

**Cryptography (CTG)**

Year 1 (2020/21), Semester 2

**SCHOOL OF INFOCOMM TECHNOLOGY**

Diploma in Cyber Security & Digital Forensics

**ASSIGNMENT**

|  |  |
| --- | --- |
| **Tutorial Group:** | **P02** |
| **Tutor:** | **Fabian Ng** |
| **Date of Submission:** |  |

| **Student Number** | **Student Name** | **Grade** |
| --- | --- | --- |
| **S10222998** | **Han Xihe** |  |
| **S10222894** | **Dominic Lee Yu Yee** |  |
| **S10221773** | **Neo Jia Chen** |  |
| **S10227852** | **Ho Quang Minh** |  |

**Cryptographic Algorithms**

1. **Introduction**

The digital revolution in recent years led the rise of many people going online, creating enormous amounts of data transfers across the internet everyday. Some confidential data, such as passwords and banking information, must be protected from hackers. This is where cryptographic algorithms are involved.

Many cryptographic algorithms in use aim to fulfil P.A.I.N. as much as possible. When P.A.I.N (privacy, authenticity, integrity, non-repudiation) is ensured, we can be confident that data is secured.

In this report, we will be examining Scrypt, RC5, Elgamal and Salsa20, which are algorithms that we think are worth learning about. The report examines their notable features, inner workings, intended purposes, strengths and weaknesses, and any relevant information, helping have a better understanding of these algorithms.

1. **Scrypt**
   1. **Features / Characteristics**

Scrypt is a memory-intensive key derivation function algorithm that is costly to perform large-scale attacks on. As the memory in Scrypt is accessed in strong dependent order at each step, the memory access is the algorithm’s bottleneck. (Percival, STRONGER KEY DERIVATION VIA SEQUENTIAL MEMORY-HARD FUNCTIONS)

* 1. **Detail Operations**

The overview of Scrypt:

1. It will use the user input to create a HMAC
2. It then runs PBKDF2 function on the hashed value
3. Running a loop, for P iterations, it will apply a memory hard function called SMIX
4. It will then do the PBKDF2 function again on the value that is outputted when the loop ends.
   * 1. **Parameters**

- N = Number of iterations

- r = Block size

- P = Parallelism factor

- Password = Input password

- Salt = Securely generated random bytes

- Derived key length = Number of bytes to generate as ouput

Memory required to compute scrypt key = 128 \* N \* r \* p bytes.

Parallelism factor is responsible for fine-tuning the relative CPU-Cost, the derived key length allows the user to derive the output size, N sets the CPU cost, and r sets the memory cost.

* + 1. **HMAC**

**Graphical user interface, chart

Description automatically generated**  
(CSBreakdown, 2015)

The HMAC object is created with its key and data both being null. The HMAC object is initialized with a key that is hashed from the input message. The data’s value is still null and will only be added during the PBKDF2 function. Note that the notation for HMAC is HMAC(key, data). (Percival, STRONGER KEY DERIVATION VIA SEQUENTIAL MEMORY-HARD FUNCTIONS)

Timeline

Description automatically generated  
(CSBreakdown, 2015)

* + 1. **PBKDF2**

The PBKDF2 passes in the Mac object and the salt and returns a key (K) that will be used by the SMIX in the next step.

PBKDF2 is a pseudo-random function that produces a random message by inputting the message into the HMAC function that has its key as the hash of the user message HMAC(h(message), RandMsg). This will be done for several iterations that is defined by various variables such as the MAC length. Each iteration will update the final value of the return value K. (Percival, STRONGER KEY DERIVATION VIA SEQUENTIAL MEMORY-HARD FUNCTIONS)

Diagram

Description automatically generated  
(CSBreakdown, 2015)

* + 1. **SMIX**

The number of iterations of SMIX is defined by the parallelization factor, P. The SMIX takes in the input K which was the key produced by the PBKDF2 function earlier. It then sets X = K and creates two arrays Y[ ] and Z[ ].

