1 RSA Encryption Scheme

Done by Dylan Ross

In our project, we chose to implement the RSA public encryption scheme. This scheme has been around since the 1970s when it was invented by Ron Rivest, Adi Shamir, and Leonard Adleman. The general idea of the scheme is to use the factorization of a large composite integer as a trapdoor function in order to be able to reverse encryption that was done via modular exponentiation.

1.1 Background Math

This encryption scheme relies heavily on modular arithmetic, notably exponentiation. Below are necessary sections to understand the math behind the RSA algorithm.

1.1.1 GCD, Prime Numbers, and Coprimality

A prime number is an integer greater than 1 that has exactly 2 factors: 1 and itself. Examples of such numbers include 2, 3, 5, 7, and 11. The exact distribution of prime numbers is not known, and this fact plays a large role in various parts of cryptography. However, the distribution of prime numbers can be approximated by the prime number theorem, which states that $\pi(N) \sim \frac{N}{\log(N)}$ where $\pi(N)$ is the number of prime numbers p such that $p \leq N$. Every integer can be written as a product of primes, but finding such primes is a difficult problem. This problem is what creates the security of RSA.

We will define $gcd(a, b)$ as the greatest common divisor of integers a and b. That is, $gcd(a, b)$ is the largest integer g such that $g|a$ and $g|b$. For example, $gcd(4, 6) = 2$ as the factors of 4 are $\{1, 2, 4\}$, the factors of 6 are $\{1, 2, 3, 6\}$ and 2 is the largest integer in the intersection of these factors. We say that two numbers are coprime if their gcd is 1. For example, 4 and 9 are coprime as they share no factors besides 1 even though neither number is prime. By the definition of prime numbers, a prime number p is coprime with all numbers a when $a < p$. That is, $\forall a < p, p \in$ prime numbers : $gcd(a, p) = 1$. The gcd of two integers a and b can efficiently be computed via the Euclidean algorithm, which states that $gcd(a, b) = gcd(b, a \bmod b)$.

1.1.2 Fermat's Little Theorem and Primality Testing

Fermat's little theorem states that, for any prime p and any integer a such that $a \not\equiv 0 \bmod p$, $a^{p-1} \equiv 1 \bmod p$. While deterministic primality testing is infeasible due to the unknown distribution of primes, Fermat's little theorem provides an efficient way to perform a probabilistic primality test. For any number n, we can test if n is prime by generating a random number a where $1 \leq a < n$ and test if $a^{n-1} \equiv 1 \bmod n$. If this test does not hold, n is definitely composite,

while the test holding implies that n may be prime. Repeating this test several times can greatly reduce the probability of incorrectly labelling a composite as prime.

Using the primality test described above and the approximate distribution of primes discused in section 1.1.1, we can efficiently generate random primes of a desired bit length. This can be done by generating random numbers of the correct length and testing if they are prime. Due to the approximation of πN, we can expect this process to be approximately linear in time complexity.

1.1.3 Euler's Theorem

We will define Euler's totient function $\phi(n)$ as the number of integers $a < n$ such that $gcd(a, n) = 1$. In the case of a prime p, as was mentioned in section 1.1.1, $\phi(p) = p-1$ and as such Euler's theorem is consistent with Fermat's little theorem. Given a composite number $n = pq$ where p and q are primes, we can calculate the totient of n by $\phi(n) = \phi(p) * \phi(q) = (p-1) * (q-1)$. As such, we can calculate the totient of a composite number in constant time if we have its prime factorization. This fact serves as the basis of the trapdoor function of RSA.

Euler's Theorem states that, for a and n such that $gcd(a, n) = 1$, $a^{\phi(n)} \equiv 1 \bmod n$. From here, it follow that $a^{\phi(n)+1} \equiv a \bmod n$. This is because $a^{\phi(n)+1} = a^{\phi(n)} * a \equiv 1 * a = 1 \bmod n$.

1.2 Key Generation

RSA keys have two parts: the public key and the private key. The public key consists of the integers N and e and is used for message encryption while the private key consists of the integers N and d and is used for message decryption. As the names imply, the public key can be seen by anyone while the private key is kept as a secret.

