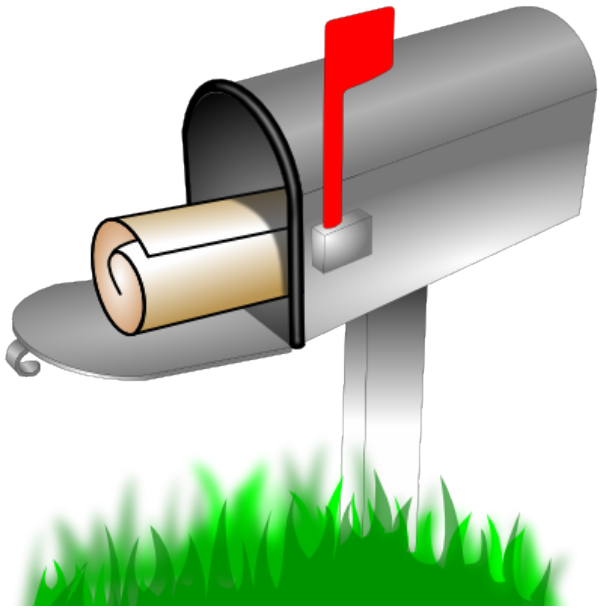
Email security

P1 project - Group B119
Computer science, Aalborg University Fall semester 2008

Authors:
Karsten Jakobsen
Jonas F. Jensen
Anne K. Jensen
Sabrine Mouritsen
Thomas Nielsen

Supervisors:
Simon Kongshøj
Lone Stub Pedersen

Det Ingeniør-, Natur- og Sundhedsvidenskabelig Basisår
Aalborg Universitet



AALBORG UNIVERSITET

Det Teknisk-Naturvidenskabelige Basisår
Elektronik og Elektroteknik
Strandvejen 12-14
Telefon 96 35 97 31
Fax 98 13 63 93
http://tnb.aau.dk

Titel:

Email security

Tema:

Cryptology and numberthoery

Project period:
P1, fall semester 2008

Project group:
B119

Group members:
Anne Kathrine Jensen
Jonas Finneman Jensen
Karsten Jørgensen
Sabrine Mouritsen
Thomas Søndersø Nielsen

Supervisors:
Simon Kongshøj
Lone Stub Petersen

Synopsis:

This report investigates the usability of Thunderbird with Enigmail through a qualitative usability test, with the intent of implementing an end to end encrypting email client. Selected mathematical theorems of RSA are proven in order to create the algorithms needed to digitally sign and encrypt mails. The report does not attempt to change the concept of RSA, but instead shows how encryption can be done without requiring that the user has a deeper understanding of RSA and the math behind it.

Prints: 8

Report page count: 64

Appendix page count: 12

Concluded the 15^{th} of Dec. 2008

Preface

This report is written to document the P1 project of group B119 at Aalborg University. The report is written in English because we, the authors, decided that since most of the books and common terminology regarding cryptography are in English. Therefore, writing the report in English was a logical approach for us.

The report is aimed at readers with little to no prior experience with number theory, RSA and usability - a common first year computer science student. Some topics in this report do require familiarity with programming in both Python and C, however, the sourcecode should in general be fairly well documented, though some parts of it require a solid understanding of the previously mentioned languages.

Obtaining RaptorMail sourcecode

For this project we have implemented an encrypting email client for GMail, with a local storage backend for offline usage and a simple graphical user interface in GTK. The application is written in Python, with minor parts optionally optimised in C.

A CD with the sourcecode and a readme file for running the application and installing dependencies should follow this report. If this is not the case, the sourcecode may be obtained from `http://jopsen.dk/blog/2008/12/RaptorMail-an-encrypting-GMail-client`, where a digital version of this report can also be found. For integrity verification the sha1 hashsum of the tarball is "d87c3aa60ff723a08e5f144314a67518cfbb67e2".

Contents

List of Figures

CHAPTER 1

Project description

1.1 Introduction

1.1.1 Encryption history

Throughout history, there has always been a need to write secret messages. Political and military information could cause severe damage in the wrong hands. The first documented example of a secret text is from 499 BC. Histiaeus, the tyrant of Miletus[1] who wanted to send his son a message. So he shaved the head of a slave and tattooed the message on the top of his head. When the hair grew out again, the slave was sent to Histiaeus' son, with the oral message to have him shaved. That way, if the slave was captured, the attacker would not find the message. Steganography is the name of this type of secret messaging. Steganography can be many things, but the basic definition is to hide the actual message in something which looks innocent. For example, a message with a secret text written in invisible ink. Concealment ciphers are, as the name suggests, also steganography and it is the same system used in pin key reminders.

Another historical messaging system was the Spartans' scytale. It consists of two wooden sticks with the exact same diameter. One of them was given to the commander on the battlefield, and the other to the general in the headquarters. When a message was written, a long strip of paper was rolled around the stick and the message was written on the edges of the paper. The paper was then removed from the stick and sent to the commander. Now only a stick with the same size as the one the general used, could decipher the message. If another stick was used, the letters in the message would not align. And if the enemy were to get their hands on the paper, they would not know the size of stick, the Spartans used, and thus they would be unable to decipher the message. The special thing about this system is that it does not just simply hide the message where the enemy would not be able to find it, but the enemy would not be able to read it unless they also got their hands on a similar stick. The scytale uses transposition to conceal its message. Transposition is when a message is written within the cipher text, but in another order. A simple example of transposition could be writing the message backwards.

Caesar code is, as the name suggests, the way Caesar made sure that the uninitiated would not be able to read classified information. It worked by replacing all the letters with the letter three spots to the right, in the latin alphabet. This resulted in complete nonsense to anyone, who did not know, what they were looking at. However, anyone who knows the system can easily decipher the message. The Caesar code is a substitution system and has a lot in common with modern encryption even though the keys and algorithms are much more complicated than simply moving a few letters [Landrock and

[1] A city in, what is now, Turkey

Nissen, 1997].

1.2 Problem statement

It is not unusual that the Internet traffic between the sender of an email and recepient is routed through many countries, even if they live on the same street. Considering how Internet traffic is routed today, and how emails are used for everyday activities, it is problematic that most of this traffic is unsigned and unencrypted.
According to the simple mail transfer protocol "mail is inherently insecure" quote Klensin [2008, Section 7.1], as anyone with a little technical insight may write an email that appears to come from somebody else, by changing the "from" field. Emails may also be read or altered by anyone routing the mail, this includes intelligence agencies in many countries.

It must be assumed, the average user believes emails to be secure, as it is so widely used. When shopping online, the sellers rarely digitally sign their emails, and thus it can be hard to prove, they were indeed the ones who sent the receipt for a purchase. Emails can be secured, though, with digital signing and encryption.
Stopping people from using emails for shopping and personal purposes is quite unrealistic, and should not be necessary. Instead a better solution could be to make email communication secure. This can be achieved through asymmetric cryptography.

Many email clients support the use of asymmetric cryptography today, such as Thunderbird with the Enigmail add-on. However, it seems that these features have not made any major breakthroughs in the general population. It could be postulated that this is due to the unawareness of the issues, or perhaps that people do not know that these features exist.
After attempting to use one of these programs, it could also be postulated that the problem is caused by usability issues in the applications. If this is the case, an attempt could be made to write a more usable secure email client, that fully supports relevant cryptographic features, so that the everyday user could digitally sign and encrypt their emails with the click of a button, without having to worry about the technical aspects of cryptography.

1.3 Demarcation

In this project we will not investigate the magnitude of the problem of whether or not people use secure email clients in society at large, but only investigate it on a smaller scale. The main focus will be on RSA, and we will not work with other asymmetric algorithms, nor will we work to support existing standards, such as OpenPGP. However, we will study the mathematical aspects of RSA in order to understand its foundation, with the intention of implementing an email client, which encrypts emails. This project will not investigate the various attacks on RSA, though we will introduce tools to work with computional complexity and discuss the complexity of some relevant problems.

Through this project we will also shed light on the possible usability issues of some existing software for secure email correspondence. This will be done in order to assess

which usability issues can be addressed in the implementation of an encrypting email client. We will also discuss this aspect with regards to implementing our own secure email client, and by looking at compexity classes, we intend to look briefly at the safety of RSA.

To sum up, the goal of this project is to study usability issues in Thunderbird with the Enigmail add-on, and understand the mathematical foundation of RSA in order to develop a usable email client, which will be able to encrypt, decrypt, digitally sign and verify signatures. Through the development of this email client, we hope to be able to provide a simple and usable solution to the shortcomings of the simple mail transfer protocol, if not for all email users then at least for a subset of the average email users.

1.4 Method

A relatively small, nonrepresentative questionnaire will be conducted, to investigate people's knowledge of email security. This will give a clue as to whether or not people are aware of the shortomings of the email protocols, and whether or not secure email clients are widely adopted.

We will investigate the usability issues in Thunderbird with the Enigmail add-on, by performing a qualitive usability test on one test subject in a usability lab. The results of this usability test will be analysed and, based on these results, we will gain insight as to what usability issues we need to address, in our implementation of a usable secure email client.

Before developing our own secure email client we will investigate the mathematical foundation of RSA in order to implement it. This will also be done to properly understand which elements can be hidden from the end-user, and which elements cannot be avoided. An example of this is a situation, in which an end-user encounters a new contact. The email client would not have any data to compare this new contact with, and would therefore require the user to make a decision on how to proceed. It will then be responsible of the program to provide the end user with enough information to make an informed choice on whether to trust this new contact or not.

The email client, along with the RSA implementation and an IMAP interface to GMail, will be written in Python with C optimisations. Python was chosen here as it is an easy and intuitive language to learn, and has standard libraries for handling mails. Further discussions of our choice of tools and libraries can be found in section 7.1. We hope to be able to provide another solution to the shortcomings of the email protocol.

Finally, when we have implemented our own secure email client, a minor usability test will be conducted. It will not be conducted in a usability lab, as it is not possible to reserve the lab for this purpose. The test will therefore be performed less formally, but will still be based on a realistic scenario to help produce useful data. This data will then be compared with the data from the original usability test, in order to help conclude if the goal has been reached.

CHAPTER 2

Cryptography in context

Imagine average Joe receiving an email from "Danske spil", informing him that he won a big prize. All he has to do, is to send his credit card information, and he will receive the prize. Joe suspects the email, but decides to trust it, as it looks legitimate and he makes his bets on "Danske spil" every week. However, the sender was a criminal that forged the email, making it look like something "Danske spil" would send. Since Joe chose to believe the from email address, he passes on his credit card information, and the criminal was able to withdraw all his money.

The example shows how important signing an email can be. Digitally signing emails would make situations like the one above unlikely, and it can be done very easily by using asymmetric encryption as described in section 5.2.2. Even though many do not notice it, cryptography is already deeply embedded in our daily routine. The bank uses cryptography to allow us access to our bank accounts at home, and using the Danish digital signature, taxes can be checked or corrected. So cryptography is already a part of normal peoples lives, however, some procedures such as emails is still insecure.

Emails travel across borders, due to the IP-addresses and the TPC/IP network[Gilbert, 1995]. However, when an email travels to a different country other than its origin, the authorities are allowed to read it. Thereby the content of the email is accessible to almost everybody. Some may not care that, for example, the Swedish can read their emails, however in principle it is wrong. As expressed in an article from "IT-og Telestyrlesen Ministeriet for Videnskab [2008]" the Danish Act on Processing of Personal Data prevents the Danes from reading personal information processed in Denmark, but this does not apply to emails being routed through Denmark [Justitsministeriet, 2000]. Because of this the TPC/IP network allows the Danish law to be bypassed. It is, however, possible to protect ones privacy by encrypting every email. If the emails are encrypted, the fact that other countries are allowed to read them does not matter, since they can not interpret them.

Another issue is verifying who the email sender is. As in the example many people do not know that the sender can "spoof" the sender address, making it appear to origin from a trusted source [Klensin, 2008, section 7.1]. Therefore signing emails is a way to ensure that the email conversation is actually done with the expected user. Just as encrypting emails is important for privacy, signing the emails is almost more important.

2.1 A Survey at Aalborg University

A small survey has been done amongst the first year students at Aalborg University to estimate whether email encryption is used and whether a program similar to the one we are creating[1] would be used. 77 students were aksed the following questions.

[1]See chapter 4 for more information about our secure email client.

1. Do you know what spoofing is? Spoofing: When the stated return email address is not the same as the senders email address.
2. Do you know that Google saves and analyses the information on every email sent with GMail in order to improve their advertisements?
3. Do you know whether or not the email client you use is able to encrypt and sign emails?
4. If yes: Do you use it?
5. If the new version of Thunderbird or the email client you use, or Hotmail or Gmail, were updated to use transparent encryption and signing, would you use it, based on the fact that every email can be read and/or changed, and that the sender may not be the correct?

Question 1 and 2 are designed to find out whether the students know of some of the problems discussed in this section, and to inform those who do not. Respectively 29% and 40% answered yes[2], which indicates that between 1/3 and 1/2 of the students already know about some of the issues with emails. 84% answered that they do not know or do not use any program that can encrypt emails. However, on question 5 96% said that they would use a program that automatically encrypts and decrypts emails. This clearly shows that the participants of this survey do not bother acquainting themselves with encrypting their emails, but they would use it if easily available.

2.1.1 Problems with method

The survey is based on the answers from 16 groups, in total 77 students, but with a very limited variety[3]. Because of this, this survey is not representative, however, it does show a tendency. The fact that the questions were asked in person must also be taken into account, because it can have a misleading effect.

2.2 Who benefits?

There were in general two comments to the fifth question in the survey from section 2.1. About half of the students said that based on the problematics described in questions 1 and 2, and the direct derivative of these, they would use encryption and signing to prevent anyone from messing with their emails. The other half, however, said that they did not care, and would not bother encrypting and/or signing their mails, but on the other hand would not bother turning it off either, given that it did not create any problems for the recipient. They believed that for a private person, email security would not have the same importance as it would for companies and authorities.

In America some states already have made it mandatory for companies to encrypt every outgoing email containing personal information [World, 2008]. The authorities and the companies will, without doubt, be the ones gaining the most from the adoption of secure email standards. Just as nobody would send a letter with personal information

[2]A table of the results can be found in appendix B.

[3]The different fields of the survey were ”Land surveyor”, ”Energy”, ”Technology of health”, ”Machine and production” and ”Global business engineering”

without putting it into an envelope, everybody should also encrypt their emails. The issue is very much the same in the two situations, however, many people do not see it this way.

A somewhat controversial group that would also benefit from the adoption of email encryption might be the terrorist, paedophiles, and other criminals. When every email is encrypted, the authorities will be unable to find and locate these criminals using their email conversations. Thereby this group of people is able to communicate without anyone looking over their shoulder. However, today this group probably already uses cryptography or some kind of steganography to hide their communication, if they use the internet for email communication at all.

2.3 Losers by adoption of encryption

As explained above the intelligence service will detriment if all emails are encrypted by standard. The Danish Act on Processing of Personal Data [Justitsministeriet, 2000] allows the intelligence service to read emails processed in Denmark if they suspect terrorism. But if every email is encrypted, they cannot do that.

Some companies like Google, who offers free email hosting services, would suffer from the adoption of end to end email encryption. They reserve the right to keep the email correspondence, and thereby improve their services [Google, 2008, Brugerkommunikation]. At first it seems fine that they use our data to improve our searches and the other services we use, but after greater reflection one realises the problematics. Not only do they customise their advertisements, but distribute the information to their subsidiary companies. If email encryption was adopted by the general population they would no longer be able to direct their advertisements according to the content of our emails, and Google might loose a great source of revenue.

2.4 Is email encryption good?

Whether the world would be a better place if every email was encrypted and signed, depends on the perspective. The average Joe would mainly gain from digitally signing emails, while companies and authorities would mostly gain from encryption of secret information in emails. It is our belief that, based on the survey, and the general opinions we met, the world could benefit from the encrypting and signing of emails.

It is not a question of whether the email contains anything anyone would want to read, or whether anyone is trying to cheat us, it is basically the fact that it is possible. It is possible for anyone with just a little technical insight to forge an email, and thereby trick people. It is about protecting our right to send awkward photos to a friend, without worrying if anyone sees it and uses it in an unintended manner. It is about knowing with whom one is talking, and knowing whether the person on the other end is the actual individual. Today, it is perhaps not our greatest concern, but as our civilization becomes more and more dependent on electronic communications, it is necessary to take precautions, and prepare for the problems that may arise in the near future.

CHAPTER 3

Usability test of Thunderbird with Enigmail

3.1 Purpose of the usability test

The purpose of this usability test of Thunderbird with the Enigmail addon, is to determine which usability issues may arise, when an inexperienced user tries send and recieve secure mail.

Since the test also is a completion of the lectures "Evaluation of IT-systems", the test will be performed in a usability laboratory, where the needed equipments are accessible. However, the usability lab is only available for a limited time, and therefore it is only possible to test the program with test subject. But will still the test will still give an estimative idea of the usability issues.

3.1.1 Choice of test person

Encryption should be use d by everyone, and so everyone who uses email is in the target group. For the purpose of this test, we feel the target group should be young individuals. This is because they in general have more experience with emails and computers. The youth is also often the first to use new programs, and is willing to try new ideas.

The test subject must therefore be young and be a part of the group that uses computers and emails daily. We want to see whether it is possible for the test subject to use the program without an introduction to RSA and OpenPGP or not, because it should not be necessary to understand RSA to use it.

As there are plenty of people who fulfill these demands on the University's grounds, we chose a first year student at random to perform this test. The test subject has no experience in encrypting emails but uses a computer and emails daily, both in relation to the university and home.

3.1.2 The test setup

The test is, as mentioned before, performed in the usability lab. To test the program, our test subject is given different tasks such as encrypting, decrypting, signing and verifying an email. The purpose of these tasks is to check if a user is able to use the encryption add-on. Because we only want to test how the program works we will preinstall all the programs, add-ons and email accounts needed to complete the tasks given. The email client Thunderbird is to be open and ready to use when the test subject begins the test.

Since it is not possible to determine how long the test subject will need to complete every task, because it can vary from subject to subject, we have constructed more tasks than necessary. We only have 30 minutes to complete the usability test, and seven assignments is assumed to be suitable.

3.2 The term usability

The term usability is used to denote how easy it is for a user to use a certain tool. In the context of this report, it is referred to as how easy the user finds it to interact with a given program. The primary factors concerning usability are:

- How efficient the program is to use
- How easy the program is to use
- How satisfying the program is to use

These fundamental questions are what a developer should be asking himself while he is writing an application.

The term usability itself is becoming more and more common these days, now that computer systems are quite common in the household. This market increase means more companies competing on the software market, and many companies have discovered that creating user-oriented software yield better results, compared to technology-oriented software. This can only be considered natural, when it is known, what kind of primary user, the program is meant to have. Specific task oriented software, such as development tools have their own exceptions, but software developed for everyday use, such as media players or Internet browsers have a userbase that does not necessarily have a great computer knowledge. For these users, software that is easy to use and feels intuitive, is preferred over software that takes hours to figure out how works [Sharp et al., 2007].

3.2.1 Usability tests

One method of helping software developers write user friendly software, is performing usability tests. These tests can be performed in a variety of ways, such as controlled laboratory tests or field tests. However, how one chooses to perform usability tests depends on the type of software that is being developed. If it is software that for example is used outside on the move, field tests would very likely be a better option, as the laboratory cannot simulate certain outside events that could effect how the software performs. On the other hand if the software being developed is for office use a usability test could easily be performed in a laboratory.

Usability tests also help developers correct mistakes in the software while it is still under development, as the user might use the software in a completely different way than the developer had imagined. Usability tests do, however, require a great deal of work, and a lot of time and effort goes into examining and analysing the data generated from the test. Several test users are also required - how many depends on the test type. Also it can often be advantageous to test the software on the targeted user group, but this is not always necessary [Sharp et al., 2007].

