Mathematics Behind the RSA Cryptosystem

1 Introduction

Cryptography is the technique of using codes to hide information, which is greatly needed for computers to be able to transmit information securely, and one of the most popular cryptosystem computers use is the Rivest–Shamir–Adleman (RSA) cryptosystem (Mohamad et al., 2021). Barge (2019) described the RSA Cryptosystem as an asymmetric encryption algorithm, that is 2 different keys are used to encrypt and the decrypt a message.

I first encountered the RSA cryptosystem when I generated RSA keys for my home server, so I can authenticate myself when I want to connect to it via the internet. When I generated the keys, it took only a few seconds to do so. This made me quite worried, my server contained very sensitive information, how secure are these keys if they took only a few seconds to generate? And so, I wanted to explore more regarding the security of the RSA cryptosystem.

Modular arithmetic is a fundamental branch of Number Theory that deals with integers where after the number reaches a certain value, called the modulo, it reverts back into zero. For example 11 is equivalent to 2, modulo 9, which is written as:

$$11 \equiv 2 \pmod{9} \tag{1}$$

When counting from 0 to 11 on the left-hand side and the value reaches 9, it will revert back to 0 and continue counting, which lands at 2.

Modular arithmetic shows itself in the RSA cryptosystem in its use of modular exponentiation, where a base, b, is raised to the power of 2 numbers, p and q, and will be equivalent to the original base, b, modulo some number n:

$$b^{pq} \equiv b \pmod{n} \tag{2}$$

In this extended essay, I aim to explore the underlying number theory that forms the basis of the RSA cryptosystem, focusing on the role of modular arithmetic and prime numbers as the security of the RSA cryptosystem relies on the difficulty of prime factorisation. And then, I will evaluate some methods of prime factorisation that can pose a threat to the security of the RSA cryptosystem.

2 Background Information

2.1 Public Key Cryptography

A public key/asymmetric cryptosystem (Hellman, 2002) has 2 keys: a public key and a private key, where the 2 keys are used in one-way functions that are inverse of each other: where one key can only be used to encrypt a message, which is transforming plaintext into ciphertext, and the other can only be used to decrypt a message.

In public key cryptography, there are 3 parties involved Alice, the person who wishes to send a message; Bob, the person who receives a message; and Eve, a potential eavesdropper when Alice and Bob and communicating as Alice and Bob are unable to find a secure channel for communication. The steps for public-key cryptography are as follows:

1. Bob generates a public and a private key
2. Bob sends his private key to whoever wishes to communicate with him, and hence, both Alice and Eve will have the public key
3. Alice uses Bob's private key to encrypt her message and sends the ciphertext to Bob, which Eve can see.
4. Bob receives The ciphertext and decrypts it using his private key to obtain Alice's message, while Eve, only having Bob's public key, will be unable to decrypt the ciphertext as the private key is difficult to deduce from just the private key.

With public key cryptography, Alice and Bob are able to communicate without any eavesdroppers reading their messages. Alongside this, Alice and Bob didn't need to hold any agreed upon information before their exchange, great for computers because communications via the internet are usually one-time and between 2 completely unknown parties.

Public key cryptography can also be used in a reverse manner, where Alice and Bob can agree upon a pre-generated public key, held by Alice, and a private key, held by

Bob. Bob can use his private key to encrypt a message, which Alice can decrypt using her public key, confirming to Alice that Bob has the corresponding private key (Kamiński & Mazurczyk, 2023). This is mainly used as a form of authentication, which is how I used the RSA cryptosystem in my server, to authenticate any computer who wishes to connect to my server.

2.2 Pre-requisite Mathematics

2.2.1 Number Theory

Before we can discuss any of the underlying mathematics of the RSA cryptosystem, we must define a few things first.

Definition 2.2.1 (Divides)**.** *For $a, b \in \mathbb{Z}$ we can say that a divides b, written as $a \mid b$, if $aQ = b$ for $Q \in \mathbb{Z}$*

For example, $3 \mid 12$ because $3 \times 4 = 12$. We can also interpret this as when dividing 12 by 3, it leaves no remainder, and hence, 3 divides 12.

Definition 2.2.2 (Greatest Common Divisor, $gcd()$)**.** *The greatest common divisor, or GCD, of 2 numbers, $n, m \in \mathbb{Z}$ is defined as:*

$$gcd(n, m) = max\{x \in \mathbb{Z} : x \mid n \ and \ x \mid m\} \tag{3}$$

Definition 2.2.3 (Coprimality)**.** *Two numbers; $n, p \in \mathbb{Z}$ are considered **coprime** or **relatively prime**, if:*

$$gcd(n, p) = 1 \tag{4}$$

Coprimality can also be seen as 2 integers not sharing any prime factors. For example, 63 and 50 are coprime because $gcd(63, 50) = 1$ or that 63 and 50 don't share any prime factors as $63 = 3^2 \times 7$ and $50 = 2 \times 5^2$.

