

CSCI 4230 Final Project White Hat Report

Glenn Smith, Max Thomas, Billy Zhang

Max Thomas

I was responsible for implementing Paillier, RSA, the SSL handshake, and for modifying the server side of our application to use our custom SSL instead of the unencrypted TCP it had been using previously. The particular client application we used was chosen based on what we had available and already written from other assignments that could be easily converted to use for this project (it was converted from a Network Programming homework).

RSA and Paillier were both relatively easy to implement. The key size chosen for both was 1024 bits, for a couple of reasons. Primarily, the key was chosen to be big enough to permit encryption of the session key as a single block, without having to chop it up. Multi-block RSA isn't terribly hard to implement since the ciphertext block size is the same as the plaintext block size. Paillier, however, has a linear expansion factor on the order of the key size, which makes block-by-block decryption somewhat trickier. Since we only use the PKC algorithms to send one message, it was easier to just increase the key size. Secondly, a 1024-bit key is really hard to brute force, and since the RSA and Paillier implementations were written to be key size agnostic, the key size could be arbitrarily large.

Implementing RSA was fairly trivial, and other than a slight bug in key generation it went without a hitch. Paillier, on the other hand, was somewhat more complicated. The original implementation was supposed to be multi-block, supporting arbitrary-length

messages, but this caused several problems related to ciphertext block sizes and message partitioning and had to be abandoned early in development. There were also problems with key generation, mostly related to Python being strange and poorly suited to low-level arithmetic and bitwise/bytewise operations. In the end, the first successful run of Paillier encryption/decryption was done with a tiny, hand-computed key while I worked at trying to figure out what was wrong with key generation.

