Prime Number Encryption

King's Certificate 2022–23

Mentor

Lloyd Kelly

Authors

Andras Bard   Ataera Eyton

Karolina Nguyen   Calum Wong

Abstract

Prime number encryption is the backbone of modern society. Its applications encompass everything from banking and finance to web browsing and instant messaging. Our article provides an in-depth analysis of prime number encryption, exploring its origins and history and reflecting on its vulnerability to quantum algorithms in the future. We present our findings on the exciting world of prime number encryption in an easily digestible format, and we provide a programmatic implementation of the RSA (Rivest-Shamir-Adleman) cryptosystem in Python to demonstrate how data can be encrypted and decrypted using prime numbers, ensuring confidential communication between two parties.

Acknowledgements

We would like to give our thanks to everyone who helped us on the wonderful journey to writing this paper. We would like to thank Jenny Esslemont, the King's Certificate organiser, for her unending encouragement, and for giving us the opportunity to undertake this research project. We would like to thank Lloyd Kelly, our mentor from Man Group, for his feedback and support, and for helping us edit this article. We would like to thank Dr Peter Shor, the inventor of Shor's Algorithm, for answering some questions we had about the future of prime number encryption. We would like to thank Frederick Beale, Ananya Eshwar, and Joan Pau Jaen Mendoza, the members of the group investigating spring particle systems, for peer reviewing this paper, and for being all-round awesome. Finally, we would like to thank our families and friends for always being there for us, and helping us do our best.

> "Can the reader say what two numbers multiplied together will produce the number 8,616,460,799? I think it unlikely that anyone but myself will ever know." – *William Stanley Jevons, 1874*

Contents

1 Introduction **4**

2 Literature Review **6**

3 Development and Methodology **9**

3.1 Roles 9

3.2 Organisation 9

3.3 Key Concepts 10

3.4 Development Strategy 10

4 Results **11**

4.1 Introduction 11

4.2 Proofs 11

4.2.1 Euclid's Theorem 11

4.2.2 Fermat's Little Theorem 12

4.2.3 RSA Encryption 13

4.3 Python Program 13

4.3.1 Prime Number Generation 13

4.3.2 Key Generation 14

4.3.3 Encryption 14

4.3.4 Decryption 14

4.4 Analysis 14

4.5 Evaluation 15

4.6 Conclusion 15

5 Conclusion **16**

6 Evaluation **17**

6.1 Speed test 17

6.2 Quantum Computers 17

6.3 Elliptic Curves 18

7 Glossary **19**

8 References **20**

Appendix A Python Program **21**

Appendix B Gantt Chart **23**

Appendix C RSA Algorithm Example **24**

1 Introduction

Numbers have been a fundamental cornerstone of human civilisation for millennia. Ever since we first felt the need to count our fish and measure our temples, numbers have been essential to our finance, architecture, communication, medicine, exploration, and innovation [1]. The first numbers we started using were some of the simplest: the *natural numbers*. These are the positive integers obtained by counting $1, 2, 3, \ldots$. Yet even among the seemingly mundane natural numbers, *prime numbers* have fascinated mathematicians for centuries. A prime number has no factors except one and itself, and as a result of this simple definition, prime numbers can be multiplied together in endless combinations to construct all of the other natural numbers, referred to as composite numbers. We will later prove the (perhaps intuitive) theorem that there is an infinite number of prime numbers.

Communication has also been a fundamental cornerstone of human civilisation for millennia. The sharing of information has enabled us to develop unlike any other species on the planet (for better or worse). However, perhaps the greatest problem with sending messages is that they are at risk of being intercepted and read by unintended recipients. To prevent this, people have invented increasingly complex methods of scrambling messages such that they cannot be understood by any unauthorised parties. The field of *cryptography* has developed to facilitate confidential communication between two parties, forever immortalised in academic literature as Alice and Bob [2]. Generally, Alice uses a secret *key* to *encrypt* the *plaintext* she wants to send to Bob, producing a *ciphertext* that is unintelligible to anyone without the key. Upon receiving the ciphertext, Bob can apply the secret key to recover the plaintext. However, this *symmetric-key cryptosystem* relies on a critical fact: that Alice had already shared her key with Bob prior to the entire transaction. This in turn puts the key at risk of being intercepted and read by unintended recipients, jeopardising the entire cryptosystem.

