- viewpoint[2].

The use of ECC has increased over the last few years and is still advancing. There is an increasing amount of documentation available on the subject and there are several organizations publishing standards of ECC parameters, such as; National Institute of Standards and Technology (NIST), The American Standard Institute (ANSI), and The Institute of Electrical and Electronics Engineers (IEEE).

To achieve the desired security level, when implementing ECC, it is important to use approved elliptic curve parameters. NIST are recommending curves based on the two different groups, prime and binary groups. LibTomCrypt only supports curves defined in the NIST prime-group.

Security level		
Bits of security [bits]	RSA key size [bits]	ECC key size [bits]
80	1024	160-223
112	2048	224-255
128	3072	256-383
192	7680	384-511
256	15360	512+

Table 7.3: *The table describes the variety of keysizes in ECC for each level of security.*

Most asymmetric systems in use today, uses 1024 bit parameters for Diffie-Hellman and RSA which is the NIST recommendation until 2010[3]. The corresponding ECC key size, with equivalent security, is 163 bit, according to Certicom[4].

[2]This is discussed at Bruce Schneier's website [27].

[3]This is described in NIST's special publication 800-57 [28].

[4]The keysize recommended by NIST and Certicom [29] differs 3 bit.

A competition[5] in cryptanalysis of ECC was introduced in November 1997 by Certicom to stimulate further analysis on the security of elliptic curve cryptography. It was also developed to increase the understanding of the difficulty of the elliptic curve discrete logarithm problem.

One of the challenges, comprising 109 bit key, was solved on November 6th in 2002 utilizing the power of 10.000 computers running 24 hours a day for 549 days. This was done by Chris Monico and his team of mathematicians at Notre Dame. It is approximately a hundred million times harder to break the next challenge, of the 163 bit key size which is the recommended keysize from Certicom.

7.4 Hardware performance

The hardware used in this evaluation was chosen with little prior knowledge of the requirements associated with cryptographic algorithms or computations on large integers. Therefore the choice of hardware was not optimal, the main limitation was the memory. However, the goal of this test implementation was to study the feasibility, not to construct an optimal implementation.

7.4.1 Processor architecture

The ARM7 core of the AT91SAM7XC256 processor is a commonly used and widely available architecture for embedded systems, which made it suitable for this test.

Processor clock The processor speed is the most obvious parameter when choosing processor, it is measured in million instructions per second, MIPS. Increasing the speed of the processor is usually associated with increasing power consumption, therefore the manufacturers implement additional functionality to the processor to speed up the system without increasing the clock frequency. So for the performance of the embedded processor there are several other parameters to consider which have more impact on the performance than the pro-

[5]The competition announced by Certicom is further described at their website [30].

cessor clock frequency.

Architecture The choice of which internal bus-size to use for the implementation has a large impact on the performance. The internal bus-size on the chosen architecture is an important factor, since performing some of the arithmetic operations can be performed more than four times faster on an 32 bit architecture, than on a corresponding 8 bit architecture.

Built-in modules Some processors have additional modules included inside the processor to simplify common tasks, for example a hardware multiplier module can significantly improve processor speed, since an integer multiplication can be done in one processor cycle.

Several manufacturers provide processors with application specific modules, such as symmetric en-/decrypt modules and coprocessors for asymmetric cryptography.

Furthermore, some processor architectures, mostly Digital Signal Processors, DSP, have hardware support for performing fixed point arithmetic. This would speed up calculations considerably, since TFM operates using fixed point calculations. An article, by Fujitsu Laboratories [31], describes implementing various asymmetric algorithms on a Texas Instruments DSP provided results of 3.97 ms for a 160 bit ECDSA verification. Compared to the fastest results achieved in this thesis, it is faster by a factor of 261 times, or 39 times accounting for differences in processor clock speed.

Generally it can be reasoned that the optimal processor architecture for use with asymmetric cryptography is an architecture with support for fixed point operations, alternatively an architecture with a coprocessor for cryptography operations. However, there is always a compromise of cost versus performance, a faster and/or more complex platform is usually more expensive.

7.4.2 Memory

One of the toughest constraints with the implementation on the test platform was the limited memory available. More memory would most likely result in a

much faster implementation and enable larger keysizes to be used.

There are large performance gains to be had by increasing the amount of memory, mostly because the program can reuse previously calculated results by storing them in tables. This also allows for faster and more efficient large integer arithmetic libraries to be used.

