Cryptography and Network Security Final Project

Ryan Prashad Leith Reardon Dylan Ross

August 10, 2020

Contents

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 Message Signing

In addition to encrypting messages to be sent to others, RSA can also be used to generate a signature for a message. A signature is a message that is encrypted in some way using a user's private key that can be verified using their public key in order to prove the user's identity.

Because RSA encrypts and decrypts via exponentiation, the scheme is commutative. That is because, for all m, e, and d, $(m^e)^d = (m^d)^e$. As such, a

message can be signed by encrypting it using your own private key. Alternatively, a signature for a message can be generated by doing the aforementioned private key encryption on the hash of the message. This type of signature can be verified by computing the hash of the message and checking that it equals $s^e \bmod p$ where s is the signature.

1.6 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 Encryption Scheme

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 $g^a \equiv 1 \mod p$ 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 $g^b \mod p$ 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 Message Signatures

Like RSA, ElGamal can also be used to generate signatures for a message. Given the private key a, prime modulus p, and generator α, a signature for message m can be constructed as follows. First, we will define $H(m)$ to be the hash of m using some cryptographic hash function. First, the signer generates a random integer k such that $1 < k < p - 1$ and k is coprime to $p - 1$. They can then use k to compute $r \equiv g^k \bmod p$ and $s = (H(m) - xr)k^{-1} \bmod p$. The signature consists of the tuple (r, s).

In order to verify this signature for message m, a verifier can use the signer's public key $= (p, \alpha, \beta)$ to test if $g^{H(m)} \equiv y^r r^s \bmod p$. If this condition holds, the the signature is valid.

2.6 Security Measures in Implementation

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.

3 AES Encryption Scheme

Done by Dylan Ross

We implemented AES according to FIPS 197. AES is a private key cryptosystem that was designed to replace the previously standard DES encryption algorithm. AES works by splitting the message into blocks of 16 bytes and encrypting each block seperately.

3.1 Mathematical Definitions

AES treats each byte as a polynomial with its bits as the coefficients. This polynomial representation results in the byte 0x55, with binary representation 0101 0101 being treated as the polynomial $x^6 + x^4 + x^2 + x^0$.

3.1.1 Addition

Within AES, adding bytes is done by adding their representative polynomials mod 2. On a bit level, this is the equivalent of the bitwise XOR operation, which will be represented by the symbol $\oplus$. Additionally, this means that addition and subtraction are the same, as XOR is its own inverse.

3.1.2 Multiplication

Within AES, multiplication is done in GF(2^8) and reduced mod an irreducible polynomial m, which is defined as $m(x) = x^8 + x^4 + x^3 + x + 1$. This multipli-

cation is represented by the $\bullet$ symbol.

This multiplication can be drastically simplified using bitwise operations. Firstly, the byte 0x01 is the identity byte, so $x \bullet 0x01 = x$ for all bytes x. In order to multiply a byte b by 0x02, first you must take note of the most significant bit of b. To calculate the result, you then shift b left by 1 bit, then XORing it with 0x1b if the most significant bit was a 1 (you only do the shift if the bit was a 0). As such, the result when multiplying is $b << 1$ if the most significant bit is 0, else $(b << 1) \oplus$ 0x1b. Applying this formula multiple times allows multiplying by any power of 2, which can be combined with the commutative property of multiplication to multiply any 2 arbitrary bytes together.

3.2 Algorithm Parameters

The FIPS 197 publication specifies three versions of AES: AES-128, AES-192, and AES-256. These versions only differ in a few parameters while the algorithm itself stays the same between them. In our project, we chose to implement AES-256. As such, our implementation uses keys that are 8 words long, blocks that are 4 words long, and has 14 rounds of encryption.

3.3 The Algorithm

The AES algorithm first splits the 16 byte (4 word) block into a 4x4 matrix that is referred to as the state array. The encryption is done by performing a variety of transformations on this matrix, which are defined below.

3.3.1 SubBytes Transformation

The SubBytes transformation uses a substitution box to change every byte in the state array. The substitution box is a pre-defined 16x16 matrix of bytes. Each byte in the state array is split into its upper 4 bits and its lower 4 bits, and is replaced by the byte in the row defined by the upper 4 bits and the column defined by the lower 4 bits. Unlike in DES, only one S-box is used and each byte is used by it individually.

