The limitations of RSA

By: Erdembileg.A

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

Computer science

… words

Table of ontents

Introduction

In the rapidly evolving digital world, cybersecurity is becoming an increasingly essential field for us. Numerous new strategies are emerging at the blink of an eye, including the detection of vulnerabilities and injection techniques. On the one hand, this is great news for technological development. On the other hand, the world is grappling with dire consequences, such as an increase in the number of victims, the theft of personal information, and damage to the reputations of companies, resulting in a loss of trust from users, investors, and partners, among other issues.

In this 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.

RSA (Rivest–Shamir–Adleman), named after its inventors Ron Rivest, Adi Shamir, and Leonard Adleman, has been a stalwart in the realm of secure data transmission and digital communication for several decades. Its longevity and continued relevance can be attributed to its mathematical elegance and robust security model, based on the presumed difficulty of factoring the product of two large prime numbers.

Nonetheless, as computing power has advanced exponentially over the years, RSA's original key length recommendations have become increasingly inadequate in the face of modern threats. The security of RSA encryption fundamentally relies on the practical impossibility of efficiently factoring the product of two large prime numbers. Thus, the size of the key, defined by the number of bits used to represent these primes, plays a crucial role in determining the encryption's resistance to attacks.

Background Information

What is RSA?

RSA (Rivest-Shamir-Adleman) is an asymmetric encryption algorithm that is widely used in for secure data transmission. It is named after its creators, Ron Rivest, Adi Shamir, and Leonard Adleman, who first proposed it in 1977. RSA is a public-key cryptosystem, meaning it uses two different keys: a public key for encryption and a private key for decryption.
The security of RSA is based on the mathematical difficulty of factoring large composite numbers. The algorithm involves generating a pair of mathematically related keys: a public key and a private key. The public key is made available to anyone who wants to send encrypted data to the owner of the corresponding private key. The private key is kept secret and is used to decrypt the received messages.

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. 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. $\varphi(n) = (p-1)*(q - 1)$.

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 $(d * e) \bmod \varphi(n) = 1$. 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). 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, φ(n) = (p - 1) * (q - 1). φ(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 φ(n). In other words, (d * e) mod φ(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).

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 obtaining both private and public keys, the process of encryption and decryption for a given text becomes available through simple mathematical operations. To begin, let's represent the message we want to encrypt as the cipher text 'A,' which we'll denote as '1' for eligibility. Next, we designate our private key as 'd' and our public key as 'e.' These keys play a crucial role in the encryption and decryption process.

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

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

The Modular Inverse ensures that the RSA encryption and decryption operations are inverses of each other within the modular arithmetic domain. This means that, given the public and private keys, it is computationally infeasible to derive the private exponent "d" from the public exponent "e." The Modular Inverse is crucial for the security of RSA, making it a robust encryption and decryption algorithm for secure communication.

Modular inverse algorithms are used to compute the multiplicative inverse of a number modulo a given modulus. In modular arithmetic, the modular inverse of a number "a" modulo "m" is another number "b" such that (a * b) mod m = 1. In other words, it is the number that, when multiplied by "a" and then reduced modulo "m," gives a result of 1.

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.

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.

According to the https://brilliant.org, Euler's totient function (also called the Phi function) counts the number of positive integers less than n that are coprime to n.
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:
$\varphi(n) = \text{Count of } \{k \in [1, n] : \gcd(n, k) = 1\}$
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 φ(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.

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.

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 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:
Managing RSA keys, especially in complex systems, can be challenging. RSA uses both public and private keys, and protecting the private key is of utmost importance. If an attacker gains access to the private key, they can decrypt sensitive data, impersonate the entity with the private key, and potentially cause severe 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. 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 is ongoing research into post-quantum cryptography, which aims to develop encryption methods that are resistant to attacks from quantum computers. Organizations and security experts are closely monitoring advancements in quantum computing and preparing for a future where RSA's security may no longer be sufficient.

Overcoming RSA's Key Size Limitations

To overcome this limitation, one can increase the key size. Common key sizes in use today are 2048 bits and 3072 bits, with 4096 bits or more being recommended for long-term security. Increasing the key size exponentially increases the difficulty of factoring N, thus making it more resistant to attacks.

Mitigate performance issues by using hybrid encryption. RSA can be used to establish a secure channel for exchanging a symmetric encryption key. The majority of data can then be encrypted and decrypted using a faster symmetric encryption algorithm like AES, minimizing the impact on performance.