Therefore, for a complete evaluation of the LibTomCrypt library on this embedded platform the code should be tested on the AT91SAM7XC512, which has double the RAM- and Flash-memory.

7.4.3 Extension modules

Another approach would be to design an external FPGA with customized VHDL code for accelerating the large integer arithmetic, this is a similar approach to choosing a processor with an embedded coprocessor. Furthermore, some manufacturers provide processors with a builtin programmable FPGA-module for this purpose.

7.4.4 Power consumption

The focus of this testplatform has been on feasibility, not implementing efficiently. However, since a lockunit most likely would be battery-powered, power consumption is a relevant subject. In general, the more performance the processor has, the more power it requires and therefore there is a negative relationship between the performance and battery time.

Most processor manufacturers have enabled certain features to solve this problem, these are mostly; low-power sleep modes, ability to lower clock frequency and disabling unused function modules in the processor. These features should be carefully considered by the designer when choosing which processor platform to use in a commercial product.

7.5 Software performance

The performance of the software is hard to measure, since it has to be properly defined what a feasible implementation is. The goal of this implementation was to achieve as high speed of the algorithms as possible, with the limited resources of the embedded platform. The focus on algorithm speed has resulted in a code size requiring 90% of the total memory, this is probably insufficient for most applications. However, in this implementation this means the algorithm speed results are close to the limit for what is feasible on this platform.

7.5.1 Language and compiler

The program is written mostly in C, only the TFM library uses assembler macros to speed up certain operations. There could probably be a performance gain by further optimizing some of the algorithm code by using assembler. Although, this would require a lot of time and limit the objectivity of the code.

The code was compiled using the *arm-elf-gcc* compiler. To enable debugging support of the code the optimization was turned off. Therefore, there are still performance which could be gained by tuning this parameter. Moreover, another compiler such as the commercial IAR Workbench Compiler, could probably produce more optimized code for the ARM7 core which would give a smaller and more efficient application.

7.5.2 Cryptographic library

The performance impact of the choice of a cryptographic library is hard to measure, another library might have implemented some algorithms differently. The LibTomCrypt library was easy to get started with since it is extensively documented and commented, especially compared to its main contender Crypto++, where only the programmers reference API is provided.

LibTomCrypt uses a simple API with a highly modularized structure, this means that for example the CBC-mode encryption function is independent of which block cipher it is to use. This gives a highly portable code where the cryptographic algorithms can be replaced easily, and support for a variety of algo-

rithms can easily be implemented. The downside is that it requires more overhead resources to support every possible algorithm and therefore gives a larger code size.

There are many tuning parameters for the library, mostly allowing the programmer to decide whether to optimize the algorithms for speed or for size. Also, there are parameters controlling whether the asymmetric cryptography should use faster methods, requiring more memory, such as fixed precision arithmetic or Shamir's trick.

7.5.3 Large integer libraries

There are two large integer libraries evaluated in this project. Both are extensively documented and several macros are enabled providing an easy interface to the LibTomCrypt library.

LibTomMath The LibTomMath library is a MPI library implemented for educational purposes. Therefore it might not provide the fastest routines. However, in this implementation only this library was able to perform the RSA operations with the limited RAM available.

The code-size of the library is the same for all sizes of the integer it performs calculations on. This means the library has the same code size for both ECC and RSA implementations. However, the required RAM memory depends on the size of the input variables, therefore LTM fails to perform calculations on 4096 bit RSA keys due to insufficient amount of RAM memory.

TomsFastMath The TomsFastMath library is a fixed precision integer library including assembler support for processor-platform specific optimizations. Using fixed precision operations with assembler routines results in much faster calculations than LTM.

However, the fixed precision operations require a larger code size than the MPI operations. The required memory is heavily dependent on the maximum precision which the library has to support, since every calculation is done with integers of the maximum precision.

When compiling TFM with ECC support (max 521 bit input integers), the resulting library is smaller and much faster than LTM. The result of enabling support for the 1024 bit input integers required for the smallest RSA key, compiles into a three times larger (163 KiB) codesize than LTM. The RSA configuration also requires more RAM due to the larger integers. The ARM7 processor only has 154 KiB of code memory remaining therefore using TFM with RSA was not tested.

This library would probably speed up RSA operation as much or even more than ECC if there were more memory available on the processor.