3.3.2 ShiftRows Transformation

The ShiftRows transformation performs a circular left shift on each of the rows, which each row being shifted a variable amount. The rows are indexed from r_0 to r_3 with r_0 being the topmost row and r_3 being the bottommost row. Each row is shifted by it's index, so r_0 remains stationary while r_2 is shifted by 2 places.

3.3.3 MixColumns Transformation

The MixColumns transformation works on individual columns at a time, treating them as a four-term polynomial and multiplying them by a fixed polynomial

$a(x) = 03x^3 + 01x^2 + 01x + 02$, with the multiplication happening mod $x^4 + 1$. If we label the bytes in the column as b_0, b_1, b_2, b_3 where b_0 is in the topmost row and b_3 is in the bottommost row, we can represent the multiplication by the following equations, using the mathematical definitions from section 3.1:

$$b_0 = (0x02 \bullet b_0) \oplus (0x03 \bullet b_1) \oplus b_2 \oplus b_3$$
$$b_1 = b_0 \oplus (0x02 \bullet b_1) \oplus (0x03 \bullet b_2) \oplus b_3$$
$$b_2 = b_0 \oplus b_1 \oplus (0x02 \bullet b_2) \oplus (0x03 \bullet b_3)$$
$$b_3 = (0x03 \bullet b_0) \oplus b_1 \oplus b_2 \oplus (0x02 \bullet b_3)$$

3.3.4 AddRoundKey Transformation

The AES key is expanded into various round keys, as is defined in section 3.3.5. Each of these round keys is 16 bytes, which is made of 4 different 4 4-byte chunks. Each of these chunks is labelled w_0 to w_3, with w_0 being the leftmost and w_3 being the rightmost. In this transformation, each of these chunks is XORed with its corresponding column in the state array to create the transformed state.

3.3.5 Key Expansion

AES-256 takes as input a 256 bit key. This key is then expanded into a number of words equal to $4(N_r + 1)$ where N_r is the number of rounds. For AES-256, $N_r = 14$, so the key is expanded from 256 bits to 60 words, or 1920 bits. This is done through a combination of utilizing the S-box defined in section 3.3.1, circularly left-shifting bytes similarly to section 3.3.2, and utilizing a round constant *rcon* defined by $rcon[i] = \{x^{i-1}, 0x00, 0x00, 0x00\}$. The full algorithm is shown in FIPS 197.

3.4 Encryption

To begin the encryption process, the message is split into 16-byte blocks to be encrypted seperately. We chose to implement a CBC version of AES, so the plaintext of each block is XORed with the ciphertext of the previous block before encrypting.

When encrypting a block, it is first transformed into its corresponding state array and an initial AddRoundKey is performed. From there, 14 rounds take place, where each round consists of a SubBytes, a ShiftRows, a MixColumns, and an AddRounKey in that order. The only exception is the final round, where the MixColumns transorm is skipped. At this point, the resultant state array can be unpacked and returned as the ciphertext.

3.5 Decryption

Each of the transformations used in encryption can be easily inverted. For SubBytes, an inverse S-box exists that, when used the same way, results in the

exact inverse of the original. For ShiftRows, each row is circularly shifted to the right by the same amount it was during encryption. AddRoundKey consists of only XOR operations, and as such is its own inverse. The MixColumns transformation can be inverted in the same method that it was performed, but using a different set of computations. The round keys can be generated in the same way as they are for encryption and then simply used in reverse order.

When decrypting a block, it is first transformed into its corresponding state array and an initial AddRoundKey is performed. From there, each of the 14 rounds consist of inverting ShiftRows, inverting SubBytes, an AddRoundKey, and inverting MixColumns. The only exception once again is the last round, where inverting MixColumns is skipped.

3.6 Implementation

Unlike the public key systems previously mentioned, AES does not really have pitfalls to avoid during implementation because all variables are pre-defined. Additionally, FIPS 197 provides a step by step example of key scheduling in Appendix A and a full encryption example in Appendix C. Both of these were extremely helpful for ensuring that my implementation worked properly.

