An investigation on the effect of increasing bit rate on security

Research Question: How does the magnitude of the prime numbers affect the time it takes to decrypt an RSA encrypted message?

Subject: Computer Science

Personal Code: jbb161

Word Count: 3998

Contents

1 Introduction

The content of this essay concerns encryption in computer science. With the constant rapid increase in computing power, as well as more reliance on online banking, or online medical records, the need for security on the internet has been more important than it ever has been.

One of the methods of securing data on the internet is by encrypting it — essentially turning the original message (plaintext) into another form (ciphertext) — making sure that when information is intercepted over the internet, the data received would be worthless. All encryption methods use a one-way function [Weisstein, 2020], where a result is very easy to compute, but inverting this result is difficult, meaning that you cannot use information from the encrypted ciphertext to find out what the inputs were.

In this way, **how does the magnitude of the prime numbers affect the time it takes to decrypt an RSA encrypted message?**

1.1 Goal

The goal of this extended essay is to explore how the size of the prime numbers selected by the RSA encryption algorithm affects the time taken by a computer to crack the private key.

2 The Concepts Behind RSA Encryption

One example of an encryption algorithm is the RSA encryption algorithm, which is a type of *public-key cryptology*, A cryptology system where the key used to encrypt a message is different to the key used to decrypt a message, used for transactions over the internet [Simmons, 2012].

2.1 Determinism

The RSA encryption system is a deterministic function — meaning that for every input, the output will always be the same given that no padding (including redundant data somewhere in an unchanged message prior to encrypting it) is used. However, this leaves some security vulnerabilities, as people could simply encrypt common messages in order to find a matching ciphertext, which can then be used to find the plaintext. Padding is typically placed in a random position in the plaintext prior to encryption, making each ciphertext different from the other.

However, for simplicity, the program coded will not apply any padding, and will assume that the original plaintext is not common. This will not affect the integrity of the results — even with padding, the ciphertext can still be decrypted with the same method.

2.2 P Not Equal to NP

The core idea behind every single encryption system is that P $\neq$ NP holds true. According to an MIT news article from 2009 [Hardesty, 2009], P, standing for polynomial, can roughly be defined as a set of relatively easy problems, while NP, standing for non-deterministic polynomial time, can be defined as a set of extremely difficult problems that can be verified in polynomial time (which will be expanded upon later).

For instance, imagine an algorithm that outputs every message from a given array. With this algorithm, there is no way to avoid any cases, so the program must iterate through the whole list, and therefore, the execution time is directly proportional to the number of elements, commonly denoted as N in the world of computer science. This would be an example of a P function. P functions can be more complex, possibly even taking the time complexity of N^2 or N^3. While these functions may be more resource intensive than one with a time complexity of N, they are still considered polynomial functions. The reason is quite clear when you compare a polynomial graph

to a non-polynomial one.

On the other hand, if we look at the properties of a NP function, you are able to verify the solution in polynomial time, but solving it is far more time and resource intensive. One example would be of finding the prime factors of a number. To verify that the prime factors are correct, all you need to do is multiply the numbers together and see if it matches the intended result. However, to find the prime factors of a number, you must go through every solution with brute force to find the prime factors. Such functions have an exponential time complexity. As the number of elements goes up, the number of time taken to solve it increases drastically. NP functions have a time complexity of something proportional to e^N.

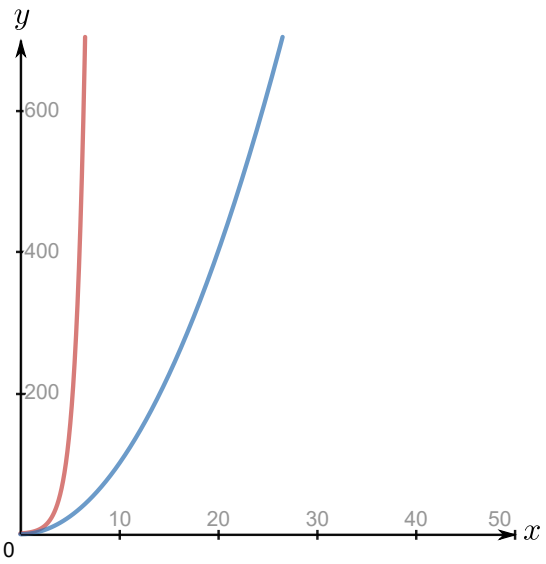


Figure 1: A comparison between a P and an NP function.

In Figure 1, the red line is an NP function, while the blue line is a P function. In this case, assuming the x axis shows number of elements, while the y axis shows the time taken, we can see that as number of elements goes up, the time taken for the NP function increases significantly more than the P function.

In this way, many encryption systems use the properties of NP functions in order to ensure that their encryption is secure to any attacks, but is also easy to verify. In fact, all encryption systems

around the world hold the belief that P $\neq$ NP, and if P $\neq$ NP were to be proven false, all encryption systems around the world would be exploitable.

