A Deep Understanding of RSA Algorithm

By Subhajit Das and Daniel Hoogasian

CS 566 - Term Project

Abstract

RSA algorithm is a commonly used asymmetric algorithm, perhaps the most widely used asymmetric algorithm. The algorithm was publicly described in 1977 by Ron Rivest, Adi Shamir, and Leonard Adleman at MIT. The letters RSA are the initials of their surnames. The algorithm is based on some interesting relationships with prime numbers. The security of RSA derives from the fact that it is difficult to factor a large integer into its prime factors.

This paper will describe the math behind the steps in RSA and present an easily understandable example. It will then describe how RSA utilizes the unique characteristics of prime numbers for encryption and decryption. Lastly, it will present problems with RSA and potential solutions.

Mathematical Foundations

Math and Mechanics

The unique properties of prime numbers provide the basis for the strength of RSA. By utilizing related theorems and number theory concepts, the three researchers were able to build off some already publicized ideas and create a usable model. Some of these theories have been around for hundreds of years, but are quite important here.

The generation of prime numbers, say p and q, is the first main step in the algorithm. In practice, these numbers are very large, often around 155 digits. For the sake of simplicity, we will choose small numbers for our example, say 7 and 19. These numbers are then multiplied to create the modulus, say n. For our example, $n = 7*19 = 133$.

Then the totient of this number is calculated. This is done using Euler's totient function (AKA Euler's phi (ϕ) function) and is the total number of integers less than n and greater than 0 that are relatively prime to n. Two numbers are relatively prime if they share no common denominates other than 1. This can be written $\gcd(n, \phi(n)) \equiv 1$.[4] Here, the $\equiv$ symbol denotes congruence. For prime numbers, $\phi(prime) \equiv prime - 1$ (called Fermat's little theorem), so $\phi(n) \equiv (p-1)*(q-1)$. In our example, $\phi(n) = 6*18 = 108$.

The next step is selecting the keys. First, a prime number for the public key, e, is chosen, such that $1 < e < \phi(n)$ and e is relatively prime to $\phi(n)$. The public key is now (n, e). We will choose 29 for e, so our key is (133, 29). Then a prime number for the private key, d, is selected, again such that $1 < d < \phi(n)$ and d is relatively prime to $\phi(n)$. Additionally, $e*d$ needs to satisfy the condition $e*d \% \phi(n) \equiv 1$. The private key is now (n, d). We will select 41 for d, so our private key is (133, 41).

Next is to compute the ciphertext. The first step here is to convert the text message to a digitally transmittable form. This is done by converting it to one of the traditional character encoding representations, like ASCII, Unicode, or UTF-8. "RSA" will be our text example and we will use ASCII encoding, which comes out to be (82,83,65).

The ciphertext formula is, $c = m^e \% n$, where m is the message character and c is the ciphertext. This needs to be calculated for each character. The first character, 'R', is $82^{29} \% 133 = 17$. The second letter, 'S', is $83^{29} \% 133 = 125$. The third letter, 'A', is $65^{29} \% 133 = 88$. So, our complete ciphertext is (17,125,88).

To decrypt the ciphertext, we use the same formula, but instead of e, we use the private key, d, as the exponent. The new formula is $m = c^d \% n$. So, for the first ciphertext character encoding, $17^{41} \% 133 = 82$, which maps to 'R'. Second, $125^{41} \% 133 = 83$, which maps to 'S'. Third, $88^{41} \% 133 = 65$, which maps to 'A'. And we get "RSA" as our decrypted message.

Main Mathematical Principles

RSA takes advantage of a handful of mathematical concepts. Here, we will discuss the two primary principles. Both principles rely heavily on the unique qualities of prime numbers.

The first principle deals with prime generation and integer factorization, and is as follows: prime generation is easy (i.e., choosing p and q), multiplication is easy (i.e., $p*q$), but finding the prime factors (i.e., of n) is very hard. People have been studying this problem for hundreds of years, but even with modern advancements, we have yet to see any major advancements. Even to this day, finding the prime factors of a 1024-bit number would take one year on a $10M supercomputer.[1]

The first step, which involves testing whether a number is prime, used to be a hard task, but faster methods that check for certain properties were developed in the 1970s. These methods enable this step to be quick. Fortunately, this is still true today, even when the typical size of one of these prime numbers is 155 decimals.