The problem of sharing keys is solved by *public-key cryptography* [3]. In a public-key cryptosystem, two keys are used instead of one. Importantly, the two keys are mathematically linked, such that one of the keys can only be used for encryption, and the other can only be used for decryption. (A useful analogy is a padlock with two keys, where one of the keys can only turn clockwise to lock the mechanism, and the other can only turn the anticlockwise to unlock the mechanism.) The key used for encryption is referred to as the *public key*, because it can freely be distributed to the public, allowing anyone to encrypt messages with it. The key used for decryption is referred to as the *private key*, because it should be kept private, so only its owner can decrypt messages with it. Generally, Alice uses Bob's public key that everyone knows to encrypt the plaintext she wants to send to Bob, producing a ciphertext that is unintelligible to anyone without Bob's private key (i.e. anyone except Bob). Upon receiving

the ciphertext, Bob can apply his private key to recover the plaintext. Note that they did not need to share keys in secret: Bob shared his public key with everyone and his private key with no one. But how do we implement this mechanism? That's where prime numbers make their entrance to the art of cryptography.

In 1978, Ron Rivest, Adi Shamir and Leonard Adleman published the RSA algorithm, a pioneering public-key cryptosystem which generates strong keys using a prime number trapdoor function [4]. (It is relatively easy for a computer to quickly multiply two numbers, but relatively hard for a computer to quickly factorise a number into its prime factors.) This makes it very easy to encrypt messages but extremely difficult to decrypt messages without knowing the private key. This method of encryption is used today in messaging apps such as Telegram to encrypt messages that are sent between devices securely and easily. RSA is also used to encrypt the keys for a symmetric-key cryptosystem facilitating secure Internet connections. With RSA being so important to cryptography and with technology at the center of our world, it is imperative that the system remains secure. In this paper, we will explore how RSA works in detail, analyse why it has worked so well for so long, and speculate how it might be affected by the advent of quantum computers.

This paper provides a deep dive into the RSA cryptosystem — as well as providing our own implementation in Python — and discusses the future of the technology in the age of quantum computing. We hope that this paper contributes to the knowledge of the reader.

2 Literature Review

Prime numbers have been documented by mathematicians as far back as Euclid's *Elements* [5]. The RSA algorithm, published in 1977, marked a seismic shift in the use of prime numbers by moving them from the world of abstract number theory to centre stage at the cornerstone of modern cryptography [4]. We reviewed a range of different literature, ranging from overviews of prime numbers to in depth analyses of the RSA cryptosystem.

We began our research by looking at an overview of prime numbers in general. One of the leading pieces of literature for this is Marcus du Sautoy's *The Music of the Primes* [6]. In this book, du Sautoy describes how mathematicians are searching for a formula to describe the distribution of prime numbers. He documents the research conducted by Bernhard Riemann in the 19th Century, highlighting why the Riemann hypothesis on the distribution of primes remains unsolved to this day. Riemann's research focused on trying to derive a formula for the primes. Chapter 4 of *The Music of the Primes* was especially useful to our research topic, as it helped to explore how graphs could represent the distribution of primes. Moreover, the book elaborates how primes appear in fields ranging from music to nature. The book also explains that there are infinite prime numbers, which can be proven with Euclid's Theorem of Infinite Primes. This book formed the basis of our research when analysing prime numbers; it shows how prime numbers are represented, highlights their importance and explains why we are yet to find a way to describe their exact sequence.

After learning why there is no perfect formula for primes, we decided to investigate how they can still be generated efficiently. We reviewed the proceedings from the *International Workshop on Cryptographic Hardware and Embedded Systems* [7]. In these proceedings, Joye et al. give methods of generating random prime numbers efficiently and efficient ways of checking if numbers are prime. More specifically, their research focuses on creating a new algorithm for generating pseudo-random numbers with no *small* factors. The main limitation of these proceedings was that the theories discussed may be too advanced, given that less efficient but simpler ways to generate prime numbers are readily available. This research did not form the basis of our project, but it provided useful insights into how to efficiently generate prime numbers.

Content with our understanding of prime numbers, we moved on to studying the history of encryption. *The Code Book* by Simon Singh is a book about the history of codes and code-breaking [2]. This book begins with early ciphers, progressing to the history of the RSA cryptosystem in later chapters. This book was very useful to our research, as it lays out the broad history of codes and the mathematics behind them, providing good examples for each cipher. The only limitation of this book was that not all of the ciphers mentioned are relevant to prime numbers. Nevertheless, this book provides an insightful introduction to

RSA encryption and formed the basis of our research.

Before investigating the RSA algorithm, we read a paper titled *New directions in cryptography* by Whitfield Diffie and Martin Hellman [3]. In this paper, Diffie and Hellman explore the revolution in cryptography over the years. Firstly, they define the best known cryptographic problem: privacy - “preventing the unauthorized extraction of information from communications over an insecure channel”. This problem can be solved by sending keys through a secure channel and the main body of the message through the insecure channel. In further sections they consider approaches to transmit key information through public channels. The authors also discuss trapdoor functions, which are easy to compute in one direction, yet difficult to compute in the opposite direction. Diffie and Hellman emphasise throughout the paper that “secrecy is at the heart of cryptography”. The article perfectly showcases the revolutionary perspective on cryptography over the years alongside presenting various methods and implementations. Therefore this paper formed the basis of our research, as it deeply explores the importance of cryptography as well as why it is widely used.