Implement strong key management practices. Use hardware security modules (HSMs) for secure storage and management of private keys. Regularly rotate keys, promptly revoke compromised keys, and enforce robust access controls to mitigate the risk associated with key management.

Keep an eye on developments in post-quantum cryptography. Researchers are working on new cryptographic algorithms that are resistant to quantum attacks. Organizations should be prepared to transition to these quantum-resistant algorithms when quantum computers become a threat to RSA's security.

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.

2) The second experiment aims to provide valuable information on how the key length affects overall security. I made a prepared challenge similar to the Haruul Zangi 2022

Capture The Flag Competition ‘'hard-rsa'. It requires us to decrypt the ciphertext using given long-bits numbers.

Investigation

- 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().

- To clarify, the values n, e1, e2, c1, and c2 represent the product of two prime numbers, encryption key 1, encryption key 2, ciphertext 1, and ciphertext 2, respectively.

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

n	4096
e1	2
e2	17
c1	4092
c2	4095

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 methodology to determine the multiplicative inverse of e modulo φ(n). Therefore, d ≡ e^(-1) mod φ(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”)

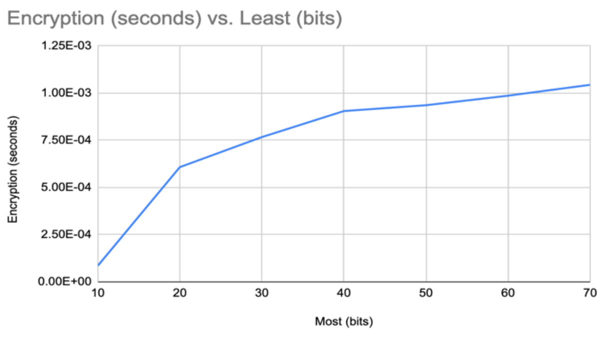
Least significant bits

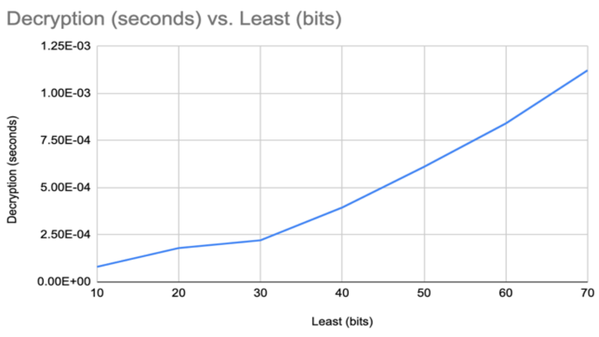
Key Size (bits)	Encryption (seconds)	Decryption (seconds)
10	8.487701416015625e-05	8.0108642578125e-05
20	0.0006079673767089844	0.0001800060272216797
30	0.0007669925689697266	0.00022101402282714844
40	0.0009050369262695312	0.00039505958557128906

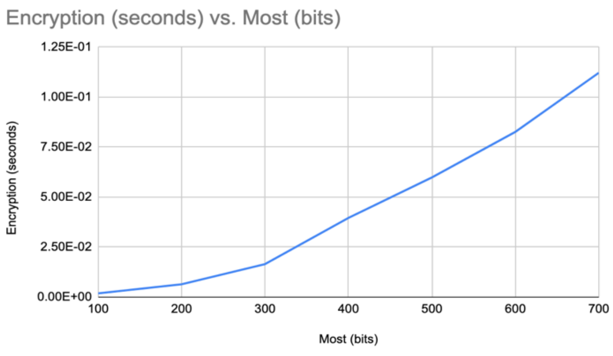
50	0.0009357929229736328	0.0006108283996582031
60	0.0009860992431640625	0.0008409023284912109
70	0.0010437965393066406	0.001123189926147461

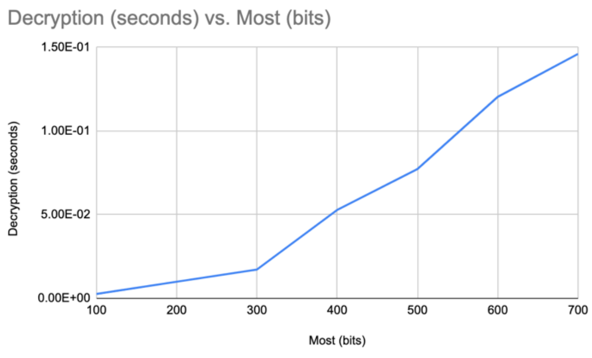
Most significant bits

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









2) As a result of finding the private key using the Modular Inverse function, we obtain an output with the same number of bits. Next, by utilizing the online decoder from dcode.fr, we can retrieve the FLAG plaintext as a character string.

