Extended Essay

Topic: The limitations of RSA

Research Question: What are the limitations of RSA algorithm in terms of the key size, and how can these limitations be overcome?

Subject: Computer science

Word Count: 3990

Table of Contents

Introduction

I think cybersecurity which is the cornerstone of the digital world today, is one the areas that is widely considered to be the most important. Continuously the strategies are being discovered at the snap of a finger such as the disclosure of the weaknesses and a given way of filling them. On the one hand, that is generally an amazing breakthrough for new version of some product or technology. Now on the other side of the scale stands problems that are less visible but which have alarming outcomes such as the number of casualties increasing, personal data getting stolen, among other issues all of which cause negative brand awareness which leads to loss of trust among with customers, investors or partners, among. By providing me with intrigue, this case has resulted in my investigation of reliability in one of the most ancient algorithms, RSA, as to be able to prevent any future concerns.

In this Extended Essay, I will be investigating deeper into the critical limitations associated with the key length of one of the oldest and most renowned asymmetric cryptography algorithms, RSA. It is essential to recognize that while RSA has stood the test of time and remains a prevalent encryption system, its vulnerability to factorization attacks demands a thorough examination of its key length limitations and the implications for modern cybersecurity.

I believe that simulating this incredible encryption in Python would be an interesting approach to showcasing the relationship between key size and overall security. However, the precision of the results is questionable, primarily because the code was independently developed by me.

Background Information

What is RSA?

RSA (Rivest-Shamir-Adleman) is an asymmetric encryption method commonly used for secure data transfer. It is named for its founders, Ron Rivest, Adi Shamir, and Leonard Adleman, who suggested it in 1977. RSA is a public-key cryptosystem, which means it has two keys: a public key for encryption and a private key for decryption. [1]

The mathematical difficulty of factoring big composite numbers provides the foundation for RSA's security. The procedure generates two mathematically related keys: a public key and a private key. Anyone can use the public key to communicate encrypted data to the owner of the associated private key. The private key is kept secret and used to decode received communications.

Key Generation

1. Choose two distinct prime numbers, p and q. These should be large random prime numbers. The security of the RSA key relies on the difficulty of factoring the product of these two primes. [2]
2. Compute n, the modulus, by calculating the product of p and q: $n = p * q$.
3. Calculate Euler's totient function of n, denoted as $\varphi(n)$. Euler's totient function counts the positive integers up to n that are relatively prime to n. $\varphi(n) = (p-1)*(q - 1)$.

[1] A STUDY AND PERFORMANCE ANALYSIS OF RSA ALGORITHM, https://ijcsmc.com/docs/papers/June2013/V2I6201330.pdf. Accessed 4 Jan. 2023.
[2] "Two Prime Numbers - FasterCapital." FasterCapital, https://fastercapital.com/keyword/two-prime-numbers.html. Accessed 8 Feb. 2024.

4. Choose an integer e such that $1 < e < \varphi(n)$ and gcd(e, $\varphi(n)$) = 1. The value of e is typically chosen as a small prime, such as 65537 ($2^{16} + 1$), which is commonly used due to its efficient computation.
5. Compute the modular multiplicative inverse of e modulo $\varphi(n)$. In other words, find d such that (d * e) mod $\varphi(n)$ = 1. This can be done using the Extended Euclidean Algorithm or other modular inverse algorithms.

6. The public key is the pair (e, n), and the private key is the pair (d, n).[3] It's important to keep the private key secure, as it's used for decryption and signing operations.

An example of the RSA key generation step:

1) Let's say p = 61 and q = 53.
2) Calculate the modulus, n. The modulus is the product of p and q. n = p * q = 61 * 53 = 3233.
3) Calculate the totient, φ (phi). For RSA, $\varphi(n) = (p - 1) * (q - 1)$. $\varphi(n) = (61 - 1) * (53 - 1) = 60 * 52 = 3120$.
4) Common choices for e include 3, 17, or 65537. For this example, we'll use e = 17.
5) Compute the decryption exponent, d. The decryption exponent is the modular multiplicative inverse of e modulo $\varphi(n)$. In other words, (d * e) mod $\varphi(n)$ = 1. In this case, d = 2753.
6) The public key is composed of the modulus, n, and the encryption exponent, e. It is shared openly. Public key (e, n) = (17, 3233).

[3] "Public Private Keys - FasterCapital." FasterCapital, https://fastercapital.com/keyword/public-private-keys.html. Accessed 8 Feb. 2024.

7) The private key is composed of the modulus, n, and the decryption exponent, d. It must be kept secret and should not be shared. Private key (d, n) = (2753, 3233).

Encryption & Decryption

After getting both private and public keys, the method of encryption and unscrambling for a given content gets to be accessible through simple scientific operations. To start, let's speak to the message we need to scramble as the cipher content 'A,' which we'll indicate as '1' for qualification. Another, we assign our private key as 'd' and our open key as 'e.' These keys play a vital part within the encryption and decoding handle.

Encryption: Take our plaintext message 'A' marked as '1.' Raise it to the power of our public key 'e.' Apply modulo arithmetic with a large prime number 'N' to ensure that our cipher text remains within a manageable range: Cipher Text (C) = (1^e) % n The resulting 'C' will be our encrypted cipher text.