Next, we decided to analyse the original research paper written by the inventors of the RSA algorithm: Ronald Rivest, Adi Shamir and Leonard Adleman [4]. In this article Rivest et al. describe a new cryptosystem that makes it possible to share encryption keys without revealing the corresponding decryption keys. The authors use a “trap-door one-way function” of prime numbers to eliminate the need to transmit encryption keys through a secure messenger, instead allowing them to be shared publicly. Their research focuses on finding such a function, first proposed by Diffie and Hellman, to propose a ground-breaking new cryptosystem [3]. This article was extremely useful to our research topic, as we are researching prime numbers as well as the RSA algorithm. The main limitation of the article was that it was one of the first public key cryptosystems, thus the authors indicated that more research is needed to verify the proposed security of such a system. (Indeed, in the 44 years since the article’s publication, no known methods of defeating the system have been found, if a large enough key is used.) This article formed the basis of our research; it has lots of useful information on how prime numbers can be used to encrypt messages.

Finally, to further deepen our understanding of prime number encryption, we read an educational worksheet by Kathryn Mann, an assistant professor of mathematics at Cornell University [8]. The worksheet details the role of prime numbers in public-key encryption, as well as explaining the mathematics behind the RSA algorithm. It also identifies several problems with symmetric-key encryption, namely the difficulty of rotating keys and the threat of secret keys being intercepted by a third party. We found this worksheet to be an very trustworthy source, as it is intended to educate the reader, and it was recommended to us by our mentor. Overall, this worksheet formed the basis of our research, because we learned a lot of key information from it.

We have successfully examined the significance and main contribution of prime numbers in prime number encryption [2], especially RSA algorithm which we further researched in RSA paper [4]. Prime numbers are the fundamentals of all numbers and cannot be further simplified unlike composite numbers. This was proven to be beneficial as in case of any interception, the key will remain secure as it cannot be easily derived from either public key or encrypted messages [8]. We also looked at generating random prime numbers for the RSA algorithm and made a simple random prime number generator [7]. We have thoroughly looked at how throughout time people have observed and tried to formulate a pattern for the generation of primes [6].

3 Development and Methodology

3.1 Roles

Although we did not explicitly assign roles at the beginning of the project, each member of our group ended up leading some aspect of the research.

- Andras organised the group's meetings, set internal deadlines and co-wrote the program. He also spent most time in perfecting the article and taking care of any correspondence. He made sure to keep the group connected on various collaborative platforms.
- Ataera researched the generation of large prime numbers, implemented an efficient algorithm in Python, and did some research on the history of RSA. He also read the Code Book, which contributed to this article.
- Karolina researched quantum computing to evaluate the effectiveness of RSA encryption, including how Shor's algorithm could be implemented and corresponded with Dr Peter Shor. She also researched Elliptic Curve Cryptography and the Riemann Hypothesis.
- Calum wrote much of the introduction and literature review. He also provided rigorous proofs of Euclid's Theorem of Infinite Primes, Fermat's Little Theorem, and the RSA cryptosystem.

We met every Tuesday after our King's Certificate lesson to plan tasks and conduct research for one hour.

We met every Friday afternoon at King's College London's Franklin-Wilkins Library to work on the Python program, obtain results and typeset this paper in LaTeX for a further three hours.

3.2 Organisation

We created a Gantt chart using Microsoft Excel to organise our work on this project. A snippet of this chart is shown in Figure 1, while the full chart can be found in Appendix B. We communicated with each other via a WhatsApp group to ensure that we consistently met all of our deadlines.

We created a shared notebook using Microsoft OneNote to keep track of academic literature, meeting agendas, success criteria for each stage for the project, and drafts of each section of this paper.

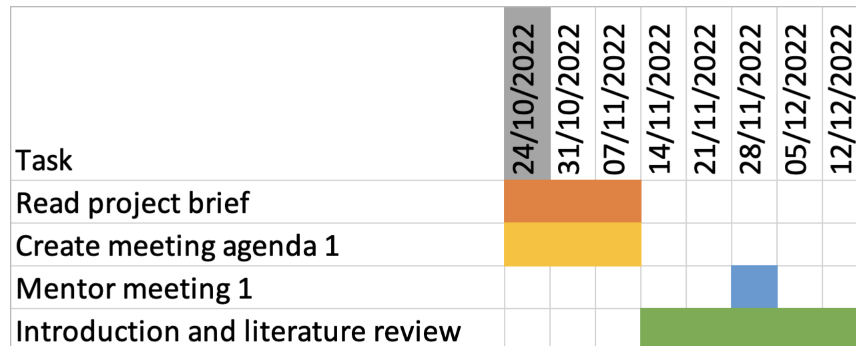


Figure 1: A snippet of the Gantt chart used to organise our work on this project.

