MODERN CRYPTOGRAPHY: THE RSA SYSTEM

Delft, 2021

Floor Straver (4581830)

Technische Universiteit Delft

1 Preface

Before you lies the thesis 'Modern cryptography: the RSA sytem'. This thesis is written as part of my graduation for the bachelor Applied Mathematics on the TU Delft.

During my time on The Hague University of Applied Sciences someone gave a guest lecture about his job at Ministry of Defence. His job was to crack encoded phone conversations in war zones. This is when I released I was really interested in cryptography. It was very nice that I could combine this interest with my bachelor thesis.

I would like to thank my supervisors Nikolaas D. Verhulst and Jos Weber for the support with this thesis. I also would like to thank the other member of the committee Tom Vroegrijk for reviewing this thesis.

Floor Straver
Delft, August 2021

Contents

2 Introduction

In 1977 the RSA[1] system was designed by Ron Rivest, Adi Shamir and Len Adleman [12]. The RSA system is a part of modern cryptography. Modern cryptography makes sure that one can secure important information - like data, communication or banking accounts - on the computer.

The main idea of the RSA system is that of a one-way function, a function that is easy to compute in one way, but very difficult to compute the other way. This concept resembles the idea of a mailbox. If for example Alice wants to send a message to Bob, she can send a letter to his address. But once it is in the mailbox it is hard to get it out of there.

The encryption and decryption of information with RSA happens with a public and a private key. In the concept of the example of the mailbox the address would be the public key and the physical key of the mailbox would be the private key. Everyone who wants to send a letter to Bob can do so if they have the address, but only Bob has the key of the mailbox and can read the letters.

Generating the keys for the RSA system is left to the receiver. The keys are linked to each other, such that if the sender has encrypted the message with the public key the receiver can decrypt it with his private key.

2.1 Structure of this thesis

This section will describe the structure of this thesis by chapter. In chapter 3 we will get into some history of cryptography to get an idea of how encryption and decryption of messages began. Chapter 4 describes some preliminary knowledge before getting into the main concept of RSA systems. Here we will discuss the Euler totient function, Fermat's little theorem and continued fractions. This chapter also gives a short explanation about time complexity of algorithms. In chapter 5 we will get into the RSA system. We will discuss how to set up the RSA system and how to encrypt and decrypt messages with this RSA system. We also will give a proof of correctness of the RSA system, which means that if an encoded message is decoded the original message will be obtained. In chapter 6 the concept of signatures with an RSA system is described, such that malicious users cannot forge a message. In chapter 7 we will give two examples of attacks on the RSA system: the Wiener attack and Fermat factorization. The final chapter 8 will give the conclusion and discussion of this thesis.

[1] abbreviation for Rivest, Shamir & Adleman

3 History

As mentioned, RSA is part of cryptography. Cryptography is the study of securing communication through codes such that unintended recipients can not read it. Most people may think about computers when talking about cryptography, but cryptography has a long history before this.

For example in 400 before Christ in Greece scytales were used, especially by the Spartans in military campaigns, see figure 3.1. The scytale is a tool consisting of a cylinder and a strip of leather or parchment. They wound the piece of leather or parchment around a cylinder and wrote a message on it. The person who received the leather or parchment could wrap it around a similar size cylinder and read the message. Through the thickness and shape of the cylinder people could encrypt a message. Only the people with the same sort of cylinder could decrypt the message. Further information about this can be found in [3].

Figure 3.1: Scytale *[3]*.

Another quite famous sort of cryptography is the Caesar-cipher. This code was named after Julius Caesar, who used it for his secret communication. Julius Caesar lived around 60 years before Christ. In the Caesar-cipher code the letters of the alphabet are shifted three places in the alphabet. So for example the letter “a” became the letter “d”, see figure 3.2. It is clear that it is very easy to decipher this Caesar-cipher.

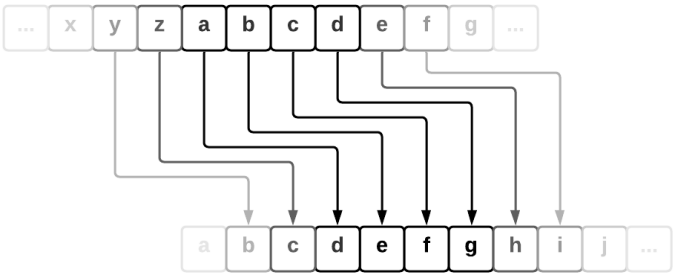

Figure 3.2: Caesar-Cipher code

Also later on in the Second World War cryptography was really important. Encryption was used to encode messages in the military [1]. The most famous kind of encryption was the use of Enigma machines by the German military. The most important messages were sent with the use of this machine. English mathematician Alan Turing cracked the code of the Enigma machine during World War II. The whole operation and cracking of the Enigma machine had to be kept secret, this is why there was not much recognition for Alan Turing back then. In 1951 however Alan was elected to the Fellowship of the Royal Society. This society of scientists is where you get elected if you did a substantial contribution to the improvement of natural knowledge, including mathematics, engineering science and medical science. The Fellowship of the Royal Society is mostly for English scientists. Later in 2014 there was made a famous movie about his live and work.

The importance of encoding messages increased with the arrival of the internet. E-mail and internet banking are examples of applications that use encoding. This is very important, because there are a lot of malicious users on the internet.

There are two types of cryptography: symmetric or asymmetric encryption. Symmetric encryption is encryption and decryption of a message using the same key. This is not convenient, the key needs to be sent to the receiver over a secure connection, for example by a driver or in person. The problem with this way of sending the key is how to make this connection secure. Until 1970 only the symmetric encryption was known. Asymmetric encryption works with public and private keys, such that no key has to be delivered from the sender to the receiver. The asymmetric RSA system gave a solution for this.

Rivest, Shamir and Adleman weren't the first to come up with the idea of the RSA system. In 1997 a document was declassified showing that Clifford Cocks already described the idea of the RSA system in 1973.

4 Preliminaries

4.1 The Euler totient function

The Euler totient function is an important function that is used in the RSA algorithm. Based on [4] we get the following definition and theorems for the Euler totient function:

Definition 4.1 (The Euler totient function)**.** The Euler totient function of an integer n counts the number of integers between 0 and n that are relatively prime with n. It is denoted as $\varphi(n)$.

For a prime number p the Euler totient function is equal to:

$$\varphi(p) = p - 1. \tag{1}$$

This is clear from the fact that all numbers except for 1 and the prime number p itself are relatively prime to the prime number p. For the calculation of the Euler totient function in setting up the RSA algorithm we also need the following theorems about the Euler totient function:

Theorem 4.1. Let m, n integers that are relatively prime, then

$$\varphi(mn) = \varphi(m)\varphi(n).$$

Theorem 4.2. Let n be a positive integer with prime factorization: $n = p_1^{k_1} p_2^{k_2} \cdots p_r^{k_r}$, with p_i prime number for $i \in 1, \ldots, r$ and k_i real number for $i \in 1, \ldots, r$. Then

$$\varphi(n) = (p_1^{k_1} - p_1^{k_1 - 1})(p_2^{k_2} - p_2^{k_2 - 1}) \cdots (p_r^{k_r} - p_r^{k_r - 1}).$$

4.2 Fermat's little theorem

For the proof that the RSA algorithm encoding is correct we need the following theorems from [13].