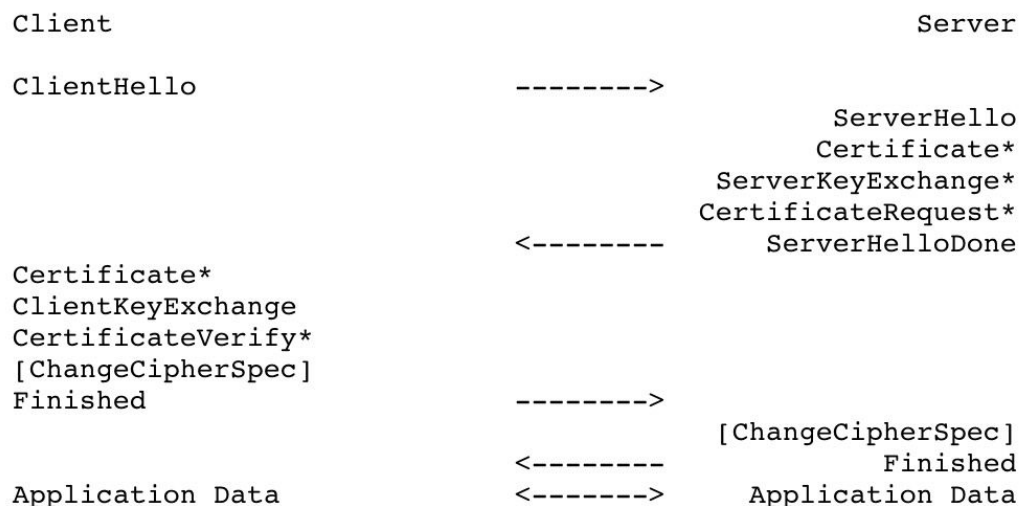
```
(cryptoproj) maxwell@perihelion:~/Dropbox/CryptoProj/tests$ ./ptest ppubkey pprivkey
n: 6758308081397623482157942879889366423848436991082988321144005795271841991431962699602828917967443760077
3551133127871211568045899702572767347354925508777316681696934293016864367353162895732527688531561106371837
989603763883685734948507085698623984253928102436416411161691057213158808545424552494028469372197717
g: 6758308081397623482157942879889366423848436991082988321144005795271841991431962699602828917967443760077
3551133127871211568045899702572767347354925508777316681696934293016864367353162895732527688531561106371837
989603763883685734948507085698623984253928102436416411161691057213158808545424552494028469372197718
lambda(n): 67583080813976234821579428798893664238484369910829883211440057952718419914319626996028289179674
4376007735511331278712115680458997025727673473549255087773002299933726430951719582962023323770707261031312
3864330436698173011383240837872625533624848958284666756033578341029297369882362984054984149788794072922000
0
mu: 373516616294222226380388113102436313683643340195557353299429059952646372690712961435956037596821707914
9564154430088805696699160404497127373593166649566385840496489285348607643539572816156058517088552793052864
8210142114386723193644889179622820114156972565609317615340023792845830868164233298234578124538399948
m = ABCDEFGH
C(m) = {0x54, 0xFE, 0xEF, 0x43, 0x51, 0x79, 0x5B, 0xC4, 0x75, 0x21, 0x1D, 0x87, 0x85, 0x9A, 0x44, 0x90, 0x
BD, 0xD5, 0x9F, 0xB6, 0xC6, 0xE5, 0x26, 0xD2, 0x23, 0x2C, 0xE6, 0xFA, 0x24, 0x87, 0xE9, 0x57, 0x88, 0x7A, 0
xF7, 0x57, 0x78, 0x13, 0x82, 0xB8, 0x79, 0xC5, 0xD7, 0x80, 0xC7, 0x57, 0x5E, 0xD1, 0xE5, 0x32, 0xC4, 0xCB,
0x3F, 0xB0, 0x4F, 0x83, 0xE5, 0x22, 0x0D, 0x4D, 0xED, 0x80, 0x3F, 0x7F, 0x6B, 0xA6, 0x30, 0x9A, 0x99, 0xA
C, 0x70, 0xC9, 0x83, 0x9F, 0xE6, 0x77, 0x62, 0x53, 0xFF, 0x32, 0x50, 0x7E, 0xC1, 0x63, 0xB6, 0x67, 0xDD, 0x
B0, 0x86, 0x95, 0x39, 0x92, 0x89, 0x3D, 0xEF, 0xF4, 0xEC, 0xF3, 0x9F, 0xB1, 0x1D, 0x9C, 0x7A, 0x10, 0x85,
0xEC, 0xA5, 0x55, 0x22, 0x5A, 0x5E, 0xB4, 0x5E, 0x77, 0x29, 0xE6, 0x5D, 0x6A, 0x25, 0x05, 0xF3, 0x3F, 0x55
, 0x5B, 0xE9, 0x1B, 0xDC, 0x50, 0x45, 0xC0, 0x29, 0xDE, 0x24, 0xE2, 0x50, 0x01, 0x10, 0x47, 0xD4, 0x0F, 0x6
8, 0xEA, 0x63, 0xD3, 0x40, 0xDD, 0x48, 0x87, 0x5F, 0x3C, 0x8E, 0xC0, 0xE6, 0xD1, 0x98, 0x5E, 0xFF, 0x4C, 0
x91, 0xE3, 0x9F, 0x82, 0xC0, 0x7E, 0x08, 0xAF, 0x0E, 0xC5, 0x21, 0x81, 0xDC, 0x60, 0x10, 0x02, 0xDA, 0xB3,
0xB7, 0x71, 0x73, 0x0D, 0x4B, 0x22, 0x38, 0x29, 0xA7, 0x56, 0xDB, 0xE6, 0x9A, 0x5D, 0x9F, 0x12, 0x50, 0x3F
, 0xCC, 0x17, 0x19, 0x92, 0x62, 0xD1, 0x98, 0xE4, 0xE2, 0x0A, 0x0D, 0x90, 0x0C, 0x75, 0xD5, 0xCB, 0x04, 0x
AF, 0xE9, 0xCC, 0x2B, 0xE0, 0xEC, 0x9B, 0xC3, 0x09, 0xAB, 0xFE, 0x19, 0xA5, 0x38, 0x7F, 0xF0, 0x7B, 0xFB,
0xEB, 0x65, 0x74, 0x60, 0x09, 0x2A, 0x29, 0x52, 0xCC, 0xE8, 0xAF, 0x5D, 0xA1, 0x3E, 0x65, 0x44, 0xC0, 0xF4,
0x7D, 0xC6, 0x7D, 0x9D, 0x16, 0x88, 0x2A, 0x09}
D(C(m)) = ABCDEFGH
```

