Will The Encryption of Today be Made Obsolete by The Computers of Tomorrow?

Introduction

Encryption is a method of encoding messages, so that they can be decoded only by the intended receiver. This is used to protect sensitive information from anyone who might intercept the message. Though encryption does not stop someone from intercepting the message, the interceptor would be unable to decode and understand the message[1].

The main application of encryption today is through information sent over the internet, such as private messages, login information and passwords, personal and confidential information regarding finances, health records, and more. Thus, the implication of being able to overcome the security of encryption is that all of the private information of people using the internet becomes compromised and no longer private[2].

The security of Rivest–Shamir–Adleman (RSA) encryption, a widely used modern encryption method, is based on the fact that it is easy to multiply large numbers, however to factor a large number, meaning to find which numbers are multiplied to produce it, is very difficult[3]. In fact, in 1991 RSA Laboratories released a set of progressively larger numbers, and offered up to $200,000 USD to anyone who could factor them — showing their confidence in the security of their encryption. No one has been able to factor the largest of the numbers, leaving the $200,000 USD prize unclaimed. However, in May 2007, RSA Laboratories withdrew the challenge[4]. Why? Are they worried that soon, these numbers will be factored, putting the security of their encryption under question?

As computers get more powerful it becomes harder for encryption to stay ahead of the technology, but it is important that the flaws in encryption are understood to prevent someone exploiting them. Will computers be able to break encryption in the near future? Or is encryption still safe for years to

[1]"What Is Encryption and How Does It Work? | Google Cloud," Google, accessed March 7, 2024, https://cloud.google.com/learn/what-is-encryption.
[2] "What Is Encryption and How Does It Work? | Google Cloud," Google
[3] "RSA Encryption," Brilliant Math & Science Wiki, accessed March 7, 2024, https://brilliant.org/wiki/rsa-encryption/.
[4] The RSA Factoring Challenge, 2007, https://web.archive.org/web/20130921043459/http://www.emc.com/emc-plus/rsa-labs/historical/the-rsa-factoring-challenge.htm.

come? This investigation will look at the feasibility of an attack on information encrypted using RSA by analyzing potential methods of a brute force factoring attack.

To demonstrate, I have generated an encrypted message using a simple form of RSA:

```
5093797491863370116
9948631472706393745
4576814005525896971
9386295457731255133
2414985034203462679
7580559741887485016
```

I will determine the most efficient method for factoring large numbers and determine the feasibility of being able to successfully attack RSA encryption using these methods. While using the best algorithm and technique found in this investigation, at the end I will use the learnt techniques to hopefully crack the encryption and decipher this message, the same as someone performing an online privacy attack would.

For this investigation, a number n will be calculated as the product of two randomly chosen prime numbers in a certain range. Different algorithms will be used to find the prime factors of n, and this will be repeated for each algorithm while keeping track of the number of iterations required by each algorithm until the factors are found. Since the standard for RSA today involves a value of n that is a 617 digit number[5], much too large to factor repeatedly, the results will then have to be extrapolated to try and determine how many iterations it might take to factor a much larger number.

[5] "The RSA Challenge Numbers," RSA Laboratories - The RSA Challenge Numbers, accessed March 7, 2024, https://web.archive.org/web/20130921041734/http://www.emc.com/emc-plus/rsa-labs/historical/the-rsa-challenge-numbers.htm.

Example: Caesar Cipher

One of the simplest and most well known methods of encryption is the Caesar cipher, a cipher supposedly used by Julius Caesar to encrypt important military documents. In the Caesar cipher, each letter is shifted three places forward in the alphabet, thus 'A' becomes 'D' and 'B' becomes 'E' during encryption. The last three letters of the alphabet wrap around back to the start, so 'X' becomes 'A', 'Y' becomes 'B' and 'Z' becomes 'C'.

