Diffie-Hellman Key Exchange Modification using Blowfish Algorithm to Prevent Logjam Attack

Aldo Adrian, Maya Cendana, Silvester Dian Handy Permana
Informatics Engineering Study Program, Faculty of Creative Industry and Telematics, Trilogi University
Aldoadrian999@gmail.com

***Abstract*—Diffie-Hellman Key Exchange promises secure connections using modulus computation. However, there is a flaw in its implementation which makes it vulnerable, especially to an attack called *Logjam Attack*. Therefore, a new key exchange algorithm was developed to prevent this attack. The proposed algorithm is the result of modified Diffie-Hellman Key Exchange using another algorithm, namely the Blowfish algorithm. Modifications that occur in the Diffie-Hellman Key Exchange are at the modulus computation, which were replaced by customized Blowfish encryption algorithm. The encryption process of the Blowfish algorithm used in the proposed algorithm used 136 XOR operations every 64-bits messages, which were about to be encrypted. The Diffie-Hellman modified algorithm was implemented into programs using Java programing language. The modified algorithm program has less memory usage and execution time than Diffie-Hellman Key Exchange program, which was tested. With the replacement of modulus computations with Blowfish encryption at the main process could make the modification algorithm immune to *Logjam Attack*. Therefore, the use of the modification algorithm is more secured than the one without modification.**

***Index Terms*—Blowfish, Diffie-Hellman Key Exchange, Java, Logjam Attack, SSL, TLS**

I. Introduction

The Diffie-Hellman Key Exchange has the ability to provide secure connection by using modulus (*mod*) operation [1]. In this case, it can provide confidential value known only to two authorized parties but unnoticed by other parties. However, it has a weakness in its implementation as it has a misuse value of *Prime*, configured with a relatively similar value [2]. The assault that is able to hack the key exchange using its flaw was put forward by Adrian et al. (2015) is called *Logjam Attack* [2]. According to [2], there were ±78,000 HTTPS servers vulnerable to *Logjam Attack*. The existence of this attack has affected the security of using the Diffie-Hellman Key Exchange. However, there have been 123,754 out of 137,992 most popular sites still using Diffie-Hellman Key Exchange as its connection security algorithm [3].

In relation to the mentioned issues, this study aims to modify the Diffie-Hellman Key Exchange for better connection security. The modification of this key exchange is done by utilizing the Diffie-Hellman Key Exchange with an algorithm called the Blowfish algorithm. It is believed that this modification is able to prevent the success of *Logjam Attack*.

The Blowfish algorithm itself is a symmetric cryptography algorithm that uses only one key for both encryption and decryption process. The advantage of this algorithm is that the encryption process can provide reliable security; requires relatively small process memory and utilises short execution time [4]. Additionally, the Blowfish algorithm is safe to be used [5][6], considering that there has been no effective attack known to hack this algorithm,

The modification algorithm uses the Diffie-Hellman Key Exchange as the cornerstone of its development. The modification in Diffie-Hellman Key Exchange occurs at the modulus computation process, which is replaced by the encryption process of Blowfish algorithm. Therefore, the modified Diffie-Hellman Key Exchange using Blowfish algorithm is invulnerable to *Logjam Attack,* so it is more secure to use.

II. Diffie-Hellman Key Exchange and Its Flaw

Diffie-Hellman Key Exchange is a key exchange/lock management protocol that can provide two different parties to establish a connection in an unsafe network [7]. The algorithm was originally introduced in 1976 by Whitfield Diffie and Martin Hellman, and was separately constructed by Malcolm Williamson [1].

This key exchange algorithm requires six parameters, which are the Prime (*p*), Generator (*g*), private key of party one (*a*), private key of party two (*b*), public key of party one (*A*), and public key of party two (*B*). The mechanism of the Diffie-Hellman algorithm is shown in Figure 1.

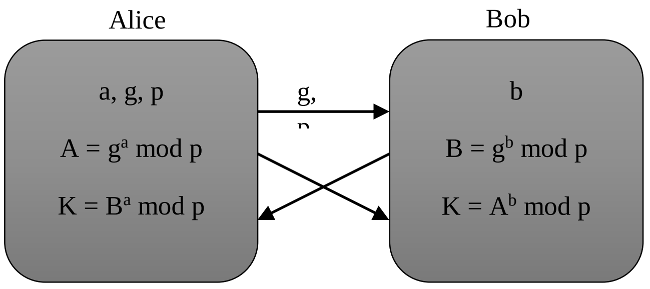


Figure 1: Diffie-Hellman Key Exchange
(Source: [8])

In Figure 1, there are two parties that establish a connection, called Alice and Bob based on the following steps:

1. Alice and Bob agreed on the two values: *p* and *g*. In the case of Client-Server, the Server determines both values. According to Figure 1, the Server is Alice.
2. Each party creates a private key. Alice has the private key *a*, whereas Bob has the private key *b*.
3. Both parties calculate the public key. Alice computes *A*, while Bob computes *B*.

4. Next, is the public key exchange phase. Alice sends her public key (*A*) to Bob, while Bob sends his *B* to Alice.
5. After each party gets the public key from another party, each party then calculate the value of *K* as shown in Figure 1. The value of *K* is the key that only Alice and Bob can know, while the other party cannot get the *K* value even if he/she is capable to receive all values that pass through the connection (*p*, *g*, *A*, and *B*). Therefore, using the Diffie-Hellman Key Exchange, Alice and Bob (*Client-Server*) can communicate securely afterwards.

The main computation of the Diffie-Hellman Key Exchange can be represented by this single formula:

$$y = g^x \bmod p \tag{1}$$

where: y = Public key
g = Generator
x = Parties' private key (a or b, or ab)
p = Prime number

However, the study of Adrian *et al.* (2015) [2] and Revuelto *et al.* (2016) [9] concluded that there is a gap in *Prime* value (*p*) usage. In the implementation of Diffie-Hellman Key Exchange, the *p* value is made relatively similar. This condition has been applied to facilitate the calculation operation process so that it can run faster [2]. Yet, the similarity of *p* value creates a crucial security gap that makes Diffie-Hellman Key Exchange vulnerable to threats, especially with discrete logarithmic method.

Formula (1) could be reversed using discrete logarithmic to earn the value of *x*. The formula is as follows:

$$x = dlog_{a,p}\, y \tag{2}$$

However, formula (2) is difficult to compute and requires a long period of time to complete. The difficulty and process time are also increased as the value of variables increases. Researches done by Adrian *et al.* [2] and Revuelto *et al.* [9] also indicated the importance of the length of bits used during the establishment of the connection. According to Adrian *et al* [2], the Diffie-Hellman Key Exchange with the length of 512-bits, 768-bits and 1024-bits are no longer safe.

Adrian *et al.* [2] then proposed an attack with *Active Man in the Middle* (AMitM) method called *Logjam Attack*. This particular attack uses discrete logarithmic method (formula (2)) to cripple the Diffie-Hellman Key Exchange's main formula (1). Not only it is capable to attack Diffie-Hellman Key Exchange in general, *Logjam Attack* is also capable of decreasing the length of bits used to build the connection to fasten formula (2)'s hacking process (downgrade method).