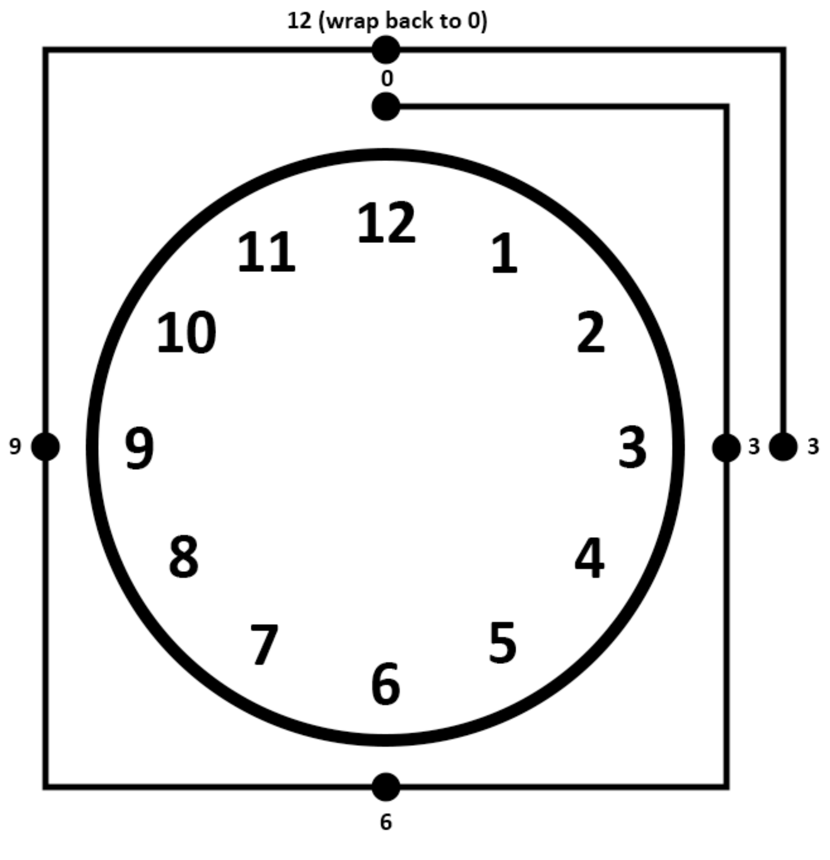
2.2.2 Modular Arithmetic

As explained before, modular arithmetic is a system for counting integers where after reaching a certain value called the modulo, the numbers 'wraps around' back to zero (Stein, 2009).

Modular arithmetic is also known as clock arithmetic because clocks use modular arithmetic for counting. When counting up hours on a clock, numbers wrap around back to 0 after reaching 12.

Figure 1

Modular Arithmetic - Clock Arithmetic



Note. Figure created by Author

For example in Figure 1, counting up 15 hours starting at 12 o'clock, we count from 0 up to 12, where we wrap back to 0 and continue counting, landing at 3.

In this system, 2 integers are said to be equivalent if after counting up to each of those integers and reverting 0 once reaching the modulo, they reach the same number. From the previous clock example, we can see that

$$15 \equiv 3 \mod 12 \tag{5}$$

Another example would be

$$15 \equiv 26 \mod 11 \tag{6}$$

As when counting up to both 15 and 26, wrapping back to 0 after reaching 11, results in the same number, 4.

Modular Arithmetic can also be viewed as comparing the remainder of 2 numbers when divided by the modulo. An example is as follows:

$$16 \equiv 27 \pmod{11} \tag{7}$$

16, when divided by 11, has a remainder of 5, and so does 27 when divided by 11. Because they both have a remainder of 5 when divided by 11, they are equivalent modulo 11.

However, if one side of the modular equivalence is less than the modulo like in this example:

$$16 \equiv 5 \pmod{11} \tag{8}$$

this relationship can also be expressed in an equation without any modular arithmetic:

$$16 = Q(11) + 5 \tag{9}$$

where $Q \in \mathbb{Z}$. This means that 16 can be expressed as some multiple of 11, plus 5.

Definition 2.2.4. *For 3 integers $p, r, n \in \mathbb{Z}$, if $p \equiv r \pmod{n}$ and $r < n$, then*

$$p = Qn + r \tag{10}$$

where $Q \in \mathbb{Z}$

Modular reduction is a more specific operation relating to modular arithmetic. It's an operation that takes in 2 values and returns the lowest positive integer congruent to one of the values, modulo the other value (R. Omondi, 2020)

Definition 2.2.5 (Modular Reduction)**.** *For 2 numbers $a, n \in \mathbb{Z}$*

$$a \bmod n = min\{x > 0 \mid a \equiv x \pmod{n}\} \tag{11}$$

For example, 32 mod 7 = 4, as although there are other integers that 32 is equivalent to, modulo 7, like 11 and 18 as $32 \equiv 11 \pmod{7}$ and $32 \equiv 18 \pmod{7}$, 4 is the smallest positive integer equivalent to 32, modulo 7. Modular reduction can also be viewed as directly taking the remainder of a number when divided by the modulo as 32 divided by 7 has a remainder of 4.

2.2.3 Equivalences Modulo n

With these unique properties of modular arithmetic, manipulating equivalences, modulo n, have many constraints that regular equations don't have.

Proposition 2.2.6 (Cancellation)**.** *For four numbers $a, b, c, n \in \mathbb{Z}, b > a$, it holds that:*

$$a \equiv b \pmod{n} \Leftrightarrow ac \equiv bc \pmod{n} \tag{12}$$

if $gcd(c, n) = 1$

For example, we have the equivalence $20 \equiv 2 \pmod{9}$ which is the same as $2 \times 10 \equiv 2 \times 1 \pmod{9}$, we can divide both sides of the equivalence by 2 resulting $10 \equiv 1 \pmod{9}$ and the equivalence still holds as $gcd(2, 9) = 1$.

However, for the equivalence $10 \equiv 4 \pmod{6}$, if we divide both sides of the equivalence by 2, the equivalence will not hold as $5 \not\equiv 2 \pmod{6}$, which is because $gcd(4, 6) \neq 1$.