4 SHA-1

Done by Dylan Ross

SHA-1 is a cryptographic hashing function that was first published in 1995. It works by padding messages to specific lengths and doing various computations on pieces of the message at a time to create a fixed-length unique output for each input, regardless of the input size. We implemented the SHA-1 algorithm according to FIPS 180.

4.1 SHA-1 Functions

Each version of SHA has its own function definitions. In all cases, the function takes 3 words as inputs, and returns a different result depending on which iteration of the function is being called which is referred to as t. Using the symbol $\oplus$ for the XOR operation, $\wedge$ for the bitwise and operation, and $\neg$ for the bitwise negation operation, $f_t(x, y, z)$ where each of x, y, z is a 32-bit word, SHA-1 defines its functions as follows:

$$f_t(x,y,z) = \begin{cases} (x \wedge y) \oplus (\neg x \wedge z) & 0 \leq t \leq 19 \\ x \oplus y \oplus z & (20 \leq t \leq 39) \vee (60 \leq t \leq 79) \\ (x \wedge y) \oplus (x \wedge z) \oplus (y \wedge z) & 40 \leq t \leq 59 \end{cases}$$

4.2 SHA-1 Constants

In addition to defining their own functions, each version of SHA also defines its own constants. SHA-1 defines eighty words as constants that it treats as an array called K_t. Similarly to the SHA-1 functions, the values of K_t are one of four possible values, where each value appears twenty consecutive times.

4.3 Padding

In order to work on inputs of any length (to a point), SHA-1 pads messages until their bitlength is a multiple of 512. This is important as the SHA-1 algorithm works by processing the message in blocks of 512 bits at a time, so the entire message must have a whole number of blocks. This is done by first appending a 1 bit to the message, followed by the minimum number of 0 bits such that the new bitlength $b \equiv 448 \bmod 512$. To finish the padding, 64 bits are appended to the end, where the 64 bits are the binary representation of the original message size.

4.4 Calculating the Hash

To compute a message's hash, first pad the message using the padding algorithm described in section 4.3. Additionally, 5 initial hash values must be defined, which are given in FIPS 180. At this point, the message is split into 512-bit blocks and each block is worked on seperately.

For each block, a message schedule is created by either copying a word from the original message or manipulating previous parts of the message schedule, depending on which part of the schedule is being calculated. From there, the current hash values are modified through a combination of addition, the functions defined in section 4.1, and circular left shifts. Within the SHA-1 algorithm, all addition is done mod 2^{32} to maintain each part fitting within a word.

After the final block has been hashed, the message's resultant hash is computed by concatenating the 5 resultant hash values that have been modified by every block.

4.5 Implementation

When it came to implementing the SHA-1 algorithm, the process was fairly straightforward. The entire process was a matter of implementing three small functions, having logic to determine which one to execute, and initializing constants. The padding algorithm was also easy to implement as it involved simple modular arithmetic to calculate padding size. However, an example hash calculation provided by NIST was very helpful in ensuring that my algorithm was working as intended at every step.

5 HMAC

Done by Dylan Ross

HMAC is a standard for using a cryptographic hash function (such as SHA-1 as defined in section 4) in order to create a keyed message authentication code. The standard treats the chosen hash function as an oracle, and as such is very simple and easy to implement. We implemented the HMAC algorithm according to RFC 2104.

5.1 MAC Generation

We will define $H(m)$ as a cryptographic hash function to be used by HMAC, and B to be the block length of $H(m)$ in bytes.

Firstly, the HMAC key must be B bytes long, so some key manipulation is likely needed. If the key K is longer than B bytes long, then replace K with $H(K)$. Next, if K is less than B bytes long, then K is padded by appending null bytes (0x00) until K has exactly B bytes.

We will define *ipad* as the byte 0x36 repeated B times, and *opad* as the byte 0x5c repeated B times. Using the symbol $\oplus$ to represent the XOR operation and $+$ as string concatenation, the HMAC of message m is computed as $HMAC = H(K \oplus opad + H(K \oplus ipad + m))$.