So the message: "SECRET MESSAGE"

Becomes: "VHFUHW PHVVDJH"

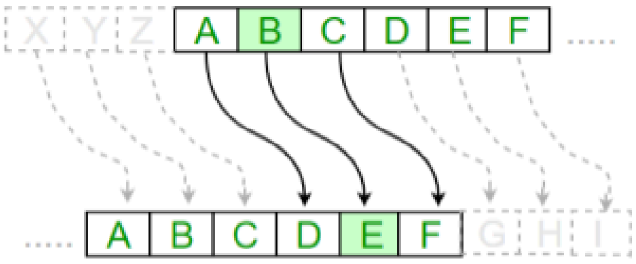


Image from Geeks for Geeks

This method of encryption may have worked a long time ago, but today has many flaws that prevent it from being used in modern encryption applications. The first flaw is that this method can be easily reverse engineered, by shifting each letter back three spaces in the alphabet. And even if a shift other than three were to be used, it would first have to be secretly communicated to the intended receiver of the message. Second, there are only 26 possible shifts, which can each be individually checked by an interceptor in a short amount of time.[6] These are common shortcomings of simple encryption methods that an efficient and secure encryption method would need to overcome.

RSA Encryption[7]

Rivest–Shamir–Adleman (RSA) encryption is an encryption method that was invented in 1977 and is still widely used today in online encryption. It is an asymmetrical encryption technique, meaning that unlike the Caesar cipher, it makes use of a public key, which is available to anyone and could be used to encrypt a message. However, this message can only be decoded using the private key, which is only available to the intended receiver of the message. I find it helps to think of the public key as a lock that is

[6] "Caesar Cipher in Cryptography," GeeksforGeeks, May 11, 2023, https://www.geeksforgeeks.org/caesar-cipher-in-cryptography/.
[7] "RSA Encryption," Brilliant Math & Science Wiki

provided to anyone who wishes to lock up a message sent to the holder of the private key. By providing a public key, anyone may encrypt or “lock up” their own messages to be sent to the holder of the private key, who can use the private key to “unlock” and decrypt the message. This allows for a message to be encrypted and decrypted without the need of sending another key beforehand. Here's how it works:

First of all, RSA relies on the fact that the message being encrypted and decrypted be in the form of a number. For the purposes of this investigation, I can correspond each letter to its index in the alphabet plus three, since for the encryption to work the number must be three or greater. Therefore each letter would be converted to a number according to the table below.

Letter:	A	B	C	D	E	F	G	H	I	J	K	L	M
Becomes:	3	4	5	6	7	8	9	10	11	12	13	14	15

Letter:	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
Becomes:	16	17	18	19	20	21	22	23	24	25	26	27	28

I can represent encryption of any number m as some function $E(m)$, which simply returns the ciphertext, or encrypted message. I can represent decryption of any ciphertext c as some function $D(c)$, which returns the original message before encryption. To put it another way:

$$D(E(m)) = m$$

Before encrypting with RSA, the public and private keys must be generated which are used to encrypt and decrypt the message. There is a process to key generation which allows RSA to work (see Appendix A), but for now I can just take the private key to be (11) and the public key to be the ordered pair $(131, 533)$. I will create a simple example message to encrypt and decrypt. Let the example message, denoted by m_o be 4, which would represent the letter ‘B’.

The intended message m_o can be encrypted into the ciphertext $E(m)$ using the equation below[8].

$$E(m) \equiv m^k \; mod \; n$$

Where k is the first number of the public key ordered pair, and n is the second number of the public key ordered pair.

Following the example, substituting in the message '4' into the formula along with the public key $(131,\ 533)$, I get:

$$E(4) \ \equiv \ 4^{131} \; mod \; 533$$

$$E(4) \ = \ 127$$

Therefore the encrypted message is the number 127. This encryption function is known by anyone who wishes to send a message, and the public key would also be available to anyone. From this function, it seems that information about the private key cannot easily be determined.