Proposition 2.2.7. *For 5 numbers $a, b, c, d, n \in \mathbb{Z}$, where $a \equiv b \pmod{n}$ and $c \equiv d \pmod{n}$. It holds that*

$$ac \equiv bd \pmod{n} \tag{13}$$

For example, we have 2 equivalences $10 \equiv 1 \pmod{3}$ and $8 \equiv 2 \pmod{3}$, both of which are true. Because they have the same modulo (which is 3), we can multiply the equivalences together and obtain an equivalence that is still true:

$$\begin{aligned} 10 \times 8 &\equiv 1 \times 2 \pmod{3} \\ 80 &\equiv 2 \pmod{3} \\ 26(3) + 2 &\equiv 2 \pmod{3} \end{aligned}$$

Proposition 2.2.8. *For 2 numbers $a, n \in \mathbb{Z}$ where a is coprime with n, that is $gcd(a, n) = 1$, we can say that $a \bmod n$ is also coprime with n:*

$$gcd(a \bmod n, n) = 1 \tag{14}$$

For example, 40 is coprime with 7 as $gcd(40, 7) = 1$, however if we reduce 40 modulo 7, it results in 5 ($40 \bmod 7 = 5$) which is also coprime with 7.

2.2.4 Euler's Totient Function

Euler's Totient Function, denoted by ϕ, takes in a number $n \in \mathbb{Z}$ and returns the number of positive integers less than n that are coprime with n (Stein, 2009).

Definition 2.2.9 (Euler's Totient Function, ϕ)**.** *For any number $n \in \mathbb{Z}$*

$$\phi(n) = n\{x : 0 < x < n \wedge gcd(n, x) = 1\} \tag{15}$$

For example, when finding $\phi(10)$ we can list down the greatest common divisor of 10 and all numbers below 10:

$$\begin{array}{lll}
\underline{gcd(1,10)} = 1 & gcd(2,10) = 2 & \underline{gcd(3,10)} = 1 \\
gcd(4,10) = 2 & gcd(5,10) = 5 & gcd(6,10) = 2 \\
\underline{gcd(7,10)} = 1 & gcd(8,10) = 2 & \underline{gcd(9,10)} = 1
\end{array}$$

Because there are 4 numbers that are coprime with 10 that are less than 10: 1, 3, 7, and 9; this tells us that $\phi(10) = 4$

Euler's Totient Function has an interesting relationship with prime numbers. Prime numbers are defined as integers whose prime factors are only 1 and itself (Stein, 2009). Because of this, a prime number will always be coprime with all integers smaller than itself as these two numbers will not share any prime factors greater than 1. For example, We can take the prime number 7 and list the greatest common divisor of 10 and all numbers below 7:

$$\begin{array}{lll}
\underline{gcd(1,7)} = 1 & \underline{gcd(2,7)} = 1 & \underline{gcd(3,7)} = 1 \\
\underline{gcd(4,7)} = 1 & \underline{gcd(5,7)} = 1 & \underline{gcd(6,7)} = 1
\end{array}$$

As we can see, **all** numbers below 7 are coprime with 7, and hence, $\phi(7) = 7 - 1 = 6$

Lemma 2.2.10. *for any prime number n,*

$$\phi(n) = n - 1 \tag{16}$$

Alongside this, Euler's totient function is also distributive over multiplication (Stein, 2009)

Lemma 2.2.11. *for any number $a, b \in \mathbb{Z}$ where a and b are coprime,*

$$\phi(a \times b) = \phi(a) \times \phi(b) \tag{17}$$

For example, $\phi(3) = 2$ and $\phi(4) = 2$, Therefore:

$$\begin{aligned} &\phi(12) \\ =&\phi(3 \times 4) \\ =&\phi(3) \times \phi(4) \\ =&2 \times 2 \\ =&4 \end{aligned}$$

Which is true because when laying out the GCD of all numbers below 12:

$$\begin{array}{ccc} \underline{gcd(1,12)} = 1 & gcd(2,12) = 2 & gcd(3,12) = 3 \\ gcd(4,12) = 4 & \underline{gcd(5,12)} = 1 & gcd(6,12) = 6 \\ \underline{gcd(7,12)} = 1 & gcd(8,12) = 4 & gcd(9,12) = 3 \\ gcd(10,12) = 2 & \underline{gcd(11,12)} = 1 & \end{array}$$

There are only 4 numbers below 12 that are coprime with 12.

Both these properties of Euler's totient function will be extremely important when discussing the RSA cryptosystem.

2.2.5 Euler's Totient Theorem

The definition and proof for Euler's Totient Theorem is as follows (Stein, 2009).

Theorem 2.2.12 (Euler's Totient Theorem)**.** *For 2 numbers $a, n \in \mathbb{Z}$ where a and n are coprime, that is $gcd(a, n) = 1$, it holds that*

$$a^{\phi(n)} \equiv 1 \pmod{n} \tag{18}$$

Proof. Lets define a set S that contains all the integers coprime to n:
$S = \{x \mid 0 < x < n \wedge gcd(n, x) = 1\} = \{x_1, x_2, x_3, \ldots, x_{\phi(n)}\}$. All the elements of S are coprime to n and distinct, and therefore, are incongruent with each other, modulo n.

Then lets define another set, T, that contains all the elements of S multiplied by a:
$T = \{ax_1, ax_2, ax_3, \ldots, ax_{\phi(n)}\}$. Since we only multiplied all the elements of S by a, by Proposition 2.2.6, all the elements in T are still incongruent with each other, modulo n, and hence, are distinct. Alongside this, since a is coprime with n, all the elements of T are still coprime with n.