Paillier encryption/decryption test run

Server modifications were also simple. We designed our SSL implementation to match as closely as possible the usage of POSIX socket syscalls (accept(), connect(), send(), recv()), with the idea that any program using POSIX sockets can switch to using our custom SSL with a minimum of hassle. Modifying the server therefore only involved

writing a custom `ssl_dprintf()` built around `ssl_send()` to take the place of the `dprintf()` calls used for most network traffic, swapping the calls to `accept()`, `connect()`, and `recv()` with calls to `ssl_accept()`, `ssl_connect()`, and `ssl_read()`, and storing each user's SSL session state along with their socket descriptor.

The real headache was the handshake protocol. Originally, we had intended to write a standards-compliant implementation of SSL 3.0, as specified in RFC 6101 (<https://tools.ietf.org/html/rfc6101>). However, upon closer examination of the spec, it was determined that many of the elements of the spec (authentication, compression, etc) were either out of the scope of this project, or not in line with project objectives. A diagram of the handshake protocol from the RFC is provided for convenience.



The major deviation from the standard was the complete omission of the Certificate, CertificateRequest, and CertificateVerify messages. As mentioned during the design presentation, support of these would require some sort of certificate authority, which is both beyond the scope of this project and not desirable, since it would require establishment of additional infrastructure beyond a client and a server. This means that the handshake protocol as we implemented has no optional messages - every message must be sent by client and server in the correct order for the key exchange to proceed correctly.

In addition to the omission of authentication, changes were made to every remaining handshake message. The ProtocolVersion field was removed from every message, if present. The Random, SessionID, and CompressionMethod fields were removed from ClientHello and ServerHello; our implementation doesn't support compression, making CompressionMethod useless, and the Random and SessionID fields aren't used by our implementation (the latter is used for authentication, which we don't do, and the former appears to be useless). The ServerKeyExchange message was modified to contain either RSA, Blum-Goldwasser, or Paillier keys (depending on the cryptosystem selected), instead of the RSA, D-H, and Fortezza keys it could contain in an SSL 3.0 key exchange. ClientKeyExchange is similarly modified to contain RSA, Blum-Goldwasser, or Paillier ciphertext. Finally, Finished is modified to contain only a SHA-1 hash, as implementing MD5 is outside the scope of this project.

The set of supported ciphertext suites is also modified and highly reduced from the spec. Three key exchange algorithms are supported (RSA, Blum-Goldwasser, Paillier), as in the spec, but not the same three, as the spec calls for RSA, Diffie-Hellman, and Fortezza. Additionally, we only provide support for one symmetric key algorithm (AES), as opposed to the spec's laundry list of practically every even moderately secure symmetric key cryptosystem used by anyone within the last 40 years.

One further change was made in the interest of simplifying the protocol. In SSL 3.0, the client and server exchange a fixed-size master secret, which is then hashed or otherwise munged to produce the required key or keys for whatever cipher suite was selected. However, since we are only supporting one symmetric key cipher, we just have the client send the server the encrypted session key.

Session key generation is done by reading 20 bytes from `/dev/urandom`. `/dev/urandom` is a special file that provides an interface to a cryptographically secure random number generator maintained by the kernel. It generates an entropy pool using environmental noise from system drivers (among other sources). Whenever a user reads from the file, the kernel generates a seed from the entropy pool, which is used to start a pseudorandom number generator. For the purposes of our small client/server, this provides adequate security. However, for a high-traffic server the time between connections is not sufficient for the entropy pool to be re-seeded, and a better method of pseudorandom number generation is required to maintain cryptographic security.