SMIX will first perform Block Salsa which takes in the value X and begin its own loops. In this loop, BlockXOR is performed on X which also means the current value of X xor Ki where i is the current iteration of the loop that it is in.

It then uses salsa 20 to hash the value X and appends this hashed value into array Y where X = Yi where i is the current iteration of the loop. The loop repeats until the iteration limit before going back to the SMIX algorithm.

X is then parsed into a new loop using Integerify which takes the current value of X and converts it into an integer value. This integer value of X is then parsed into another BlockXOR which xors the current value of X with Ki.

It then performs Block Salsa again where this time X is appended to Z each iteration where X = Zi until the loop ends.

A variable K’ is then created by appending array Y to array Z where the evens come before the odds. (Percival, STRONGER KEY DERIVATION VIA SEQUENTIAL MEMORY-HARD FUNCTIONS)

Waterfall chart

Description automatically generated with medium confidence  
(CSBreakdown, 2015)

* + 1. **PBKDF2**

For the final PBKDF2 in Scrypt, K’ which is computed from the SMIX algorithm is used as the salt which will output the final message key. (Percival, STRONGER KEY DERIVATION VIA SEQUENTIAL MEMORY-HARD FUNCTIONS)

* 1. **Purposes**

Scrpyt was created by Colin Percival originally used for the Tarsnap online backup service, published in 2012 by the Internet Engineering Task Force (Percival, n.d.). It was not originally developed for blockchain networks but in the end found great use in cryptocurrencies (Bradbury, 2013).

Scrypt was designed to resist FPGA and ASICs attacks as that was the main flaw of many other Key Derivation Function (KDF) such as PBKDF2. To do this, it had to be both computationally and memory intensive, unlike other password based KDFs which are only computationally expensive. Its aim is to improve upon SHA-256 (Percival, Tarsnap, Josefsson, & Simon Josefsson Datakonsult, 2016).

Currently, scrypt is currently used in many cryptocurrencies as a proof of work algorithm. Currencies ranging from Dogecoin to Litecoin to Einsteinium all uses scrypt as their hashing algorithm and scrypt had been previously used in major projects such as Reddcoin (RDD), Vertcoin (VTC) and Monacoin (MONA) (ADVFN, n.d.).

* 1. **Strengths**
* Unlike PBKDF2 which has a glaring weakness because it uses a small amount of RAM, Scrypt which is an updated version of Bcrypt, takes a lot of CPU time and RAM memory.

This makes it more secure against GPU (parallel password cracking using video cards) and ASIC (specialized password cracking software) attack. It also does what PBDKF2 does well which is protecting against brute-force, rainbow, and dictionary attacks.

* It a simple, efficient, and secure algorithm requiring little resources compared to its counterpart such as SHA-256.
* Scrypt is less complex compared to other mining algorithms and runs four times faster than mining Bitcoin
* Scrypt generally require lower fees for transactions on their blockchains.
  1. **Weaknesses**
* There are better alternatives to Scrypt. Argon2 is considered as a more secure modern key derivation function and SHA-256 might be a better algorithm to use if the context of use is not suitable for Scrypt
* It takes up a lot of CPU time and RAM also makes it disadvantages in certain situations as cost will increase with the high computing power needed.

(Percival, STRONGER KEY DERIVATION VIA SEQUENTIAL MEMORY-HARD FUNCTIONS)

1. **RC5 (Ron Cipher 5)**
   1. **Features / Characteristics**

RC5, short for Ron Cipher 5, is a feister-like word-oriented symmetric key block algorithm that is fast, has a variable key-length, and is a low-memory symmetric block cipher. It has a focus on speed, simplicity, and compatibility across a wide range of computing hardware. It features variable rounds and key sizes that can be increased according to computation power that is available for increased security. (Rivest, 1997)

* 1. **Detail Operations**

RC5 has three stages: key expansion, encryption, and decryption.

* + 1. **Parameters**

RC5 has three stages: key expansion, encryption, and decryption.

- w = Word size: consists of a two-word input and two-word output, where both are 16-bits or higher

- r = Number of rounds.