2.3 One-Way Functions

Another key concept in cryptology is the one-way function [Weisstein, 2020]. As mentioned earlier, this is a function where finding the range from the domain is very easy, but finding the domain from the range is essentially impossible. This is useful for encryption because it allows for a plaintext message to be encrypted very easily, but decryption is impossible using the same function. Most, if not all, of the one-way functions used in cryptology have NP time complexity as they would need to take a lot of resources for computers to solve in order to be effective.

2.4 Asymmetric Encryption

In an asymmetric encryption system (known also as public key cryptology), a public key, a key publicly available for everyone in use for encryption, and a private key, a key only available to the recipient for decryption, are both used. In this system, the sender is the only person that will ever receive the private key, and the receiver only has access to the public key. This means that only the sender will be able to decrypt the ciphertext, but anyone is able to encrypt plaintext, allowing for extra security. A good analogy for this would be like a mailbox. Anyone can put mail inside this hypothetical mailbox using the 'public key', but only the recipient is able to open the mailbox with their 'private key'. Anyone will be able to encrypt plaintext using the public key, but will not be able to decrypt it due to the properties of one-way functions 2.3

On the other hand, a symmetric encryption system (private key cryptology) is faster than an asymmetric system, but at the expense of greater security issues. This system uses only a private key for both encryption and decryption, meaning that both the sender and the receiver will need

to have a copy of the same private key. This leads to transmission of the public key through the internet, which leaves it more susceptible to interception.

For this reason, the RSA encryption system uses an asymmetric encryption system, as it allows for fewer security risks. Additionally, as RSA is "widely used for data encryption of e-mail and other digital transactions over the internet" [Simmons, 2012], speed is not a top priority, so an asymmetric encryption system would be more suitable.

3 How RSA Encryption Works

All the computations below will be explained in greater detail in the sections below.

Broadly speaking, to encrypt plaintext using the RSA encryption system, two prime numbers (referred to as p and q are picked, and the product of p and q (referred to as n is then computed, and is released as part of the public key, and used in subsequent calculations. In this case, p and q should be very large, such that finding the factors of n is well beyond current computing power.

After computing n, it is then ran through Carmichael's totient function [Brilliant.org, 2020] in order to compute $\lambda(n)$. This is done by taking the lowest common multiple of $(p-1)$ and $(q-1)$. This value needs to stay hidden as it can be used to derive the private key.

With $\lambda(n)$ now computed, an integer e is selected that is co-prime, where the numbers do not have any common factors other than 1, where $\lambda(n)$ and is in the range $1 < e < \lambda(n)$. This number is then released as part of the public key.

Now that e has been selected, d — the ciphertext — is calculated using the property $d \equiv e^{-1} (\text{mod } \lambda(n))$. In this case, d is the modular multiplicative inverse of e, which means that you can recover the original message m from a ciphertext c. [Khan Academy, 2020]

Afterwards, in order to encrypt a message m, and turn it into a ciphertext c, the equation $m^e \equiv c(\text{mod } n)$ is used.

To decrypt the ciphertext c, recall that c comes from $m^e(\text{mod } n)$. Additionally, as d is the modular multiplicative inverse, it means that $(m^e)^d$ is equivalent to m. From this, we can deduce that $c^d \equiv m(\text{mod } n)$, which will allow us to find the original message m.

3.1 Primality Check

Instead of looking at all the possible factors for a number and seeing that there are none, encryption systems like these use a primality check instead, which significantly speeds up the process of finding large prime numbers.

The primality check is a probabilistic test, where if a number passes through the test once, the probability that the number is prime gets higher, and with enough repetitions, the chances of the number being prime is high enough that it is assumed to be prime.

One example is the Miller-Rabin primality test [Lynn, 2020]. This is built off the Fermat test stating that if a number n is prime, for any value of a in the range $1 < a < n$, the expression $a^n - a = 1(\text{mod } n)$ holds true, meaning that the value of $a^n - a$ is always 1 greater than the multiple of n if the number is prime. However, this also sometimes has false positives — 561 passes the Fermat test despite being a composite number.

The Miller-Rabin test expands upon the Fermat test by iterating through the same concept. Using another property of prime numbers where the only solutions to $x^2 = 1(\text{mod } n)$ are ± 1, we can then use Fermat's little theorem. If a number passes the Fermat's test, it means that $a^n - a = 1(\text{mod } n)$, and that the number is potentially prime. Using the above identity, if the number is prime, $a^{(n-1)\div 2} = \pm 1$ should also hold true.

Additionally, $n-1$ can always be written as the product of 2^s and q, where s is the "largest power of 2 dividing $n-1$", and q is an odd number. Using this, we can then see that a^{n-1} is equal to $a^{2^s q}$, $a^{2^{s-1}q}$ all the way to a^q. If n is prime, these values will either be equal to -1 or 1.

This process is iterated either until the 2^s component is equal to 1, or up to a user limit for search depth.

Strictly speaking, this is a "compositeness test" as it does not prove that a number is prime, but only proves that it is composite. However, with more iterations, it is safe to say that a number is prime. For "any composite n, the probability n passes the Miller-Rabin test is **at most** 25%" This means that each iterations decreases the likelihood of a composite number passing the Miller-Rabin test exponentially, so with 25 iterations, the chance of a composite number being returned is $\frac{1}{4^{25}}$. In the RSA encryption algorithm, it is used to generate the prime numbers p and q which would otherwise take exponentially longer the larger the bit depth.