In the case of *Logjam Attack*, this attack can decrease the number of bits from 768-bits and 1024-bits into 512-bits length. The number of bits is the length of *p* value used during the public key hacking process. The shorter the *p* value, the faster the *Precomputation* process in *Logjam Attack* is completed. Furthermore, the existence of a database in the *Precomputation* stage that stores the previous successful computations results also helps the operation in *Individual Log* to run faster, thus the security of the Diffie-Hellman Key Exchange can be hacked more easily and rapidly.

III. Blowfish Algorithm

Blowfish algorithm is a symmetric cryptography algorithm proposed by Bruce Schneier in 1994. Blowfish was defeated by the Rijndael algorithm to become a standard algorithm, but with its reliability and speed [4], this algorithm is the fastest algorithm used within the global scale [10].

Blowfish algorithm uses blocks of 32-bits length as the requirement of key generation process, during the process of encryption and decryption and requires the block that is equal to 64-bits. The key or data, which is about to be processed should have a minimal 32-bits of length or multiples. If the number of bit is lesser than the multiple of 32, there is a step called *padding* that fills the empty bits in the block until the block is full (32-bits per block) and qualified to be processed [11].

In the key generation process, it requires several variables, namely 18 blocks of variable *P* and four *S* variables, each variable S holds 256 blocks of different values. There are four variables in this process: *P* as temporary key series, *S* as series of bits for the substitution of the data series, *K* as the ideal key to generates variable, and *P'* as the ideal key to be used in the encryption process later [11].

The key generation process requires a raw key sequence of 32-bits to 448-bits. The key sequence is divided into several blocks of 32-bits and given the name of variable *K*. Then, the variable *K* is operated by XOR operation (Exclusive OR logic operator) with *P* variables. Each process starts from the first *K* (*K1*) and the first *P* (*P1*) to produce the first *P'* value (*P'1*). Next, it is continued with the second XOR *K* (*K2*) with the second *P* (*P2*) operation, which results in the *P'2*, and so on until *P18*. If *P* has not touched the number of 18, but the *K* variable is running out, then the variable *K* is reset from the *K1* and the XOR operation is performed with the next *P* variable. The result of this process is the *P'* variable: This is the key variable that is used in the encryption process later [11].

The encryption process in Figure 2 is as follows:

1. The data are divided into blocks. Each block holds 62-bits length of data.
2. Each block has its own process path. The first block is divided into two variables, *xL* and *xR*, each with the length of 32-bits.
3. Iteration starts from 1 to 16, in which each iteration has the following steps:
   a. *xL* is operated by XOR with *P'* (current -th iteration) is obtained from the key generation process.
   b. The product of XOR is then processed by the *F Function*, as shown in Figure 3. The steps in the *F Function* are as follows:
      1) The XOR is the result of xL with P' (current–th iteration) is divided into four 8-bits chunks: Ck1, Ck2, Ck3, and Ck4.
      2) The value held by Ck1 is used to refer to the number index on S1 variable, as well as Ck2 to S2 variable, Ck3 to S3, and Ck4 to S4.

3) After referring to the corresponding *S* index, the *S1* to *S4* variables each returns a block of 32-bits data.
4) The *S1* and *S2* are then computed by modulus 2^{32} (equivalent to XOR operation).
5) The result is then operated with S3 using XOR, then the XOR result is computed with S4 using modulus 2^{32}. As a result, a block of data with 32-bits length is the final product from the *F Function*.

c. The result of the *F Function* is then operated with *xR* using the XOR operation.

d. Next, the values of *xL* and *xR* are exchanged. Now, *xL* = *xR* ,while *xR* = *xL*.

4. The *Feistel Network* iteration stops at the 16th iteration. The value of *xL* and *xR* are no longer be operated by the *F Function* nor exchange their value to one another. Instead, the *xL* is plainly operated with variable *P'17*, while *xR* is operated with *P'18*, both using the XOR operation.
5. The final step of this process is to reunite xL with xR back as one block of 64-bits. Thus, the first block of message has been encrypted. Step 1 to 5 is done until all 64-bits block of message are successfully encrypted.
6. All 64-bit blocks are then reunited into a full length of message right after they are all passed the encryption process. The encryption process is completed.

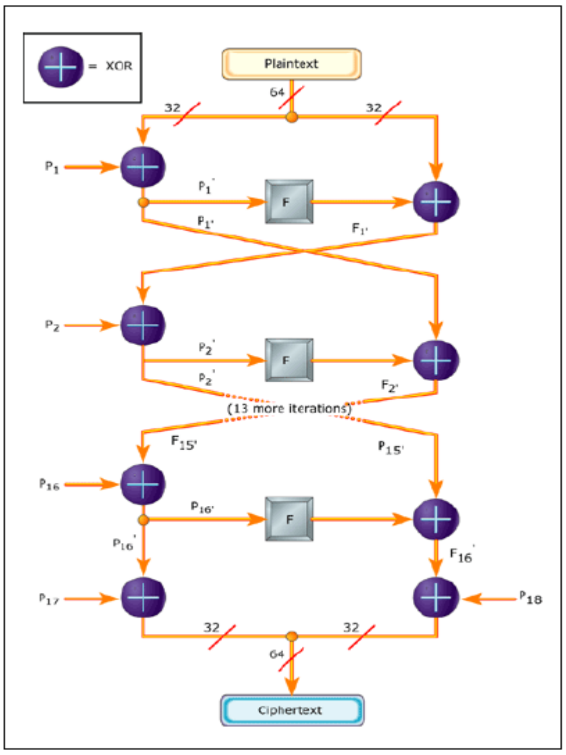


Figure 2: Encryption Process of Blowfish Algorithm
(Source: [11])

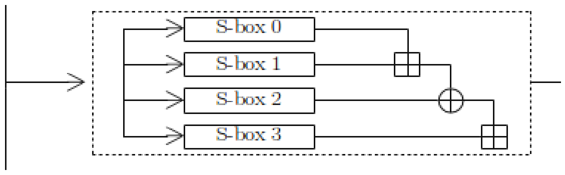


Figure 3: *F Function*
(Source: [5])

A prominent difference between the encryption and the decryption process lies in the use of *P'* variable sequence. In the decryption process, *P'* is processed in a reverse order (starting from *P'18* to *P'1*). In addition to these differences, the process of encryption and decryption also is distinguished by whether there is a key generation process or not. In the encryption process, a new key generation process is required, while the decryption process is only needed to call the key used previously in the encryption process.

IV. Proposed Algorithm

The modified Diffie-Hellman Key Exchange using the Blowfish algorithm can be seen in Figure 4.

