2022

On Implementing And Testing The RSA Algorithm

Kien Trung Le
The College of Wooster, kle22@wooster.edu

On Implementing and Testing the RSA Algorithm

Independent Study Thesis

Presented in Partial Fulfillment of the Requirements for
the Degree Bachelor of Arts in Mathematics and Computer
Science in the
Department of Mathematical and Computational Sciences
at The College of Wooster

by
Kien Le
The College of Wooster
2022

Advised by:

Sofia Visa (Mathematical and

Computational Sciences)

THE COLLEGE OF
WOOSTER

Abstract

In this work, we give a comprehensive introduction to the RSA cryptosystem, implement it in Java, and compare it empirically to three other RSA implementations. We start by giving an overview of the field of cryptography, from its primitives to the composite constructs used in the field. Then, the paper presents a basic version of the RSA algorithm. With this information in mind, we discuss several problems with this basic conception of RSA, including its speed and some potential attacks that have been attempted. Then, we discuss possible improvements that can make RSA runs faster and more secure. On the software side, we implement the RSA algorithms in Java and compare its performance and security against the other three implementations.

This work is dedicated to the future generations of Wooster students.

Acknowledgments

I would like to thank Dr. Visa for her guidance and support throughout this project. The advises she gave me throughout the year has been invaluable in this work. Furthermore, her listening to me ranting really helped me see the flaws in my ideas and further develop them. I would also like to thank Dr. Kelvey for sparking my interest for mathematics and, with it, cryptography.

I would like to express my gratitude to my family for their love and support. Without them, I would not have been able to survive through IS.

Vita

Fields of Study Major field: Mathematics and Computer Science

Specialization: Cryptography

Contents

Abstract v

Dedication vii

Acknowledgments ix

Vita xi

Contents xiii

List of Figures xv

List of Tables xvii

List of Listings xix

CHAPTER PAGE

1 Introduction 1

2 Introduction to cryptography 3
- 2.1 What is encryption? . . . 3
- 2.2 Encryption algorithm example: Caesar cipher . . . 4
- 2.3 Types of encryption algorithms . . . 6

3 The RSA algorithm 9
- 3.1 Background mathematics . . . 9
  - 3.1.1 Modular arithmetic . . . 9
  - 3.1.2 Euler's totient function . . . 12
- 3.2 The RSA algorithm . . . 12
  - 3.2.1 A small note on the terminology of RSA . . . 15
- 3.3 Applying the RSA algorithm . . . 15
- 3.4 Proof of correctness . . . 16

4 Algorithm complexity analysis - execution time 21
- 4.1 Quantifying execution time . . . 22
  - 4.1.1 Asymptotic notation and algorithm complexity . . . 23
  - 4.1.2 Ordering of asymptotic notation . . . 26
  - 4.1.3 Asymptotic notation with number-theoretic algorithms . . . 27

5 Problems with basic RSA 33
5.1 Prime selection . . . 33
5.1.1 Trial division . . . 34
5.1.2 Fermat's algorithm . . . 35
5.1.3 Pollard's $p-1$ algorithm . . . 38
5.2 Chosen texts attack . . . 40
5.2.1 Chosen plaintext attack . . . 40
5.2.2 Chosen ciphertext attack . . . 42
5.3 Run time of basic RSA . . . 43
5.3.1 Implementation . . . 44
5.3.2 Complexity of RSA . . . 48
5.3.3 Runtime in context . . . 51

6 Improving the RSA cryptosystem 53
6.1 Strong key generation . . . 53
6.2 Padding . . . 54
6.3 Improving RSA's runtime . . . 55
6.3.1 Choice of public key . . . 56
6.3.2 Chinese remainder theorem . . . 58

7 Implementing and testing MyRSA 61
7.1 Research question . . . 61
7.2 The implementations of RSA . . . 62
7.2.1 Python-RSA . . . 62
7.2.2 PyCryptodome . . . 64
7.2.3 JDK . . . 66
7.2.4 Our Java implementation of RSA: MyRSA . . . 69
7.3 Testing methodology . . . 74
7.4 Result . . . 76
7.4.1 Performance test . . . 76
7.4.1.1 Vulnerability test . . . 79

8 Conclusions and Future Works 81

References 83

List of Figures

Figure		Page
1.1	Basic scenario in cryptography. Here, Eve intercepts and reads the message Alice is sending to Bob.	1
2.1	Basic scenario in cryptography when encryption is used. Unlike the scenario presented in Figure 1.1, Eve cannot decrypt and read the ciphertext she intercepts without the key.	4
2.2	Illustration of encryption process in Caesar cipher [5].	5
2.3	Illustration of private-key encryption [25].	8
2.4	Illustration of public-key encryption [13].	8
3.1	Modular arithmetic visualized [33].	11
4.1	The sin function is bounded by the values 1 and −1. (Made in Desmos)	24
4.2	Graphs of some functions whose O-notation is used in algorithm analysis. Here, n is the size of the input and N is the number of steps executed [7].	28
7.1	The results of testing encryption speed.	76
7.2	The results of testing encryption speed, limited to the Java implementations.	77
7.3	The results of testing decryption speed.	78

List of Tables

Table		Page
5.1	Calculations made by Fermat's method with the given example. . . .	37
7.1	Results for timing attack test. Each data point is in second per encryption.	80

List of Listings

Listing Page

3.1 Pseudocode for the RSA algorithm. The details of the helper functions used in RSA_INIT() will be provided in section 5.3.1 15

4.2 Pseudocode for insertion sort 21

5.3 Pseudocode for trial division algorithm. 35
5.4 Pseudocode for Fermat's method. 37
5.5 Pseudocode for Pollard's $p - 1$ algorithm. 38
5.6 Pseudocode for prime generation algorithm. 44
5.7 Pseudocode for calculation of public key in RSA algorithm. 45
5.8 Pseudocode for extended Euclidean algorithm [8]. 46
5.9 Pseudocode for modular exponentiation using repeated squaring. . . 48

6.10 Pseudocode for Euclidean algorithm [8]. 57

7.11 Example usage of Python-RSA. 63
7.12 Example usage of PyCryptodome 66
7.13 Example usage of Java's builtin RSA. Some code was taken from [21]. 68
7.14 Using Java's BigInteger library to generate the two primes needed for the RSA algorithm. 70
7.1 Function used to generate the keys for the RSA algorithm. 71
7.15 The complete code for MyRSA 74

CHAPTER 1

Introduction

Secure exchange of messages is important and is the main concern of cryptography. Consider the scenario illustrated in Figure 1.1. Alice sends Bob a message that contains some very important secrets. However, on the way to Bob, the message is taken and read by Eve, who then proceeds to leak those secrets to the world. If the content of the message is some mundane secret, such as Alice's secret recipe for a cake, there would not be much of a problem. However, suppose that the message contains important state and military secret. If those are to be revealed, significant issues would arise.

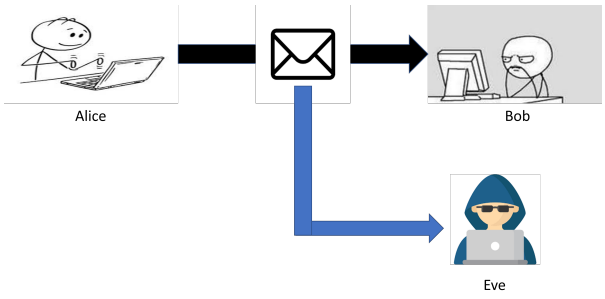

Figure 1.1: Basic scenario in cryptography. Here, Eve intercepts and reads the message Alice is sending to Bob.

As have been shown, the consequence of secret messages being revealed to the unintended recipients can be significant. As a result, many methods have been used to prevent this from happening, ranging from hiding the messages in a location that is hard to access[30] and various encryption techniques. Encryption, the method of

transforming a message to make it undecipherable, has consistently been the most important and remains so up till now.

In this project, we present and implement the RSA encryption algorithm. Then, we test our implementation against existing ones. chapter 2 and 3 introduce the field of cryptography the RSA algorithm, respectively. Chapter 4 defines the complexity an algorithm and how it is evaluated. In chapter 5, we point out three weaknesses of the RSA algorithm presented in chapter 3, and then presenting ways to address them in chapter 6. In chapter 7, we describe our own Java RSA implementation and compare it against three existing implementations. Finally, we include concluding remarks and point out future work in chapter 8.

CHAPTER 2

Introduction to cryptography

In this chapter, we give a brief introduction to encryption and presents one example algorithm, the Caesar cipher. Afterwards, we introduce the two types of encryption, public-key and private-key.

2.1 What is encryption?

Cryptography is, at its core, "the study of methods of secure communication between two parties" [26]. Generally, we call the two parties Alice and Bob. The need for secure communication comes up due to someone trying to intercept Alice and Bob's communication and read it. We call them the eavesdropper, and henceforth, Eve based on the convention shared by [30] and [26].

In cryptography, we are first concerned with **messages** (also called **plaintexts**), which are the communications between Alice and Bob. These can take many forms, including characters, words, and numbers. In order to ensure that Eve cannot read their messages, Alice and Bob will **encrypt** their message, transforming them into *ciphertexts*, which usually have no discernible meaning. These ciphertexts will be sent in places of the original messages. With the ciphertext in hand, Alice and Bob can then **decrypt** it to obtain the messages. In the process of encryption and decryption, visualized in Figure 2.1, a **key**, which controls how a message is encrypted and decrypted, is used [18].

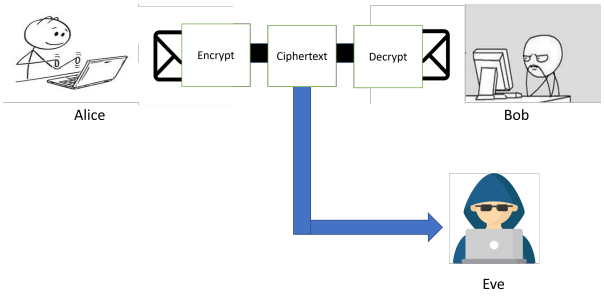

Figure 2.1: Basic scenario in cryptography when encryption is used. Unlike the scenario presented in Figure 1.1, Eve cannot decrypt and read the ciphertext she intercepts without the key.

Mathematically, we can describe an encryption algorithm as a function $f(m, k)$, where m is the message to be encrypted and k is the key used. Then, the corresponding decryption algorithm will be the inverse of the encryption algorithm, $f^{-1}(c, k)$, with $c = f(m, k)$ being the ciphertext to be decrypted [18].

2.2 Encryption algorithm example: Caesar cipher

One of the simplest and oldest encryption algorithm is the Caesar cipher, so named because Julius Caesar used it frequently in his communication [35]. The idea behind this cipher is that shifting each letter in a message to the right by 3 position in the alphabet is a procedure that

- is easily reversible
- turns a message into a form that cannot be understood
- is easy to reverse by the intended recipient, but presents difficulty for everyone else.

These three properties make the shifting operation a great candidate for an encryption algorithm.

Now, we define the encryption function in the Caesar cipher as the action of shifting each character in the message to the right by three positions in the alphabet

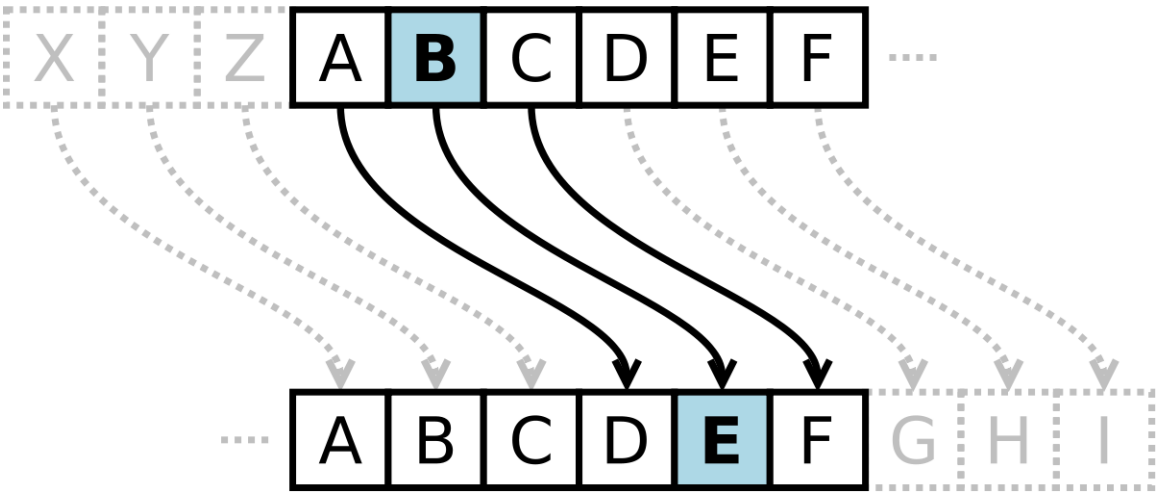

Figure 2.2: Illustration of encryption process in Caesar cipher [5].

[35], as shown in Figure 2.2. Thus, b becomes e, f becomes i, and so on. This procedure, however, does not work on x, y, or z. A workaround is to prefix the alphabet with them, as seen in Figure 2.2. This way, we can move x, y, and z three characters to the left in the alphabet. Applying this encryption algorithm to the message $m = Helloworld$, we obtain the ciphertext $c = f(m,k) = Khoorwruog$, where $k = 3$ is the move right by three as explained above. This ciphertext has no discernable meaning (in English, at the very least), and so no one would know what we were trying to say with it.

On the side of the receiver, however, decrypting this ciphertext to obtain the original message is simple. We only need to move each character in the ciphertext three positions to the left in the alphabet. We can see that moving right three positions and then moving left three positions takes us back to our original position, which means that this operation turns the ciphertext back into the message.

We now define a more general version of the Caesar cipher in formal terms. Let the key k be a natural number between 0 and 25. Here, k represents the number of positions to shift each character in the alphabet. If we let $k = 3$, we get the classical Caesar cipher. Now, for each character C in the message m, we let p be the position

of that character in the alphabet. So, we define the encryption function as

$$f(C,k) = C' = \text{Letter at position } \begin{cases} p+k & \text{if } p+k \leq 26 \\ p+k-26 & \text{otherwise} \end{cases}$$

and the decryption function as

$$f^{-1}(C',k) = \text{Letter at position } \begin{cases} c-k & \text{if } c-k \geq 1 \\ c-k+26 & \text{otherwise} \end{cases}$$

2.3 Types of encryption algorithms

Encryption algorithms are divided into two main types based on the mechanism behind encryption and decryption, **private-key** (or **symmetric**) and **public-key** (or **asymmetric**). In a private-key encryption algorithm, the message is encrypted and decrypted using the same key (Figure 2.3). One example of this type of encryption is the Caesar cipher we presented above. Because it would be trivial for Eve to decrypt a ciphertext if she has the key, it is kept private between Alice and Bob.

The main challenge of private-key encryption algorithms is key establishment [30]. Because Alice and Bob use the same key for both encryption and decryption, they must first come to an agreement of what key to use. They cannot use a public channel for this purpose, for Eve can easily intercept the key there. One option is for Alice and Bob to meet in person to establish the key. While this might be possible when they are in the same general geographic location, in many cases, such as in warfare or over the internet, it is infeasible to do so. The issue gets even worse when more people are involved, seeing as each person needs to exchange (preferably distinct) keys with the others. As a result, the number of keys increases quickly.

To avoid the problem of key distribution, public-key encryption algorithms were

invented. As illustrated in Figure 2.4, public-key encryption algorithms use two sets of keys, public keys and private keys. The public keys are released to a public channel, where everyone, including Eve, can access it, while the private keys are only known to the receiver of a message. In order to encrypt a message, the sender only needs the public key. The receiver can then use the private key to decrypt the received ciphertext. This method of managing keys is a massive improvement over the method of private-key encryption, seeing that secure communication are now possible without Alice and Bob having to agree on a key beforehand.

With that said, it is also important to understand that private-key encryption algorithms have some important advantages over public-key algorithms. For one, private-key algorithms are usually faster then public-key algorithms [12]. While this speed advantage does not matter when only considering the communication between two people, there are many instances where communication can be between billions of entities, with the internet being the most prominent example. As a result, it is preferable to use private-key encryption in those cases. Furthermore, there are many more private-key than public-key algorithms, owing to the fact that public-keys encryption were only conceived recently in the 1970s by Diffie and Hellman [18], while private-key algorithms has been in use for about 2000 years as indicated by the Caesar cipher [1]. Because of this, it is feasible to choose the algorithms used to best suit an application with private-key encryption, while the same tasks might be impossible with public-key encryption.

The RSA algorithm explained in details in the next chapter is an example of a public-key cryptosystem, and is the focus of our study.