3.3 Identifying usability problems

In this section we will identify the usability problems in the usability test done on the 4^{th} of November 2008. First the problems will be identified and described in relation to the test, and afterwards the problem will be classified and analysed.

3.3.1 Checking emails before verifying - P1

In assignment 1 the test subject has received two emails, and the assignment is to verify which email is signed and which is not. Problem 1 is that the test subject does not check both emails before verifying. The test subject reads the first email, and then goes to find the key. The fact is that if the test subject knew what to look for, it would be possible to identify the correct message without fetching the key. This is of course not the assignment, hence the test subject is supposed to download the key, but looking at both emails, would have helped to find the difference.

The problem is a cosmetic problem, since it is not a problem that stops the test subject completely. However, some of the following problems is magnified by this problem.

3.3.2 Accessing the server - P2

After having read assignment 1 the test subject knows that the key must be found on the server pgp.mit.edu. The test subject tries to open the website to recover the key, even though it should be found via the program. As there was a very limited time on this test, the test leader stopped the test person from opening the browser, suspecting it would take too long.

Had the test subject been allowed to continue it would probably have taken some time before the test subject would had realised that the answer was elsewhere. This problem must then be classified as serious problem, since it would have taken several minutes and the result would be very different from the expected.

3.3.3 Searching keys - P3

When the test subject was directed back into the program after facing problem 2, there was no problems in finding the place to search for the key. Problem 3 is a problem that the test subject never realized existed. The test subject is uncertain of what to search for and chooses to search for the key owner's name. A more precise and accurate way is to search by the email. Because names can be shared by any number of people, it wields many more results, than searching for the email. Furthermore, the risk of importing the wrong key is greater when one must look for the correct email manually.

Problem 3 is only a cosmetic problem, as the test subject quickly finds the correct key, but in some cases searching by name could result in a critical problem, if the wrong key is downloaded.

3.3.4 Downloading keys - P4

After finding the correct key the test subject asks the test leader if this is enough for verifying the photo studio's signature. This question clearly indicates that the test subject does not know about private-public keys, and that the program is supposed to compare the two keys.

This problem is a critical problem, because if a user only checks for the existence of a public key, but never compares it, the user will never be able to verify the signature of an email. The problem arises because the test subject was not told about the principles

behind asymmetric keys, which answers one of our initial questions. One of our intentions with the test was finding out whether it was possible to use the program without knowing about the principles, and problem 4 indicates that it is not.

3.3.5 Updating key database - P5

After downloading the public key, the test subject minimizes the window. Problem 5 is that the program does not update after the download of a new key. Thus the openPGP-bar still says ”Unverified signature”, even though the key had just been downloaded. It proved necessary to open another email, and then return to the signed email, for the update to appear.

This is a serious problem, but because problem 5 and 6 happen at the same time, it is not possible to determine which problem is the main problem, or if the problem is a combination of the two. The way to update is not obvious. If the program still says the signature is unverified after one has downloaded the public key, it can be very confusing and frustrating. If the user then does not update the program, he or she may try to download the key again, which does not help. An other way to handle this problem is to click at the padlock, but the results of this will be handled in problem 10.

3.3.6 Finding the encryption-bar - P6

As mentioned above most of the issues in problem 5 may be caused by problem 6. But problem 6 may be caused by problem 1. In assignment 1 the test subject only checks the first email, which was accidentally the signed one. And because the test subject does not know Thunderbird beforehand, the - at this time - yellow bar in the middle of the screen does not catch the test subject’s attention. After looking around in the program for some time, the test leader chooses to help due to lack of time. By suggesting the test subject to read the unread email in the inbox, the test subject becomes aware of the yellow bar disappearing, and when going back to the signed email the message now says “UNTRUSTED Good signature”. Figure 3.1 shows the placement of the encryption-bar in Thunderbird.

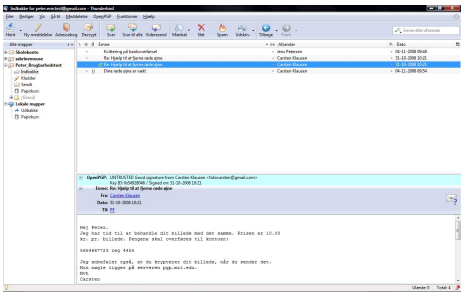


Figure 3.1: Encryption-bar

Problem 6 is a serious problem mainly because it takes 2 minutes and 46 seconds before the test subject finds the bar, and that is after some help. But it is essential to mention that is only is a problem the first time.

3.3.7 Verifying signature - P7

The test subject has found the bar, but is not sure if it is enough to verify the signature. The subject asks the test leader, who chooses to help in this question since the test subject is still working with assignment 1.

Figure 3.3: Padlock that that shows whether the mail is encrypted and/or signed.

This problem is difficult to categorize, since the test subject does not get time to solve the assignment. But we can assume that the problem would have been serious. The test subject would probably have spend some time on convincing himself, but would have decided to trust it.

3.3.8 Signing email - P8

Assignment 2 is about signing an email. The test subject quickly finds the right buttons, but is uncertain if the email is correctly signed.

Problem 8 is a cosmetic problem, as the test subject somewhat quickly decides that it was done correctly.

3.3.9 Search in local directories - P9

In assignment 3 the test subject has to fetch a public key on the server. The test subject has done this before, but this time the test subject searches in the wrong place. Instead of searching on the server the test subject searches in the already downloaded keys. This of course gives no result. Figure 3.2 shows the correct place.

As the test subject quickly finds out, that the search was done in the wrong place, and finds the right one, it must be viewed as a cosmetic problem.

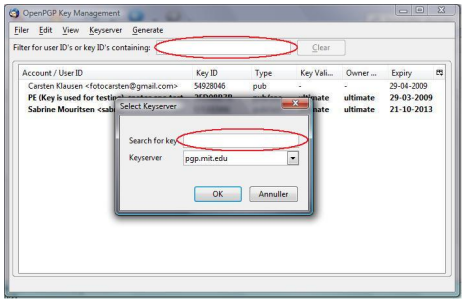


Figure 3.2: The correct place to search for keys, not the textentry behind.

3.3.10 Updating via padlock - P10

As mentioned in problem 5, there is another way to update the key database. If pressing the padlock viewed in figure 3.3 the key database will be updated. But this results in an error message. The dialog box asks if the test subject wants to import the public key. But the test subject has already done this, although not updating. When clicking OK, a dialog box appears with an opportunity to import the key again.

This error message is a serious problem, since it can be very confusing to a user, if the program wants to download the public key twice.

3.3.11 Decrypting attached pictures - P11

Assignment 5 is about decrypting an attached picture. The test subject tries to press several buttons, but never finds the correct one. The button ”Decrypt” (see figure 3.4) is pressed twice and summons an error message (which we cannot recreate). The test subject also tries to download the public key, even though you only need your own private key. Problem 11 is in general about the problems one can experience when trying to decrypt an attachment. The most obvious buttons do not work, and when saving the attachment on the desktop it does not suggest to decrypt. One has to right-click the attachment and press ”Decrypt and save as”.

Problem 11 is a critical problem, since the test subject never finds the correct approach. We ran out of time, at this point, thus we cannot know, how long it would have taken the test subject to figure it out, but the test subject spends about six minutes on the assignment, without result. This is quite some time, considering the expected simplicity of the assignment.

3.3.12 Distribution of problem types

In all we found three cosmetic problems, five serious problems and one or two critical problems.

Overall the program has a rather steep learning curve. A lot of things do not seem logical at first sight, though they might do, once one knows the program. Once the user has learned how to use the program, it is much easier to use, as the procedure to each feature is not complicated as such.

After finishing the test, the user says that it is obviously a good program.

Figure 3.4: Decrypt button

3.4 Dealing with usability problems

After identifying the usability problems, this section will try to describe how to avoid some of them.

The problem of what to search for on the key server is the first one the test subject experiences. It was without major problems for the test subject to find the correct place, but it was unclear what to search for. The test subject tried to search by name, which is not strictly wrong, instead of via email. The result was between 10 and 30 different names, but the test subject then identified the correct key by matching with the email. Had there been a label suggesting that it was possible to search by email, this would have shortened the list to only the key(s) associated with the relevant email.

But one of the most significant problems is that the program does not update itself after downloading a public key. Therefore it still says ”unverified signature” even though the matching public key has been downloaded. The problem could easily be solved by the user by pressing another email, and then pressing back to the correct email. But

this is tiresome for the user, and not expected. It could therefore be a solution to make the program update its key database frequently, or at lest after a new key has been downloaded.

The program's way of showing if an email is encrypted or signed is a small bar in the middle of the screen. This is fine, if the user knows it is there, but otherwise it is somewhat difficult to find and understand. The syntax used could also seem odd to the uninitiated. If a user has downloaded a newly created key the message in the bar is "UNTRUSTED good signature". This means that the public key *does* match the encrypted message, but the word "untrusted" can be confusing. In fact it means that the user has not told the program they trusted the key. This means that the signature of the email is correct, but no one has verified that the key holder is legitimate. The fact that "untrusted" stand out more than "good" could make some users uncertain of whether the mail is signed properly. It would therefore be an idea to make the two words seem equally important, and describe the meaning of them.

The test subject experiences some misleading error messages in the last assignments of the test. Especially when trying to decrypt the attachments the test subject does not know what to do. The test subject tries several buttons, none of which bring about the expected result. A dialog box could suggest a way to solve the problem. The test subject presses the button "Decrypt" many times, but never receives a message with ideas of what else to press. The "Decrypt" button has a tooltip explaning what it does: "Decrypt or verify the message with OpenPGP(Enigmail)". It says nowhere that it cannot decrypt attachments. An idea could be if the program checked if the mail has any attachments, and if so, suggesting that the user right-click the attachment and decrypt, if that is the intention.

3.5 Evaluation of method

This section will evaluate our method of performing the usability test.

First and foremost we did not check if the test subject had previous experience with the program Thunderbird. This is actually an essential aspect, since we can not fully conclude whether the experienced problems occurred because of Thunderbird or Enigmail. We chose Thunderbird, because it is a very popular client, and many use Outlook, which has a similiar structure. We therefore made the wrong assumption that because many people use Thunderbird, our test subject would too. Our test subject does not use either Thunderbird or Outlook, but Hotmail. Hotmail has a completely different structure, and many of the things that would seem obvious to a Thunderbird/Outlook user is not that obvious to a user of Hotmail. For example problem 6 could be caused by this.

Another aspect that did not work well was receiving emails. One email was received 5 minutes and 36 seconds later than intended, which forced us to skip the assignment, and return later when the email was received [1].
We considered having all the correspondances present in the inbox from the beginning, but decided against it for a sense of realism.
The low speed was likely due to using Gmail-accounts. This was done so that we could

[1] See appendix for log file

easily create temporary email accounts that would seem realistic to the test subject - for example the photo studio's email-address was "fotocarsten@gmail.com".

Another problem is that we forgot to debrief our test subject after the actual test. An interview could have given more details about the test subject's experience and feelings when working with the program. Luckily some of the time spent waiting for delayed emails, as mentioned above, was used discussing the test subject's opinion so far. The test subject felt that it was pretty clear, if one just opens their eyes. However considering our analysis this statement can be considered inadequate.

Another issue is the allotted time and number of tests with different subjects. Testing usability issues with a single test subject is not enough to properly verify usability issues. Furthermore our time constraints meant not fully exploring the interesting aspects of the program. This however was something that could not have been changed, as this was decided by our lecturer of Evaluation of IT-systems.

A positive discovery from the usability test was our ability to fulfill our roles. We had one individual assigned to receive and send email from and to the test subject. This was peeformed adequately. Also a test leader was present with the test subject. This individual's job was to support the test subject, but not to help solving the assignment. This role is very important, since the test leader has to make decisions during the test, but also ensure that the test subject proceeds in a way that can be used in the following analysis. Taking into account that it was the first time for the test leader, there were no issues. In the technical room a cameraman worked the cameras, so that optimal usage of screenspace was used. Also two loggers logged all important events, so that we later had both video and notes, to create a log file from. This was also satisfactory.

As the secondary objective was to learn to use the usability laboratory, we must evaluate this. We were given an introduction to the laboratory and learned how to use the camera, video recorder and microphones. In general we gained experience on what such a laboratory can look like, and how it is used, so that we can use it again in an other project.

3.6 Conclusion

In general this usability test answered our initial questions. Our purpose was to find out whether a user, with no understanding of RSA and asymmetric encryption, could solve some real life situations regarding encryption of emails. We found that some of the assignments were easy to solve, but some were not. Looking at problem P4 and P11 we can conclude that the test subject did not know what asymmetric encryption is, and therefore had serious problems with solving some of the assignments. Problem 4 is where the test subject thinks that that the presence of the public key is enough, and in assignment 5 (problem 11) the user downloads a public key to decrypt a message. Because of this the test subject cannot solve the assignment without help from the test leader. In addition the assignments were phrased to slightly help the test subject in solving the assignments. This was done to increase the chances of the test person completing the assignments, and thereby making sure that we had something to analyse. Had there been more time, this would not have been done, but it was not recommended in this usability test.

However we found that the program is easier to use, if it is not the first time that the user uses the program. The first time the test subject uses a feature it is difficult, but the second time it is easier. It is possible to access the manual by pressing "About OpenPGP" in "OpenPGP", but since the test subject never attempts to use it, we assume that is not obvious.

In conclusion we found that it was difficult to encrypt and decrypt emails with Enigmail for Thunderbird, without knowing about the principles of RSA and asymmetric encryption.

CHAPTER 4

Graphical user interface

One of the purposes with this project, is to determine the usability problems with an existing program, using asymmetric encryption of emails. Based on the results a simple and more logical program will be made. This chapter will describe how usability problems found in Enigmail, will be dealt with in our program RaptorMail.

The first section will describe those features implemented on the basis of the usability problems. The second section will describe additional features, made to improve the usability. At last a section will describe a small usability test done on RaptorMail.

4.1 Addressing usability issues in Thunderbird

The major problem experienced in the usability test of Thunderbird with Enigmail was the key management. The test subject had problems with fetching the public keys, and understanding the messages from Enigmail.

Enigmail's public keys lie on a public server, and when somebody wants to encrypt or verify the integrity of a message, they must download the pubic key from here. However, Enigmail have four different key servers installed, and it is up to the user to figure out which one to use. In general the public key server is a bit confusing to the test subject and it is time consuming.

With RaptorMail we have attempted to make a more logical way of distributing the public key. Basically Alice sends her public key to Bob along with a signed email. When writing an email the window from figure 4.1 is shown. Below "Subject" it is possible to select whether the email is to be signed or encrypted. Because of the limitations with asymmetric encryption, an email cannot be encrypted without the receiver's public key, and this checkbox is therefore disabled. When the public key have been received, new emails can be encrypted, and the program automatically enables the encryption checkbox.

However, this way of distribution only works if all who have a public/private key sign their emails. It also contains the problem of how to be certain of the identity, of the person, with whom one is corresponding. Alice can receive Trudy's public key, in the belief, it is Bob's. Alice and Trudy then have a long conversation over email. When Trudy uses her key, Alice can only be certain, the one who sent this mail, is the same as the one who sent the previous mail. The same problem is present in Enigmail, and can best be solved by Alice and Bob exchanging public keys in person.

The test subject also experienced problems with the verification of emails. Enigmail uses a bar in the middle of the screen to show whether the email was encrypted and/or signed. In addition Enigmail enables one to trust the public key, and thereby verifying that the public key actually belongs to the person. Because of this a somewhat confusing message is displayed, if the key is not trusted (Se figure 4.2).

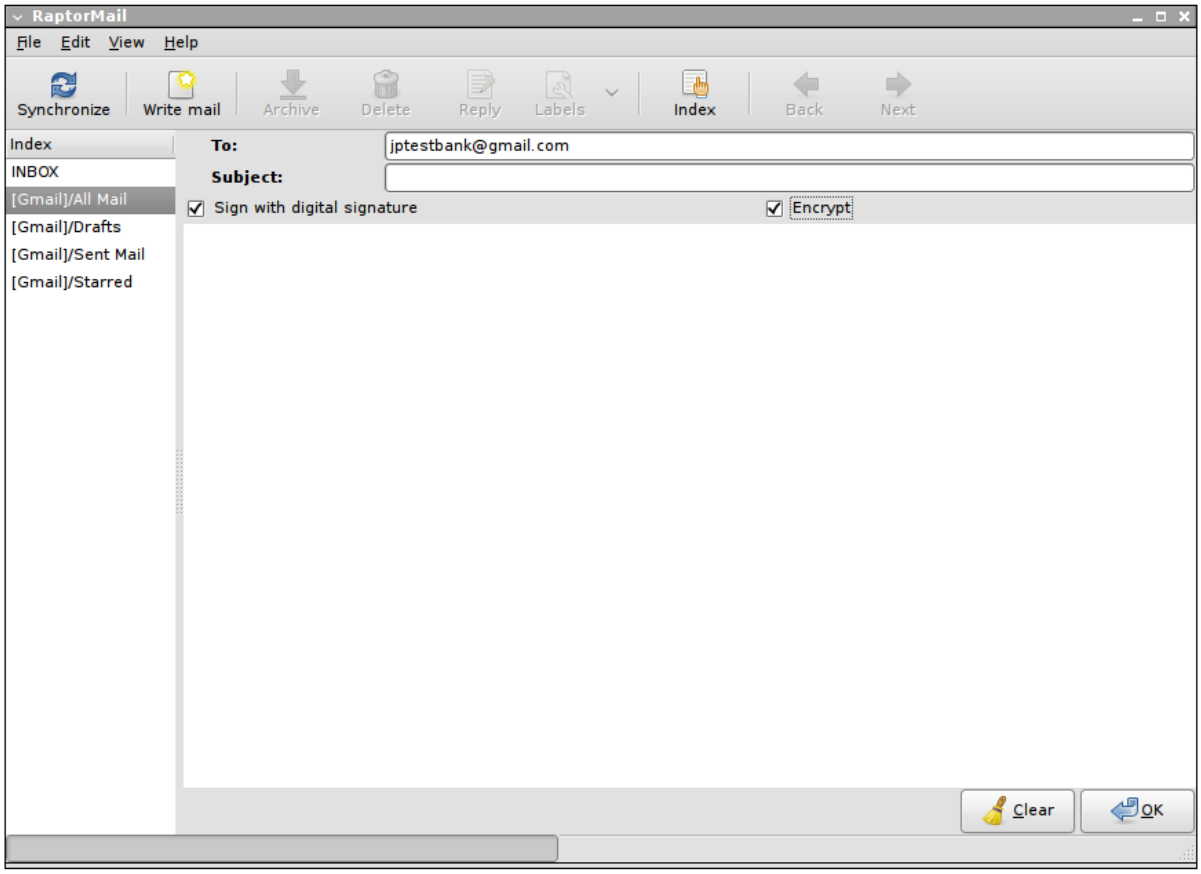


Figure 4.1: Writing an email in RaptorMail

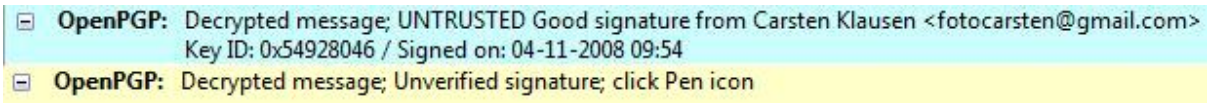


Figure 4.2: Encryption bar in Enigmail

In RaptorMail we try to avoid this text by using icons. When Alice receives a signed email in RaptorMail, a green check mark is shown in the upper right corner inside the mail. Once she exits the new mail, the check mark will also appear in the inbox, along with the other pieces of information about the email (email, subject, date), as seen on figure 4.3.