7.6 Implementation performance

7.6.1 Symmmetric performance

The results of the execution times of the symmetric algorithms implemented on the testplatform are presented in table 6.1 on page 62.

There is no difference between encryption and decryption done by the same algorithm. This is due to symmetry in the structure of the encryption and decryption algorithms.

Both AES and Twofish had very similar performance and codesizes in this implementation. From a cryptanalysis viewpoint could be argued that the Twofish cipher is more secure, this was mentioned in section 7.3.1. However, the AES cipher is the recommended standard cipher and will therefore receive more public cryptanalysis attention, resulting in public exposure if its security is compromised. Therefore the AES cipher is the better choice.

There were large performance gains by using the hardware acceleration modules compared to the software implementation. The 3DES algorithm was improved by a factor of 42 and AES improved by a factor of 16. Using the hardware acceleration module the AES cipher was by far the fastest cipher in the implementation.

7.6.2 Asymmetric performance

The choice to only implement asymmetric decryption and verification was made since they do not require a random number generator to be implemented on the platform, thus some memory could be saved.

A summary of the test results from the fastest implementation of the respective algorithm is shown in table 7.4

Time comparison of RSA vs ECC

Bits of security [bits]	RSA key size [bits]	ECC key size [bits]	Best RSA dec. time [Sec]	Best ECC dec. time [Sec]	Best RSA ver. time [Sec]	Best ECC ver. time [Sec]
80	1024	160	2.694	0.765	0.191	1.042
112	2048	224	14.734	1.187	0.665	1.626
128	3072	256	44.274	1.375	1.378	1.905
192	7680	384	-	2.277	-	3.428
256	15360	521	-	3.248	-	5.646

Table 7.4: *The table shows the differences in decryption- and verification time between the two algorithms within the same bits of security. Note the large amount of time the RSA algorithm requires when the security is 128 bits.*

RSA requires the longest computational time for decryption while it is rather efficient on verification. This difference is dependent on which of the keys are public respectively private, by switching the keys, the decryption would be faster while verification would be slower.

For ECC verification requires more time than decryption, this is due to the fact that verifications requires two point multiplications while decryption only requires one. The speed of verification can be increased by using Shamir's Trick.

Due to the limitations in memory on the embedded system, there were fewer available improvements for RSA than for ECC, therefore the comparison could not be completely fair.

To be able to benchmark the algorithms against each other they were compared by the sum of decryption time and verification time. A comparison of the calculated sums for each algorithm at each security level is shown in table 7.5.

This table clearly shows that ECC gives much faster performance and delivers higher security bits.

Decryption- and verification time

Work Factor [bits]	RSA [Sec]	ECC [Sec]
80	2.89	1.81
112	15.4	2.81
128	45.7	3.28
192	-	5.71
256	-	8.89

Table 7.5: *The table present the summation of both decryption- and verification times for ECC and RSA.*

CHAPTER 8

Conclusions

This chapter presents the final conclusions of the thesis work, which consisted of implementing cryptographic algorithms on a ARM7 based test platform. Here the conclusions from the benchmarks of the cryptographic algorithms will be presented, including a summary of the constraints related to the limited performance of the embedded system. Furthermore, some suggestions for future hardware and software improvements will be discussed.

8.1 Implementation feasibility

The aim of this thesis was to answer the question of weather it is feasible to implement advanced cryptography on an embedded platform to be used in an access system, and if the resulting performance would be usable.

The comparison of ECC and RSA in this implementation showed that it is feasible to implement asymmetric cryptography algorithms with very high security, RSA keysizes up to 3076 bits and ECC keysizes up to 521 bits, on an embedded platform. The ECC algorithm outperformed the RSA implementation with a large margin. ECC with a 521 bit key was able to do both decryption and verification in a very reasonable time of 10 seconds.

The same operations on RSA with a 3076 bit key required 45.7 seconds, this compares in security to a 256 bit ECC key which only required 3.3 seconds.

The recommendation for a future product where asymmetric cryptography is required and the implementation is software based is to use an ECC imple-

mentation rather than RSA. This is since the algorithm is clearly faster and requires less memory to implement on embedded platforms, also enabling use of a higher security level than is possible with RSA. The smaller keysizes also reduce the size of the data to be transfered in the system resulting in smaller blocks of encrypted data and signatures.