Decryption: To decrypt the cipher text and reveal the original message 'A,' we'll use our private key 'd' and perform the following mathematical operation: Decrypted Message

1 = (C^d) % n.[4]

[4] "The RSA Algorithm - University of Washington." DEPARTMENT OF MATHEMATICS, https://sites.math.washington.edu/~morrow/336_09/papers/Yevgeny.pdf. Accessed 20 Oct. 2023.

GCD (Greatest Common Divisor)

GCD plays a fundamental role in RSA by ensuring that the chosen prime numbers are coprime, which contributes to the security of the algorithm.

GCD is a mathematical concept used to find the largest positive integer that divides two or more numbers without leaving a remainder. In other words, the GCD is the largest number that divides the given numbers evenly.

For example, let's consider the numbers 12 and 18. The divisors of 12 are 1, 2, 3, 4, 6, and 12, while the divisors of 18 are 1, 2, 3, 6, 9, and 18. The largest number that appears in both lists is 6, so the GCD of 12 and 18 is 6.

The GCD is often calculated using the Euclidean algorithm, which involves repeatedly dividing the larger number by the smaller number and replacing the larger number with the remainder until the remainder becomes zero. The final non-zero remainder obtained using this process is the GCD.

Co-prime

Co-prime numbers, also known as relatively prime or mutually prime numbers, are two positive integers that have no common positive integer divisors other than 1. In other words, when you calculate the greatest common divisor (GCD) of two co-prime numbers, the result is 1.

For example, the numbers 6 and 35 are co-prime because their only common divisor is 1. The divisors of 6 are 1, 2, 3, and 6, while the divisors of 35 are 1, 5, 7, and 35. The only divisor they have in common is 1, so they are co-prime. On the other hand, numbers like 8 and 12 are not co-prime because they share a common divisor other than 1. The divisors of 8 are 1, 2, 4, and 8, while the divisors of 12 are 1, 2, 3, 4, 6,

and 12. They both have a common divisor of 2, so they are not co-prime. Co-prime numbers are frequently used in number theory and various mathematical applications, including cryptography, modular arithmetic, and other areas of mathematics.

Modular Inverse

RSA encryption heavily relies on the Modular Inverse to ensure that decryption operations within modular arithmetic are inverses of their corresponding encryption operations. It is extremely challenging computationally speaking, even with access to both public and private keys, to compute the private exponent "d" from the public counterpart "e". The importance of this concept cannot be underestimated as it guarantees dependable communication through sturdy cryptographic protocols facilitated by RSA's use of its Modular Inverse for security purposes.

Modular inverse algorithms are utilized to compute the opposite of a given number with respect to a specific modulus. The principle behind modular arithmetic is that there exists another value, denoted as "b", which fulfills (a * b) mod m = 1 where "a" denotes the original number and "m" denotes its modulo. Essentially, this means that upon being multiplied by "a" and then subjected to reduction using "m", it produces an output equaling one.

Step 1: Find the modular multiplicative inverse of 7 modulo 15 using Euler's algorithm. To find the modular multiplicative inverse, we need to find a number, let's call it a, such that (7 * a) mod 15 = 1. Using Euler's algorithm, we have: 15 = 2 * 7 + 1 7 = 7 * 1 + 0 Working backward: 1 = 15 - 2 * 7 Therefore, the modular multiplicative inverse of 7 modulo 15 is -2.

Step 2: Multiply both sides of the equation by the modular multiplicative inverse. (-2) * (7 * x) mod 15 = (-2) * 1 mod 15 Simplifying: (-14 * x) mod 15 = (-2) mod 15

Step 3: Compute the result on the left side. (-14 * x) mod 15 is equivalent to (1 * x) mod 15 since (-14 mod 15) is congruent to 1. So, the equation simplifies to: x mod 15 = (-2) mod 15.

Step 4: Find the value of x. To find the value of x, we need to find an integer that is congruent to -2 modulo 15. Since -2 is congruent to 13 modulo 15, we can conclude that: x = 13 Therefore, the value of x that satisfies the equation (7 * x) mod 15 = 1 using Euler's algorithm is x = 13.[5]

Euler's totient function

The Euler Totient Function (n) is crucial to RSA because it makes it computationally challenging for an attacker to factorize n into its prime factors (p and q) from the public key components (n and e). This helps to assure the security of the method. The existence of the private exponent d, which for safe encryption and decryption, is ensured by the condition that e and (n) be co-prime.

Euler's totient function, also known as Euler's phi function, is a mathematical function denoted by the symbol φ (phi). It is named after the Swiss mathematician Leonhard Euler, who introduced it in the 18th century. The totient function is defined for a positive integer n and calculates the count of positive integers that are relatively prime to n, i.e., the number of positive integers less than n that do not share any common divisors with n except for 1.[6]

[5] "Modular Inverses (Article) | Cryptography." *Khan Academy*, Khan Academy, www.khanacademy.org/computing/computer-science/cryptography/modarithmetic/a/modular-inverses. Accessed 22 Jan. 2024.

[6] "The RSA Algorithm - University of Washington." DEPARTMENT OF MATHEMATICS, https://sites.math.washington.edu/~morrow/336_09/papers/Yevgeny.pdf. Accessed 20 Oct. 2023.