Billy Zhang

So I was mainly responsible for implementing two algorithms: AES and SHA-1. AES is used to encrypt the traffic we send over the network in SSL. We chose AES as a superior alternative to the toy DES used in class. We debated over the usage of the full DES or even potentially 3DES, but AES's superiority in its efficiency and strength of encryption ultimately won out and persuaded us to add an option to our SSL implementation for it. AES is also extremely well documented, so there were a lot of resources online to help me find out what to do and what I was doing wrong. The most helpful resource I found was FIPS 197, which goes into great detail about how the algorithm works. More importantly, there are several step-by-step guides described that were helpful to debugging my work. They helped me out so much in keeping me on track.

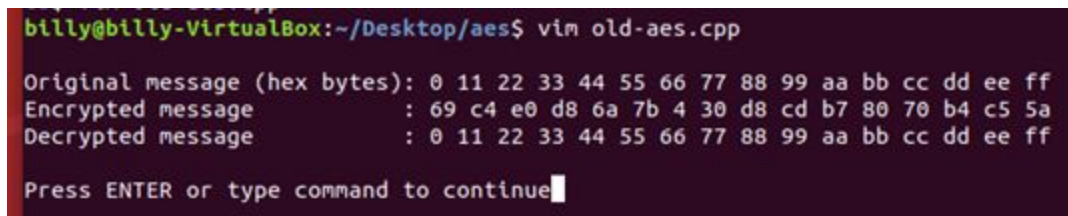
So for AES, I went into the implementation rather clueless about how it worked. I had written a full version of DES before as well as the toy DES, so I went in with some preliminary knowledge. AES has several different options for its key size in 128, 192 and 256. Our SSL implementation uses AES-128 and doesn't support the other options. In that case, AES-128 possesses a key length of 4 words (16 bytes or characters), a block size of 4 words (elaborated later on what exactly a block is), and a total of ten rounds. A round largely consists of four operations: a byte substitute, row shifting, column mixing, and adding a round key into the data. At the start of my implementation, I take the 16-byte key given to me and generate a series of round keys based off it. I

first turn the key into a block, which is just the key written top-down from left to right. Given the new key state, I rotate each column before plugging each value in the key state into the AES s-box. AES has one predetermined s-box, which is generated by interpreting polynomials over $GF(2^8)$. It maps an 8-bit input to an 8-bit output, essentially a byte substitute. It does this by mapping the input to its multiplicative inverse in $GF(2^8)$ before being transformed by a given affine transformation. The key state is then xor'd with RCON, which is something AES uses to help diffusion. Now I repeat the process with the newly generated round key until I have enough for all the rounds (in this case it's 10).

With the round keys in my hand, I move on to the basic steps of a round in AES. I first turn the message into a state similar to what I did with a key, and then substitute each byte via the s-box I mentioned earlier. AES only has one s-box as opposed to DES, with its 8 s-boxes. I then rotate the state (block) by a set amount described by the documentation. I then mix the columns. This is a pretty vague description of what's really going on. AES multiplies a given column of the block by given polynomial $a(x) = 3x^3 + 1x^2 + 1x + 2$. $a(x)$ can also be interpreted as a matrix of the form $\begin{bmatrix} 3 & 1 & 1 & 2 \end{bmatrix}$. This multiplication is also done modulo over $x^4 + 1$. I found a way to simplify the math by reducing everything down to work with multiplication by two in the galois field and left shift. For example, I was capable of galois multiplication by three by multiplying it by two and xor'ing it with the original column value. I then xor in an unused round key with the

current state of the block. This is one round - I repeat this process another ten times, and I end up with AES-128.

I also ended up implementing the decryption for AES. This was pretty simple, as it largely inverses the original transformations of encryption. I used an inverse s-box for byte substitution and I changed up mix columns a bit by editing the matrix the selected column was multiplied by, but it was largely as you would expect elsewhere in the program.