Then, we can reduce all the elements in T modulo n:
$T\prime = \{ax_1 \bmod n, ax_2 \bmod n, ax_3 \bmod n, \ldots, ax_{\phi(n)} \bmod n\}$. All of these elements are also still incongruent with each other, modulo n, and hence, are distinct. The elements of $T\prime$ are also still coprime with n by Proposition 2.2.8 as any number that is coprime with n when reduced modulo n, will still result in a number coprime with n. Another effect of being reduced modulo n is that all the elements are all positive integers that are less than n, as any number greater than n can still be reduced to a smaller number. Because of these properties of $T\prime$ where:

1. all elements are distinct (the number of elements in $T\prime$ is the same in S)

2. all elements are positive integers less than n

3. all elements are coprime with n

we can say that $T\prime = S$. For example, if we take $a = 3$ and $n = 10$, we can place all integers coprime with n into a set S:

$$S = \{1, 3, 7, 9\}$$

Multiplying all elements by a to get the set T:

$$T = \{1(a), 3(a), 7(a), 9(a)\}$$
$$T = \{1(3), 3(3), 7(3), 9(3)\}$$
$$T = \{3, 9, 21, 27\}$$

And then, if we reduce all elements of T modulo n to get $T\prime$:

$$T\prime = \{3 \bmod n, 9 \bmod n, 21 \bmod n, 27 \bmod n\}$$
$$T\prime = \{3 \bmod 10, 9 \bmod 10, 21 \bmod 10, 27 \bmod 10\}$$
$$T\prime = \{3, 9, 1, 7\}$$
$$T\prime = \{1, 3, 7, 9\}$$

Which is the exact same as S.

Because of this, we can write $\phi(n)$ many equivalences modulo n:

$$x_1 \equiv ax_1 \pmod n$$
$$x_2 \equiv ax_2 \pmod n$$
$$x_3 \equiv ax_3 \pmod n$$
$$\ldots$$
$$x_{\phi(n)} \equiv ax_{\phi(n)} \pmod n$$

which can all be combined into one equivalence due to Proposition 2.2.7 by multiplying all the equivalences with each other:

$$\begin{aligned} ax_1 \times ax_1 \times ax_1 \times \cdots \times ax_{\phi(n)} &\equiv x_1 \times x_1 \times x_1 \times \cdots \times x_{\phi(n)} \pmod{n} \\ a^{\phi(n)}(x_1 \times x_1 \times x_1 \times \cdots \times x_{\phi(n)}) &\equiv x_1 \times x_1 \times x_1 \times \cdots \times x_{\phi(n)} \pmod{n} \end{aligned}$$

and by Proposition 2.2.6, we can divide both sides of the equivalence by $(x_1 \times x_1 \times x_1 \times \cdots \times x_{\phi(n)})$ as the product is coprime with n, resulting in

$$a^{\phi(n)} \equiv 1 \pmod{n} \tag{19}$$

proving Euler's Totient Theorem. □

For example, let's take 2 numbers that are coprime: 21 and 5. $\phi(5)$ would be 4. If we took 21 and raised it to $\phi(5)$, we would get:

$$\begin{aligned} &21^{\phi(5)} \\ &= 21^4 \\ &= 194481 \\ &= 38896(5) + 1 \end{aligned}$$

which shows that $21^{\phi(5)} \equiv 1 \pmod{5}$

3 RSA Cryptosystem

Rivest et al. (1978) describes the RSA cryptosystem in their paper that publicised this cryptosystem as a public key cryptosystem that relies on the difficulty of factoring large numbers as its difficulty of deriving the private key from a public key, and hence, its security. The RSA cryptosystem uses modular arithmetic for its encryption and decryption operations. Let's say Alice wishes to send Bob a message $m \in \mathbb{Z}$, the encryption operation would be:

$$m^e \bmod n = c \tag{20}$$

where $e, n, c \in \mathbb{Z}$, e is the encryption exponent, n is the modulo, and c is the resulting ciphertext generated after the encryption operation. The encryption operation would raise the message by encryption exponent and reduce it modulo n to generate the ciphertext to be sent. This means that Bob's public key must contain e and n in order for Alice to generate the ciphertext.

On the receiver's side, the corresponding decryption operation must then be:

$$c^d \bmod n = m \tag{21}$$

Where $d \in \mathbb{Z}$ is the decryption exponent. The decryption algorithm takes the ciphertext, raises it to the d-th power and reduces it modulo n, and it should result in the original message m. This shows that Bob's private key is d, the decryption exponent.

We can substitute c with the modulo in the first equation, granting:

$$\begin{aligned}(m^e \bmod n)^d \bmod n &= m \\ m^{ed} \bmod n &= m\end{aligned}$$

And hence, for Bob to receive Alice's message, the following equivalence must be satisfied:

$$m^{ed} \equiv m \pmod{n} \tag{22}$$

The message raised to both the encryption and decryption exponents must be equivalent to the same message modulo n, where it must be difficult to obtain the decryption exponent, d, which is the private key, from just the encryption exponent, e, and the modulo, n, and somehow, it must also be easy for Bob to generate all these values.