We created a shared LaTeX document using Overleaf to collaborate online, aggregate references, and regularly save our progress using version control.

3.3 Key Concepts

As a result of reading literature surrounding the topic we have developed our understanding of the generation of random prime numbers and how the unique properties of prime numbers is what facilitates the security of prime number encryption. Discovering the two prime numbers becomes exponentially more difficult the greater the prime numbers used are.

During our literature review, we learned about what prime numbers are, their significance in encryption, and how to generate them efficiently [7]. Then we focused on the RSA algorithm, investigating how prime numbers can be used to create a one-way trapdoor function [4].

3.4 Development Strategy

To begin, we will prove essential theorems such as Euclid's theorem of infinite primes and Fermat's little theorem, which will form the basis to our prime number generation program. Using the generated primes, we can make a public and private key, and implement them into both encryption and decryption programs to convert back and forth from the plaintext to the ciphertext. We will also discuss any possible limitations of our algorithms and how they might be exploited, for example with quantum computing, and show how Shor's algorithm could be used to do this.

4 Results

4.1 Introduction

Our goal for this section was to demonstrate the effectiveness of the RSA algorithm, and implement the encryption technique using prime numbers we have generated in Python [4]. In order to do this, we first proved some useful theorems: Euclid's Infinitive Prime Theorem, as well as Fermat's Little Theorem. Afterwards, we developed a program which generates random prime numbers with a bit length specified by the number inputted. Once these primes are generated, we can use them in our encryption program, which uses the RSA encryption algorithm, to convert plaintext into ciphertext. Alongside the encryption program, we have designed a corresponding decryption program which will convert ciphertext back into plaintext.

The final Python program can be found in Appendix A. An interactive demo can be found online at Replit, an online program editor.

4.2 Proofs

4.2.1 Euclid's Theorem

Euclid's Theorem of Infinite Primes states that there is an infinite number of prime numbers. This can be proven in the following way:

Consider a list of all prime numbers in increasing order, assuming that there is a finite number of them.

$$p_1, p_2, p_3, \dots p_n$$

Let P be the product of all the prime numbers in this list:

$$P = p_1 p_2 p_3 \dots p_n$$

Let $q = P + 1$.

If q is not prime, then some prime number is a factor of q. However, because q is one more than a multiple of every prime number in our list, it is not divisible by any prime in our list, so it must be divisible by another prime, outside of our list, so our list does not contain all the prime numbers.

If q is prime, then it is also not in our list, as it is higher than the last prime in our list. Therefore, it is shown that our list does not contain all the prime numbers.

By contradiction, it is proven that there is an infinite number of prime numbers.

4.2.2 Fermat's Little Theorem

Fermat's little theorem states that any number a raised to the power of any prime p gives a remainder of a when divided by the prime p.

$$a^p = a \ (mod\ p)$$

This can be proven by induction.

The base case occurs when $a = 1$. We know that

$$1^p = 1 \ (mod\ p)$$

For the inductive hypothesis, assume that $a^p = a \ (mod\ p)$ for some integer a. The goal is to show that $(a+1)^p = a + 1 \ (mod\ p)$.

The left hand side can be expanded by using the binomial theorem:

$$(a+1)^p = a^p + \binom{p}{1} a^{p-1} + \binom{p}{2} a^{p-2} + \cdots + \binom{p}{p-1} a + 1$$

We know that $\binom{p}{k} = \frac{p!}{k!(p-k)!}$. We can see that p divides the numerator, i.e. $p \mid p!$. We can also see that p does not divide the denominator, i.e. $p \nmid k!$ and $p \nmid (p-k)!$. Therefore $p \mid \binom{p}{k}$ for $1 < k < p - 1$.

So the terms in the middle are equal to $0 \ (mod\ p)$, reducing the left hand side to

$$(a+1)^p = a^p + 1 \ (mod\ p)$$

By substituting $a^p = a \ (mod\ p)$ from the inductive hypothesis, we have $(a+1)^p = a + 1 \ (mod\ p)$, which is equal to the right hand side.

By induction, Fermat's Little Theorem holds for all positive integers.