One of the other problems that the test person experiences is the decryption of attachments. This feature is however not implemented in RaptorMail, due to lack of time. The encryption process is much the same, however implementing it in the email client is rather complex. We have also chosen not to implement the ability to trust a public key, for to the same reasons.

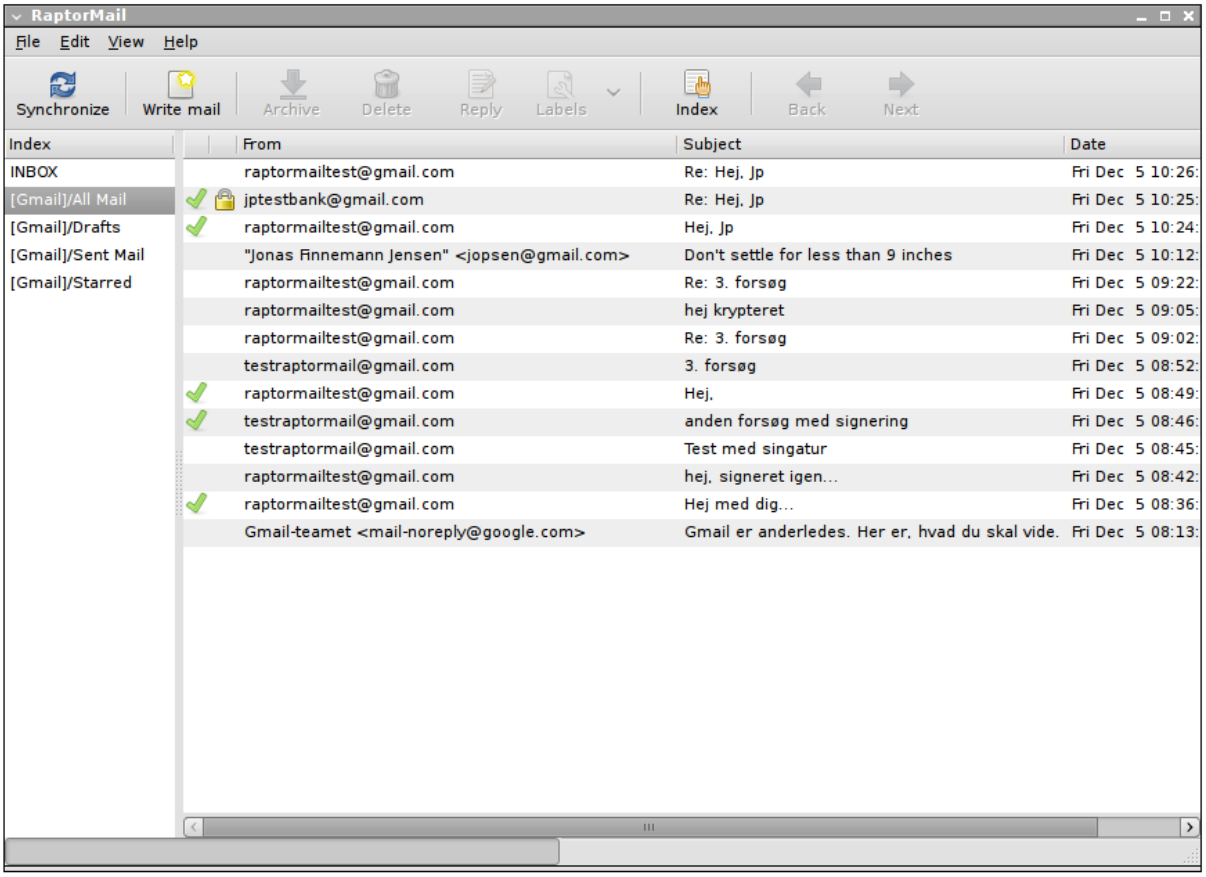


Figure 4.3: All mail in RaptorMail

4.2 Additional usability features

From the experience made in the lectures VIT[1] some additional features have been implemented to improve the usability.

Thunderbird shows the currently selected email in a window in the bottom of the screen, as seen in figure 4.4. Thereby the inbox is stationary and visible at all times.

Instead of this RaptorMail only have the menu, the toolbar and the index stationary. The main window changes view according to the selected widget. This way almost the entire window is available to read and write emails and viewing the index.

Another major feature is the toolbar buttons in the toolbar menu. Figure 4.3 shows the toolbar from the view of the inbox. If a toolbar button is not usable in the current window it is disabled and grayed out. In addition the functions available from the toolbar are also available in the menu. This is usable, as some prefer the menu, and some prefer the toolbar. This way both needs are fulfilled. And as people are different it is different how the names of functions is interpreted. Therefor every function in the program has a tooltip, which describes the button's functionality.

[1] Vurdering af IT-systemer (Evaluation of IT systems)

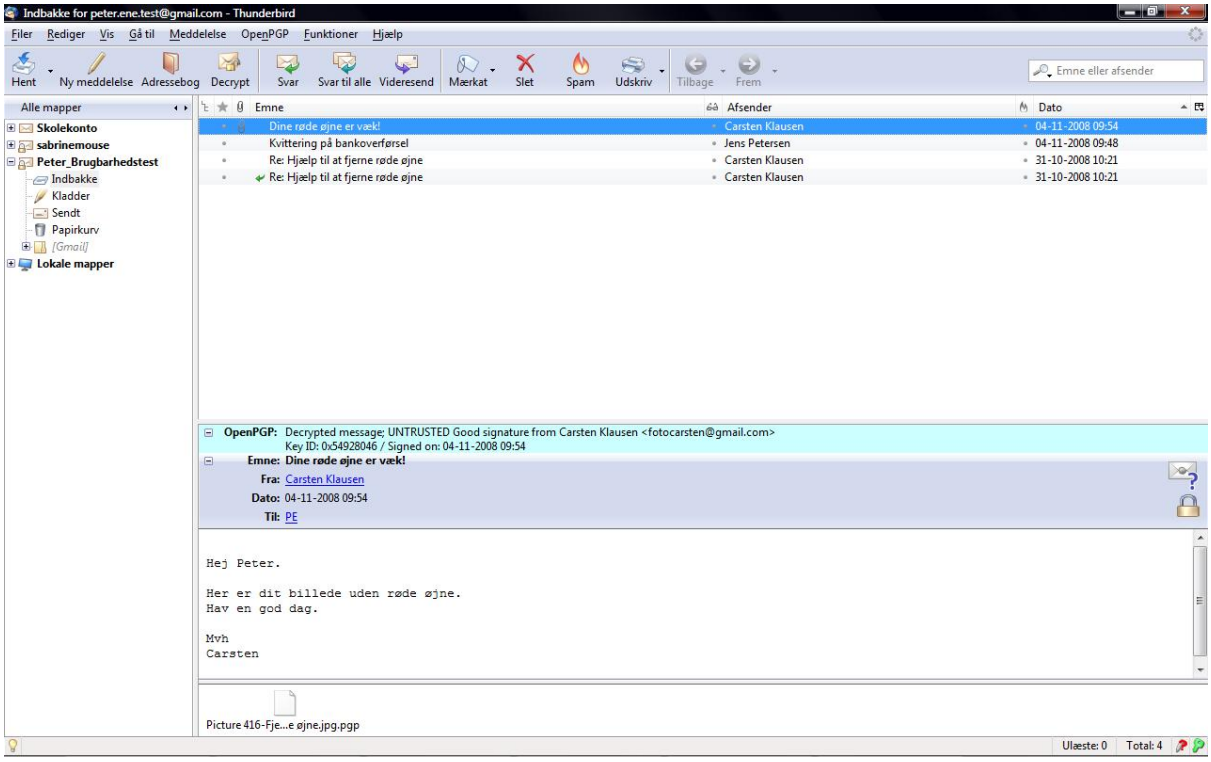


Figure 4.4: Stucture of Thunderbird

The program has nine toolbar functions, where the three of them are ”Write mail”, ”Delete” and ”Reply”. These functions are pretty much self-explanatory, however, the additional functions may need a tooltip. The first button is labeled ”Synchronize” and synchronizes the program with GMail. This means that the user is able to read and edit labels offline. The next time the user is online, ”Synchronize” is pressed, and the changes is updated to GMail. The next function is ”Archive”. This function simply removes the selected email from the inbox, and if addressed a label, it is accessible from ”All mail” in the index, or the respective label. Labels are addressed by selecting an email, and either chosing an existing label, or creating a new one. ”Index” is a very important function that moves the user back to the current index. If the user enters an email from a label, and presses ”Index” the main window will change to an overview of all emails with that label. In addition pressing ”Index” in an email entered from inbox, will view the inbox. The last functions are the ”Back” and ”Next”. Because RaptorMail is based on GMail the emails are sorted by conversations. By pressing ”Back” or ”Next” the user can go back and forth in the conversation.

The purpose of these functions is to make it simple and manageable for the user. With few functions, it is easier to use the program, and to remember the buttons’ functions.

As in Enigmail, RaptorMail runs a startup guide the first time the program is launched. This guide helps the user generate the public and private keys, and synchronize with GMail the first time. Meanwhile information about asymmetric encryption and when and where to update to Gmail next time is displayed. This informs the user of the basic principles with the cryptosystem, so that the functions in the program makes sense. When the guide is finished it is possible to read a manual from the menu “Help”.

4.3 Small usability test on RaptorMail

Because the intention with RaptorMail is to try and make a more simple program that uses asymmetric encryption with emails, it necessary to evaluate the outcome. Therefore a small usability test was conducted with one test person. This time the usability lab was not available, hence we did not have the same equipment. The computer screen was recorded while a logger noted important things. As the information with this test is limited in comparison the the usability test done on Thunderbird with Enigmail, this report will not go into the details of the test. Instead the experienced problems will be discussed.

The program was launched, and the test person then had to follow the introduction guide. This went without any problems, however, the test person asked whether emails are encrypted automatically. This indicates that the test person does not know the deeper aspects of emails, which makes the test person suitable.

As it takes a random time to generate the key set, the progress bar does not go from left to right, but moves back and forth until the keys are generated. As there is no way of knowing how long time it takes to generate the key, it is not possible to make a progress bar that shows how much time is left. It was however confusing for the test person, who began doubting whether the program had frozen, as he could see no progress. (Se figure 4.5). Because of this the text message ”This may take a while” was added in the dialog.

The first assignment was to send a signed email. This raised no problems. However, the test person subject suggested, the button for sending emails should be labeled as ”Send” instead of ”O.k.”.

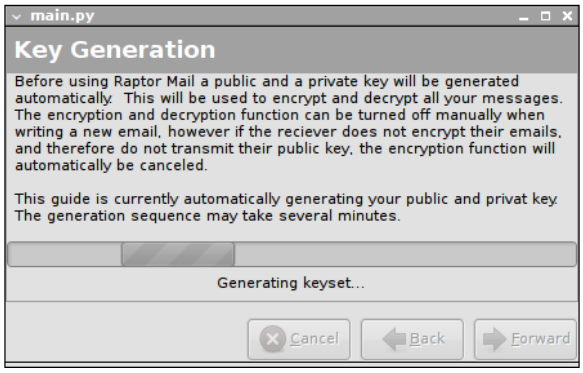


Figure 4.5: Progress bar when generating key set

The second assignment was to receive an email. The purpose of this assignment was to determine whether the function ”Synchronize” is understandable. The test person immediately presses ”Synchronize”, and a dialog box appears. When the test person received the email, a dialog box appeared. It said, there was already a public key for that particular email, that did not match the public key in the email, and asking if the test subject trusts the new key, which would overwrite the old one. This setup was made, so we could document how the test person would react to such a message. The dialog can be seen at figure 4.6. The test person decides to trust it, even though he makes it clear that he is in great doubt. Enigmail's public keys are uploaded to a server and can then be revoked. That is, if the owner of the private key forgets the password, or has no use for that particular key set any more, they can revoke the public key from the server. The key still appears on the server, but is clearly marked ”Revoked”. With RaptorMail, one would simply have to create a new key if the old one is lost. Others using RaptorMail then receive emails signed with the new key. This results in the above mentioned dialog box where the recipient has

no immediate way of finding out, whether this is an imposter, or just an issue of a lost password.

In ”view mail” the test person does not immediately discover the icons symbolizing that the email has been encrypted and signed. The test subject's comment was to make a status bar telling it in words, however as concluded from the test of Thundebird with Enigmail, that does not necessarily work either.

The next assignment was to reply with a signed and encrypted email. The test person does what was intended, but is surprised that the previous mail is not included in the response. But as the buttons ”Back” and ”Next” shows these parts of the email, including them would only repeat every email.

The fourth assignment is to apply a label to the received email. The test person has no trouble, and comments: ”Smart. That wasn't so difficult.”

Figure 4.6: Dialog box for a new key for an existing mail

The fifth and last assignment was to remove a spam mail from the inbox. This assignment was to investigate which procedure the test person would choose. He chose to delete it via the toolbar. A dialogue-box appeared, asking him if he was sure, he wanted to delete the email, and, at the same time, told him he needed to synchronize for it to take effect. He synchronized, and the email disappears.

At last the test person was asked to try some of the other functions, and tell if anything surprises him. The buttons to go forward and and backwards in the conversations, did not give the result, the test subject expected, based on their position. However when seeing the tooltip, the function was clear. The ”Index” button confused the test subject shortly, as nothing happened, when he pressed it in his Inbox, but he quickly figure out its function.

The test subject comments the program as being very simple. He also enjoys the fact that the ”Archive” function from GMail, also works in RaptorMail, but would like the tool tip to be changed (it said ”Archive the selected email. This removes the mail from the Inbox.”).

4.4 Conclusion on Graphical user interface

The conclusion from this chapter must be that RaptorMail is not a perfect program, which is logical and 100 % usable. The program does however solve some of the problems experienced in the usability test done on Thunderbird with Enigmail.

One of the major issues that RaptorMail had to solve was the distribution of public keys. We believe that RaptorMail's version is more logical than Enigmails, since one does not have to upload and download keys. One only need to send a signed message. However it has not been possible to find an better way of verifyng that the public key belongs to the expected person.

Regarding choice of test subject, he belonged in the same group as the test subject used in the test of ThunderBird. He was in our scope of age, 15-35, and also attends Aalborg University. This test subject was already a user of GMail, which might help with the understanding of the archive and label functions, making it easier for him to understand the program.

RaptorMail is a simple email client that can encrypt, decrypt, sign, verify and distribute keys, using GMail. Due to the limitations on this project it has not been possible to make RaptorMail able to run on Windows, but it does run without problems on Linux.

CHAPTER 5

The mathematical fundation of RSA

5.1 Encryption types

When encrypting emails there are two types of cryptography: the symmetric algorithms and the asymmetric algorithms. This section briefly explains the difference.

5.1.1 Symmetrical encryption

Symmetric encryption can be used if both sender and receiver have access to a shared secret key denoted e. Consider Alice as the sender of a message m, and Bob as the intended receiver of the message. Then, if Alice only has access to an insecure transmission channel such as a postcard or email, Alice may wish to encrypt the message m before transmitting it to Bob. Alice may use a symmetric encryption algorithm $f_{encrypt}(e, m)$ to compute the ciphertext $c = f_{encrypt}(e, m)$. The ciphertext should appear to be random and should not be decipherable without knowledge of the shared secret key. Alice could then transmit the ciphertext over an insecure channel to Bob, and given that Bob knows about the algorithm and the shared secret he may also decrypt the message using $f_{decrypt}(e, c)$ which is the inverse function of $f_{encrypt}(e, m)$.

5.1.2 Asymmetric encryption

Asymmetric encryption can be compared to a snap lock. Alice sends Bob a letter and uses Bob's snap lock on the envelope. Now only Bob can unlock the message as only Bob has the key to the snap lock. The benefit of this system is that Bob never has to give his key to Alice. Alice just needs one of Bob's snap locks to send him a letter. The keys are called private and public keys where the public key is the snap lock and the private key fits the snap lock. Now Bob can send his public key to anyone without fear because the private key cannot easily be recovered from the public key. If Alice wants to send Bob an encrypted message she will have to retrieve Bob's public key e_{Bob}. Because Bob's public key is public, Alice can easily acquire a copy of it. Alice then encrypts the message m using the public key e_{Bob} in the function, $f_{encrypt}(e_{Bob}, m)$ and sends the ciphertext to Bob. Bob now receives the encrypted message. To decrypt it he uses his private key d_{Bob} to decrypt the cipher message c with the function $f_{decrypt}(d_{Bob}, c)$.

5.2 Introduction to RSA

In 1977 Ron Rivest, Adi Shamir and Leonard Adleman, of MIT, proposed the asymmetric encryption system RSA [Landrock and Nissen, 1997]. RSA is an asymmetric crypto

system and is quite different from the usual symmetric crypto systems. RSA uses a key set instead of just a single key.

The key set consist of two keys, a public key and a private key. The keys match so that messages encrypted with the public key can only be decrypted with its unique matching private key and vice versa. However unlike the symmetrical encryption key, the RSA public key can be given away to anyone. This is because the public key can only be used for two things: Encrypting messages that can only be decrypted with the private key and decrypting messages that have been encrypted with the private key.

5.2.1 An encryption example

In reality it is not physically possible to add a snap lock to an email. This is where the RSA comes to play.

Key generation

Before Alice and Bob can encrypt their emails they need a key set. Bob begins generating a key set, by selecting two secret prime numbers. This is usually done by generating two random prime numbers. He chooses 17 and 23 and denotes them p and q respectively. He then calculates the product of p and q to be 391 and denotes this number $n = p \cdot q = 391$. Bob must then calculate $\varphi(n) = (q-1) \cdot (p-1) = 352$. Now Bob selects another prime number less than $\varphi(n) - 1$, and he chooses 83. This number together with n is Bob's public key (e, n). In reality e is not required to be prime, only coprime to $\varphi(n)$, however it can be easier to generate a third prime number. Bob's private key will be n and the integer d such that $e \cdot d \bmod \varphi(n) = 1$, this is also known as the modular multiplicative inverse of e under modulo $\varphi(n)$. This way Bob calculates his private key d to be 123, and he has now completed the key generation.

The security in generating the keys lies in the fact that no one knows what prime numbers Bob uses. There is a small chance that two people select the same primes p, q and e and will therefore generate the same keys. If that happens these people will be able to read and or sign each other emails. Because of the large prime numbers used, when generating the keys, it is extremely unlikely that this happens but there is still the risk of programming errors. This happened in 2006 when a Debian developer by accident made a change to the random number generator. As a result it was only possible to generate a very limited number of keys, and an attacker could easily try them all to crack the encryption [Weimer, 2008].

RSA email

Now Alice wants to send Bob a message, however because of the technical limits of RSA the message can only be an integer and it has to be less than the value of n in Bob's public key.

However, Alice could translate her message into a sequence of integers and encrypt them one by one. Or she could use RSA to encrypt a key to a symmetric encryption algorithm, and then use the symmetric algorithm to encrypt the message. But today

Alice just wants to send Bob the number 3. She knows that to encrypt a message she has to raise her message $m = 3$ to the power of e_{Bob} under modulo n_{Bob}.

$$RSA \quad Encryption \quad : \quad c = m^e \bmod n \tag{5.1}$$
$$Alice \quad Encrypt \quad : \quad 3^{83} \bmod 391 = 384 \tag{5.2}$$

Now only an entity in possession of the private key d such that $d \cdot e \bmod \varphi(n) = 1$ can recover the message m from the ciphertext 384, and Alice can send Bob the ciphertext over an insecure channel. Once Bob receives the message he has to decrypt the message before he can read it. Bob raises the ciphertext 384 to the power of d_{Bob} under modulo n_{Bob}. This is basically the same as raising the message m to the power of $e \cdot d = \varphi(n) \cdot k + 1$, and thus by Euler's theorem the result is 3, which is what Bob gets when he decrypts the message.

$$RSA \quad Decryption \quad : \quad c = m^d \bmod n \tag{5.3}$$
$$Bob \quad Decrypt \quad : \quad 384^{123} \bmod 391 = 3 \tag{5.4}$$

5.2.2 Signature and verification

Another way of using asymmetrical cryptography is by signing messages. This can be used to confirm the integrity of a message. First the sender calculates a hash value of the message. The hash value is like a fingerprint, it is unique to the message and it is theoretically impossible to convert to the original message, as a hash function is not a bijective function. This means that a hash value of a message can be shorter than the message. There are also situations where the hash algorithm can create collisions, where two message have the same hash value, however, these collisions cannot be predicted.