3.2 Carmichael's Totient Function

Carmichael's totient function associates every number n to another number $\lambda(n)$ [Brilliant.org, 2020]. $\lambda(n)$ is defined to be "the smallest positive integer k such that $a^k \equiv 1 (\text{mod } n)$ for all a such that the greatest common denominator of a and n is 1".

As a numerical example, let n be 7. We first need to find a case where the $\gcd(a, n) = 1$. In this function, all values of a satisfy the equation, so we will let a be 3 in this case as it satisfies the equation above.

From this, we simply need to find a number k such that $3^k \equiv 1 (\text{mod } 7)$. What this means is that we need to find a number k where 3^k is 1 greater than a multiple of 7. In this case, if we set $k = 6$, we then get $3^6 \equiv 729$. As 728 is a multiple of 7, this then satisfies the equation. This equation also holds true for other values of a where $\gcd(a, n) = 1$.

As we can see from above, the value of $\lambda(n)$ is one less than our selected number n. This holds true for other *prime* numbers, where the totient function returns a value one less than the input if the input is prime.

Another property of Carmichael's totient function is that $\lambda(ab) \equiv \lambda(\text{lcm}(a, b)) \equiv \text{lcm}(\lambda(a), \lambda(b))$. Using this property, and the fact that our value n is generated from multiplying two prime numbers p and q, we can then see that $\lambda(n) = \text{lcm}(\lambda(p), \lambda(q))$. This value is used to derive the private key.

3.3 Plaintext Conversion

As all the above equations deal with numbers, but plaintext is a string, meaning that it can contain any form of letter, there needs to be a way to convert this plaintext to a number. Thankfully, there are different systems that can do this, the most common being ASCII and Unicode, which map a character to an integer.

However, there are some limitations to how large the integer can be due to limitations with modular arithmetic. This is because the mod function outputs the remainder of a quotient. For example, 32 mod 3 would be equal to 2 as $32 \div 3$ is 10 with a remainder of 2. As the mod function only returns 2 in this case, the information 'stored in' 10 would be lost.

If we look at section 3, we can see that the equation for encryption is $m^e \equiv c(\text{mod } n)$, which is the same as $c = m^e \text{mod } n$. As a result, the integer plaintext m must be smaller than n.

4 Methodology

The theory from section 3 was used in order to write a program that encrypts a message.

For all results below, the plaintext message was "0", which converts to the integer 48 in ASCII.

Therefore, the plaintext message in an integer form m will be equal to 48 in the following code.

Additionally, as the prime numbers were generated randomly, a set of fixed seeds were used such that the test could be replicated in the exact same way every time the prime numbers are generated. The following array has 20 different pairs of numbers generated randomly using random.org to generate 20 different values of n that can be repeatable on every attempt.

```
1 seeds = [[468, 2739], [2483, 6257], [4732, 8191], [5999, 3579], [6836, 8699],
   [7033, 6239], [4914, 7296], [316, 8561], [3918, 1914], [5897, 2767], [8625,
   3002], [8334, 7544], [6834, 1383], [397, 2694], [8450, 238], [3992, 789],
   [8573, 6704], [7286, 2697], [365, 6181], [8087, 4730]]
2 random.seed(seeds[0][0]) # This sets the random seed to 468, so all random
   calculations from here will use this seed until changed.
```

Listing 1: Array of seeds for random number generation.

The program then generated 20 different values of n for each magnitude k where $8 \leq k \leq 114$, the ciphertext generated, as well as the value of e, and stored them in a text file.

A separate script then reads from the text file, and times how long the program takes to decrypt each line, and adds that to another file storing the magnitude and the time in seconds.

The data from the new text file was then graphed using Python modules.

4.1 Assumptions

- The generated values n are distributed randomly, and do not have any sort of skew. This means that searching in ascending order will be efficient.
- The magnitude of n is equal to double the magnitude of one of the prime numbers generated. The magnitude will be defined as a variable k where $n = 2^k$ For example, if the magnitude of p was set to be 8, the magnitude of n would be 16.

- When the program is unable to decrypt a line of the file, it is because n was generated using one or more non-prime numbers.
- The program only runs on one core in order to save on having to worry about multi-core timing.

4.2 Encryption

The code for encryption uses the concepts in section 3. The program uses a nested for loop in order to generate 20 numbers from the same magnitude.

```
for j in range(8, 115): # The indented code loops with the value of j
 increasing incrementally.
      for i in range(len(seeds)): # The indented code then loops 20 times.
          random.seed(seeds[i][0]) # Sets random seed
          prime_1 = random_prime(j) # Generates a prime
          random.seed(seeds[i][1]) # Sets random seed again
          prime_2 = random_prime(j) # Generates a separate prime
          product = prime_1 * prime_2 # Gets the value of n.
```

Listing 2: Generating n.