Theorem 4.3 (Fermat's little theorem)**.** If p prime, then for any integer a we have

$$a^p \equiv a \pmod{p}.$$

and if a is not divisible by p, then

$$a^{p-1} \equiv 1 \pmod{p}.$$

4.3 Euclidean algorithm

For the calculation of the public key in the set up of the RSA algorithm we need the extended Euclidean algorithm. Before we go to the extended Euclidean algorithm, we first explain the Euclidean algorithm. The Euclidean algorithm finds the gcd[2] of two integers. From [10] we have the following theorem:

Theorem 4.4 (Euclidean algorithm)**.** Define for integers a and $b \neq 0$ a sequence of non-negative integers $r_0, r_1, \ldots$ by $r_0 = |a|, r_1 = |b|$ and

$$r_{i+1} = (\text{rest of } r_{i-1} \text{ by division by } r_i) \text{ if } r_i \neq 0. \tag{2}$$

Then there exists an index $k > 0$ with $r_k = 0$ and with $\gcd(a, b) = r_{k-1}$.

Now the extended Euclidean algorithm finds for known a and b the solution of

$$ax + by = \gcd(a, b). \tag{3}$$

The algorithm goes as follows:
We start with the following two equations:

$$x_0 \cdot a + y_0 \cdot b = r_0 = |a| \qquad \text{and} \qquad x_1 \cdot a + y_1 \cdot b = r_1 = |b|. \tag{4}$$

[2] abbreviation for greatest common divisor

Where $x_0 = \pm 1, y_0 = 0, x_1 = 0$ and $y_1 = \pm 1$. The signs of x_0 and y_1 depends on the numbers a and b, choose the sign such that the equations in (4) hold. Dividing with remainder in theorem 2 gives that there exists an integer q_i such that

$$r_{i-1} = q_i \cdot r_i + r_{i+1}$$

With this we have for every $i = 2, 3, \ldots$ the following formulas

$$x_{i-1} = q_i \cdot x_i + x_{i+1},$$
$$y_{i-1} = y_i \cdot r_i + y_{i+1}.$$

We now find for $i = 2, 3, \ldots$ the following formula

$$ax_i + by_i = r_i.$$

When r_k is equal to 0 for some $k = 2, \ldots$ then the solution is then equal to:

$$ax_{k-1} + by_{k-1} = r_{k-1} = \gcd(a, b).$$

Example 4.1. Let $a = 65789$ and $b = 23456$, then we have:

i	Equation	x_i	y_i	q_i	r_i
0	$1 \cdot 65789 + 0 \cdot 23456$	1	0		65789
1	$0 \cdot 65789 + 1 \cdot 23456$	0	1	2	23456
2	$1 \cdot 65789 - 2 \cdot 23456$	1	-2	1	18877
3	$-1 \cdot 65789 + 3 \cdot 23456$	-1	3	3	4579
4	$5 \cdot 65789 - 14 \cdot 23456$	5	-14	8	561
5	$-41 \cdot 65789 + 115 \cdot 23456$	-4	115	6	91
6	$251 \cdot 65789 - 704 \cdot 23456$	251	-704	6	15
7	$-1547 \cdot 65789 + 4339 \cdot 23456$	-1547	4339	15	1

Now we have our solution, because $r_8 = 0$. We found that the $\gcd(a, b)$ is equal to 1. This means that in this case a and b are relatively prime.

4.4 Continued fraction

Later on it will be useful to know a bit about continued fractions. Based on [9] we get the following theory about continued fractions.

Definition 4.2. (*Continued fraction of the rational numbers* $\mathbb{Q}$). A finite continued fraction expansion is a fraction of the form

$$a_0 + \cfrac{1}{a_1 + \cfrac{1}{a_2 + \cfrac{1}{a_3 + \ldots + \cfrac{1}{a_m}}}}.$$

Where a_i are non-negative integers, for $i = 0, 1, \ldots, m$ and a_0 can be any integer. The notation of this continued fraction is $[a_0, a_1, a_2, a_3, \ldots, a_m]$.

Remark 4.1. Every $x \in \mathbb{Q}$ has a finite continued fraction expansion.

Remark 4.2. In this report we will only use continued fractions of rational numbers in $[0, 1)$. For this reason we will only use continued fractions of the form where $a_0 = 0$. Further in this report we will write continued fractions as $[a_1, a_2, a_3, \ldots, a_m]$, where we omit the a_0.

Any rational number can be written as a continued fraction. The continued fraction of a rational number $x \in [0, 1)$ can be found in the following way, where we first initialize $r_0 = x$, $a_0 = 0$ and next[3]:

$$a_i = \left\lfloor \frac{1}{r_{i-1}} \right\rfloor, \qquad r_i = \frac{1}{r_{i-1}} - a_i, \qquad \text{for } i = 1, 2, \ldots n.$$

When r_i is an integer for some $i \in \{0, 1, 2, \ldots\}$, the continued fraction is found and $x = [a_1, a_2, \ldots, a_i]$ that is constructed.

[3]With $\lfloor c \rfloor$ we mean c rounded down to a whole number.

Example 4.2. (*Finding the continued fraction of a rational number*) Let $x = \frac{29}{39}$. We calculate the continued fraction of x.

$$
\begin{aligned}
a_0 &= 0, & r_0 &= \frac{29}{39},\\
a_1 &= \left\lfloor \frac{1}{\frac{29}{39}} \right\rfloor = \left\lfloor \frac{39}{29} \right\rfloor = 1, & r_1 &= \frac{1}{\frac{29}{39}} - 1 = \frac{39}{29} - 1 = \frac{10}{29},\\
a_2 &= \left\lfloor \frac{1}{\frac{10}{29}} \right\rfloor = \left\lfloor \frac{29}{10} \right\rfloor = 2, & r_2 &= \frac{1}{\frac{10}{29}} - 2 = \frac{29}{10} - 2 = \frac{9}{10},\\
a_3 &= \left\lfloor \frac{1}{\frac{9}{10}} \right\rfloor = \left\lfloor \frac{10}{9} \right\rfloor = 1, & r_3 &= \frac{1}{\frac{9}{10}} - 1 = \frac{10}{9} - 1 = \frac{1}{9},\\
a_4 &= \left\lfloor \frac{1}{\frac{1}{9}} \right\rfloor = \left\lfloor \frac{9}{1} \right\rfloor = 9, & r_4 &= \frac{1}{\frac{1}{9}} - 1 = \frac{9}{1} - 1 = 8.
\end{aligned}
$$

r_4 is equal to an integer, so now we found a continued fraction for our rational number $\frac{29}{39}$. $\frac{29}{39}$ is equal to $[1, 2, 1, 9]$:

$$x = \frac{1}{1 + \frac{1}{2 + \frac{1}{1 + \frac{1}{9}}}}.$$

Definition 4.3. (*Convergent*) Let $x = [a_1, a_2, \ldots, a_m]$ be continued fraction. The convergent $\frac{p_i}{q_i}$ of x is defined by

$$\frac{p_i}{q_i} = [a_1, a_2, \ldots a_i], \qquad \text{for some } i \in 0, 1, \ldots, m.$$

There is also an algorithm to convert a continued fraction back to a rational number. Assume we have the continued fraction $x = [a_1, \ldots, a_m]$. Then we can calculate all of the convergents $\frac{p_i}{q_i} = [a_1, \ldots, a_i]$ of x for $i = 1, \ldots, m$ in the following way:

$$p_1 = 1, \qquad q_1 = a_1, \tag{5}$$

$$p_2 = a_2, \qquad q_2 = a_2 a_1 + 1, \tag{6}$$

$$p_i = a_i \cdot p_{i-1} + p_{i-2}, \qquad q_i = a_i \cdot q_{i-1} + q_{i-2}, \qquad \text{for } i = 2, \ldots, m. \tag{7}$$

Now we have that our x, which was equal to the continued fraction $[a_1, \ldots, a_m]$, is equal to the rational number $\frac{p_m}{q_m}$.