- B = Length of key K bytes

- Pw = magic constant Odd((e – 2) \* 2 ^ w), with Odd here being the closest odd integer to (e – 2) \* 2 ^ w

- Qw = magic constant Odd((ϕ - 1) \* 2 ^ w), with Odd here being the closest odd integer to (ϕ - 1) \* 2 ^ w

Parameters B and r are directly proportionate to security as the higher/longer they are, the higher the security.

But the higher the security, the lower the speed of the algorithm

The nominal parameters for RC5 are, a 32-bit word, 16 round, and 7-byte key length. (Rivest, 1997)

* + 1. **Key Expansion**

Key expansion is the step before encryption and decryption to obtain the expanded key S for future use in the encryption and decryption algorithms. The secret key will be processed through bitwise XOR, left circular shift and addition to produce t = 2r + 2 subkeys, where each subkey is w bits long.

In the algorithm, the array of secret key K is K [0], K [1], K [2]... K [B - 1] is converted into L [0], L [1], L [2] ... L [C – 1]. To convert the arrays, we first divide w by 8, which we will call u = w/8. Next, we calculate the length of array L by comparing values B and 1, and then use the larger value and divide it by u. We then round the results of the operation to the nearest integer, the obtained result known as C. Array L will therefore be C in length. To calculate array L, we loop with B iterations from B-1 to 0. L[i / u] is then equivalent to key array K bytes K[i], added to the left circular shift rotation of L[i/u] and 8, where L[i/u] = (L[i /u] <<< 8) + K[i]. This provides all values in array L from L[0] to L[C – 1]

In a separate operation, the subkey array S, S[0], S[1], S[2], ... S[t – 1] is initialised. To calculate S[i], take the previous index, i-1 and add the magic constant Qw to S[i-1], where S[i] will now be equal to S[i – 1] + Qw. This operation is iterated t times, from 1 to t – 1, starting from S[0] being equal to magic constant Pw. This iteration provides all values in subkey array S.

In the following operation, the word array L is mixed with subkey array S, where i = j = X = Y = 0, taking the larger of t and C and multiplying the value by 3, the obtained result would be the number of iterations, i of the mixing operation. For each iteration in the mixing operation, X is equal to S[i], which is also equivalent to the left circular rotation of S[i] + X + Y to obtain a value which we will call G. A left circular shift of G and 3 is then performed, where G <<< 3 = X. In the next operation, a similar calculation is performed, where Y is equal to L[i], equivalent to L[j] + X + Y to obtain a value which we will call P. A left circular shift of P and the results of X + Y is performed, where P <<< (X + Y) = Y. Afterwards, the iterative values i and j have a modulus performed on them to “prepare” them for the next iteration of key expansion, where i is equivalent to i + 1 modulus t and j is equivalent to j + 1 modulus c. These operations produces the final expanded key array S, S[0], S[1], S[2] to S[t - 1]. This expanded key array S is then passed onto the encryption/decryption algorithms. (Rivest, 1997)

Diagram

Description automatically generated

(Lectures, 2021)

* + 1. **Encryption**

After the key expansion, the next step for RC5 is the encryption of the data which uses various primitive operations to do so. They are bitwise XOR, left/ right circular rotation, and addition/ subtraction.

The plaintext resides in two w-bit registers as A and B. LE refers to the left half of the data and RE refers to the right half of the data after the round has been completed. There will be r rounds occurring during encryption and in each round, there will be a substitution using both words of data, a permutation using both words of data, and a substitution that depends on the key.

For each round, assuming this is round 1, LE = S [0] + A and RE = S [1] + B. LE is then XORed with RE to get a value which we will call K. A left circular rotation is then performed on K and RE, where K <<< RE, and this value is then added to S [2] which forms LE1, the left half of the data of the first round. LE1 is then XORed with RE to form a result which we will call L. Similarly, a left circular rotation is then carried out on L and LE1, where L <<< LE1 and the result is added to S [3] to form RE1, the right half of the data after round 1. This process is repeated for r rounds.