3.1 Proof of Correctness

At the heart of the RSA cryptosystem lies Euler's Totient Theorem (Barge, 2019). The base in Euler's Totient Theorem is the message being sent:

$$m^{\phi(n)} \equiv 1 \pmod{n} \tag{23}$$

We can multiply both sides of the equivalence by itself an arbitrary number of times $k \in \mathbb{Z}$:

$$\begin{aligned} m^{\phi(n)} \times m^{\phi(n)} \times \cdots \times m^{\phi(n)} &\equiv 1 \times 1 \times \cdots \times 1 \pmod{n} \\ (m^{\phi(n)})^k &\equiv 1^k \pmod{n} \\ m^{k\phi(n)} &\equiv 1 \pmod{n} \end{aligned}$$

And since one of the conditions of Euler's Totient Theorem is that the base, in this case m, is coprime with n, we can multiply both sides of the equivalence by m, resulting in

$$\begin{aligned} m \times m^{\phi(n)} &\equiv m \pmod{n} \\ m^{\phi(n)+1} &\equiv m \pmod{n} \end{aligned}$$

This lines up with the equivalence we established in the intro for the RSA algorithm (Equation 22), which means that we can form the equation:

$$k\phi(n) + 1 = ed \tag{24}$$

We can also express ed as a modular equivalence because from this, ed is some multiple of $\phi(n)$ plus 1:

$$ed \equiv 1 \pmod{\phi(n)} \tag{25}$$

We do this so that we can generate d after we found our $\phi(n)$ and picked our e, which is done via the Extended Euclidean Algorithm, which could find an unknown in a modular equivalence (Naranjo et al., 2010). The algorithm for which can be found in Appendix A as Figure 3 taken from the article "Euclidean algorithms (Basic and Extended)", 2015.

Now, let's take a closer look into our $\phi(n)$. This $\phi(n)$ must be difficult to compute from n as n is part of the public key and $\phi(n)$ is used to calculate the private key, d, using the Extended Euclidean Algorithm explained before.

Let's say n is a prime number, then our $\phi(n) = n - 1$ by Lemma 2.2.10, which would be very easy for anyone with n to compute. However, let's say that n is a product of 2 prime numbers p and q, then $\phi(n) = \phi(pq) = \phi(p) \times \phi(q) = (p-1) \times (q-1)$. For example, let's say $p = 11$ and $q = 13$, hence $n = 143$. This means that

$$\begin{aligned}
&\phi(143) \\
&= \phi(11) \times \phi(13) \\
&= (11-1) \times (13-1) \\
&= 120
\end{aligned}$$

This configuration of n makes it far more difficult for a computer to obtain $\phi(n)$ from just n. Computers could either manually count all numbers below n that are coprime, which takes an extremely long time for large values of n, or they could try to prime factorise n to calculate $\phi(n)$ the same way it was generated:
$\phi(n) = \phi(pq) = \phi(p) \times \phi(q)$. The latter method would also take an extremely long time for very large value of n as computers are terrible at prime factorising large intergers, which will be shown in Section 3.3.

Now that we have a way to ensure that $\phi(n)$ is difficult to obtain from n, Alice can generate her encryption and decryption exponents.

3.2 Algorithm

For Bob to generate her private and public keys (Mansour, 2017):

1. Bob generates 2 large prime numbers p and q and calculate $n = pq$.

   For example, Bob sets $p = 11$ and $q = 13$ meaning her $n = 11 \times 13 = 143$.

2. Then, he calculates his $\phi(n) = (p - 1)(q - 1)$.

   Continuing our previous example, this means that $\phi(143) = (13 - 1)(11 - 1) = 120$

3. Bob generates his public exponent, e, such that e is coprime with $\phi(n)$. Usually this exponent is small, so that encryption operations are faster to execute.

   For this example, Bob takes e to be 7 as it is small and coprime with 120

4. Then the private key is generated using the 'Extended Euclidean Algorithm', an algorithm used to find unknowns in modular equivalences, because e, d, and $\phi(n)$ satisfy

   $$ed \equiv 1 \pmod{\phi(n)}$$

   In this example, when we use the Extended Euclidean Algorithm where $e = 7$ and $\phi(n) = 120$, Bob's $d = 103$

Now, Bob has a public key, which would be $e = 7$ and $n = 143$, and a private key, which would be $d = 103$

Bob then sends out her public key to Alice. If Alice wishes to send a message, m, to Bob, Alice would only need to calculate his ciphertext, c, using:

$$c = m^e \bmod n \tag{26}$$

Let's say Alice wishes to send Bob the message '`hello`', to do so, Alice needs to *encode* her message, which means that her message needs to be converted into a string of numbers. To do so, Alice can use a character encoding standard like ASCII, which assigns each character with a binary number.

Table 1

ASCII Table of Lowercase Letters with Decimal and Binary Codes

Character	**Decimal**	**Binary**	**Character**	**Decimal**	**Binary**
a	97	01100001	b	98	01100010
c	99	01100011	d	100	01100100
e	101	01100101	f	102	01100110
g	103	01100111	h	104	01101000
i	105	01101001	j	106	01101010
k	107	01101011	l	108	01101100
m	109	01101101	n	110	01101110
o	111	01101111	p	112	01110000
q	113	01110001	r	114	01110010
s	115	01110011	t	116	01110100
u	117	01110101	v	118	01110110
w	119	01110111	x	120	01111000
y	121	01111001	z	122	01111010

We can convert each character in ’`hello`’ into a string of binary numbers:

$$\texttt{h} = 01101000_2 \qquad \texttt{e} = 01100101_2 \qquad \texttt{l} = 01101100_2$$

$$\texttt{l} = 01101100_2 \qquad \texttt{o} = 01101111_2$$

However, if we combine all the binary numbers into one and convert it into only 1 decimal number we get:

$$01101000\ 01100101\ 01101100\ 01101100\ 01101111_2$$

$$=448377742703_{10}$$

This is a very large number and RSA can only encrypt a number less than n or else the message will be reduced modulo n. Despite this, we can just send Alice multiple messages, each message containing 1 character:

$$m_1 = 01101000_2 = 104_{10} \qquad m_2 = 01100101_2 = 101_{10} \qquad m_3 = 01101100_2 = 108_{10}$$
$$m_4 = 01101100_2 = 108_{10} \qquad m_5 = 01101111_2 = 111_{10}$$

Converting each message into ciphertext,

$$c_1 = 104^7 \bmod 143 = 91 \qquad c_2 = 101^7 \bmod 143 = 62 \qquad c_3 = 108^7 \bmod 143 = 4$$
$$c_4 = 108^7 \bmod 143 = 4 \qquad c_5 = 111^7 \bmod 143 = 45$$

Then Alice sends Bob his ciphertexts.

