RSA Encryption Final Project

Ariel Young, Nashir Janmohamed

June 14th, 2020

Summary

1 Introduction

RSA seemed cool, so we decided to implement it.

2 Theory

2.1 RSA Algorithm

Given two very large (in our case, 512-bit, 150 digit) prime numbers p and q, it can be proven that the product $n = pq$ cannot be factored in any reasonable

amount of time using modern computing hardware. This is the basis for the RSA cryptographic system. *TODO: This paragraph needs work, expansion*

In addition to n, the two other components required by RSA are the encryption exponent, e, and the decryption exponent, d. We should choose e such that e is coprime to the product $(p-1)(q-1)$. An accepted practice is to choose a constant prime value for e, rather than computing a random coprime value. In our implementation, we have set e to be the constant $2^{16}+1 = 65,537$. This choice of e is generally regarded as more secure than smaller primes, and also allows for more efficient decryption [1]. The public key is formed by the pair (n, e).

After having chosen p, q, and e, we can compute our decryption exponent d, such that $d * e \equiv 1 \ (\mathbf{mod}\,(p-1)(q-1))$. In our implementation, we are computing the solution for d by using the Extended Euclidian Algorithm to solve for the coefficients of Bézout's identity, which is both more efficient and far more scalable than other solutions. This increased efficiency was necessary because of the size of the primes our implementation uses. The private key is the formed by the pair (n, d).

From here, the rest of the algorithm is relatively simple. After packing the characters from a message into integers (we used 32 bit integers, each of which holds 4 ASCII characters), the cipher c_i for each packed block m_i can be computed using the formula:

$$c_i = m_i^e \,\mathbf{mod}\, n$$

Although our implementation does not have this feature (we store ciphers as arrays of unsigned 8-bit integers in JSON files for transport), the ciphers can be reassembled into one string. Many implementations also perform some form of symmetric encryption on this string before outputting it.

In order to decrypt, we can find the packed value m_i from each cipher block c_i using the formula:

$$m_i = c_i^d \,\mathbf{mod}\, n$$

At this point, the original text of the message can be unpacked and fully recovered.

2.1.1 Random number generation

To implement random number generation, we used the Linear Congruential Method. This is a recursive sequence of pseudo-random numbers, defined by the relation:

$$x_n = (ax_{n-1}) \ \mathbf{mod} \ m$$

In this relation, a is a multiplicative constant, c is an additive constant, and m is the modulus (also a constant). For these values, we have chosen to mirror those used by the glibc implementation of the linear congruential method. Our seed x_0 is the current Unix time, in microseconds. In order to normalize the output of our random number generator, we divide by the modulus before returning the result of each iteration.

An important note is that our random number generator is most definitely not secure. While it passes the Chi squared test, we still have a direct linear dependence on the initial seed, which most secure implementations (particularly glibc) usually avoid with an additional recursive term [2]. Our generator would most likely fail the Spectral test [3].

2.1.2 Generating large primes

Initially, we had intended to use prime number sieves and random selection to generate our primes p and q. Various approaches were investigated, including using the *Sieve of Eratosthenes* [4], and then the *Sieve of Atkin* [5] when we realized that Eratosthenes was not sufficiently efficient. However, this approach has two main flaws. The first of these is that prime number sieves generate *all* prime numbers under a certain boundary, and random selection from these sets could yield two values of drastically different sizes. Optimally, p and q should differ by (at most) a few orders of magnitude. The second (and more notable) flaw was that we only needed two primes, not all primes below some limit. Moreover, these primes must be extremely large, of a size where sieves become entirely impractical.

Thus, we eventually settled on implementing the Miller-Rabin primality test.

2.1.3 Modular Multiplicative Inverse

To compute the exponent used in the private key, we implemented the Extended Euclidean Algorithm for computing the *modular multiplicative inverse* [7] ...

3 Implementation

3.1 one subsection for each portion of program

Maybe put pseudocode?

4 Analysis

4.1 Chi Squared Test

4.2 Bitmap

4.3 Runtime Analysis

Talk about big O to make him excited :o *bruh this takes so long to run it might as well be big P*

5 Usage

5.1 CLI

Show that using a new key will produce garbage

6 References

References

[1] Twenty Years of Attacks on the RSA Cryptosystem. *Dan Boneh*
`https://crypto.stanford.edu/~dabo/pubs/papers/RSA-survey.pdf`

[2] The GLIBC random number generator. *Peter Selinger*
`https://www.mscs.dal.ca/~selinger/random/`

[3] Spectral test. *Wikipedia contributors*
`https://en.wikipedia.org/wiki/Spectral_test`

[4] Eratosthenes, sieve of. *Encyclopedia of Mathematics*
`http://encyclopediaofmath.org/index.php?title=Eratosthenes,_sieve_of&oldid=34160`

[5] Add source

[6] Add source

[7] Add source