The final cipher text is contained in the two variables LEr and REr which is of length = 2w.

As both halves of data are updated in 1 round, 1 round of RC5 = 2 rounds of DES. (Rivest, 1997)

Diagram, schematic

Description automatically generated  
(Kamel, 2012)

* + 1. **Decryption**

Decryption in RC5 is similar to its encryption except that some of the operations are reversed. Like encryption, there are two w-bits of ciphertext assigned to one-word variables LDr and RDr and r round in total.

In each round, unlike the encryption where it starts with LE and RE, decryption starts with LDr and RDr. S [2r+1] is subtracted from RDr and right circular rotation is then performed on the LDr and this result to obtain a result which we will name M. M is the XORed with LDr to obtain RDr-1. Following, S [2r] is then subtracted from LDr and the obtained value then combined with RDr-1 goes through a right circular rotation and this value N, is then XORed with RDr-1 to obtain LDr-1.

This will be done for r round to obtain the plain text which will also be 2w bits which will be contained in LD and RD. (Rivest, 1997)

* 1. **Purposes**

RC5 was designed by Ronald Rivet in 1994 as a successor to RC4 with a focus on simplicity. One design point around RC5 was to encourage the study and cryptoanalysis of simplistic, low-level algorithms targeted for use in computing applications. (Rivest, 1997)

* 1. **Strengths**
* RC5 is a fast algorithm as it only uses primitive computer operations which takes little computing power.
* Due to the natures of its operations, RC5 is not memory intensive and can be used in systems with severe memory shortages such as checkout kiosks.
* It is known for its simplicity as a symmetric block cipher. The encryption and decryption routines can be written in a few lines of code.
* RC5 is known to be a very secure algorithm which is extremely difficult to break. A reason it is so secure is that its key expansion is a one-way function as it uses the golden ratio in Qw and logarithmic e in Pw which makes it impossible to reverse the key expansion process. A reason it is so secure is that its key expansion is a one-way function as it uses the golden ratio in Qw and logarithmic e in Pw which makes it impossible to reverse the key expansion process. Also, using powerful distributed computing systems or parallel computing systems will be too time-costly and thus unfeasible if the parameters were high. Thus, there is no obvious way RC5 is weak other than the key length being too short, or number of rounds is set too low.
  1. **Weaknesses**
* A high number of rounds are needed for increased security in the applications of RC5. 18 or more rounds are recommended for sufficient protection against attacks. However, a higher number of rounds would make the algorithm slower in less powerful computing platforms, while using a lower number of rounds will make it more susceptible to attacks, such as differential attacks using 12 rounds only. This means that for increased security, there would be the trade-off of having decreased speed on less powerful computing platforms, and vice versa.
* If the length of the key is too short and there are too little rounds, it would be possible to break it. This introduces implementation limitations where there are computing limitations limiting the key length and number of rounds that can be used. For example, using 64-bit blocks and 12 rounds, RC5 can be broken with a 2^44 chosen plaintexts via a differential attack (Kushilevitz & Biryukov, 1998).

1. **ElGamal**
   1. **Features / Characteristics**

ElGamal is a public key algorithm that was designed as an RSA alternative and a practical implementation of the Diffie-Hellman key exchange. It is probabilistic in nature, with its security being dependent on discrete logarithmic operations in a cyclic group generated for encryption in the algorithm. (Elgamal, 1985)

* 1. **Detail Operations**

ElGamal encryption consists of 3 components: Key generation, encryption, and decryption. Suppose Alice wants to communicate with Bob…

* + 1. **Key Generation**

Bob generates a very large number q and a cyclic group Fq. He then chooses any element g and an element a such that a and q are coprime. GCD(a,q) = 1. He then computes h by taking g to the power of a, h = g^a. He will then publish the cyclic group Fq, h, q and g where q and g are his public key. He also retains a as his private key. (Elgamal, 1985)

* + 1. **Encryption**

Alice then encrypts her message using Bob’s public key (q,g) through the following steps.