With Alice's ciphertexts, Bob can decrypt her message with his private key by using:

$$m = c^d \bmod n \tag{27}$$

and so,

$$\begin{aligned} m_1 &= 91^{103} \bmod 143 & m_2 &= 62^{103} \bmod 143 & m_3 &= 4^{103} \bmod 143 \\ &= 104 & &= 101 & &= 108 \\ m_4 &= 4^{103} \bmod 143 & m_5 &= 45^{103} \bmod 143 \\ &= 108 & &= 111 \end{aligned}$$

Converting each message back into characters:

$$m_1 = 104_{10} = \text{'}\texttt{h}\text{'} \qquad m_2 = 101_{10} = \text{'}\texttt{e}\text{'} \qquad m_3 = 108_{10} = \text{'}\texttt{l}\text{'}$$
$$m_4 = 108_{10} = \text{'}\texttt{l}\text{'} \qquad m_5 = 111_{10} = \text{'}\texttt{o}\text{'}$$

Decoding the numbers back into text we get '`hello`', which is the exact same as Alice's original message! Which shows that the RSA cryptosystem can be used to send messages between Alice and Bob.

3.3 Difficulty of Prime Factorisation

3.3.1 Brute Force

Since there are no easily discernible ways to know the prime factors of a number, a naive way to prime factorise is to individually check prime numbers whether they are prime factors of n (Riesel, 2013). Let's say a computer already has a list of prime numbers stored in a file, the computer can then individually check each prime number, starting from 2, on whether it is a prime factor of n. The following is an example algorithm in C++ that I have written that takes in a number n and individually checks whether n is divisible by a prime factor taken from a list of pre-calculated primes:

Figure 2

Brute Force Prime Factorisation Implementation in C++

```cpp
void primeFactorise(int n, int* primes[])//take the number to be prime
                                        //and the list of prime numers
                                        //as arguments
{
    vector<int> factors; //Declaring a list that stores the
                         //prime factors of n
    int i = 0; //Counter variable
    while(n>1) //Keep repeating until n = 1
    {
        if(n%primes[i] == 0) //Conditional asking whether primes[i] is a
                             //factor of n
        {
            factors.push_back(primes[i]); //add primes[i] to the list
                                          //of factors
            n = n / primes[i];
        }
        i++; //increment i to check the next prime number
    }
}
```

Note. Code written by Author

The problem lies, however, in the generation of the list of prime numbers that the algorithm from Figure 2 uses. For this, I used a separate algorithm found in Appendix

B. I generated multiple files that contained prime numbers below 1,000 to prime numbers bellow 1,000,000,000 and then measured the time it took to generate along with the size of the file generated.

Table 2

Size of Files Containing Prime Numbers of Differing Ranges

Prime Number Ranges	Size of file (KB)	Runtime
<1,000	13	0.0848 seconds
<10,000	149	0.838 seconds
<100,000	1,714	8.75 seconds
<1,000,000	5,820	1.51 minutes
<100,000,000	53,643	2.51 hours
<1,000,000,000	502,132	24.54 hours

The table above shows that as we increase the number of prime numbers we wish to generate, the more computer storage space needed to store the file and the longer it will take to generate, which can get very long and take up a lot of space for prime numbers below 1,000,000,000. However, the most common implementation of the RSA cryptosystem (RSA-2048) sets the size of n to 2048 bits (Kamiński & Mazurczyk, 2023). The possible integers n, that can be stored in 2048 bits is:

$$2^{2047} \leq n \leq 2^{2048} - 1$$

$$1.61 \times 10^{616} \leq n \leq 3.23 \times 10^{616}$$

The possible prime numbers that can be stored in 1024 bits are extremely large, hundreds of orders of magnitude larger than the number of prime numbers I generated. There are $1.61 \times 10^{616} - 3.23 \times 10^{616} \approx 1.61 \times 10^{616}$ integers that the algorithm in Appendix B needs to search through in order to obtain a list of primes that can be stored in 1024 bits. Even assuming that it takes 1 nanosecond ($10^{-9}s$) to check if a single number is prime, the program will take 5.12×10^{599} **years** to check 1.61×10^{616}

integers, the sun is expected to explode earlier than that! This tells us that it is *computationally impossible* to use the algorithm in Figure 2 to try and factorise numbers used in the RSA cryptosystem, we need another method of prime factorisation that doesn't involve pre-generated prime numbers.

3.3.2 General Number Field Sieve

The General Number Field Sieve (GNFS) algorithm is the fastest known factorisation algorithm currently (Cho et al., 2016). It does this by using algebraic number fields and polynomial relationships to find two numbers whose squares are congruent modulo the target number (Haarberg, 2016). Using an implementation written by Brown (2015), I measured the runtime of the GNFS algorithm as I increased the size of the prime factors of the number to be factorised.

