Paradigm Academic Press
Innovation in Science and Technology
ISSN 2788-7030
NOV. 2024 VOL.3, NO.6

Unveiling RSA Encryption: A Study of Cryptographic Security and Digital Signatures

Eric Huang[1]

[1] Junipero Serra High School, San Mateo, CA 94403, USA

Correspondence: Eric Huang, Junipero Serra High School, San Mateo, CA 94403, USA.

doi:10.56397/IST.2024.11.09

Abstract

This project explores digital cryptography through the cryptography security system of RSA encryption. It also explores digital signature encryption with a simple hashing mechanism. This project resulted in a program that encoded and decoded messages using RSA encryption, handled large numbers, and calculated encryptions in a viable amount of time.

Keywords: digital cryptography, RSA encryption, digital signature, fast power algorithm

1. Introduction

1.1 Introduction to RSA Encryption

First, I will define two terms for those inexperienced with modulo functions: Modulo of a number X by number Y means taking the remainder of X divided by Y. The modular inverse of a number X by number Y means the number Z that results in the following being true: X*Y mod Y = 1.

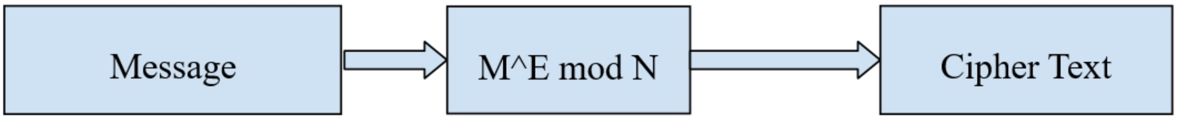
Many organizations use RSA for strong online security, an example of which is online website logins or online banking systems. RSA encryption utilizes an asymmetric public and private key system, with the public key being defined as N and E, and the private key being defined as D. N is derived as the product of two large primes, and the totient of N($\lambda(n)$) is derived as (p-1)(q-1), given the rule that the totient of any prime number is the number minus one. E is then chosen as a number greater than one and less than the totient of N while being coprime to the totient of N. D is then determined as the modular inverse of E with N as the base, which can be done using the extended Euclidean algorithm.

After this is done, messages can be decoded and encoded by anyone who knows the private and public keys. Given a message M (be consistent in capitalization), one can encode a message by using this equation:

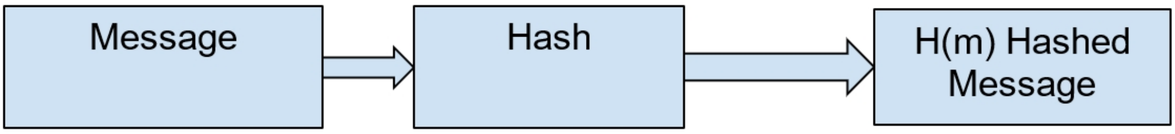
m^e (mod N) = C (ciphertext)

And decode an encrypted message by using this equation:

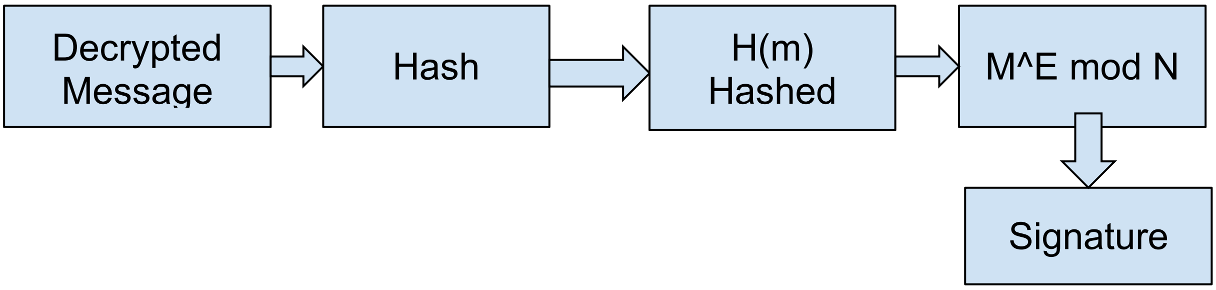
c^d (mod N) = M (message)



The RSA digital signature scheme in order to confirm messages that have not been altered follows a similar format: First, hash the message M, then apply the RSA encryption to the hashed message. Then, the hash is implemented because a single base signature is not secure enough, and thus we need to hash it for further security.



The receiver can then decrypt the message using the private key and hash their own decrypted message; if both the decrypted signature and the hashed decoded message are equal, then the message has not been altered. If the message has been changed in any way, then the signature will not match up.



2. Challenges

2.1 Storage Challenges

During the development of this project, I ran into a couple of challenges. The first challenge was the storage of large primes. RSA encryption is powerful because cracking the product of two large primes is unreasonable, so RSA encryption programs need to be able to handle these large numbers. In C++, the language that I coded the project in, the largest number is a long long which can only hold about 19 digits.