The code then finds the totient by getting the lowest common multiple of one less than the variables *prime_1* and *prime_2*, and uses it to generate e. Then, to generate the ciphertext and input it into a file, the code does the following:

```
ciphertext = pow(message, e, product) # This does (message^e)mod product
 efficiently using a native Python function.
f.write(f"{j}|{ciphertext}|{e}|{product}\n") # f refers to the file created by
  python
```

Listing 3: Generate ciphertext and append to file.

4.3 Decryption

As mentioned earlier, the RSA encryption function is a one-way function, so trying to decrypt the private key from the public key is impossible without brute force.

However, there are some optimizations that can be done. When finding prime factors of a number n, all that needs to be done is searching every odd number from 3 up to $\sqrt{n}$, as after that, you will simply be searching through the same set of numbers. This saves time compared to searching every single number for prime factors.

After finding the prime factors p and q, the rest of the process can be done similarly to encrypting the number — following the same steps should give the same results as long as n was generated using two prime numbers.

5 Hypothesis

As the decryption algorithm checks for the factors of n by checking if every odd number between 3 and $\sqrt{n}$ is a factor of n, if the magnitude of n increases, it means that n increases exponentially, even though the prime factor searching function works linearly. Thus, the time taken by the decryption algorithm to find a factor increases exponentially, and for this reason, the curve should be in the form $a \times 2^{bx}$ where a and b are constants. This means that *when plotted on a log scale*, we should be able to see a linear curve between magnitude and time taken.

6 Data

A sample of the data is as follows:

```
56|461.0019738000001
```

```
56|793.2054947000006
56|449.61914160000015
56|633.6598563000007
```

The left side signifies the magnitude of one of the prime numbers, while the right side signifies the time taken in seconds for Python to decrypt that line.

This data was then plotted on a graph, with the (arithmetic mean) average time taken for each magnitude on the y axis, and the magnitude of n on the x axis.

6.1 Linear Scale

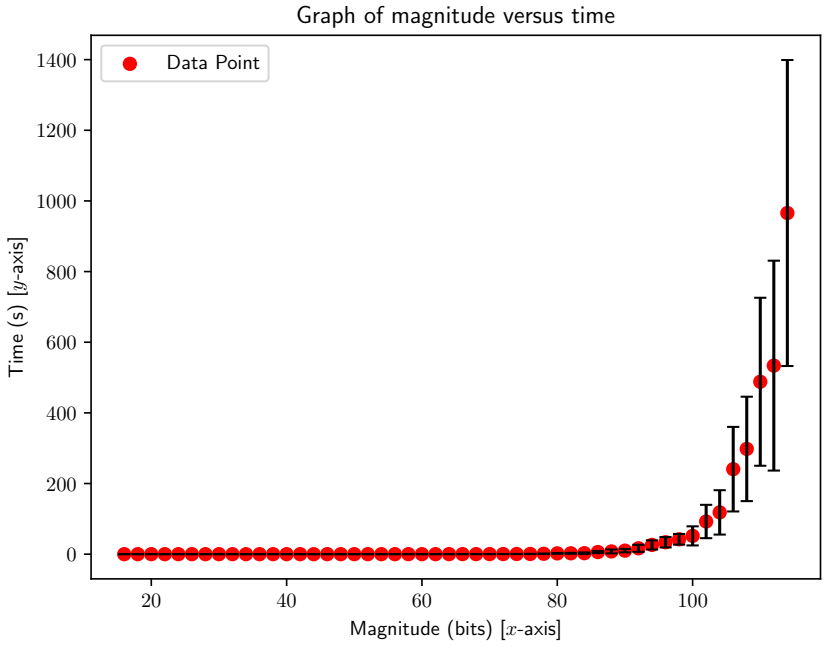


Figure 2: Data points plotted on a linear scale

The error bars for the data set are calculated as the standard deviation of the mean for each magnitude.

As we can see, the time taken in seconds remains increases drastically at the end, and can either be

shaped exponentially as hypothesized, or could even be a $y = -\frac{1}{x}$ style graph. However, this data is not useful as the data points from the lower end appears to be the same from the naked eye, so plotting it on a log scale will give more useful data, and can also show whether our Hypothesis is likely to be true.

