The Implementation of XChaCha20-Poly1305 in MQTT Protocol

Ignatius Timothy Manullang (13517044)[1]
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jl. Ganesha 10 Bandung 40132, Indonesia
[1]13517044@stei.itb.ac.id

***Abstract—Message Queue Telemetry Transport* (MQTT) is a network protocol that transports messages between devices through publish-subscribe, which is designed to be able to run at low power and low bandwidth. The MQTT protocol only provides authentication mechanism for security as a default therefore it does not guarantee data privacy and data integrity during data transmission in the implementation of the MQTT protocol. Hence, in order to provide data privacy and data integrity during data transmission, encryption and authentication is needed. The Authenticated Encryption with Associated Data (AEAD) algorithm, XChacha20-Poly1305, is suggested to be implemented with the MQTT Protocol. This is because XChaCha20-Poly1305 is fast, well suited to low-powered devices and in real-time communications, and more secure than the standard ChaCha20-Poly1305. Based on testing, the XChaCha20-Poly1305 algorithm is able to deal with data changes, data insertion and data substitution attacks. This paper presents the results of the XChaCha20-Poly1305 algorithm which has good performance based on the value of memory usage and resistance to attacks.**

Keywords—IoT, MQTT, Security, XChaCha20-Poly1305

I. Introduction

The Internet of Things (IoT) is a system of computing devices, objects, mechanical and digital machines, people or animals that are interrelated and are provided with unique identifiers and the ability to transfer data over a network without requiring human-to-human or human-to-computer interaction. IoT is increasingly used since it helps people. [1]

IoT devices communicate with each other using protocols. There are a number of protocols which can be used. One of those protocols is the Message Queue Telemetry Transport (MQTT) protocol. The MQTT protocol is a lightweight and flexible network protocol due to the fact that it uses a publish-subscribe model which decouples the publisher and subscriber therefore the clients are very small, therefore it can be run at low power and low bandwidth, which is a good fit for IoT. Since the MQTT protocol is a good fit for IoT, it is increasingly used. [2]

However, the MQTT protocol itself has disadvantages. One of which is that the MQTT protocol only serves an authentication mechanism for its security. Therefore, by default, encryption in data transfer is not present which makes data privacy and integrity a problem in the implementation of the MQTT protocol. [3]

One of the solutions that can be implemented to solve data privacy and integrity is to implement an Authenticated Encryption with Associated Data (AEAD) algorithm. This type of algorithm has the capability to perform authenticated encryption which provides a way to protect data privacy and integrity [7]. An example of an AEAD algorithm is ChaCha20-Poly1305, proposed through RFC 7539 [9] which is a combination of ChaCha20 stream cipher algorithm and Poly1305 MAC algorithm. Currently, the ChaCha20-Poly1305 algorithm acts as a replacement for AES, should it become vulnerable, and has been used by Google as a replacement for RC4 [4] which is vulnerable to attacks and prohibited by RFC 7465 [5].

However, there is a more secure AEAD algorithm than ChaCha20-Poly1305, which is XChaCha20-Poly1305. XChaCha20-Poly1305 is an AEAD algorithm which is a combination of XChaCha20 stream cipher algorithm and Poly1305 MAC algorithm. It is more secure due to the fact that it uses an extended nonce, which is a192-bit nonce, over the 92-bit nonce used by ChaCha20-Poly1305 [12]. Therefore, the implementation of XChaCha20-Poly1305 in MQTT protocol, which is discussed in this paper, is expected to provide better data security solutions, especially in terms of privacy and data integrity, over ChaCha20-Poly1305.