4.2.3 RSA Encryption

Given two prime numbers p and q, and a public key e *co-prime* to $(p-1)(q-1)$, the RSA algorithm computes the private key d such that $ed \equiv 1 \mod (p-1)(q-1)$. The product $n = pq$ is also released with the public key.

During encryption, the ciphertext is computed using $c \equiv m^e \mod n$. During decryption, the plaintext is computed using $m \equiv c^d \mod n$. We will now use Fermat's Little Theorem and modular arithmetic to prove why $m \equiv m^{ed} \mod n$, i.e. why RSA works.

Since $ed \equiv 1 \mod (p-1)(q-1)$, we can write $ed - 1 = h(p-1) = k(q-1)$ for some non-negative integers h and k.

To prove that $m \equiv m^{ed} \mod pq$, it is sufficient to prove that $m \equiv m^{ed} \mod p$ and $m \equiv m^{ed} \mod q$. If $m - m^{ed}$ is divisible by p and it is also divisible by q, then it must be divisible by pq.

If $m \equiv 0 \mod p$, then $m^{ed} \equiv 0 \mod p$. So $m \equiv m^{ed} \mod p$.

If $m \not\equiv 0 \mod p$, then $m^{ed} = m^{ed-1}m = m^{h(p-1)}m = \left(m^{p-1}\right)^h m$.

Fermat's Little Theorem states that $m^p \equiv m \mod p$, so $m^{p-1} \equiv 1 \mod p$.

Using this, $\left(m^{p-1}\right)^h m \equiv 1^h m \equiv m \mod p$.

This proves that $m \equiv m^{ed} \mod p$.

Using similar logic, we can prove that $m \equiv m^{ed} \mod q$.

Therefore $m \equiv m^{ed} \mod n$, so the RSA algorithm works.

4.3 Python Program

4.3.1 Prime Number Generation

First, the function takes a number n which will form the length in bits for the generated prime number. Then, it iterates through a loop checking whether a randomly generated odd number is co-prime with the small prime numbers up to 29, i.e. if it is not divisible by any of the first few prime numbers. If it passes the verification, it moves on to the following branch and iterates through dividing the large random number, "big" by all the prime numbers up to the square root of the number. If it finds no factors of "big", it returns it, and the resulting number should be prime. If, however, it finds a factor, the loop exits and tries a number which is greater than the previous value of "big" by 2. This

is because, the next possible prime number would be 2 spaces up, meaning it would need to be odd, as even numbers share a common factor of 2.

4.3.2 Key Generation

To begin, the function generates two 48-bit prime numbers using the generate prime function (see above), which are assigned to p and q. If p and q are equal, then q is generated again to ensure that p and q are two unique prime numbers. n is set to $p*q$, and ϕ is set to $(p-1)*(q-1)$ which therefore makes ϕ co-prime with n. It then generates a random number e, not higher than ϕ, that is co-prime to ϕ. It then creates d through taking a modular multiplicative inverse of e. This means choosing d such that $ed \equiv 1 \mod \phi$. The public key is (n, e) and the private key is (n, d). The keys are now ready to be used in both encryption and decryption for secure data transfer.

4.3.3 Encryption

Before a message can be encrypted using the RSA algorithm, it needs to be converted to a number. This is accomplished using the encode function, which converts each letter to its corresponding ASCII code. For example: 'h' has the code = 104 (1101000 in binary).

To encrypt the resulting number, we raise it to the power of e modulo n.

4.3.4 Decryption

To decrypt the ciphertext, we raise it to the power of d modulo n.

Before the resulting message can be read, it needs to be converted back to letters. This is accomplished using the decode function, which converts each ASCII code to its corresponding letter.

4.4 Analysis

The Python program is a demonstration of the RSA algorithm which utilises the ideas used in Fermat's little theorem and Euclid's theorem of infinite primes to generate large primes used for private and public keys. This then demonstrates encryption and decryption using the RSA algorithm. The program can quickly generate primes up to 48 bits long.

4.5 Evaluation

Our code has successfully shown the ability to generate large prime numbers (to specified length of bits) which are then used to implement RSA encryption [4]. In order to break RSA encryption; prime factors of a product number must be found to derive the private key, and therefore obtain the original message. This is however rather challenging if the product is significantly large, as it can contain multiple pairs of prime factors which is incredibly time consuming thus inefficient. Nevertheless, prime factorization could be improved in terms of time using Shor's algorithm - developed by Dr Peter Shor [9]. Shor's algorithm transforms an initial possible prime factor guess into a more reliable possible prime factor. In conjunction with quantum computers, it contains enough power to potentially break RSA. Therefore, it does raises concerns and further implications to our project. Nonetheless, as long as generated and implemented prime numbers are relatively large, issues should be temporarily eliminated.