2.2 Time Challenges

The second challenge was calculating the operations within a reasonable amount of time, specifically the power function, and calculating m^e (a large prime), and c^d (also a large prime). The base power function does the operation in O(n) time, which would often give the wrong answer and take much too long of a time.

3. System Design

3.1 Fast Power Algorithm

In order to solve this problem, I utilized the fast power algorithm. In simple terms, the algorithm uses the fact that every power can be broken down into binary powers of two, which can be calculated in O(log n) time because all that is needed is squaring the number until it reaches the largest power of 2 necessary to calculate it. Below is the pseudo-code for this function:

```
Input (Power, Base, N)
While Power is More than 0:
If Power is Not Divisible By 2:
Multiply the stored answer by the base
Apply the Modulo of N onto the stored answer
Subtract the current power by 1
Square the base
Modulo it by N
Divide power by 2
```

3.2 Other Functions

Decryption Function simply uses the fast power algorithm and the formula c^d (mod n) = m to decrypt messages.

Encryption Function is a bit more complicated:

Run Fast Power to encrypt the message using m^e (mod n)

Hash message m by multiplying it with a large prime

Run Fast Power on the hashed message to get the digital signature

Finally, the digital signature validation function follows as such:

Hash decrypted message m with the same large prime

Run Fast Power on the digital signature with the formula s^d (mod n)

Compare the hash with the decrypted signature

3.3 GMP Library

In order to solve the storage limitations mentioned before, I read through the GMP C++ library and implemented the methods and new variable type (mpz_t). This type can hold even up to 200 million bits of data, given the right amount of memory allocated. Because of the special type of new variable, many of the basic functions, such as modulo or arithmetic, needed to be replaced by the respective functions, and returning an mpz_t in a function was not possible, requiring an extra variable in order to save changed values.

4. Evaluation

The current code ran with the test primes in 0.000102318 seconds, using 3840 kb of memory. In the future, there are a few functions that could be implemented that could improve this coding.

4.1 RSA Key Generation

First: Key generation and validation could be added later. Key generation in RSA is specific, as they need to fulfill specific requirements. First, large prime generation and randomization must be coded to generate secure private and public keys for senders. Then, keys E and D must be chosen specifically using the concept of modular inverses. There are quite a few algorithms for figuring out the modular inverse to a number, like the extended Euclidean algorithm.

4.1.1 Extended Euclidean Algorithm

An example of the extended Euclidean algorithm proceeds as follows:

Finding the modular inverse of 240 mod 17:

$1 = 240x + 17y$

$240x \equiv 1 \pmod{17}$

Applying the Euclidean algorithm to 240 and 17 yields

$240 = (17*14) + 2$

$17 = (2*8) + 1$

This can be rewritten as

$2 = 240-(17*14)$

$1 = 17-(2*8)$

Substituting the value of 2 we have above into the second equation we get:

$1 = 17-((240-(17*4)) * 8)$

Breaking the right side down into factors of 17 and 240, we can get:

$1 = (17*113) - 8*240$

Then, the modular inverse can either be -8 or 9, which is equivalent to -8 mod 17.

We can quickly check by multiplying 9 by 240, to see that it is equal to a multiple of 17+1, the requisites for being a modular inverse.

4.2 Key Security

The next thing that could be added is a muti key system, with parts of the message being decrypted and encrypted with parts of the key. In this way, an adversary cannot decrypt and break messages even if they get part of the key from a message or an insider.

4.3 Storage and Memory Consumption

Finally is storage and memory use of the program: currently, the most time-consuming function of the program is the fast power algorithm: running in O(log n) time. Other than that, the gmp addition and multiplication basic functions run in O(n) and O(n log(n) log(log(n))) time respectively. The memory usage is small, with gmp working with pointers to store larger numbers and adaptively allocating as much memory as needed.

5. Discussion

Currently, the program is unable to run outside of its compiler, and it cannot receive input outside of the terminal/stdin. It also cannot send alphabetic characters/encrypt them, requiring messages to be numeric only. This means that communication would require some form of number-to-letter translation.

The program also uses a very simple hash by multiplying with a large prime: this could be improved by using a library hash function that would be more secure and more easily shared between sender and receiver.

6. Conclusion

The main functions resulting from the project are the fast power algorithm, encryption, decryption, and digital signature validation. The project utilized the gmp library in order to manage large primes, and the fast power algorithm in order to make sure the program would run in a viable amount of time. The digital signature validation utilized a hash function for security. RSA encryption provides a secure way for us to digitally transfer information or currency, things that people do every day without giving much thought. It provides us with the safe assurance that our data will not be attacked or stolen and that we can carry on with our daily operations without worries.

References

Galla, Lavanya K., Koganti, Venkata SreeKrishna and Nuthalapti, Nagarjuna, (2016). “Implementation of RSA”. 2016 International Conference on Control, Instrumentation, Communication and Computational Technologies (ICCICCT).

Goyal, Vipul, (2024). “Index of /~Goyal/Handout2h”, www.cs.cmu.edu/~goyal/handout2h/. Accessed August 31, 2024.

Copyrights