5.2 Implementation

The implementation of HMAC is very simple, given an already working hash function. As such, we used the SHA-1 algorithm discussed in section 4 as our hash function for generating HMACs. The HMAC algorithm can be made more efficient by utilizing a map data structure. Because *opad* and *ipad* are XORed with the key and used for nothing else, each key can have the values $K \oplus ipad$ and $K \oplus opad$ calculated the first time the key is used and retrieved by future uses of the same key. This way, a relatively large calculation can be skipped many times during long communications that involve many packets being sent, each with its own MAC.

6 Networking Protocols

Made by all three members
Writeup written by Dylan Ross

For our project, we chose to implement a combination of the SSL and the SSH protocols. The networking of our project can be segmented into three main sections: the handshake, ATM verification, and user sign in and banking.

6.1 Handshake Protocol

The handshake protocol is responsible for sharing secret keys between the ATM the bank for use in symmetric encryption and MAC generation. For our handshake, we chose to implement the TLS protocol. The bank acts as a server in our project, and as such must be running when the ATM is run in order to establish a connection.

The ATM begins the handshake by sending a "hello" packet to the bank. This packet contains all of the public key cryptosystems that the ATM can support for connecting to the bank. It is assumed that the ATM already knows the bank's public keys for these systems. This list of supported systems is in the order of the ATM's preference. Upon receiving this packet, the bank uses the received list to select the system that will be used according to its own preferences. Once a system is selected, the bank sends its selection to the ATM as a response to the "hello" packet.

Once the ATM receives the bank's selected cryptosystem, the Diffie-Hellman (DH) protocol can begin. The ATM and the bank both have a shared modulus p and generator g, which is defined by the 1536-bit MODP group from RFC 3526. The ATM will generate a random number a and the bank will generate a random integer b, both of which are in the range $[1, \frac{p-1}{2}]$. The ATM will begin the DH protocol by sending to the bank the value of $g^a \mod p$. Due to the discrete logarithm problem discussed in section 2.1, an eavesdropper over the network cannot recover a from g^a. Upon receiving g^a, the bank will generate a message that contains g^a and g^b and send both this message and its signature back to the ATM. This signature is using the public key system that was previously decided upon and serves to both verify that the bank received g^a correctly and that the value for g^b is actually from the bank. At this point, $(g^a)^b \equiv (g^b)^a \mod p$ is a shared 1536-bit number between the bank and the ATM. In order to generate a key from this large number, we do $K \equiv g^{ab} \mod 2^{256}$ in order to generate a 256-bit key.

Because our protocol needs two seperate keys for encryption and message authentication, the DH protocol shown above is done twice simultaneously in order to generate two distinct keys.

6.2 ATM Verification

Once the DH protocol is finished, both systems are ready to use AES symmetric encryption. However, before banking can start, the bank must authenticate that the client it just exchanged a key with is actually an ATM. If this step was skipped, a malicious user could connect to the bank and repeatedly send packets telling the bank to deposit money into their own account. In order for this authentication, we assume that every ATM has a unique identifier and that the bank has access locally to the public keys of every ATM.

The ATM uses AES to send its identifier to the bank in order to start the authentication process. Upon receiving this message, the bank will look up the ATM's public key using the identifier and will generate a random number. The public key that is selected is the one corresponding to the same public key system used in the handshake. The bank encrypts the random number using the ATM's public key, and uses AES to send it as a challenge. When the ATM receives this challenge, it uses its private key to retrieve the random number and responds with the hash of the random number concatenated with the session's encryption key. This hash is sent to the bank using AES and the bank verifies it in order to verify the ATM.

6.3 Sign in and Banking

At this point, the ATM has been verified to the bank and they share private keys for both message encryption and authentication. As such, all future messages are encrypted using AES and authenticated using HMAC with these shared keys.

Before the user can begin managing their account, they must first sign in. In order to do so, the ATM will prompt the user for their username and password. The ATM then sends the username and the hash of the password to the bank, which will verify the login credentials and respond to the ATM accordingly.

Once the user has signed in, they can deposit money, withdraw money, and check their balance. For each of these actions, the ATM generates a corresponding packet that it sends to the bank, which the bank then verifies, performs the corresponding action, and sends resulting data back to the ATM to be shown to the user.