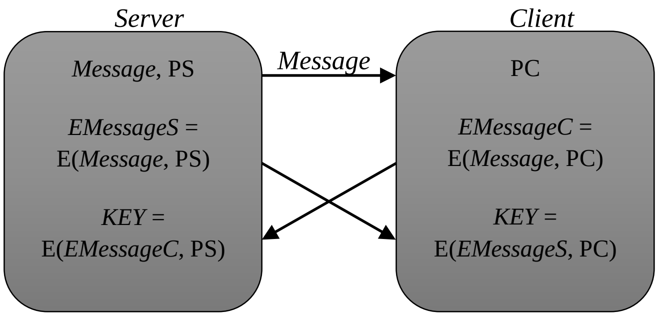


Figure 4: Modified Algorithm

The process described in Figure 4 can be explained as follows:

1. The Server and the Client agree with the *Message* value.
2. Each party generates a private key. The Server generates the Private-key Server (*PS*), while the Client generates the Private-key Client (*PC*).
3. Both sides then encrypt the *Message* with the Blowfish encryption algorithm using their private key. The encryption results are the Encrypted Message-Server (*EMessageS*) for Server and the Encrypted Message-Client (*EMessageC*) for Client.
4. The Server and the Client exchange their encrypted values (*EMessageS* and *EMessageC*).
5. The Client encrypts *EMessageS* with *PC,* while the Server encrypts *EMessageC* with *PS*. Both encryption processes produce the same result known as *KEY,* which is the main product.

In the modification algorithm, Blowfish algorithm's encryption process shown in Figure 4 undergoes several changes. The changes in Blowfish algorithm encryption process used in the modification algorithm is shown in Figure 5.

The steps in Figure 5 can be summarized as follows:

1. This iteration repeats as much as *i* = total chunk/2:
   a. Chunk (block/smallest part of *Message* divided by bit block) *i* * 2 called *L*, while (*i* * 2) + 1 called *R*.
   b. The following iteration repeats as much as *n* = 15 starting from *n* = 0:

1) *P*(*n*) is processed using the *F Function* (Figure 3).
2) *L* is the result of the *F Function* from *P*(*n*-th), which is operated by the XOR with the previous *L*.
3) *P*(*n*) is processed using the *R Function* (Figure 6).
4) *R* is the result of the *R Function* from *P*(*n*-th), which is operated by the XOR with the previous *R*.
5) *L* and *R* exchange values.

c. *L* is operated by the XOR with *P16* after *P16* is operated by the *F Function*.

d. *R* operated by the XOR with *P17* after *P17* is operated by the *R Function*.

e. *L* is now *EL* (Encrypted *L*), the encrypted result for the *i*-th chunk * 2 of the message.

f. *R* is now *ER* (Encrypted *R*), the encrypted result for chunk to-(*i* * 2) + 1 of message.

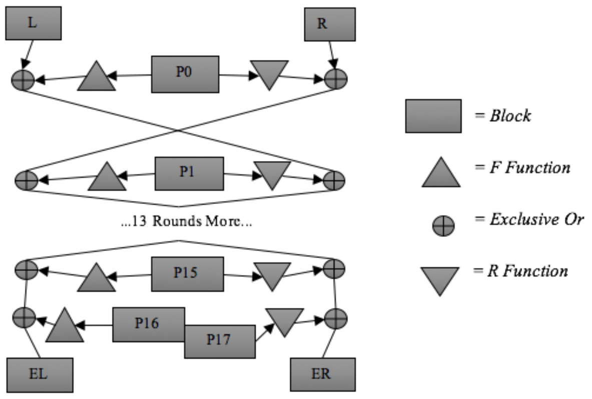


Figure 5: Encryption Algorithm

2. All encrypted chunks are then reassembled together as *Result*, the main output of the Encryption process.

R Function (*Reverse-F Function*) is the inverted form of the *F Function* (Figure 3). The *R function* shown in Figure 6 has a slight difference with the *F Function*, which lies in the order usage of S variable. In the *F Function*, the S variable is divided into four variables: S0, S1, S2, and S3 that are used sequentially. However, in *R Function*, S variables are used in reverse order, beginning from S3, S2, S1, until S0. *R Function* Algorithm can be seen in Figure 6.

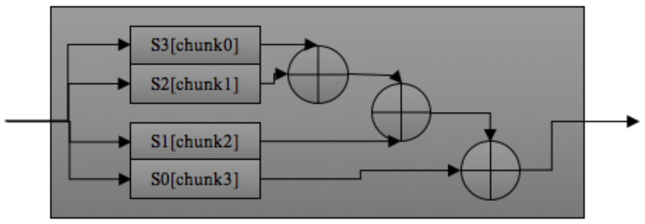
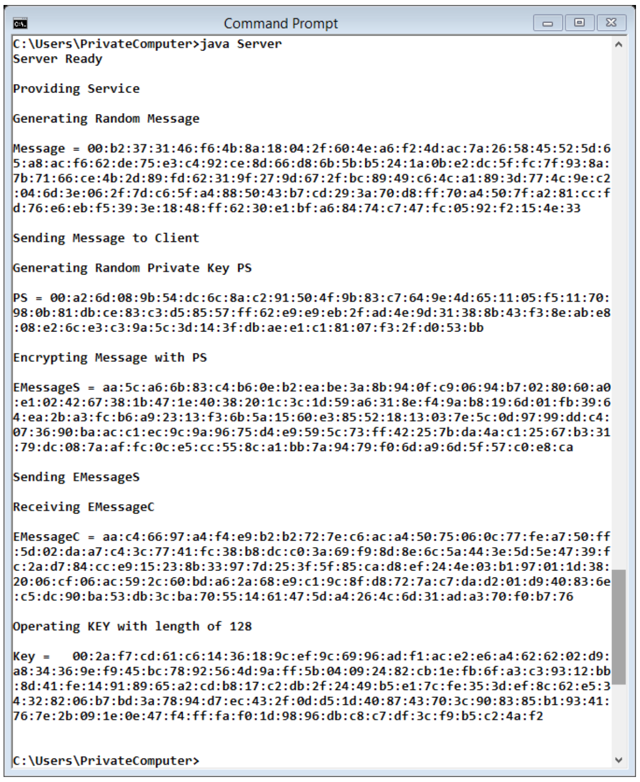


Figure 6: *R Function*

V. IMPLEMENTATION AND VALIDATION

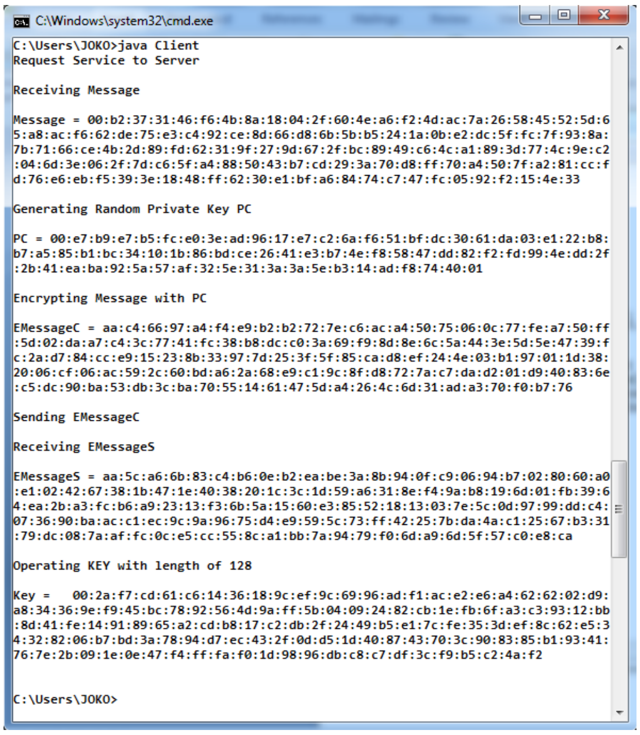
The modified Diffie-Hellman Key Exchange using Blowfish algorithm was then implemented into the form of the program. The program was divided into two main programs, the program for the Client and the Server side.