4.6 Conclusion

To summarise, we have proven both Euclid's Infinitive Prime Theorem as well as Fermat's Little Theorem. These theorems were then used in our development of a prime number generation and encryption program. All messages with suitable length (up to 48 bits) are successfully encrypted and decrypted while preserving the original information. The main limitation is the power and efficiency of our program as only message up to 48 bits can be successfully converted. In the future however, we are planning to expand our program so that message of larger length can be passed into the algorithms. One of the processes adapted could be splitting the message into smaller segments of 48 bits (or less) to be encrypted and decrypted separately and concatenated at the end when displaying the original message.

5 Conclusion

Throughout this paper, we developed a deeper understanding of prime numbers, cryptography, RSA encryption, and its dependence on the difficulty of factoring large prime numbers. We researched various topics in relation to cryptography and previously proven theorems, using the applied knowledge to design a Python program to implement the RSA algorithm.

Through reviewing previous work of scholars surrounding the use of RSA, we learned more about their research and gained insight into why it works so well and why it has retained its functionality over time. Understanding of why prime numbers are best suited for encryption was pivotal in understanding why the encryption method remains so practical to this day.

We believe this research will be useful to understand the limitations of RSA and what are the risks as quantum technology advances. To understand this, we studied the process of how modular arithmetic is used in the process of RSA. We also studied how a core part of the effectiveness of RSA is from the unique properties of prime numbers. We achieved this by proving Fermat's last theorem and Euclid's theorem of infinite primes. We investigated Shor's algorithm that aims to factor a number into its primes. We began exploring the use of quantum computers to gain insight into limitations of Shor's algorithm to gain insight into the limitations of RSA.

From gaining understanding of how the algorithm works we managed to produce a Python program that is representative of the process a message undergoes both decrypting and encrypting a message using RSA.

6 Evaluation

6.1 Speed test

Bit length	Average time (s)
32	0.03
40	0.5
48	7

Table 1: Speed test of the Python program in Appendix A.

Table 1 shows a speed test of our final program. The bit length shows the number of bits required to store each prime number. The average time shows the how long the program took to run over 10 tests. The speed test was conducted on an Apple M1 system on a chip with 8 GB RAM.

6.2 Quantum Computers

The main threat to the perfect security offered by RSA encryption is quantum computers implementing Shor's algorithm [9]. This algorithm, developed by Dr Peter Shor in 1994, can be used to factor integers into their prime factor components. This algorithm is highly reliable as it is probabilistic - results have a high probability of success while probability of failure can be lowered by repeating the algorithm. As a hybrid algorithm, it is separated into two parts: reduction of the factoring problem into problem of order finding (can be done on a classical computer), and solving the problem of order finding (algorithm implemented on a quantum computer).

Below is a simplified description of the algorithm:

1. Let the large integer that we want to factor be N.
2. Pick a guess for potential factor, k, which is less than N.
3. Find the greatest common divisor of k and N.
4. If $\gcd(k, N) \neq 1$, then we have found a factor of N; however, if it is equal to 1 then we need to use a quantum subroutine that finds the period, r, such that if $f(x) = k^x \mod N$, then $f(x) = f(x + r)$.*Note: this subroutine implements a quantum algorithm using $log(2N)$ qubits to solve the problem, however we will not discuss further details of how this subroutine works as it is not the main focus of this paper. This algorithm will be slow for a classical computer, but extremely fast for a quantum computer.*

5. If r is odd, then repeat from step 1.
6. If $k^{\frac{r}{2}} = -1 \mod N$, then repeat from step 1.
7. The factors of N are $\gcd(k^{\frac{r}{2}} \pm 1, N)$.

We emailed Dr Shor about the future of quantum computing and prime factorisation. Currently, quantum computers do not have enough qubits to factorise large prime numbers, and thus break the RSA cryptosystem. This is because "quantum gates are too noisy to run full scale fault tolerance on quantum computers". It is really hard to shield quantum gates from noise from the outside world. However, the field of quantum computing is rapidly developing, and it is "reasonably likely" that breakthroughs in the next 20 years could make it possible to execute Shor's Algorithm on quantum computers, although "it is really hard to predict the progress of technology" [10].

6.3 Elliptic Curves

Elliptic curve cryptography (ECC) is a popular public-key cryptosystem. Similarly to RSA, it uses public and private keys and, critically, depends on prime numbers. However, it provides a higher level of security for the same key size, thus making it more effective. Furthermore, unlike RSA, ECC can offer a perfect trapdoor due to the elliptic curve discrete logarithm problem over finite fields. Dr Neal Koblitz is an independent co-creator of elliptic curve cryptography alongside his partner Dr Victor Miller [11]. They suggest using an elliptic curve that is a set of solutions (x, y) for the equation

$$y^2 = x^3 + ax + b$$

Each x value has 2 corresponding y values which are symmetrical about the x-axis, and any line will not intersect the curve in more than 3 points. To begin encryption, a starting point A on a curve is picked. Then, tangents are repeatedly drawn to find new points on the curve.