Remark 4.3. Every continued fraction represents a rational number. Every rational number can be presented as two finite continued fractions. These two representations differ only in the last term of the continued fraction. Let x be a rational number with continued fraction $[a_1, \ldots, a_{n-1}, a_m + 1]$, then x is also represented by the continued fraction $[a_1, \ldots, a_{m-1}, a_m, 1]$. The second representation is one element longer.

Theorem 4.5. (*Legendre's theorem*) Let $x \in \mathbb{Q}$; $p, q \in \mathbb{Z}$, $q > 0$ and $\gcd(p, q) = 1$, then

$$\left| x - \frac{p}{q} \right| < \frac{1}{2q^2}.$$

implies that $\frac{p}{q}$ is equal to a convergent of x.

For the proof of this theorem we refer to [7]

4.5 Time complexity of algorithms

The security of an RSA system depends on time complexity of algorithms. The time complexity of an algorithm with input n is equal to a function $T(n)$. The time complexity represents the number of units of time that an algorithm takes for an input of size n. Because of the fact that the unit of time is different for every computer - one computer is faster than the other - the time complexity is determined by the maximal number of operations of an algorithm.

Example 4.3. Let's take a look at the time complexity of the following algorithm:

Input: Natural number n and array A of real numbers
Output: The largest element of A up to and including the n^{th} element

```
1 large ← 0;
2 for i ← 0; i ≤ n; i++ do
3 |   if large < A[i] then
4 |   |   large ← A[i]
  |   end
  end
5 return large;
```

Algorithm 1: Calculation of the largest element of an array

This is an algorithm to get the largest element of an array up to and including the n^{th} element. To determine the time complexity i.e. the number of operations of the algorithm, we go through it line by line. In the first line 1 we initialize the variable *large*. This is one operation. In the second line 2 we first initialize variable i at 0, next check if $i < n$ and finally add 1 to i. The initializing of i happens only the first time going into the for-loop, however the other two operations happens every time before going in the for loop. The check if $i < n$ also happens once when the statement is not satisfied anymore and don't go in the for loop, this is also 1 operation. Now watching what happens inside the for-loop at lines 3 and 4 we see that we always check the statement in 3, but do not always go through line 4. So in line 3 and 4 we have to do 1 or 2 operations, because the time complexity is the worst case scenario we assume we always have to do the operation in line 4. So line 3 and 4 give 2 operations. We are going $n+1$ times through the for-loop, so from line 1 up to and including 4 we have $4(n+1)+2$ operations. At the end of line 5 we only return a variable which is only one operation. In total we have $T(n) = 4(n+1)+3$ operations.

4.5.1 Big $\mathcal{O}$ notation

A general notation to classify the time complexity of functions is the big $\mathcal{O}$ notation. The general definition of the big $\mathcal{O}$ notation from [5] is as follows

Definition 4.4. (Big $\mathcal{O}$ notation) Let $f(n)$ and $g(n)$ be two real valued functions defined on some subset of the real numbers. One writes

$$f(n) = \mathcal{O}(g(n)) \qquad \text{as } n \to \infty$$

if and only if, for sufficiently large values of n, $f(n)$ is at most a constant multiplied by $g(n)$ in absolute value.
That is, $f(n) = \mathcal{O}(g(n))$ if and only if there exists a positive real number M and a real number n_0 such that:

$$|f(n)| \leq M|g(n)| \qquad \text{for all } n > n_0.$$

Example 4.4. Assume we have an algorithm with time complexity $T(n) = 3(n^2+n+1)$. Then we have $\mathcal{O}(T) = \mathcal{O}(n^2)$. To show this take $n_0 = 1$ and $M = 9$. For $n \geq 1$ we have $3n \leq 3n^2$ and $3 \leq 3n^2$. So we have:

$$T(n) = 3(n^2+n+1) \leq 3n^2 + 3n^2 + 3n^2 = 9n^2.$$

Now the definition of the big $\mathcal{O}$ notation implies that the time complexity of this algorithm is $\mathcal{O}(n^2)$

There are different classes of functions in terms of their corresponding time complexity. The following are the most common classes with their corresponding names, where c is constant:

time complexity	Class
$\mathcal{O}(1)$	Constant
$\mathcal{O}(\log(n))$	Logarithmic
$\mathcal{O}(n)$	Linear
$\mathcal{O}(n\log(n))$	Linearithmic
$\mathcal{O}(n^2)$	Quadratic
$\mathcal{O}(n^c)$	Polynomial
$\mathcal{O}(c^n)$	Exponential

The classes of these running times go from fastest to slowest. Computers can take years to solve a problem that is for example exponential.

Remark 4.4. For the time complexity in the logarithmic class the base of the logarithm doesn't matter, because change of base can be done in constant time. This can be shown by proving that ${}^a\log x \in \mathcal{O}({}^b\log x)$. For this we need the following identity for some $a, b, x \in \mathbb{R}$:

$$ {}^a\log x = \frac{{}^b\log x}{{}^b\log a}. \tag{8}$$

By definition we need to prove that there exists a $M \in \mathbb{R}$ and a $n_0 \in \mathbb{R}$ such that ${}^a\log x \leq M\ {}^b\log x$ for all $n > n_0$. Now because b is a constant we have with (8) that:

$$ {}^a\log x \leq \frac{1}{{}^b\log a}\ {}^b\log x \qquad\qquad \text{for all } x \geq 1.$$

This implies that ${}^a\log x \in \mathcal{O}({}^b\log x)$ by definition with $M = \frac{1}{{}^b\log a}$ and $n_0 = 1$.

4.5.2 Time complexity of finding convergents

To look at the time complexity of the algorithm to calculate the convergents of a continued fraction $[a_1, \ldots, a_m]$ we first need the following lemmas:

Lemma 4.1. The nominator and denominator of the convergents of a rational number grow exponentially.

Proof. In the algorithm to find the convergents described in section 4.4 we see in equation (7) that the denominator of a convergent $\frac{p_i}{q_i}$ for $i = 2, \ldots, m$ is of the form

$$ p_i = a_i p_{i-1} + p_{i-2}.$$

Let's try $p_i = \lambda^i$. Then we get

$$ \lambda^i = a_i\lambda^{i-1} + \lambda^{i-2} \equiv \lambda^2 - a_i\lambda + 1 = 0.$$

This is a quadratic formula with solutions: $\lambda_1 = \frac{-a_i + a_i^2 + 4}{2}$ and $\lambda_2 = \frac{-a_i - a_i^2 + 4}{2}$ Now we get something of the form

$$ p_m = c_1(\lambda_1)^m + c_2(\lambda_2)^m.$$

With c_1 and c_2 constants. We see that the nominator p_i is a function of exponential growth. For the denominator we have the same equation and in the same way find exponential growth. □

Lemma 4.2. The number of convergents of a continued fraction $\frac{p}{q} = [a_1, \ldots, a_m]$ is of order $\mathcal{O}(\log q)$.