Table 3

GNFS Algorithm Runtime with Differing Prime Factor Sizes

Size of Prime Factors	Runtime
10 decimal digits	0.0150 seconds
12 decimal digits	0.243 seconds
14 decimal digits	1.73 seconds
16 decimal digits	7.26 seconds
18 decimal digits	65.8 seconds
20 decimal digits	8.56 minutes
22 decimal digits	1.12 hours
24 decimal digits	8.97 hours
26 decimal digits	1.17 days

As seen in the table above, as I increase the number of digits of the prime factors of the number the GNFS algorithm factorises, the runtime of the algorithm still increases exponentially. The number of binary digits a prime factor of a key in RSA-2048 is 1024 bits, which has a maximum value of $2^{1024} - 1 \approx 1.79 \times 10^{308}$ which is a number with 308 digits, still hundreds of orders of magnitude larger than the number I prime factorised. In fact, the largest RSA key prime factorised with the GNFS algorithm is an RSA-250 key by Zimmermann (2020). The key had a size of 250 decimal digits with prime factors of 125 decimal digits:

$$\begin{aligned} n = {} & 2140324650240744961264423072839333563008614715144755017797754920881418 \\ & 0234471401366433455190958046796109928518724709145876873962619215573 6 \end{aligned}$$

$$\begin{aligned}
n ={} & 2140324650240744961264423072839333563008614715144755017797754920881 41\\
& 802344714013664334551909580467961099285187247091458768739626192155736\\
& 304745477052080511905649310668769159001975940569345745223058932597669\\
& 747168173806936489469987157849497593749793 7\\
={} & 641352894770715802787901901705773890848250147429434472081168596320245\\
& 32344630238623598752668347708737661925585694639798853367\times\\
& 333720275949781565562260106053551142279407603447675546667845209870238\\
& 41729210037080257448673296881877565718986258036932062711
\end{aligned}$$

No larger RSA key has been factorised. The factor size of an RSA-250 key is still more than 200 orders of magnitude smaller than the factor size of an RSA-2048 key, and hence, even with the fastest prime factorisation algorithm, it is still *computationally impossible* to factorise a key from the current RSA standard, RSA-2048, at least for *today* in 2024

3.3.3 Future Attacks

As time progresses, however, computers get computationally more powerful, enough to factor larger RSA keys. Even so, RSA standards can always increase the size of RSA keys being used as the standard for communications. Before RSA-2048 became the standard, RSA-1024 was the standard that used an n with the size of 1024 bits. However, RSA Security LLC had recommended that by 2010, the new standard should be RSA-2048 (Kaliski, 2003), and so over time, companies transitioned from RSA-1024 to RSA-2048: Firefox required an RSA key with a modulo of at least 2048 bits, any lower will display an 'Untrusted Connection' error. Similarly, the National Institute of

Standards and Technology suggests that by 2030, RSA-3072 should be the new RSA standard, with RSA-2048 only allowed for legacy use (Regenscheid, 2024). And so, the security of RSA can be ensured by continuously increasing the size of RSA keys.

However, that is assuming that there aren't any new technologies that could introduce faster methods to prime factorise. Quantum Computing is an example of such technology that poses a threat to the RSA cryptosystem. Quantum algorithms like Shor's algorithm can be used to factorise a large number (Singleton, 2023), the details of how it works won't be discussed in order to keep this EE focused. Gidney et al. (2021) believe that it would take 80 million qubits to factor an RSA-2048 key in only 8 hours. However, the largest number of qubits ever achieved was 1121 qubits from IBM (AbuGhanem, 2024). IBM believes that they would be able to achieve 2000 qubits by 2033, according to their roadmap (AbuGhanem, 2024), which is still multiple orders of magnitude smaller than the 80 million qubits outlined by Gidney et al., and hence, RSA encryption is still relevant *today*.

4 Conclusion

The RSA cryptosystem is a vital public-key encryption scheme used all over the internet to secure communications despite the existence eavesdroppers. Due to the difficulty of prime factorising extremely large numbers, users on the internet are freely able to send their public-keys to whoever that wish to communicate with them, knowing that it will take billions of years to prime factorise a product of 2 large primes using current methods of prime factorisation.

However, there have been concerns regarding the security of the RSA cryptosystem due to prime factorisation algorithms that may be feasible in the future due to advancements in quantum computing, like Shor's Algorithm. However, such algorithms are still not feasible in 2024 due to the limited processing power of quantum computers in 2024.

Other than these attacks, the RSA still provides the internet a high enough level of security in 2024, and the fact that it's still one of the most popular public-key cryptosystem is proof of that.

References

AbuGhanem, M. (2024). Ibm quantum computers: Evolution, performance, and future directions. Retrieved October 25, 2024, from https://arxiv.org/abs/2410.00916

Barge, W. C. (2019). The mathematics behind rsa encryption. *ISSA Journal*, *17*(5), 26–30.

Brown, L. (2015). Primefac: A python library for integer factorization [Version 2.0.12]. Retrieved October 25, 2024, from https://pypi.org/project/primefac/

Cho, G. H., Koo, N., & Kwon, S. (2016). On nonlinear polynomial selection and geometric progression (mod n) for number field sieve. *Bulletin of the Korean Mathematical Society*, *53*(1), 1–20. https://doi.org/10.4134/BKMS.2016.53.1.001

Euclidean algorithms (Basic and Extended). (2015). Retrieved September 18, 2024, from https://www.geeksforgeeks.org/euclidean-algorithms-basic-and-extended/

Gidney, C., Ekerå, M., & Ekerå, M. (2021). How to factor 2048 bit rsa integers in 8 hours using 20 million noisy qubits. *5*, 433–. https://doi.org/10.22331/Q-2021-04-15-433

Haarberg, H. R. (2016). *The number field sieve for discrete logarithms* [Master's thesis, NTNU].

Hellman, M. E. (2002). An overview of public key cryptography. *IEEE Communications Magazine*, *40*(5), 42–49.