6.2 Logarithmic Scale

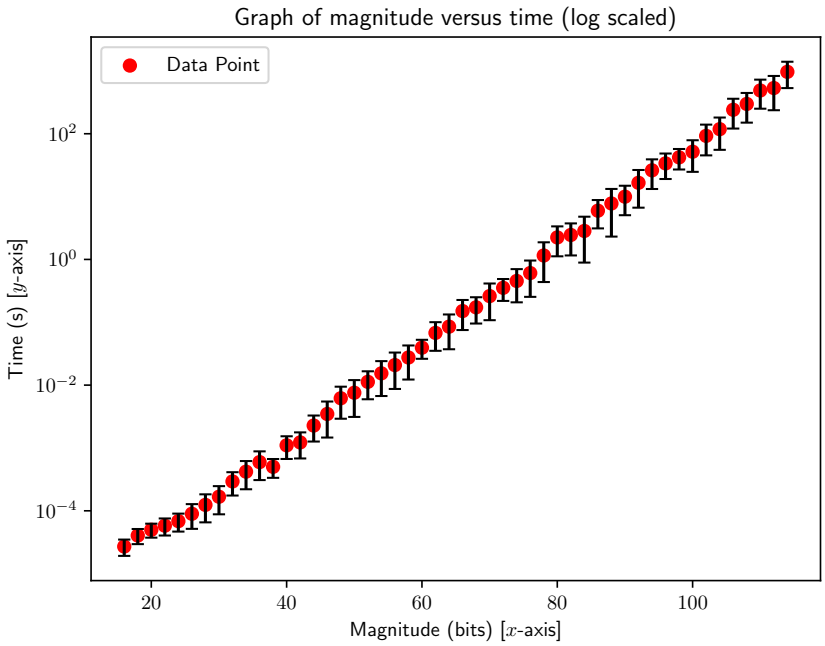


Figure 3: Data points plotted on a logarithmic scale

Under a logarithmic scale, we can see that the time taken against magnitude is generally linear, which further supports the claim that decrypting this has an NP time complexity. As a result of this, the relationship is exponential, supporting our hypothesis from section 5.

6.2.1 Line of Best Fit

Assuming that the relationship is exponential, it means that it follows a $y = a \times 2^{bx}$ relationship. Thus, we can use a Python module to find the curve of best fit for the above data, and plot it on the graph. Additionally, this graph can then be used to extrapolate to find how long it would take this computer to decrypt an RSA encrypted message using current RSA standards.

In order to plot the data on a linear scale to find the line of best fit, we can take the natural log of all results of time (the y axis), and then this should allow the data to be plotted linearly. Using the following properties, we can then form a linear equation.

$$y = a \times 2^{bx}$$
$$\log_2(y) = \log_2(a) + \log_2(2^{bx})$$
$$\log_2(y) = bx + \log_2(a)$$

This means that if we plot magnitude against the natural log of time, the constants b and a can be derived by the gradient and the y-intercept respectively.

Using *polyfit* from the module *numpy*, we are able to programmatically generate values for both b and $\log_2(a)$, allowing us to find the constants, which in turn will allow us to plot the graph.

```
1 d = np.polyfit(magnitude, np.log2(time), 1, w=np.sqrt(time)) # log2 refers to
   log base 2
2 print(d)
```

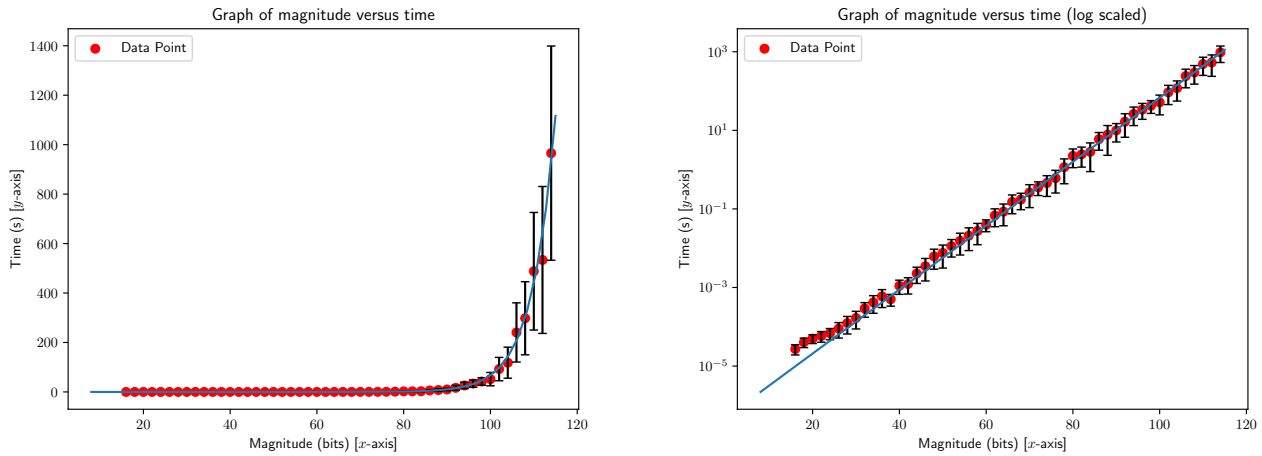
Listing 4: Finding the value of b and $ln(a)$.

The above code snippet[1] generated $b = 0.2703$ and $\log_2(a) = -20.96$. Using basic properties of

[1]The variable w in the code accounts for the fact that taking the log of a number puts more weighting on the

logarithms, we can then calculate the value of a as being $2^{-20.96}$. Thus, our equation for the curve of best fit is $y = \frac{1}{2^{20.96}} \times 2^{0.2703x}$.

Figure 4: Graphs with curves of best fit.



(a) On a linear scale.

(b) On a logarithmic scale.

As we can see, the blue line is the curve of best fit for both graphs, and the data points generally follow the curve of best fit other than a few of the lower points in the logarithmically scaled graph. This may be due to the fact that time calculations at the lower end of magnitude may be less precise than it is at the higher end.