To generate the keys, first pick two large primes p and q that are a similar size. We will let $N = pq$ be the public (and private) modulus. We will also pick a random integer e such that $e < \phi(N)$ and $gcd(e, \phi(N)) = 1$. As was mentioned above, the public key consists of this pair (N, e). We then calculate $d \equiv e^{-1} \bmod \phi(N)$, and this makes up the private key along with N.

1.3 Encryption

In order to send an encrypted message using RSA, the sender will encrypt the message using the intended receiver's public key. As such, anyone can encrypt and send a person a message, but only the intended recipient can decrypt and read the message using the corresponding private key.

Recall that the receiver's public key is made up of two parts, N and e. First, the sender encrypts their message as an integer m where $m < N$. They can then calculate the ciphertext $c = m^e \bmod N$. This ciphertext can then be sent to the recipient to be decrypted and read.

1.4 Decryption

In order to decrypt a message that was encrypted using RSA, the receiver must use their own private key. Recall that the private key consists of two parts, N and d. Upon receiving ciphertext c, the message can be decrypted to the original message m by computing $m = c^d \bmod N$. This works due to the construction of e and d. Because $d \equiv e^{-1} \bmod \phi(N)$, we know that $de = k\phi(n) + 1$ for some integer k. Due to the fact that $c \equiv m^e \bmod N$, $c^d \equiv (m^e)^d = m^{ed} = m^{k\phi(N)+1}$. It follows from the results of section 1.1.3 that this is equivalent to the original message m.

1.5 Security Measures in Implementation

RSA has many pitfalls that can result in a loss of security if implemented improperly. For instance, there are an entire class of attacks that stem from the public exponent e being too small. Selecting $e = 3$ is a common example of this, and the decreased size of e makes the encryption process faster. However, it allows attacks such as Hastad's Broadcast attack, where the same message being sent to more than e people can be decrypted without knowing d by using the Chinese Remainder Theorem. Even more simply, it is very important that $m^e \geq N$. If e and m are small enough that this is not true, then the message can be decrypted by calculating $m = \sqrt[e]{c}$. Another potential pitfall of RSA is if the private exponent d is too small. In cases where $d < \frac{1}{3}N^{\frac{1}{4}}$, Wiener's Attack can be used to recover d and break the encryption using Wiener's theorem and continued fractions.

Even in the case that the parameters are chosen appropriately, some vulnerabilities may remain due to the malleable nature of RSA. Given ciphertext $c \equiv m^e$, an attacker can choose a desired constant r and compute a new ciphertext c' that decrypts to $m' = rm$. This is done by calculating $c' = r^e c \bmod N$. This works because, during the decryption process, the receiver will compute $m' \equiv (c')^d = (r^e c)^d = (r^e m^e)^d = r^{ed} m^{ed} \bmod N \equiv rm$.

During our implementation of RSA, we took all of these potential vulnerabilities into account. In order to prevent small public exponent attacks, we set a fixed value for e of 65537. This value of e is large enough to prevent small public exponent attacks, yet it a prime number that can be written as $2^k + 1$. The fact that e is prime is important since e must be coprime with $\phi(N)$ and a prime e makes this much more likely. The fact that e is one more than a power of 2 means that the exponentiation of $m^e \bmod N$ can be computed quickly as well. In order to facilitate using a chosen value for e, we had to add some additional

logic when generating p and q. We only accepted values such that $p-1$ and $q-1$ were coprime with our desired value of e. Because $\phi(N) = (p-1) * (q-1)$, if both factors were coprime to e then ϕ must also be coprime. In order to prevent small private exponent attacks, we added a check during key generation to see if the resulting d was too small (see Wiener's attack bound above). In cases where d is too small, we restart the key generation. In order to prevent our ciphertexts from being malleable, each one also contains a hash of the message. As such, each ciphertext is plaintext aware, and multiplying the ciphertext by a constant will prevent the hash from matching and alert the receiver that the message has been tampered with. In order to ensure that we did not make any errors in our RSA implementation, we made sure to reference the paper "Twenty Years of Attacks on the RSA Cryptosystem" by Dan Boneh to check if anything mentioned in the paper applied to our system.

2 ElGamal

Done by Dylan Ross

In addition to RSA, our project also supports the ElGamal encryption scheme for initializing the connection between the atm and the bank. This scheme has been around since the 1980s, when it was invented by Taher Elgamal. The scheme takes advantage of the difficulty of the discrete logarithm problem in order to create a secure encryption algorithm with a trapdoor.