Proof. Because of the fact that the nominator and denominator grow exponentially - as we saw in lemma 4.1 - the number of convergents will be of logarithmic order. In the continued fraction $\frac{p}{q}$ we assumed that a_0 is 0 in definition 4.2, this is why we have that $p \leq q$. This implies that the number of convergents of $\frac{p}{q}$ is of the order of $\mathcal{O}(\log q)$. □

Lemma 4.3. The time complexity of calculating convergents of a continued fraction $[a_1, \ldots, a_m]$ with (5), (6) and (7) is of order $\mathcal{O}(m)$.

Proof. An efficient way to calculate the convergents of a continued fraction of at least length 2, is with the following algorithm, where array A contains the coëfficients of the continued fraction:

Input: Array A and some integer k between 1 and m
Output: The k^{th} convergent
1 $P[1] \leftarrow 1$;
2 $Q[1] \leftarrow A[1]$;
3 $P[2] \leftarrow A[2]$;
4 $Q[2] \leftarrow A[2] \cdot A[1] + 1$;
5 **for** $i \leftarrow 3$; $i \leq k$; $i++$ **do**
6 | $P[i] \leftarrow A[i][i-1] + P[i-2]$;
7 | $Q[i] \leftarrow A[i] \cdot Q[i-1] + Q[i-2]$;
end
8 $convergent = \frac{P[k]}{Q[k]}$;
9 **return** *convergent*;

Algorithm 2: Calculation convergents of a continued fraction

The first lines 1 up to and including 4 are 4 initialize assignments, this are 4 operations. Initializing i in the for loop in line 5 only costs 1 operation, however the other two operations happens every time before going in the for loop. The check if $i \leq k$ also happens once when the statement is not satisfied anymore and don't go in the for loop, this is also 1 operation. Line 6 and 7 are both also 1 operation. We are going $k-3$ times through the for-loop, so from line 5 up to and including 7 give $4(k-3)+2$ operations. The initialization of *convergent* in line 8 also fives one operation and so does the return statement in line 9. In total we then have $T(k) = 4(k-3)+8$ operations. This means that our algorithm is of order $\mathcal{O}(k)$. Now because of the fact that $k \leq m$ we have that the calculation of the convergent of a continued fraction is of order $\mathcal{O}(m)$. □

Theorem 4.6. The time complexity of finding all convergents of a continued fraction $\frac{p}{q} = [a_1, \ldots, a_m]$ is of order $\mathcal{O}(q \log q)$

Proof. In lemma 4.2 we saw that the number of convergents of a continued fraction is of order $\mathcal{O}(\log q)$. According to lemma 4.3 the calculation of the convergent takes order $\mathcal{O}(m)$. Now because of the fact that $m \leq q$, we have that finding all convergents of a continued fraction is of order $\mathcal{O}(q \log q)$. □

5 The RSA algorithm

5.1 Key generation

To set up a secure connection with the RSA algorithm the first thing to do is to generate the public and private keys. The public key consists of two integers (n, e) and the private key of one (d). The generation of the keys goes with the following algorithm:

1. Generate two prime numbers p and q.
2. Calculate the so-called modulus n by
$$n = p \cdot q.$$
3. Calculate $\varphi(n)$. From (4.2) it is clear that:
$$\varphi(n) = (p-1)(q-1). \tag{9}$$
4. Choose the private key d such that:
$$1 < d \leq \varphi(n) \qquad \text{and} \qquad \gcd(d, \varphi(n)) = 1$$
where gcd stands for the greatest common divisor.
5. The public key e is a multiplicative inverse of d with respect to $\varphi(n)$. So e is chosen such that
$$e \cdot d \equiv 1 \pmod{\varphi(n)}. \tag{10}$$

The public key can be calculated using the Euclidean algorithm, see (4.4).

5.2 Encryption and decryption

To send a message one first has to represent the message as a number between 0 and n. This representation of this message is not part of the encoding and is made public. The ASCII[4] or binary code are examples of methods to represent a message in a number.

Encryption happens with the encryption function E, which is defined as

$$E(M) \equiv M^e \pmod{n}, \tag{11}$$

where M is the message to be send and e and n form the public key (n, e).

The decryption also happens with a function. Let $C = E(M)$ be the encrypted message, then the decryption function D is defined as:

$$D(C) \equiv C^d \pmod{n}. \tag{12}$$

where d is the private key.

Example 5.1. To understand the algorithm it is good to consider an example. First we generate the keys:

1. Choose two prime numbers, for example:
$$p = 3 \text{ and } q = 11.$$
2. Compute $n = p \cdot q$:
$$n = 3 \cdot 11 = 33.$$
3. Compute $\varphi(n)$.
$$\varphi(33) = \varphi(3 \cdot 11) = (3-1) \cdot (11-1) = 2 \cdot 10 = 20.$$

[4] ASCII stands for American Standard Code for Information Interchange, see http://www.asciitable.com/

4. Choose the private key d such that $\gcd(d,(p-1)(q-1)) = \gcd(d,20) = 1$. Choose for example

$$d = 7.$$

In this case it would also have been correct to choose $d = 3, 9, 11, 13, 17$ or 19. $d = 1$ is left out of these options, because with this key you would just send the original message.

5. Finally the public key e must be such that $e \cdot d \equiv 1 \pmod{\varphi(n)}$. In this case e such that $e \cdot 7 \equiv 1 \pmod{20}$. Then:

$$e \equiv d^{-1} \pmod{\varphi(n)} \equiv 3 \pmod{20}.$$

The public and private keys are now created. The public key is equal to $(33, 3)$ and the private key to (7). Suppose we want to send the letter 'i', then we first have to describe this message in an integer between 0 and n. Say for example that we convert 'i' to the integer 9. Now the message can be encrypted as follows:

$$C = 9^3 \pmod{33} \equiv 729 \pmod{33} \equiv 3.$$

Decoding this again gives:

$$M = 3^7 \pmod{33} \equiv 2187 \pmod{33} \equiv 9.$$

5.3 Correctness

A correct RSA system has to guarantee that every encoded message can be decrypted such that the right message appears again. In other words the RSA system is correct when:

$$D(E(M)) = M.$$

The correctness of the RSA algorithm can be shown using Fermat's little theorem.

Theorem 5.1. The algorithm of RSA is correct, i.e. with E and D as in (11) and (12) and M between 1 and n, if we have:

$$D(E(M)) = M.$$

Proof. Euler's totient function is a multiplicative function, see (4.1), together with (1) we find:

$$\begin{aligned} \varphi(n) &= \varphi(p) \cdot \varphi(q), \\ &= (p-1) \cdot (q-1). \end{aligned}$$

In the RSA algorithm the e and d were chosen such that they were each other's multiplicative inverse with respect to $\varphi(n)$:

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

This implies that $e \cdot d$ can be written as $k \cdot \varphi(n) + 1$, with k some integer. Decoding an encoded message gives the following:

$$D(E(M)) \equiv (E(M))^d \equiv (M^e)^d \pmod{n} = M^{e \cdot d} \pmod{n}.$$

So we have to proof that $M^{e \cdot d} \pmod{n} = M$. So the encrypted and decrypted message can be written as:

$$M^{e \cdot d} \pmod{n} \equiv M^{k \cdot \varphi(n)+1} \pmod{n}.$$

Fermat's little theorem 4.2 imply the following for all M that are not divisible by p:

$$M^{p-1} \equiv 1 \pmod{p}.$$

As $\varphi(n) = (p-1)(q-1)$, φ is divisible by $p-1$. This implies

$$M^{k\varphi(n)+1} \equiv M \pmod{p}. \tag{13}$$

If M does divide p the following holds: $M \equiv 0 \pmod{p}$. This implies that $M^{k\varphi(n)+1} \equiv M \pmod{p}$, so the equation (13) holds for all M. The same holds for the other prime number q:

$$M^{k\varphi(n)+1} \equiv M \pmod{q}. \tag{14}$$

Consequently, with equations (13) and (14) it follows that:

$$M^{e \cdot d} \equiv M^{k\varphi(n)+1} \equiv M \pmod{n}. \tag{15}$$

Now we have what we need to prove the correctness. This yields

$$D(E(M)) = M.$$

This proves the correctness of the RSA algorithm for all M, where $0 < M < n$. □

5.4 Security

The security of the RSA algorithm is very important. If one sends a message to a certain person, no one else should be able to read it.