[1] Julius Caesar was alive before the year 1 AD

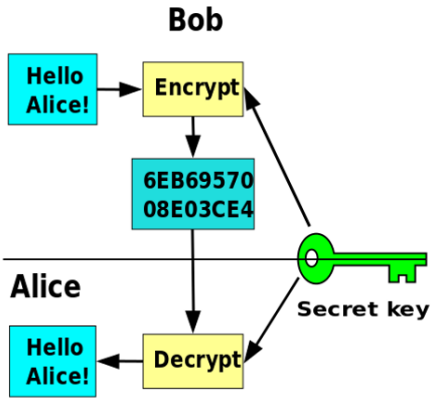

Figure 2.3: Illustration of private-key encryption [25].

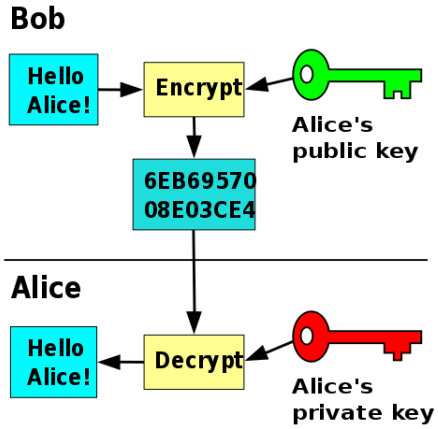

Figure 2.4: Illustration of public-key encryption [13].

CHAPTER 3

The RSA algorithm

The RSA cryptosystem was created in 1977 by Ron **R**ivest, Adi **S**hamir, and Leonard **A**dleman [30]. It is the culmination of the efforts to create a public-key cryptosystem, which was first proposed by Difle and Hellman a few years ago [30]. The core idea behind the RSA algorithm is simple: it is easy to multiply two numbers, but to take a composite number and find all its prime factors is incredibly difficult. For example, it is very easy to calculate the product of $p = 19,211$ and $q = 35,291$, with most modern computers being able to do it practically instantly. However, if given the product $pq = 677,975,401$, it can take some time to find p and q.

In this chapter, we introduce the mathematical concepts used in the RSA and subsequently a formal description of the algorithm alongside with its pseudocode. Furthermore, we give a concrete example of the application of the RSA algorithm and prove its correctness. Finally, we introduce the RSA cryptosystem.

3.1 Background mathematics

3.1.1 Modular arithmetic

One of the central mathematical construct used in RSA is congruence mod n.

Definition 3.1 Congruence: *For some integer n, two integers a and b are **congruent modulo** n, denoted $a \equiv_n b$ or $a \equiv b \pmod{n}$, if they have the same remainder when divided by n.*

In this definition, n is called the modulus. By convention, we abbreviate the word modulo to mod. For an example of this concept, we have $17 \equiv 5 \pmod{3}$ because they both have a remainder of 2 when divided by 3.

Sometimes, it is useful to state the definition of congruence using the division lemma.

Theorem 3.1 (Division Lemma)**.**

For two arbitrary integers a, b, there exists unique integers p, q such that $a = qb + r$ [14].

In Theorem 3.1, q, r are respectively the quotient and remainder of the division a/b. In the special case where $r = 0$, we have the concept of a number dividing another.

Definition 3.2 Divide: *Given two integers a, b, we say that b divides a, denoted $b \mid a$, if there exists some integers q such that $a = qb$ [14].*

Now, we can construct an alternative definition of congruency. Suppose that we have two integers a, b such that $a \equiv b \pmod{n}$ for some integer n. Per the definition of congruence, we know that a/n and b/n have the same remainder. Applying the division lemma, we get $a = q_1 n + r$ and $b = q_2 n + r$. We than have $a - b = q_1 n + r - q_2 n - r = (q_1 - q_2)n$. Seeing as $q_1 - q_2$ is an integer, by Definition 3.2, we know that $a - b \mid n$. So, we have another definition of congruence.

Definition 3.3 Congruence: *For some arbitrary integer n, two integers a and b are **congruent mod** n if $(a - b) \mid n$ [14].*

The concept of congruence gives rise to a new system of arithmetic, modular arithmetic. In modular arithmetic, equality is replaced with congruence mod some value. Visually, this transition is like going from moving on a line to taping the

two ends of that line together and moving on the resulting circle. An example of modular arithmetic can be observed in Figure 3.1, where we are working modulo 12. In this figure, we are moving 4 units away from 9. In normal addition, this would give us $9 + 4 = 13$, but in modular arithmetic, we get $9 + 4 \equiv_{12} 13 \equiv_{12} 1$.

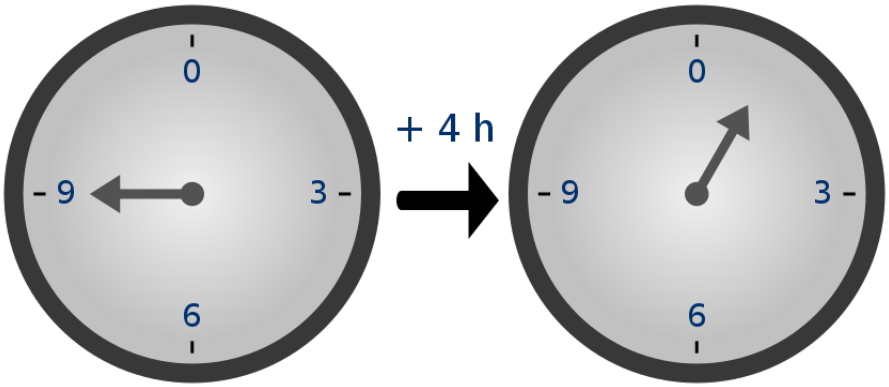

Figure 3.1: Modular arithmetic visualized [33].

As is the case in normal arithmetic, we can solve equations in modular arithmetic. In our case specifically, we are interested in linear modular equations, which are equations of the form $ax \equiv b \pmod{n}$. Here, we will note that an equation in modular arithmetic is satisfied by any number that has the same remainder as ax when divided by the modulus. In other words, an equation can have multiple solutions. The majority of the time, however, only the smallest positive solution is useful, and so we will usually take that as the solution to the equation.

Before ending this section on modular arithmetic, we would like to introduce a related concept, the modulo operation, denoted mod. This operation takes in two natural numbers a, n, with the whole expression taking the form of $a \bmod n$, and returns the remainder of a/n. We can easily see that this operation solves the problem of finding the smallest number that is congruence to $a \bmod n$. Thus, we can use the modulo operation to solve linear modular equations of the form $x \equiv b \pmod{n}$.

3.1.2 Euler's totient function

Another important mathematical construct for the RSA algorithm is the Euler's totient function

Definition 3.4 Euler's Totient Function: *Euler's totient function for a natural number n, denoted $\phi(n)$, is the number of natural numbers less than n that is coprime to n. Here, we call two numbers coprime if they share no prime factors.*

One important fact about the totient function is that because a prime number is coprime to every number less than it, $\phi(n) = n - 1$ if n is prime. For instance, $\phi(13) = 13 - 1 = 12$, seeing that 13 is the only prime factor of 13, and no number in the range $[1, 12]$ can have 13 as a factor. Another important property of ϕ is that it is a multiplicative function, or $\phi(xy) = \phi(x)\phi(y)$ for any two natural numbers x, y. For example, this property lets us know that $\phi(15) = \phi(3 * 5) = \phi(3) * \phi(5) = 2 * 4 = 8$. This is especially useful when we are calculating the value of ϕ for a product of primes, seeing that we can easily compute ϕ for the prime factors, and then multiply them together to get ϕ for the whole product.

With these basic math concepts defined, we are ready to explain the RSA algorithm.

3.2 The RSA algorithm

The description of the RSA algorithm given below is based on the one in [16].

The first step in the RSA algorithm is generating two distinct prime numbers p, q, as shown in Listing 3.1. This can be accomplished through many means, such as generating a large number of primes and picking two. With these numbers in hand, we proceed to calculating their product $N = pq$. This number will be released to the public as part of the public key.

After this, we calculate a number e that is coprime to $\phi(N)$. In order to do this, we first need to calculate $\phi(N)$. Because p, q are primes, we know that $\phi(p) = p - 1$ and $\phi(q) = q - 1$. Then, using the fact that ϕ is multiplicative, we obtain

$$\phi(N) = \phi(p)\phi(q) = (p-1)(q-1). \tag{3.1}$$

With $\phi(N)$ calculated, we can calculate e, the details of which will be covered in 5.3.1. After this, we have obtained the set of public keys, the number N and e. Now, we need to calculate the private key d, which is defined as the multiplicative inverse of e mod $\phi(N)$. Borrowing from the definition of multiplicative inverse in conventional arithmetic, we can write this as

$$de \equiv 1 \pmod{\phi(N)} \tag{3.2}$$

where d is the private key.

While they are defined in the same way, the notion of multiplicative inverse in modular arithmetic is different from the one in conventional arithmetic. When working in conventional arithmetic, the inverse of a number a is the fraction $\frac{1}{a}$. In modular arithmetic, however, taking the inverse this way does not work because we can only work with integers, which $\frac{1}{a}$ is not unless $a = \pm 1$. Thus, we have to find some other way to calculate the multiplicative inverse in modular arithmetic. We will leave this to an upcoming chapter.

With both set of keys having been calculated, we now move on to the encryption and decryption steps. Let m be the message that we are trying to encrypt. In the RSA cryptosystem, m is a natural number such that $1 \leq m < N$. If the message that needs to be sent is not in this form, it is not too difficult to construct a method to transform it into a natural number. For instance, if our message is composed solely of the original 127 ASCII characters, we can assign each character in the

message their ASCII code, which is always a natural number. Then, to calculate the ciphertext, we solve the modular equation

$$c \equiv m^e \pmod{N} \tag{3.3}$$

On the other hand, in order to decrypt the ciphertext, we solve the modular equation

$$m \equiv c^d \pmod{N} \tag{3.4}$$

Now, as have been shown in the previous section, we can solve modular equations with the mod operation. Using this, we can define the encryption function as

$$f(m, e, N) = m^e \mod N$$

and the decryption function as

$$f(c, d, N) = c^d \mod N$$

Converting the algorithm described above into pseudocode, using the modulo operation to solve Equations 3.3 and 3.4, we obtain the code in Listing 3.1.

```
1 GLOBAL_VAR p, q, e, d
2
3 FUNCTION RSA_INIT():
4     p = GENERATE_PRIME_NUMBER()
5     q = GENERATE_PRIME_NUMBER()
6     N = p * q
7     e = MAKE_PUBLIC_KEY(N)              // Public key
8     d = FIND_INVERSE_MOD_N(e, N)        // Private key
```

```
9
10 FUNCTION RSA_ENCRYPT(m):
11     RETURN m^e mod N
12
13 FUNCTION RSA_DECRYPT(c):
14     RETURN c^d mod N
```

Listing 3.1: Pseudocode for the RSA algorithm. The details of the helper functions used in RSA_INIT() will be provided in section 5.3.1

3.2.1 A small note on the terminology of RSA

In the above definition of the RSA algorithm, there is no way to refer to any of the individual key without calling them by their symbolic name. However, this requires us to have assigned them the symbols to the keys and to remember what each symbol stands for, which can be tedious and confusing. Thus, we define plain language terms to describe each key. We call the pair p and q the prime numbers as this is their main role in the RSA algorithm. We call the public key e the public exponent and the private key d the private exponent due to their role as the exponents in the modular exponentiation operations used for encryption and decryption. Finally, we call N the modulus based on how it acts as the modulus of the modular equations in the encryption and decryption steps.

3.3 Applying the RSA algorithm

Suppose that we have two prime numbers $p = 3, q = 5$. Then, their product is $N = pq = 3 * 5 = 15$. We choose the public exponent so that it is coprime to $\phi(N) = (p-1)(q-1) = 2 * 4 = 8$. In this case, we let the public exponent be $e = 7$. Now, we find the private exponent d as the inverse of the public exponent mod

$\phi(N)$. In this example, this happens to be 7, same as the public exponent. We can verifying this by performing the multiplication $de = 7 * 7 = 49 \equiv 1 \pmod 8$.

Now that we have the keys $e = 7$ and $d = 7$ in hand, we can proceed to encrypt a message. We choose the message to be 3. Encrypting the message with RSA, we get $3^7 = 2187 \equiv_{15} 12$ (according to Equation 3.3. So, the ciphertext is 12, and in order to decrypt this to obtain the original message, we compute $12^7 = 35831808 \equiv_{15} 3$ (using Equation 3.4).

3.4 Proof of correctness

Proving the correctness of RSA amounts to proving two statements: that we can always generate a private key from the pair of public keys, and that applying encryption on a message and decryption on the resulting ciphertext yields the original message. We will express these statements symbolically in Lemmas 3.1 and 3.2 below and prove them.

Lemma 3.1 (Existence of Private Key)**.**

For all pairs of integers e, N such that e is coprime to $\phi(N)$, there exists some integer d such that $de \equiv 1 \pmod{\phi(N)}$.

Proof. To prove this lemma, we will use Bezout's identity.

Theorem 3.2 (Bezout's Identity)**.**

For all integers a, b, there exists integers s, t such that $sa + tb = gcd(a, b)$ [24].

Seeing that $e, \phi(N)$ are coprimes, we know that $gcd(e, \phi(N)) = 1$. Therefore, according to Theorem 3.2, there exists integers s, t such that $se + t\phi(N) = 1$. This equation is equivalent to $se - 1 = -t\phi(N)$, which by Definition 3.2 implies that $\phi(N) \mid (se-1)$. Then, by applying Definition 3.3, we get $se \equiv 1 \pmod{\phi(N)}$. Therefore, there exists some integer $d = s$ such that $de \equiv 1 \pmod{\phi(N)}$. In conclusion, for any

pairs of integers e, N such that e is coprime to $\phi(N)$, there exists some integer d such that $de \equiv 1 \pmod{\phi(N)}$. □

Now, we set the public key as e, and the modulus as N. We know, from the description of the RSA algorithm in Section 3.2, that e is coprime to $\phi(N)$. So, by the lemma we just proved, we can always find some d such that $de \equiv 1 \pmod{\phi(N)}$. Now, if we let d be the public key, we have proven that we can always generate a private key. Next, we prove that the procedure of encrypting a message and then decrypting the resulting ciphertext gives us the original message.

Lemma 3.2 (Decryption Reverses Encryption)**.**

For a message m, $m^{ed} \equiv m \pmod{N}$.

Proof. To prove that the RSA algorithm encrypts and decrypts correctly, we will use Fermat's little theorem and its generalization, Euler's theorem.

Theorem 3.3 (Fermat's Little Theorem)**.**

Let p be any prime number and suppose that $p \nmid a$. Then, $a^{p-1} \equiv 1 \pmod{p}$ [16].

Theorem 3.4 (Euler's Theorem)**.**

Let n and a be integers such that $n > 0$ and a is coprime to n. Then, $a^{\phi(n)} \equiv 1 \pmod{n}$ [16].

We will consider two cases: where m is coprime to N and where m is not.

Case 1: m is coprime to N.

Because m is coprime to N, by Euler's theorem, $m^{\phi(N)} \equiv 1 \pmod{N}$. Then, we have $ed \equiv 1 \pmod{\phi(N)}$, and so by Definition 3.3, $\phi(N) \mid (ed - 1)$. By applying Definition 3.2 to the previous expression, we obtain $ed - 1 = k\phi(N)$ for some integer k, which is equivalent to $ed = k\phi(N) + 1$. Substituting this value of ed into m^{ed}, we get $m^{ed} = m^{1+k\phi(N)} \equiv mm^{k\phi(N)} = m(m^{\phi(N)})^k \equiv m * 1^k = m \pmod{N}$.

Case 2: m is not coprime to N.

The proof of this case is inspired by the proof in [29].

In order to prove the correctness of RSA in this case, we need to use the Chinese Remainder Theorem.

Theorem 3.5 (Chinese Remainder Theorem)**.**

Let u, v be coprime. Then the system of equations

$$x \equiv_u a$$

$$x \equiv_v b$$

has a unique solution x modulo uv [22].

One implication of Theorem 3.5 is that if $x \equiv_p a$ and $x \equiv_q a$, then $x \equiv_{pq} a$. This fact can be shown by noticing that a is a solution to the system of equations

$$x \equiv_u a$$

$$x \equiv_v a$$

Seeing that Theorem 3.5 says that all solutions of this system of equations is congruent modulo pq, we have $x \equiv a \pmod{pq}$.