The programming language used in the Implementation stage is Java. Implementation of this modification algorithm was made using Xcode application with TCP/IP protocol approach (*Transmission Control Protocol/Internet Protocol*–communication *protocol* used by HTTPS). This protocol used *socket/port* number as the Client and the Server's intermediary to exchange messages and establish a secure connection. Both interface of the programs at their running stage can be seen in Figure 7 and 8.



```
C:\Users\PrivateComputer>java Server
Server Ready

Providing Service

Generating Random Message

Message = 00:b2:37:31:46:f6:4b:8a:18:04:2f:60:4e:a6:f2:4d:ac:7a:26:58:45:52:5d:6
5:a8:ac:f6:62:de:75:e3:c4:92:ce:8d:66:d8:6b:5b:b5:24:1a:0b:e2:dc:5f:fc:7f:93:8a:
7b:71:66:ce:4b:2d:89:fd:62:31:9f:27:9d:67:2f:bc:89:49:c6:4c:a1:89:3d:77:4c:9e:c2
:04:6d:3e:06:2f:7d:c6:5f:a4:88:50:43:b7:cd:29:3a:70:d8:ff:70:a4:50:7f:a2:81:cc:f
d:76:e6:eb:f5:39:3e:18:48:ff:62:30:e1:bf:a6:84:74:c7:47:fc:05:92:f2:15:4e:33

Sending Message to Client

Generating Random Private Key PS

PS = 00:a2:6d:08:9b:54:dc:6c:8a:c2:91:50:4f:9b:83:c7:64:9e:4d:65:11:05:f5:11:70:
98:0b:81:db:ce:83:c3:d5:85:57:ff:62:e9:e9:eb:2f:ad:4e:9d:31:38:8b:43:f3:8e:ab:e8
:08:e2:6c:e3:c3:9a:5c:3d:14:3f:db:ae:e1:c1:81:07:f3:2f:d0:53:bb

Encrypting Message with PS

EMessageS = aa:5c:a6:6b:83:c4:b6:0e:b2:ea:be:3a:8b:94:0f:c9:06:94:b7:02:80:60:a0
:e1:02:42:67:38:1b:47:1e:40:38:20:1c:3c:1d:59:a6:31:8e:f4:9a:b8:19:6d:01:fb:39:6
4:ea:2b:a3:fc:b6:a9:23:13:f3:6b:5a:15:60:e3:85:52:18:13:03:7e:5c:0d:97:99:dd:c4:
07:36:90:ba:ac:c1:ec:9c:9a:96:75:d4:e9:59:5c:73:ff:42:25:7b:da:4a:c1:25:67:b3:31
:79:dc:08:7a:af:fc:0c:e5:cc:55:8c:a1:bb:7a:94:79:f0:6d:a9:6d:5f:57:c0:e8:ca

Sending EMessageS

Receiving EMessageC

EMessageC = aa:c4:66:97:a4:f4:e9:b2:b2:72:7e:c6:ac:a4:50:75:06:0c:77:fe:a7:50:ff
:5d:02:da:a7:c4:3c:77:41:fc:38:b8:dc:c0:3a:69:f9:8d:8e:6c:5a:44:3e:5d:5e:47:39:f
c:2a:d7:84:cc:e9:15:23:8b:33:97:7d:25:3f:5f:85:ca:d8:ef:24:4e:03:b1:97:01:1d:38:
20:06:cf:06:ac:59:2c:60:bd:a6:2a:68:e9:c1:9c:8f:d8:72:7a:c7:da:d2:01:d9:40:83:6e
:c5:dc:90:ba:53:db:3c:ba:70:55:14:61:47:5d:a4:26:4c:6d:31:ad:a3:70:f0:b7:76

Operating KEY with length of 128

Key =   00:2a:f7:cd:61:c6:14:36:18:9c:ef:9c:69:96:ad:f1:ac:e2:e6:a4:62:62:02:d9:
a8:34:36:9e:f9:45:bc:78:92:56:4d:9a:ff:5b:04:09:24:82:cb:1e:fb:6f:a3:c3:93:12:bb
:8d:41:fe:14:91:89:65:a2:cd:b8:17:c2:db:2f:24:49:b5:e1:7c:fe:35:3d:ef:8c:62:e5:3
4:32:82:06:b7:bd:3a:78:94:d7:ec:43:2f:0d:d5:1d:40:87:43:70:3c:90:83:85:b1:93:41:
76:7e:2b:09:1e:0e:47:f4:ff:fa:f0:1d:98:96:db:c8:c7:df:3c:f9:b5:c2:4a:f2


C:\Users\PrivateComputer>
```

Figure 7: Server Program



```
C:\Users\JOKO>java Client
Request Service to Server

Receiving Message

Message = 00:b2:37:31:46:f6:4b:8a:18:04:2f:60:4e:a6:f2:4d:ac:7a:26:58:45:52:5d:6
5:a8:ac:f6:62:de:75:e3:c4:92:ce:8d:66:d8:6b:5b:b5:24:1a:0b:e2:dc:5f:fc:7f:93:8a:
7b:71:66:ce:4b:2d:89:fd:62:31:9f:27:9d:67:2f:bc:89:49:c6:4c:a1:89:3d:77:4c:9e:c2
:04:6d:3e:06:2f:7d:c6:5f:a4:88:50:43:b7:cd:29:3a:70:d8:ff:70:a4:50:7f:a2:81:cc:f
d:76:e6:eb:f5:39:3e:18:48:ff:62:30:e1:bf:a6:84:74:c7:47:fc:05:92:f2:15:4e:33

Generating Random Private Key PC

PC = 00:e7:b9:e7:b5:fc:e0:3e:ad:96:17:e7:c2:6a:f6:51:bf:dc:30:61:da:03:e1:22:b8:
b7:a5:85:b1:bc:34:10:1b:86:bd:ce:26:41:e3:b7:4e:f8:58:47:dd:82:f2:fd:99:4e:dd:2f
:2b:41:ea:ba:92:5a:57:af:32:5e:31:3a:3a:5e:b3:14:ad:f8:74:40:01

Encrypting Message with PC

EMessageC = aa:c4:66:97:a4:f4:e9:b2:b2:72:7e:c6:ac:a4:50:75:06:0c:77:fe:a7:50:ff
:5d:02:da:a7:c4:3c:77:41:fc:38:b8:dc:c0:3a:69:f9:8d:8e:6c:5a:44:3e:5d:5e:47:39:f
c:2a:d7:84:cc:e9:15:23:8b:33:97:7d:25:3f:5f:85:ca:d8:ef:24:4e:03:b1:97:01:1d:38:
20:06:cf:06:ac:59:2c:60:bd:a6:2a:68:e9:c1:9c:8f:d8:72:7a:c7:da:d2:01:d9:40:83:6e
:c5:dc:90:ba:53:db:3c:ba:70:55:14:61:47:5d:a4:26:4c:6d:31:ad:a3:70:f0:b7:76

Sending EMessageC

Receiving EMessageS

EMessageS = aa:5c:a6:6b:83:c4:b6:0e:b2:ea:be:3a:8b:94:0f:c9:06:94:b7:02:80:60:a0
:e1:02:42:67:38:1b:47:1e:40:38:20:1c:3c:1d:59:a6:31:8e:f4:9a:b8:19:6d:01:fb:39:6
4:ea:2b:a3:fc:b6:a9:23:13:f3:6b:5a:15:60:e3:85:52:18:13:03:7e:5c:0d:97:99:dd:c4:
07:36:90:ba:ac:c1:ec:9c:9a:96:75:d4:e9:59:5c:73:ff:42:25:7b:da:4a:c1:25:67:b3:31
:79:dc:08:7a:af:fc:0c:e5:cc:55:8c:a1:bb:7a:94:79:f0:6d:a9:6d:5f:57:c0:e8:ca

Operating KEY with length of 128

Key =   00:2a:f7:cd:61:c6:14:36:18:9c:ef:9c:69:96:ad:f1:ac:e2:e6:a4:62:62:02:d9:
a8:34:36:9e:f9:45:bc:78:92:56:4d:9a:ff:5b:04:09:24:82:cb:1e:fb:6f:a3:c3:93:12:bb
:8d:41:fe:14:91:89:65:a2:cd:b8:17:c2:db:2f:24:49:b5:e1:7c:fe:35:3d:ef:8c:62:e5:3
4:32:82:06:b7:bd:3a:78:94:d7:ec:43:2f:0d:d5:1d:40:87:43:70:3c:90:83:85:b1:93:41:
76:7e:2b:09:1e:0e:47:f4:ff:fa:f0:1d:98:96:db:c8:c7:df:3c:f9:b5:c2:4a:f2


C:\Users\JOKO>
```