For security it is very important that the parameters in the RSA algorithm are chosen in a certain way. If some parameters are very small it is easy to crack this encryption. On the other hand choosing parameters very large slows down the calculations in the RSA algorithm and then sending a message takes a long time. So you want the parameters to be big enough to prevent people from cracking it, but small enough to let the communication go fast.

The reason why the RSA system is hard to crack is that it depends on a one-way function. A one-way function is a function that is easy to compute in one way, but very hard in the other way.

The one-way function in the RSA system is the generation of the modulus n. n is equal to the product of two primes. A multiplication is done in linear time and is relatively easy. But the other way around, so getting the two primes from the modulus n depends on factorizing n. Factorizing a number can be very hard if the number is big enough. With hard we mean that there is no good algorithm such that the prime factor can be calculated quickly when the modulus n is growing.

One of the fastest algorithm to find the prime factors is the algorithm using number field sieve [11]. This algorithm has a time complexity of $\mathcal{O}(e^{c \cdot (\log n)^{\frac{1}{3}} (\log \log n)^{\frac{2}{3}}})$. If n is very large this will blow up.

6 Signature

A beautiful property of the RSA system is that messages can be sent with a signature. Even the recipient of the message can't forge the signature of the sender.

This concept of signature is based on the fact that encryption and decryption of the message gives the same result as first decryption of the message and then encryption:

$$E(D(M)) \equiv (D(M))^e \equiv (M^d)^e \pmod{n} = M^{e \cdot d} \pmod{n}.$$

Assume Alice wants to send a message to Bob just like before. Alice has the public key (n_A, e_A) and private key (d_A) and Bob has the keys (n_B, e_B) and (d_B). Alice wants to send a signed message S. First she applies her decryption function:

$$S = M^{d_A} \pmod{n_A}.$$

and only then apply the encryption function of Bob to the signed message S:

$$C = S^{e_B} \pmod{n_B}.$$

To let Bob know that the message came from Alice one can add text to the signature S.

Now C is going to be sent to Bob. When Bob receives this message he first decodes it with his own private key d_B:

$$S = C^{d_B} \pmod{n_B}.$$

Now Bob has the signed message S of Alice. Subsequently Bob can find the message with the public key from Alice:

$$M = S^{e_A} \pmod{n_A}.$$

If Alice really sent the message, it should now appear unscrambled. If Bob wants to modify the message of Alice, he needs the d_A from Alice, which is private.

Example 6.1. Let's say Alice wants to send the message "14" to Bob. They both set up their RSA system. For example Alice has the following keys:

$$(n_A, E_A) = (143, 97) \qquad \text{and} \qquad d_A = 73.$$

And Bob has the keys:

$$(n_B, E_B) = (221, 85) \qquad \text{and} \qquad d_B = 61.$$

Alice first applies the decryption function to her message with her private key:

$$S = 14^{97} \mod 143 = 27.$$

next she encodes the message with the public key from Bob:

$$C = 27^{85} \mod 221 = 40.$$

Alice now sends the encoded message "40" to Bob. When Bob receives this message he first decodes the message with his private key:

$$S = 40^{61} \mod 221 = 27.$$

Next he applies the encoding function to the message with the public key from Alice:

$$M = 27^{97} \mod 143 = 14.$$

As you can see Bob received the right message.
But what happens when, say Carol modified the message in between Alice and Bob. So he changed the message "40" into say "32". Then Bob decodes the message with his private key:

$$S = 32^{61} \mod 221 = 14.$$

Next he encodes the message with the public key from Alice:

$$M = 14^{97} \mod 143 = 53.$$

Bob now ends up with the message "53". This was not the message Carol wanted to send. With big messages this last message will make no sense for Bob and he will notice that something is wrong.

7 Attacks

There are several attacks on the RSA system where the private key can be cracked. In this section two of such attacks are discussed.

7.1 Wiener attack

In 1990 Michael J. Wiener [14] found a way to solve the RSA encryption when d isn't large enough, to be precise, when d is smaller than $\frac{1}{3}n^{\frac{1}{4}}$. This proof is taken from [2].

Theorem 7.1. (*Wiener's attack*) Assume an RSA system is such that p and q are about the same size, $p < q < 2p$ and $e < \varphi(n)$. When d is smaller than $\frac{1}{3}n^{\frac{1}{4}}$, so

$$d < \frac{1}{3}n^{\frac{1}{4}}. \tag{16}$$

then the private key from this RSA system can be calculated in $\mathcal{O}(n \log n)$

Proof. Equation (10) is equivalent with saying that there exists an integer k such that

$$ed = k\varphi(n) + 1. \tag{17}$$

Since $n = p \cdot q > p^2$ we have that

$$\sqrt{n} > p \tag{18}$$

With (9) we have the following inequality:

$$\begin{aligned} -\varphi(n) &= -n + p + q - 1 \\ &< -n + 3p - 1 \\ &\overset{(18)}{<} -n + 3\sqrt{n}. \end{aligned}$$

So

$$n - \varphi(n) < 3\sqrt{n}. \tag{19}$$

Now together with $k(n - \varphi(n)) > 0$, we have the following estimate for $\left|\frac{e}{n} - \frac{k}{d}\right|$:

$$\begin{aligned} \left|\frac{e}{n} - \frac{k}{d}\right| &= \left|\frac{k}{d} - \frac{e}{n}\right| \qquad (20) \\ &= \left|\frac{kn - ed}{nd}\right| \\ &\overset{(17)}{=} \left|\frac{k(n - \varphi(n)) - 1}{nd}\right| \\ &\overset{(19)}{<} \frac{3k}{d\sqrt{n}}. \qquad (21) \end{aligned}$$

Together with (17) and the fact that we chose $e < \varphi(n)$ we now find that $k < d$ in the following way:

$$k\varphi(n) = ed - 1 < ed < d\varphi(n).$$

Since $k < d$ and by the assumption $d < \frac{1}{3}n^{\frac{1}{4}}$ we have $3k < 3d < n^{\frac{1}{4}}$. So with (21) we have the following estimate for $\left|\frac{e}{n} - \frac{k}{d}\right|$

$$\left|\frac{e}{n} - \frac{k}{d}\right| < \frac{n^{\frac{1}{4}}}{d\sqrt{n}} = \frac{1}{dn^{\frac{1}{4}}}. \tag{22}$$