Now, consider the condition that m is not coprime to $N = pq$. Seeing that $m < N$, we then have that p or q divides m. Without loss of generality, suppose that p divides m. Then, $m = px$ for some x. This gives us

$$m = px \equiv 0x = 0 \pmod{p}$$

By raising both sides of the equality $m = px$ to the power of ed, we get $m^{ed} = (px)^{ed}$. Using this, we then get

$$m^{ed} = (px)^{ed} = p^{ed}e^{ed} \equiv 0^{ed}e^{ed} = 0 \pmod{p}$$

Seeing as we have $m \equiv 0 \pmod{p}$ and $m^{ed} \equiv 0 \pmod{p}$, we can conclude that

$$m \equiv m^{ed} \pmod{p}$$

Now, we want to show that $m \equiv m^{ed} \pmod{q}$. First, by Theorem 3.3, we know that $m^{q-1} \equiv 1 \pmod{q}$ because q is a prime number and $q \nmid m$. We can raise both sides of this congruence to the power of $k(p-1)$ to obtain $m^{k(q-1)(p-1)} = m^{k\phi(n)} \equiv 1 \pmod{q}$. We have shown in the proof of case 1 that $ed - 1 = k\phi(n)$, and so this congruence is equivalent to $m^{ed-1} \equiv 1 \pmod{p}$. Multiplying both sides of this congruence by m, we then get

$$m^{ed} \equiv m \pmod{q}$$

Now that we have proven that $m^{ed} \equiv m \pmod{p}$ and $m^{ed} \equiv m \pmod{q}$, we apply Theorem 3.5 to conclude that $m^{ed} \equiv m \pmod{n}$. □

CHAPTER 4

Algorithm complexity analysis - execution time

Before we discuss the issues with the RSA algorithm presented above, it would be useful to introduce a precise language for discussing execution time of an algorithm. This language is provided by algorithm analysis, which defines precisely what it means for one algorithm to be better than another under some metrics.

When defining the constructs of algorithm analysis, it is convenient to have a simple algorithm to use as an example. For this purpose, we are going to use the algorithm insertion sort described in Listing 4.2, which we adapted from [8].

```
1 FUNCTION INSERTION_SORT(arr):
2 for j from 1 to arr.length:
3     key = arr[j]
4     i = j - 1
5     while i > 0 and arr[i] > key:
6         arr[i + 1] = a[i]
7         i = i - 1
8     arr[i + 1] = key
```

Listing 4.2: Pseudocode for insertion sort

The basic idea behind insertion sort is that if we have a sorted array, we can insert a new element into the array in such a way that the new array is sorted. So, insertion sort splits the original array into two subarrays, with the left one being sorted. Then, for each element in the (right) unsorted portion, insertion sort removes that element

from the unsorted subarray and inserts it into the sorted left subarray in such a way that it remains sorted. Once the algorithm goes through all the elements, we can be assured that the array has been sorted, for every element of the original array is in the sorted subarray.

4.1 Quantifying execution time

When analyzing an algorithm, we can represent its run time as a function of the number of steps that it executes. Here, a step is defined according to the model of computation we are working under. For the purpose of this section, we will make use of the RAM model introduced in [8]. In this model, we consider arithmetic, data, and control operations to be our primitive operations (i.e. they take a step each). While there are many problems with this model of computation (as mentioned by [31]), it provides a simple framework that is accurate enough for this section.

Using the RAM model, we analyze the insertion sort algorithm. First of all, we consider the while loop in lines 5 to 7 of Listing 4.2 (in the remaining portion of this analysis, any reference to a line number is assumed to be about this listing). In the worst case, the element at index `j` has to be moved `j` -1 times. Seeing that the for loop in line 2 is run for `j` from 1 to n, where n is the size of `arr`, the total number of movements is $0 + 1 + 2 + \cdots + n - 1 = \frac{n(n-1)}{2}$. Meanwhile, lines 3, 4, and 8 take 1 step each to run, and seeing as they are run when `j` goes from 1 to n (i.e. n times), they take $3n$ steps in total. So, insertion sort takes $\frac{n(n-1)}{2} + 3n$ steps to run in the worst case. If we expand this expression into a polynomial, we get the run time of insertion sort as $\frac{1}{2}n^2 + \frac{5}{2}n$. When n, the size of the array to be sorted, is very large, the dominant factor in this execution time is n^2, as demonstrated in the next paragraph. So, we can evaluate this algorithm as having complexity of order n^2, or $O(n^2)$ as defined below.

When we work with the run time of an algorithm, we care mostly for its behavior when the input size becomes large. At this scale, one term in the run time function tends to make up most of its value. For the sake of demonstration, consider the function $f(n) = n^2 + 1000n + 1000000$. For a large value of n, $n = 1000000$ for example, we have $f(n) = 1001001000000 = 10^{12} + 10^9 + 10^6$ and $n^2 = 1000000000000 = 10^{12}$. In this case, the value of terms other than n^2 amounts to a bit more than one percent of the value of n^2. As n gets larger, that percentage will become smaller. So, the amount of information lost when we ignore the smaller terms is negligible, especially when the input size to the algorithm becomes very large. Thus, we can safely ignore all but the greatest terms of a function in our analysis. In order to formalize this idea, we introduce asymptotic notation.

4.1.1 Asymptotic notation and algorithm complexity

When using asymptotic notation, we are not concerned with the exact values of a function. Instead, we only care about its behavior as its input grows past a certain point. More specifically, we look at the bounds of the function.

Definition 4.1 Bounded function: *A function $f : A \to B$, where $A, B \subseteq \mathbb{R}$, is said to be **bounded** if for all $x \in \mathbb{A}$, $| f(x) | \leq L$ for some $L \in \mathbb{R}$. Furthermore, f is said to be **bounded above** if $f(x) \leq L$ and **bounded below** if $f(x) \geq L$.*

Intuitively, a bounded function is a function whose value can never reach above or below certain threshold. Some examples of this kind of functions include the sin and cos functions, which are bounded because $|\sin(x)| \leq 1$ and $|\cos(x)| \leq 1$ for all $x \in R$. If we are to visualize a bounded function, as done in Figure 4.1 with the sin function, we would see that the function is **bounded** by two lines, one above it and one below it. The upper line is called the upper bound, while the lower one is called the lower bound.

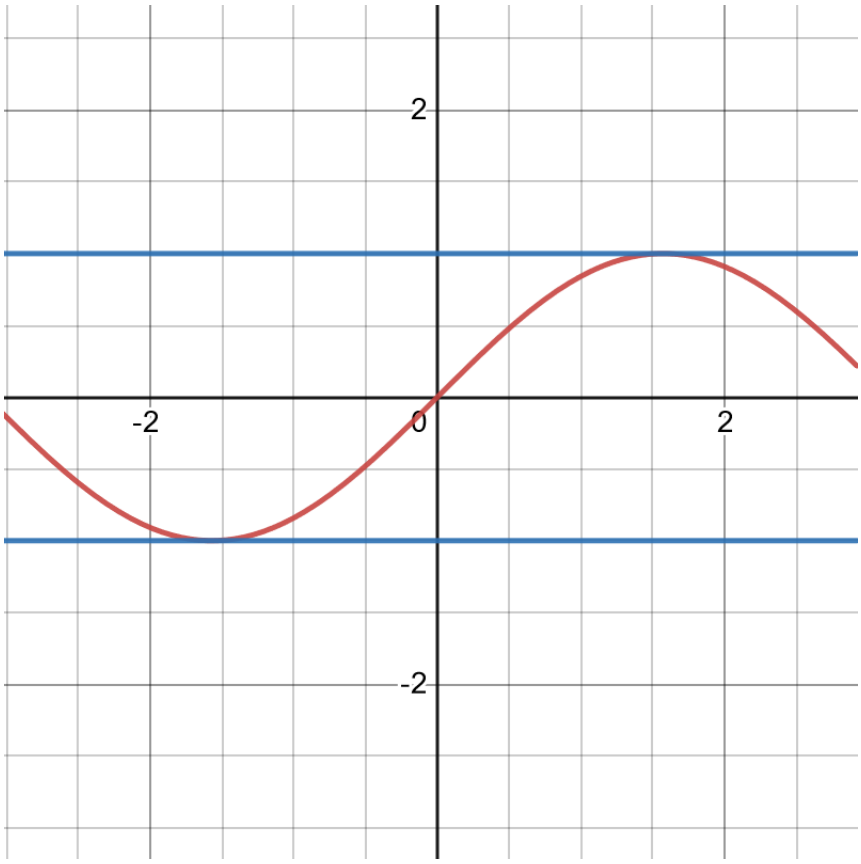

Figure 4.1: The sin function is bounded by the values 1 and −1. (Made in Desmos)

Looking at Figure 4.1, we can notice that the sin function looks like it is bounded by two functions, in this case the constant function $u(x) = 1$ and $l(x) = -1$, the two blue lines in the figure. In fact, this observation is generalized to all bounded functions. One might then ask if it is possible to generalize this further, with $u(x)$ and $l(x)$ being any functions instead of constant functions. The θ notation is one attempt to do just that.

Definition 4.2 *For a function $f(n)$, we have*

$$\theta(f(n)) = \{g(n) : \exists c_1, c_2, n_0 \text{ such that } 0 \leq c_1 f(n) \leq g(n) \leq c_2 f(n) \text{ for all } n \geq n_0\} \text{ [8]}$$

There is one major difference between the notion of being bounded in Definitions 4.1 and 4.2. In Definition 4.1, the function must be bounded by the value on its whole domain. On the other hand, in Definition 4.2, a function needs only be bounded above some value in the domain. This difference reflects our main usage for the θ-notation, comparing the difference between two algorithms as their input size gets very large.

One problem with θ-notation is that it can be too restrictive. In order to prove that a function $f(x)$ is $\theta(g(x))$, we have to prove that $g(x)$ bounds $f(x)$ above and below. A lot of the time in algorithm analysis, however, we only care about the upper bound of a function for it indicates the limit of what an algorithm can do. For this purpose, we define the O notation.

Definition 4.3 *For a function $f(n)$, we have*

$$O(f(n)) = \{g(n) : \exists c, n_0 \text{ such that } 0 \leq g(n) \leq c f(n) \text{ for all } n \geq n_0\} \text{ [8]}$$

We can see that the O-notation is very similar to θ-notation, which makes sense seeing as O-notation is simply θ-notation without a lower bound. There is,

however, one thing that makes O-notation distinct from θ-notation: O-notation does not provide an asymptotically tight bound. By this, we mean that we can have $f(x) = O(g(x))$ and $f(x) = O(k(x))$ even if $O(g(x)) \neq O(k(x))$.

For an example of this phenomenon, consider the set of functions $f(x) = x + 1$, $g(x) = x$, and $k(x) = x^2$. We can see that for $x \geq 1$, $f(x) \leq 2g(x)$, and so $f(x) = O(g(x))$. At the same time, $f(x) \leq 2k(x)$ for all $x \geq 1$, which means that $f(x) = O(k(x))$. Now, consider the function $a(x) = x^2 + 1$. We can see that $a(x) \leq 2k(x)$ for all $x \geq 1$. Meanwhile, for all c, there is some point n where if $x \geq n$, $a(x) > cg(x)$. So, $a(x) = O(k(x))$ but $a(x) \neq O(g(x))$, and thus $O(k(x)) \neq O(g(x))$.

In this project, when we say that an algorithm runs in $O(f(n))$ time or that it has complexity $f(n)$, we assume that $f(n)$ is a tight bound for the run time of the algorithm. Otherwise, the O-notation presents a loose bound. The reason we want to use O-notation, in the first place, is to be able to describe the maximum potential of an algorithm (the upper bound of its execution time) without needing to describe its performance guarantee (the lower bound of its execution time) as θ-notation does. If we allow for asymptotically loose bound when talking about the O-notation with respect to an algorithm, our discussion can potentially overestimate the run time of that algorithm, which is undesirable.

4.1.2 Ordering of asymptotic notation

In this section, we assume that the O-notation presents a loose bound. Consider Figure 4.2, which shows the graphs of most common algorithm complexities. We can see that, for instance, n is less than 2^n for all n. So, by Definition 4.3, we can say that $n = O(2^n)$. In other words, n is bounded above by 2^n. Using the same process, we can also see in the graph that n is bounded above by n^2, which is in turn bounded above by 2^n.

Using these observations, we can construct an ordering for the O-notation. For

notational convenience, we say that for two functions $f(n), g(n)$, $O(f(n)) < O(g(n))$ if $f(n)$ is bounded above by $g(n)$ but not vice versa. Using this and Figure 4.2, we have the following ordering of the O-notation.

$$O(\log_2 n) < O(n) < O(n \log_2 n) < O(n^2) < O(2^n) < O(n!) \tag{4.1}$$

The most important takeaway from Equation (4.1) is that there are different classes of execution time, and that there is a strict ordering among these classes. The most important classes for this project are constant, polynomial, and exponential time. These correspond to $O(1), O(n^k), O(2^n)$ respectively. Furthermore, we divide polynomial time into multiple subclasses, with the most important ones being linear, quadratic, and cubic time, corresponding to $O(n), O(n^2)$, and $O(n^3)$. In general, we aim to create linear time algorithms, seeing as they are the most performant without needing to require special properties of the input. Failing that, polynomial time algorithms are acceptable, but exponential algorithms should be avoided as much as possible.

4.1.3 Asymptotic notation with number-theoretic algorithms

In this section, we have established how asymptotic notation is used to measure and compare the run time of different algorithms. However, we have only considered algorithms that work on arrays, which have an intuitive notion of size. Indeed, we can extend the above discussion to most data structures seeing that we can also easily assign them a size, namely the number of objects contained in them. However, we cannot apply this concept of size to number-theoretic algorithms, seeing as the input to such algorithms are numbers instead of containers.

Before we can discuss what size of input means when it comes to number-theoretic algorithms, we need a notion of what a number-theoretic algorithm is.

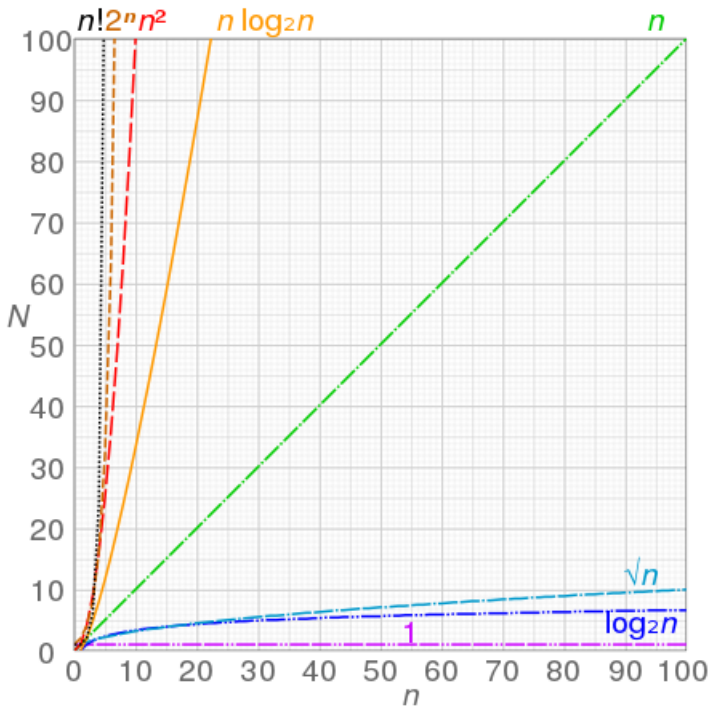

Figure 4.2: Graphs of some functions whose O-notation is used in algorithm analysis. Here, n is the size of the input and N is the number of steps executed [7].

For the purpose of this thesis, a number-theoretic algorithm is an algorithm that takes a number as input and gives out some information about that number using constructs of number theory. Under this definition, multiplication, division, and prime factorization algorithms count as number-theoretic algorithm. On the other hand, algorithms such as those used to find the n-th Fibonacci numbers are not number-theoretic algorithms, seeing that they do not output information about n.

Now that we have a definition of number-theoretic algorithms, we can proceed to talk about the computational complexity of such algorithms. Before that, however, we still need to think about what input size would mean for them. To do this, we set up an analogy with the concept of size for data structures. A way we can think of size for a data structure is as the number of discreet memory units needed to represent that data structure. Applying this notion to numbers, we obtain the notion of size for a number as the count of digits required to represent that number. So, for example, the size of 2412414 is 7 because it is represented by 7 digits. In general, the

size of a number can be obtained by

$$size(n) = \lfloor \log_{10}(n) \rfloor + 1$$

There is more than one way to represent a number, however. In the last paragraph, we are representing a number in base 10. The choice of base here is quite arbitrary, however. We could represent a number in base 2, and that could have work just as well for defining the size of that number. In fact, there is a general formula to define the size of a number in any base x

$$size(n) = \lfloor \log_x(n) \rfloor + 1 \tag{4.2}$$

We can prove this fact with a modified version of the proof given in [27].