Figure 8: Client Program

After the modified Diffie-Hellman Key Exchange was successfully implemented, the next step is Validation. This Validation stage aims to find the reliability of modification algorithm to the basic security aspects of a special key exchange algorithm, namely the aspect of Confidentiality (secrecy). According to Pfleeger *et al.* (2015) [12], the aspect of secrecy aims to ensure that secret messages remain secret. This means that no other party knows the messages other than the authorities. In this case, the modified algorithm program must be able to keep confidential variables unsniffed. Those variables are *PS*, *PC*, and *KEY*.

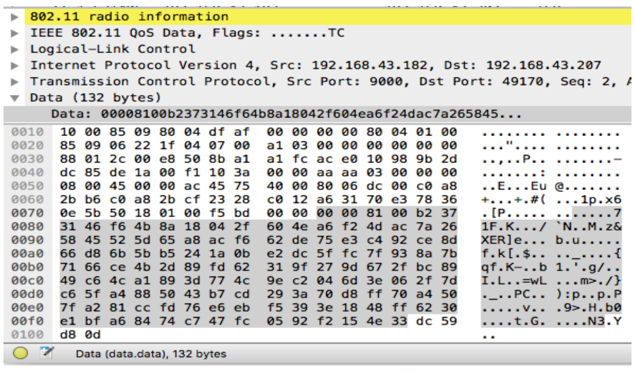
Scenario in the Validation stage are as follows:

1. The Client and the Server computers communicate wirelessly with the connections provided by the Connection Provider. Both parties are connected in *socket*/*port* number 9000.
2. The Sniffer computer is also connected to the Connection Provider. The Sniffer party then captures any data packet that passes within the connection, especially *socket* number 9000.
3. The data packets captured by the Sniffer are then reviewed to determine whether the confidential variables are successfully sniffed or not.

On the Sniffer side, the application used in the Validation stage is Wireshark version 2.2.1. Wireshark was set to run in *Monitor Mode* that is capable to capture every data packets from a specific *interface*. The *Interface* used to be sniffed is Wi-Fi *interface* with TCP protocol filter.

The Validation scenario have been done once as a trial. The results of the trial are shown in Figure 7 (Server side) and Figure 8 (Client side). From the results of the trial, Wireshark on Sniffer managed to capture as many as 26 packages of data packet. The captured data packets consist of:

1. The [SYN] package contains sync between *Client* and Server based on *port/socket*. One package has been successfully sniffed.
2. The [SYN, ACK] package contains the connection approval based on the *port/socket*. One package has been successfully sniffed.
3. The [ACK] package contains confirmation that the submitted data packet has been received by the intended party. Four packages have been successfully sniffed.
4. The [PSH, ACK] package contains messages to be processed by Client and Server. Seven packages have been successfully sniffed.
5. Other packages consist of confirmations and/or transactions packages that do not contain important messages. Eleven packages have been successfully sniffed.



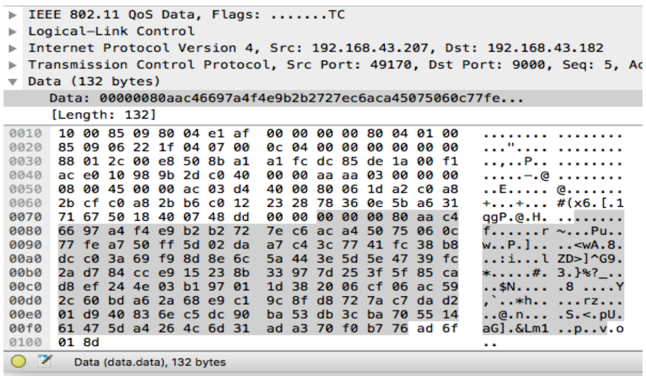
```
▶ 802.11 radio information
▶ IEEE 802.11 QoS Data, Flags: .......TC
▶ Logical-Link Control
▶ Internet Protocol Version 4, Src: 192.168.43.182, Dst: 192.168.43.207
▶ Transmission Control Protocol, Src Port: 9000, Dst Port: 49170, Seq: 2, A
▼ Data (132 bytes)
    Data: 00008100b2373146f64b8a18042f604ea6f24dac7a265845...
0010  10 00 85 09 80 04 df af  00 00 00 00 80 04 01 00   ........ ........
0020  85 09 06 22 1f 04 07 00  a1 03 00 00 00 00 00 00   ...".... ........
0030  88 01 2c 00 e8 50 8b a1  a1 fc ac e0 10 98 9b 2d   ..,..P.. .......-
0040  dc 85 de 1a 00 f1 10 3a  00 00 aa aa 03 00 00 00   .......: ........
0050  08 00 45 00 00 ac 45 75  40 00 80 06 dc 00 c0 a8   ..E...Eu @.......
0060  2b b6 c0 a8 2b cf 23 28  c0 12 a6 31 70 e3 78 36   +...+.#( ...1p.x6
0070  0e 5b 50 18 01 00 f5 bd  00 00 00 00 81 00 b2 37   .[P..... .......7
0080  31 46 f6 4b 8a 18 04 2f  60 4e a6 f2 4d ac 7a 26   1F.K.../ `N..M.z&
0090  58 45 52 5d 65 a8 ac f6  62 de 75 e3 c4 92 ce 8d   XER]e... b.u.....
00a0  66 d8 6b 5b b5 24 1a 0b  e2 dc 5f fc 7f 93 8a 7b   f.k[.$.. .._....{
00b0  71 66 ce 4b 2d 89 fd 62  31 9f 27 9d 67 2f bc 89   qf.K-..b 1.'.g/..
00c0  49 c6 4c a1 89 3d 77 4c  9e c2 04 6d 3e 06 2f 7d   I.L..=wL ...m>./}
00d0  c6 5f a4 88 50 43 b7 cd  29 3a 70 d8 ff 70 a4 50   ._..PC.. ):p..p.P
00e0  7f a2 81 cc fd 76 e6 eb  f5 39 3e 18 48 ff 62 30   .....v.. .9>.H.b0
00f0  e1 bf a6 84 74 c7 47 fc  05 92 f2 15 4e 33 dc 59   ....t.G. ....N3.Y
0100  d8 0d                                              ..
Data (data.data), 132 bytes
```