To sign the message the sender now uses his private key to encrypt the hash value to create a signature which is attached to the message. Now anyone with the matching public key can use it to decrypt the signature and by comparing it to the actual hash value of the message, it is possible to conclude that only the person with the private key could have generated the signature. Otherwise if the signature comparison does not match either the identity of the sender is wrong or the message has been altered.

5.2.3 A digital signature example

Alice wants to send Bob the integer 24, and Bob needs to verify that Alice did in fact send him that integer, and that it was not altered during transmission through the insecure channel. First Alice needs a private key - she creates one in the same way as Bob did - but she uses different prime numbers as these are random, and as a result she gets different key set.

$$n \quad public = 49 \tag{5.5}$$
$$e \quad public = 23 \tag{5.6}$$
$$d \quad private = 11 \tag{5.7}$$

Before transmitting Alice needs to create a signature of the message. To do this she uses a cryptographic hash algorithm to generate a unique hash value of the message. To keep it simple Alice uses the function 24 mod 9 = 6 to get the hash value of the message. This is not a secure cryptographic hash algorithm, but it will do for this example. Next Alice uses her private key to encrypt the hash value 6 to create the signature.

$$RSA \quad Signature \quad : \qquad Signature = 6^{11} \bmod 49 \tag{5.8}$$

The result is the integer 27, now Alice transmits the message to Bob along with the signature. When Bob receives the message 24 and the signature 27, he would like to verify that it was Alice who sent the message. He does this by decrypting the signature with Alice's public key.

$$RSA \quad vertify \quad : \qquad Signature = 27^{23} \bmod 49 \tag{5.9}$$

The result is 6, and now Bob can compare the signature value with the actual hash value of the message. He just needs to use the same hash function as Alice.

$$Signature \quad = \quad 24 \bmod 9 = 6 \tag{5.10}$$

The signatures match, and Bob concludes that Alice was in fact the sender of the message, given that he has retrieved her public key through a trusted entity and that Alice is the only one in possession of her private key.

5.3 Euler's theorem

Euler's theorem is the foundation of the RSA cryptosystem as it proves why messages can be encrypted and decrypted using public and private keys respectively. Strictly it does not say anything about the difficulty of decrypting a message without the private key, however, this aspect will be briefly covered in section 6.8. Before we can state Euler's theorem, we need two definitions about integers regarding their divisors.

Definition 1 $a, b \in \mathbb{Z}$ are said to be *coprime* or *relatively prime* if their greatest common divisor is 1 [Lauritzen, 2003, A. Menezes and Vanstone, 1996]. That means, there exists no $k \in \mathbb{Z}$ integer such that $a \cdot k = b$ or $a = b \cdot k$, e.g. they have no common divisors apart from 1, thus the greatest common divisor is 1.

Definition 2 Euler's totient function $\varphi(n)$ denotes the number of integers less than n, greater than or equal to 1 and coprime to n [A. Menezes and Vanstone, 1996]. That is $a \in \mathbb{Z}$, $1 \leq a < n$ and $a \cdot k \neq n$ for $k \in \mathbb{Z}$.

By definition 2 above $\varphi(9) = 6$, as the integers $1, 2, 4, 5, 7$ and 8 do not divide 9, thus $\varphi(9) = 6$ since the number of integers coprime to 9 is 6.

Notice also that $\varphi(7) = 6$, since 7 is a prime and no integer divides a prime. Thus we may also write $\varphi(p) = p - 1$ for all $2 > p \in \mathbb{P}$ where $\mathbb{P}$ denotes the set of all primes.

If n and m are coprime $\varphi(n \cdot m)$ can be calculated as $\varphi(n \cdot m) = \varphi(n) \cdot \varphi(m)$. In proposition 3 it is proved that if and only if a and n are coprime and a and m are coprime then a and $n \cdot m$ are coprime. This along with the proof that mapping from the set of integers coprime to $n \cdot m$ is one-to-one to the set of all possible pairs of integers coprime to n and m respectively, proves that $\varphi(n \cdot m) = \varphi(n) \cdot \varphi(m)$. For a complete formal proof see Lauritzen [2003] or Clark [1985].

Proposition 1 Euler's theorem states that $a^{\varphi(n)} \bmod n = 1$ for all $a \in \mathbb{Z}$ where a and $n \in \mathbb{N}$ are coprime [Lauritzen, 2003]. Usually expressed using congruence relation as shown below, where "≡" denotes a congruence relation. See appendix D for an explanation of modulo and congruence relations.

$$a^{\varphi(n)} \equiv 1 \pmod{n} \tag{5.11}$$

The proposition above will be proved later in this chapter, but for now we will just look at how Euler's theorem is used in asymmetric cryptography.

Consider a raised to the power of $\varphi(n) + 1$ instead of $\varphi(n)$, then it would be the same as multiplying a by 1, as shown in equation 5.12. Actually we could raise a to the power of 1 plus any constant $k \in \mathbb{N}$ times $\varphi(n)$ as this can be written as a times 1 k times, which gives 1 (5.13).

$$\begin{aligned}
a^{\varphi(n)+1} &\equiv a \cdot a^{\varphi(n)} \equiv a \cdot 1 \equiv a \pmod{n} && (5.12)\\
&\equiv a^{\varphi(n)\cdot k+1} \equiv a \cdot a^{\varphi(n)} \cdot a^{\varphi(n)} \cdots = a \cdot 1 \cdot 1 \cdots \pmod{n}\\
&\quad k \in \mathbb{N} && (5.13)\\
e \cdot d &\equiv \varphi(n) + 1 \equiv \varphi(n) \cdot k + 1 \pmod{n} \quad k \in \mathbb{N} && (5.14)\\
a^{e^d} &= a^{e \cdot d} = a^{\varphi(n)\cdot k+1} && (5.15)
\end{aligned}$$

Now imagine that we are able to express $\varphi(n) \cdot k + 1$ as a factor of two integers e and d, as illustrated in equation 5.14, then if a is first raised to the power of e, nobody except those who knows d and n would be able to determine a. That is of course given that $\varphi(n)$ is difficult or practically impossible to compute from n, as will be discussed later in this report. In equation 5.14 d is the modular multiplicative inverse of e under modulo $\varphi(n)$, that is $e \cdot d \equiv 1 \pmod{n}$. In the next section techniques needed for computing modular multiplicative inverses will be explained, and later Euler's theorem will also be proved.

5.4 Greatest common divisor

The greatest common divisor is the greatest number that divides a and b. By now it is useful to help establish the public key e as it is required to verify that e and $\varphi(n)$ are coprime. This section deals with everything required to find the greatest common divisor of two numbers.

5.4.1 Division theory

Every rational number $a \in \mathbb{Z}$ can be written as

$$a = q \cdot d + r \tag{5.16}$$

where $q, d, r \in \mathbb{Z}$ and $0 \leq r < q$ [Lauritzen, 2003]. In the above representation of a, d is a divisor of a and r the reminder of a divided by d. The reminder r may be found by calculating a modulo d, denoted $r = a \bmod d$. If $r = 0$ then d divides a or d is a divisor of a, denoted $d \mid a$.

5.4.2 Common divisors

The set of all divisors of n are denoted as:

$$div(n) = \{d \in \mathbb{N}; d \mid n\} \tag{5.17}$$

Example 1 Below is three examples of $div(n)$:
$div(32) = \{1, 2, 4, 8, 16\}$
$div(11) = \{1\}$
$div(32 \bmod 11) = div(10) = \{1, 2, 5\}$

Proposition 2 The set of common divisors of a and b are equal to the set of common divisors of $a \mod b$ and b, if $a > b$.

$$div(a) \cap div(b) \quad = \quad div(a \bmod b) \cap div(b) \tag{5.18}$$

Proof Any number d divides a and b if and only if it divides $a \bmod b$ and b, given that $a > b$. Since $d \mid a$ and $d \mid b$, the following must be true:

$$\begin{aligned} a &= q_1 \cdot d && (5.19)\\ b &= q_2 \cdot d && (5.20)\\ a - b \cdot y &= q_1 \cdot d - q_2 \cdot d \cdot y = (q_1 - q_2 \cdot y) \cdot d, y \in \mathbb{Z} && (5.21) \end{aligned}$$

As can be seen above, d divides $a - b \cdot y$ since $d \mid ((q_1 - q_2 \cdot y) \cdot d)$ because $d \mid d \cdot x$ for every $x \in \mathbb{Z}$. Given 5.16 it is known that $a \bmod b$ may be written as $a - b \cdot y$, thus the proof is complete.[Landrock and Nissen, 1997] □

5.4.3 Euclid's algorithm

The common divisors of a and b are the set $div(a) \cap div(b) = div(d)$ where d is the greatest common divisor of both a and b [Lauritzen, 2003].

Euclid's algorithm, or the Euclidean algorithm as it is also called, can be used to find the greatest common divisor of $a, b \in \mathbb{Z}$, denoted $gcd(a, b)$.

If $a, b \in \mathbb{Z}$, then

$$\begin{aligned} gcd(a, 0) &= |a| && (5.22)\\ gcd(a, b) &= gcd(a \bmod b, b), \quad a > b && (5.23) \end{aligned}$$

It follows from (5.17) that (5.22), since $div(0) = \mathbb{N}$ because every integer $\in \mathbb{N}$ times 0 is 0. This is only true for $div(0)$. If $div(0)$ contains all natural numbers, then the greatest common divisor of a and 0 must be a, hence $gcd(a, 0) = a$ if $a \in \mathbb{N}$ and $gcd(a, 0) = |a|$ if $a \in \mathbb{Z}$.

From (5.18) it can be concluded that (5.23) must be true. Since (5.18) says that the set of common divisors of a and b are equal to the set of common divisors of a mod b and b, given that $a > b$. This concludes that $gcd(a, b) = gcd(a \bmod b, b)$ [Lauritzen, 2003].

Example 2 This example will demonstrate how the Euclidean algorithm is used to find the greatest common divisor of 48 and 17.

$$gcd(48, 17) = gcd(17, 14) = gcd(14, 3) = gcd(3, 2) = gcd(2, 1) = gcd(1, 0) = 1 \tag{5.24}$$

Notice that the greatest common divisor of 48 and 17 in example 2 is calculated by recursively applying the fact from equation 5.23 untill $gcd(a, 0)$ is reach at which point the greatest common divisor is a. It is also possible to compute the greatest common divisor without recursive function calls, pseudocode for this can be seen in algorithm 1.

Algorithm 1 The Euclidean algorithm

Input: $a, b \in \mathbb{Z}$ and $a > b$
Output: $d = gcd(a, b)$

```
1: d ← a
2: while b ≠ 0 do
3:     t ← b
4:     b ← d mod b
5:     d ← t
6: end while
```

5.5 The extended Euclidean algorithm

The extended Euclidean algorithm is an extension to the Euclidean algorithm. The extension lies in that, apart from computing $gcd(a, b)$ it also computes $x, y \in \mathbb{Z}$ such that $gcd(a, b) = a \cdot x + b \cdot y$ [Lauritzen, 2003]. The extended Euclidean algorithm is especially useful if the values a and b are coprime, because then x is the modular multiplicative inverse of a modulo b. It is this special trait that is useful when generating keys for RSA, where the modular multiplicative inverse of e mod $\varphi(n)$ must be found [A. Menezes and Vanstone, 1996].

The algorithm is used to write:

$$ax + by = d \tag{5.25}$$

where a and b are input values and x, y, and d are output values, d being $gcd(a, b)$. The algorithm achieves this by working with 2 equations:

$$a \cdot u + b \cdot v = w \tag{5.26}$$

$$a \cdot x + b \cdot y = z \tag{5.27}$$

The equations begin like this:

$$a \cdot 1 + b \cdot 0 = a \tag{5.28}$$
$$a \cdot 0 + b \cdot 1 = b \tag{5.29}$$

The trick is then to modify the values u, v, x, y, w and z while maintaining that $gcd(w, z) = gcd(a, b)$. This is where the Euclidean algorithm is extended. To begin with w is replaced by $w = w \bmod z$. The values u and v then need to be updated:

$$a \cdot (u - qx) + b \cdot (v - qy) = w - qz \tag{5.30}$$

The equation 5.30 is true, if the value $q = w/z$. The equations 5.26 and 5.27 are then swapped to update the values x, y and z and the procedure is repeated.
This method of modifying the values continues until either w or z reaches 0, at which point the opposing value will equal $gcd(a, b)$. In this example w reaches zero, and the second equation then holds the output:

$$a \cdot u + b \cdot v = 0 \tag{5.31}$$
$$a \cdot x + b \cdot y = gcd(a, b) \tag{5.32}$$

Therefore x, y and $gcd(a, b)$ denoted d, are the output values. A generalization of the extended Euclidean algorithm can be see in algorithm 2.

Algorithm 2 The extended Euclidean algorithm

Input: $a, b \in \mathbb{N}$ where $a \geq b$
Output: $d = x \cdot a + y \cdot b = gcd(a, b)$

1: $x_2 \leftarrow 1$, $x_1 \leftarrow 0$, $y_2 \leftarrow 1$, $y_1 \leftarrow 1$
2: **while** $b < 0$ **do**
3: $q \leftarrow a/b$
4: $r \leftarrow a - qb$, $x \leftarrow x_2 - qx_1$, $y \leftarrow y_2 - qy_1$
5: $a \leftarrow b$, $b \leftarrow r$
6: $x_2 \leftarrow x_1$, $x_1 \leftarrow x$
7: $y_2 \leftarrow y_1$, $y_1 \leftarrow y$
8: **end while**
9: $d \leftarrow a$, $x \leftarrow x_2$, $y \leftarrow y_2$
10: Return d, x, y

5.5.1 Modular multiplicative inverses

The following explains how the modular multiplicative inverse of $a \bmod b$ *can be calculated using the extended Euclidean algorithm.*

Definition 3 The modular multiplicative inverse of $a \in \mathbb{Z}$ under modulo $b \in \mathbb{Z}$ is the integer $x \in \mathbb{Z}$ which satisfied equation 5.33.

$$a \cdot x \equiv 1 \pmod{b} \tag{5.33}$$

The values a and b must be coprime to satisfy the following equation, since 1 mod $b = 1$ it can be rewritten as:

$$a \cdot x \mod b = 1 \tag{5.34}$$

ax mod b can be also be rewritten as:

$$a \cdot x - qb = 1 \tag{5.35}$$

If $q = -y$ then values that satisfy equation 5.35 can be found using the extended Euclidean algorithm (Algorithm 2). However, the extended Euclidean algorithm can only find such values if and only if the greatest common divisor of a and b is 1. Thus the modular multiplicative inverse of a mod b does not exist, if a and b are not coprime. With this in mind, the extended Euclidean algorithm can be used to find the elusive private key hiding in equation 5.36.

$$d \cdot e \equiv 1 \pmod{\varphi(n)} \tag{5.36}$$

5.6 Proof of Euler's theorem

Before we can prove Euler's theorem, we need to prove some interesting properties of coprimes and the sets of all coprimes to a given integer. It is by far easiest to prove these things separately from the proof of Euler's theorem, since it would otherwise be rather complicated.

Proposition 3 If a is coprime to b and a is coprime to c, then a is also coprime to the product of b and c.

Proof When a is coprime to b their greatest common divisor is 1 (5.37), the same thing can be said about a and c (5.38). Using the Extended Euclidean algorithm we may write (5.39) and (5.40). By multiplication we get an equation that may be written as in 5.41 [Lauritzen, 2003].

$$\begin{aligned}
gcd(a,b) &= 1 && (5.37)\\
gcd(a,c) &= 1 && (5.38)\\
1 &= a \cdot u_1 + b \cdot v_1 && (5.39)\\
1 &= a \cdot u_2 + c \cdot v_2 && (5.40)\\
1 \cdot 1 &= (a \cdot u_1 + b \cdot v_1) \cdot (a \cdot u_2 + c \cdot v_2) &&\\
&= (u_1 \cdot u_2 \cdot a + u_1 \cdot v_2 \cdot c + u_2 \cdot v_1 \cdot b) \cdot a + v_1 \cdot v_2 \cdot b \cdot c && (5.41)\\
x &= (u_1 \cdot u_2 \cdot a + u_1 \cdot v_2 \cdot c + u_2 \cdot v_1 \cdot b) && (5.42)\\
y &= v_1 \cdot v_2 && (5.43)\\
1 &= x \cdot a + y \cdot b \cdot c && (5.44)
\end{aligned}$$

If we then set x and y as in equation 5.42 and 5.43 respectively, we may write 5.44. Which given the Extended Euclidean algorithm (section 5.5) only is possible if and only if the greatest common divisor of a and $b \cdot c$ is 1, thus we have proven proposition 3. □

In addition it is needed to define how a set of integers under modulo n is calculated, as this is used in the proof of Euler's theorem.

Definition 4 The set $\mathbb{Z}'_n$ denotes the set of all integers coprime to n, thus $\varphi(n) = |\mathbb{Z}'_n|$.

$$\mathbb{Z}'_n = \{x \in \mathbb{Z}_n | gcd(x, n) = 1\} \tag{5.45}$$

This can then be used to prove that the set $\mathbb{Z}'_{an}$ equals the set $\mathbb{Z}'_n$, which is the last aspect to be covered before the proof of Euler's theorem.

Proposition 4 The set $\mathbb{Z}'_{an}$, of $a \cdot a_i \bmod n$ for all $a_i \in \mathbb{Z}'_n$ and a given a coprime to n, thus a also in $\mathbb{Z}'_n$, is equal to the set of $\mathbb{Z}'_n$. This is the same as saying that $\mathbb{Z}'_n$ is equal to the set of all members of $\mathbb{Z}'_n$ multiplied by $a \in \mathbb{Z}'_n$ under modulo n.

$$\mathbb{Z}'_{an} = \{a \cdot a_i \bmod n | a_i \in \mathbb{Z}'_n\} \tag{5.46}$$
$$\mathbb{Z}'_{an} = \{a \cdot a_1 \bmod n, \ldots, a \cdot a_{\varphi(n)} \bmod n\} \quad a, a_1, \ldots, a_{\varphi(n)} \in \mathbb{Z}'_n \tag{5.47}$$
$$\mathbb{Z}'_{an} = \mathbb{Z}'_n \tag{5.48}$$

Proof We prove the proposition above by proving that all the elements of $\mathbb{Z}'_{an}$ are different, and that these elements are all coprime to n [Lauritzen, 2003].