Overall, ECC is an alternative to RSA encryption, which provides a more perfect trapdoor function, but can be slower to execute than RSA.

7 Glossary

Natural number	A positive integer.
Prime number	A number with no factors other than one and itself.
Co-prime	Two numbers that share no factors other than one.
Cryptography	The field of confidential communication techniques.
Encryption	The process of making data unintelligible to parties without the key.
Decryption	The process of making encrypted data intelligible again using the key.
Plaintext	The original message that can be read by anyone.
Ciphertext	A secret message that can only be read by parties with the key.
Cryptosystem	A specific set of algorithms used for encryption and decryption of data.
Symmetric-key cryptosystem	A cryptosystem that uses the same key for encryption and decryption of data.
Public-key cryptosystem	A cryptosystem that uses different keys for encryption and decryption of data.
RSA	A public-key cryptosystem named after its inventors: Ron Rivest, Adi Shamir and Leonard Adleman.
Python	A high-level, general-purpose programming language created by Guido van Rossum in 1991.
ASCII	The American Standard Code for Information Interchange, used to convert letters to binary numbers for use in computing.
Trapdoor	A function that is easy to compute in one direction, yet difficult to compute in the opposite direction.

8 References

[1] Knott R. *The Development of Number Systems*. Mathematics in School. 1979;8(4):23-5. Available from: https://www.jstor.org/stable/30213485.

[2] Singh S. *The Code Book: The Secret History of Codes and Code-Breaking*. Fourth Estate; 1999.

[3] Diffie W, Hellman M. *New directions in cryptography*. IEEE Transactions on Information Theory. 1976;22(6):644-54. doi:10.1109/TIT.1976.1055638.

[4] Rivest RL, Shamir A, Adleman L. *A Method for Obtaining Digital Signatures and Public-Key Cryptosystems*. Communications of the ACM. 1978;21(2):120-6. doi:10.1145/359340.359342.

[5] Euclid. *Elements*. Alexandria, Ptolemaic Egypt: Euclid; 300 BC. Available from: https://mathcs.clarku.edu/~djoyce/java/elements/toc.html.

[6] du Sautoy M. *The Music of the Primes: Why an Unsolved Problem in Mathematics Matters*. Fourth Estate; 2003.

[7] Joye M, Paillier P, Vaudenay S. Efficient Generation of Prime Numbers. In: Koç ÇK, Paar C, editors. Cryptographic Hardware and Embedded Systems — CHES 2000. Berlin, Heidelberg: Springer Berlin Heidelberg; 2000. p. 340-54. doi:10.1007/3-540-44499-8_27.

[8] Mann KP. *The science of encryption: prime numbers and mod n arithmetic*. The Atlantic; 2011. Accessed: 2022-12-06. https://math.berkeley.edu/~kpmann/encryption.pdf.

[9] Shor PW. *Algorithms for quantum computation: discrete logarithms and factoring*. In: Proceedings 35th Annual Symposium on Foundations of Computer Science; 1994. p. 124-34. doi:10.1109/SFCS.1994.365700.

[10] Shor PW. Email correspondence; 2023.

[11] Koblitz N. *Elliptic Curve Cryptosystems*. Mathematics of Computation. 1987;48(177):203-9. Available from: https://www.jstor.org/stable/2007884.

Appendix A Python Program

```
'''
Python program to implement RSA encryption.
By Andras Bard, Ataera Eyton, Karolina Nguyen, and Calum Wong.
King's Certificate 2022-23.
'''

import random
import math

def _encode(message):
    number = 0
    for x in message:
        number = number << 8
        number += ord(x)
    return number

def _decode(number):
    message = ""
    while number > 0:
        x = number % 256
        message = chr(x) + message
        number = number >> 8
    return message

def _fast_exponent_modulo(a, b, n):
    return pow(a, b, mod=n)

def _generate_prime(n):
    small_primes = 2*3*5*7*11*13*17*19*23*29
    large = 2
    while math.gcd(large, small_primes) != 1:
        large = random.randint((2**(n-1)), 2**n)
    while True:
        isPrime = True
        for i in range(3, (int(round(large ** 0.5))) + 1, 2):
            temp = large % i
            if temp == 0:
                isPrime = False
                break
        if isPrime == True:
            return large
        else:
            large = large + 2
```