There is a problem with this way of defining size, however. The size of a number differs between any two representation. For an example, we will use the number $2,412,414$ again. As mentioned above, this number has size 7 when represented in base 10. However, when it is represented in base 2, it has size $\lfloor \log_2(2,412,414) \rfloor + 1 = 22$. This can be a problem, for it is possible that different representation of a number can lead to different analysis of an algorithm. However, due to the way O-notation and logarithm work, this is not an issue.

Theorem 4.1.

For any real numbers a, b, $O(\log_a(n)) = O(\log_b(n))$.

Proof. Let a, b be arbitrary real numbers. By applying the change of base formula for logarithm, we have

$$\log_a(n) = \frac{\log_b(n)}{\log_b(a)}$$

Because a, b are constants, $\log_b(a)$ is also a constant. Seeing as the complexity order

of a constant is 1, we have

$$O(\log_a(n)) = O\left(\frac{\log_b(n)}{\log_b(a)}\right) = O\left(\frac{\log_b(n)}{1}\right) = O(\log_b(n))$$

□

The theorem we just proved shows that the base of a logarithm does not matter when using O-notation. As a result, when working with logarithms in O-notation, we omit the base.

Now, consider a number-theoretical algorithm with its input being a number n. We say that this algorithm runs in polynomial time if its run time is polynomial in its size or, in other words, $O(size(n)^k) = O(\log^k(n))$ for some number k. Analogously, the algorithm in question runs in exponential time if its run time is exponential in its size or $O(a^{size(n)}) = O(a^{\log(n)})$ for some real number a. One interesting thing that we note here is that a number-theoretical algorithm is exponential if its run time is polynomial in n. This comes down to the observation that $n = e^{\log_e(n)}$. So, if the run time is polynomial in n, we have $O(n^k) = O((e^{(\log_e(n))})^k) = O(e^{k\log_e(n)}) = O(e^k e^{\log_e(n)}) = O(e^{\log(n)})$, which is exponential in the size of n.

On a final note, we discuss the arithmetic of large numbers. In the RAM model, which we have been working in, arithmetic operations are considered primitive, and so their run time is constant. However, when the scale of the numbers we work with gets very large, such as with the RSA algorithm, this assumption fails. Instead, we have to use non-constant-time algorithms to compute basic arithmetic operations.

For this project, we only concern ourselves with algorithms for subtraction, multiplication, and modulo operation, seeing that these are the operations used in the RSA algorithm. In [20], Donald Knuth created a program for subtraction that takes $12N+3$ steps, where N is the number of digits in each operand. Using O-notation, we

then have the complexity of subtraction as $)(12N+3) = O(N) = O(\log(n))$, where n is one of the operands in the subtraction. In [8], the basic algorithm for multiplication is said to have complexity $O(\log^2(n))$. While there are asymptotically faster algorithms for these two operations, their time complexity is not as straightforward. For the sake of keeping our analyses simple, we will only consider the basic algorithms. As for the modulo operation, we can use Lemma 3.1 to obtain the remainder of a division using one subtraction after we have obtained the integer part of the division. From [8], we also learn that the complexity of division is $O(\log^2(n))$. Combining these two facts, we have the complexity of the modulo operation as $O(\log^2(n)) - O(\log(n)) = O(\log^2(n))$.

CHAPTER 5

Problems with basic RSA

In this chapter, we present two main security concerns with the version of RSA presented in Chapter 3. In addition, we include a complexity analysis of the RSA algorithm and a analysis of its run time in the context of its usual use case.

5.1 Prime selection

Conceptually, the simplest attack on the RSA is to factor the public key N into its prime constituents p, q. Once an attacker obtains these two numbers, they can easily calculate the Euler's totient of the modulus N. Seeing as the public exponent e is available to everyone, the attacker can then easily calculate the private key d through the process described in Section 5.3.1 and use it to decrypt all messages sent by an individual. Thus, it is imperative to chose p, q such that it is difficult to factor N.

Currently, the integer factorization algorithm with the best time complexity is the general number field sieve, which have a subexponential time complexity (i.e. its runtime grows faster than all polynomials, but slower than all exponential functions). Even with this algorithm, however, it takes a 2.1 GHz CPU running for an equivalent of 2700 years to factor a 829-bit (or 250 digits) number [38]. As we can see, it takes an incredibly large amount of time to factor large numbers.

While generally, it may take a very long time to factor a large integer, there

are cases where a number might have some special structure that an algorithm can exploit to factor it very quickly. We call factoring algorithms of this kind special-purpose algorithms.

A problem with the RSA algorithm in Section 3.2 is that we do not place any constraint on the kind of prime numbers that the algorithm can generate. Thus, the primes used in the encryption and decryption process might be of a form where their product possesses the special property required for some special-purpose factoring algorithm. Below, we list a number of special-purpose integer factorization algorithms and the kind of numbers they work efficiently on. In order to simplify the algorithms, we will assume that the number to be factored has only two odd prime factors.

5.1.1 Trial division

The simplest algorithm for factoring integers is trial division, which is shown in Listing 5.3. The basic idea behind this algorithm is that the factors of a number n must lie somewhere between 1 and n. Thus, if for each of these values, we check if it is a factor of n, we are guaranteed to find all factors of n. It should be noted that in Listing 5.3, only numbers up to $\sqrt{n}$ are checked. This optimization is possible due to the fact that at least one prime factor of a number n is less than $\sqrt{n}$.

Theorem 5.1.

For all $n \in N$, there exists at least one prime factor p of n such that $p \leq \sqrt{n}$.

Proof. Let n be an arbitrary natural number with prime factors p, q. For the sake of contradiction, suppose that $p, q > \sqrt{n}$. In that case, we have that $pq > n$, which is a contradiction because $pq = n$ due to p, q being prime factors of n. Thus, it is the case that $p \leq \sqrt{n}$ or $q \leq \sqrt{n}$. □

From Theorem 5.1, we know that we only need to check numbers up to $\sqrt{n}$ to

find a prime factor of n. Furthermore, because n only has two prime factors, we only need to find the smaller one to completely factor n. Thus, it suffices to check all numbers up to $\sqrt{n}$.

```
1 FUNCTION TRIAL_DIVISION(n):
2 for i from 1 to sqrt(n):
3     if remainder of n / i is 0:
4 return i, n / i
```

Listing 5.3: Pseudocode for trial division algorithm.

In the worst case scenario, trial division would need to check all numbers from 1 to $\sqrt{n}$. Thus, there are $\sqrt{n}$ checks to be done, and so the function has a time complexity of $O(\sqrt{n}) = O(n^{1/2})$. So, trial division is exponential in the size of n, which means it is very slow for large n. However, as mentioned above, if n has a small prime factor, the algorithm only needs to check numbers up to that prime factor. If that factor is small enough (e.g. it has less than 7 digits), it can be found instantly. Thus, if one of the primes we generate for RSA is too small, the private key can be easily obtained using trial division.

5.1.2 Fermat's algorithm

Another simple algorithm for factoring integers is Fermat's factorization method, which is named after the mathematician Pierre de Fermat. The core of this algorithm is based on the algebraic identity

$$a^2 - b^2 = (a + b)(a - b) \tag{5.1}$$

Based on the above equation, if we are able to write an integer n as a difference of two squares, we have already obtained its factors. Indeed, for odd n, it is always possible to obtain such a representation.

Lemma 5.1.

If an odd number n has a factorization $n = pq$, then we have $n = \left(\frac{p+q}{2}\right)^2 - \left(\frac{p-q}{2}\right)^2$ [9].

Proof. Let n be an odd number with factorization $n = pq$. Then, we have

$$\begin{aligned}
n &= pq \\
&= \frac{4pq}{4} \\
&= \frac{4pq + p^2 + q^2 - p^2 - q^2}{4} \\
&= \frac{p^2 + 2pq + q^2 - (p^2 - 2pq + q^2)}{4} \\
&= \frac{(p+q)^2 - (p-q)^2}{4} \qquad ((a \pm b)^2 = a^2 \pm 2ab + b^2) \\
&= \frac{(p+q)^2}{4} - \frac{(p-q)^2}{4} \\
&= \frac{(p+q)^2}{2^2} - \frac{(p-q)^2}{2^2} \\
&= \left(\frac{(p+q)}{2}\right)^2 - \left(\frac{(p-q)}{2}\right)^2
\end{aligned}$$

□

Fermat's factorization method, shown in Listing 5.4 finds a number a such that $a^2 - n$ is a square number. Once such an a has been chosen, if we set $b^2 = a^2 - n$, we can get $n = a^2 - b^2$, which is a difference of squares. Thus, we have found a factorization of n. As seen in Listing 5.4, Fermat's method finds a by trying all possible positive integers. Here, the algorithm starts with $\lceil \sqrt{n} \rceil$ because it is the smallest number for which $a^2 > n$, which is the requirement for $a^2 - n$ to be positive, and thus possibly a square number [9].

```
1 FUNCTION FERMAT_METHOD(n):
2 a = ⌈√n⌉
3 b² = a * a − n
4 while b² is not a square number
5     a = a + 1
6     b² = a * a − n
7 b = √b²
8 return a − b, a + b
```

Listing 5.4: Pseudocode for Fermat's method.

For an example, we will factor the number $n = 217$. We have $\lceil \sqrt{n} \rceil = 15$, and so we will start checking from there. Setting $a = 15$, we get $b^2 = 8$, which is 2^3 and thus not a perfect square. Then, if we try $a = 16$, we get $b^2 = 39$, which is again not a square number. In fact, $b^2 = a^2 - n$ is not a square number until $a = 19$, where its value is $144 = 12^2$ as shown in Table 5.1. So, with a, b having been found, we can find the factors of n, which are $a + b = 19 + 12 = 31$ and $a - b = 19 - 12 = 7$.

a	15	16	17	18	19
b2	8	39	72	107	144
b	2.828	6.245	8.485	10.344	12

Table 5.1: Calculations made by Fermat's method with the given example.

In the worst possible scenario, i.e. when n is prime, we have to test all possible a from $\lceil \sqrt{n} \rceil$ to N. In this case, Fermat's method is the same as a trial division starts from the halfway point instead of the beginning. So, its run time is exponential in the size of n, which is very bad.

We can notice, however, that if two factors of n are close together, we know that $a + b$ and $a - b$ are close together. Seeing that the size of the distance between these

two numbers, $2b$, grows larger as a increases, we can conclude that the closer the two factors are, the smaller a is. Seeing that a small a indicates that the algorithm did not run for a lot of steps, we can say that Fermat's algorithm runs faster the closer the two factors of n are to each other.

5.1.3 Pollard's $p-1$ algorithm

Let n be the input integer. The first step in Pollard's $p-1$ algorithm is to pick a number B, called the smoothness bound. Then, we calculate M as the product of all $q^{\lfloor \log_q(B) \rfloor}$, where q are primes less than B. After that, we randomly pick a coprime a of n, and calculate $g = \gcd(a^M - 1, n)$. Finally, if $1 < g < n$, we return g, otherwise an error is raised. The pseudocode for this algorithm is shown in Listing 5.5.

```
1  POLLARD-ALGORITHM(n)
2  B = RANDOM-NUMBER()
3  primesList = FIND-ALL-PRIME-LESS-THAN(B)
4  M = 1
5  for i = 1 to primesList.length
6      q = primesList[i]
7      M = M * q^⌊log_q(B)⌋
8  a = PICK-COPRIME(n)
9  g = GCD(a^M - 1, n)
10 if 1 < g < n
11     return g
12 else
13     error "Algorithm failed"
```

Listing 5.5: Pseudocode for Pollard's $p-1$ algorithm.

The main idea of this algorithm can be found at [37]. There are, however, some unclear steps in this resource, which we will fill in the following paragraphs. In

Pollard's $p-1$ algorithm, a was chosen to be coprime to n, and is consequently also coprime to all factors of n. Let p be a factor of n. Then, by Fermat's Little Theorem (Theorem 3.3), we have $a^{p-1} \equiv 1 \pmod{p}$, which is equivalent to $a^{k(p-1)} \equiv 1 \pmod{p}$ for some integer k. Then, we set $M = k(p-1)$. Now, a consequence of $a^M \equiv 1 \pmod{p}$ is that $p \mid a^M - 1$. This means that there is some integer c such that $a^M - 1 = pc$, per Definition 3.2.

Consider the value $g = \gcd(a^M - 1, n)$. Seeing as p is a multiple of n, there is some q such that $n = pq$. So, we have $g = \gcd(pc, pq)$. A property of the gcd function allows us to factor the common factor of its arguments, and so $g = p\gcd(c, q)$. Here, seeing as g is a product of two factors of n[1], if $1 < g < n$, then g must be a factor of n.

At this point, there is still one issue that needs to be addressed. In the above paragraph, M is set as a multiple of $p-1$, which is impossible to determine without knowledge of p, the number that is being searched for. This is where B is used. The algorithm assumes that $p-1$ is B-powersmooth.

Definition 5.1 *An integer n is called k-**powersmooth** if, when n is written as a product of prime powers, all of those prime powers are at most k.*

Now, instead of calculating M as a multiple of $p-1$, the algorithm calculates M as the smallest number that is a multiple of all B-powersmooth numbers, which results in the formula for M used in the algorithm. However, this method is not guaranteed to work. If there is no factor p of n such that $p-1$ is B-powersmooth or $p-1$ is B-powersmooth for all p, the algorithm won't be able to give a nontrivial factor, seeing that the former situation results in a factor of 1, while the latter results in a factor of n.

If it is desired, however, it is possible to modify the algorithm so that a factor is always given. At the final step, instead of raising an error, consider the value of g.

[1] $\gcd(c, q)$ gives us a factor of q, which is also a factor of n.

If $g = 1$, B should be increased. On the other hand, if $g = n$, B should be decreased. This way, B should eventually reach a value where $1 < g < n$.

The time complexity of Pollard's $p - 1$ algorithm is $O(B \log^2 n)$ [23]. The idea of the derivation is that there are three main steps in the algorithm: finding all primes in $[1, B]$, calculating $a^M - 1$, and computing $\gcd(a^M - 1, n)$. In most cases, where B is not small, the second step would dominate the computation, and thus its time complexity is the algorithm's complexity. Here, it should be noted that the value of M is not calculated directly. Instead, $a^{q^{\lfloor \log_q(B) \rfloor}}$ is calculated for each q, and then these values are combined to form a^M.

Having said that, we can see that if we choose B to be relatively small, say, 10^7, the algorithm runs very quickly. However, it only succeeds if $n - 1$ is B-powersmooth. While this might not always be the case, the fact is that there is a condition on the factor of n under which Pollard's $p - 1$ algorithm is efficient.

5.2 Chosen texts attack

In this section, we present a chosen plaintext attack and a chosen ciphertext attack on the RSA algorithm.

5.2.1 Chosen plaintext attack

Suppose that we are an eavesdropper on a conversation and we intercepted a ciphertext c. Using some external information, we know what the original message might be. For example, the message is the answer to the question "What is you favorite day of the week?", of which there are seven possibilities. Now, by using the publicly-available public exponent of the sender, we encrypt all the possible plaintexts we prepared. If one of those plaintexts encrypt to the sent ciphertext, we know that that plaintext is the original message.

This attack relies on the deterministic nature of modular exponentiation. Given a message and an RSA key, the RSA algorithm will only send that message to one ciphertext. Thus, if an eavesdropper can limit the message space to a reasonable size, they can encrypt everything in that limited subspace and obtain the original message through finding which message encrypts to the intercepted ciphertext.

Now, it might be said that the problem of limiting the message space to a reasonable size is not always possible to solve. While this might be true, when it is possible, this is a very powerful attack, especially if the public exponent is small. Moreover, it is easiest to limit the message space when the messages have a common pattern. Sometime, the cause for this can be very silly. Simon Singh describes one such situation, where the Germans in WW2 use two keys to encrypt each message, one key to encrypt the message, and the other to encrypt the message key. The message key is a random sequence of three characters and changes between different messages, while the key used to encrypt the message keys remain the same for the entire day. Furthermore, the encrypted message key is attached to the top of the message it was used to encrypt. The problem came when overworked personnel sometimes cannot bring themselves to care enough to choose complex message keys, and instead use things such as three consecutive letters [30].

Now, suppose that in the situation above the Germans were using RSA combined with a symmetric cipher, with RSA being used to encrypt the key for the symmetric cipher. Normally, a brute force attack would entails checking the whole message space, the space of all sequence of three letters. Meanwhile, in the scenario where the operator uses three consecutive letters as the key for the symmetric cipher, we know that this key is a sequence of three consecutive letters. So, we can limit our search to the space of sequences of three consecutive letters.