Euler's totient function (also called the Phi function) counts the number of positive integers less than n that are coprime to n.[7]

Formally, for a positive integer n, the totient function φ(n) is defined as the number of positive integers k ($1 \leq k \leq n$) such that gcd(n, k) = 1, where gcd stands for the greatest common divisor. In other words:

φ(n) = Count of {k ∈ [1, n] : gcd(n, k) = 1}

Some important properties of Euler's totient function are:

1. If n is a prime number, then φ(n) = n - 1, as all positive integers less than n are relatively prime to n.
2. If p is a prime number and k is a positive integer, then $\varphi(p^k) = p^k - p^{(k-1)}$.
3. If m and n are co-prime (i.e., gcd(m, n) = 1), then φ(m * n) = φ(m) * φ(n).
4. In general, for any positive integer n, φ(n) = n * (1 - 1/p1) * (1 - 1/p2) * ... * (1 - 1/pk), where p1, p2, ..., pk are the distinct prime factors of n.
5. Euler's totient function finds applications in various areas of number theory, cryptography, and algorithms, especially in RSA encryption, where it is used to calculate the totient of large numbers as part of the key generation process. The totient function also plays a crucial role in modular arithmetic and the study of groups and cyclic groups.[8]

[7] "Totient Function - FasterCapital." FasterCapital, https://fastercapital.com/keyword/totient-function.html. Accessed 8 Feb. 2024.
[8] Nitaj, Abderrahmane. The Mathematical Cryptography of the RSA Cryptosystem, https://nitaj.users.lmno.cnrs.fr/RSAnitaj1.pdf. Assessed 10 Dec. 2023.

Certain RSA Limitations

Key Size:

One of the most critical aspects of RSA is the size of its keys. RSA encryption relies on the difficulty of factoring the product of two large prime numbers. This product is known as the modulus, denoted as N. The security of RSA is closely tied to the length of N, which is determined by the size of the prime numbers used in its generation.

As computational power advances, the required key size to ensure security must also increase. Smaller key sizes become progressively less secure as attackers can use more advanced techniques to factor N. For example, the widely used 1024-bit RSA key, which was once considered secure, is now vulnerable to attacks due to the increased processing power and more sophisticated algorithms available to attackers. Today, a key size of 2048 bits or higher is recommended for RSA to maintain adequate security.[9]

The reason behind this is that as computing capabilities improve, it becomes more feasible for attackers to perform brute force or mathematical factorization attacks on smaller keys. Longer key sizes make these attacks significantly more time-consuming and resource-intensive, making them impractical for most attackers.

Performance Impact:

While longer key sizes enhance security, they also come with a performance cost. Larger RSA keys require more computational resources for both encryption and

[9] "RSA Encryption Maximum Data Size." *RSA Encryption Maximum Data Size - Mbed TLS Documentation*, https://mbed-tls.readthedocs.io/en/latest/kb/cryptography/rsa-encryption-maximum-data-size/#:~:text=RSA%20is%20only%20able%20to,is%20not%20designed%20for%20this. Accessed 18 Jan. 2023.

decryption processes. This performance impact can be a concern for systems that demand high-speed data transmission, such as real-time communication and data transfer applications.

The time required to perform RSA encryption and decryption operations scales with the key size, which means that as key sizes increase, processing times also increase. This trade-off between security and performance should be carefully considered when implementing RSA in a system. In some cases, alternative cryptographic algorithms with lower computational requirements may be preferred if performance is a critical factor.

Key Management:

Overseeing RSA keys, particularly in complex frameworks, can be challenging. RSA employments both open and private keys, and securing the private key is of most extreme significance. In case an assailant picks up get to to the private key, they can decode delicate information, mimic the substance with the private key, and possibly cause extreme security breaches.

Effective key management practices are essential to safeguard RSA keys. This includes secure storage of keys, regular key rotation, and access controls to restrict who can use or access the private key. Failure to manage RSA keys properly can have severe consequences for an organization's security posture.

Quantum Computing Threat:

One of the emerging threats to RSA is the advent of quantum computing. Quantum computers have the potential to efficiently factor large numbers using algorithms like Shor's algorithm.[10] Unlike classical computers, quantum computers can perform certain calculations exponentially faster, which includes factoring large numbers.

This poses a significant threat to RSA's security because the strength of RSA encryption is directly related to the difficulty of factoring large numbers. If quantum computers capable of running Shor's algorithm become practical, they could break RSA encryption, rendering all data encrypted with RSA vulnerable.

As a result, there's progressing inquire about into post-quantum cryptography, which points to create encryption strategies that are safe to assaults from quantum computers. Organizations and security specialists are closely observing headways in quantum computing and planning for a future where RSA's security may not be adequate.

Security:

The concerns regarding the security of RSA, particularly in relation to the potential compromise of the private key, highlight some of the challenges associated with traditional public-key cryptosystems. The reliance on the secrecy of the private key for decryption poses a significant risk in scenarios where the private key is exposed or compromised. So, there appears an issue of loss of private key.

[10] Moore, Tristan. Quantum Computing and Shor's Algorithm, 7 June 2016, https://sites.math.washington.edu/~morrow/336_16/2016papers/tristan.pdf. Accessed 12 Feb. 2024.