Say that two elements in $\mathbb{Z}'_{an}$ are the same, then we may write as in 5.49. We may also write this with a congruence relation as in 5.50 We know by definition that a must be coprime to n, thus by the Extended Euclidean algorithm a modular multiplicative inverse a^{-1} of a modulo n must exist. If we multiply by the modular multiplicative inverse of a on both sides we get 5.51, and as $a \cdot a^{-1} \equiv 1 \pmod n$, we may write as in 5.52. Notice that 5.52 may also be written as 5.53. Since $a_i, a_j \in \mathbb{Z}'_n$ both a_i and a_j must be smaller than and coprime to n. Thus we can remove the mod$n$ since, any integer x smaller than n modulo n gives x, thus we may conclude that $a_i = a_j$ and all elements of $\mathbb{Z}'_{an}$ are different, thus there are $\varphi(n)$ elements in $\mathbb{Z}'_{an}$.

$$a \cdot a_i \bmod n = a \cdot a_j \bmod n \tag{5.49}$$
$$a \cdot a_i \equiv a \cdot a_j \pmod n \tag{5.50}$$
$$a \cdot a_i \cdot a^{-1} \equiv a \cdot a_j \cdot a^{-1} \pmod n \tag{5.51}$$
$$a_i \equiv a_j \pmod n \tag{5.52}$$
$$a_i \bmod n = a_j \bmod n \tag{5.53}$$

To prove that all elements of $\mathbb{Z}'_{an}$ are coprime to n is easy, since any element of $\mathbb{Z}'_{an}$ is a product of a and a_i an element of $\mathbb{Z}'_n$. Since both a and a_i are coprime to n, the product of these must also be coprime to n, given proposition 3.

We have now proven that there is $\varphi(n)$ different elements of $\mathbb{Z}'_{an}$ the same as the number of elements of $\mathbb{Z}'_n$, and all the elements of $\mathbb{Z}'_{an}$ are coprime to n. Thus $\mathbb{Z}'_{an} = \mathbb{Z}'_n$, since $\mathbb{Z}'_n$ contains all coprimes to n. □

Finally we have the tools to prove Euler's theorem as stated in proposition 1.

Proof Since we know that $\mathbb{Z}'_{an} = \mathbb{Z}'_n$ as proved above, the product sum of all elements of $\mathbb{Z}'_{an}$ and $\mathbb{Z}'_n$ must be equal, as can be seen in 5.54, where a is coprime n and $a_1, a_2, \ldots, a_{\varphi(n)}$ are the members of $\mathbb{Z}'_n$ [Lauritzen, 2003]. We may simplify this (5.54) by writing it with a congruence relation (5.55). By reordering the a on the left handside we may also write as in equation 5.56. Since we know that $a_1 \in \mathbb{Z}'_n$ then a_1 must be coprime to n, and thus by the Extended Euclidean algorithm there must exists a modular multiplicative inverse of a_1, here denoted a_1^{-1}. If we multiply by a_1^{-1} on both sides of the congruence relation we get equation 5.57, now remember that $a_1 \cdot a_1^{-1} \equiv 1 \pmod{n}$, since that is the definition of modular multiplicative inverse, we may replace $a_1 \cdot a_1^{-1}$ with 1. Doing this for elements of $\mathbb{Z}'_n$, e.g. $a_1, a_2, \ldots, a_{\varphi(n)}$ we get equation 5.58, thus we have proved Euler's theorem as stated in proposition 1.

$$
\begin{aligned}
(a \cdot a_1 \bmod n) \cdot (a \cdot a_2 \bmod n) \cdots & \\
\cdots (a \cdot a_{\varphi(n)} \bmod n) &= a_1 \cdot a_2 \cdots a_{\varphi(n)} && (5.54)\\
a \cdot a_1 \cdot a \cdot a_2 \cdots a \cdot a_{\varphi(n)} &\equiv a_1 \cdot a_2 \cdots a_{\varphi(n)} \pmod{n} && (5.55)\\
a^{\varphi(n)} \cdot a_1 \cdot a_2 \cdots a_{\varphi(n)} &\equiv a_1 \cdot a_2 \cdots a_{\varphi(n)} \pmod{n} && (5.56)\\
a^{\varphi(n)} \cdot a_1 \cdot a_1^{-1} \cdot a_2 \cdots a_{\varphi(n)} &\equiv a_1 \cdot a_1^{-1} \cdot a_2 \cdots a_{\varphi(n)} \pmod{n} && (5.57)\\
a^{\varphi(n)} &\equiv 1 \pmod{n} && (5.58)\\
& && (5.59)
\end{aligned}
$$

□

5.7 Generating prime numbers

This section describes methods used for generating and testing prime numbers for use in the RSA algorithm. This section will not discuss the generation of pseudorandom numbers, which are required to generate the prime numbers.

Finding prime numbers can be difficult. To find several prime numbers in an interval, one of the best ways is to simply remove the none prime numbers in the interval one by one. However, this is inefficient when only a few individual prime numbers are required, like for RSA key generation. It is instead preferred to generate a random number, round the number down to an odd number, and then either subtract or add two, and test the primality of the number, and repeat this until a prime number is found. This section will describe the use of the Miller-Rabin primality test.

5.7.1 Miller-Rabin primality test

The Miller-Rabin is based on the following fact: If n is an odd prime, then write $n - 1 = 2^s \cdot r$ where r is an odd number.
Then, if we let a be an integer that satisfies $gcd(a, n) = 1$, e.g. any integer less than n is coprime to n, either equation 5.60 or equation 5.61 must be true.

$$
\begin{aligned}
a^r &\equiv 1 \pmod{n} && (5.60)\\
a^{2^r \cdot j} &\equiv -1 \pmod{n} && (5.61)
\end{aligned}
$$

where $0 \leq j \leq s-1$ [A. Menezes and Vanstone, 1996].
It is however not enough to test an odd integer n with only one integer a. There exists $\varphi(n)/4$ numbers, where $\varphi(n)$ is the Euler φ function, which are known as 'strong liars' to each odd integer n. These numbers allow the odd integer n to pass the primality test without necessarily being a prime number. Numbers that are not "strong liars" but allow the odd integer n to pass are known as "strong witnesses". The algorithm should therefore be repeated several times with different values of a in order to increase the probability of a number being a prime number. However, there is still a chance that all the integers a are 'strong liars' and the test will return prime, even if n is not a prime number. The chance of this happening is very small, less than $(1/4)^o$ where o is the number of random integers used to test for primality [A. Menezes and Vanstone, 1996].

The full algorithm is displayed in algorithm 3

Algorithm 3 The Miller-Rabin primality test

```
def is_prime(n, t = 30):
    """ Test if a is a prime """
    if n == 2: return True
    if n%2==0 or n == 1: return False
    r = n-1
    s = 0
    while r%2 == 0:
        r = r/2
        s +=1
    for i in range(0,t):
        a = random_number(n-2)+2
        y = mod_exp(a,r,n)
        if y != 1 and y != n-1:
            j = 1
            while j < s and y != n-1:
                y = y**2 % n
                if y == 1: return False
                j += 1
            if y != n-1: return False
    return True
```

This algorithm can be rather slow. One way of making it faster is by reducing the number of random integers t however this also increases the chance that the test will fail and return True on a composite number. Another modification that can make the test run faster is to check if the number n is divisible by a prime number in the interval 1 to 2000, before running a Miller-Rabin primality test [Schneier, 1996].

CHAPTER 6

Mail encryption with RSA

6.1 Key generation

As mentioned earlier in this report the fundamentals of RSA encryption is

$$Encryption: \quad c = m^e \bmod n \tag{6.1}$$
$$Decryption: \quad m = c^d \bmod n \tag{6.2}$$

Here m is the message represented as an integer, (e, n) is the public key, and (d, n) is the private key. e, d and n are generated from two large randomly generated prime numbers. These will be denoted p and q, and are generated as mentioned in section 5.7. The product of p and q is denoted n, and this is half of both the public and private key (6.3). Afterwards $\varphi(n)$ is deduced from Euler's totient function, definition 2, $\varphi(n)$ is calculated as shown in equation 6.4.

$$n = pq \tag{6.3}$$
$$\varphi(n) = (p-1) \cdot (q-1) \tag{6.4}$$

With $\varphi(n)$ set it is possible to determine the public part e and the private part d. The public key e must be chosen while maintaining the conditions 6.5 and the equation 6.6, for instance a prime that satisfied (6.5) could be used. Once the public key e has been chosen the private key d can be calculated to satisfy equation 6.7.

$$1 < e < \varphi(n) \tag{6.5}$$
$$gcd(e, \varphi(n)) = 1 \tag{6.6}$$
$$ed \equiv 1 \bmod \varphi(n) \tag{6.7}$$

The safety lies in the fact that despite knowing e and n it is not possible to determine d without knowing $\varphi(n)$. $\varphi(n)$ can be found when the prime factors of n is known, from (6.3) and (6.4). But p and q are primes and cannot be deduced from n in polynominal time, and when choosing primes sufficiently large, 512 bits for instance, it is not possible to deduce them from n within any reasonable timeframe. Therefore only the individual in possession of d can read the message.

The message can only be kept secret if the numbers d, p, q and $\varphi(n)$ are kept secret, since these numbers are used to compute the keys or decrypt the message [A. Menezes and Vanstone, 1996].

6.2 Encryption and decryption

As mentioned in the above section RSA encrypts a message expressed in numbers. However, it is not possible to decrypt it if the message m is greater than n. Therefore our

email client uses the symmetric-key algorithm IDEA, with a randomly generated number, x, as a key to encrypt the message, and RSA to encrypt x. Below a snippet of our Python code used to encrypt a message can be seen. This code picks a pseudorandom number and uses it as the key for encrypting the message with IDEA. The base64 encoding is used because emails do not[1] carry binary data within the "plain/text" payload.

```
#encrypt message with IDEA and RSA
x = numbertheory.random_number(2**128)  #IDEA key
i = idea.IDEA(x, "CBC")     #Create instance of IDEA
#encrypt message with Idea and base64 encode
ciphertext = base64.b64encode(i.encrypt(message))
#Encrypt IDEA key with RSA and base64 encode the result
cipherkey = str(encrypt(x, recipient_public_key))
cipherkey = base64.b64encode(cipherkey)
```

6.2.1 Symmetric-key algorithms

Symmetric-key algorithms are algorithms that use either identical keys, or encryption and decryption subkeys from the same key, or shared secret as it is also called.

There are two main types of symmetric-key algorithms; stream-ciphers and block-ciphers. Stream-ciphers encrypt a message bit by bit, while block-ciphers encrypt block by block, usually 64 or 128 bits per block. In general symmetric-key algorithms are faster and more efficient than asymmetric-key algorithms, however once you have the key, you can decrypt any message encrypted using the algorithm. Communication is therefore often initiated with asymmetric-key algorithms, and symmetric-keys are exchanged once the communicating parties have verified each others identities [A. Menezes and Vanstone, 1996].

6.2.2 IDEA algorithm

International Data Encryption Algorithm (IDEA) is the successor to Data Encryption Standard (DES). IDEA uses a 128bit key to encrypt a 64bit datablock. IDEA is based on mixing three operations between integers. These operations are:

- $\oplus$ bitwise exclusive or (XOR)[2]
- $\boxplus$ Addition modulo 2^{16}
- $\odot$ (Modified) multiplication modulo $2^{16}+1$, "modified" because operands that has the value 0 is replaced by 2^{16} and if the result is 2^{16} it is replace by 0.

The IDEA algorithm runs these three operations over eight identical rounds, with a final ninth round which prepares the ciphertext or plaintext for output. The algorithm is initiated by splitting the 128bit key into six 16bit subkeys. The key must then be rotated 25 bits to the left and six more subkeys can be extracted. This process is repeated until 52 subkeys have been created. Six subkeys for each round one to eighth and four subkeys

[1] This might depend on SMTP implementation and email client, but it certainly would not be pretty if a user who did not use RaptorMail got a signed message from a RaptorMail user.

[2] Se appendix C.1 for descriptions of the bitwise operations

for the outputting round. Furthermore the 64bit datablock must be split into four 16bit parts denoted X_1 to X_4 [Schneier, 1996, chap. 13.9].

6.2.3 IDEA encryption / decryption process

The following describes the procedure for rounds one to eight. The procedure is the same for both encryption and decryption. The varying factor is the creation of the 52 subkeys, where the decryption subkeys are the inverse of the encryption keys. The following is quoted directly from the book "Applied Cryptography":

1. Multiply X_1 and the first subkey.
2. Add X_2 and the second subkey.
3. Add X_3 and the third subkey.
4. Multiply X_4 and the fourth subkey.
5. XOR the results of step 1 and 3.
6. XOR the results of step 2 and 4.
7. Multiply the results of step 5 with the fifth subkey.
8. Add the results of step 6 and 7.
9. Multiply the results of step 8 with the sixth subkey.
10. Add the results of step 7 and 9.
11. XOR the results of step 1 and 9.
12. XOR the results of step 3 and 9.
13. XOR the results of step 2 and 10.
14. XOR the results of step 4 and 10.

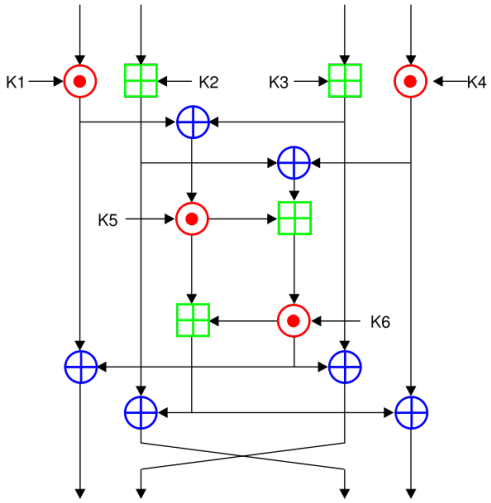


Figure 6.1: A diagram of one of the 8 rounds by Surachit [2008, Image]

The output for each round is step 11 to 14, however the output of rounds 12 and 13 are swapped. The final outputting round is special:

1. Multiply X_1 and the first subkey.
2. Add X_2 and the second subkey.
3. Add X_3 and the third subkey.
4. Multiply X_4 and the fourth subkey.

Quote Schneier [1996, chap. 13.9]

Below is a snippet of our C implementation of IDEA that runs the eight main rounds and the final outputting rounds as can be seen. IDEA was implemented in both Python and C. However, our Python implementation was horribly slow, therefore a C implementation was written. A discussion of this can be found in section 7.4: IDEA implementation considerations. In the C code below "mod_mul" is the function that performs the modified multiplication, and "mod_add" is the function that adds two integers under modulo 2^16.

```
/* Process a block
 * By Handbook of Applied Cryptography, Algorithm 7.101
```

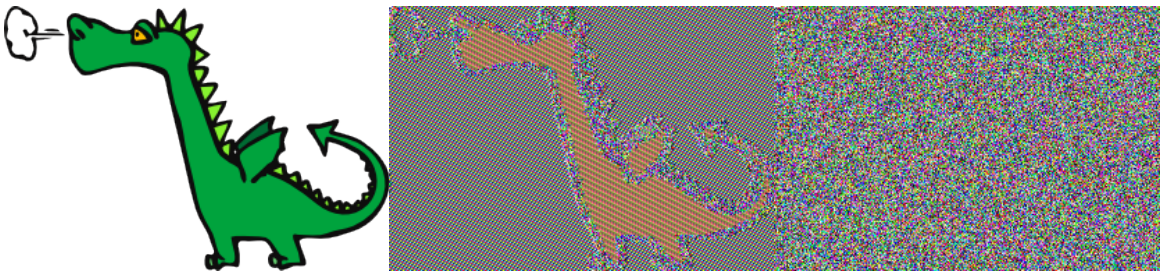
```
 */
void process_block(unsigned short *block, unsigned short* keys)
{
    /* Temporary variables */
    unsigned short t0, t1, t2;

    /* Run the eight main rounds */
    short i;
    for(i=0; i<8; i++){
        /* Step A */
        block[0] = mod_mul(block[0], keys[0+6*i]);
        block[3] = mod_mul(block[3], keys[3+6*i]);
        block[1] = mod_add(block[1], keys[1+6*i]);
        block[2] = mod_add(block[2], keys[2+6*i]);
        /* Step B */
        t0 = mod_mul(keys[4+6*i], block[0]^block[2]);
        t1 = mod_mul(keys[5+6*i], mod_add(t0, (block[1]^block[3])));
        t2 = mod_add(t0, t1);
        /* Step C */
        block[0] = block[0]^t1;
        block[3] = block[3]^t2;
        t0 = block[1]^t2;
        block[1] = block[2]^t1;
        block[2] = t0;
    }

    /* Run the output round */
    block[0] = mod_mul(block[0], keys[6*8]);
    t0 = block[1];
    block[1] = mod_add(block[2], keys[1+6*8]);
    block[2] = mod_add(t0, keys[2+6*8]);
    block[3] = mod_mul(block[3], keys[3+6*8]);
}
```

6.2.4 Modes of operation

In order to use a block cipher to encrypt data longer than the block size of the given block cipher, the data must be split into multiple blocks. Then the data can be encrypted block by block using the same key. This method is known as Electronic Code Book (ECB), but it is not a safe method of encrypting large amounts of data. If the block size is too small the ciphertext may fail to appear random and form patterns that may make it vulnerable to frequency analysis. A good example of this is images, especially graphics with a limited set of colors. Take a look at figure 6.2 and it is obvious that ECB mode is insufficient to hide the motive of the image, whereas other chaining operation modes such as CBC succeeds in making the ciphertext appear random. The images were encrypted in binary portable pixmap format (e.g. uncompressed), but the images headers were not encrypted in order to make it presentable.



(a) An unencrypted dino (b) An encrypted dino, using ECB (c) An encrypted dino, using CBC

Figure 6.2: Example of modes of operation

Cipher Block Chaining mode

Cipher Block Chaining mode, or CBC, is a method of handling multiple data blocks that require encryption. The principle of the CBC mode is to perform an XOR operation between the previous cipherblock and the current plaintext block, and then run it through the symmetric encryption algorithm in order to get the cipherblock of the current plaintext block. It is clear that an XOR operation between the first plaintext block and previous ciphertext block is not possible, as there initially will not be any ciphertext blocks. To fix this an initialization vector IV is used as the first ciphertext block. It is suggested that the IV value is unique for each session, but it is not required, and a zero value can be used. Because of the XOR operation with the previous cipherblock before encryption, the ciphertext becomes scrambled and will not form patterns like ECB mode. Decryption is equally simple. It is just a matter of an XOR operation between the previous cipherblock and the current unencrypted cipherblock [Schneier, 1996].

6.3 Encryption with RSA and IDEA

Since the IDEA-algorithm uses the random number x as the key in both encryption and decryption, the receiver needs to know it. However x can not be sent along in plaintext, since any intruder then can deduce M. Therefore x is encrypted with RSA, and sent along with the IDEA-encrypted message. The receiver can then use the private key to decipher x, and then use x to decipher M. Thereby the equations mention earlier (6.8) (6.9) will not encrypt the message M but the IDEA key, x.

$$Encryption: \quad c = x^e \bmod n \tag{6.8}$$

$$Decryption: \quad x = c^d \bmod n \tag{6.9}$$

However, raising a number by the power of a 512bit number is not a ressource friendly operation. The following section will describe methods of modular exponentiation.

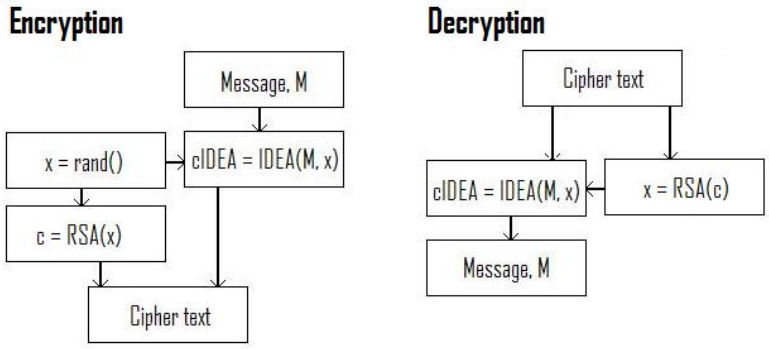


Figure 6.3: The encryption and decryption structure with RSA and IDEA

6.4 Exponentiation by repeated squaring

Exponentiation of a number can be done by multiplying the number with itself, and exponentiation of x to the power of 2 can be done as shown in equation 6.10 below. However, this technique can be slow since many multiplications are needed when the exponent is large. This is shown in (6.11) where the exponent is 10. However, rewriting (6.11) to (6.12) makes the calculation of the number a lot faster. (6.11) requires 9 multiplications, while (6.12) only requires 5, since $x^{2^2} = x^2 \cdot x^2$ and x^2 can be reused once first calculated. The difference is small between (6.11) and (6.12), but it has a large effect when exponentiating with greater exponents.