Once the message is encrypted, the receiver of the encrypted message can use the private key to decrypt the message using the formula for decryption below.

$$D(c) \equiv c^h \; mod \; n$$

Where h is the private key. So in this case, using the private key (11) and our encrypted message (127) as well as the second number in the public key again (533), I get:

$$D(127) \equiv 127^{11} \; mod \; 533$$

$$D(127) \ = \ 4$$

So I get back to 4, the original message. The method of RSA is well known, but the security of RSA lies in the generation of the public and private keys to allow the original message to be decrypted, and to keep it secure.

[8] "A mod B" returns the remainder when A is divided by B. For example, 5 mod 3 = 2, because 5 divided by 3 has a remainder of 2. Computers are able to perform this calculation very quickly, even with large numbers. The symbol "≡" is the symbol for congruence, meaning that if C ≡ A mod B, then C and A have the same remainder when divided by B.

The first step in generating these keys is choosing two random prime numbers, which can be called p and q. In the example above, the numbers 13 and 41 are used for values of p and q. By multiplying these two prime numbers, I get the product n.

Following the example to solve for n gives:

$$n = 13 \times 41$$

$$n = 533$$

The product n is a semiprime number, meaning it is the product of exactly two prime numbers[9]. This means that the only factors of this number are the two prime numbers that were multiplied to produce it, as well as itself and 1. This is important. This value n will be the second value of the public key. The generation of the private key depends on the two unique primes used to produce the second value of the public key (See Appendix A for full key generation process). These two values are kept secret, but could be determined by factoring the public key to produce these two prime numbers. That is where the security of RSA lies — in the difficulty of factoring the public key. While 13 and 41 are small primes which are easy to determine, the standard for RSA used in online data encryption is to use primes that are 617 digits long[10]. Factoring a number into primes this large becomes incredibly difficult, to the point that it is currently impossible using modern computers — it would take even the most powerful computer of today several thousand years to crack one RSA key[11].

Cracking RSA

There are many ways to attempt to find the factors of n. The simplest way would be to attempt to divide n by all numbers up to $\sqrt{n}$, and if the quotient is an integer value, meaning there is no remainder, then the quotient and the divisor are the two factors p and q. This can be slightly optimized, by first

[9] N. J.A. Sloane and R.K. Guy, "A001358 - Semiprimes," OEIS, March 7, 2024, https://oeis.org/A001358.
[10] "The RSA Challenge Numbers," RSA Laboratories
[11] Dave Krauthamer, "Q-Day: The Problem with Legacy Public Key Encryption," Help Net Security, July 11, 2022, https://www.helpnetsecurity.com/2022/07/15/legacy-public-key-encryption-problem/#:~:text=Using%20the%20public%20key%20in,an%20RSA%202048%2Dbit%20key.

checking if 2 is a divisor, and from then on only odd numbers up to $\sqrt{n}$ must be tested, since if 2 does not divide n, then a multiple of 2 also won't divide n. It could also be done by only checking if the number is divisible by all primes up to $\sqrt{n}$, but this involves precalculating potentially numerous prime numbers, which would make it overall less efficient.

For example, factoring 533 using this method would look something like this:

Iteration	Divisor	Quotient	Remainder
1	2	266	1
2	3	177	2
3	5	106	3
4	7	76	1
5	9	59	2
6	11	48	5
7	13	41	0

Therefore it takes seven iterations of this method to find the factors of 533. Not bad, but it is important to remember that 533 is a much smaller value than what would be used in RSA.

Another way of factoring a number would be using Fermat's factoring method. Fermat's method is based on the principle that any number can be represented by the difference of two perfect square numbers. That is to say that any integer n can be written as $y^2 - z^2$, where y and z are both integers. This can be rearranged as:

$$n = y^2 - z^2$$

$$z^2 = y^2 - n$$

Then it is a matter of guessing and checking, incrementing values for y, starting with y as the smallest integer that is either greater than or equal to $\sqrt{n}$.