The criticism outlined by Patidar and Bhartiya underscores the fundamental vulnerability of RSA encryption: if an unauthorized party gains access to the private key, they can decrypt messages intended to be confidential. [11]This highlights the importance of safeguarding private keys through robust key management practices, including encryption, access controls, and regular key rotation. However, there is another problem that arises from preferring robustness when it comes to manipulating the speed of encryption and decryption. Setting up strong keys consumes a significant amount of time, as will be demonstrated in the experiment section, which outlines how slow it can be.

Overcoming RSA Key Size Limitations

To enhance the security of RSA encryption, increasing the key size is a fundamental approach. The key size directly influences the difficulty of factoring the product of two large prime numbers (N), a task essential for breaking RSA. Commonly used key sizes, such as 2048 bits and 3072 bits, provide a reasonable level of security for today's applications. However, for long-term security, especially in the face of advancing computational capabilities, key sizes of 4096 bits or more are recommended. The exponential growth in key size significantly raises the computational complexity of factoring N, bolstering the resilience of RSA against various cryptographic attacks.

To address potential performance bottlenecks associated with RSA encryption, a hybrid encryption strategy can be employed. This involves using RSA to establish a secure communication channel for exchanging a symmetric encryption key. Once this secure channel is established, the bulk of the data can be encrypted and decrypted

[11] Al-Kaabi, Shaheen Saad, and Samir Brahim Belhaouari. “Methods toward Enhancing RSA Algorithm: A Survey.” SSRN, 2 July 2019, https://shorturl.at/hsCGN. Accessed 4 Feb. 2024.

using a faster symmetric encryption algorithm, such as Advanced Encryption Standard (AES). By incorporating symmetric encryption for data transmission, the computational overhead of RSA is minimized, resulting in improved overall system performance. [12]

Effective key management practices are essential to the robustness of any encryption system, including RSA. Hardware Security Modules (HSMs) provide a secure environment for storing and managing private keys, protecting them from unauthorized access. Regular key rotation is crucial for minimizing the impact of potential key compromise. Compromised keys should be promptly revoked, and stringent access controls must be enforced to prevent unauthorized use.[13] A comprehensive key management strategy ensures the long-term integrity and security of the RSA encryption system.

Looking toward the future, it's essential to monitor developments in post-quantum cryptography. With the advent of quantum computers, traditional cryptographic algorithms, including RSA, may become vulnerable to rapid factorization by quantum algorithms. Researchers are actively exploring and developing new cryptographic algorithms that resist attacks from quantum computers. Organizations should stay informed about these advancements and be prepared to transition to post-quantum cryptographic algorithms when the threat of quantum computing to RSA's security

[12] Sa'idu Sani. "A Comparative Analysis of Cryptographic Algorithms: AES & RSA and Hybrid Algorithmfor Encryption and Decryption." Ijisrt, Aug. 2022, https://ijisrt.com/assets/upload/files/IJISRT22AUG773.pdf. Accessed 9 Feb. 2024.
[13] Mavrovouniotis, Stathis, and Mick Ganley. "Hardware Security Modules." SpringerLink, Springer New York, 1 Jan. 1970, https://link.springer.com/chapter/10.1007/978-1-4614-7915-4_17. Accessed 9 Jan. 2023.

becomes imminent. Proactive planning for this transition is vital for maintaining the confidentiality and integrity of sensitive data in the long run.[14]

One example of enhancement of RSA is the combination of RSA and ElGamal algorithm.

1. Choose a random prime number p.
2. Choose two random number, g and x , where (g < p) and (x < p).
3. Calculate y = g*x mod p.
4. y is the public key and x is the private key.

Combining RSA and ElGamal in cryptographic schemes can offer a balance between security and computational efficiency. In the author's proposal, the utilization of 256-bit prime numbers instead of the traditional 1024-bit primes in RSA key generation aims to reduce computational overhead while maintaining adequate security levels.

RSA relies on the difficulty of factoring large composite numbers, typically achieved through the use of large prime numbers. The security of RSA hinges on the challenge of factoring the product of two large primes. However, as computational power increases over time, what constitutes a "large" prime number for RSA evolves.

On the same note, the ElGamal mechanism algorithms deduce rotating information in discrete fields or elliptical curves. The system however acts as an efficient single key cryptography method which possesses some enhancements over RSA, employed the most widely, particularly under some specific conditions. By combining both RSA and ElGamal, the proposal grips the strengths of each scheme while mitigating their weaknesses. The use of

[14] "Methods Toward Enhancing RSA Algorithm: A Survey." SSRN, https://papers.ssrn.com/sol3/Delivery.cfm/SSRN_ID3819524_code3121922.pdf?abstractid=3819524&mirid=1. Accessed 9 July. 2023.

smaller prime numbers in RSA key generation reduces computational overhead, which is particularly beneficial in resource-constrained environments or applications where efficiency is paramount.[15]

On the one hand, this hybrid scheme is sufficiently secure provided that the factoring of large composite numbers and the solution of discrete logarithms are considered to be hard problems. The cryptography with prime numbers of lesser magnitude will continue to be computationally complex, which will result from ElGamal and RSA crypto systems hybrid challenges.

Overall, the proposal will contribute to obtaining such balance of security and efficiency as creating cryptographic algorithm whose both key and protection against attacks can be optimized. This approach addresses the ever changing nature of cryptographies where ability to adapt and innovate is keys to protecting against new type of malicious threats.