This equation can also be used to extrapolate with meaningful information, as the method of decryption would be the exact same, and the difference in magnitude will affect n in the same way as it did for this experiment.

Nowadays, the recommendation for the RSA key bit-depth is 2048 [Barker and Dang, 2015] in order for modern supercomputers to be unable to crack the private key within several generations. In fact, the largest RSA key to be factored is 829 bits long, and it took approximately 2700 CPU years to do so [Zimmermann, 2020].

lower portion, so the variable w is used to counteract this.

However, it is worth putting this number into perspective using our equation.

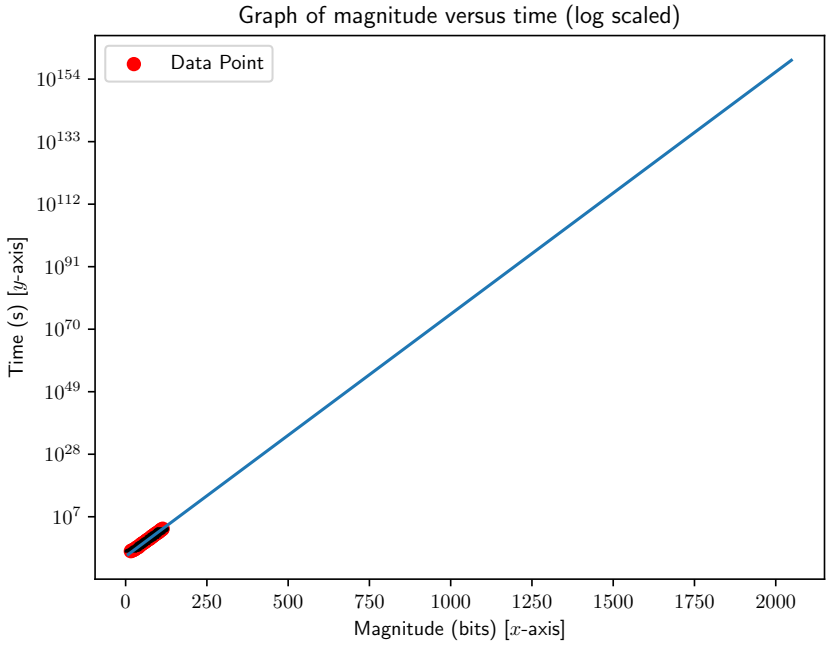


Figure 5: Extrapolating using the curve of best fit to a magnitude of 2048.

If we can see, the small portion of the graph at the beginning contains our data points collected, while the rest of the blue line contains the extrapolated data for how time taken changes in relationship to the magnitude.

If we look at the largest marking on the y-axis, we can see that the time taken for a 2048-bit encrypted message is *slightly* greater than 10^{154} seconds. To put that into perspective, 10^{154} is equal to around 3×10^{146} years, or around 2×10^{136} the age of the universe.

It is worth noting that these calculations are based on the time taken for a personal computer using one core to brute force a solution — supercomputers would be far quicker, but would still be taking a polynomial amount of time to crack this NP problem. Thus, a magnitude of 2048 is likely to be safe until quantum computing starts taking over.

7 Conclusion

Then, we come to our research question: **How does the magnitude of the prime numbers affect the time it takes to decrypt an RSA encrypted message?**

In conclusion, the RSA encryption system is one with an NP time complexity, which means that as the magnitude of the prime number increases, the time taken to brute force a solution increases exponentially.

According to primary data, the likelihood of RSA encryption being cracked using modern security standards is in essence, impossible, at least not until quantum computing comes around [Shor, 1994], because Short proved that quantum computers were able to find factors of an integer in polynomial time.

Overall, the data collected suggests that RSA encryption is safe for many years — or even decades — to come, and private information such as bank and medical records are likely to be kept private and secure for a long time.

However, there are some security flaws with RSA [Boneh et al., 2000], some of which include the fact that without padding, people could find the value of a common string by simply encrypting it, and seeing if it matches the ciphertext. To avoid this, random padding should be implemented such that every single string returns a unique ciphertext.

7.1 Further Investigation

One assumption (section 4.1) was that the program would run on a single core. However, this slows down the process, especially cause different cores could search different intervals, which should linearly increase the speed of the program. It would be interesting to track and compare the time taken between a single-core approach and a multi-core approach.

Moreover, comparing a public key cryptology system to a private key could be rewarding in terms of showing differences in security.

7.2 Evaluation

In evaluation, the time taken would be several times faster on a supercomputer. This would mean that the time taken to brute force a 2048-bit encrypted ciphertext would be slightly faster than one from my computer, but should still be outside the scope of current computing.

Additionally, experimental data may be slightly imprecise as each line in the text file is measured only once, and there could be other CPU processes interfering with the efficiency of the program.

8 References

Barker, E. and Dang, Q. (2015). Recommendation for key management. DOI: 10.6028/NIST.SP.800-57pt3r1.

Boneh, D., Joux, A., and Nguyen, P. Q. (2000). Why textbook elgamal and rsa encryption are insecure. In Okamoto, T., editor, *Advances in Cryptology — ASIACRYPT 2000*, pages 30–43, Berlin, Heidelberg. Springer Berlin Heidelberg.