Figure 9: *Message* sniffed

In Figure 9, a row of hexadecimal values (left highlight) is exactly at the same line as the *Message* variable found in Figure 7 and 8. This specific series refer to the *Message* variables, which was exchanged during the connection building process.

The hexadecimal rows in Figure 10 are identical to those shown in Figure 7 and 8. The hexadecimal rows are the contents of the *EMessageC* variable, which is the result of *Message* encryption performed by the Client using the *PC* variable.

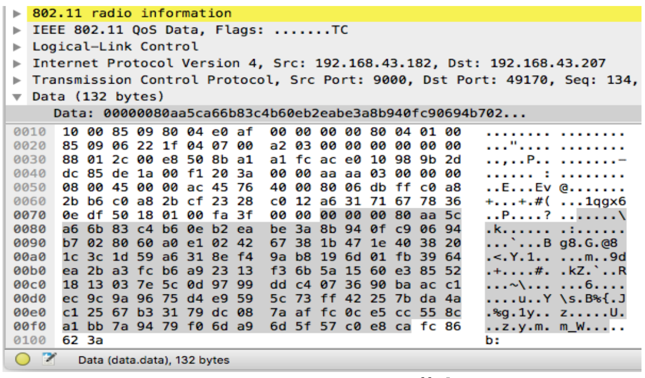
Figure 11 shows a hexadecimal row that is similar to Figure 7 and 8, which is the value of *EMessageS* variable. This variable is the result of *Message* encryption performed by the Server using its private key known as *PS*.



```
▶ IEEE 802.11 QoS Data, Flags: .......TC
▶ Logical-Link Control
▶ Internet Protocol Version 4, Src: 192.168.43.207, Dst: 192.168.43.182
▶ Transmission Control Protocol, Src Port: 49170, Dst Port: 9000, Seq: 5, Ac
▼ Data (132 bytes)
    Data: 00000080aac46697a4f4e9b2b2727ec6aca45075060c77fe...
    [Length: 132]
0010  10 00 85 09 80 04 e1 af  00 00 00 00 80 04 01 00   ........ ........
0020  85 09 06 22 1f 04 07 00  0c 04 00 00 00 00 00 00   ...".... ........
0030  88 01 2c 00 e8 50 8b a1  a1 fc dc 85 de 1a 00 f1   ..,..P.. ........
0040  ac e0 10 98 9b 2d c0 40  00 00 aa aa 03 00 00 00   .....-.@ ........
0050  08 00 45 00 00 ac 03 d4  40 00 80 06 1d a2 c0 a8   ..E..... @.......
0060  2b cf c0 a8 2b b6 c0 12  23 28 78 36 0e 5b a6 31   +...+... #(x6.[.1
0070  71 67 50 18 40 07 48 dd  00 00 00 00 00 80 aa c4   qgP.@.H. ........
0080  66 97 a4 f4 e9 b2 b2 72  7e c6 ac a4 50 75 06 0c   f......r ~...Pu..
0090  77 fe a7 50 ff 5d 02 da  a7 c4 3c 77 41 fc 38 b8   w..P.].. ..<wA.8.
00a0  dc c0 3a 69 f9 8d 8e 6c  5a 44 3e 5d 5e 47 39 fc   ..:i...l ZD>]^G9.
00b0  2a d7 84 cc e9 15 23 8b  33 97 7d 25 3f 5f 85 ca   *.....#. 3.}%?_..
00c0  d8 ef 24 4e 03 b1 97 01  1d 38 20 06 cf 06 ac 59   ..$N.... .8 ....Y
00d0  2c 60 bd a6 2a 68 e9 c1  9c 8f d8 72 7a c7 da d2   ,`..*h.. ...rz...
00e0  01 d9 40 83 6e c5 dc 90  ba 53 db 3c ba 70 55 14   ..@.n... .S.<.pU.
00f0  61 47 5d a4 26 4c 6d 31  ad a3 70 f0 b7 76 ad 6f   aG].&Lm1 ..p..v.o
0100  01 8d                                              ..
Data (data.data), 132 bytes
```

Figure 10: *EMessageC* sniffed

After analyzing all the data packets that had been stacked, there are messages/variables used in the modification algorithm process that have been successfully sniffed. Messages that have been successfully sniffed by Sniffer are: *Message* (Figure9), *EMessageC* (Figure 10), and *EMessageS* (Figure 11). Those three messages were found in each of the three different packets, while the remaining four [PSH, ACK] data packets did not contain any important messages used during the connection building process.



```
▶ 802.11 radio information
▶ IEEE 802.11 QoS Data, Flags: .......TC
▶ Logical-Link Control
▶ Internet Protocol Version 4, Src: 192.168.43.182, Dst: 192.168.43.207
▶ Transmission Control Protocol, Src Port: 9000, Dst Port: 49170, Seq: 134,
▼ Data (132 bytes)
    Data: 00000080aa5ca66b83c4b60eb2eabe3a8b940fc90694b702...
0010  10 00 85 09 80 04 e0 af  00 00 00 00 80 04 01 00   ........ ........
0020  85 09 06 22 1f 04 07 00  a2 03 00 00 00 00 00 00   ...".... ........
0030  88 01 2c 00 e8 50 8b a1  a1 fc ac e0 10 98 9b 2d   ..,..P.. .......-
0040  dc 85 de 1a 00 f1 20 3a  00 00 aa aa 03 00 00 00   ...... : ........
0050  08 00 45 00 00 ac 45 76  40 00 80 06 db ff c0 a8   ..E...Ev @.......
0060  2b b6 c0 a8 2b cf 23 28  c0 12 a6 31 71 67 78 36   +...+.#( ...1qgx6
0070  0e df 50 18 01 00 fa 3f  00 00 00 00 00 80 aa 5c   ..P....? .......\
0080  a6 6b 83 c4 b6 0e b2 ea  be 3a 8b 94 0f c9 06 94   .k...... .:......
0090  b7 02 80 60 a0 e1 02 42  67 38 1b 47 1e 40 38 20   ...`...B g8.G.@8 
00a0  1c 3c 1d 59 a6 31 8e f4  9a b8 19 6d 01 fb 39 64   .<.Y.1.. ...m..9d
00b0  ea 2b a3 fc b6 a9 23 13  f3 6b 5a 15 60 e3 85 52   .+....#. .kZ.`..R
00c0  18 13 03 7e 5c 0d 97 99  dd c4 07 36 90 ba ac c1   ...~\... ...6....
00d0  ec 9c 9a 96 75 d4 e9 59  5c 73 ff 42 25 7b da 4a   ....u..Y \s.B%{.J
00e0  c1 25 67 b3 31 79 dc 08  7a af fc 0c e5 cc 55 8c   .%g.1y.. z.....U.
00f0  a1 bb 7a 94 79 f0 6d a9  6d 5f 57 c0 e8 ca fc 86   ..z.y.m. m_W.....
0100  62 3a                                              b:
Data (data.data), 132 bytes
```