Alice selects an element k from the cyclic group Fq that Bob published but must ensure that k and q are coprime where GCD(k,q) = 1. Alice then calculates p using g to the power of k, p = g^k and s by taking h to the power of k.

As h = g^k, s = h^k = g ^ ak

Next, Alice will multiply M (the value of the message) with s and will send (p, M\*s) to Bob.

(Elgamal, 1985)

* + 1. **Decryption**

Bob receives (p, M\*s) from Alice and will then calculate p ^ a which will give him g ^ak which will be the same value as s. He then divides M \* s by s to obtain M the message.

(Elgamal, 1985)

* 1. **Purposes**

Elgamal was created by Tahar Elgamal in 1985 to be an algorithm that has its security reliant on the computational demand of calculating discrete logarithmic problems over finite fields. It has seen use as a component of cryptosystems in computing applications such as the free GNU Privacy Guard software, recent versions of PGP and other cryptosystems.

(Elgamal, 1985)

* 1. **Strengths**
* Its probabilistic nature causes it to produce several different ciphertext for a single plaintext. This is because a random k is chosen each time. As attackers are not able to link broken ciphertexts to other unbroken ciphertexts since they will be different.
* It has a faster decryption speed compared to RSA, the most popular asymmetric encryption algorithm. (Javed, 2012)

(Elgamal, 1985)

* 1. **Weaknesses**
* It has a slower encryption speed compared to RSA (Javed, 2012)
* During encryption, ElGamal requires two exponentiations meaning that the ciphertext is twice as long as the plaintext. This problem will arise with larger, longer message.
* ElGamal is unconditionally malleable and is hence not secure under chosen ciphertext attacks. Meaning that an attacker can transform a ciphertext into another ciphertext which decrypts to f(m) where function f is know wihout knowing m.
* Requires an appropriate padding scheme to achieve security against chosen ciphertext attacks.

(Elgamal, 1985)

1. **Salsa20**
   1. **Features / Characteristics**

Salsa20 consists of a hash function, expansion function and an encryption function, all working under similar principles. We will be covering the encryption function in this document.

Salsa20’s encryption function is a symmetric key stream cipher that is designed by Daniel J. Bernstein around add-rotate-XOR operations within a psuedorandom function. It was designed in response to the U.S. government’s restriction on the publication of fresh traditional ciphers, except for hash functions, where the hash function of Salsa20 is used to encrypt data as a workaround of said restrictions.

(Bernstein)

* 1. **Detail Operations**

Salsa20 consists of 3 components: key stream generation, encryption and decryption.

* + 1. **Key stream generation**

An initial internal state (arranged into a 4 by 4 matrix) is created, where each cell in the matrix will usually hold a 32-bit word. These values will be fed into a quarter-round function depending on whether if a line of values are arranged in columns or rows, and whether if the round number is odd or even. The formula for the quarter-round operation is as follows, given the notation QR(a, b, c, d):

b ^= (a + d) <<< 7

c ^= (b + a) <<< 9

d ^= (c + b) <<< 13

a ^= (d + c) <<< 18

For example, to obtain b, perform a constant-distance left rotation of the sum of a and d by 7. Bitwise XOR assign the result of said operation to obtain b.

The initial internal state is arranged like so:

|  |  |  |  |
| --- | --- | --- | --- |
| Constant | Key | Key | Key |
| Key | Constant | Nonce | Nonce |
| Stream position | Stream position | Constant | Key |
| Key | Key | Key | Constant |

With the indexing of each cell like so:

|  |  |  |  |
| --- | --- | --- | --- |
| 0 | 1 | 2 | 3 |
| 4 | 5 | 6 | 7 |
| 8 | 9 | 10 | 11 |
| 12 | 13 | 14 | 15 |

Diagonally across, the constant consists of an ASCII “nothing-up-my-sleeve" value (values that look innocous to deter attackers) that is broken down into four words. The key consists of a user-generated value that can be 128 or 256 bits long (if the key is 128-bits, each key cell holds 16-bits, but if the key is 256-bits, each key cell holds 32-bits). The 64-bit user-generated nonce (random arbitrary value generated for this encryption session) is split into two 32-bit words for each cell.