Brilliant.org (2020). Carmichael's lambda function. `https://brilliant.org/wiki/carmichaels-lambda-function/`. Visited on 2020, August 4th.

Hardesty, L. (2009). Explained: P vs. np. `https://news.mit.edu/2009/explainer-pnp`. Visited on 2020, 15th May.

Khan Academy (2020). Modular inverses. `https://www.khanacademy.org/computing/computer-science/cryptography/modarithmetic/a/modular-inverses`. Visited on 2020, August 7th.

Lynn, B. (2020). Primality tests. `https://crypto.stanford.edu/pbc/notes/numbertheory/millerrabin.html`. Visited on 2020, August 4th.

Shor, P. W. (1994). Algorithms for quantum computation: discrete logarithms and factoring. In *Proceedings 35th Annual Symposium on Foundations of Computer Science*, pages 124–134.

Simmons, G. J. (2012). Rsa encryption. `https://www.britannica.com/topic/RSA-encryption`. Visited on 2020, April 28th.

Weisstein, E. (2020). From mathworld–a wolfram web resource. `https://mathworld.wolfram.com/One-WayFunction.html`. Visited on 2020, May 10th.

Zimmermann, P. (2020). Factorization of rsa-250. `https://lists.gforge.inria.fr/pipermail/cado-nfs-discuss/2020-February/001166.html`. Visited on 2020, August 7th.

9 Appendix

9.1 Computer Specs

AMD Ryzen R5 2600 / 16 GB DDR4 (3200MHz) / RX 5700

9.2 Code

9.2.1 Encryption

```
import random
from math import gcd

def lcm(a, b):
    return a*b//gcd(a,b)

def isPrime(n):
    if n == 1:
        return False
    i = 2
    while i*i <= n:
        if n % i == 0:
            return False
        i += 1
    return True


def coprime(a, b):
    return gcd(a, b) == 1


```

```
22 def get_e(n):
23     for i in range(1, n+1):
24         if isPrime(i):
25             if coprime(i, n):
26                 return i
27 
28 def extended_gcd(a, b):
29     lastremainder, remainder = abs(a), abs(b)
30     x, lastx, y, lasty = 0, 1, 1, 0
31     while remainder:
32         lastremainder, (quotient, remainder) = remainder, divmod(lastremainder,
    remainder)
33         x, lastx = lastx - quotient*x, x
34         y, lasty = lasty - quotient*y, y
35     return lastremainder, lastx * (-1 if a < 0 else 1), lasty * (-1 if b < 0
    else 1)
36 
37 def modinv(a, m):
38     g, x, y = extended_gcd(a, m)
39     y = y
40     if g != 1:
41         raise ValueError
42     return x % m
43 
44 first_primes_list = [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53,
     59, 61, 67,  71, 73, 79, 83, 89, 97, 101, 103,  107, 109, 113, 127, 131,
    137, 139,  149, 151, 157, 163, 167, 173, 179,  181, 191, 193, 197, 199, 211,
    223, 227, 229, 233, 239, 241, 251, 257, 263, 269, 271, 277, 281, 283, 293,
    307, 311, 313, 317, 331, 337, 347, 349]
45 
46 def nBitRandom(n):
47     return random.randrange(2**(n-1)+1, 2**n - 1)
48 
```

```
def getLowLevelPrime(n):
    while True:
        pc = nBitRandom(n)
        for divisor in first_primes_list:
            if pc % divisor == 0 and divisor**2 <= pc:
                break
        else: return pc

def isMillerRabinPassed(mrc):
    maxDivisionsByTwo = 0
    ec = mrc-1
    while ec % 2 == 0:
        ec >>= 1
        maxDivisionsByTwo += 1
    assert(2**maxDivisionsByTwo * ec == mrc-1)

    def trialComposite(round_tester):
        if pow(round_tester, ec, mrc) == 1:
            return False
        for i in range(maxDivisionsByTwo):
            if pow(round_tester, 2**i * ec, mrc) == mrc-1:
                return False
        return True

    numberOfRabinTrials = 20
    for i in range(numberOfRabinTrials):
        round_tester = random.randrange(2, mrc)
        if trialComposite(round_tester):
            return False
    return True

def random_prime(m):
    while True:
```

```
        prime_candidate = getLowLevelPrime(m)
        if not isMillerRabinPassed(prime_candidate):
            continue
        else:
            return prime_candidate

seeds = [[468, 2739], [2483, 6257], [4732, 8191], [5999, 3579], [6836, 8699], [7033, 6239], [4914, 7296], [316, 8561], [3918, 1914], [5897, 2767], [8625, 3002], [8334, 7544], [6834, 1383], [397, 2694], [8450, 238], [3992, 789], [8573, 6704], [7286, 2697], [365, 6181], [8087, 4730]] # These are the seeds that will be used for the pseudo-random number generator in order to make sure that all tests are repeatable
message = 48 # converts to 0 in ascii which is what's used as the message (THIS WILL BE TALKED ABOUT IN FURTHER INVESTIGATIONS, BUT I DON'T HAVE ENOUGH WORDS FOR IT (mention something about this at the end of your EE or something))w

with open('data.txt', 'w') as f:

    for j in range(8, 65):
        for i in range(len(seeds)):

            random.seed(seeds[i][0])
            prime_1 = random_prime(j)

            random.seed(seeds[i][1])
            prime_2 = random_prime(j)

            product = prime_1 * prime_2

            totient = lcm(prime_1-1, prime_2-1)

            e = get_e(totient)  # released as part of the public key