Example 3

$$x^2 = x \cdot x \tag{6.10}$$
$$x^{10} = x \cdot x \cdot x \cdot x \cdot x \cdot x \cdot x \cdot x \cdot x \cdot x \tag{6.11}$$
$$= x^2 \cdot (((x^2)^2)^2) \tag{6.12}$$

6.5 Binary exponentiation

The computer represents all numbers in binary. These binary representations can be used to create an algorithm which is much faster at exponentiating numbers. For an introduction to binary numbers see appendix C.1.1 [A. Menezes and Vanstone, 1996].

6.5.1 Exponentiating theory

A general formula to raise a given integer x to power of k is shown below:

Definition 5

$$x^k = x^{k_0 \cdot 2^0)} \cdot x^{(k_1 \cdot 2^1)} \cdots x^{(k_n \cdot 2^n)} \tag{6.13}$$

where k_n represents the n'th bit in the binary representation of k.

Example 4

$$1010_2 = 10 \tag{6.14}$$
$$x^{1010_2} = x^{(1\cdot 2^3)} \cdot x^{(0\cdot 2^2)} \cdot x^{(1\cdot 2^1)} \cdot x^{(0\cdot 2^0)} \tag{6.15}$$
$$x^{1010_2} = x^8 \cdot 1 \cdot x^2 \cdot 1 \tag{6.16}$$
$$x^{1010_2} = x^{10} \tag{6.17}$$

When exponentiating x^{0^2} the result is 1 which is discarded as $x \cdot 1 = x$.

6.5.2 The Python algorithm

Algorithm 4 is the Python implementation of modular exponentiation as used in RaptorMail. For an explanation of the operator & see appendix C.1.2.

Algorithm 4 Modular exponentiation

```
def mod_exp(a, k, n):
  """ Modular exponentiation
      Returns: a**k mod n
      Implementation of repeated square
      and multiply for exponentiation in Z_n
  """
    if k < 0:   #If k is negative
        a = invert(a,n) #Set a to the modular multiplicative
                        # inverse of a under modulo n
        k = -k          #And make k positive
    b = 1
    if k == 0: return b #If k is 0 then return 1
    # A is set to a to preserve the value a.
    A = a
    for i in range(0, bitlength(abs(k))):
        if k & (1 << i): b = (A * b) % n
        A = (A * A) % n
    return b
```

First the exponent k is checked to be negative. If k is negative, the result is the modular multiplicative inverse of a is raised to the power of $|k|$. Then if k is 0 the result is 1, since $a^0 = 1$ for all $a \in \mathbb{Z}$.

Then for each bit i in k, we set $b = b \cdot A \bmod n$ if the bit i is 1. We check if bit number i is 1 by using bitwise AND and bitshift operations, see appendix C.1. Once we have done that we raise A to the power of 2 under modulo n, and continue to the next bit. Finally the algorithm returns the b [A. Menezes and Vanstone, 1996].

By performing the modulo operation in each step through the loop, the load on the computer is reduced and the algorithm is faster because the numbers remain as small as possible.

6.6 Signature with RSA

The basic principle of digitally signing messages is described in section 5.2.2. The following section will describe this process in more depth. To begin with, an understanding of hash algorithms must be obtained.

6.6.1 Hash-function

Hashing functions can be used for different things. Some are used to search through large amounts of data and some for making sure a message has not changed from it being sent to being received.

In relation to RSA encryption, hashing is used in signing messages. It is used to ensure that the message integrity is intact, based on the requirement that it should be difficult, if not impossible, to make a message with the same hash value - otherwise, Bob could not be sure, the message was really written by Alice. A simple hashing method is to XOR the file - this can be useful for looking up files in a database, but it is not safe to use in a cryptographic context.

A hashing function must be one-way, to be safe. This means that given the hash value, it should be very difficult, and likely impossible, to compute the original message. It should also be hard to find another message that gives the same hash value, otherwise the message could be substituted, and it would be impossible to know the difference.

Three rules must apply, for the function to be one-way: M is the message, h is the hash value of the message and $H(M)$ is the hash function that converts M to h.

- When M is known, h is easy to compute.
- When only h is known, it must be hard, if not impossible, to compute M, so that $H(M) = h$.
- When M is known, it must be hard to find another M', so that $H(M') = H(M)$.

For the purpose of this project, we will be using SHA, as it is a hash function widely used and respected [Landrock and Nissen, 1997].

6.7 Digitally Signed email

When a unique hash value of the message has been created, it can be used along with RSA. The sender encrypts the hash value with his private key.

$$h = H(M) \tag{6.18}$$

$$Signature = h^d \bmod n \tag{6.19}$$

In algorithm 5 a snippet of our Python code to compute the signature of a message can be seen.

The signature (6.19) is then sent along with the message. When the receiver wants to confirm the integrity of the email, he decrypts the signature with the sender´s public key (6.20).

Algorithm 5 Signature generation

```
def sign(m,d):
    """
        Sign the message $m$ using the secret Key d
        Return encryped Hash value
    """
    a = sha.new(m).digest() #return the sucure hash value
    b = struct.unpack("B"*20,a)
    hashvalue = 0
    for i in range(0,20):
        hashvalue += b[19-i]*(256**i)

    c = hex(encrypt(hashvalue,d))

    #parse the output c
    c = c[2:]
    if c[-1] == "L":
        c = c[:-1]
    return c
```

$$h = Signature^e \bmod n \tag{6.20}$$

The hash value of the current message must also be generated by the receiver, who can then compare it to the hash value sent in the signature.

$$h = H(M) \tag{6.21}$$

If they match the receiver can conclude that the person who created the mail was in possession of the private key, and that no one has altered the message.

6.8 Complexity theory

The following section presents methods for classifying a problem based on its complexity, e.g. resources, such as time and space, needed to solve the problem. Furthermore it explains the integer factorisation problem.

6.8.1 Turing machines

When investigating the amount of resources, time, space and processes, used by an algorithm to compute a certain result, it is necessary to define how a computation is performed. Usually we would express an algorithm as the steps we perform in order to get a result. Thus, an algorithm for doubling a given number could be described as shown below.

Algorithm 6 Algorithm for doubling a given number
Input: $a \in \mathbb{R}$
Output: $r = 2 \cdot a$
1: $r \leftarrow a + a$

The algorithm above is very straightforward, and when discussing its resource usage one may get the impression that it takes one process one step to evaluate the algorithm using twice as much memory as a allocates. However, it can be concluded that this completely depends on your programming language, compiler and hardware.

Most modern processors and microprocessors will let you read a number from input to internal memory in one instruction, and then bit shift[3] one left in one instruction in which case the computation would be performed in two steps. This requires the same amount of memory as a, given that the result is not larger than the register size the processor operates with.

The problem when discussing resource usage is that it depends on the platform on which a computation is performed. Thus in order to work with resource usage we need a common platform for defining computation. However, benchmarking different algorithms against each other on a common hardware platform is not preferable either, since some algorithms cannot be evaluated in a realistic time on any known hardware. A Turing machine, proposed by Alan Turing in 1936 [Sipser, 2006], is a fundamental mathematical model for computation, that can be used to calculate how many steps a computation takes, without actually having to execute these steps.

A Turing machine has a finite set of states; a current state; an unlimited tape; a transition function that, given the current state and an input character, returns a new state; an output character and a direction left or right to move the tape head. A Turing machine has unlimited memory, which is accessed as a tape. This tape is read by a reading head and with each step of the transition function, it writes a new character and moves either right or left. There are many variations of Turing machines, but there is a common formal definition, as given in John E. Hopcroft and Ullman [2001], a similar definition can also be found in Sipser [2006].

Definition 6 A Turing machine TM is tuple of 7 components $TM = \langle Q, \Sigma, \Gamma, b, \delta, q_0, F \rangle$ where

Q is the set of states,
Γ is the set of tape characters,
$\Sigma \subset \Gamma$ is the set of input characters,
$b \in \Gamma$ is a blank character that appears infinitely on the tape,
δ is the transistion function which takes a current state $q \in Q$ and a read character $\gamma \in \Gamma$, and returns a new state, a new character to write to the tape and direction $\{L, R\}$ to move the head.
$q_0 \in Q$ is the initial state, and
$F \subset Q$ is the set of finishing states

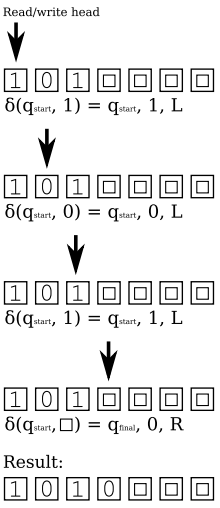
[3]Se appendix C.1 for explanation of bit shifting.

Example 5 A simple Turing machine that multiplies any given binary number by 2, see figure 6.4 for an example of how to evaluate it.

$Q = \{q_{start}, q_{final}\}$
$\Gamma = \{0, 1, \square\}$
$\Sigma = \{0, 1\}$
$b = \square$
$\delta(q, \gamma)$: if $\gamma \neq \square$
    return $\langle q_{start}, \gamma, R\rangle$ else return $\langle q_{final}, 0, L\rangle$
$q_0 = q_{start}$
$F = \{q_{final}\}$

Before execution, a Turing machine is in its start state and the input for the machine is written to the tape starting from the initial position of the read/write head and followed by blank characters. Then, when executed, the current input character under the read/write head is read and passed to the transition function along with the current state. The transition function then returns a new state, a character to write to the tape and a direction to move the read/write head. This step is then repeated until the machines reaches a terminating state.

Figure 6.4: Evaluation of example 5.



If we look at the Turing machine from example 5 give it 101_2 as input and take a snapshot of the tape and current state for each step of the execution, the snapshots would look like figure 6.4. As can be seen from figure 6.4 it takes the Turing machine 4 steps to reach a terminating state. Notice that the Turing machine doubles the integer 5 written as 101_2 in binary to 10 written as 1010_2 in binary.

Recall the discussion that it would take most processors 2 steps to evaluate algorithm 6, given that the result was smaller than the register size of the processor being operated on. However, given an integer of arbitrary size, most common processors would probably evaluate in a timeframe propertional to the length of the input, under the restriction that the result is within the memory limits of the computer. It is interesting to see that the Turing machine from example 5 evaluates in time proportional with the length of the input, given an integer of arbitrary size.

There is more than one type of Turing machine - for instance a single tape machine as described above, multiple tape machines and non-deterministic Turing machines. However, a multiple tape Turing machine may be emulated using a single tape Turing machine[Sipser, 2006]. Thus there is really only two different types of Turing machines, the deterministic and the non-deterministic Turing machine. The deterministic Turing machine is the one described in the previous paragraph, which is a good model of an ordinary computer and its limitations. The non-deterministic Turing machine basically operates in the same manner as the deterministic Turing machine, except for the extra feature that it can branch into multiple Turing machines evaluating at the same time. If a deterministic Turing machine is viewed as a normal computer with unlimited memory, a non-deterministic Turing machine may be viewed as a computer with unlimited memory and an unlimited number of processors.

Turing machines are only used as mathematical models of how computation is performed, as the description of these machines is very basic and not meant for expressing algorithms. However, when working with algorithms we can always go back and look at Turing machines to see if a computation is within the capabilities of a computer.

6.8.2 Big-O notation

When looking at a Turing machine, in its formal definition, it should be possible to determine the maximum operating time as a function of input of length n. If $f(n)$ denotes this function, then $f(n) = 5{\cdot}n^3{+}4{\cdot}n$ means that the Turing machine evaluates in $5{\cdot}n^3{+}4{\cdot}n$ given an input of the length n. However, calculating the exact maximum operating time given an arbitrary input length, can be quite difficult. Thus, it is normal to only look at the term of the highest order, as it is the rate at which the maximum operation time grows the most, when the input length grows. This is called big-O notation.
For an algorithm that evaluates in $5 \cdot n^3 + 4 \cdot n$ steps, the big-O notation of the operation time would be $O(n^3)$. $4 \cdot n$ is disregarded because it has an infinitely small influence when $n \to \infty$, and the constant factor 5 is disregarded too [Sipser, 2006].

It may not seem sensible to disregard all details, however. Generally it is only interesting how the operating time grows with respect to the input length, in a worst case scenario. Besides, when determining the big-O notation, operating time for an algorithm, calculating all the details is usually not necessary, as it is easier to see how fast the operation time grows in big-O notation. Recall the Turing machine from example 5 that can multiply by 2. It has an exact operating time of $f(n) = n + 1$ - in big O notation the operating time would be $O(n)$, as the constant 1 has an infinitely small impact when $n \to \infty$.

6.8.3 Complexity classes

Problems are categorised into different classes, depending on the complexity of the most efficient algorithm, which can solve them. This is a basic introduction to two of these classes, since a detailed description of these classes is beyond the scope of this report.

The P-Class

All problems that can be solved in polynomial time, with respect to the input length, are in the complexity class P [Sipser, 2006]. That is any problem, which can be solved by a deterministic Turing machine in polynomial time, with respect to the input length, is in the complexity class P. Formulated with big-O notation all problems, which can be solved using an algorithm that evaluates in $O(n^k)$ time for a constant k are in the complexity class P. This means that the problem of doubling a given integer is in P, since the Turing machine from example 5 can solve the problem in $O(n)$ time, which is polynomial time with respect to input length n. In short every problem that can be solved in polynomial time is in P.

The NP-Class

All problems for which a solution can be verified using a *certificate*, in polynomial time with respect to the input is in the complexity class NP. A certificate to a solution for a problem is some information that enables verification of the solution [Sipser, 2006].

All problems in P are thereby also in NP, since given a solution to a problem, a solution can be computed and compared to the given solution in polynomial time. Thus since it is possible to verify the solution to any problem in P in polynomial time, all problems in P are also in NP.

The integer factorization problem, finding prime factors of a given integer, is in NP, but not in P. Given the prime factors of an integer, as certificate, we can easily check if the product of those primes is the integer. Using primality tests as discussed in section 5.7.1, the primality of the certificate can easily be verified. Thus, the integer factorization problem is in NP, but not in P since no algorithm is known to be able to factor integers in polynomial time [A. Menezes and Vanstone, 1996].

6.8.4 Polynomial time reduction

Polynomial time reduction is to reduce one problem to another problem, in polynomial time.

A simple example could be to prove that the doubling of an integer, $m = n + n$, is a problem which can be solved in polynomial time. If it is assumed that the multiplication of two integers can be done in polynomial time, there is a point in rewriting m: $m = 2 \cdot n$. Now m is the product of two integers and must be solvable in polynomial time.

This was a relatively simple example, but it shows that some problems can be reduced to other problems. Then, if one problem can be solved, so can the other. It can also be used to show that a problem is just as easy as, or easier than another problem.

6.8.5 Complexity of the RSA problem

The RSA problem is, given the public key e, n and a ciphertext $c = m^e \bmod n$, it is not possible to recover the plaintext m in polynomial time. No algorithm to determine m, given e, n, c is known to operate in polynomial time [A. Menezes and Vanstone, 1996]. Simply multiplying c by c under modulo n d times until the result is m is not efficient, as the time to do this would be proportional with the size of d, and the size of d is exponential to the length of d.

Instead of just trying to multiply c by c d times we would be able to reconstruct d if we knew $\varphi(n)$. However, the only efficient method of computing $\varphi(n)$ involves finding the prime factors of n, which we have no algorithm for computing in polynomial time. But we may conclude that the RSA problem is polynomial reducable to the problem of computing the totient function $\varphi(n)$ to an arbitrary integer, which is polynomial reducable to the problem of integer factorization. Thus if either the integer factorization or the problem of computing the totient function is solved, so that either one of them can be computed in polynomial time, then the RSA problem may also be solved in polynomial time and the RSA cryptosystem would be broken.

6.9 Integer factorisation

The security of RSA depends on the difficulty of factoring large numbers.

All natural numbers greater than 1 have a unique prime factorisation.

Definition 7 Given a positive integer n, the prime factorisation is:

$$n = p_1^{e_1} p_2^{e_2} ... p_k^{e_k} \tag{6.22}$$

where p_i is a distinct prime and each $e_i \geq 1$ [A. Menezes and Vanstone, 1996, 3]

Every natural number n≥1 can be factored into a unique product of primes:

$$n = p_1 p_2 ... p_k \tag{6.23}$$

Proof We assume n > 1. Suppose that n factors into two distinct set of primes:

$$n = p_1 ... p_r = q_1 ... q_s \tag{6.24}$$

We divide by all primes present in both strings, so that no p_j appears in both.
We already know that if a number $m|ab$, $m|a$ if $m \nmid b$ and vice versa.
Thus, as $p_1|n$ we can deduce that either $p_1|q_1$ or $p_1|q_2$ or... or $p_1|q_s$. As we had already removed every common prime factor, this cannot be, and thus our assumption has been proven [Lauritzen, 2003]. □

RSA encryption is built upon the fact that factoring large numbers is quite time consuming. There is still no fast way of doing it, so finding the prime factors p and q of n is not easy.

$$n = pq \tag{6.25}$$

The safety of RSA depends on the difficulty of factoring an integer. As any given natural number has a unique factoring into primes, there are no other numbers that fit n. Thus once one of the numbers has been found, the other is quite easy. And so, the safety of RSA depends on p and q being so big that it is impossible to solve in any reasonable timeframe [A. Menezes and Vanstone, 1996, Chap. 3].

CHAPTER 7

RaptorMail, an encrypting GMail client

7.1 Tools and libraries

This section discusses which tools and libraries we decided to use during the development of RaptorMail.

We have decided to use Python for the development of RaptorMail, even though we currently have a course in C programming. Python was chosen, because many of us have little to no programming experience and Python is an easy-to-learn highlevel language, with an extensive set of libraries. Specifically not having to manage dynamic memory allocation is an advantage, since this is a rather complex issue, especially for novice developers. This feature comes at the expense of performance, though for the most part this is not a critical issue.

Another nice feature of Python is its type system, which does not require the programmer to explicitly declare variable type. Also it is not, as in C, possible to accidentally cast data to any arbitrary type. However, sometimes, as we will show later, this feature makes it virtually impossible to implement some data processing algorithms efficiently, which is why the IDEA implementation was written in C as well.

To create the graphical user interface we decided to use GTK, as it is free, platform independent and has good Python bindings. We have also used Glade, a graphical user interface builder for GTK, to build the graphical user interface. This has enabled us to develop the graphical user interface rather quickly. To retrieve messages from GMail we have used the IMAP library, which is a part for Python standard libraries. To store emails locally we have used SQLite, a small embeddable serverless SQL database engine, with Python bindings. Using a database for storing emails enables us to search mails and determine relations between different mails, such as labels and conversations, as known from the GMail web interface.

7.2 Interfacing GMail

How to interface GMail using IMAP and SMTP for receiving and sending messages respectively.

7.2.1 Internet Message Access Protocol

The Internet Message Access Protocol, abbreviated *IMAP*, is an email retrieval protocol like POP3. An email retrieval protocol is used between email servers and email clients, and enables email clients to get emails from the servers. An email retrieval protocol is not used for transmitting emails. Other protocols such as *SMTP* can be used for this purpose.