5.2.2 Chosen ciphertext attack

In the previous subsection, we have covered the chosen plaintext attack, a method to retrieve a message from a ciphertext without access to the private exponent. However, that attack only works on cipher for which we have a good guess of what the original message might be. Furthermore, because the attack does not give us any information about the private exponent, we can only decrypt ciphertexts on a case-by-case basis. In this subsection, we introduce a more powerful attack, presented in [10], that allows us to retrieve the original message from a ciphertext without needing information about the message. While this attack still can only decrypt individual ciphertexts instead of giving us access to the private key, it presents an improvement over the chosen plaintext attack in that it works on any ciphertext.

The core of the chosen ciphertext attack relies on the fact that modular congruence is compatible with multiplication, i.e. we can multiply both side of a modular congruence to get a new modular congruence with the same modulus. Thanks to this property, we can easily transform a valid ciphertext into another valid ciphertext. We call this ease of transformation malleability. The attack proceeds as follow. Suppose we have a ciphertext c where

$$c \equiv m^e \pmod{N}$$

We choose an integer x and multiply both side of the above congruence by x^e to obtain

$$cx^e = c' \equiv m^e x^e \pmod{N}$$

This is always possible because the public exponent e is known by everyone.

Then, if we can somehow make the sender decrypts c', we can get

$$(c')^d \equiv (m^e x^e)^d = m^e d x^e d \equiv mx \pmod{N}$$

At this point, in order to obtain the original message, we simply multiply both side by the multiplicative inverse of x, which we can efficiently compute as shown in sections 5.3.1 and 5.3.2 below.

The method outlined above works for all ciphertexts. The roadblock for this attack, however, lies in the fact that it requires the sender of the message to decrypt a random ciphertext and send the corresponding plaintext back. While the first part is not too hard to do, it is the second part that poses a problem. Usually, decryption is the end of the communication process. So, unless a request is made to have the result of the decryption, most people will not send the decryption result back, especially since the process we used causes decryption to usually give out nonsensical plaintext. If the result of the decryption process is saved, however, even if the save location is the trash, we can possibly retrieve it and get back the original message.

5.3 Run time of basic RSA

In this section, we analyze the asymptotic run time of the RSA algorithm presented in Chapter 3. To do this, we provide a simple implementation of the algorithm. After obtaining the time complexity of RSA, we then proceed to discuss what the result means in the context of the actual numbers used in the RSA cryptosystem.

5.3.1 Implementation

The first step in the RSA algorithm is to generate two primes p, q. When p, q are large enough, the methods mentioned in Section 3.2 are infeasible. Instead, we employ the approach of generating an odd number and testing if it is prime. If it is not, we check all successive odd numbers until a prime number is found. The algorithm used in this implementation to test whether a number is prime is called the Miller-Rabin algorithm. The pseudocode for this approach can be found in Listing 5.6.

By the prime number theorem, the number of prime numbers less than a number n, denoted $\pi(n)$, is approximately $\frac{n}{\ln(n)}$ for very large n [36]. We denote this as $\pi(n) \approx \frac{n}{\ln(n)}$. Using this fact, Apostol proved that the nth prime p_n is approximately $n \log(n)$ [2]. Building from this, Marco Cantarini proved that the average gap between two primes near a number n is approximately $\log(n)$ [4]. So, in order to find a prime number with the method outlined in Listing 5.6, we only need to check around $\log(n)$ numbers on average, which makes this method fast.

```
1  GLOBAL_VAR p, q, e, d
2
3  FUNCTION GENERATE_PRIME_NUMBER():
4  temp = GENERATE_RANDOM_NUMBER()
5  if temp is equal to 2:
6      return temp
7  if temp mod 2 == 0:
8      temp = temp / 2
9  while temp is not prime:
10     temp = temp + 2
11 return temp
```

Listing 5.6: Pseudocode for prime generation algorithm.

The next step to consider in the algorithm is the method for calculating the public exponent e. The simplest way to do this, as seen in Listing 5.7, is to generate random odd numbers [2] and check if they are coprime to $\phi(N)$. This would entail using Euclid's algorithm to calculate the greatest common divisor between the candidate number and $\phi(N)$ until we get a value of 1. Unlike the prime number generation step, however, there is no upper bound on the numbers that must be tested with this method, and so its execution time might vary a lot between iterations.

```
1 // n = φ(N)
2 FUNCTION MAKE_PUBLIC_KEY(n):
3 e = GENERATE_RANDOM_NUMBER()
4 if e mod 2 == 0:
5     e = e / 2
6 while GCD(e, n) is not equal to 1:
7     e = e + 2
8 return e
```

Listing 5.7: Pseudocode for calculation of public key in RSA algorithm.

With the public exponent and the modulus in hand, we can now calculate the private exponent d. As noted in Section 3.2, d is the solution to Equation (3.2). In order to solve this equation, we transform it into an instance of Bezout's identity. First, by using Definition 3.3, we get $\phi(N) \mid (de - 1)$, which is equivalent to $de - 1 = -t\phi(N)$ for some integer t per Definition 3.2. By moving the term around in this equation, we get $de + t\phi(N) = 1 = \gcd(e, \phi(N))$, which is an instance of Bezout's identity. In order to calculate d, we can use the Extended Euclidean Algorithm, as described in [8]. We include the pseudocode provided below in order to make analyses of this algorithm easier. One important thing that we assume in Listing

[2] Because p and q are primes and so necessarily odd, $\phi(N) = (p-1)(q-1)$ will always be even. Thus, an even e cannot be coprime to $\phi(N)$.

5.8 is that a is greater than b. In our specific use case, a would then be $\phi(n)$ and b would be e.

```
1 EXTENDED-EUCLID(a, b)
2 if b == 0
3     return (a, 1, 0)
4 else
5     (d', x', y') = EXTENDED-EUCLID(b, a mod b)
6     (d, x, y) = (d', y', x' - ⌊a/b⌋y')
7     return (d, x, y)
```

Listing 5.8: Pseudocode for extended Euclidean algorithm [8].

The final two steps in the RSA algorithm is the actual encryption and decryption. Both of those operations boil down to a problem of modular exponentiation. This problem is solved using the repeated squaring algorithm, which is shown in Listing 5.9. The basic observation behind the repeated squaring algorithm is that every positive number can be written as a sum of distinct powers of 2.

Theorem 5.2.

For all positive integer n, n can be written as a sum of distinct powers of 2.

Proof. The proof below was adapted from [6].

For the case where $n = 1$, we have $n = 2^0$. Thus, n can be written as a sum of distinct powers of 2.

Consider some arbitrary positive integer n. Now, for the sake of strong induction, suppose that for all $k \leq n \in \mathbb{Z}$, k can be written as a sum of distinct powers of 2. We will consider two cases, where $n + 1$ is odd and where $n + 1$ is even.

In the case where $n + 1$ is odd, we know that n can be written as a sum of distinct powers of 2. Furthermore, because n is even, that sum cannot include 2^0 because $2^0 = 1$ is an odd number, and it being in the sum would result in n being odd. Now,

we know that $n+1 = n+2^0$. Seeing that n is a sum of distinct powers of 2 excluding 2^0, $n+1$ is a sum of distinct powers of 2.

In the case that $n+1$ is even, we know that $\frac{n+1}{2}$ is an integer less than or equal to n. Thus, $\frac{n+1}{2}$ can be written as a sum of distinct powers of 2 by the induction hypothesis. If we are to multiply this sum by 2, which would yield $n+1$, every power of 2 in the sum would have its exponent increase by 1. So, $n+1$ can be written as a sum of powers of 2. Now, because addition by 1 is injective ($a+1 = b+1$ implies $a = b$), and the exponents of the powers of 2 in $\frac{n+1}{2}$ are distinct, the exponents of the powers of 2 in $n+1$ is also distinct. Thus, $n+1$ can be written as a sum of distinct powers of 2.

In conclusion, by strong induction, all positive integer n can be written as a sum of distinct powers of 2. □

Suppose we want to calculate $a^k \mod n$. There are two steps in the repeated squaring method. The first one is to calculate $a^{2^i} \mod n$ where $0 \leq i < \lfloor \log_2(n) \rfloor + 1$. In simpler terms, we need to calculate the powers of a where the exponent is a power of 2, up to $a^{2^{\lfloor \log_2(n) \rfloor}} \mod x$. In order to do this, we can use the recurrence relation

$$a^{2^0} \equiv_x a$$
$$a^{2^i} \equiv_x (a^{2^{i-1}}) * (a^{2^{i-1}})$$

Once this step is done, we proceed to write k as a sum of distinct powers of 2, $k = \sum_{p_i} 2^{p_i}$. Theorem 5.2 lets us know that this is possible for all k. Then, we can compute a^k as $a^{2^{p_1}} \cdots a^{2^{p_u}}$. Now, we can see that the maximum value of 2^{p_i} is $2^{\lfloor \log_2(n) \rfloor}$. Because $\lfloor \log_2(n) \rfloor + 1$ is the number of bits required to represent n, we have that $2^{p_i} \leq k$ for all i. As a result, we have already calculated $a^{2^{p_i}} \mod n$ in the first step. Thus, we only have to multiply together the results of the calculations in the first step. This is justified because if $x \equiv_n y$, then $xb \equiv_n yb$.

Having said that, the implementation of this algorithm is a bit different from the mathematics behind it. The main thing we have to consider is that writing the exponent n as a sum of powers of 2 is not a trivial task. It would, in fact, require calculating all powers of 2 less than or equal to n, which is not a simple task. What is simple, however, is to detect whether the kth bit of n is a 1. This amounts to bit shifting n by $k-1$ positions and performs a logical AND operation between n and 1. Both of these operations can be done in, at worst, linear time, which is much better than the quadratic time multiplications required for the other method.

```
1  FUNCTION REPEATED_SQUARING(n, exponent, modulus):
2  pows_a = array of size ⌊log2(n)⌋
3  pows_a[0] = a mod modulus
4  for i from 1 to ⌊log2(n)⌋ inclusive
5      pows_a[i] = pows_a[i - 1] * pows_a[i - 1] mod n
6  a^n = 0
7  for k from 0 to ⌊log2(n)⌋ inclusive
8      if (exponent >> k) & 1 is 1
9          a^n = a^n * pows_a[k] mod n
10 return a^n
```

Listing 5.9: Pseudocode for modular exponentiation using repeated squaring.

5.3.2 Complexity of RSA

In order to analyze the time complexity of the RSA algorithm, we will separate it into three steps: primes generation, modular inverse, and modular exponentiation.

Prime generation has complexity $O(\log^4(p))$, where p is one of the primes generated. As mentioned in 5.3.1, we generate a prime number by checking consecutive odd numbers using the Miller-Rabin algorithm. The Miller-Rabin algorithm is a probabilistic algorithm to check whether a number is prime. Due

to it being a probabilistic algorithm, Miller-Rabin cannot determine with certainty that a number is prime. Indeed, it has a probability of at most 1/4 of declaring a composite number as a prime [19]. We can, however, reduce this probability by running the test multiple times. In general, if we run Miller-Rabin k times on a number, the probability that that number is falsely said to be a prime is at most 2^{-2k}. So, if we run Miller-Rabin enough time on a number, we can be reasonably sure that the number is prime.

For the basic RSA algorithm presented in 3.2, we will assume that a probability of 2^{-100} of Miller-Rabin declaring a composite number to be prime, which corresponds to 50 tests, is acceptable. Now, suppose that we are trying to generate the prime p. We know that the complexity of one Miller-Rabin test using classic multiplication is $O(\log^3(p))$ [19]. Seeing that we are doing 50 runs, we have a final run time of $50O(\log^3(p))$, which reduces to $O(50)O(\log^3(p)) = O(50\log^3(p)) = O(\log^3(p))$. Thus, the cost of testing one number for primality is $O(\log^3(p))$. Now, we have established in 5.3.1 that we only need to test $\ln(p)$ numbers on average to get a prime number. Combining this with the complexity of Miller-Rabin and the fact that we need to generate two prime numbers, we get an overall complexity of

$$\begin{aligned} 2 * \ln(p)O(\log^3(p)) &= 2 * O(\ln(p))O(\log^3(p)) \\ &= O(\log(p))O(\log^3(p)) \\ &= O(\log(p)\log^3(p)) \\ &= O(\log^4(p)) \end{aligned}$$

for the prime generation step.

The second step in the RSA algorithm is finding a modular inverse, which has complexity $O(\log^3(e))$, where e is the public exponent. We use the Extended Euclidean algorithm for this step as shown in Section 5.3.1. The analysis in [8]

indicates that the number of recursive call in the Euclidean algorithm is $O(\log(\mathsf{b})) = O(\log(e))$. Further down in the same chapter, it is said that the number of recursive calls in the extended Euclidean algorithm is the same as the Euclidean algorithm. Seeing as one modulo operation and one subtraction is done every recursive call, we do $O(\log(e))$ modulo operations and $O(\log(e))$ subtractions in total. So, the final complexity of this step is

$$O(\log(e))O(\log(e)) + O(\log(e))O(\log^2(e)) = O(\log^2(e))O(\log^3(e)) = O(\log^3(e))$$

The final step in the basic RSA algorithm is modular exponentiation, which has complexity $O(\log^3(m))$, where m is the exponent. The algorithm used for this step is exponentiation by squaring, which was described in Listing 5.9. In order to calculate $a^m \mod p$ with this algorithm, we first pre-calculate all $a^{2^i} \mod p$ where $2^i \leq m$. In other words, we are calculating all powers of a where the exponent is composed of fewer bits then m. Seeing that m is composed of $\lfloor \log_2(m) \rfloor + 1$ bits, we need to pre-compute $\lfloor \log_2(m) \rfloor + 1$ powers of x. Furthermore, as seen in Listing 5.9, each $x^{2^a} \mod p$ is calculated by multiplying $x^{2^{a-1}} \mod p$ by itself. So, this first step takes $\lfloor \log_2(m) \rfloor$ multiplication (we do not have to use multiplication to calculate a^{2^0}) and $\lfloor \log_2(m) \rfloor + 1$.

For the second step of repeated squaring, we calculate a product of $a^{2^i} \mod p$, where $i < \lfloor \log_2(m) \rfloor + 1$. In the worst case scenario, we need to multiply all such values, which would yield us $\lfloor \log_2(m) \rfloor + 1$ multiplications and $\lfloor \log_2(m) \rfloor + 1$ modulo operation. So, in the repeated squaring methods, we need to perform $2\lfloor \log_2(m) \rfloor + 1$ multiplications and $2\lfloor \log_2(m) \rfloor + 2$ modulo operations. So, the complexity of modular

exponentiation is

$$
\begin{aligned}
&(2\lfloor \log_2(m) \rfloor + 1)O(\log^2(m)) + (2\lfloor \log_2(m) \rfloor + 2)O(\log^2(p)) \\
&= 2\lfloor \log_2(m) \rfloor O(\log^2(m)) + O(\log^2(m)) + (2O(\lfloor \log(m) \rfloor) + 2)O(\log^2(p)) \\
&= O(\lfloor \log(m) \rfloor)O(\log^2(m)) + O(\log^2(m)) + O(\log(m))O(\log^2(p)) \\
&= O(\log(m))O(\log^2(m)) + O(\log^2(m)) + O(\log(m))O(\log^2(p)) \\
&= O(\log^3(m)) + O(\log^2(m)) + O(\log(m))O(\log^2(p)) \\
&= O(\log^3(m)) + O(\log(m))O(\log^2(p))
\end{aligned}
$$

The RSA cryptosystem is the combination of the steps above. So, to calculate the time complexity of RSA, we simply add the complexity of all those operations together. Doing this would give us

$$O(\log^4(p)) + O(\log^3(e)) + O(\log(e))O(\log^2(N)) + O(\log^3(d)) + O(\log(d))O(\log^2(N))$$

where p is one of the primes generated, e is the public exponent, d is the private exponent, and N is the modulus.

This analysis indicates that the RSA algorithm is somewhat slow, seeing that it's execution at least of complexity order n^3. This complexity, however, is not very informative, for all we can glean from it is that RSA is polynomial in terms of three variables. In order to make the RSA algorithm faster, we need to understand its run time in terms of the numbers it operates on.

5.3.3 Runtime in context

In the above section, we have analyzed the complexity of RSA and received a function that depends on three variables, which does not offer much insight into how fast RSA is. In order to obtain more information from this analysis, we consider

which of the three parts of the RSA algorithm is executed the most. At the same time, we keep in mind the typical execution time of each part. After all, an operation that only has to run once a year but takes three whole months to finish would use up a larger amount of time then an operation that runs for one hour everyday.

In a realistic scenario, we try to generate primes that are at least 256-bit long. If we let the generated primes be p, q, we have an approximation for the run time of this step as $\log^4(p) + \log^4(p) = 256^4 + 256^4 \approx 8.4 \times 10^9$ operations. As for calculating the private exponent, we need to know the size of the public exponent. While the method we use to generate it does not give much of guarantee on size, we know that the public exponent should be at least smaller than the modulus. Because we have two primes of size 256 bits, the modulus should be around 512 bits. Now, we assume that the public and private exponents e, d each have half the number of bits of the modulus, or 256 bits. So, we have the approximation for the run time of the calculation of the private exponent as $\log^3(e) = 256^3 \approx 1.7 * 10^7$ operations.