```

```
            ciphertext = pow(message, e, product)

            print(f"{j}|{ciphertext}|{e}|{product}")

            f.write(f"{j}|{ciphertext}|{e}|{product}\n")
```

9.2.2 Decryption

```
import time
import random
import math
from itertools import count
from math import gcd

def lcm(a, b):
    return a*b//math.gcd(a,b)

def extended_gcd(a, b):
    lastremainder, remainder = abs(a), abs(b)
    x, lastx, y, lasty = 0, 1, 1, 0
    while remainder:
        lastremainder, (quotient, remainder) = remainder, divmod(lastremainder
 , remainder)
        x, lastx = lastx - quotient*x, x
        y, lasty = lasty - quotient*y, y
    return lastremainder, lastx * (-1 if a < 0 else 1), lasty * (-1 if b < 0
 else 1)

def modinv(a, m):
    g, x, y = extended_gcd(a, m)
    y = y
    if g != 1:
```

```
        raise ValueError
    return x % m

def primeFactors(number):
    x = 2
    for cycle in count(1):
        y = x
        for i in range(2**cycle):
            x = (x*x + 1) % number
            factor = gcd(x-y, number)
            if factor > 1:
                return factor, number//factor

class Data:
    def __init__(self, magnitude, ciphertext, e, product):
        self.magnitude = magnitude
        self.ciphertext = ciphertext
        self.e = e
        self.product = product

decrypted_message = 48

data_set = []

with open("data.txt") as f:
    for line in f:
        line = line.rstrip("\n")
        elements = line.split("|")
        data_set.append(Data(elements[0], elements[1], elements[2], elements
 [3]))

with open("decrypt_data.txt", "w") as f:
    for i in range(len(data_set)):
```

```
        start_time = time.perf_counter()
        p, q = primeFactors(int(data_set[i].product))
        lambda_n = lcm(p-1, q-1)
        d = modinv(int(data_set[i].e),lambda_n)
        result = pow(int(data_set[i].ciphertext), d, int(data_set[i].product))
        if result == decrypted_message:
            elapsed_time = time.perf_counter()-start_time
        else:
            elapsed_time = "didn't solve"
        print(f"{data_set[i].magnitude}|{elapsed_time}|")
        f.write(f"{data_set[i].magnitude}|{elapsed_time}|\n")
```

9.2.3 Graphing

```
import numpy as np
import matplotlib
matplotlib.rcParams['text.usetex'] = True
import matplotlib.pyplot as pyplot
import scipy
import pandas
import math
from scipy.optimize import curve_fit

data_set = []
grouped_data = {}

class Data:
    def __init__(self, magnitude, time):
        self.magnitude = magnitude
        self.time = time

class Stats:
```

```
        def __init__(self, magnitude):
            self.magnitude = magnitude
            self.values = []
            self.mean = 0
            self.stdev = 0

    for i in range(8,58):
        grouped_data[i] = Stats(i)

    def isNumber(x):
        try:
            float(x)
            return True
        except:
            return False

    with open("decrypt_data.txt") as f:
        for line in f:
            elements = line.split("|")
            if (isNumber(elements[1])):
                data_set.append(Data(int(elements[0]),float(elements[1])))

    for i in range(len(data_set)):
        stats_class = grouped_data.get(data_set[i].magnitude)
        stats_class.values.append(data_set[i].time)

    magnitude = []
    time = []
    error = []

    for x in grouped_data:
        grouped_data[x].mean = np.mean(grouped_data[x].values)
        grouped_data[x].stdev = np.std(grouped_data[x].values)
```

```
        magnitude.append(grouped_data[x].magnitude*2)
        time.append(grouped_data[x].mean)
        # time.append(math.log(grouped_data[x].mean))
        error.append(grouped_data[x].stdev)

    magnitude = np.array(magnitude)
    time = np.array(time)

    d = np.polyfit(magnitude, np.log2(time), 1, w=np.sqrt(time))
    print(d)

    pyplot.errorbar(magnitude,time,yerr=error,fmt=' ',capsize=3,color='black')
    pyplot.scatter(magnitude,time,label="Data Point",color='red')
    x = np.linspace(8,115)
    y = (2**-20.96)*np.exp2(0.2703*x)
    pyplot.plot(x,y)
    pyplot.xlabel('Magnitude (bits) [$x$-axis]')
    pyplot.ylabel('Time (s) [$y$-axis]')
    pyplot.yscale('log')
    pyplot.title('Graph of magnitude versus time (log scaled)')
    pyplot.legend()
    pyplot.show()
```