The two most popular email retrieval protocols are POP3 and IMAP. POP3 only allows a single connection per user at a time, and it only allows users to retrieve new emails and then asks, if the user wishes to delete the message from the POP3 server once retrieved. IMAP on the other hand, allows users to upload emails and categorise emails into different mailboxes[1]. Nor does IMAP prevent the user from having multiple connections to a server at the same time. IMAP also allows serverside searches and partial fetching, e.g. parse and search or fetch a particular part of an RFC2822 formatted email [Crispin, 2003].

Compared to IMAP, POP3 is a lousy email retrieval protocol, whereas IMAP enables synchronisation between different email clients. POP3 is only designed for short-term connections, while the users receives new emails, while IMAP enables the connection to be kept open while the user reads his mail.

GMail supports both IMAP and POP3. When accessed through POP3, the user can ask GMail to either archive, delete or leave all emails accessed in the inbox. When accessed through IMAP, there are no such complex settings on the GMail serverside. Since IMAP supports different mailboxes, labels as known from the GMail web interface can be representated as mailboxes, same thing goes for the special labels such as INBOX, Spam, Trash, All mails, etc. Emails can also be flagged as seen, using the IMAP protocol, so most of GMails features can be accessed and controlled using an IMAP interface. For this reason we have chosen to use the IMAP interface for our encrypting GMail frontend.

GMail synchronisation over IMAP

RaptorMail has a local database, which contains the same information as is available through the GMail web interface. That is: labels, emails and relations between emails. This database also contains actions that are to be executed on the remote server, when the database is synchronised. The synchronisation process is highly inspired by Austein [1994], and is as follows:

1. Synchronise list of labels/mailboxes
   (a) Fetch list of labels from server
   (b) Add new labels to database
   (c) Remove remotely deleted labels
2. Process pending actions
   (a) Delete emails marked for deletion
   (b) Create new label associations
   (c) Remove label associations
   (d) Flag read emails as seen
3. Fetch new emails and label associations, by iterating the following for each label
   (a) Fetch Message-IDs of messages from last known UID[2] for the label
   (b) Download and store messages if they were not in the database
   (c) Add label associations if message was not known to be in the label

[1] IMAP terminology for folders or directories.
[2] Universal Identification

When interfacing GMail, messages are distinguished using their Message-ID from the RFC2822 formated header. By fetching all Message-IDs from a mailbox, it is possible to decide whether or not a particular message is within the mailbox/label. Ideally some sort of IDs should have been used, however, IMAP does not support labels or tags, so to an IMAP aware application a message would normally exist twice if it is in two different labels. IMAP does have a unique identifier called UID, however, it does not persist across different mailboxes [Crispin, 2003]. The local emails are identified by their Message-ID, and when executing actions, such as deleting emails (step 2a) we first select the "[Gmail]/All Mail" mailbox and search for the UID of a message with a given Message-ID. Once that UID is found, it is flagged for deletion and an expunge command is executed. For more specific information, see sourcecode or Crispin [2003].

7.2.2 Simple Mail Transfer Protocol

Simple Mail Transfer Protocol, abbreviated *SMTP*, is a protocol for transmitting emails. SMTP is used to transmit emails from client to email server, e.g. outgoing email whereas IMAP handles incoming email. In RaptorMail GMail's SMTP server is interfaced using the "smtplib" which is one of Python's standard libraries. All we do to send an email is initiating the connection, initiate TLS[3], authenticate against the server, pass the raw RFC2822 formated email to be transmitted along with the to and from addresses and then close the connection.

As mentioned before SMTP does not ensure the integrity of the email that is being delivered, if the traffic is being routed through insecure channels, nor does SMTP guarantee that the email address in the "from" field actually belongs to the entity who sent the email [Klensin, 2008]. Therefore, as discussed before this is why asymmetric end to end cryptography ought to be used by everyone.

7.3 Graphical User Interface with GTK

This section will work through some of the technical issues of working with GTK for a graphical user interface.

7.3.1 Glade integration with Python

To connect the Python program with the Glade file, the "signal_autoconnect" function is used. Below is the Python code from the constructor.

```
class GladeWindow:
    def __init__(self, gladefile, window):
        #Load the window
        self.widgets = gtk.glade.XML(gladefile, window)
        #Connect signals to methods on this instance
        self.widgets.signal_autoconnect(self);
```

[3]TLS is a secure layer ontop of TCP which provides authentication and security through asymmetric cryptography.

```
        for widget in self.widgets.get_widget_prefix(''):
            name = widget.get_name()
            assert not hasattr(self, name)
            setattr(self, name, widget)
```

The name of the Glade file and the name of the Glade widget is given to the constructor as the arguments *gladefile* and *window*. The Glade file parameter is the path of the Glade xml file, and the window parameter is the name of the toplevel widget to be loaded from the Glade file. The toplevel widget is usually a window, and when loaded using the "XML" function all the child widgets of this widget are loaded too.

Then the "signal_autoconnect" function connects methods on the instance self, with signals in the Glade file. This way a signal called close would execute a method called close on the GladeWindow class or derivated class of this, if a method called close exists on the instance self.

Then a *for* loop runs through *self.widgets* and selects all those with the prefix "" - which results in all widgets being selected. The widget name is saved in *name* using the function "get_name()".

The "hasattr" function checks whether the string *name* is the name of one of the attributes on *self*, that is if *name* is set. If so, it returns *True*. Assert raises an assertion if it is given the boolean value False, but because of the "not" it raises an assertion if *hassattr* returns *True*. Thereby this line checks if *self* already has an attribute by the name *name*. This should not be the case, and an assertion is raised. If *self.name* does not exist already, nothing happens. However it is important to remember that assert is only executed if the document is run in debugging mode. If run in runtime mode, the assert is ignored.

The "setattr" function assigns the value of *widget* (or a reference to this value) to the attribute *name* of the object *self* - that is *self.name = widget*. This is why assertion is useful during debugging, as this may overwrite existing attributes.

As this function completely depends on the inputs *self*, *gladefile* and *window*, it can be inherited to all other Python documents. To import the base class in other Python documents, it is only required to import the python document containing the gladeWindow class and writing the below as the base class:

```
class myClass(gladeWindow.GladeWindow):
    def __init__(self):
        gladeWindow.GladeWindow.__init__(self, "program.glade",
                                                    "MainWindow")
```

where *myClass* is the name of the base class, *program.glade* is the name of the Glade program, and *MainWindow* is the name of the toplevel Glade widget. Thereby the right values will be transfered to the GladeWindow class. The result in calling the base class with the given arguments is that every widget and attribute can be called by "self" and its name as defined in the Glade file.

7.3.2 Threading with PyGTK

When executing a program, the program can be thought of as a sequence of commands being processed one by one - if the program is single threaded, that is. When a command is

being executed, it is being executed by a thread, so if two commands are being executed at once we must have two threads. A thread can be thought of as a midget with a calculator performing simple commands that is passed to him, having multiple midgets then enables the execution of multiple commands at once.

A thread is kind of like a process, since two processes can execute two commands at once too. A detailed description of the difference between processes and threads is far beyond the scope of this report, however, in short the difference is that processes are managed by the operating system, whereas threads are managed by the program. This means that threads within a program can share memory, while sharing data between processes can be far more complicated. However, if two threads access the same memory at once, bad things may happen.

When writing applications with a graphical user interface that communicates over the Internet it is an absolute necessity to use multiple threads. Otherwise the user experience and usability of the application will suffer greatly. To understand this, it is necessary to understand that communication over network causes a thread to wait for a response, and thus while this thread is waiting for a response, it cannot perform any other commands. So if RaptorMail was not multithreaded the UI[4] would not update while the application was synchronising with GMail. Failure to update the UI causes the users to believe the application has frozen or crashed. Multithreading is also needed if computational expensive tasks are being performed, such as generating a keyset, since the UI would otherwise freeze while the key was being generated.

Threading in Python is not so difficult, simply import the threading module and derivate a class from the ”threading.Thread” class, and overwrite its ”run” method. Python also locks variables so that while a variable is being accessed from one thread, it cannot be accessed from any other threads. The other threads will then wait until the first thread is done, before they can gain access. However, when interfacing external libraries, some of these may not be thread-safe[5], as is the case for both SQLite and GTK.

A PySQLite connection complains if it is accessed from a different thread than it was created on, however, this issue is solved by creating a new database connection for each thread. In the class ”GMail” of the ”gmail_cache” module the ”get_database” method checks if the thread name matches the thread name, which the instance was initialised on. If so, it returns the default database connection, if not it creates a new connection. The method ”return_database” then closes the database connection, unless the connection is the default connection. This way, methods of the ”GMail” class are threadsafe, and opening too many database connections is avoided, which could have resulted in performance issues.

GTK, as many other popular graphical user interface toolkits, is not thread-safe. This means that you cannot access the UI from a worker thread. A hack that gets you around this issue is to implement a custom mainloop, and add stuff that needs to be handled on a queue. This is an ugly hack, though. Alternately, the PyGTK bindings for GTK offers two functions called ”threads_enter”[6] and ”threads_leave”, the first one marks the beginning of a code segment that only one thread may execute, the second marks the end

[4]UI is short of **U**ser **I**nterface.

[5]An object is *thread-safe* if it can be accessed from multiple threads at once.

[6]These functions are available on the ”gtk.gdk” class.

of such segment. Using these functions, the progress bar can be updated from a worker thread. For these two function to work, the "threads_init" must be called from the main thread before the mainloop is started [PyG, 2008].

7.4 IDEA implementation considerations

For many purposes Python is a functional language, although data processing is not one of its strengths. We specifically experienced this while implementing IDEA. This sections discusses some of these issues and how they can be addressed using native C modules.

The IDEA implementation for RaptorMail can be found in "idea.py". It is a class that needs a key for initialisation. Optionally it can also be given a mode of operation (ECB or CBC), and an implementation to use: native, managed[7] or auto. The native implementation requires the compilation and installation of the "idea_optimized module", which is written in C.

When encrypting a message using IDEA, the message is first padded to be a multiple of 64bits, or, in other words, be a multiple of 8 characters. This is done by adding a single character with the byte value of 1 plus the number of null characters to follow before a multiple of 8 characters is reached. This single character will always be added, so a message that is already a multiple of 8 will be padded with a $0x08$ byte and 7 $0x00$ bytes. In our IDEA implementation the message is just regarded as a raw byte stream, this means that 16bit utf8 characters are regarded as two bytes. Also the message is not compressed which would protect against some attacks. Once the message has been padded, it is encrypted either using the managed or the optimized unmanaged implementation.

The managed implementation first converts the message from string into a list of lists of 4 integers of 16 bit. To do this the "unpack" function of the "struct" module is used. It reads the byte value of a string and converts it to an integer value. Once this is done these values are added to a list of list of integers. The first list of the list of blocks contains a list of 4 integers which is a block in IDEA. When the message string has been converted to a list of blocks, each block is encrypted, xor'ed with the previous block if in CBC mode, and added to the list of encrypted blocks. This process is iterated for the entire list of blocks. When all blocks have been encrypted the list of encrypted blocks is run through and algorithm that converts each block into a string using the "pack" function of the "struct" module and joins the resulting string into the final ciphertext.

Encrypting a message as described above is possible, but Python is not very good at data processing, and this comes to show here. While our Python implementation of IDEA probably is anything but efficient, it is most certainly unreasonably slow. It is believed that the problem is caused by the fact that Python's datatypes are memory managed and less limited than those of other languages. While this normally a good thing, it comes at the price of performance. When creating the list of blocks, we use a lot of integers, none of which need to be more than 16 bits long, nevertheless Python's integer type is of arbitrary size and thus checks to see if it needs to allocate more memory whenever it is assigned a new value. The same thing goes for lists. These also allocate memory dynamically, and this can be the cause of bad performance. Most likely the issue

[7]Managed implementation means that memory allocation is managed by a runtime, e.g. managed implementation is a Python implementation.

is that we edit values and assign values to a list when we repeat the rounds of the IDEA encryption algorithm. So it is most likely possible to optimize our implementation to a level where the performance may be acceptable, however, we chose to implement the critical parts of the algorithm as a C module instead.

When encrypting a message using the native implementation the padded message string is passed, along with encryption key and a string representation of operations mode, to the "encrypt" function in the "optimized_idea" module, and this function then returns the ciphertext as a string. The "optimized_idea" module is written in C and built against the Python C API. First it parses the arguments using the "PyArg_ParseTuple" function, then it checks that the length of the input string is a multiple of 64bits. Once this is done it uses the function "PyString_FromStringAndSize" to allocate an uninitialized string with length of the ciphertext. A pointer[8] to the string allocated for the ciphertext is obtained using "PyString_AsString". This pointer is cast to an "unsigned short"[9] array, and the input string is copied to this pointer using the "memcpy" of the standard C library, so that four entries in this array represents a block. We then loop through the array incrementing the counter by 4 for each iteration, thus the pointer to the block array plus the counter will refer to a block during each iteration, this way all blocks are encrypted using the internal "process_block" function.

Encrypting messages using the native implementation is a lot faster than using the Python implementation we have written. This is probably because we only create one copy of the data, and thus only allocate $O(n)$ memory, whereas the Python implementation allocates for both plaintext, input blocks, output blocks and ciphertext. Another reason is that our native implementation, being written in C, can use a 16 bit integer type, and cast the input string to an array of this type instead of having to use a function to convert its byte value to an integer. The extensive use of lists also makes it very difficult to determine the total runtime of our Python implementation, whereas a qualified guess is that our native implementation executes in $O(n^2)$ time, not counting the time required to allocate $O(n)$ memory and the time needed to parse the input parameters, however, these are probably constant as we are only parsing a constant number of parameters. The parsing function does count the number of characters in the input string from Python, but this is likely buffered as the string cannot be null terminated since it is allowed to contain characters with a byte value of 0.

7.4.1 Benchmarking managed and native implementations

It is no surprise that our C implementation of IDEA is a lot faster than our Python implementation. However, just because Python is slower does not always mean that it will be noticeable. For instance implementing our IMAP synchronization process in C, would probably not make any noticeable difference since the synchronization time mostly is limited by the network latency and server capacity.

In figure 7.1 the benchmark results of the native implementation vs. the Python implementation can be seen, tested running both CBC and ECB mode. The benchmark consisted of encrypting a 1262 KiB pdf file, and was executed 10 times, of which the

[8] A pointer is a integer with the value of a memory address.

[9] An unsigned short is 16 bits long and cannot be negative.

average operation time was the result, on a 1.6 GHz Pentium M computer with 1280 MiB ram, running Ubuntu 7.10. On Figure 7.1 there are 4 pillars, two executed with ECB mode, one with native implementation and one with managed implementation, and two executed using CBC mode, again there is one for both native and managed implementations.

It can be difficult to see all the pillars in figure 7.1, as the average native implementation completes the encryption process 270 time faster than thee managed implementation. In average the encryption of the 1262 KiB file took 0.101 seconds using the native implementation running in CBC mode, the managed CBC implementation took an average of 27.186 seconds to perform the same operation. In ECB mode the native implementation took an average of 0.102 seconds, while the managed implementation took an average of 27.728 seconds to complete the same operation.

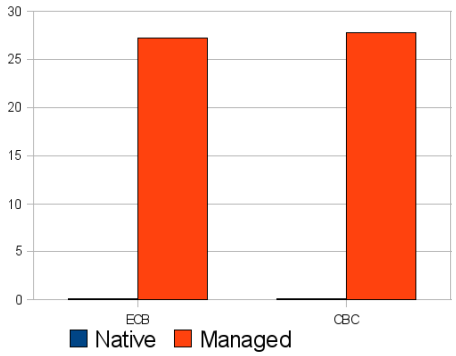


Figure 7.1: Benchmark of IDEA implementation in Python vs. C.

Based on these results it can easily be concluded that our native implementation is far more efficient than our managed implementation. However, we have to acknowledge that our Python implementation is not as good as it could be, even within the limitations of Python, so there is no basis for concluding that making an usable IDEA implementation in Python is impossible. Nevertheless, we can conclude that it is easier, for a knowledgeable C programmer to write an efficient IDEA implementation in C than in Python.

7.5 Raptor email message format

When an email is sent signed and/or in an encrypted form, it is sent as raw text through the email protocol. Therefore RaptorMail uses its own message suffix that can be recognised by RaptorMail but does not look messy if it is opened with another email client. The format contains the message, signature and public key of the sender. If the message is encrypted the signature is encrypted along with it. The public key consists of two numbers e and n and they have also been separated. RaptorMail uses the suffix $\backslash r \backslash n$ on each line to identify the different message and encryption parts.

Encrypted email suffix:

```
\r\nRaptor encrypted mail:\r\n    #control string
<Base-64 string>\r\n              #RSA encrypted IDEA key
<Base-64 string>                  #IDEA encrypted message
```

Signed email suffix: If the mail is both signed and encrypted the following suffix is inside the encrypted message

```
\r\nRaptor signature:\r\n          #control string
<Base-64 string>\r\n               #RSA signature
Raptor public key:\r\n             #control string
<Base-64 string>\r\n               #RSA public key e
<Base-64 string>                   #RSA public key n
```

This is an example of the raw text of a RaptorMail signed email, the keys have been reduced to simplify the example.

```
Hi Alice,

I just though you should know that you can encrypt
mails, using the public key attached to this mail.

Regards Bob.\r\n
Raptor signature:\r\n
4d83c4d5ccc5af.....573d2aeb0e8ccd\r\n
Raptor public key:\r\n
851b0f28c6dc93....3421b41421\r\n
5ba807f92a0c6....4a70d0fdc5
```

This is an example of the raw text of a RaptorMail encrypted email. The keys have been reduced again to simplify the example.

```
Hi,
This message has been encrypted using RaptorMail you need to decrypt it
using proper private key and RaptorMail.
Regards your guardian Raptor.\r\n
Raptor encrypted mail:\r\n
NjgxMTY5Mx.....NTA5NDc2MjE=\r\n
AwAOAA8AXA......4VSc3auTjU5B1XFGIw==
```

When RaptorMail receives an email it uses the control suffix to split the message. For full source code, see the preface. This is how RaptorMail identifies the suffix in a signed email. First the program removes the $\backslash r \backslash n$ suffix of the message. This is required because Gmail attaches this to the message. The message is then split into the suffix msg_parts.

```
#Remove added CRLF if needed
if message[-2:] == "\r\n":
    message = message[:-2]

msg_parts = message.split("\r\n")
```

In Python the n element at the back of an array can be retrieved with the following syntax

```
Element = Array[-n]
```

This can be used to get the signature and other suffixes attached to the end of the email when it is split into an array. The following code shows how RaptorMail retrieves the different parts of the public key and the signature. It does not matter if the public key or the signature comes first in the mail.

```
#Get signature or public key
if len(msg_parts) >= 2 and msg_parts[-2] == "Raptor signature:":
    signature = msg_parts[-1]
    metaparts += 2
elif len(msg_parts) >= 3 and msg_parts[-3] == "Raptor public key:":
    publickey += [msg_parts[-2]]
    publickey += [msg_parts[-1]]
    metaparts += 3
#Get signature or public key
if len(msg_parts) >= (metaparts+2) and msg_parts[-(metaparts+2)]
                                        == "Raptor signature:":
    signature = msg_parts[-(metaparts+1)]
    metaparts += 2
elif len(msg_parts) >= (metaparts+3) and msg_parts[-(metaparts+3)]
                                        == "Raptor public key:":
    publickey += [msg_parts[-(metaparts+2)]]
    publickey += [msg_parts[-(metaparts+1)]]
    metaparts += 3
```

Because of the chance that the user writes $\backslash r \backslash n$ in the message a precaution must be taken. If that happens the message text itself will be split into parts as well, therefore the parts are joined again in the end, and then joined with $\backslash r \backslash n$ thereby the message will appear as originally intended.

```
    if metaparts == 0: metaparts = None
    else: metaparts = -metaparts
    message = "\r\n".join(msg_parts[0:metaparts])
```

CHAPTER 8

Conclusion

Based on a limited questionnaire, performed amongst the students of Aalborg University, it can be concluded that the majority of this group is unaware that the email correspondence is insecure. 71 % of the questioned did not know that emails can be spoofed, and from this, it can be concluded that many users of email are unaware of these problematics.