Furthermore we have with assumption (16) that $2d < 3d < n^{\frac{1}{4}}$, so $\frac{1}{2d} > \frac{1}{n^{\frac{1}{4}}}$ and this implies together with (22)

$$\left|\frac{e}{n} - \frac{k}{d}\right| < \frac{1}{2d^2}.$$

Now the theorem of Legendre 4.5 implies that $\frac{k}{d}$ is a convergent of $\frac{e}{n}$. With theorem 4.6 and the fact that $e < \varphi(n) < n$ we see that finding all convergents of a fraction $\frac{e}{n}$ has a time complexity of order $\mathcal{O}(n \log n)$.
Let's now look at the test that a guess for $\frac{k}{d}$ is correct. Let $\frac{x}{y}$ be a convergent of $\frac{e}{n}$ and so a guess for $\frac{k}{d}$. Then we can calculate $\varphi(n)$ if the guess of the convergent is correct with (17) like

$$\varphi(n) = \frac{ey - 1}{x}.$$

Then we calculate a guess for $\frac{p+q}{2}$ with the following identity:

$$\frac{n - \varphi(n) + 1}{2} = \frac{pq - (p-1)(q-1) + 1}{2} = \frac{p+q}{2} \tag{23}$$

If the guess for $\frac{p+q}{2}$ is not an integer, then the guess for k and d is wrong. Next we can guess $\frac{p-q}{2}$ using the following identity

$$\left(\frac{p+q}{2}\right)^2 - pq = \left(\frac{p-q}{2}\right)^2. \tag{24}$$

If the guess for $\frac{p-q}{2}$ is now an integer the guess for k and d is correct. It is easy to obtain p and q from $\frac{p+q}{2}$ and $\frac{p-q}{2}$. The test to see whether a guess for $\frac{k}{d}$ is correct is in constant time. Together with the time complexity of finding the convergents we have that the private key can be found in order $\mathcal{O}(n \log n)$ □

Example 7.1. Let the public key (n, e) be equal to $(26667829759, 13173097379)$. We assume that $d < \frac{1}{3}n^{\frac{1}{4}} \approx 134$. Calculation of the continued fraction of $\frac{e}{n}$ gives

$$[2,\ 40,\ 1,\ 22,\ 13,\ 24,\ 1,\ 1,\ 11,\ 1,\ 2,\ 3,\ 1,\ 1,\ 25,\ 3].$$

The convergents of the continued fraction of $\frac{e}{n}$ are:

$$\frac{1}{2}, \frac{40}{81}, \frac{41}{83}, \frac{942}{1907}, \frac{10403}{21060}, \frac{11345}{22967}, \frac{21748}{44027}, \cdots$$

We are going to use the Wieners test to check if a convergent is the correct guess for $\frac{k}{d}$. Let's start with the first guess $\frac{x}{y} = \frac{1}{2}$. Calculation of the Euler totient function with (17) gives then

$$\varphi(n) = \frac{13173097379 \cdot 2 - 1}{1} = 26346194757$$

Then with the identity (23) we find the guess for $\frac{p+q}{2}$:

$$\frac{26667829759 - 26346194757 + 1}{2} = \frac{p+q}{2} = 160817501.5$$

This guess is not an integer, so the guess $\frac{1}{2}$ for $\frac{k}{d}$ is wrong.
Now we check the next convergent that is equal to $\frac{40}{81}$. The Euler totient function is now equal to

$$\varphi(n) = \frac{13173097379 \cdot 81 - 1}{40} = 26675522192.45$$

This can not be the case, because $\varphi(n)$ should always be an integer, so this choice for $\frac{k}{d}$ is also incorrect. The next convergent is equal to $\frac{x}{y} = \frac{41}{83}$ For $\varphi(n)$ we now have:

$$\varphi(n) = \frac{13173097379 \cdot 83 - 1}{41} = 26667489816$$

This guess will give the following outcome for $\frac{p+q}{2}$

$$\frac{p+q}{2} = \frac{26667829759 - 26667489816 + 1}{2} = 169972.$$

For the calculation of $\frac{p-q}{2}$ we use (24) and get:

$$\frac{p-q}{2} = \sqrt{169972^2 - 26667829759} = 47145.$$

From these two equations we get $p = 217117$ and $q = 122827$. The convergent was the right choice so we also found the private key $d = 83$.
We can do a little check to see if this choice of d is correct. Let's encrypt for example the message $M = 196$. The encoded message becomes $C = 196^{13173097379} \mod 26667829759 = 15142018378$. If we now decode this with our found d we find: $M = 15142018378^{83} \mod 26667829759 = 196$.

7.2 Fermat's factorization algorithms

Fermat factorization algorithm [6] gives a way of breaking the RSA code fast when the prime factors are too close to each other. The algorithm depends on the following identity, where $n = pq$:

$$(p+q)^2 - (p-q)^2 = p^2 + 2pq + q^2 + -p^2 + 2pq + -q^2 = 4pq = 4n.$$

So

$$4n = (p+q)^2 - (p-q)^2.$$

To find p and q we rewrite the equation a little with two new variables x and y to:

$$4n = x^2 - y^2. \tag{25}$$

The idea now is to find a solution for x and y of the equation (25). If we have x and y we can easily get p and q. Initially we choose x equal to $\lceil 2\sqrt{n} \rceil$[5] and $y = 0$. Next we look at identity (25) and rewrite it a little introducing a new variable df such that:

$$df = x^2 - (y^2 + 4n).$$

Now the algorithm can distinguish three different cases.

Case 1: $df = 0$
If $df = 0$, then we have $n = \frac{x}{2}^2 - \frac{y}{2}^2$. In this case we have found the two prime numbers such that $p = \frac{x+y}{2}$ and $q = \frac{x-y}{2}$.

Case 2: $df > 0$
If $df > 0$, then we have that x^2 is bigger than $y^2 + 4n$. This is why we increase y by 2. We increase y by 2 because $p + q$ is an even number. This correction we also have to apply to the df. df decreases in this case by $4y + 4$.

Case 2: $df < 0$
If $df > 0$, then we have that x^2 is smaller than $y^2 + 4n$. This is why we increase x by 2. We increase x by 2 because $p - q$ is an even number. This correction we also have to apply to the df. df increases in this case by $4x + 4$.

Formally this looks as follows:

Input: n
Output: n: p and q
$x = \lceil 2\sqrt{n} \rceil$;
$y = 0$;
$df = x^2 - (y^2 + 4n)$;
while $df \neq 0$ **do**
  if $df > 0$ **then**
    $df = df - (4y + 4)$;
    $y = y + 2$;
  else
    $df = df + (4x + 4)$;
    $x = x + 2$;
  end
end
$p = \frac{x+y}{2}$;
$q = \frac{x-y}{2}$;

Algorithm 3: Fermat's factorization algorithm

The time complexity of the Fermat factorization algorithm depends on how long we stay in the while loop. If $df > 0$ we stay in the while loop until x is at our desired $p + q$. Because we increase x every iteration by 2, this means we have $\frac{p+q}{2} - (\sqrt{n} - 1)$ iterations in the while loop for $df > 0$. For $df < 0$ we proceed in the same way, because we stay in the while loop until y is at our desired $p - q$. Because we increase y every iteration by 2, this means we have $\frac{p-q}{2} - 1$ iterations in the while loop for $df > 0$. So the

[5] With $\lceil c \rceil$ we mean c rounded up to a whole number.

total number of operations is $p - (\sqrt{n} + 1)$. So the algorithm has time complexity $\mathcal{O}(p - \sqrt{n})$. Fermat's factorization will be especially fast if p and q are very close to each other, then p and $\sqrt{n}$ will lie very close to each other and the order of the algorithm will then become almost of the order $\mathcal{O}(1)$.