There are several devices customized with RSA coprocessors on the market which probably could be used to accelerate the RSA calculations and thereby gain an advantage over the software ECC implementation.

From this study it is clear that the symmetric ciphers play a smaller role of the overall performance of the system than the asymmetric algorithms. However, it is clear that processor accelerators for symmetric ciphers offer a performance gain in the magnitude of 10-20 times faster symmetric encryption/decryption and reduces the codesize considerably.

8.2 Further studies

There are several improvements that could be made in a future implementation such as; further optimizing the configurations of the cryptographic and mathematical libraries, also configuring and/or purchasing a better compiler.

Choosing a different embedded platform would have a large impact on performance where there are several items which should be considered as a future improvement:

- **Memory** Increasing the available memory, both RAM and Flash, would increase the performance of the large integer arithmetic libraries.
- **Hardware module** Using a processor with an accelerator for large integer arithmetic would probably significantly increase the performance of both ECC and RSA.
- **Processor architecture** Switching to a fixed point processor architecture such as a DSP which has support for hardware multiplications would probably provide the ideal platform for the ECC cryptographic algorithms.

List of Figures

List of Tables

References

[1] Linus Fredriksson and Martin Gyllensten. Modellering av delvis kända bilder med hjälp av bilder. URL: *http://www.divaportal org/diva/getDocument?urn_nbn_se_liu_diva-6724-1__fulltext.pdf*, 2007.

[2] Securitas AB. Passersystem - generellt om passersystem. URL: *http://www.securitassystems.se/97888027-1404-4b55-b007-5c51bb545b26.fodoc*, 2007.

[3] Bruce Schneier. Attack trees - modeling security threats. URL: *http://www.schneier.com/paper-attacktrees-ddj-ft.html*, 1999.

[4] Bruce Schneier Niels Ferguson. Practical Cryptography. Number ISBN 0-471-22357-3.

[5] David Kahn. The Code-Breakers. Number ISBN 0-684-83130-9.

[6] Keshava P. Subramanya. Brute force searches in cryptography. URL: *http://www.cs.ucsb.edu/~keshava/bruteforce/bruteforce.html*, 2007.

[7] National Institute of Standards, Technology, and the agency of Commerce Department´s Technology Administration. Commerce department announces winner of global information security competition. URL: *http://www.nist.gov/public_affairs/releases/g00-176.htm*, 2007.

[8] Federal Information Processing Standards Publications. Announcing the advanced encryption standard (aes), fips 197. URL: *http://www.csrc.nist.gov/publications/fips/fips197/fips-197.pdf*, 2001.

[9] National Institute of Standards and Morris Dworkin Technology. Recommendation for block cipher modes of operation methods and techniques, nist special publication 800-38a. URL: *http://csrc.nist.gov/publications/nistpubs/800-38a/sp800-38a.pdf*, 2007.

[10] NSA National Security Agency. Announcing the secure hash standard. URL: *http://csrc.nist.gov/publications/fips/fips180-2/fips180-2withchangenotice.pdf*, 2002.

[11] William Stanley Jevons. The Principles of Science: A Treatise on Logic and Scientific Method. 1874, 2nd ed. 1877, 3rd ed. 1879.

[12] Martin E. Hellman Bailey W. Diffie. New directions in cryptography. URL: *http://www.cs.rutgers.edu/~tdnguyen/classes/cs671/presentations/Arvind-NEWDIRS.pdf*, 1976.

[13] RSA Laboratories. What is the factoring problem? URL: *http://www.rsa.com/rsalabs/node.asp?id=2189*, 2007.

[14] RSALaboratories. Rsa algorithm. URL: http://www.rsa.com/rsalabs/node.asp?id=2146, 1977.

[15] RSA Laboratories. Pkcs #1 v2.1: Rsa cryptography standard. URL: *ftp://ftp.rsasecurity.com/pub/pkcs/pkcs-1/pkcs-1v2-1.pdf*, June 14, 2002.