One important observation here is that we do not need to generate a new key every time we encrypt a message. In fact, we can reuse a set of keys to encrypt and decrypt thousands of messages before generating a new one. For this analysis, we assume that we use a set of keys $10,000$ times before making new ones. So, the run time of the encryption and decryption step is approximately $10,000(\log^3(e) + \log(N)\log^2(e) + \log^3(d) + \log(N)\log^2(d)) = 10,000 * 2(\log^3(e) + \log(N)\log^2(e)) = 10,000 * 2(256^3 + 512 * 256^2) \approx 10^{12}$ operations.

From this brief (and imprecise) analysis, we found that the encryption and decryption step takes the most time of all, executing around 100 times more operations than the other two steps combined[3].

[3] $10^{12}/(8.4 \times 10^9 + 1.7 * 10^7) \approx 118$

CHAPTER 6

Improving the RSA cryptosystem

In this chapter, we present some ways to make the RSA algorithm less vulnerable to the attacks presented above. Furthermore, we also include optimization techniques that will result in faster execution.

6.1 Strong key generation

As covered in section 5.1, the way that the naive RSA algorithm generates keys might lead to security vulnerabilities. The most important of such vulnerabilities is that a small modulus can be easily factored. Thus, it is important to ensure that our modulus is large. Generally, 512 bits is the minimum acceptable size, but due to advances in factoring algorithms and computing power, $1,024$ bits is accepted as the minimum size of the modulus where RSA starts to be truly secure.

Another problem with naive RSA prime generation is that it might generate primes with properties that the algorithms in Section 5.1. So, we must select our primes such that they do not have these properties. Trial division is easy enough to deal with. We simply have to choose primes that are not too small. This is not too hard a task, seeing that we have 512 bits to distribute between two primes. Generally, a prime above 10^8 can be considered good, and we need fewer than 32 bits for such a number.

As for Fermat algorithm, we simply have to generate keys that are not too close to each other. A difference of a few bits is good enough to render factorization by Fermat's algorithm infeasible a majority of the time. However, we don't want the distance between the factors to be too large either, seeing as trial division will become feasible due to the smallness of one of the factors. Finally, for Pollard's $p-1$ algorithm, we can do nothing but check and regenerate as needed. Still, this is not a problem, seeing as most large primes should not have a predecessor composed of small primes, and prime generation is computationally inexpensive overall.

6.2 Padding

In Chapter 5, we have shown that the deterministic and malleable nature of the naive RSA algorithm makes it susceptible to chosen texts attack. In order to remove these properties from RSA, we introduce the concept of padding. Padding is a technique in which we add some random information to a message. Pure random information, however, would make it so that we cannot retrieve the unpadded message. Thus, we add the restriction that the padding must have some structure that makes it reversible. We will show that padding makes RSA less susceptible to the two attacks presented in 5.2.

Suppose that an attacker is attempting a chosen plaintext attack on a system using RSA with 100-bit padding. By the nature of this kind of attack, the attacker has access to a small number of plaintexts that is likely to be a message. In this case, suppose that these are days in a month, and so there are 31 of them. Normally, the attacker only needs to encrypt these 31 plaintexts and compare them with the ciphertext to complete the attack, but due to padding, they also need to take into account all padded versions of the plaintext. In total, a sequence of 100 bits has 2^{100} possible states. Even if we suppose only one percent of these are valid for padding,

that is still around 10^{28} valid padding, which is still a massive amount. A successful chosen plaintext attack will need to encrypt all message-padding combinations, which would take an incredibly long time.

On the other hand, suppose that the attacker is trying to attempt a chosen ciphertext attack. The core idea behind this attack is that in naive RSA, multiplying a ciphertext by any number yields a valid ciphertext. RSA with padding, however, does not have this property. We have mentioned above that the information used for padding must have some kind of internal structure. This structure makes it so that there are fewer valid plaintexts then there are plaintexts, and the same holds for ciphertexts. A consequence of this is that we can define a padding scheme for RSA in which multiplying a valid ciphertext by a number yields an invalid ciphertext most of the time. This makes the chosen ciphertext attack infeasible, seeing as the attacker would then have to check a large count of integers before finding one that would allow for an attack. What this entails is that the attacker has to then request for the sender to decrypt a large number of messages, which might raise red flags.

So, we have shown that padding makes the chosen text attacks in Section 5.2 infeasible. The question now is what are some padding schemes commonly used for RSA. Generally, the industry standard at the writing of this paper is the Optimal Asymmetric Encryption Padding (OAEP) standard, explained in details in [**?**].

6.3 Improving RSA's runtime

In this section, we present some methods for increasing the speed at which the RSA algorithm executes.

In the RSA algorithm, the two most time-consuming steps are encryption and decryption as shown in 5.3.3. Thus, we focus on these two operations when improving RSA's execution time. Each of these operations amount to a modular

exponentiation. So, there are two things we can influence, the size of the exponent and the size of the modulus. However, the modulus must be very large for the RSA algorithm to be secure, as shown in Section 5.1. As a result, we can only make RSA faster by changing the exponents.

Generally, in order to minimize execution speed, we want to pick the exponent to be as small as possible. However, we can only choose the exponent for either encryption or decryption, seeing that their relationship as modular inverse over a large modulus almost guarantees that one of them has to be large if the other is small. One problem with choosing a small private exponent is that it is vulnerable to Weiner's attack, a linear time method for retrieving a small private exponent [3]. So, ultimately, the only way by which we can speed up RSA is by choosing a small public exponent. Doing this would mean that we have to spend more time generating the prime pair, but that is an acceptable trade-off to speed up the costliest part of the RSA algorithm.

6.3.1 Choice of public key

In order to choose a good exponent, we need to examine how modular exponentiation is computed. As discussed in Section 5.3.1, one way to calculate modular exponentiation is to use the repeated squaring method. This is, in fact, the canonical algorithm for modular exponentiation, with most others being based on it. Thus, we only need to examine the repeated squaring method in order to choose a good exponent.

There are two factors that influence the run time of repeated squaring: number of bits required to express the exponent and the number of 1 bits in the exponent. We first proceed with optimizing the second quantity. It is easy to see why it cannot be 0 or 1, seeing as the only numbers that have this many 1 bits in their binary representation are 0 and 1. If the public exponent is 0, 1 is the only possible

ciphertext, while if the public exponent is 1, we are encrypting a message to itself, which is massively insecure. So, the minimum number of 1 bits in the public exponent is 2. Seeing as the public exponent must be odd, we know that it must be in the form $2^n + 1$.

Now, we find the optimal number of bits required to express the public exponent. First of all, we want the public exponent e to be prime, seeing as it would then be a lot easier to check that e is coprime to $\phi(N)$, where N is the modulus. This stems from how the algorithm used to find the gcd of two numbers, the Euclidean algorithm, functions. For the sake of demonstration, we include the pseudocode for this algorithm in Listing 6.10. We can see that this algorithm finishes when $b = 0$. The only way for this to happens is if `b` divides `a`. So, the only scenario in which the Euclidean algorithm calculates $\gcd(\phi(n), e)$ in one step is if e divides $\phi(n)$. Because the Euclidean algorithm is the best way to check that e is coprime to $\phi(n)$ if e can be non-prime, we always have the possibility of performing more than one modulo operation.

```
1  EUCLID(a, b)
2  if b = 0
3      return a
4  else return EUCLID(b, a mod b)
```

Listing 6.10: Pseudocode for Euclidean algorithm [8].

If e is a prime, however, there is a method that guarantees one modulo operation. Seeing that e is prime, if e is not coprime to $\phi(N)$, e must be a prime factor of $\phi(N)$. So, we can check if e and $\phi(N)$ are coprime by calculating $\phi(N) \mod e$. This expression evaluates to 0 only if e is coprime to $\phi(N)$. So, we can always determine the coprimeness of e and $\phi(N)$ in one modulo operation.

Now, we notice that e should be of the form $2^n + 1$. It has been proven that if a number of this form is prime, then $n = 2k$ for some k [1]. We call these

primes Fermat primes, and only 5 of them have been found so far: $3, 5, 17, 257$, and 65537 [32]. We might be tempted, for the sake of maximum performance, to pick 3. There is an attack that exploits this choice, however. Consider an RSA key with modulus $27,371$. Suppose we want to encrypt the message 21. We have $21^3 = 9,261 \equiv 9,261 \mod 27,371$. Seeing as we did not reduce the ciphertext over $27,371$, an eavesdropper only needs to take the cube root of the ciphertext to obtain the original message.

As the public exponent gets larger, the distance between the message and the modulus required to make this attack possible becomes incredibly large. Ideally, in order to ensure that this attack is infeasible for most practical purpose, we want our public exponent to be as large as possible. For that reason, we choose the public exponent to be $65,537$. This choice does not sacrifice too much in terms of run time (we still need fewer than 20 multiplication and modulo operations) while giving us a greater degree of security.

6.3.2 Chinese remainder theorem

Due to choosing a relatively small public exponent, the private exponent becomes large, which results in a long decryption time. One way we can make decryption faster is through the Chinese Remainder Theorem (Theorem 3.5). The idea is that, instead of modular exponentiating with a large exponent, we split the calculation into two modular exponentiations with smaller exponents. Seeing as the run time of modular exponentiation is cubic in terms of the input size, computing two exponentiations with smaller exponents is faster than computing a large one.

Suppose that we have $p > q$, where p, q are the prime pairs in an RSA key. The first step in this new decryption algorithm is to precompute some values at key

generation. More specifically, we need to find the solutions of three equations.

$$d_p e \equiv 1 \pmod{p-1}$$

$$d_q e \equiv 1 \pmod{q-1}$$

$$q_{inv} q \equiv_1 \pmod{p}$$

where p, q are the prime numbers, and e is the public exponent.

Now, let c be the ciphertext we are trying to decrypt. In order to obtain the original message m, we first calculate $m_1 = c^{d_p} \mod p$ and $m_2 = c^{d_q} \mod q$. Then, we calculate the quantity $h = q_{inv}(m_1 - m_2) \mod p$, which finally allows us to obtain the original message as $mm_2 + hq$ [28].

At this point, we have discussed a basic conception of the RSA algorithm, some security and speed problems with this version of RSA and solutions to the aforementioned problems. In the next chapter, we present our implementation of the RSA algorithm and some other implementations, and our test to determine how our implementation compare to others.

CHAPTER 7

Implementing and testing MyRSA

In this chapter, we present the the goal of this project, namely, to determine whether it is a good idea to implement an encryption algorithm instead of using existing implementations, using the RSA algorithm as a case study. We then discuss our implementation of RSA and the existing algorithms. Finally, we present our testing methodology and the results that we obtained.

7.1 Research question

When we browse forums that work with cryptography, there is one advice that is echoed by everyone, "do not roll your own crypto". This advice stems from the fact that cryptography is one of the most important lines of defense a system has against attacks. Thus, many attempts have been made, with varying success, to find ways to reverse encryption algorithm without access to the necessary keys. We have demonstrated several such attacks on the RSA algorithm in Chapters 5 and 6. As a result, implementing a truly secure cryptographic algorithm involves researching those attacks and implementing countermeasures, which is something most people lack either the time or skill to do.

Having said that, there has not been any research into empirically testing this claim of "do not roll your own crypto". In this project, we aim to provide a preliminary step for further research on this topic through implementing our own

version of the RSA algorithm and testing it against some existing implementations in terms of execution speed and vulnerability to attacks.

In this project, we implement the RSA algorithm in Java, with some simple optimizations and security measures applied. Then, we empirically compare it against the implementation of RSA provided with the Java Development Kit. In addition, we test two additional cryptography libraries implemented in Python, python-rsa and PyCryptodome, in order to compare the cryptographic ecosystem of different programming languages.

There are two reasons Python is chosen for this project. First of all, Python is known for having slow execution speed. Thus, it would be enlightening to compare optimized Python code and relatively unoptimized Java code. The second reason for our choosing Python is its support for arbitrary precision arithmetic in the core language. Thanks to this, the number of third-party libraries that has to be installed is minimal.

7.2 The implementations of RSA

Here, we give some details about the implementations of the RSA algorithm that we test. More specifically, we give a high level overview of how each implementation handles key management, encryption, and decryption. Furthermore, we also give example code to perform these three operations. For the section on our implementation of the RSA algorithm, we also discuss our design process and the details of the implementation.

7.2.1 Python-RSA

```
1 import rsa
2 # Generate new keys
```

```
3  (pubkey, privkey) = rsa.newkeys(512)
4
5  # Load (private) key from file. The procedure is the same
       for public key.
6  with open('private.pem', mode='rb') as privatefile:
7      keydata = privatefile.read()
8  privkey = rsa.PrivateKey.load_pkcs1(keydata)
9
10 # Load pre-generated keys in number form
11 pubkey = rsa.PublicKey(n=15, e=7)
12 privkey = rsa.PrivateKey(n=15, e=7, d=7, p=3, q=5)
13
14 # Encryption and decryption
15 message = 'Hello World!'.encode('UTF-8')
16 cipher = rsa.encrypt(message, pubkey)
17 message = rsa.decrypt(cipher, privkey)
```

Listing 7.11: Example usage of Python-RSA.

Python-RSA is, as described in [34], "a pure-Python RSA implementation". In this library, key management are handled with the classes `rsa.PublicKey` and `rsa.PrivateKey`. As their names indicate, `rsa.PublicKey` contains the public keys and `rsa.PrivateKey` contains the private keys. There are two ways to obtain these keys in Python-RSA, generating new ones or loading pre-generated values. The code used to do so is included in Listing 7.11.

There are two things that we would like to note in this piece of code. First of all, in line 3, we are creating a new key pair by using the method `rsa.newkeys`, which takes an argument called `keysize`. This argument denotes the size of the modulus. Generally, if an implementation of RSA provides a method to generate keys, their

interface will be similar to `rsa.newkeys`, taking an argument that determines the size of the modulus. The second point of interest is on lines 11 and 12. In these two lines, we create a key pair by explicitly defining what the individual keys should be. We can see that the names Python-RSA uses for each keys are almost identical to the name we used in Section 3.2, with the sole difference being the use of a lowercase n instead of an upper case. In fact, Python-RSA share this feature with PyCryptodome, an implementation of RSA we also test in this project.

Once we have one instance each of `rsa.PublicKey` and `rsa.PrivateKey`, we proceed on to the encryption and decryption steps. First, we process the message from a string into a byte string, which is the only form of message that Python-RSA accepts. We can do this by using some common character encodings such as ASCII and the various Unicode standards. Once we have done this, encryption and decryption can be accomplished using two methods, `rsa.encrypt` and `rsa.decrypt`. The signatures for these methods, as provided in [34], are as follow

- `rsa.encrypt(message: bytes, pub_key: rsa.PublicKey)` → `bytes`
- `rsa.decrypt(crypto: bytes, priv_key: rsa.key.PrivateKey)` → `bytes`

7.2.2 PyCryptodome

PyCryptodome is "a self-contained Python package of low-level cryptographic primitives" [11]. Cryptographic primitives are simple cryptographic algorithms, such as hash functions, and encryption and decryption algorithms, from which more sophisticated systems can be built. The unique thing about PyCryptodome, however, is that it is not a wrapper for other libraries. Instead, PyCryptodome implements everything from scratch, mostly using Python but deferring to C extension for performance-critical operations [11]. The point of interest for us is

that PyCryptodome implements the cryptographic primitives that form the RSA cryptosystem.

Before exaplaining the usage of PyCryptodome for RSA, we first discuss some details of the structure of the library. PyCryptodome contains a number of different modules, each of which implements a different type of cryptographic primitives. In particular, key management for RSA resides in the module `Crypto.PublicKey`, while encryption and decryption resides in `Crypto.Cipher`. If we include these modules when discussing an object in the library, it would make the object unwieldy and unnecessarily confusing. As a result, we will omit these modules' name in the paragraphs below, aside from when we first introduce the submodule we work with.

```
1  from Crypto.PublicKey import RSA
2  # Create a new (private) key.
3  key = RSA.generate(2048)
4
5  # Load keys from file.
6  f = open('mykey.pem','r')
7  key = RSA.import_key(f.read())
8
9  # Load pre-generated keys in number form
10 # The keys are, in order, n, e, d, p, q
11 # d is only required if we are loading a private
       key
12 # p, q are optional
13 key = rsa.construct((15, 7, 7, 3, 5))
14
15 # Encryption and decryption
16 message = 'Hello World!'.encode('UTF-8')
```

```
17          cipher = PKCS1_OAEP.new(key)
18          ciphertext = cipher.encrypt(message)
19          message = cipher.decrypt(ciphertext)
```

Listing 7.12: Example usage of PyCryptodome

In PyCryptodome, key management for RSA is handled by the module `Crypto.PublicKey.RSA`, and more specifically with the class `RSA.RsaKey`. Similar to Python-RSA, we can either generate new keys or loading pre-generated keys. The code used to do so is included in Listing 7.12. One interesting design choice we want to showcase is that in PyCryptodome, there is no distinction between public keys and private keys on the class level, where everything is a `RSA.RsaKey`. Instead, the two types of keys are differentiated on the attribute level, where private keys contain the private exponent `d`, while public keys do not. In the context of PyCryptodome being a collection of a large number of cryptographic primitives, this design choice makes logical sense, seeing that it would help cut down on the number of classes that exists in an already large library.