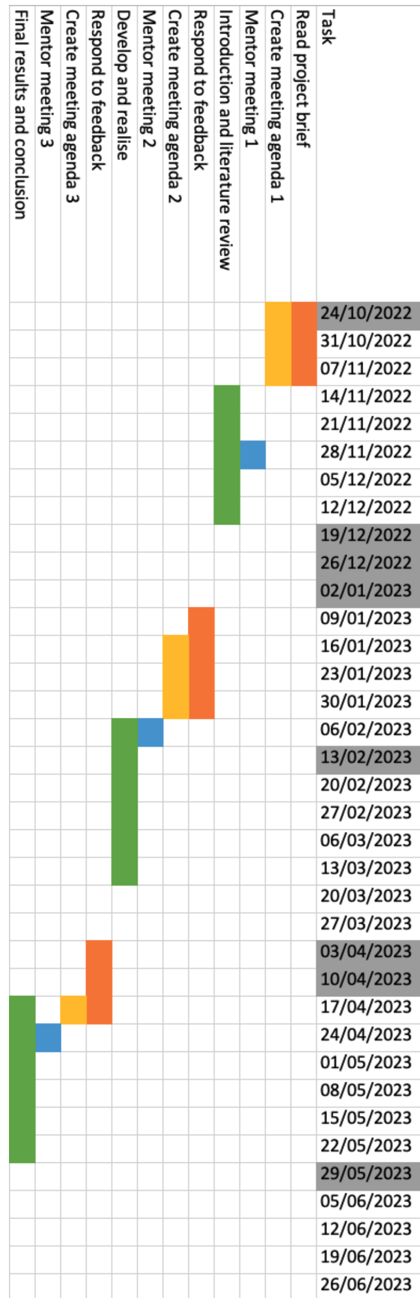
```python
def generate_keypair():
    p = _generate_prime(48)
    q = _generate_prime(48)
    while p == q:
        q = _generate_prime(48)
    n = p * q
    phi = (p-1) * (q-1)
    e = random.randrange(1, phi)
    while math.gcd(e, phi) != 1:
        e = random.randrange(1, phi)
    d = _fast_exponent_modulo(e, -1, phi)
    public_key = (n, e)
    private_key = (n, d)
    return public_key, private_key

def encrypt(plaintext, public_key):
    n, e = public_key
    m = _encode(plaintext)
    if m >= n:
        raise ValueError('Plaintext is too long')
    c = _fast_exponent_modulo(m, e, n)
    ciphertext = c
    return ciphertext

def decrypt(ciphertext, private_key):
    n, d = private_key
    c = ciphertext
    m = _fast_exponent_modulo(c, d, n)
    plaintext = _decode(m)
    return plaintext

if __name__ == '__main__':
    public_key, private_key = generate_keypair()
    ciphertext = encrypt('PRIME NUMBER', public_key)
    print(ciphertext)
    plaintext = decrypt(ciphertext, private_key)
    print(plaintext)
```

Appendix B Gantt Chart



Appendix C RSA Algorithm Example

This is a worked example of the RSA algorithm at work with small primes.

First, we choose 2 prime numbers. We will use $p = 3$ and $q = 5$. Next, we compute $n = p \times q = 15$, and $\phi = (p-1) \times (q-1) = 8$. Next, we choose a number e co-prime with ϕ. We choose $e = 7$.

Next, we find a number d such that $ed = 1 \mod n$. In other words, $7d$ should give a remainder of 1 when divided by 8. The lowest such number d is 7, as $7 \times 7 = 49$, which is 1 more than $48 = 8 \times 6$.

We have now generated the public key (n, e) and the private key (n, d). At this stage, p, q and ϕ are no longer needed and should be destroyed to eliminate the risk of attackers using them to find the private key. The public key can be released to the public, and the private key must be kept secret.

We can now use the public key to encrypt plaintext. We will encrypt the letter H. First, we convert it to a number by finding its position in the alphabet. Our plaintext becomes the number 8.

Next, we encrypt this number by calculating $8^e \mod n = 8^7 \mod 15 = 2$. Note that we only used the public key (n, e), meaning this step can be performed by anyone.

At this point, the ciphertext 2 can be transmitted to the intended recipient.

We can now decrypt this number by calculating $2^d \mod n = 2^7 \mod 15 = 8$, a familiar number. Note that we only used the private key (n, d), meaning this step can only be performed by the intended recipient.

Next, we convert the number back into a letter by finding the 8th letter of the alphabet. Our resulting plaintext is the letter H. We have successfully recovered the original plaintext.

Note that the algorithm for encryption and decryption is very similar, with exponents of e and d used respectively.