Figure 11: *EMessageS* sniffed

In addition to the trial in Figure 7 and 8, the Validation scenario performed 39 more times. From a total 40 iterations, 40 data were obtained using four different message bit lengths: 1,024, 2,048, 3,072, and 4,096 bits. The reason of bit length variety is in accordance to the suggestion of Adrian *et al* (2015) [2] to use messages with the length of 1024-bits or higher, while the other bit lengths were taken from multiple value of 1,024.

The analysis result of the 40 collected data states that there are no confidential variables successfully sniffed. Results from the existing data analysis shown that *Sniffer* can only sniffed *Message*, *EMessageC*, and *EMessageS* variables, but not *PC*, *PS*, and *KEY*. From the collected data, it can be concluded that the modified Diffie-Hellman Key Exchange using Blowfish algorithm has reliability in the aspect of Confidentiality, which is the major aspect for connection security algorithm. Therefore, the modified algorithm has been proved as reliable and valid to build a secure connection.

VI. Proposed Algorithm's Performance

The security issue which Diffie-Hellman Key Exchange has lies in its real-world implementation that uses relatively similar value of *Prime* (*p*) [2][9]. The problem of using relatively similar *p* value can affect the process of *Precomputation* in *Logjam Attack* to run faster. As the process of *Precomputation* becomes faster, the *Individual Log* stage in *Logjam Attack* can also be completed a lot quicker [2]. This issue makes the data that are transported within the developed connection in an unsecured state. Hence, Diffie-Hellman Key Exchange is no longer safe to be used.

On the other hand, the modified Diffie-Hellman Key Exchange using Blowfish algorithm—theoretically is against *Logjam Attack*. The value of *p* in the Diffie-Hellman Key Exchange have been replaced with a new variable called *Message*, which does not have to be a prime number and must be built randomly. The *Message* variable makes *Precomputation* stage in *Logjam Attack* becomes useless. This is because the *Precomputation* stage has sub-stages devoted to handle a prime value to generate a factorial matrix. On the other hand, the modified algorithm has no rules to generate and use a prime value for the *Message* variable so that the *Precomputation* stage does not have any effect even if it was working properly (gives result, but the result is useless due to its irrelevance).

In addition to the *Precomputation* stage, the *Individual Log* stage of *Logjam Attack* was also useless to attack the modified algorithm. Essentially, the *Individual Log* stage was specially designed to find the value of *x* (private key) from formula (1) using discrete logarithmic shown by the formula (2). However, since the modified algorithm was no longer using formula (1) but instead the *Feistel Network* algorithm (Figure 5) that uses XOR operations only, the computation process in *Individual Log* stage is irrelevant.

While Logjam Attack's Individual Log stage uses formula (2), the modified algorithm repeatedly uses this formula instead:

$$\begin{gathered} EL = (((L\ XOR\ F(P0))\ XOR\ R(P1)) \ldots XOR\ F(P16)) \\ ER = (((R\ XOR\ R(P0))\ XOR\ F(P1)) \ldots XOR\ R(P16)) \\ EM = EL,\ ER \end{gathered} \tag{3}$$

where: L = Left block with length of 32-bits produced by dividing 64-bits block of *Message*
R = Right block with length of 32-bits produced by dividing 64-bits block of *Message*
EL = Encrypted L
ER = Encrypted R
P-th = Private key block with 32-bits length
F() = *F Function* (Figure 3)
R() = *Reverse F Function* (Figure 6)
EM = Encrypted Message with 32-bits length

An XOR operation can indeed be represented in modulus equation with the same output as XOR's. Thus, XOR operation was also vulnerable to logarithmic computation that *Individual Log* can handle. However, the *Individual Log* compute logarithmic computation in one single time only, to determine the value of *x* in formula (1) using formula (2) without any additional iterations. Meanwhile, the XOR operations in the modification algorithm were not only performed one time but 136 times in every 64-bits message which is about to be processed (Figure 5, Figure 3, and Figure 6 combined). Formula (2) can only handle one single computation of modulus (modulo 2^{32} = this particular XOR case) but incapable to handle as many as 136 computations multiplied by every 64-bits of *Message*. This modification does provide a fact that the logarithmic computation in *Individual Log* stage is incapable to hack *Feistel Network* algorithm used by the modified algorithm.

Therefore, *Logjam Attack* is irrelevant as an attack method to hack the modified algorithm. Furthermore, the absence of patterns in the variables used while building a connection also results in higher difficultly to be hacked. It can also be concluded that the modified Diffie-Hellman Key Exchange using Blowfish algorithm can minimize the flaw of Diffie-Hellman Key Exchange prior to the modification.

Apart from its immunity to *Logjam Attack*, the modified algorithm has been found to be more advanced than the basic Diffie-Hellman Key Exchange's performance. As shown in Figure 12, the performance of the modified algorithm and the original one (Diffie-Hellman Key Exchange) in the memory usage has quite a large range of difference. At its maximum point (4096-bit message length), the memory usage performed by the modified algorithm reached 344,522.4 B or 344.5 KB only, while the original Diffie-Hellman Key Exchange could reach 363.828.4 B or 363.8 KB. The range of memory usage in both algorithms is 19.3 KB. This fact suggests that the modified algorithm requires less memory usage than the algorithm before it was modified.

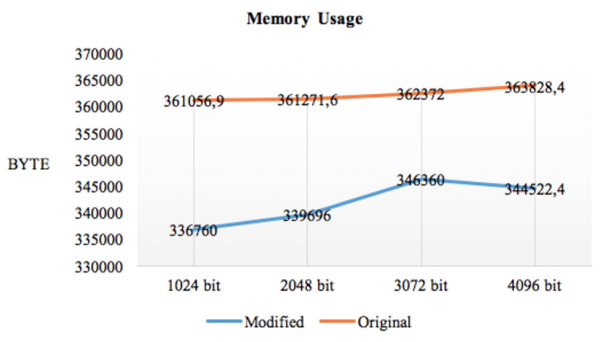


Figure 12: Memory Usage Comparison

The 40 data received from a series of trials are also used to analyze the execution time speed comparison as shown in Figure 13. The execution times for both modified and original algorithms increase as the length of bit message increases, but the execution time of the original algorithm increases greatly than the modified algorithm's. The original algorithm took an average time of 126.957 seconds or 2 minutes 6.956 seconds for messages with 4096-bit length, while the modified algorithm was consistent with the processing time under 1 second, specifically, 0.7 second for the same message length. This comparison can provide a fact that the modified Diffie-Hellman Key Exchange using Blowfish algorithm has faster processing time than the original algorithm.