If the difference between y^2 and n is a perfect square (meaning it has an integer square root), the factors of n can be found using the difference of squares property since integer values for y and z have been found[12].

$$n = y^2 - z^2$$

$$n = (y - z)(y + z)$$

Therefore the two factors of n are $(y - z)$ and $(y + z)$.

Substituting the public key (533) for n, $\sqrt{533} \approx 23.087$, so the starting value for y is 24. From here, factoring using Fermat's method would look something like this:

Iteration	y	$y^2 - n$	$\sqrt{y^2 - n}$ (z to 4 significant figures)
1	24	43	6.557
2	25	92	9.592
3	26	143	11.96
4	27	196	14.00

$$n = (y - z)(y + z)$$

$$n = (27 - 14)(27 + 14)$$

$$n = (13)(41)$$

Therefore in four iterations, the factors of n are found to be 13 and 41. For factoring 533, Fermat's method took three fewer iterations to factor than the first method shown.

[12] Eric W. Weisstein, "Fermat's Factorization Method," from Wolfram MathWorld, accessed March 7, 2024, https://mathworld.wolfram.com/FermatsFactorizationMethod.html.

Quantum Computers[13]

Quantum computers are on the leading edge of computer research, and while factoring the large numbers used in RSA is currently impossible using classical computers, quantum computers *could* potentially factor them faster. Quantum computers are not just faster classical computers, they have a fundamentally different architecture which allows them to do calculations in a different way than classical computers. This means that for some tasks, classical computers actually perform better than quantum computers. But because of the different architecture, quantum computers could theoretically perform some tasks much faster than a classical computer due to their ability to run different algorithms. One of these tasks is factorization.

Shor's algorithm is an algorithm created to run on a quantum computer that allows for factoring a number n, in on the order of $log^3(n)$ iterations. This means that it can factor a number n in $C \times log^3(n)$ steps, where C is some real number constant coefficient. However, for the large numbers that are used in RSA, this constant C makes a negligible difference in the number of iterations taken to factor a number n given that n is sufficiently greater than C. As the numbers become very large, what is more important is the nature of the function itself in terms of how it grows.

Currently, Shor's algorithm cannot be effectively performed, due to accumulating errors when working with current quantum computers. However, if in the future quantum computers are able to run either error free or with efficient error correction, Shor's algorithm would be a revolutionary way to factor large numbers and could make RSA generally obsolete.

Method Testing

To see experimentally the efficiency of these algorithms, I will write code to test both Fermat's factoring method and the first factoring method described above. I will run both of these algorithms where two prime numbers between 0 and 1000 are randomly chosen and multiplied to produce the product d.