Kaliski, B. (2003, May 3). RSA Laboratories - TWIRL and RSA Key Size — web.archive.org.

Kamiński, K., & Mazurczyk, W. (2023). Rsa keys quality in a real-world organizational certificate dataset: A practical outlook. *International Journal of Electronics and Telecommunications.*

Mansour, A. H. (2017). Analysis of rsa digital signature key generation using strong prime. *International Journal of Computer*, *24*(1), 28–36.

Mohamad, M. S. A., Din, R., & Ahmad, J. I. (2021). Research trends review on rsa scheme of asymmetric cryptography techniques. *Bulletin of Electrical*

Engineering and Informatics, *10*(1), 487–492. https://doi.org/10.11591/EEI.V10I1.2493

Naranjo, J., López-Ramos, J., & Casado, L. (2010). Applications of the extended euclidean algorithm to privacy and secure communications. *Proc. of 10th international conference on computational and mathematical methods in science and engineering*, 702–713.

New Algorithm to Generate Prime Numbers from 1 to Nth Number. (2015). Retrieved October 22, 2024, from https://www.geeksforgeeks.org/new-algorithm-to-generate-prime-numbers-from-1-to-nth-number/

R. Omondi, A. (2020). Modular reduction. In *Cryptography arithmetic: Algorithms and hardware architectures* (pp. 105–141). Springer International Publishing. https://doi.org/10.1007/978-3-030-34142-8_4

Regenscheid, A. (2024, July 15). *Cryptographic algorithms and key sizes for personal identity verification :* (tech. rep.). National Institute of Standards; Technology (U.S.) https://doi.org/10.6028/nist.sp.800-78-5

Riesel, H. (2013, March 14). *Prime numbers and computer methods for factorization.* Springer Science & Business Media.

Rivest, R. L., Shamir, A., & Adleman, L. (1978). A method for obtaining digital signatures and public-key cryptosystems. *Commun. ACM*, *21*(2), 120–126.

Singleton, R. L. (2023). Shor's factoring algorithm and modular exponentiation operators: A pedagogical presentation with examples.

Stein, W. (2009, January 8). *Elementary number theory: Primes, congruences, and secrets.* Springer Verlag.

Zimmermann, P. (2020). Factorization of rsa-250 [Message posted to the NMBRTHRY mailing list, 28 Feb 2020].

Appendix A

Figure 3

Extended Euclidean Algorithm

```cpp
bool isPrime(int n)
{
// C++ program to demonstrate working of
// extended Euclidean Algorithm
#include <bits/stdc++.h>
using namespace std;
// Function for extended Euclidean Algorithm
int gcdExtended(int a, int b, int *x, int *y)
{
    // Base Case
    if (a == 0)
    {
        *x = 0;
        *y = 1;
        return b;
    }
    int x1, y1; // To store results of recursive call
    int gcd = gcdExtended(b%a, a, &x1, &y1);
    // Update x and y using results of
    // recursive call
    *x = y1 - (b/a) * x1;
    *y = x1;
    return gcd;
}
}
```

```
bool isPrime(int n)
{
// Driver Code
int main()
{
    int x, y, a = 35, b = 15;
    int g = gcdExtended(a, b, &x, &y);
    cout << "GCD(" << a << ", " << b
         << ") = " << g << endl;
    return 0;
}
}
```

Note. Figure obtained from “Euclidean algorithms (Basic and Extended)” (2015)

Appendix B

Figure 4

Generating Prime Numbers

```cpp
#include <bits/stdc++.h>
using namespace std;
int countPrimesUpto(int n)
{
        int count = 0;
        bool arr1[n + 1];
        bool arr2[n + 1];
        int d = 5;
        arr1[2] = arr2[2] = 1;
        arr1[3] = arr2[3] = 1;
        memset(arr1, 0, sizeof(arr1));
        memset(arr2, 1, sizeof(arr2));
        while (d <= n) {
                memset(arr1 + d, 1, (sizeof(arr1)) / (n + 1));
                memset(arr1 + (d + 2), 1, (sizeof(arr1)) / (n + 1));
                d = d + 6;
        }
        for (int i = 5; i * i <= n; i = i + 6) {
                int j = 0;
                while (1) {
                        int flag = 0;
                        int temp1 = 6 * i * (j + 1) + i;
                        int temp2 = ((6 * i * j) + i * i);
                        int temp3 = ((6 * (i + 2) * j)
                                                + ((i + 2) * (i + 2)));
                        int temp4 = ((6 * (i + 2) * (j + 1))
                                                + ((i + 2) * (i + 2)) - 2 * (i + 2));
                        if (temp1 <= n) {
                                arr2[temp1] = 0;
                        }
```

```
            else {
                                flag++;
                        }
            if (temp2 <= n) {
                                arr2[temp2] = 0;
                        }
                        else {
                                flag++;
                        }
                        if (temp3 <= n) {
                                arr2[temp3] = 0;
                        }
            else {
                                flag++;
                        }
                        if (temp4 <= n) {
                                arr2[temp4] = 0;
                        }
                        else {
                                flag++;
                        }
                        if (flag == 4) {
                                break;
                        }
                        j++;
                }
        }
        if (n >= 2)
                count++;
        if (n >= 3)
                count++;
        for (int p = 5; p <= n; p = p + 6) {
                if (arr2[p] == 1 && arr1[p] == 1)
                        count++;

                if (arr2[p + 2] == 1 && arr1[p + 2] == 1)
                        count++;
        }
        return count;
}
```

```
int main()
{
        int n = 100;
        cout << countPrimesUpto(n);
}
```

Note. Figure obtained from “New Algorithm to Generate Prime Numbers from 1 to Nth Number” (2015)