[15] Al-Kaabi, Shaheen Saad, and Samir Brahim Belhaouari. "Methods toward Enhancing RSA Algorithm: A Survey." SSRN, 2 July 2019, https://shorturl.at/hsCGN. Accessed 4 Feb. 2024.

Experiment Methodology

Overview

1) The Key Length Experiment aims to explore how the size of the prime numbers used in generating the keys affects the security and efficiency of RSA encryption and decryption. In general, longer key lengths tend to provide stronger security, as they increase the difficulty of factoring the keys and thus make it more challenging for attackers to break the encryption. However, longer keys also come with a performance cost, as longer key lengths require more computational resources and time for encryption and decryption operations. A Python script provides the whole process of RSA encryption algorithm respect to the key length. RSA simulation script

2) The second experiment aims to provide valuable information on how the key length affects overall security. I made a deliberate challenge from mobilefish[16] website to simulate the process. It requires us to decrypt the ciphertext using given long-bits numbers.

Investigation

1) Following the Python code used 'random' function to generate random both public and private keys respect to the number of bits. Moreover, it has nine functions symbolized by 'def(name)' – gcd(), mod_inverse(), extended_gcd(), is_prime(), generate_prime(), generate_keypair(), encrypt(), decrypt(), and main().

[16] Lie, Robert. "Online RSA Key Generation." *Mobilefish.Com - Online RSA Key Generation*, www.mobilefish.com/services/rsa_key_generation/rsa_key_generation.php. Accessed 3 Jan. 2023.

2) To clarify, the values n, e, c and φ(n) represent the product of two prime numbers, public/encryption key, ciphertext, and phi function are given respectively. The quantities are displayed under the input title.

The table below displays the number of bits (binary digits) for each of these components. Find the number of bits

Table 1. The table below displays the number of bits (binary digits) for each of given components

n	1024
e	17
c	1023
φ(n)	1024

Procedure in Steps

1) A. Inserting high/low bits

B. Measuring the time takes to complete.

C. Comparing the security of two cases.

2) A. Private key: As we know that (d * e) mod φ(n) = 1; In order to find d from this equation, we use the modular inverse function to determine the multiplicative inverse of e modulo φ(n). Therefore, $d \equiv e^{-1} \bmod \varphi(n)$.

B. Decryption: Text Message = (c^d) % n. By using this equation, we will be able to decrypt the ciphertext and find our FLAG.

The Experimental Results

1) Table of Key Generation Results (Text Message: “Hello RSA”)

Table 2. Least significant bits

Key Size (bits)	Encryption (seconds)	Decryption (seconds)
10	8.487701416015625e-05	8.0108642578125e-05
20	0.0006079673767089844	0.0001800060272216797
30	0.0007669925689697266	0.00022101402282714844
40	0.0009050369262695312	0.00039505958557128906
50	0.0009357929229736328	0.0006108283996582031
60	0.0009860992431640625	0.0008409023284912109
70	0.0010437965393066406	0.001123189926147461

Table 3. Most significant bits

Key Size (bits)	Encryption (seconds)	Decryption (seconds)
100	0.0018916130065917969	0.002602815628051758
200	0.006426095962524414	0.00992894172668457
300	0.016495943069458008	0.017183780670166016
400	0.039626121520996094	0.05285501480102539
500	0.05984187126159668	0.07729101181030273
600	0.08260893821716309	0.12044405937194824
700	0.1121511459350586	0.1461191177368164

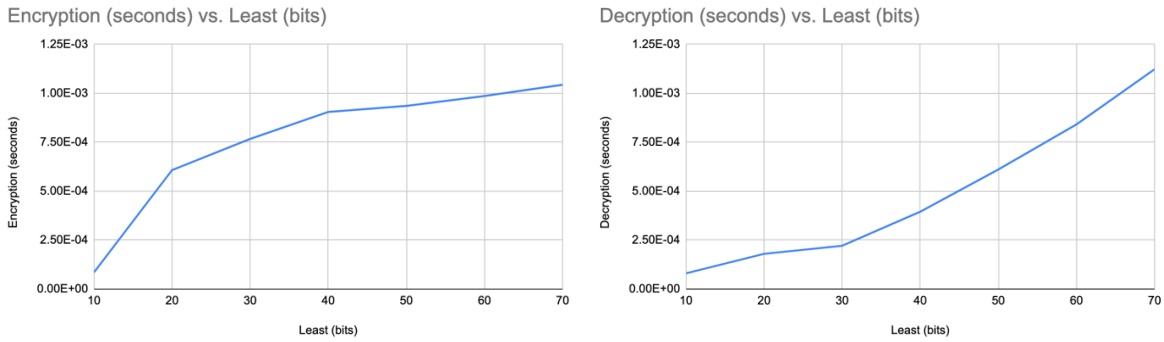

Figure 1. Graphical presentation of least bit (LSB)