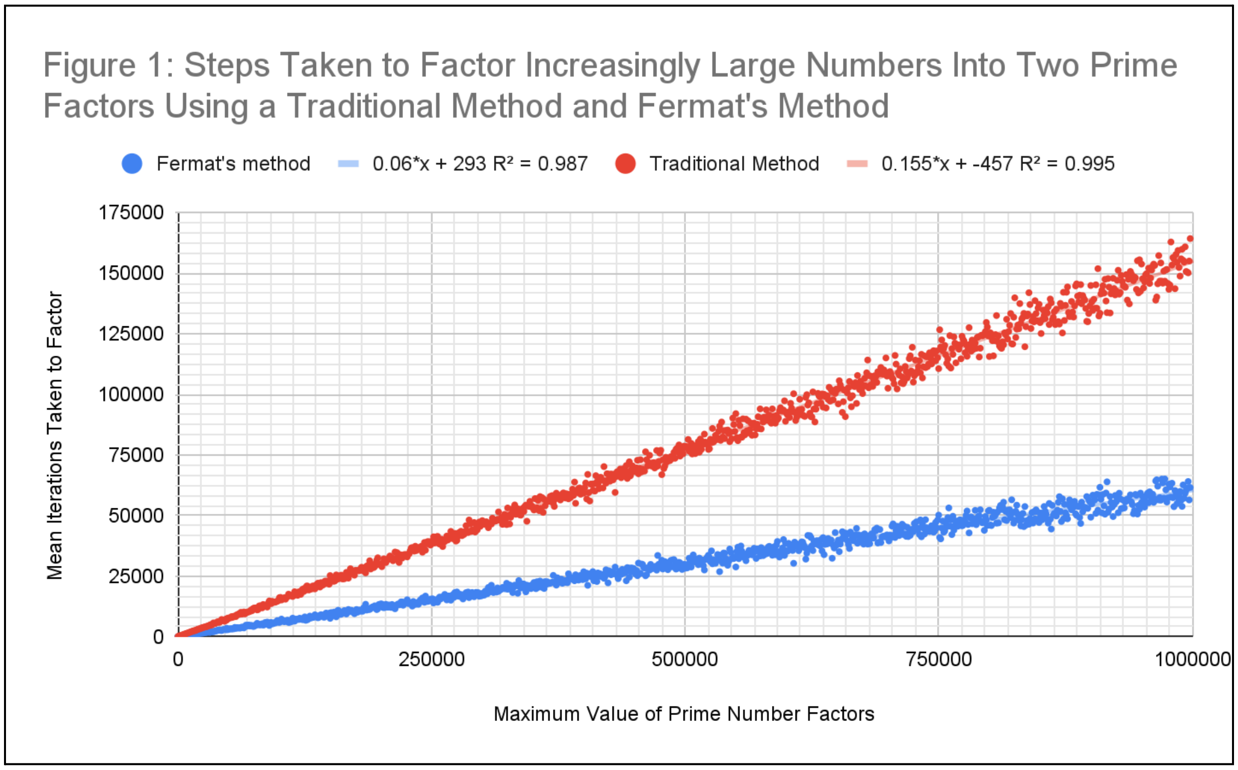
[13] Peter W. Shor, "Polynomial-Time Algorithms for Prime Factorization and Discrete Logarithms on a Quantum Computer," arXiv.org, January 25, 1996, https://arxiv.org/abs/quant-ph/9508027.

Then, both Fermat's method and the classical factoring method are performed on d, and the number of iterations taken to factor the number by each algorithm is recorded. Let $S(d)$ be the number of steps taken by an algorithm to factor d. This is repeated 500 times, each time randomly choosing two new primes in the same range, then a mean of the number of iterations taken is recorded individually for each algorithm as such:

$$Mean = \frac{1}{500}\sum_{k=0}^{500} S(d_k)$$

Where d_k is the product of the two prime numbers randomly chosen in trial k.

This is then repeated, but with the upper end of the range being increased by 1000, so the range continues as 0-2000, 0-3000, 0-4000, all the way up until 0-1 000 000. This gives the graph below (See for complete code).



From this graph, it is clear that Fermat's factoring method is significantly faster than the traditional factoring, as it consistently takes fewer iterations to factor. They both display strong positive linear correlations, with an R^2 of 0.987 for Fermat's factoring method and 0.995 for simple factoring. They both pass approximately through the origin, but the slope of the line of best fit for Fermat's factoring theorem has a smaller gradient. This means that as the number being factored gets larger, the difference between Fermat's factoring method and the traditional method becomes greater and greater. Between these two methods of factoring, Fermat's factoring method shows the best performance. But how would it fare with the large numbers that would actually be used in RSA? While multiplying two prime numbers under 1 million may seem like it would produce a very large product, the product would still be orders of magnitude smaller than those used in RSA today. Using linear regression, I can approximate the number of iterations this would take. The current standard for RSA today is RSA-2048, meaning that the length of the semiprime part of the public key — the part that would need to be factored — is 2048 binary bits, or in base ten, 617 digits long. Recall at the start how I mentioned that RSA Laboratories offered $200,000 USD to anyone who could factor a very large number, and later withdrew the offer? Well here is the actual 2048 bit RSA key that they challenged people to factor:

251959084756578934940271832400483985714292821262040320277771378360436620207075
9555626401852588078440691829064124951508218929855914917618450280848912007284499268739
2807287776735971418347270261896375014971824691165077613379859095700097330459748808428
4017974291006424586918171951187461215151726546322822168699875491824224336372590851418
6546204357679842338718477444792073993423658482382428119816381501067481045166037730605
6201619676256133844143603833904414952634432190114657544454178424020924616515723350778
7077498171257724679629263863563732899121548314381678998850404453640235273819513786365
64391212010397122822120720357

To this day, the number above remains unfactored. Even after withdrawing the challenge, RSA Laboratories did not release the factors of this RSA key[14]. I will call this RSA key j, to avoid having to re-write it. Now approximately how long would it take to factor j using Fermat's factoring method? Since the smaller of the two prime factors of j is no greater than $\sqrt{j}$, I can plug $\sqrt{j}$ into the equation for the line of best fit of Fermat's factoring method $0.06x + 293$ for x.

[14] The RSA Factoring Challenge, 2007

This gives, to three significant figures:

$$Iterations = 0.06x + 293$$

$$Iterations = 0.06\sqrt{j} + 293$$

$$Iterations \approx 9.52 \times 10^{306}$$

Therefore it would take about 9.52×10^{306} iterations to factor this number using Fermat's method.

This is an absurdly large number, and shows how even using the more efficient algorithm, RSA is impossible to crack. To put it into perspective, there are about 10^{80} atoms in the known universe[15]. If somehow every single one of these atoms were used to each complete an iteration every millisecond, the amount of time it would take would be equal to:

$$Duration = \frac{9.52 \times 10^{306}}{10^{80}} \times \frac{1}{1000} seconds$$

$$Duration = 9.52 \times 10^{223} seconds$$

Therefore, it would take about 9.52×10^{223} seconds to factor this number. This is the equivalent of about 3.02×10^{214}millenia, using every atom in the known universe. Clearly, using this algorithm is unfeasible even if it were to be run on a more powerful computer.

But what if I were able to use Shor's algorithm — the factoring algorithm for quantum computers, that can theoretically factor a number n in on the order of $log^{3}(n)$ iterations. Plugging the large RSA-2048 key j into this formula, gives the number of iterations it would take Shor's algorithm to factor this number.

$$Iterations = log^{3}(n)$$

$$Iterations = log^{3}(j)$$

$$Iterations \approx 2.432 \times 10^{8}$$

Therefore, the number of iterations that Shor's algorithm would take is about 2.43×10^{8}.

[15] John Carl Villanueva, "How Many Atoms Are There in the Universe?," Universe Today, April 22, 2018, https://www.universetoday.com/36302/atoms-in-the-universe/.

Though this seems like a large number, for a computer it wouldn’t take very long to get through this many iterations. This is a lot fewer iterations than what would be necessary when using Fermat's method with a classical computer. If this were to be supported by a quantum computer, defeating RSA encryption would become possible.

Cracking an Example

Going back to the encrypted message from the start, which recall was:

$$\begin{array}{c} 5093797491863370116 \\ 9948631472706393745 \\ 4576814005525896971 \\ 9386295457731255133 \\ 2414985034203462679 \\ 7580559741887485016 \end{array}$$

I know the public key to be (8721800936746939897, 13195650934101362003), which is a 32-bit RSA key (much smaller than the standard 2048-bit key). This key would be known by anyone in the event of a real attack on RSA. The second number, 13195650934101362003, is the number I need to factor to crack this code. Using Fermat’s method to factor this number pretty quickly gives:

$$13195650934101362003 = 3610663543 \times 3654633221$$

So the prime factors are 3610663543 and 3654633221. Using these two factors, I can then generate the private key in the same way as the proper holder of the private key would (See Appendix A for key generation). Using RSA’s private key generation with these two prime factors of the public key, I get the private key to be 5254218084937185193. Now determining the message is simple. I know the decryption function $D(c)$, which uses the private key and the second part of the public key.

$$D(c) \equiv c^{5254218084937185193}\ mod\ 13195650934101362003$$

Plugging in each number of the encrypted message into $D(c)$, and then converting it back from a number to its corresponding letter (as shown in Appendix B), gives the final message:

$$"ENIGMA"$$

Therefore, I have cracked the encrypted message, which was “ENIGMA”.