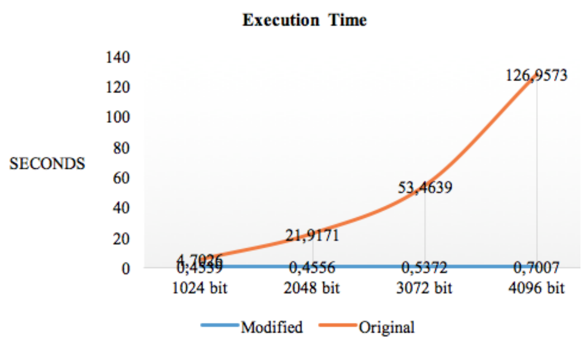


Figure 13: Execution Time Comparison

VII. Conclusion and Suggestion

The conclusions of this study are as follows:

1. The Modified Diffie-Hellman Key Exchange using Blowfish algorithm was done by replacing the modulus computation process with the *Feistel Network* algorithm from the Blowfish algorithm. Substitution of this process has successfully provided an immunity from the *Logjam Attack*. This was because the *Precomputation* and *Individual Log* stages in *Logjam Attack* were incapable to hack the XOR operations performed as much as 136 times per 64-bits of processed messages.
2. The modified Diffie-Hellman Key Diffie using the Blowfish algorithm was successfully fulfilled the secrecy aspect of Confidentiality. This fact was proven by secret variables used during the connection building that cannot be sniffed by Sniffer (attacker), implying that it is secure for general use.

From the research that has been discussed, the researchers suggest several things, which are:

1. The part of Blowfish algorithm used in the modified Diffie-Hellman Key Exchange was only in its encryption process. The decryption process in the Blowfish algorithm was not used in any process within the modified algorithm. Therefore, the decryption process can be incorporated into the modified algorithm so that the security level of the built connection is—perhaps—even higher. It is a recommendation to change a few steps in the decryption process to match the purpose of building the secure connection.
2. In the modified Diffie-Hellman Key Exchange using the Blowfish algorithm implementation program, there were *S* variables (*S0*, *S1*, *S2*, and *S3*), which are bunch of libraries. These variables are recommended not to be used anymore. These *S* variables can be replaced with other variables that are randomly generated and always different in each process, during the process of building new connections. This way of modification could increase the difficulty of hacking because the modification algorithm will no longer use static libraries but fully dynamic variables.

References

[1] Boni, S., Bhatt, J., & Bhat, S. 2015. “International Journal of Computer Applications”, *Improving The Diffie-Hellman Key Exchange Algorithm by Proposing the Multiplicative Key Exchange Algorithm*. Vol 130 (15). 7-10.

[2] Adrian, D., Bhargavan, K., Durumeric, Z., Gaudry, P., Green, M., Halderman, J. A., Heninger, N., Springall, D., Thome, E., Valenta, L., VanderSloot, B., Wustrow, E., Zanella-Béguelin, S., & Zimmermann, P. 2015. “CCS '15 Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security”, *Imperfect Forward Secrecy: How Diffie-Hellman Fails In Practice*. pp 5-17.

[3] Trustworthy Internet Movement. *SSL Pulse*. Accessed date: 22 February 2017. https://trustworthyinternet.org/ssl-pulse/

[4] Shrivastava, V. & Singh, G. 2013. “IJCST”, *Computer Trend with Security by RSA, DES and BLOWFISH Algorithm*. Vol. 4, pp. 618-620.

[5] Patil, S. D. 2013. “IJRRE ST: International Journal of Research Review in Engineering Science and Technology”, *Passwords Management using Blowfish Algorithm*. Vol. 2, pp. 48-52.

[6] Bhanot, R. & Hans, R. 2015. “International Journal of Security and Its Applications”, *A Review and Comperative Analysis of Various Encryption Algorithm*. Vol 9 (4). 289-306.

[7] Ristic, I. 2014. *Bulletproof SSL and TLS*. London: Fiesty Duck.

[8] Ahmed, M., Sanjabi, B., Aldiaz, D., Rezaei, A., & Omotunde, H. 2012. “IJESTI”, Diffie-Hellman and Its Application in Security Protocols. Vol 1(2). 69-73.

[9] Revuelto, V. & Socha, K. 2016. “CERT-EU Security Whitepaper”, *Weakness in Diffie-Hellman Key Exchange Protocol*. Vol 16 (2). 1-7.

[10] Sharma, S. & Bisht, J. S. 2015. “International Journal of Scientific Research in Network Security and Communication”, *Performance Analysis of Data Encryption Algorithms*. Vol. 3, pp. 1-5.

[11] Valmik, N. K. & Kshirsagar, V. K. 2014. “IOSR – Journal of Computer Engineering”, *Blowfish Algorithm*. Vol. 16, 2014, pp. 80-83.

[12] Pfleeger, P. C., Pfleeger, S. L, & Margulies, J. 2015. *Security in Computing Fifth Edition*. Prentice Hall.

[13] Ammarah, P. S., Kaul, V., & Narayankhedkar, S. K. 2014. “Proceeding ICWAC 2014”, *Security Enhancement Algorithm for Data Transmission using Elliptic Curve Diffie-Hellman Key Exchange*. No. 2. 10-16.

[14] Deshmukh, S. & Patil, R. 2014. “International Journal of Computer Science and Information Technologies”, *Hybrid Cryptography Technique Using Modified Diffie-Hellman and RSA*. Vol 5 (6). 7302-7304.

[15] Ibrahem, M. K. & Ali, T. A. M. 2013. “IJCSET”, *Secure Messaging System Using ZKP*. Vol 3 (11). 388-393.

[16] Kaushik, A. & Satvika. 2013. “Proceeding 2nd ICETEM”, *Extended Diffie-Hellman Algorithm for Key Exchange and Management*.

[17] Kurose, J. F. & Ross, K. W. 2014. *Computer Networking Sixth Edition*. Boston: Pearson.

[18] Madhuri, D. M. S., Annapurna, G, Venkataramana, C. H., & Swetha, G. 2015. “BEST: IJMITE”, *Text Hiding Using RSA and Blowfish with Hash-Based LSB Tecnique*. Vol 3 (4). 5-12.

[19] Meyer, C. 2014. *20 Years of SSL/TLS Research an Analysis Of The Internet’s Security Foundation*. Ruhr-University Bochum.

[20] Rachmawanto, E. H. 2010. *Teknik Keamanan Data Menggunakan Kriptografi Dengan Algoritma Vernam Cipher Dan Steganografi Dengan Metode End of File (EoF)*. Semarang.

[21] Singh, S. 2013. “IJRET”, *A Combined Approach Using Triple DES and Blowfish*. Vol 2 (7). 63-67.

[22] Thangavelu, S. & Vijaykumar, V. 2016. “The International Arab Journal of Information Technology”, *Efficient Modified Elliptic Curve Diffie-Hellman Algorithm for VoIP Networks*. Vol 13 (5). 492-500.