Applying the quarter-round function to the initial internal state, for a total of 20 rounds, the following is performed in odd-numbered rounds:

Column 1: QR( 0, 4, 8, 12)

Column 2: QR( 5, 9, 13, 1)

Column 3: QR(10, 14, 2, 6)

Column 4: QR(15, 3, 7, 11)

And the following is performed for even-numbered rounds:

Column 1: QR(0, 1, 2, 3)

Column 2: QR( 5, 6, 7, 4)

Column 3: QR(10, 11, 8, 9)

Column 4: QR(15, 12, 13, 14)

A mixed array operation is then performed 15 times, where the output from the quarter-round operation is mixed with each value in each cell in indexed order to produce the key stream block. (Bernstein, Salsa20 specification)

* + 1. **Encryption**

When Alice wants to send Bob something, an XOR operation is performed on the plaintext and the key stream block generated. This produces the ciphertext that Bob will receive for decryption. (Bernstein, Salsa20 specification)

* + 1. **Decryption**

When Bob receives the ciphertext, an XOR operation is performed on the ciphertext using the key stream. This produces the plaintext. (Bernstein, Salsa20 specification)

* 1. **Purposes**

Salsa20 has seen a wide range of uses. It’s derivative cousin, ChaCha20, has also been implemented in a wide range of uses in software.

Salsa20 has been implemented in multiple chat and commmunication apps. One such app is Viber. Whenever users send each other files and other media, Salsa20 is used to encrypt the file, sent to Viber’s servers with the file’s hash. The encrypted file data is hashed again for the server to verify the data integrity of the file. The recipient then receives the file with it’s associated metadata and the symmetric private salsa20 key for decryption to read the file. Salsa20 is also used in chat groups, where all members of this “Secure Group” shares the same symmetric private key.

(Rakuten Viber, 2016)

* 1. **Advantages**
* Excellent performance across a wide range of CPUs as a stream cipher
* Flexible implementation as it can be used as a hash function, an encryption function or as a key expansion function

(Bernstein, Salsa20 specification)

* Not patented with several public domain implementations for different CPU architectures (ARM, x86, PowerPC, etc.) written by Bernstein, allowing for easy implementation without licensing difficulties

(Bernstein, Snuffle 2005: the Salsa20 encryption function, 2021)

* No effective, practical attack found as of writing for a full 20 round encryption.
  1. **Disadvantages**
* Under circumstances where there are limited computing resources, different, lower parameter implementations of Salsa20 can be broken. For example, in 2005, an attack of Salsa20/5 with an estimated time complexity of 2^165 was found by Paul Crowley (Crowley, 2005).

1. **Cotton Cipher**
   1. **Scenario**

Alice wants to send Bob a large message to be stored for a long time. She can transfer extremely small amount of data to Bob either through physical contact or securely through stenography / encryption.

* 1. **Features**

This cryptosystem does not have digital signature but instead uses the nonce and salt generated to replace the usual ‘digital signature’. The salt and nonce are meant to be given to the recipient through a secure channel

* 1. **Encryption**

Alice generates a random salt and stores this salt value. She then passes her message salt through the scrypt function to generate a hash value. She then takes her secret key, together with the hashed message and puts it through the Salsa 20 algorithm to encrypt it. She generates a nonce during this process and stores this nonce value too. She follows the same steps above to encrypt the plaintext message through the Salsa 20 algorithm. Next, she encrypts the secret key using Bob’s El Gamal public key. Lastly, she sends the encrypted plaintext message, encrypted message and encrpyted secret key over to Bob. She sends Bob the nonce and salt through a secured channel using one of the above methods.

* 1. **Decryption**

Bob receives the nonce and salt as well as the encrypted messages from Alice. He first decrypts the encrypted secret key using his private key to obtain the secret key. Using the nonce and the secret key, he will then proceed to use them to decrypt both the encrypted hash and message. He then runs the plaintext message and the salt through the scrypt algorithm and then comparing the value generated with the hashed message he decrypted. If they are identical, then the message can be trusted.