To test whether or not existing email clients, which can encrypt, are usable to the average user of emails, a usability test of Thunderbird, with the add-on Enigmail, was performed. This test found some issues, which might prevent users from using the software. One of the major issues was the illogical distribution method of public keys. In addition because Enigmail is an add-on, it is, from a usability point of view, poorly implemented. From these data, a basis was laid for the creation of a more intuitive and userfriendly program, called RaptorMail. RaptorMail was designed with the intention of use RSA encryption with every email. As this was the purpose from the beginning, it was possible to integrate a more logical method of key distribution, encrypting and signing emails.

A usability test on RaptorMail, showed that the cryptographic features were easier to access and use in RaptorMail, than in Thunderbird with the Enigmail add-on. From the test it can be concluded that some of the usability issues regarding encryption have been solved in RaptorMail.

However, the mathematical aspects of RSA prevents us from solving some of the other usability issues. For one it is not possible to implement a functional way of knowing whether the key actually belongs to the expected person.

Based on the complexity analysis of the RSA problem, it can be concluded that RSA relies on the problem of integer factorization. A problem that is not known to have any solution in polynomial time, which means that using a large keysize, makes it impossible to crack RSA in any reasonable timeframe.

In closing it can be concluded that it is possible to make email cryptography more usable. However, some aspects, such as key distribution, cannot be done without involving the user, and will most likely always be unintuitive and hard to comprehend for the user.

APPENDIX A

Usability log file

The log file from the usability test of Thunderbird with Enigmail.

Time	Comment	Problem no.
2.35	Scenario	
3.11	**Assignment 1** is read aloud.	
3.26	Begins assignment 1. Goes to the inbox and looks at the first open mail.	P1
4.36	Wants to open a browser, to find the key. The test leader tells the test person that it is in the program the key must be found.	P2
5.17	Finds OpenPGP in the menu.	
5.35	Clicks "Key server" and does not know, what to search for.	
5.58	Searches for the name.	P3
6.41	Finds Fotocarstens key.	
6.47	Asks if it is enough, the key lies on the server.	P4
7.3	Minimises the window and goes to the inbox. Does not know where to look, to find out whether or not it is okay.	P5 and P6
7.5	Clicks at OpenPGP again. Copies the key.	
8.1	Subject thinks it is necessary to find the key even though it already downloaded.	
9.47	Wants to search for the key again.	
10.16	Sees the signature bar, after hints from the test leader. Does not know whether the signature is sufficient.	P7
10.36	**Assignment 2** is read aloud.	

11.02	Begins assignment 2.	
11.15	Writes the email.	
11.35	Clicks the OpenPGP-button in the toolbar and signs.	
11.44	Asks if it signed now.	
12.37	Message sent.	P8
12.38	Waits and discusses the usability of the program. Subject thinks "it is pretty obvious. It says what to do, if only you open your eyes".	
15.24	**Assignment 3** is read aloud.	
	Moves on to the next assignment, because the answer from assignment 2 has not been received.	
15.45	**Assignment 4** is read aloud.	
16.04	Begins assignment 4.	
16.35	Writes a new email.	
17.11	Signs and encrypt the email.	
18.13	Email sent. Receives answer from the bank.	
18.19	**Assignment 3**	
18.36	Clicks on OpenPGP in the menu, and searches in the already downloaded keys, instead of the server.	P9
18.59	Realises it is the wrong place. Goes to the key server.	
20.3	Key is downloaded. Reads the dialog box this time.	
21.09	Receives the mail from Fotocarsten (for assignment 4).	
21.14	Clicks on the padlock. Receives a dialogbox, what asks whether subject wants to import the public key from the bank. The test subject clicks "Yes".	P10 and P5
21.34	The mail is updated and the bar writes that the mail is verified.	
21.47	**Assignment 5** is read aloud.	**P11**

21.57	Begins assignment 5.	
22.16	Opens mail.	
22.33	Clicks on a button depicting an envelope with a question mark on it. The received dialogbox says that there is a key that fits.	
23.23	Clicks the Decrypt-button. Receives a strange error message, which we cannot reconstruct. Normally it is an error message that says that there is not any encrypted block in the mail.	P11
24.25	In OpenPGP from the menu line subject chooses "Save decrypted message". Does not think it will work.	
25.1	Has only saved the message.	
25.16	Saves attached message.	
25.22	Is in doubt whether subject can save, when it is not decrypted.	
25.29	Tries anyway.	
25.38	Observes that the picture is still encrypted.	
25.46	Clicks Decrypt again. Same error message as at 23.23.	
25.55	Goes to Key management.	
26.1	Searches for Fotocarstens key. "It seems like it cannot find it".	
26.26	Dialog button that says that the key is unchanged.	
26.33	Clicks the envelope with the question mark again.	
26.46	"I am really lost right now".	
27.08	The test lead stops the test.	
27.37	"It is obviously a good program, when you know it," the test subject says.	

APPENDIX B

Survey on security awareness

Question 1: Do you know what spoofing is? Spoofing: When the stated return email address is not the same as the sender's email address.	
Yes	22
No	55
Question 2: Do you know that Google saves and analyses the information on every email sent with Gmail, in order to improve their advertisements?	
Yes	31
No	46
Question 3: Do you know whether the email client you use is able to encrypt and sign emails or not?	
Yes	12
No	65
Question 4: If yes: Do you use it?	
Yes	1
No	13
Question 5: If the new version of Thunderbird, or the email client you use, or Hotmail or Gmail were updated to use transparent encryption and signing, would you use it, based on the fact that every email can be read and/or changed, and that the sender may not be the correct?	
Yes	74
No	3

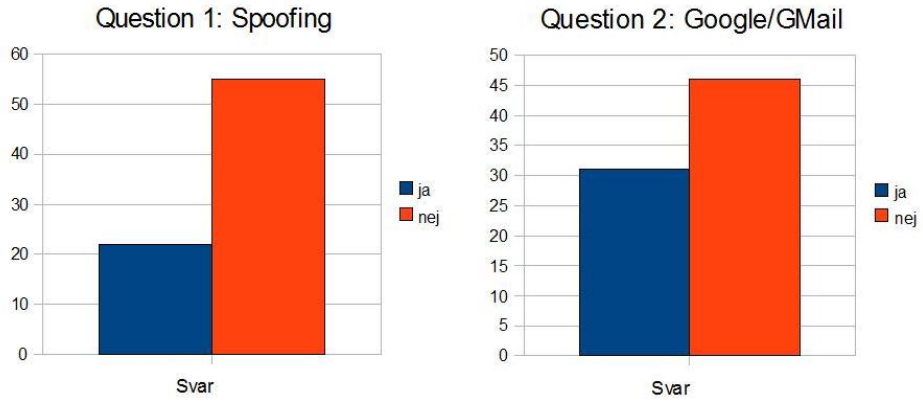


Figure B.1: Question 1 and 2

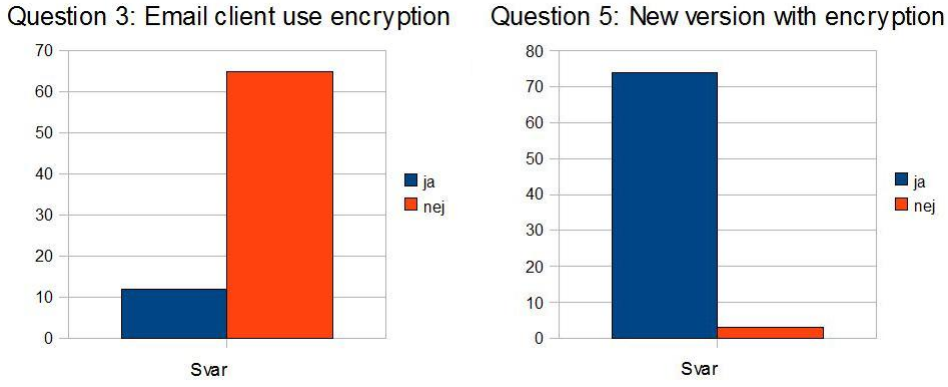


Figure B.2: Question 3 and 5

APPENDIX C

Binary numbers and bitwise operations

C.1 Binary numbers, bitwise operations and bit shift

This section will describe the basics of binary number some of the bitwise operations work.

C.1.1 Representation of binary numbers

Binary numbers are expressed by two distinct symbols, 1 and 0. Computers use binary numbers to operate - here the symbol 1 represents a high voltage and the symbol 0 represents no voltage.

The binary system functions almost like the decimal system, but instead of using the symbols 0-9 it only uses the symbols 0 and 1. For instance calculating from 0 to 10 in binary numbers will look like this:

Example 6 Numbers 0 to 10 in binary numbers:
0000, 0001, 0010, 0011, 0100, 0101
0110, 0111, 1000, 1001, 1010

Example 6 indicates how binary numbers are computed. Since the binary system is based on two digits, converting a binary number into a number of the decimal system can be done by representing every digit increasingly by the power of 2. From right to left every bit is represented by $2^0, 2^1, 2^3$, and so on. Thereby the binary number 10011 can be calculated into decimal numbers by:

Example 7

$$(1 \cdot 2^4) + (0 \cdot 2^3) + (0 \cdot 2^2) + (1 \cdot 2^1) + (1 \cdot 2^0) = 16 + 2 + 1 \quad \text{(C.1)}$$
$$= 19 \quad \text{(C.2)}$$

C.1.2 Bitwise operations

Bitwise operations compare two strings, and gives an output depending on the operation. Below five bitwise operations are explained.

NOT

The bitwise NOT converts a 1 to a 0 and a 0 to a 1. Therefore NOT 1001011 results in

NOT 1001011
0110100

AND

Bitwise AND compares two bit strings and retuns a 1 if both is 1, else it returns a 0. An example of a bitwise AND operation follows:

$$\begin{array}{c} 1001011 \\ \text{AND} \\ 1001110 \\ \hline 1001010 \end{array}$$

OR

The operation bitwise OR returns 1 if either or both of the bits are 1:

$$\begin{array}{c} 1001011 \\ \text{OR} \\ 1001110 \\ \hline 1001111 \end{array}$$

Exclusive or - XOR

The bitwise operation 'Exclusive OR' or XOR returns 1 if the corresponding places are different, and 0 if they are the same. This results in the following:

$$\begin{array}{c} 1001011 \\ \text{XOR} \\ 1001110 \\ \hline 0000101 \end{array}$$

Exclusive or can also be computed by the before mentioned bitwise operations. Since the result is true only if the bits are different, the following two equations are also true.

Definition 8

$$p \text{ XOR } q = (p \text{ AND NOT} q) \text{ OR } (\text{NOT} p \text{ AND } q) \tag{C.3}$$

$$p \oplus q = (p \wedge \neg q) \vee (\neg p \wedge q) \tag{C.4}$$

$$\tag{C.5}$$

Bit shifts

Bit shifting is often mistaken as a bitwise operation because of its name, but it is not. The bit shift operation can be denoted $\ll$ or $\gg$. The former is a bit shift left, the latter a bit shift right.

Bit shift left When making a bit shift left operation, a 0 is added at the far right, and the other bits are moved one bit to the left.

$$\underline{1001110 \ll}$$
$$10011100$$

The bit shift left operation can also be used to multiply a number by 2, as illustrated with the example below:

Example 8 Bit shift left operation

$$\begin{aligned} 1001110 &= 2^6 + 2^3 + 2^2 + 2^1 && \text{(C.6)} \\ &= 78 && \text{(C.7)} \\ 10011100 &= 2^7 + 2^4 + 2^3 + 2^2 && \text{(C.8)} \\ &= 156 && \text{(C.9)} \\ 78 \cdot 2 &= 156 && \text{(C.10)} \end{aligned}$$

Bit shift right The bit shift right operation is similar to the bit shift left operation, only instead of inserting a 0 at the right end, it inserts a 0 at the leftmost end of the bit string. Also the bit shift right operation can be used to divide a number by 2. When bit shifting right, the rightmost bit is removed. Here it is necessary to check whether the removed is 1 or 0. If 1 is removed the result is an odd number.

Example 9 Bit shift right operation

$$\begin{aligned} 1001110 &= 2^6 + 2^3 + 2^2 + 2^1 && \text{(C.11)} \\ &= 78 && \text{(C.12)} \\ 0100111 &= 2^5 + 2^2 + 2^1 + 2^0 && \text{(C.13)} \\ &= 39 && \text{(C.14)} \\ \frac{78}{2} &= 39 && \text{(C.15)} \end{aligned}$$

APPENDIX D

Symbols description

This section describes the notation form that we have used in this report. As the notation form differs from one book to another we have chosen the notation forms we like the most and listed them here.

D.1 The set of all natural numbers $\mathbb{N}$

$\mathbb{N}$ denotes the set of all natural numbers, e.g. $\mathbb{N} = (0), 1, 2, 3, \ldots$. That is all positive integers, sometimes including zero, depending on context. An example of numbers in $\mathbb{N}$ would be $1, 5, 102$ and 9974.

D.2 The set of all integers $\mathbb{Z}$

The set of all integers $\mathbb{Z}$ contains all integers, e.g. $\mathbb{Z} = \ldots, -1, 0, 1, \ldots$. That is $\mathbb{Z}$ contains all the elements of $\mathbb{N}$ and $-\mathbb{N}$ including 0. Examples would be $-594, -3, 0, 7$ and 948.

D.3 The set of integers under modulo n $\mathbb{Z}_n$

The set of integers under modulo n is $\mathbb{Z}_n = \{a \bmod n | a \in \mathbb{Z}\}$. That is $\mathbb{Z}$ contains the integers less than n and bigger than or equal to 0. Thus $\mathbb{Z}$ is a subset of $\mathbb{N}$. An example would be the set $\mathbb{Z}_6$ that contains the integers $0, 1, 2, 3, 4$ and 5.

D.4 Congruence relation $\equiv$

The symbol $\equiv$ is used to show congruence between to sides of an equation. If $a \equiv b \pmod{c}$ then $c|a - b$ which means that c divides $a - b$. It is also true that $a \bmod c = b \bmod c$.

D.5 Intersection $\cap$

This symbol denotes the intersection between to sets. Thereby $1, 2, 3, 4, 5 \cap 3, 4, 5, 6, 7 = 3, 4, 5$, and the result is the subset that is common to both sets.

D.6 The set of divisors $div(t)$

The function $div(t)$ returns all numbers that divides t. That is $div(n) = \{d \in \mathbb{N}; d \mid n\}$, and an example could be: $div(12) = 1, 2, 3, 4, 6$.

D.7 The binary modulo operation mod

The modulo used in section D.4 to describe congruence relation, denotes the remainder of an integer division. The expression $a = b \bmod c$ means that the remainder after dividing b with c is a. That is $15 \bmod 4 = 3$.

If $a_1 \equiv b_1 \pmod{n}$ and $a_2 \equiv b_2 \pmod{}$ the below expressions can be set up [Lauritzen, 2003].

$$a_1 + a_2 \equiv b_1 + b_2 \pmod{n} \tag{D.1}$$
$$a_1 \cdot a_2 \equiv b_1 \cdot b_2 \pmod{n} \tag{D.2}$$

D.8 The greatest common divisor $gcd(t, f)$

In the section about Euclid's algorithm these functions are explained. This section will only sum up the definition.

$$gcd(a, 0) = |a| \tag{D.3}$$
$$gcd(a, b) = gcd(a \bmod b, b), \quad a > b \tag{D.4}$$

By following these definitions, it is possible to determine the greatest common divisor of a and b. Calculating $gcd(13, 6)$ the following equations can be conducted: $gcd(13, 7) = gcd(7, 4) = gcd(4, 3) = 1$.

D.9 Euler's totient function $\varphi(t)$

One of the aspects in RSA is to generate $\varphi(t)$. This function is deduced from Euler's theorem and denotes the number of relative primes to t, and smaller that t. It is computed $\varphi(t) = (p - 1)(q - 1)$, where p and q are prime numbers.

Bibliography

Pygtk 2.0 reference manual. January 2008. URL `http://library.gnome.org/devel/pygtk/stable/`.

P. van Oorschot A. Menezes and S. Vanstone. *Handbook of Applied Cryptography*. CRC Press, 1996. ISBN 0-8493-8523-7.

Rob Austein. SYNCHRONIZATION OPERATIONS FOR DISCONNECTED IMAP4 CLIENTS, June 1994. URL `http://tools.ietf.org/id/draft-ietf-imap-disc-00.txt`.

Allan Clark. *Elements of Abstract Algebra*. Dover Publications Inc., 1985. ISBN 0-48664-725-0.

M. Crispin. INTERNET MESSAGE ACCESS PROTOCOL - VERSION 4rev1. RFC 3501 (Proposed Standard), March 2003. URL `http://www.ietf.org/rfc/rfc3501.txt`. Updated by RFCs 4466, 4469, 4551, 5032, 5182.

H. Gilbert. Introduction to tcp/ip. February 1995. URL `http://www.yale.edu/pclt/COMM/TCPIP.HTM`.

Google. Politik til beskyttelse af personlige oplysninger. August 2008. URL `http://www.google.dk/intl/da/privacypolicy.html`.

Teknologi og Udvikling IT-og Telestyrlesen Ministeriet for Videnskab. Privatlivets fred. December 2008. URL `http://www.itst.dk/it-sikkerhed/it-sikkerhedskomiteen/indsatsomrader/privatlivets-fred`.

Rajeev Motwani John E. Hopcroft and Jeffery D. Ullman. *Introduction to Automata Theory, Languages, and Computation*, chapter 8.2. Addison-Wesley, 2 edition, 2001.

Justitsministeriet. Lov om behandling af personoplysninger. May 2000. URL `https://www.retsinformation.dk/Forms/R0710.aspx?id=828`.

J. Klensin. Simple Mail Transfer Protocol. RFC 5321 (Draft Standard), October 2008. URL `http://www.ietf.org/rfc/rfc5321.txt`.

Peter Landrock and Knud Nissen. *Kryptologi - fra viden til videnskab*. Jelling Bogtrykkeri, 1 edition, 1997. ISBN 87-89182-62-6.

Niels Lauritzen. *Concrete Abstract Algebra*. Cambridge University Press, 2003. ISBN 0-521-53410-0.

Bruce Schneier. *Applied Cryptography*. John Wiley and Sons, Inc., 2 edition, January 1996. ISBN 0471128457.

Helen Sharp, Yvonne Rogers, and Jenny Preese. *Interaction Design - beyond human-computer interaction*. John Wiley & Sons, Ltd, 2 edition, December 2007. ISBN 978-0-470-01866-8.

Micheal Sipser. *Introduction to the Theory of Computation*. Course Technology, 2 edition, 2006. ISBN 0-534-95097-3.

Surachit. International data encryption algorithm, 2008. URL `http://en.wikipedia.org/wiki/International\_Data\_Encryption\_Algorithm`. Wikipedia article on IDEA symmetric-key algorithm.

Florian Weimer. New openssl packages fix predictable random number generator. May 2008. URL `http://lists.debian.org/debian-security-announce/2008/msg00152.html`.

Michael Osterman Network World. You need to encrypt your e-mail. September 2008. URL `http://www.networkworld.com/newsletters/gwm/2008/100608msg2.html`.