II. Basic Theory

A. MQTT

Message Queue Telemetry Transport (MQTT) Protocol is an OASIS standard messaging protocol that is designed for communication in Machine to Machine (M2M) and Internet of Things (IoT). It runs over TCP/IP or over other network protocols that provide ordered, lossless, bi-directional connections. It uses the publish/subscribe message pattern which provides one-to-many message distribution and decoupling of applications. It has advantages such as being extremely lightweight and efficient due to having MQTT clients that are very small and require minimal resources so that it can be used on small microcontrollers and due to having MQTT message headers that are small to optimize network bandwidth, being able to scale to connect with millions of IoT devices, being able to support bi-directional communications that allows for messaging between device to cloud and cloud to device and makes for easy broadcasting messages to groups of things, being able to reliably deliver message, which is important for many IoT use cases, through 3 defined quality of service levels: 0 - at most once, 1 - at least once and 2 - exactly once, and being able to easily incorporate encryption methods to encrypt messages and incorporate modern authentication protocols to authenticate clients. [10]

B. Authenticated Encryption with Associated Data (AEAD) Algorithm

The Authenticated Encryption with Associated Data (AEAD) Algorithm is a type of algorithm that has the capability to perform authenticated encryption, which is a form of encryption that, in addition to providing confidentiality for the plaintext that is encrypted, provides a way to check its integrity and authenticity, alongside the ability to check the integrity and authenticity of some Associated Data (AD) or "additional authenticated data" that is not encrypted. [7]

This type of algorithm was designed in order to fulfill the requirement of both confidentiality, which is a security service that ensures that data is available only to those authorized to obtain it, and message authentication, which is the service that ensures that data has not been altered or forged by unauthorized entities, by many cryptographic applications. [7]

The interface of an AEAD algorithm includes two operations, authenticated encryption and authenticated decryption, which has the default input and output of octet strings or 'sequence of bytes'. An implementation of an AEAD algorithm may accept additional inputs, however such extensions must not affect interoperability with other implementations. [7]

The authenticated encryption operation in AEAD algorithm has four inputs, each of which is an octet string:

1. A secret key K, which has to be generated using a method that is uniformly random or pseudorandom
2. A nonce N, which has to be distinct, unless each and every nonce is zero-length.
3. A plaintext P, which contains the data to be encrypted and authenticated
4. The associated data A, which contains the data to be authenticated, but not encrypted.

It has a single output:

1. A ciphertext C, which is at least as long as the plaintext, or an indication that the requested encryption operation could not be performed. [7]

The authenticated decryption operation in AEAD algorithm has four inputs, which are K, N, C, and A, as defined above, and a single output which is either a plaintext value P or a special symbol FAIL that indicates that the inputs are not authentic. [7]

C. Message Authentication Code

Message Authentication Code (MAC) is an identification code for message authentication as proof of data and message integrity obtained through processing the data or message using a symmetric private key. The MAC code will then be combined with the data or message and sent to the receiver. The receiver will use the same private key to generate the MAC code of the received data or message and compare it with the MAC code received from the sender. [3]

The process of using MAC for authentication is shown in the following illustration.

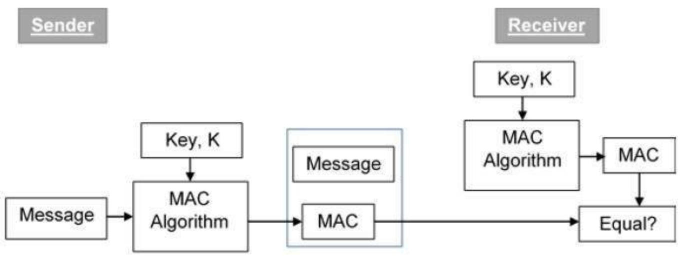


Figure 1. The process of using MAC for authentication [8]

D. ChaCha

Chacha is a stream cipher which is developed by D. J. Bernstein in 2008. It was also published through RFC7539 by Y. Nir from Check Point and A. Langley from Google, Inc. in May 2015. [5] ChaCha's initial state includes a 128-bit constant, a 256-bit key, a 64-bit counter, and a 64-bit nonce, arranged as a 4x4 matrix of 32-bit words, which is shown in the table below, alongside the matrix index from 0 to 15. [10]