Diagram

Description automatically generated

# References

ADVFN. (n.d.). *ADVFN*. Retrieved from ADVFN: https://uk.advfn.com/cryptocurrency/algorithms/Scrypt

Bernstein, D. J. (2021). *Snuffle 2005: the Salsa20 encryption function*. Retrieved from crypto snuffle: https://cr.yp.to/snuffle.html

Bernstein, D. J. (n.d.). Salsa20 design. *The University of Illinois at Chicago*. Retrieved January 30, 2021, from https://cr.yp.to/snuffle/design.pdf

Bernstein, D. J. (n.d.). Salsa20 specification. *The University of Illinois at Chicago*. Retrieved January 30, 2021, from https://cr.yp.to/snuffle/spec.pdf

Bradbury, D. (2013, October 23). *Scrypt-based miners and the new cryptocurrency arms race*. Retrieved from Coindesk: https://www.coindesk.com/markets/2013/10/23/scrypt-based-miners-and-the-new-cryptocurrency-arms-race/

Crowley, P. (2005). Truncated differential cryptanalysis of five rounds. Retrieved January 30, 2021, from http://www.ciphergoth.org/crypto/salsa20/salsa20-cryptanalysis.pdf

CSBreakdown. (2015, May 17). *Scrypt Password Based Key Derivation Function - Cryptography*. Retrieved from YouTube: https://www.youtube.com/watch?v=TkWAgeSYL\_Q&t=608s

Elgamal, T. (1985, July 4). A Public Key Cryptosystem and a Signature Scheme Based on Discrete Logarithms. *IEEE TRANSACTIONS ON INFORMATION THEORY*. Retrieved January 30, 2021, from https://caislab.kaist.ac.kr/lecture/2010/spring/cs548/basic/B02.pdf

Javed, M. Y. (2012). An experimental study on Performance Evaluation of Asymmetric Encryption Algorithm. *WSEAS 3rd European Conference of Computer Science (WSEAS ECCS 2012)*.

Kamel, S. O. (2012, June). *Novel TEA Algorithm for IP Telephony System*. Retrieved from ResearchGate: https://www.researchgate.net/figure/RC5-Encryption-and-Decryption\_fig4\_270723717

Kushilevitz, E., & Biryukov, A. (1998). Improved Cryptanalysis of RC5. *EUROCRYPT*.

Lectures, W. (2021, September 14). *NS - Unit -I - RC5 Algorithm*. Retrieved from YouTube: https://www.youtube.com/watch?v=BKQoRBezanY&t=519s

Percival, C. (n.d.). STRONGER KEY DERIVATION VIA SEQUENTIAL MEMORY-HARD FUNCTIONS. Retrieved January 30, 2021, from http://www.tarsnap.com/scrypt/scrypt.pdf

Percival, C. (n.d.). *The scrypt key derivation function*. Retrieved from Tarsnap: http://www.tarsnap.com/scrypt.html

Percival, C., Tarsnap, Josefsson, S., & Simon Josefsson Datakonsult. (2016, August). *The scrypt Password-Based Key Derivation Function*. Retrieved from Datatracker: https://datatracker.ietf.org/doc/html/rfc7914

Rakuten Viber. (2016). *VIBER ENCRYPTION.* Retrieved from Viber: https://www.viber.com/app/uploads/viber-encryption-overview.pdf

Rivest, R. L. (1997). The RC5 Encryption Algorithm. *MIT Labaratory for Computer Science*.

Team member’s contributions:

Xihe: Salsa20, half of RC5, half of ElGamal, part of source code for cryptosystem, presentation slides

Dominic: Scrypt, half of RC5, half of Elgamal, source code for cryptosystem, presentation slides

Jia Chen: Agreeable but contributed very little and did not play an integral part in the report

Quang Minh: Agreeable but contributed very little and did not play an integral part in the report