Next, we will discuss the second principle, which deals with modular exponentiation and root extraction, and is as follows: modular exponentiation is easy (i.e., solving for c in, $c = m^e \% n$, if we know n, m, and e), modular root extraction – the reverse of modular exponentiation – is easy if we know the prime factors (i.e., solving for m in, $c = m^e \% n$, if we know n, e, and c, and the prime factors p and q), but if we don't know the prime factors, modular root extraction is hard (i.e., recovering m, only knowing n, e, and c).[1]

Implementation of RSA algorithm

Our code is set up to encrypt and decrypt a sample text with our sample p and q, followed by two randomly selected prime numbers from a small list. When the prime numbers are small, the message often isn't encrypted well, or at all. When the size is sufficient, we see the message is adequately encrypted and correctly decrypted. We attempted to implement our program with the traditional 1024-bit prime numbers, but our machine was not powerful enough.

```python
def rsa_generate_key(p: int, q: int) -> \
        tuple[tuple[int, int, int], tuple[int, int]]:

    # Compute n
    n = p * q

    # Compute phi(n)
    phi_n = (p - 1) * (q - 1)

    # Randomly choose number for e until condition (i.e., it is relatively prime to phi(n))
    # is satisfied
    e = random.randint(2, phi_n - 1)
    while math.gcd(e, phi_n) != 1:
        e = random.randint(2, phi_n - 1)

    # Calculate d
    d = modular_inverse(e, phi_n)
    print("Private key: ",(n, e), "\nPublic key: ", (p, q, d))
    print('n: ', n)
    print('phi: ', phi_n)
    print('(e*d) % phi: ', (e*d)%phi_n)

    return ((p, q, d), (n, e))

# Encryption function
def rsa_encrypt_text(public_key: tuple[int, int], plaintext: str) -> str:
    n, e = public_key

    encrypted = ''
    for letter in plaintext:
        encrypted = encrypted + chr((ord(letter) ** e) % n)

    return encrypted

# Decryption function
def rsa_decrypt_text(private_key: tuple[int, int, int], ciphertext: str) -> str:
    p, q, d = private_key
    n = p * q

    decrypted = ''
    for letter in ciphertext:
        decrypted = decrypted + chr((ord(letter) ** d) % n)

    return decrypted
```

Issues with RSA

The RSA algorithm has been widely used for asymmetric cryptography since its publication in 1977, but there are concerns about its continued suitability. RSA digital certificates have been using larger key sizes to compensate for smaller keys being compromised, and some experts argue that traditional algorithms such as RSA have exceeded their useful lifespan and replacements such as Elliptic Curve algorithms should be considered. Recent studies have discovered potential flaws in RSA, particularly in implementations with smaller modulus values, which can render them susceptible to cryptanalysis attacks. Researchers have used different mathematical methodologies, such as linear equations and

lattice matrix attacks, to achieve the same results in compromising RSA implementations with smaller moduli.

Conclusions

RSA has been an effective asymmetric encryption algorithm for 30 years. However, there is a growing body of evidence that RSA is no longer the best choice for modern asymmetric applications. Other cryptography algorithms like Elliptic Curve algorithms are best suited as an alternative to RSA algorithms. However, there are indications that Elliptic Curve cryptography may have issues of its own. In the immediate future, the answer is to use larger RSA keys with more carefully chosen modulus operators.

Works Cited

- [1] Kaliski, B. (n.d.). The mathematics of the RSA public-key cryptosystem. *American Statistical Association*. Retrieved April 12, 2023, from https://ww2.amstat.org/mam/06/Kaliski.pdf
- [2] Agrawal, M. (2006). Primality tests based on Fermat's little theorem. In *Distributed Computing and Networking* (pp. 288–293). Springer Berlin Heidelberg.
- [3] (n.d.) *Euler's theorem.* PrimePages. https://t5k.org/glossary/page.php?sort=EulersTheorem#:~:text=Euler%27s%20Theorem%20states%20that,%E2%89%A1%201%20(mod%2012). Accessed 18 Apr. 2023.
- [4] Dong, C. (n.d.) *Math in Network Security: A Crash Course.* https://www.doc.ic.ac.uk/~mrh/330tutor/ch05s02.html