A terminal window with a dark purple background and light green text. The prompt is 'billy@billy-VirtualBox:~/Desktop/aes\$'. The command 'vim old-aes.cpp' has been executed. The output shows three lines of hex data: 'Original message (hex bytes): 0 11 22 33 44 55 66 77 88 99 aa bb cc dd ee ff', 'Encrypted message : 69 c4 e0 d8 6a 7b 4 30 d8 cd b7 80 70 b4 c5 5a', and 'Decrypted message : 0 11 22 33 44 55 66 77 88 99 aa bb cc dd ee ff'. The prompt 'Press ENTER or type command to continue' is at the bottom.

```
billy@billy-VirtualBox:~/Desktop/aes$ vim old-aes.cpp
Original message (hex bytes): 0 11 22 33 44 55 66 77 88 99 aa bb cc dd ee ff
Encrypted message           : 69 c4 e0 d8 6a 7b 4 30 d8 cd b7 80 70 b4 c5 5a
Decrypted message           : 0 11 22 33 44 55 66 77 88 99 aa bb cc dd ee ff
Press ENTER or type command to continue
```

Sample output for encryption/decryption with AES.

I also implemented SHA-1. This wasn't a choice but rather a feature of the project. SHA-1 is a hashing algorithm, and is largely used by SSL to hash the message as the HMAC to ensure that the original packet wasn't tampered with. I used the Wikipedia page as my reference for this algorithm. The FIPS page ended up not being as effective as it was for AES, as it was diluted with information from the other more updated versions of SHA-1 like SHA-2 and SHA-3. SHA-1 is largely outdated at this point, as it's been proven to be insecure against well-funded opponents.

SHA-1 doesn't involve too many arcane mathematics like AES, but it is certainly heavier on the bit-math side of things. This ended up being rather unfortunate for me, as I made several small errors in my calculations that ended up wasting a lot of time. SHA-1 first starts by taking in a message of variable length. This is different from AES, which only accepts a 16-byte message and key. This means that my approach for the two slightly differ in this respect. Mainly in the fact that I need to cover more corner cases inside the SHA-1 implementation. This is because in AES, I resolved the issue of long messages with ECB or CBC. I could rely on a solid past implementation of ECB. But in SHA-1, I needed to ensure that it was just as capable of dealing with millions of characters as it was with three in one cycle of the algorithm.

SHA-1 largely works in blocks of 512 bits, or 64 bytes. I made the decision to work with bytes instead of bits for the default unit in my implementation because I figured that we would be largely encrypting strings to pass over the network. It also makes it a lot easier to code, as I don't need to deal with the hassle of combining bits and tearing them apart over and over again. I pad the given message with null bytes and the length of the message until the padded message reaches a length of 512 bits. After that, I take a chunk of 512 bits and split that into 16 chunks of 32-bit words. I then use these 16 chunks to generate another 64 more chunks via a combination of xor and left-rotate. I then loop through all eighty chunks and process them one by one into an individual part of the hash. There are five parts of the hash being calculated at a time. They are marked as h0 to h4, and initially start as hard-coded values given by the algorithm.

Then they're added on to by modified versions of each chunk and continuously updated with each loop. After eighty rounds, h0 to h4 are finally updated. At this point, the last step is just to combine them into one number to finally output the SHA-1 hash. This is done by just simply placing them side by side via left shift.

```
File Edit View Search Terminal Help
billy@billy-VirtualBox:~/Desktop/aes$ ./a.out
SHA-1 Test cases:
abc: a9993e364706816aba3e25717850c26c9cd0d89d

"" : da39a3ee5e6b4b0d3255bfef95601890afd80709

abcdefghijklmnopqrstuvwxyz: 84983e441c3bd26ebaae4aa1f95129e5e54670f1

abcdefghijklmnopqrstuvwxyz: fe891a7a0af5035834f4389724eeaa7922bb56f1

one million (1,000,000) repetitions of "a": 34aa973cd4c4daa4f61eeb2bdbad27316534016f

Test vectors taken from: https://di-mgt.com.au/sha_testvectors.html
billy@billy-VirtualBox:~/Desktop/aes$
```

Some test cases run through the SHA-1 hash generator.