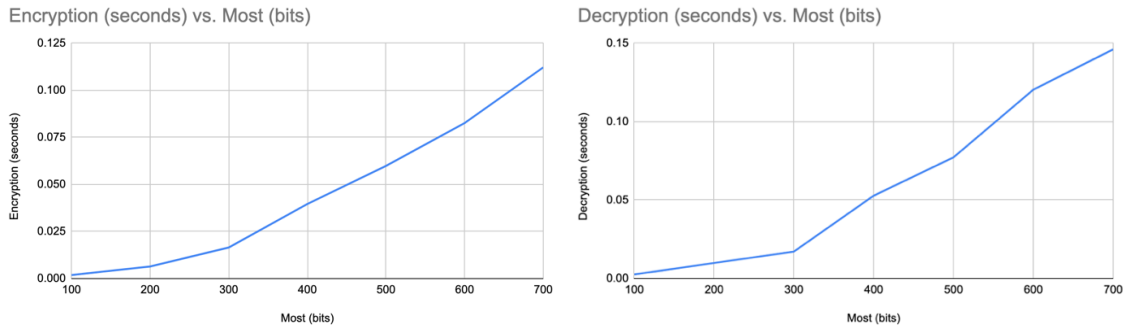

Figure 2. Graphical presentation of most bit (MSB)

2) By using the dcode.fr online decoder[17], we will be easily able to retrieve our flag without having to spend much time completing mathematical steps. In this case, both 'n' and 'phi' have the highest number of bits, which is 1024. Therefore, it would not take much time to decrypt. However, the limitation of RSA becomes evident in this experiment due to its vulnerable security.

[17] dCode. "RSA Cipher Calculator - Online Decoder, Encoder, Translator." *Calculator - Online Decoder, Encoder, Translator*, www.dcode.fr/rsa-cipher. Accessed 16 Feb. 2023.

What if a skilled hacker were to successfully decipher our RSA-encrypted password, even with its 1024-bit complexity? The security of our messages is paramount, as the consequences of losing passwords and private information can be dire for users.
In an increasingly interconnected digital world, the protection of sensitive data is essential. If a hacker were to breach our security measures and gain access to user passwords and private information, it could have devastating repercussions. Identity theft, financial loss, and personal privacy invasion are just a few of the potential dangers that users may face.

Hence, while using shorter key lengths may save precious time, it inevitably compromises the overall security. This fact is exemplified by my experiment, which demonstrated how easily a 1024-bit encryption with low bit length can be cracked using an online decryption tool. The cracking outcomes are displayed under the output title.

FLAG -> 350777886000146088143164915826747963543312578394843025817 5

Analysis

The following line graphs vividly illustrate the intricate connection between time and the number of RSA key bits, meticulously ordered from 10 bits to 700 bits. Further investigation into the analysis of these graphs reveals a startling realization: each key generation process's time complexity may be roughly estimated as O(n log(n)).

So, how efficient is O(n log(n)? According to some resources[18], it is considered to be good!!! However, this experiment is primarily done on small bits. If the number of bits increases to 2048 bits, these graphs would tend to exhibit worse time complexities.

These graphs encapsulate the essence of RSA key generation, providing a visual journey through the computational intricacies involved in producing cryptographic keys. As the bit count steadily escalates, we observe a logarithmic increase in the time required, indicating that the algorithm's efficiency scales in a way that aligns with O(n log(n)) complexity.

The second experiment aimed to demonstrate the vulnerability of RSA encryption with shorter key lengths. Using factors with components of a maximum of 1024 bits, the cracking process of the cipher becomes relatively simple when using an online tool. This exposes a security weakness in the RSA algorithm. However, what if a message is encrypted with a longer key, such as over 2048 bits? In such cases, this action would significantly improve our security, as longer keys make it much more

[18] KOCAK, Burak. “What Are These Notations"o(n), O(NLOGN), o(N2)" with Time Complexity of an Algorithm?” Medium, Medium, 1 Feb. 2023, https://shorturl.at/gKOT4. Accessed 16 Feb. 2023.

challenging for hackers to decipher the mathematical calculations, not only through online platforms but also through future strategies to break down.

According to these two factors, both are considered key weaknesses of the RSA encryption algorithm. The reason is that the relationship between key length and time complexity is inverse.[19] As the key length increases, the time it takes to encode and decrypt our message also increases in the first case. However, when there is a short key, it becomes easier for hackers to find the message. Rather than risking the loss of our information to the attacker, I would prefer to wait.

Limitations

While testing the time complexity of RSA, I faced a limitation related to the accuracy of my initial experiment involving the graphing of time complexity for various processes. I suspect that my independently written Python code may contain constraints that affect the precision of the time measurements.

Additionally, as the focus of my research, I have chosen to study one of the oldest security algorithms. However, I have encountered challenges in finding the latest resources, which has made it difficult for me to stay current. Many of the papers I have referenced were published several years ago. For instance, in the past, we did not have access to computationally powerful computers, which resulted in the RSA algorithm being relatively slow. It is inevitable to consider this as a limitation.

[19] Lenstra, Arjen Klaas. "Key Lengths - Contribution to The Handbook of Information Security." Blkcipher, https://blkcipher.pl/assets/pdfs/NPDF-32.pdf. Accessed 8 Feb. 2024.

Conclusion

Cybersecurity is an ever-evolving area that continually introduces new concepts and knowledge. As I am passionate about promoting safety, I have explored the RSA algorithm, exploring into its key limitations. This exploration aims to contribute both to knowledge enhancement and to better prepare future respondents.