2.1 Background Math

This encryption scheme relies on the discrete logarithm problem. This problem revolves around the fact that, given a modulus n, a generator for $\mathbb{Z}_n^*$ called g, and a number b such that $b \equiv g^a \mod n$ for some a, it is difficult to solve for a.

2.2 Key Generation

ElGamal keys consist of both a public and a private key. The public key contains enough information to encrypt a message such that only the receiver can decrypt it due to the discrete logarithm problem, while the private key contains the necessary information to avoid needing to solve a hard problem. The contents of the public key are the integers p, α, and β while the private key consists of the integer a. While the only private information is a, it is easier in practice to store the private key as both p and a as p will also be necessary during decryption.

To create the keys, select a large prime p and an integer α such that α is a generator for $\mathbb{Z}_p^*$. Next, select a random integer a such that $1 \leq a \leq p-2$. Finally, calculate $\beta = \alpha^a \mod p$. The public key consists of the tuple (p, α, β),

while the private key consists is a.

Selecting a generator is not a simple task. We did so by selecting the smallest integer g in $\mathbb{Z}_p$ such that the minimum a that satisfies the equation $g^a \equiv 1 \bmod p$ is $a = p - 1$ and $1 < g < p - 1$. By Fermat's little theorem (see section 1.1.2), we know that $a = p - 1$ will be a solution for all $g < p$. In order to test for the minimality of the solution, we can take the factorization of $p - 1$, say $f_{1..n}$, and see if $g^b \bmod p$ is 1 for all $b = \frac{p-1}{f_i}$. We know that g is a generator if and only if none of the values of b are result in the equation being true.

2.3 Encryption

ElGamal encryption works by converting the desired message into a number and doing computations on it using the intended recipient's public key. As such, anyone can encrypt and send a message to a desired recipient, but only the intended receiver of the message has the required information to decrypt and read the message.

Recall that an ElGamal public key consists of the numbers p, α, and β. We will call the message to be encrypted m, and will assume it has already been converted to a number such that $m < p$. First, the sender must generate a random integer k such that $0 \leq k < p - 1$. Then, the sender can calculate the numbers $y_1 \equiv \alpha^k \bmod p$ and $y_2 \equiv m\beta^k \bmod p$. The ciphertext is the tuple (y_1, y_2), and can now be sent to the receiver.

This ciphertext is secure because, in order to recover m, one must calculate $(\beta^k)^{-1} \bmod p$. While an attacker would know the values of β and p, the only way that they could solve for k would be to solve the equation $\alpha^k \equiv y_1 \bmod p$, which is difficult by the discrete logarithm problem.

2.4 Decryption

ElGamal decryption works by using the private key and the ciphertext to calculate $(\beta^k)^{-1}$ without needing to solve the discrete logarithm problem.

Recall that the ciphertext consists of the two integers y_1 and y_2, and the private key consists of a (and the number p from the public key is required as well). In order to recover the message m from the ciphertext, the receiver can compute $m \equiv y_2(y_1^a)^{-1} \bmod p$.

To understand why this works, recall the construction of the public key and the ciphertext. The above formula for recovering the message is the equivalent to $m\beta^k * ((\alpha^k)^a)^{-1} \bmod p$. Because $\beta \equiv \alpha^a \bmod p$, this is the same as $m(\alpha^a)^k * ((\alpha^k)^a)^{-1} = m\alpha^{ak} * (\alpha^{ak})^{-1} \equiv m \bmod p$.

2.5 Security Concerns

While ElGamal does not have as many well-documented potential pitfalls to fall into during implementation as RSA does, it still has some points to be careful of. Notably, the security of the scheme relies on the discrete logarithm problem, but this problem can be solved efficiently if $p - 1$ only has small factors. In order to prevent this from happening, we generated p of a desired bitlength n by generating a random prime q of bitlength $n - 1$ until $p = 2q + 1$ was prime. However, this led to a large time for key generation due to the linear time complexity of generating prime numbers, compounded over trying to generate a prime that can generate a second prime. This leads to the second pitfall of attempting to save time by reducing the size of p. While this is very effective for generating faster keys, it reduces the security of the encryption. As such, we did not use any keys where p had less than 1024 bits.