I learned a lot by implementing these two algorithms, and I encountered a lot of problems along the way. An example was my code often ended up with many errors. They were never major errors, either. It was always small, sinister bugs. Like the misplacement of a variable name, or the usage of a less than equals as opposed to a less than, or even just a mistyped constant. These ended up costing me so much time and made the process on me so much harder than it should have been. This remained constant throughout the entirety of the project, so it gave me an important lesson in just why implementing algorithms yourself is so heavily frowned upon. It was just so easy to make a mistake, and I needed some serious error checking to get my algorithm to a

pushable state. And that's not even mentioning the odd math AES made me do, which isn't exactly time efficient if cryptography is not your main subject for the project.

Nevertheless, I come out of it smarter.

Glenn Smith

I implemented Blum-Goldwasser PKC, HMAC, `ssl_send/ssl_recv` ECB, and the chat client. While you may think that having implemented Blum-Goldwasser as a homework assignment would make this easy, porting the implementation to use GMP's arbitrary precision integers proved a challenge. I ended up rewriting a considerable amount of the code using references from both the class notes and various online sources including documentation for other crypto libraries. I made sure to model the interface to the Blum-Goldwasser functions in the exact same format as the other cryptosystems so that any clients of this library can use any PKC algorithm interchangeably and not have to worry about extra arguments being required to use these functions. I store the keys in byte arrays to make it easy to load them from files; internally they are converted to GMP's `mpz` integers for performing calculations.

One of the main difficulties in integrating Blum-Goldwasser into our SSL system was the format of the ciphertexts. Typically, Blum-Goldwasser ciphertexts are a list of integers that are $\text{length}(\text{message}) / \log_2(\log_2(p \cdot q))$ bits in length, followed by one large integer,

$c[t]$, whose size can be up to $p \cdot q$. To better deal with trying to convert these arbitrary bit length numbers into a stream of bytes, I ended up serializing them in a format that first converts the mpz numbers into a list of bytes of unknown length, then prepending the length of that list, then combining all the lists together and prepending the length of the plaintext message, Nm . While this does store the length of the plaintext in the clear, an attacker using CPA would already know the length of the plaintext, and under CCA it would be relatively easy to determine by just looking at the length of the ciphertext.

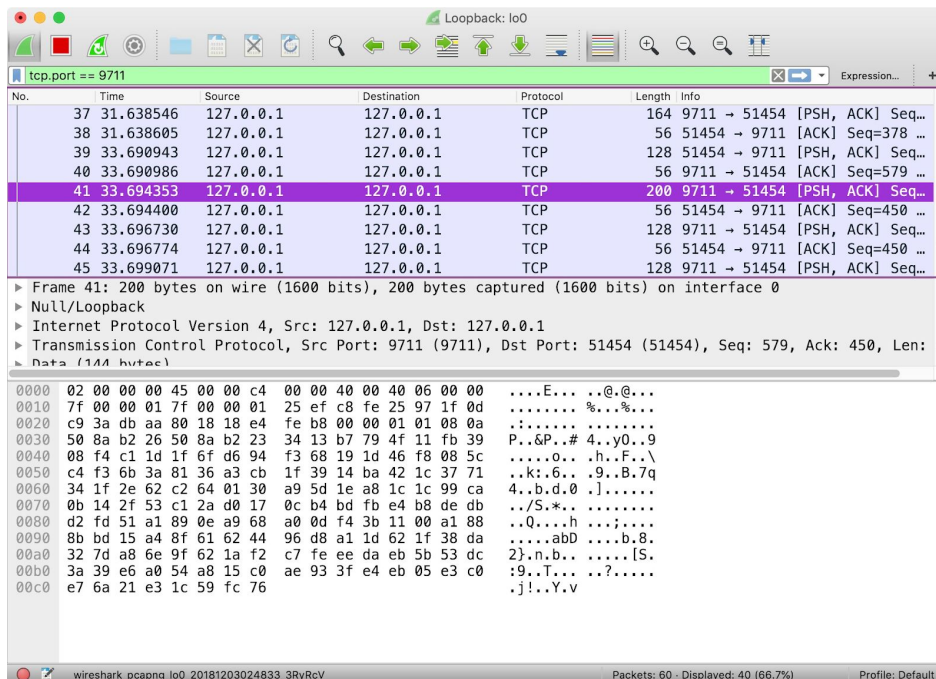
For the key format for Blum-Goldwasser, I opted to store only p and q in the private key. Included in the implementation is the Extended Euclidean Algorithm for calculating the Bézout Coefficients of the two primes to find the constants A and B required for decrypting messages. I chose this format because it reduces the amount of data required to send over the network and makes writing key generation scripts considerably easier in the event that someone else were to use our library. The public key format was simply the constant N , which is all that is necessary to encrypt messages using this cryptosystem. One small note is that the size of public and private keys is equal, as the private key is 2 1024-bit integers p and q , and the public key is 1 2048 bit integer, n . The only other requirement for the keys is that p and q be blum integers, although the library assumes you check this yourself.