In an effort to outline the limitations of RSA and propose potential solutions, various aspects are discussed. The primary drawbacks of this algorithm include issues with Key Size, Performance Impact, Key Management, and the threat posed by Quantum Computing. However, by increasing the key length, implementing hybrid encryption, adopting strong key management practices, and preparing for post-quantum cryptography, a proper approach can be established to prevent the forthcoming dire consequences.

I think the two experiments as prescribed on this EE revealed how RSA itself has a complicated key management mechanism. When there is a longer key, the overall security of the message becomes stronger, but the process of decrypting seems time-consuming. Also, I believe this EE enables us to predict the computational demands of RSA key generation with a high degree of accuracy, making it a fundamental cornerstone for securing data in modern cryptographic systems. Understanding this time complexity empowers us to make informed decisions about key length, balancing security with performance, and ensuring the resilience of our cryptographic systems in an ever-evolving digital landscape.

Works Cited

Lenstra, Arjen Klaas. “Key Lengths - Contribution to The Handbook of Information Security.” Blkcipher, https://blkcipher.pl/assets/pdfs/NPDF-32.pdf. Accessed 8 Feb. 2024.

KOCAK, Burak. “What Are These Notations"o(n), O(NLOGN), o(N2)" with Time Complexity of an Algorithm?” Medium, Medium, 1 Feb. 2023, https://shorturl.at/gKOT4. Accessed 16 Feb. 2023.

“Totient Function - FasterCapital.” FasterCapital, https://fastercapital.com/keyword/totient-function.html. Accessed 8 Feb. 2024.

“Two Prime Numbers - FasterCapital.” FasterCapital, https://fastercapital.com/keyword/two-prime-numbers.html. Accessed 8 Feb. 2024.

“Public Private Keys - FasterCapital.” FasterCapital, https://fastercapital.com/keyword/public-private-keys.html. Accessed 8 Feb. 2024.

Nitaj, Abderrahmane. The Mathematical Cryptography of the RSA Cryptosystem, https://nitaj.users.lmno.cnrs.fr/RSAnitaj1.pdf. Assessed 10 Dec. 2023.

dCode. “RSA Cipher Calculator - Online Decoder, Encoder, Translator.” Calculator - Online Decoder, Encoder, Translator, www.dcode.fr/rsa-cipher. Accessed 16 Feb. 2023.

Lie, Robert. “Online RSA Key Generation.” Mobilefish.Com - Online RSA Key Generation, www.mobilefish.com/services/rsa_key_generation/rsa_key_generation.php. Accessed 3 Jan. 2023.

Cobb, Michael. “What Is the RSA Algorithm? Definition from Search Security.” Security, TechTarget, 4 Nov. 2021, www.techtarget.com/searchsecurity/definition/RSA#:~:text=The%20RSA%20algorithm%20(Rivest%2DShamir%2DAdleman)%20is%20the,an%20insecure%20network%20such%20as. Accessed 10 Dec. 2023.

“RSA Encryption Maximum Data Size.” RSA Encryption Maximum Data Size - Mbed TLS Documentation, https://mbed-tls.readthedocs.io/en/latest/kb/cryptography/rsa-encryption-maximum-data-size/#:~:text=RSA%20is%20only%20able%20to,is%20not%20designed%20for%20this. Accessed 18 Jan. 2023.

A STUDY AND PERFORMANCE ANALYSIS OF RSA ALGORITHM, https://ijcsmc.com/docs/papers/June2013/V2I6201330.pdf. Accessed 4 Jan. 2023.

“Methods Toward Enhancing RSA Algorithm: A Survey.” SSRN, https://papers.ssrn.com/sol3/Delivery.cfm/SSRN_ID3819524_code3121922.pdf?abstractid=3819524&mirid=1. Accessed 9 July 2023.

“The RSA Algorithm - University of Washington.” DEPARTMENT OF MATHEMATICS, https://sites.math.washington.edu/~morrow/336_09/papers/Yevgeny.pdf. Accessed 20 Oct. 2023.

“Modular Inverses (Article) | Cryptography.” Khan Academy, Khan Academy, www.khanacademy.org/computing/computer-science/cryptography/modarithmetic/a/modular-inverses. Accessed 22 Jan. 2024.

Moore, Tristan. Quantum Computing and Shor's Algorithm, 7 June 2016, https://sites.math.washington.edu/~morrow/336_16/2016papers/tristan.pdf. Accessed 12 Feb. 2024.

Mavrovouniotis, Stathis, and Mick Ganley. "Hardware Security Modules." SpringerLink, Springer New York, 1 Jan. 1970, https://link.springer.com/chapter/10.1007/978-1-4614-7915-4_17. Accessed 9 Jan. 2023.

Sa'idu Sani. "A Comparative Analysis of Cryptographic Algorithms: AES & RSA and Hybrid Algorithmfor Encryption and Decryption." Ijisrt, Aug. 2022, https://ijisrt.com/assets/upload/files/IJISRT22AUG773.pdf. Accessed 9 Feb. 2024.

Al-Kaabi, Shaheen Saad, and Samir Brahim Belhaouari. "Methods toward Enhancing RSA Algorithm: A Survey." SSRN, 2 July 2019, https://shorturl.at/hsCGN. Accessed 4 Feb. 2024.