Example 7.2. Let's run this algorithm with modulus $n = 11021$. The factorization of this modulus is $n = 107 \cdot 103$, so $p = 107$ and $q = 103$. These prime numbers lie very close to each other so we would expect to be done very fast. Let's apply the algorithm.

Try	1	2	3
Difference *df*	44100 - 4·11021 = 16	16 - (4·0 + 4) = 12	12 - (4·2 + 4) = 0
x	210	210	210
y	0	2	4

So we have $x = 210$ and $y = 4$ and with this $p = \frac{x+y}{2} = \frac{214}{2} = 107$ and $q = \frac{x-y}{2} = \frac{206}{2} = 103$.

8 Conclusion and discussion

8.1 Conclusion

This thesis explained the concept of the RSA system and focused on two attacks on this system.

The RSA system is hard to crack, because it depends on a one-way function. This function is the factorization of the modulus n. It is easy to compute the modulus with two prime numbers p and q, but factoring n is hard. The fastest known algorithm that is found is based on the number field sieve. This algorithm has a time complexity that is exponential, which still takes many years to run.

With Wiener's attack we saw that it is possible to crack the RSA system, using Legendre's theorem 4.5, in a time complexity that is linearithmic, when the private key was very small.

Another attack we saw was based on Fermat's factorization algorithm. This algorithm could find the prime factors of the modulus n in a time complexity that is between logarithmic and linear. Fermat's factorization algorithm will especially be a problem when the two prime lies very close to each other. Then the time complexity will almost become constant

There are several other attacks on the RSA system. However in practice these threats are easy to avoid by taking the parameters of the system in the right way and the RSA system is still used a lot.

8.2 Discussion

The security of the RSA system is not fully guaranteed. One of the reasons for this is that the security of this algorithm depends on the fact that our computers are not yet so fast that they can crack the RSA system in reasonable time. But innovation of the computers goes fast. Scientists are working on a quantum computer. Not only is this computer much faster the the computers nowadays, but there also exists an algorithm such that the RSA system can be attacked in linear time. This algorithm is called Shor's algorithm[6]. Shor's algorithm makes a guess for a number where you can get the prime numbers from. Now Shor's algorithm tries to make a better guess from this old guess. The reason why this cannot be done on the current computers relies on superposition. Quantum computers feature superposition and the current computers not. With superposition the computer can be at several states at the same time. The idea is that with superposition the wrong guesses destructively interfere with each other. The current computers will go through all of these guesses and that takes ages. It could be that in a couple of years the RSA system isn't as safe as it is nowadays.

There is also no proof that the algorithm to factor the modulus using the number field sieve is the fastest one. There might be an algorithm to factor the modulus even faster. This would mean that the RSA system can be cracked in a decent time on the current computer. So in time it could well be possible that the RSA system is not so safe anymore.

[6]Shor's algorithm was founded in 1994 by Peter Shor. For more information about Shor's algoritme, see [8]

References

[1] Churchhouse, R. (2001). *Codes and ciphers* accessed on 3 June 2021. Cambrigde. Retrieved from http://www.ik4hdq.net/codici_cifr.pdf

[2] Dujella, A. *A variant of Wiener's attack on RSA* accessed on 16 May 2021, Retrieved from https://eprint.iacr.org/2008/459.pdf

[3] Emory Oxford College. *Transposition ciphers* accessed on 22 May 2021, Retrieved from http://mathcenter.oxford.emory.edu/site/math125/transpositionCiphers/

[4] En-naoui, E. (2021). *Some remarks on sum of Euler's totient function* accessed on 17 May 2021, Retrieved from https://www.researchgate.net/publication/348294660_Some_remarks_on_sum_of_Euler's_totient_function

[5] Gayathri, D. S.,Selvam, K., Rajagopalan, S.P. (2011) *An abstract to calculate big O factors of time and space complexity of machine code.* SEISCON. accessed on 21 July 2021, Retrieved from https://www.researchgate.net/publication/330778984_An_abstract_to_calculate_big_O_factors_of_time_and_space_complexity_of_machine_code

[6] Hazem M. Bahig, H. M., Mahdi, M. A., Alutaibi, K. A., AlGhadhban, A, Bahig, H.M. (2020) *Performance analysis of Fermat factorization algorithms* , IJACSA, Vol 11. accessed on 28 June 2021, Retrieved from https://thesai.org/Downloads/Volume11No12/Paper_42-Performance_Analysis_of_Fermat_Factorization_Algorithms.pdf

[7] Natsui, N. (2010) *On the Legendre constant of α-continued fractions.* Journal of number theory 131, 487-507 accessed on 16 July 2021, Retrieved from https://core.ac.uk/download/pdf/82704843.pdf

[8] Shor, P. W. (1991) *Algorithms for quantum computation: discrete logarithms and factoring* . IEEE transactions on information theory, Vol 94. accessed on 10 August 2021, Retrieved from http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.123.5183&rep=rep1&type=pdf

[9] Smeets, I. (2010) *On continued fraction algorithms* accessed on 22 May 2021, Retrieved from https://www.math.leidenuniv.nl/scripties/SmeetsThesis.pdf

[10] Stevenhagen, P. (2021) *Algebra 1.* accessed on 2 July 2021. Retrieved from http://websites.math.leidenuniv.nl/algebra/algebra1.pdf

[11] Stevenhagen, P. (2008) *The number field sieve.* accessed on 1 July 2021. Algorithmic Number Theory MSRI Publications, Vol 44. Retrieved from https://www.math.leidenuniv.nl/~psh/ANTproc/04psh.pdf

[12] Rivest, R.L., Shamir A. & Adleman L. (1977). *A method for obtaining digital signatures and public-key cryptosystems.* accessed on 9 May 2021, Retrieved from https://people.csail.mit.edu/rivest/Rsapaper.pdf

[13] Robinson, J. (2011) *Fermat's little theorem* accessed on 30 May 2021, Retrieved from https://www.math.arizona.edu/~ime/ATI/Math%20Projects/C1_MathFinal_Robinson.pdf

[14] Wiener, M. J. (1990) *Cryptanalysis of short RSA secret exponents.* IEEE transactions on information theory, Vol 36. accessed on 22 May 2021, Retrieved from http://jannaud.free.fr/Fichiers/Travail/wiener.pdf

A Appendix

A.1 Elucidation code RSA system

For this report a simulation of the RSA system was made in Python (version 3.9.6), see appendix A.2 for the code. The program is simple, which means that it doesn't work in practice for very large prime numbers. It is just for the purpose of understanding the algorithm better.

The program consists of 11 functions:

function	input	Explanation
GiveInputPrimes	*inputprime*	Asks the user to give two prime numbers and checks if the input is correct.
GiveInputMessage	*message*, n	Asks the user to give a message for encryption and checks if the input is correct.
IsPrime	p	Checks if p is prime
Divisor	p	Finds the first divisor of p that is not 1
Gcd	p, q	Finds the greatest common divisor of p and q
EulerTotient	n	Finds the Euler totient function of n
OptionD	*phin*	Finds and chooses an option for an private key
PublicExp	d, *phin*	Calculates the public key
Encode	m, e, n	Encodes the message
Decode	c, d, n	Decodes the message
Final	p, q	Asks input from the user and creates the RSA system

When running the program the first thing the program asks is to give two primes. Next the program runs the Final function with these two primes. From there the other functions are called.

The RSA system created by the program is not unique, because there can be several other cases depending on the choice of the public key e an the private key d. The program chooses these parameters random.