[16] Alexander Bogomolny. Euclid's algorithm from interactive mathematics miscellany and puzzles.
URL: *http://www.cut-the-knot.org/blue/Euclid.shtml*, 2007.
[17] Federal Information Processing Standards Publications. Announcing the digital signature standard (dsa), fips 186-2.
URL: *http://csrc.nist.gov/publications/fips/fips186-2/fips186-2-change1.pdf*, 2000.
[18] Certicom. Ashort history of ecc.
URL: *http://www.certicom.com/index.php?action=ecc,about_ecc*, 2007.
[19] National Institute of Standards and Technology. Recommended elliptic curves for federal government use.
URL: *http://csrc.nist.gov/groups/ST/toolkit/documents/dss/NISTReCur.doc*, 1999.
[20] Dr Jean-Yves Chouinard. Design of secure computer systemscsi4138/ ceg4394 - notes on elliptic curve cryptography.
URL: *http://www.site.uottawa.ca/~chouinar/Handout_CSI4138_ECC_2002.pdf*, 2007.
[21] Bengt Werstén. Implementing the transport layer security protocol for embedded systems. URL: *http://www.divaportal.org/diva/getDocument?urn_nbn_se_liu_diva-8767-1__fulltext.pdf*, 2007.
[22] Gordon E. Moore. Cramming more components into integrated circuits. Electronics Magazine: Volume 38, Number 8, April 19, 1965, 2007.
[23] Atmel. At91sam7xek user guide.
URL: *http://www.atmel.com/dyn/resources/prod_documents/doc6195,* 1973.
[24] Tom St Denis. BigNum Math. Number ISBN 1-597-49112-8.
[25] ATMEL. At91sam7xc256, - description and datasheets.
URL: *http://www.atmel.com/dyn/products/product_card.asp?part_id=3798*, 2007.
[26] ARM The Architecture for the digital world. Arm7tdmi, arm 32-bit risc core at 16-bit system cost.
URL: *http://www.arm.com/products/CPUs/ARM7TDMI.html*, 2007.
[27] Bruce Schneier. Elliptic curve public-key cryptography.
URL: *http://www.schneier.com/crypto-gram-9911.html*, 1999.
[28] National Institute of Standards and Technology. Recommendation of key management - part1: General (revised), nist special publication 800-57.
URL: *http://csrc.nist.gov/publications/nistpubs/800-57/SP800-57-Part1.pdf*, 2000.
[29] Certicom. The basics of ecc.
URL: *http://www.certicom.com/index.php?action=ecc,ecc_faq*, 2007.
[30] Certicom. Certicom announces elliptic curve cryptography challenge winner.
URL: *http://www.certicom.com/index.php?action=company,press_archive&view=307*, 2007.
[31] Naoya Torii Syouji Temma Yasushi Kurihara Kouichi Itoh,Masahiko Takenaka. Cryptographic hardware and embedded systems: First international workshop, ches'99, worcester, ma, usa, august 1999.

[32] TomDenis. Libtomcrypt project.
URL: *http://libtom.org/?page=features&newsitems=5&whatfile=crypt*
[33] Ross Anderson.
Serpent - A Candidate Block Cipher For Advanced Encryption Standard.
URL: *http://www.cl.cam.ac.uk/~rja14/serpent.html*,
[34] Eli Biham Ross Anderson and Lars Knudsen. Serpent:
A Proposal for the Advanced Encryption Standard.
URL: *http://www.cl.cam.ac.uk/~rja14/Papers/serpent.pdf*, 1998.
[35] RSA Laboratories. Pkcs #1 v2.0: Rsa cryptography standard.
URL: *ftp://ftp.rsasecurity.com/pub/pkcs/doc/pkcs-1v2.doc*, October 1, 1998.
[36] Certicom. Certicom announces elliptic
curve cryptosystem (ecc) challenge winner.
URL: *http://www.certicom.com/*
index.php?action=company,press_archive&view=121,
2007.
[37] NSA Natioanl Security Agency. The case for elliptic.
URL: *http://www.nsa.gov/ia/industry/crypto_elliptic_curve.cfm*, 2007.
[38] Certicom. Online elliptic curve cryptography tutorial (introduction).
URL: *http://www.certicom.com/index.php?action=ecc_tutorial,ecc_tut_1_0*,
2007.
[39] Communications Security Establishment. Cryptographic algorithms.
URL: *http://www.cse-cst.gc.ca/services/crypto-services/*
crypto-algorithms-e.html,
2007.
[40] the security division of EMC RSA Laboratories. What are rc5 and rc6?
URL: *http://www.rsa.com/rsalabs/node.asp?id=2251*, 2007.
[41] Horst Feistel. Cryptography and computer privacy.
URL: *http://www.apprendreenligne.*
et/crypto/bibliotheque/feistel/index.html, 1973.