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, it is considered to be good!!! However, this experiment is primarily done on small bits. If the number of bits increases to 1024 and 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.

This conclusion is of paramount importance, as it 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.

Conclusion

Works Cited

“What is RSA algorithm?”. Assessed 1 July. 2023

Euler's totient for RSA https://nitaj.users.lmno.cnrs.fr/RSAnitaj1.pdf and
https://www.youtube.com/watch?v=qa_hksAzpSg
https://brilliant.org/wiki/eulers-totient-function/#:~:text=Euler's%20totient%20function%20(also%20called,that%20are%20coprime%20to%20n.
https://www.youtube.com/watch?v=4zahvcJ9glg
https://www.youtube.com/watch?v=oOcTVTpUsPQ
https://github.com/haruulzangi/2022/tree/main/round-1/Crypto/hard-rsa
https://www.dcode.fr/rsa-cipher

Code

1)

```
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)

n:831867308139871055902097154226989000705322706504595156289404802433561155266201571315856883031456147017960497389910300424822370376918762819281441523551181482114011764232826568302055333719493682299333389530935638960119488346868684457210406340782414585370695775100583532883018249965727447158315120493670645384142592361833802552133691408887620248966340800173060665075576023946742406314282128028107745093627720281391616567918067772594312224566483949592823344093052961777239569080771616165328499635612317362086639719079816223153882084670738298154545289379965119652953821175543216595097067399822255310678417504467885377735104906554161915725703542362031432933712296364802259730913310450754198404779294259861577498124209199998772441993437423274807793219292574326956356556071527033622485063901453780086597577790774158080213911528035112855783951913424477734236238683632121034297601170393744299806126026396747712421574814573319804900427427958971537893289824604841303598410815408959819649650829245727511128130925656690592953782235430728934631503173233260040546769169791177888655421678135167617712282941909419566125276737670308706475944413623219712726272797791583825269157207978002627625272558989237107719772577030383862734918964634199654256167781

e1: 3

e2: 65537

c1:

3274710188721972829503846739252856943954182173523247564671613279161988317972739014789489532270566933973081404580674520171404626240364892316989840926319086608200410309699154536464178409249798498139932059611745062564675353706714095000504669530455403273190556706951136132087959144014279957375613480886051132392655221290737506301641309901469654851378961990993314504179764609565896504107476589775175644480970456242446798331038948822189264950465434628666840384402544143543908005619716733317811365701631235687129842553650417592581388785127462562878811596918065312389893603001675700927264053993525630166409353441454489330869535215907670939013819506740235300027973067604638941848909440

0781942827518086053049803649188061484772487099699057841701685268558964781105433133556884503594065244715028805211936673913444729389561360378898384542921757177836187099832019835723762159480282991067955067368007545556480792839765554234552214918850329098810872122223427614447723656972198942461751554977920675501690316471154874573911895764397603431536233263770506450915011088880903567385881521771843075785760781493960042075157186841042400448656535420012204917699072051608586592981615082556385553192094825046445067024547764196564008238453997775124767619663659 7644

c2:

4508324789327467576856900140742391046390080482442643177973579070610473445198457419243902522676556701086805487298667377741181345149937322458739716787168107271572940274682576794232201948240634032455911610639441511445869335412395929851713482159007247258425362917272751761591048123429655009422434464971769752603083651593653253490673481807652588953506714730568666258174817971835604987608971555863340493695820251367483978839182241763056541406830558451879358198218170485044392304742363361693203731520809283968918034814780704438334909201327438125977640917172146322910754563176840701806888553121651001194660004573834668783448529097790638183142974111189873727443804603330199066080807754563092016467003788456536752723481634814074806374209272104882710538709378627645003153263612151531175458712249614835432822324021952797193687288408765455390407445818163609821239955476575931456180830344973628805826815836417185654882961636800839016734804328521018912599572171771345247180252802207740676950048896399569473494315692996721512841434702792645042960203316007472894124979212602334993534791766344720795679945991343562592014834609022037710941470895080424711674117889122869463762126551240302853969819537479505553512370344379595380877031088859543409724 74583

Find the number of bits

```
import math

# Define the large number as a string
large_number_str = '#Cipher text'

# 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

```
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 = '#integer'
n = '#integer'
inverse = mod_inverse(e1, n)
print(f"The modular inverse of {e1} modulo {n} is {inverse}")
```