Limitations

The main limitation of this investigation is that while I have experimental data from two different algorithms, I can only compare these to a theoretical model of Shor's algorithm. This means that any differences in the speed of a practical implementation of Shor's algorithm and the theoretical speed are not accounted for. The theoretical speed was the most accurate speed I could find, but I still do not think it is fully accurate.

Another limitation of this investigation is the extrapolation that was required. I had to work with numbers that could be factored on my (non quantum) computer in a reasonable amount of time, which meant using numbers much smaller than the ones used in RSA. I then had to extrapolate significantly, but because of this, the uncertainty in the results is very large.

Conclusion and Findings

Brute force attacks on RSA encryption are still unfeasible with classical computers, and it doesn't seem like that is going to change in the near future. Using the currently available algorithms, even if computers became 100 times more powerful, factoring numbers as large as those used in RSA encryption today would still be impossible. And even if it did become possible, then computers would also be fast enough to produce more secure RSA encryption using even larger keys, thus keeping RSA secure. RSA would lose security if either a significantly more efficient factoring algorithm is discovered, or if the attacker has a significantly more powerful computer than the person encrypting and decrypting the message. If quantum computers become able to efficiently run Shor's algorithm, this would breach the privacy of most people, and leave information like health records, banking data, and other personal information exposed until a new encryption method is found and widely adopted. But for now, RSA encryption cannot be cracked, leaving personal data secure from attacks.

Bibliography

"Caesar Cipher in Cryptography." GeeksforGeeks, May 11, 2023. https://www.geeksforgeeks.org/caesar-cipher-in-cryptography/.

J.A. Sloane, N., and R.K. Guy. "A001358 - Semiprimes." OEIS, March 7, 2024. https://oeis.org/A001358.

Krauthamer, Dave. "Q-Day: The Problem with Legacy Public Key Encryption." Help Net Security, July 11, 2022. https://www.helpnetsecurity.com/2022/07/15/legacy-public-key-encryption-problem/#:~:text=Using%20the%20public%20key%20in,an%20RSA%202048%2Dbit%20key.

"The RSA Challenge Numbers." RSA Laboratories - The RSA Challenge Numbers. Accessed March 7, 2024. https://web.archive.org/web/20130921041734/http://www.emc.com/emc-plus/rsa-labs/historical/the-rsa-challenge-numbers.htm.

"RSA Encryption." Brilliant Math & Science Wiki. Accessed March 7, 2024. https://brilliant.org/wiki/rsa-encryption/.

The RSA Factoring Challenge, 2007. https://web.archive.org/web/20130921043459/http://www.emc.com/emc-plus/rsa-labs/historical/the-rsa-factoring-challenge.htm.

Shor, Peter W. "Polynomial-Time Algorithms for Prime Factorization and Discrete Logarithms on a Quantum Computer." arXiv.org, January 25, 1996. https://arxiv.org/abs/quant-ph/9508027.

Villanueva, John Carl. "How Many Atoms Are There in the Universe?" Universe Today, April 22, 2018. https://www.universetoday.com/36302/atoms-in-the-universe/.

Weisstein, Eric W. "Fermat's Factorization Method." from Wolfram MathWorld. Accessed March 7, 2024. https://mathworld.wolfram.com/FermatsFactorizationMethod.html.

"What Is Encryption and How Does It Work? | Google Cloud." Google. Accessed March 7, 2024. https://cloud.google.com/learn/what-is-encryption.