signatures for these methods were chosen to closely resemble those of the Linux C System Calls `send` and `recv`, with the notable difference that the SSL methods include a pointer to the current ssl session. These methods send and receive data in blocks of 16 bytes (the block size for 128-bit AES) and use the symmetric key stored in the ssl session to encrypt and decrypt data. The integrity of the plaintext data is preserved by including an HMAC appended to each block of AES-encrypted ciphertext.

To send data with `ssl_send`, we have to iterate through all blocks of plaintext data, encrypt them with the session key, and append them to a buffer that we will send to the given socket. This implementation encrypts each block separately from the first, or ECB. If the last block has extra data, the remaining space is filled in with random values to prevent possible information leakage, as filling the rest with zeroes would allow blocks with only one byte of data to be sent and would make it considerably easier to attack on the ciphertexts. However because of this padding, we also have to send the length of the data so the clients know when to stop receiving. For extra (questionably necessary) security, the length of the plaintext data is encrypted itself using AES, and sent in its own block before any of the other data.

Receiving data with `ssl_recv` functions very similarly to the `recv` System Call. The user specifies the length of the buffer they have allocated to receive data, and the function will fill as many blocks of data as possible in that buffer. First, we receive data from the socket using `recv`, which will block until we get any data. Then, we read blocks from the

received data, decrypt them with the session key, compute the HMAC of the plaintext, and verify that the message was sent correctly. Assuming no blocks are invalid, they are collected into an array of bytes that represents the plaintext data plus padding. Then we need to process all the packets that have been sent, as multiple `ssl_send` packets may have been received between two calls to `ssl_rcv`. So we greedily read packets from the decrypted buffer, stripping off the first four bytes which contain the message length, then discarding the rest of that first block since it contained only the length. We use that length to then read the rest of the data that should be in this packet, stripping off any padding that was added. Then we check for more received messages with a repeat until we have run out of data. Due to the potential security vulnerability of having user specified lengths for the messages, we check that the length of data to receive will not overflow our buffers, just in case any trio of Jacks decide to try and cause buffer overflows in this implementation.



No.	Time	Source	Destination	Protocol	Length	Info
37	31.638546	127.0.0.1	127.0.0.1	TCP	164	9711 → 51454 [PSH, ACK] Seq=...
38	31.638605	127.0.0.1	127.0.0.1	TCP	56	51454 → 9711 [ACK] Seq=378 ...
39	33.690943	127.0.0.1	127.0.0.1	TCP	128	51454 → 9711 [PSH, ACK] Seq=...
40	33.690986	127.0.0.1	127.0.0.1	TCP	56	9711 → 51454 [ACK] Seq=579 ...
41	33.694353	127.0.0.1	127.0.0.1	TCP	200	9711 → 51454 [PSH, ACK] Seq=...
42	33.694400	127.0.0.1	127.0.0.1	TCP	56	51454 → 9711 [ACK] Seq=450 ...
43	33.696730	127.0.0.1	127.0.0.1	TCP	128	9711 → 51454 [PSH, ACK] Seq=...
44	33.696774	127.0.0.1	127.0.0.1	TCP	56	51454 → 9711 [ACK] Seq=450 ...
45	33.699071	127.0.0.1	127.0.0.1	TCP	128	9711 → 51454 [PSH, ACK] Seq=...