Appendices

1)

RSA simulation

```
import random
import time
import math
# Function to calculate the Greatest Common Divisor (GCD) of two numbers
def gcd(a, b):
    while b:
        a, b = b, a % b
    return a

# Function to calculate the modular inverse of 'a' modulo 'm'
def mod_inverse(a, m):
    g, x, y = extended_gcd(a, m)
    if g != 1:
        raise Exception("Modular inverse does not exist")
    return x % m

# Function to calculate the extended GCD of two numbers
def extended_gcd(a, b):
    if a == 0:
        return b, 0, 1
    else:
        gcd, x, y = extended_gcd(b % a, a)
        return gcd, y - (b // a) * x, x

# Function to check if a number is prime using the Miller-Rabin primality test
def is_prime(n, k=5):
    if n <= 1:
        return False
    if n <= 3:
```

```
        return True
    if n % 2 == 0 or n % 3 == 0:
        return False
    for _ in range(k):
        a = random.randint(2, n - 2)
        if pow(a, n - 1, n) != 1:
            return False
    return True

# Function to generate a prime number with a given number of bits
def generate_prime(bits):
    while True:
        num = random.getrandbits(bits)
        if is_prime(num):
            return num

# Function to generate a key pair (public key and private key) for encryption
def generate_keypair(bits):
    p = generate_prime(bits)
    q = generate_prime(bits)
    n = p * q
    phi = (p - 1) * (q - 1)

    while True:
        e = random.randint(2, phi - 1)
        if gcd(e, phi) == 1:
            break

    d = mod_inverse(e, phi)
    return (e, n), (d, n)

# Function to encrypt a message using the public key
def encrypt(public_key, plaintext):
    e, n = public_key
```

```
    cipher_text = [pow(ord(char), e, n) for char in plaintext]
    return cipher_text

# Function to decrypt a cipher text using the private key
def decrypt(private_key, cipher_text):
    d, n = private_key
    decrypted_text = ''.join([chr(pow(char, d, n)) for char in cipher_text])
    return decrypted_text

# Main function to execute the encryption and decryption process
def main():
    bits = int(input("Enter the number of bits for key generation: "))

    public_key, private_key = generate_keypair(bits)

    print("Generated public key (e, n):", public_key)
    print("Generated private key (d, n):", private_key)

    message = input("Enter a message to encrypt: ")

    start_encryption = time.time()  # Record the start time for encryption
    encrypted = encrypt(public_key, message)
    end_encryption = time.time()  # Record the end time for encryption

    print("Encrypted message:", encrypted)

    start_decryption = time.time()  # Record the start time for decryption
    decrypted = decrypt(private_key, encrypted)
    end_decryption = time.time()  # Record the end time for decryption

    print("Decrypted message:", decrypted)

    encryption_time = end_encryption - start_encryption
    decryption_time = end_decryption - start_decryption
```

```
    print("Encryption Time:", encryption_time, "seconds")
    print("Decryption Time:", decryption_time, "seconds")

if __name__ == "__main__":
    main()
```

2)

Input

n =

10888529522831420795550561964927643497984456388669518397677111644807526540806566569151100256517991311612461376226662727165374449311050680596301918278967089725400947298399635820981234278600923003503254775323983399096525700694230891098519948061596588990170896469529892850853220033040894177817786618308266260734 7

e = 65537

c =

82387872436748348593453076180268891359180948547693065827225149655835925258034118402571046904225197163610835925272639379288061378617023501878748102047461831941072894962106572933493298789909340527294682192316522477679670135709883125365772129635122198835555238072513983445267589246508700831704953485512064353146

φ(n) =

10888529522831420795550561964927643497984456388669518397677111644807526540806566569151100256517991311612461376226662727165374449311050680596301918278967087638273741215707019266237573296904082403708807261651932

34762699476178388953140801003485676705682456546709681527100405470740582454016459554552393415698365769 6

Output

d =

35805529662105367893695189115485244217016802666008024625530529785807559772483073093332829032183229436281818685484658808480855815356585320892152927480662786775781222353739410135753229797149683367003144070656399601894565073235016729084704231169977899907081212192592053471825065274954073076256184846804357154081

message (flag) =

350777886000146088143164915826747963543312578394843025817 5

p =

10566800549051139758394463118138363396468329322896004878605680762478039765204860143942097422555632670096234819568308975447906101575651842378432579725610403

q =

10304471511775786407152973491678605009529615152240715607758289732745013012298115020681751486509703571771645208356144849346599767201530789963816345953339249

Find the number of bits

```
import math


# Define the large number as a string
large_number_str = ' '   #Enter the ciphertext in decimal


# Convert the string to an integer
large_number = int(large_number_str)


# Calculate the number of bits required to represent the number
num_bits = math.ceil(math.log2(large_number + 1))


print(f"Number of bits required: {num_bits}")
```

Modular Inverse function

```
def extended_gcd(a, b):
    if a == 0:
        return (b, 0, 1)
    else:
        gcd, x, y = extended_gcd(b % a, a)
        return (gcd, y - (b // a) * x, x)


def mod_inverse(a, m):
    gcd, x, y = extended_gcd(a, m)
    if gcd != 1:
        raise ValueError("The modular inverse does not exist.")
    else:
        return x % m


# Example usage:
e1 = 3
n = 100
inverse = mod_inverse(e1, n)
print(f"The modular inverse of {e1} modulo {n} is {inverse}")
```