Once we obtain a key, we can proceed to encryption and decryption. Like Python-RSA, PyCryptodome only works on byte strings, so we need to convert the message into one. Once this is done, we can create an encryption/decryption engine by passing a key to the function `PKCS1_OAEP.new`. By default, this engine can encrypt messages using the method `encrypt`. Decrypting ciphertexts with the method `decrypt`, however, is possible only if we created the engine with a private key. Otherwise, an error is raised.

7.2.3 JDK

```
1           # Generate new keys
```

```
2           KeyPairGenerator generator = KeyPairGenerator.
               getInstance("RSA");
3           generator.initialize(2048);
4           KeyPair pair = generator.generateKeyPair();
5           PrivateKey privateKey = pair.getPrivate();
6           PublicKey publicKey = pair.getPublic();
7
8           # Load (private) key from file. The procedure is
               the same for public key.
9           File publicKeyFile = new File("public.key");
10          byte[] publicKeyBytes = Files.readAllBytes(
               publicKeyFile.toPath());
11          KeyFactory keyFactory = KeyFactory.getInstance("
               RSA");
12          EncodedKeySpec publicKeySpec = new
               X509EncodedKeySpec(publicKeyBytes);
13          keyFactory.generatePublic(publicKeySpec);
14
15          # Load pre-generated keys in number form
16          RSAPublicKeySpec publicKeySpec = new
               RSAPublicKeySpec(15, 7);
17          RSAPrivateKeySpec privateKeySpec = new
               RSAPrivateKeySpec(15, 7);
18          PublicKey publicKey = keyFactory.generatePublic(
               publicKeySpec);
19          PrivateKey privateKey = keyFactory.generatePrivate
               (privateKeySpec);
20
21          # Encryption and decryption
```

```
22          String message = "Baeldung secret message";
23          byte[] messageBytes = message.getBytes(
                StandardCharsets.UTF_8);
24          Cipher encryptCipher = Cipher.getInstance("RSA");
25          encryptCipher.init(Cipher.ENCRYPT_MODE, publicKey)
                ;
26          byte[] encryptedMessageBytes  = encryptCipher.
                doFinal(messageBytes);
27          Cipher decryptCipher = Cipher.getInstance("RSA");
28          decryptCipher.init(Cipher.DECRYPT_MODE, privateKey
                );
29          byte[] decryptedMessageBytes = decryptCipher.
                doFinal(encryptedMessageBytes);
30          String decryptedMessage = new String(
                decryptedMessageBytes, StandardCharsets.UTF_8);
```

Listing 7.13: Example usage of Java's builtin RSA. Some code was taken from [21].

The main difference between Java's builtin RSA implementation (which we call JDK for the sake of brevity) and the Python implementations is that JDK is designed to handle multiple encryption algorithms under one interface. As a result, the discussion below can be applied to more than RSA.

In JDK, key management is primarily handled with the classes `PublicKey` and `PrivateKey`. As with the previous two implementations, we can either generate new keys or load pre-generated keys, and the code to do so is include in Listing 7.13.

Again, like the other two implementations we discuss, JDK only works with bytes, and so we need to convert the message from a string to a byte array. Then, in order to encrypt and decrypt in JDK, we create an instance of the `Cipher` class and

initialize it with a mode and a key. In particular, we use a `Cipher.ENCRYPT_MODE` and a `PublicKey` for an encryption `Cipher` object, and a `Cipher.DECRYPT_MODE` and a `PrivateKey` for an decryption `Cipher` object. Once this is done, encryption and decryption are both accomplished by calling the `doFinal` method of the `Cipher` class.

7.2.4 Our Java implementation of RSA: MyRSA

We aim for our implementation to be as similar to what an average programmer under time constraint might come up with. Using this model, we define some principles that our implementation must follow:

- No reimplementation of any functionality that already exists in the language.
- Search for existing implementations of nontrivial algorithms before trying to implement them.
- Keep the number of external libraries as low as possible.

For our implementation, we use Java. While the primary reason we choose Java is us being familiar with it, we also have two other reasons for this. First of all, Java supports big numbers in its standard library with the class `BigInteger`. Many other languages, including C and C++, only have support for numbers with at most 64 bits. This is not enough for our purpose, seeing that we might need to work with numbers composed of hundreds to thousands of bits. While it is possible to install external libraries to work with these numbers, that would go against our third principle.

Support for big numbers, however, is not enough of a reason to choose Java as our language seeing as Python also bundle this feature with its standard library. The other reason we choose Java is its performance. While Java is generally not the

fastest language, it is still faster than Python. In order to confirm this intuition, we conducted a small test. In 5.3.3, we established that the most significant operation in the RSA algorithm is encryption and decryption, which boils down to modular exponentiation. So, in order to compare Java and Python, we run $100,000$ modular exponentiations in both languages. In total, Java takes less than 3 seconds, while Python takes up to 9 seconds.

In our implementation, we choose the public key e to be $65,537$, as explained in section 5.3.3. Seeing that we have the public key, we need to generate the primes p, q such that $\gcd(e, \phi(pq)) = 1$. This can be accomplished with the code in Listing 7.14.

```
1 BigInteger p, q;
2 do {
3     p = BigInteger.probablePrime(keySize / 2, new Random()
          );
4     q = BigInteger.probablePrime(keySize / 2, new Random()
          );
5 }
6 while(!e.gcd(totient(p, q)).equals(BigInteger.ONE));
```

Listing 7.14: Using Java's BigInteger library to generate the two primes needed for the RSA algorithm.

Here, the most notable thing is the method `BigInteger.probablePrime`. The implementation of this method is similar to the procedure of generating random prime number in section 5.3.1. The key difference, however, is in its use of a sieve to eliminate most composite numbers [17, Lines 731-734]. This gives us a good decrease in execution time, seeing that we do not need to use the Miller-Rabin test on as many numbers. The remaining portion of the above code is mundane, being mainly a while loop to generate prime numbers until we get a pair p, q that satisfies $\gcd(e, \phi(pq)) = 1$. Now, with e, p, q found, we can find the private key d, which is

defined as the inverse of e mod $\phi(p,q)$. There is a method to do just this in Java, `BigInteger.modInverse`. This method takes in two arguments, the number to find the inverse of and the modulus. Using this method, we can easily compute d. At this point, we have finished generating the keys. The implementation of this portion of the RSA algorithm is included in Listing 7.1.

```
1  public static RSAKey generateKey(int keySize) {
2      BigInteger e = new BigInteger("65537");
3      BigInteger p, q;
4      do {
5          p = BigInteger.probablePrime(keySize / 2, new
               Random());
6          q = BigInteger.probablePrime(keySize / 2, new
               Random());
7      }
8      while(!e.gcd(totient(p, q)).equals(BigInteger.ONE));
9      BigInteger n = p.multiply(q);
10     BigInteger d = e.modInverse(totient(p, q));
11     return new RSAKey(n, e, d, p, q, keySize);
12 }
```

Listing 7.1: Function used to generate the keys for the RSA algorithm.

One thing that we would like to note is on line 11 of Listing 7.1, we create a new object of class RSAKey. This class is the main method by which we handle key management. RSAKey, however, does not do much by itself, aside from having a method to verify that the set of keys it represents is a valid RSA key. Instead, its main purpose is to wrap two other classes, RSAPrivateKey and RSAPublicKey, which are the classes that actually store the keys. In RSAPublicKey, we store the

keys e and N, while in RSAPrivateKey, we store the keys d, p, q, N. Furthermore, RSAPrivateKey also contains the class CRTNumbers, which contains the numbers required to use the Chinese Remainder Theorem to do decryption. These numbers are precalculated when an object of type RSAPrivateKey is instantiated.

With the key generation step done, we move on to the encryption and decryption process. Seeing that RSA encryption is simply a modular exponentiation, we can easily implement it with the `BigInteger.modPow` method. This method makes use of a modified form of the repeated squaring algorithm, with one major change. In this method, a sliding window of size k is introduced over the bits of the exponent. Furthermore, we keep an exponent $b = n^{k\text{th highest bits of the exponent}}$. Then, as the window slides over the bits of the exponent, we have a number of patterns that correspond to a sequence of squarings and multiplying. In essence, the sliding window permutes the steps in the repeated squaring algorithm. However, it also provides the benefit of reducing the number of squarings and multiplications needed. Per [17, Lines 2549-2550], a window of size k removes $k - 2$ squarings as well as some multiplications. So, if k is big enough, we can gain a significant speedup for our algorithm.

On the other hand, for decryption, we implement the Chinese Remainder Theorem optimization. It is relatively trivial to do so, seeing that each step of this algorithm can be implemented using one method call. So, we are finished with the implementation of the RSA algorithm. One point that we have not touch on, but used in the algorithm, is the implementation of OAEP (Optimal Asymmetric Encryption Padding), a padding standard we mentioned in Section 6.2. Due to OAEP being complex to implement, following our second principle, we searched for an implementation of OAEP and found one on GitHub. With this, we now have the code for encryption and decryption process as shown in Listing 7.15.

```
1 public CRTNumbers() {
2     dP = d.mod(p.subtract(BigInteger.ONE));
```

```
3       dQ = d.mod(q.subtract(BigInteger.ONE));
4       qInv = q.modInverse(p);
5   }
6
7   public static BigInteger encrypt(String message,
        RSAPublicKey publicKey) throws Exception {
8       byte[] encodedMessage = message.getBytes(
            StandardCharsets.UTF_8);
9       BigInteger messageAsInteger;
10      while(true) {
11          encodedMessage = OAEP.pad(encodedMessage, "SHA-256
                ␣MGF1", publicKey.getKeySize() / 8);
12          messageAsInteger = new BigInteger(encodedMessage);
13          if(messageAsInteger.compareTo(BigInteger.ZERO) >
                0) {
14              break;
15          }
16          else {
17              encodedMessage = OAEP.unpad(encodedMessage, "
                    SHA-256␣MGF1");
18          }
19      }
20      return encodedMessage.modPow(publicKey.getE(),
            publicKey.getN());
21  }
22
23  public static String decrypt(BigInteger cipher,
        RSAPrivateKey privateKey) throws Exception {
```

```
24      BigInteger m1 = cipher.modPow(privateKey.crtNumbers.dP
            , privateKey.getP());
25      BigInteger m2 = cipher.modPow(privateKey.crtNumbers.dQ
            , privateKey.getQ());
26      BigInteger h = privateKey.crtNumbers.qInv.multiply(m1.
            subtract(m2)).mod(privateKey.getP());
27      BigInteger decryptedMessage = m2.add(h.multiply(
            privateKey.getQ()));
28      byte[] paddedMessage = decryptedMessage.toByteArray();
29      byte[] unpaddedMessage = OAEP.unpad(paddedMessage, "
            SHA-256 MGF1");
30      return new String(unpaddedMessage, StandardCharsets.
            UTF_8);
31  }
```

Listing 7.15: The complete code for MyRSA

7.3 Testing methodology

The metrics we will be using in this comparison are execution time and vulnerability against some common attacks, such as brute force and side-channel attacks. Here, vulnerability against an attack is determined based on whether an implementation can be defeated by that attack and the duration of a successful attack.

For execution time, we will compare two metrics: encryption time and decryption time. For both of these tests, we will initialize the cryptosystems by generating the public and private keys. We will use two key sizes: 1,024 bits, and 2,048 bits. Here, key size refers to the size of the modulus N. For the public exponent e, we will use the value recommended by all implementations, which is 65,537.

Once we have generated the keys, we will test each implementation on two messages, one with length 13 and the other with length 52. These messages are pre-generated sentences, created in such a way that they can be encrypted in one block. In order to turn a message into a form that the RSA algorithm can use, we will encode each character to UTF-8, form a byte strings from the encoded characters, and turn that byte string into an integer.

For each message, we will test the encryption process with 100 rounds, with each round composed of $10,000$ encryptions. For the testing of decryption, we will do 50 rounds, each round again composed of $10,000$ decryptions. Here, we test decryption half the number of times of encryption because it tends to be much slower.

For the vulnerability test, we will be testing two attacks on the RSA algorithm: brute force factorization of the modulus and timing-based attack. Due to time and resource constraint, we will not attempt a full factorization of the modulus using general-purpose algorithms. Doing so would take too long, seeing as it takes $2,700$ years for a decent CPU to factor a 829-bit number [38], which is smaller than the modulus we use ($1,024$ and $2,048$ bits). Instead, we only test the special-purpose algorithms mentioned in Section 5.1.

The exact testing procedure for brute force factorization attack on RSA goes as follows. Assume that we have a key k of 1024 bits. We run each of the algorithms defined in Section 5.1 on k. Each of the three algorithms tests if k possesses a certain property. Trial division checks if k has a small factor, Fermat's algorithm checks if the two factors of k are close together, and Pollard's $p-1$ algorithm checks that $k-1$ does not have small factors. Then, for each implementation, we generate 100 such k and run these algorithms on these ks.

Now, in order to ensure that our test actually fails when k does not have the property we check for, we set some constraints on each algorithm. For trial division

and Pollard's $p - 1$, we only consider numbers less than 10^7, which we believe is a reasonable expectation for small primes. As for Fermat's algorithm, we place a limit of 10^6 iterations before stopping execution. Then, we generate 100 keys using each implementation and run the factoring algorithms on them.

As for testing vulnerability against timing-based attack, we follow the same procedure as testing performance. However, instead of two strings with a large difference in length, we will use five strings whose lengths are $10, 20, 30, 40$, and 50 respectively. Then, for each message, we encrypt it and then decrypt the resulting ciphertext $100,000$ times with each of the four implementations.

7.4 Result

7.4.1 Performance test

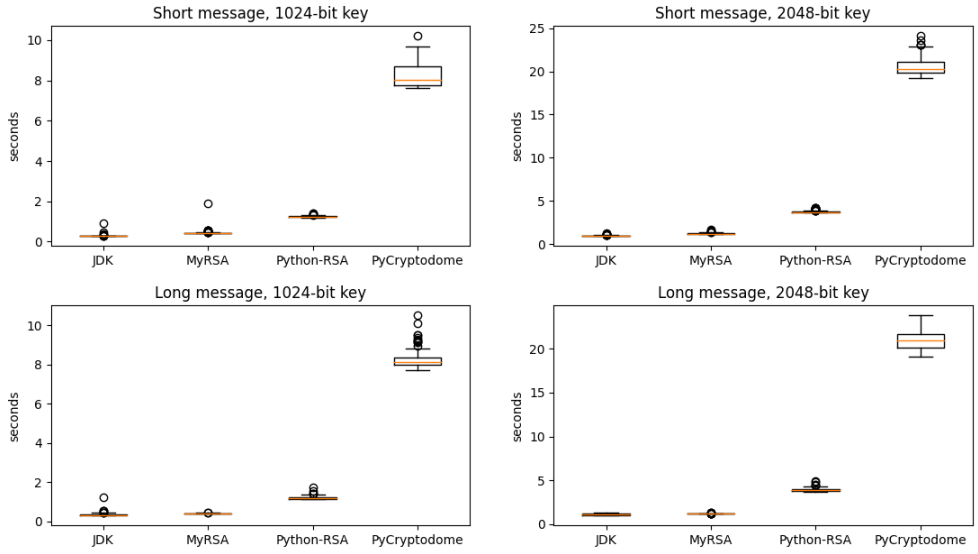

Figure 7.1: The results of testing encryption speed.

We can see in Figure 7.1 that both of the Java implementations of RSA outperform the ones in Python, which was within expectations. Unexpectedly, however, the

execution time of the PyCryptodome library is much higher than the other three, averaging around 8 seconds while the other three do not exceed 2 seconds when encrypting with a 1,024-bit key. The same holds for when we use a 2,048-bit key.