Frame 41: 200 bytes on wire (1600 bits), 200 bytes captured (1600 bits) on interface 0

Null/Loopback

Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1

Transmission Control Protocol, Src Port: 9711 (9711), Dst Port: 51454 (51454), Seq: 579, Ack: 450, Len: ...

Data (144 bytes)

```
0000 02 00 00 00 45 00 00 c4 00 00 40 00 40 06 00 00 ....E... ..@...
0010 7f 00 00 01 7f 00 00 01 25 ef c8 fe 25 97 1f 0d .....%...%...
0020 c9 3a db aa 80 18 e4 fe b8 00 00 01 01 08 0a .....
0030 50 8a b2 26 50 8a b2 23 34 13 b7 79 4f 11 fb 39 P..&P..# 4..y0..9
0040 08 f4 c1 1d 1f 6f d6 94 f3 68 19 1d 46 f8 08 5c .....o...h..F..
0050 c4 f3 6b 3a 81 36 a3 cb 1f 39 14 ba 42 1c 37 71 ..k:.6...9..B.7q
0060 34 1f 2e 62 c2 64 01 30 a9 5d 1e a8 1c 1c 99 ca 4..b.d.0 .].....
0070 0b 14 2f 53 c1 2a d0 17 0c b4 bd fb e4 b8 de db ..S.*.....
0080 d2 fd 51 a1 89 0e a9 68 a0 0d f4 3b 11 00 a1 88 ..Q....h ...;...
0090 8b bd 15 a4 8f 61 62 44 96 d8 a1 1d 62 1f 38 da .....abd ....b.8.
00a0 32 7d a8 6e 9f 62 1a f2 c7 fe ee da eb 5b 53 dc 2)..n.b... ..[S.
00b0 3a 39 e6 a0 54 a8 15 c0 ae 93 3f e4 eb 05 e3 c0 :9..T...?.....
00c0 e7 6a 21 e3 1c 59 fc 76 .j!..Y.v
```


Wireshark trace of ssl_send showing the encrypted text sent over the network.

Binwalk reports data entropy at 92.1%

The final part that I wrote was the basic network client for the crypto chat application.

This client was a simple netcat-like program that simply reads in text from the user over stdin and sends it to the chat server using ssl_send, while receiving data from the server decrypted using ssl_rcv, and printing it to stdout. It uses a single-threaded networking paradigm based on the select() Linux System Call that polls any number of sockets and returns when one or more of them receives data. The client allows the user to specify which PKC algorithm they would like to use to perform the key exchange, and otherwise performs equivalent to netcat. Messages are sent to the server when the user presses enter, and the connection can be closed by pressing Ctrl+D to send an End of Text byte.

```
glennsmith@sphere Crypto/CryptoProj (git:master !~*~*) $ cd tests
glennsmith@sphere CryptoProj/tests (git:master !~*~*) $ ../cmake-build-debug/cryp
to_chat_client
Usage: <address> <algorithm type, 0=rsa, 1=blum-goldwasser, 2=paillier>
[1] glennsmith@sphere CryptoProj/tests (git:master !~*~*) $ ../cmake-build-debug/
crypto_chat_client 127.0.0.1 0
USER test
Welcome, test
LIST
There are currently 0 channels
JOIN #test
Joined channel #test
LIST #test
There are currently 1 members
#test members: test
PRIVMSG #test test
#test> test: test
PRIVMSG #test This is a pretty long message to show how the ciphertexts get longe
r when the plaintext gets longer yet it still manages to work correctly
#test> test: This is a pretty long message to show how the ciphertexts get longer
when the plaintext gets longer yet it still manages to work correctly
```

Test run of the chat client, showing usage and basic netcat-like functionality.