Example A.1. Let's run the program one time. The first thing the program asks is to give two primes. Let's say we choose the first prime number equal to 3 and the second to 11. Now the program sets up an RSA system. Now the system will ask which message you want to encrypt, say you want to send 28 for example. The system encrypts this message and decrypts it and we see that we get 28 out of the system after encyption and decryption. In figure (A.1) we see the output of the program that is just described.

```
Choose a prime number p = 3
Choose a prime number q = 11
The public key is equal to (n, e) = (33, 19)
The choice made for the private key d is equl to d = 19
The public exponent is the equal to 19
Now we can start sending a message!
What is the message that Alice wants to send? (has to be an integer between 1 and 32)28
Bob received the message: 13
If Bob decodes this message he received the message: 28
```

Figure A.1: RSA program running

A.2 Code RSA system (RSASystem.py)

```
"""
This program implements the RSA algorithm.

First it generates the RSA keys in the following way:
1. Let the user choose two primes p and q.
2. Calculate the modulus n = pq.
3. Calculate phi(n) = (p-1)(q-1).
4. Choose a private key relatively prime to phi(n) such that gcd(d,phi(n)) = 1.
5. Calculate the public key e = d^(-1) mod (phi(n)).

Next it encrypts and decrypts the input message (message integer between 1 and n).
Encryption of a message M: M^e mod n.
Decryption of a encrypted message C: C^d mod n.
"""

import random

def GiveInputPrimes(inputprime):
    """
    Asks the user to give two prime numbers and checks if the input is correct.
    """
    while True:
        p = int(input(inputprime))
        if not (IsPrime(p)):
            print("Sorry, this last input is not a prime number")
            if p < 1:
                print("A prime number must be bigger or equal to one")
                print("Please give a prime number as input")
            else:
                dp = Divisor(p)
                print(str(p) + " is the product of " +  str(dp) + " and " + str(p//dp))
                print("Please give a prime numbers as input")
                continue
        else:
            break
    return p

def GiveInputMessage(message, n):
    """
    Asks the user to give a message (message integer between 1 and n) for encryption.
    and checks if the input is correct.
    """
    while True:
        M = int(input(message))
        if (M> n-1 or M<=0):
            print("Sorry, the message has to be between 1 and " + str(n-1))
        else:
            break
    return M

def IsPrime(p):
    """
    Checks if p is prime.
    """
    if p > 1:
        for i in range(2, p):
```

```
            if (p % i) == 0:
                return False
    else:
        return False
    return True

def Divisor(p):
    """
    Finds the first divisor of p that is not 1.
    """
    for i in range(2,p+1):
        if (p % i) == 0:
            return i

def Gcd(p, q):
    """
    Finds the greatest common divisor of p and q.
    """
    k=1
    for i in range(2, p+1):
        if (p%i)==0:
            if(q%i)==0:
                k=i
    return k

def EulerTotient(n):
    """
    Finds the Euler totient function of n.
    """
    k = 1
    if n==1:
        return 1
    else:
        for i in range(2,n):
            if Gcd(n, i) == 1:
                k = k+1
    return k

def OptionD(phin):
    """
    Finds and chooses an option for an private key.
    """
    optiond = []
    for i in range(2, phin):
        if Gcd(i, phin)==1:
            optiond.append(i)
    d = random.choice(optiond)
    return d

def PublicExp(d,phin):
    """
    Calculates the public key.
    """
    for i in range(2, phin):
        if (d*i % phin) == 1:
            return i

def Encode(m, e, n):
```

```
    """
    Encodes the message.
    """
    return (m**e % n)

def Decode(c, d, n):
    """
    Decodes the message.
    """
    return (c**d % n)

def Final(p,q):
    """
    Asks input from the user and creates the RSA system.
    """
    n = p * q
    phin = EulerTotient(n)
    d = OptionD(phin)
    e = PublicExp(d, phin)
    print("The public key is equal to (n, e) = (" + str(n) + ", " + str(e) + ")")
    print("The choice made for the private key d is equl to d = " + str(d))
    print("The public exponent is the equal to " + str(e))
    print("Now we can start sending a message!")
    M = GiveInputMessage("What is the message that Alice wants to send?" +
                         "(has to be an integer between 1 and " +
                         str(n-1) + ")", n)
    C = Encode(M, e, n)
    print("Bob received the message: " + str(C))
    print("If Bob decodes this message he received the message: " + str(Decode(C,d,n)))

p = GiveInputPrimes("Choose a prime number p = ")
q = GiveInputPrimes("Choose a prime number q = ")
Final(p,q)
```

A.3 Elucidation code Fermat factorization (FermatFactorization.py)

Appendix A.4 contains the code of the program of the Fermat factorization in Python (Version 3.9.6). The program asks a modulus and next factors this with Fermat's method. This program only make use of one function: FermatFactorization. The first thing the program does is ask for the modulus input from the user and next the program runs the Fermat factorization with the function FermatFactorization.

The program consists of 4 functions: When running the program the first thing the program asks is

function	input	Explanation
GiveInputModulus	*inputmodulus*	Asks the user to give a modulus that they want to factor and checks if the input is correct.
IsPrime	p	Checks if p is prime
FermatFactorization	n	Executes Fermat's factorization.
Final	n	Asks input from the user and executes Fermat's factorization function.

to give the modulus you want to factor. Next the program runs the Final function with this modulus. From there the other functions are called.

Example A.2. Let's run the program with the example (7.2) from before. In this example the modulus was 11021 and the factors were $p = 107$ and $q = 103$. If we fill in the modulus 11021 in the program we get the correct output as one can see in figure (A.2).

```
Choose the modulus to apply to the Fermat factorization n = 11021
The first prime in the factoriation of n is equal to p = 107
The second prime in the factoriation of n is equal to q = 103
```

Figure A.2: RSA program running

A.4 Code Fermat factorization

```
"""
This program implements the Fermat Factoriation to crack the RSA system.
"""

import math
def GiveInputModulus(inputmodulus):
    """
    Asks the user to give a modulus that they want to factor and checks if the input is
    """
    while True:
        n = int(input(inputmodulus))
        primes = FermatFactorization(n)
        p = int(primes[0])
        q = int(primes[1])
        if not(IsPrime(p) and IsPrime(q)):
            print("Sorry, this modulus is not the product of two prime numbers")
            continue
        else:
            break
    return n

def IsPrime(p):
    """
    Checks if p is prime.
    """
    if p > 1:
        for i in range(2, p):
            if (p % i) == 0:
                return False
    else:
        return False
    return True

def FermatFactorization(n):
    """
    Executes Fermat's factorization.
    """
    x = 2* math.ceil(math.sqrt(n))
    y = 0
    df = x*x - (y*y+4*n)
    t = 0
    while df != 0:
        t=t+1
        if df > 0:
            df = df - (4*y+4)
            y = y+2
        else:
            df = df + (4*x+4)
            x = x+2
        if t == n:
            return(0, 0)
    p = (x+y)/2
    q = (x-y)/2
    return (p, q)

def Final(n):
```

```
    """
    Asks input from the user and executes Fermat's factorization function.
    """
    primes = FermatFactorization(n)
    p = primes[0]
    q = primes[1]
    print("The first prime in the factoriation of n is equal to p = " + str(int(p)))
    print ("The second prime in the factoriation of n is equal to q = " + str(int(q)))

n = GiveInputModulus("Choose the modulus to apply to the Fermat factorization n = ")
Final(n)
```