We suspect that this difference in speed stems from how PyCryptodome handles large number. For performance purpose, PyCryptodome implements their own arbitrary precision numbers. In Unix-based operating system, this resolves to wrapping GMP (GNU Multi Precision), one of the best arbitrary precision arithmetic library in terms of performance. However, on Windows systems, which we use for this test, PyCryptodome uses a custom implementation of arbitrary arithmetic. While no data on its performance is available, it is a reasonable guess that it would be slower than `BigInteger` and Python's default integer type on the scale we are working at.

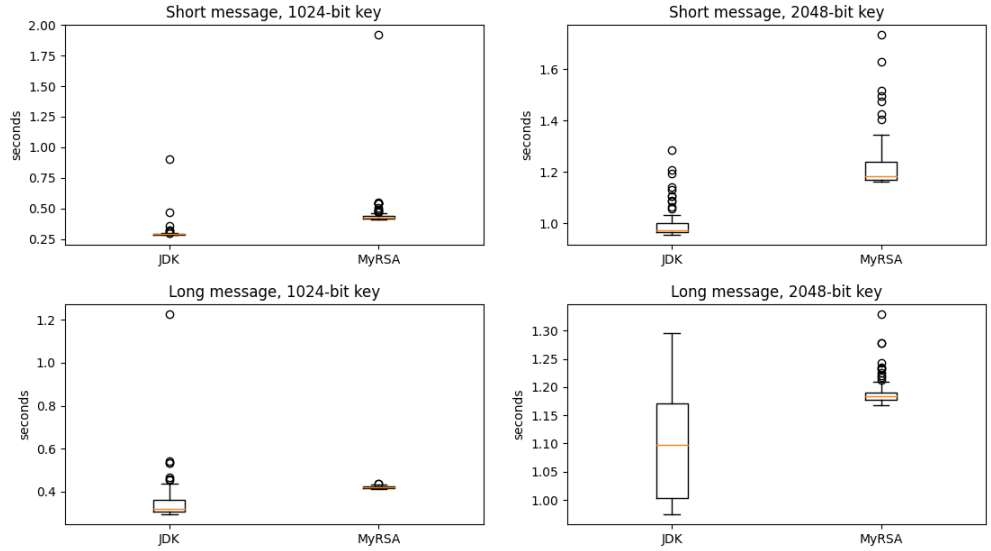

Figure 7.2: The results of testing encryption speed, limited to the Java implementations.

One thing that is difficult to observe in Figure 7.1 is the relative performance of the two Java implementations. In order to more clearly observe this, we graphed the data for those two implementations alone in Figure 7.2. Now, we can observe that our implementations are consistently slower than Java's builtin implementation.

Numerically, the difference in median performance looks to be around 0.1 to 0.2 seconds over 10,000 runs. This is not a significant difference in speed, especially when compared to the result of decryption.

One thing we would like to note is that there is a larger range in the execution time of the Java implementations compared to the range observed in the execution time of the Python implementation. We attribute this phenomenon to Java's Just-in-time (JIT) compiler, which dynamically compiles a method's bytecode into native machine code as needed. Using JIT, we can also explain the variance present in the execution time. JIT compilation in Java is not a do-it-once process, seeing that it takes time to compile bytecode into machine code. Instead, a method can be JIT compiled multiple time, with the number of optimizations made increasing each time until the method cannot be optimized any more [15]. Applying this to our testing process, we can reason that the highest data point is from before JIT compilation, the data points in the middle are from the intermediate stages of JIT, and the lowest data points are from the final stage of JIT.

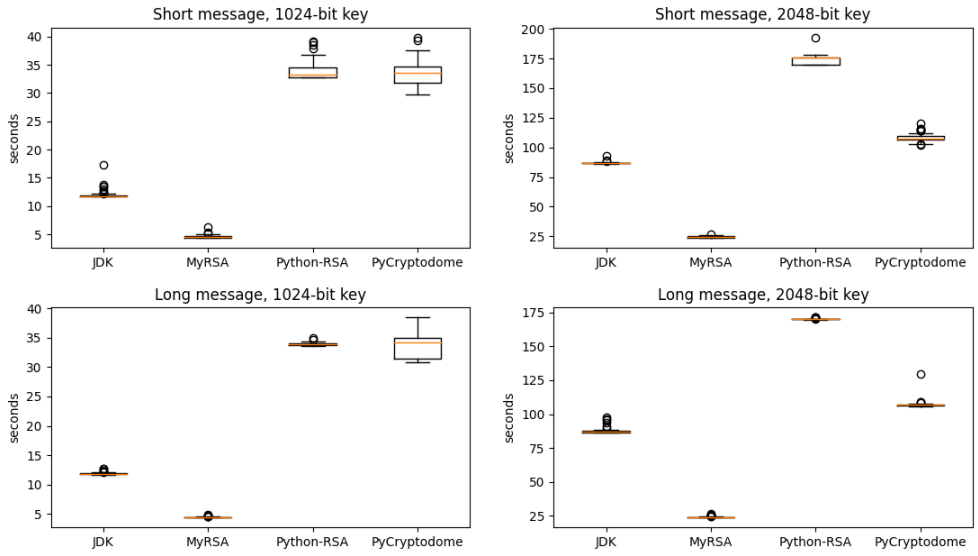

Figure 7.3: The results of testing decryption speed.

As for decryption, we graphed the data we collected in Figure 7.3. We can see at

a glance that our implementation outperforms every other ones. In fact, it is at least twice as fast as the second fastest one. We do not have a concrete hypothesis as to why this might have happened, but based on how each implementation handles the encryption process, we believe that the reason for the performance difference is that our implementation only has to undo OAEP padding, the other implementations have to undo a lot of other things they apply to their message.

7.4.1.1 Vulnerability test

For the factorization security test, no implementation has their modulus factored by the our testing algorithms. In simple terms, the all four implementations of RSA create "solid" keys that our algorithms cannot factor. This leads us to conclude that, in the case of trial division and Pollard's $p-1$ algorithm, all implementations of RSA produce keys with large enough factors to not be instantly factored by factorization algorithms that relies on small factors. Furthermore, we conclude that the two factors of the generated modulus are far enough away from each other that Fermat's algorithm will take a very long time to retrieve a factor.

As for the timing attack security test, we have compiled the collected data in Table 7.1. One interesting thing we would like to point out is the fact that execution time seems to be the highest when the message length is 10. We hypothesize that this difference comes down to the random nature of the padding process. The fact that padding is random means that given two messages a, b with a shorter than b, it is possible that the encoded value of a is greater than b. Thus, without knowledge of the encoding scheme, it is impossible to know if, for example, the ciphertext resulting from the encryption of a message of length 30 is larger than the ciphertext resulting from the encryption of a message of length 20. So, it is impossible to perform a timing attack if we only have knowledge of the message.

While we cannot gleam much information about how different message lengths

lead to a difference in decryption time, the result we got in Table 7.1 still indicates that aside from PyCryptodome, no other implementation is vulnerable to this specific type of timing attack. While Java's builtin implementation and our implementation is a lot slower on the two shortest messages, we can attribute that to the JIT compiler not optimizing the compilation to its highest level. For PyCryptodome, however, there is no such mechanism to explain why there is such a large discrepancy in run time between the first two and the other messages.

Implementation / Message length	JDK	MyRSA	Python-RSA	PyCryptodome
10	0.00177	0.00066	0.00336	0.00447
20	0.00189	0.00057	0.00335	0.00352
30	0.00088	0.00060	0.00331	0.00222
40	0.00089	0.00062	0.00334	0.00206
50	0.00091	0.00062	0.00331	0.00212

Table 7.1: Results for timing attack test. Each data point is in second per encryption.

CHAPTER 8

Conclusions and Future Works

In this project, we present the RSA algorithm, implement it in Java under the name MyRSA, and test MyRSA against three other implementations of RSA.

From the tests we conducted, we conclude that our implementation of the RSA algorithm does no worse than any other algorithm, and might in fact be better than some. More specifically, our implementation outperform all but one external implementation in encryption, and even then the performance difference is not large. The difference is even more pronounced when it comes to decryption, where our implementation is more than two times faster than the second fastest implementation. This indicates either that our implementation possesses some innate superiority in speed, or that our algorithm does not apply as many security patches as the other algorithms. Currently, we are leaning towards the latter conclusion.

On the security side, we find no difference between our implementation and the other implementations in terms of the tests we did. In fact, we found evidence that PyCryptodome, one of the external implementations, might be vulnerable to an attack while our implementation is not. However, this does not mean that our implementation is secure against all attacks. After all, we only test two attacks, and there are a lot of other attacks that can be tested.

One potential path we can take this project in the future is to improve the performance of our implementation. At the moment, we are using functions from

Java's standard library. While they serve their purpose well enough, we suspect that they might not be the fastest algorithms possible. So, we aim to research different implementations of the primitive operations in the RSA algorithm, and implement faster algorithms.

On the security side, we have only tested three special-purpose integer factoring algorithms, and not even the best ones. For future work, we believe that it would be interesting to work on the cutting edge algorithms (e.g. elliptic curve factoring algorithm) and see how they perform when matched against out RSA implementation.

References

1. anon. If $2^n + 1$ is prime, why must n be a power of 2? World Wide Web, May 2012. URL https://math.stackexchange.com/questions/140804/if-2n1-is-prime-why-must-n-be-a-power-of-2. Accessed on March 25, 2022. 57

2. Tom M. Apostol. *Introduction to Analytic Number Theory*, chapter 4, page 80–82. Springer, 2011. 44

3. Dan Boneh. Twenty Years of Attacks on the RSA Cryptosystem. *Notices of the AMS*, 46(2):203–213, 1999. URL https://www.ams.org/notices/199902/199902FullIssue.pdf. 56

4. Marco Cantarini. ln(n) - Average Length of Prime Gaps, April 2016. URL https://math.stackexchange.com/questions/1724274/lnn-average-length-of-prime-gaps. Accessed on March 25, 2022. 44

5. Cepheus. Caesar3. World Wide Web, October 2006. URL https://commons.wikimedia.org/wiki/File:Caesar3.svg. Accessed on March 25, 2022. xv, 5

6. ClownInTheMoon. Strong Induction Proof: Every natural number = sum of distinct powers of 2, September 2017. URL https://math.stackexchange.com/questions/176678/strong-induction-proof-every-natural-number-sum-of-distinct-powers-of-2. Accessed on March 25, 2022. 46

7. Cmglee. Comparison computational complexity. World Wide Web, June 2017. URL https://commons.wikimedia.org/wiki/File:Comparison_computational_complexity.svg. Accessed on March 25, 2022. xv, 28

8. Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. MIT press, Cambridge, MA, 3rd edition, 2009. xix, 21, 22, 25, 31, 45, 46, 49, 57

9. Ryan Daileda. The fermat factorization method. World Wide Web, 2004. URL http://ramanujan.math.trinity.edu/rdaileda/teach/s18/m3341/lectures/fermat_factor.pdf. Accessed on March 25, 2022. 36

10. Yvo Desmedt and Andrew M. Odlyzko. A chosen text attack on the rsa cryptosystem and some discrete logarithm schemes. In *Advances in Cryptology*, CRYPTO '85, page 516–522, Berlin, Heidelberg, 1985. Springer-Verlag. ISBN 3540164634. 42

11. Helder Eijs. Pycryptodome's documentation. World Wide Web, February 2022. URL https://www.pycryptodome.org/en/latest/index.html. Accessed on March 25, 2022. 64

12. Niels Ferguson, Bruce Schneier, and Tadayoshi Kohno. *Cryptography engineering: design principles and practical applications*, chapter 2, pages 28–29. John Wiley & Sons, 2011. 7

13. David Göthberg. Public key encryption. World Wide Web, August 2006. URL https://commons.wikimedia.org/wiki/File:Public_key_encryption.svg. Accessed on March 25, 2022. xv, 8

14. Richard H Hammack. *Book of proof*. Richard Hammack, 3rd edition, 2013. 10

15. IBM. JIT compiler overview. World Wide Web, February 2021. URL https://www.ibm.com/docs/en/sdk-java-technology/7?topic=jc-jit-compiler-overview-2. Accessed on March 25, 2022. 78

16. Thomas W Judson. *Abstract Algebra: Theory and Applications*. Virginia Commonwealth University Mathematics, 2021. URL http://abstract.ups.edu/index.html. 12, 17

17. katleman. Source code for BigInteger class. World Wide Web, March 2014. URL https://hg.openjdk.java.net/jdk8/jdk8/jdk/file/tip/src/share/classes/java/math/BigInteger.java. Accessed on March 25, 2022. 70, 72

18. Gary C. Kessler. An overview of cryptography. World Wide Web, March 2022. URL https://www.garykessler.net/library/crypto.html. Accessed on March 25, 2022. 3, 4, 7

19. Bobby Kleinberg. The Miller-Rabin Randomized Primality Test. World Wide Web, May 2010. URL https://www.cs.cornell.edu/courses/cs4820/2010sp/handouts/MillerRabin.pdf. Accessed on March 25, 2022. 49

20. Donald E Knuth. *Art of Computer Programming, Volume 2: Seminumerical Algorithms*. Addison-Wesley Professional, New York, NY, 3rd edition, 1997. 30

21. Majewski Krzysztof. RSA in Java. World Wide Web, November 2021. URL https://www.baeldung.com/java-rsa. Accessed on March 25, 2022. xix, 68

22. Ben Lynn. The Chinese Remainder Theorem. World Wide Web, February 2021. URL https://crypto.stanford.edu/pbc/notes/numbertheory/crt.html. Accessed on March 25, 2022. 18

23. Samuel Neves. Pollard's $(p-1)$ factorization method runtime. World Wide Web, August 2019. URL https://crypto.stackexchange.com/questions/72698/pollards-p-1-factorization-method-runtime. Accessed on March 25, 2022. 40

24. Sebastian Pauli. MAT 112 Integers and Modern Applications for the Uninitiated. World Wide Web, 2019. URL https://mathstats.uncg.edu/sites/pauli/112/HTML/book-1.html. Accessed on March 25, 2022. 16

25. Phayzfaustyn. Symmetric key encryption. World Wide Web, May 2014. URL https://commons.wikimedia.org/wiki/File:Symmetric_key_encryption.svg. Accessed on March 25, 2022. xv, 8

26. Simon Rubinstein-Salzedo. *Cryptography*. Springer International Publishing, Cham, Switzerland, 2018. 3

27. Brian M. Scott. Proof: How many digits does a number have? $\lfloor \log_{10} n \rfloor + 1$. World Wide Web, November 2012. URL https://math.stackexchange.com/questions/231742/proof-how-many-digits-does-a-number-have-lfloor-log-10-n-rfloor-1. Accessed on March 25, 2022. 29

28. DI Management Services. Using the CRT with RSA. World Wide Web, December 2019. URL https://www.di-mgt.com.au/crt_rsa.html. Accessed on March 25, 2022. 59

29. DI Management Services. Rsa Theory. World Wide Web, December 2019. URL https://www.di-mgt.com.au/rsa_theory.html. Accessed on March 25, 2022. 17

30. Simon Singh. *The Code Book: The Science of Secrecy from Ancient Egypt to Quantum Cryptography*. Delacorte Press, New York, NY, 1st edition, 1999. 1, 3, 6, 9, 41

31. Steven S Skiena. *The Algorithm Design Manual*. Springer-Verlag London Limited, London, 2nd edition, 1998. 22

32. N. J. A. Sloane and David W. Wilson. Fermat primes: primes of the form 2^(2^k) + 1, for some k >= 0. World Wide Web, May 2012. URL http://oeis.org/A019434. Accessed on March 25, 2022. 58

33. Spindled. Clock group. World Wide Web, October 2006. URL https://commons.wikimedia.org/wiki/File:Clock_group.svg. Accessed on March 25, 2022. xv, 11

34. Sybren A. Stüvel. Python-RSA documentation. World Wide Web, November 2021. URL https://stuvel.eu/python-rsa-doc/. Accessed on March 25, 2022. 63, 64

35. Suetonius. *Book I: The Deified Julius*. The Lives of the Caesars. Standard Ebooks, July 2021. URL https://standardebooks.org/ebooks/suetonius/the-lives-of-the-caesars/j-c-rolfe/text/book-1. 4, 5

36. Eric W. Weisstein. "Prime Number Theorem". From MathWorld—A Wolfram Web Resource. World Wide Web, November 2005. URL https://mathworld.wolfram.com/PrimeNumberTheorem.html. Accessed on March 25, 2022. 44

37. Eric W. Weisstein. Pollard p-1 factorization method. From MathWorld—A Wolfram Web Resource. World Wide Web, 2022. URL https://mathworld.wolfram.com/Pollardp-1FactorizationMethod.html. Accessed on March 25, 2022. 38

38. Paul Zimmerman. Factorization of RSA-250. World Wide Web, February 2020. URL https://listserv.nodak.edu/cgi-bin/wa.exe?A2=NMBRTHRY;dc42ccd1.2002. Accessed on March 25, 2022. 33, 75