Appendix A[16]:

How the RSA keys are generated:

Three values must be calculated, the private key, and the two values that make up the public key.

The second value of the public key, n, is calculated as the product of two large, randomly chosen prime numbers, p and q.

$$n = p \times q$$

The first value of the public key k is chosen as a value which has no common factors (other than 1) with n or $\phi(n)$. $\phi(n)$ is Euler's totient function, which can be written as:

$$\phi(n) = (p - 1)(q - 1)$$

Where p and q are the unique prime factors of the semiprime second value of the public key, n.

The private key h is chosen to be a number which satisfies the equation below.

$$h \times k \equiv 1 \; mod \; \phi(n)$$

Where k is the first value of the public key and $\phi(n)$ is Euler's totient function of the second value of the public key n.

There are often multiple possible values for the private key and the first value of the public key, both of which can be found using a guess and check method.

The public key (131, 533) with the related private key (11) is an example of a possible public-private key pair which satisfies all of the above conditions.

[16] "RSA Encryption," Brilliant Math & Science Wiki

Appendix B:

Decoding the encrypted message:

$$\begin{array}{c} 5093797491863370116 \\ 9948631472706393745 \\ 4576814005525896971 \\ 9386295457731255133 \\ 2414985034203462679 \\ 7580559741887485016 \end{array}$$

$$D(c) = c^{5254218084937185193} \, mod \, 13195650934101362003$$

Ciphertext, (c)	$D(c)$	Corresponding letter
5093797491863370116	7	E
9948631472706393745	16	N
4576814005525896971	11	I
9386295457731255133	9	G
2414985034203462679	15	M
7580559741887485016	3	A

Therefore the encrypted message is "ENIGMA"

Appendix C:

Code used to produce data seen in Figure 1, written in Python 3.12.2:

```
import random as r
import math
import csv

# "million-primes.txt" is a text file with each prime between 0 and 1 000 000 in incrementing order on
each line
file = open(r"million-primes.txt", 'r')

lines = file.readlines()

cutoffs = [0]
cap = 1_000
count = 0
```

```
for x in lines:
    if int(x) < cap:
        count += 1
    else:
        cutoffs.append(count)
        count += 1
        cap += 1_000
file.close()

# generates product of two primes in the given index range from the file
def gen_product(start, end):
    index1 = r.randint(start, end)
    index2 = r.randint(start, end)
     return int(lines[index1])*int(lines[index2])

# returns number of iterations to find two factors of n using Fermat's factoring method
def fermat(n, cap):
    iterations = 0
    a = math.ceil(math.sqrt(n))
    while (iterations < cap):
        b = (a*a) - n
        if (math.sqrt(b) == int(math.sqrt(b))):
            b = math.sqrt(b)
            return iterations
        a += 1
        iterations += 1
    return -1

# returns number of iterations taken to get factors of n using simple method
def factor_traditional(n):
    if n % 2 == 0:
        return 1
    for x in range(3, math.ceil(math.sqrt(n)) + 1, 2):
        if  n % x == 0:
            return ((x + 1) / 2)

fermat_means = {}
classical_means = {}

for x in range(999):
    if (x % 10 == 0):
        print(str((x)/10), "percent done")
    fermat_sum = 0
    classical_sum = 0
    sum_nums = 0
    cycles = 500
    for y in range(cycles):
        num = gen_product(0, cutoffs[x + 1])
        sum_nums += num
        fermat_sum += fermat(num, 1_000_000)
        classical_sum += factor_traditional(num)
    fermat_means[(x+1)*1000] = fermat_sum / cycles
    classical_means[(x+1)*1000] = classical_sum / cycles

# write results to csv (spreadsheet) file, “file.csv”
with open('file.csv', 'w', newline='') as file:
    writer = csv.writer(file)
    writer.writerow(['number', 'fermat method', 'classical method'])
    for keys, value in fermat_means.items():
        writer.writerow([keys, fermat_means[keys], classical_means[keys]])
```