Table 1. Chacha matrix, with index from 0 to 15 [10]

(0) constant	(1) constant	(2) constant	(3) constant
(4) key	(5) key	(6) key	(7) key
(8) key	(9) key	(10) key	(11) key
(12) input	(13) input	(14) input	(15) input

Chacha, in its quarter-round QR(state, a, b, c, d), also uses 4 additions, 4 xors and 4 rotations to invertibly update 4 32-bit state words. In particular, ChaCha updates each word twice rather than once, as follows:

ChaCha Quarter Round in Pseudocode [10]

```
a += b; d ^= a; d <<<= 16;
c += d; b ^= c; b <<<= 12;
a += b; d ^= a; d <<<= 8;
c += d; b ^= c; b <<<= 7;
```

Through these operations, the ChaCha quarter-round diffuses changes through *bits* with an average change of 12.5 output bits. Moreover, the fact that two of the rotates are multiples of 8 (16 and 8), allows for a small optimization on some architectures including x86 [11]. In addition, the input format has been arranged to support an efficient SSE implementation optimization, in which alternating rounds down columns and across rows is replaced by alternating rounds down columns and along diagonals. [10]

ChaCha uses a double round which is shown below

ChaCha Double Round in Pseudocode [10]

```
// Odd round
QR(state, 0, 4, 8, 12)  // 1st column
QR(state, 1, 5, 9, 13)  // 2nd column
QR(state, 2, 6, 10, 14) // 3rd column
QR(state, 3, 7, 11, 15) // 4th column
// Even round
QR(state, 0, 5, 10, 15) // diagonal 1 (main diagonal)
QR(state, 1, 6, 11, 12) // diagonal 2
QR(state, 2, 7, 8, 13)  // diagonal 3
QR(state, 3, 4, 9, 14)  // diagonal 4
```

E. ChaCha20

Chacha20 is a variant of ChaCha which includes a block function that transforms a ChaCha20 state by running 10 iterations of ChaCha double rounds.

The input of a ChaCha20 block function includes:

1. A 256-bit key which is treated as a concatenation of eight 32-bit little-endian integers
2. A 96-bit nonce which is treated as a concatenation of three 32-bit little-endian integers
3. A 32-bit block count parameter which is treated as a 32-bit little-endian integer

The output is 64 random-looking bytes. [9]

In pseudocode, the ChaCha20 Block Function is shown below.

ChaCha20 Block Function in Pseudocode. [9]

```
inner_block (state):
   QR(state, 0, 4, 8,12)
   QR(state, 1, 5, 9,13)
   QR(state, 2, 6,10,14)
   QR(state, 3, 7,11,15)
   QR(state, 0, 5,10,15)
   QR(state, 1, 6,11,12)
   QR(state, 2, 7, 8,13)
   QR(state, 3, 4, 9,14)
   end

chacha20_block(key, counter, nonce):
   state = constants | key | counter | nonce
   working_state = state
   for i=1 upto 10
      inner_block(working_state)
   end
   state += working_state
   return serialize(state)
   end
```

For each block, the ChaCha20 Encryption Algorithm calls the ChaCha20 block function with the same key and nonce. After each block, the block counter parameter is increased. The resulting state is serialized by writing the numbers in little-endian order which creates a keystream block.

The input of the ChaCha20 Encryption Algorithm includes:

1. A 256-bit key
2. A 32-bit initial counter
3. A 96-bit nonce.
4. An arbitrary-length plaintext

The output is the ciphertext with the same length as the plaintext.

In pseudocode, the ChaCha20 Encryption Algorithm is shown below.

ChaCha20 Encryption Algorithm in Pseudocode [9]

```
chacha20_encrypt(key, counter, nonce, plaintext):
   for j = 0 upto floor(len(plaintext)/64)-1
      key_stream = chacha20_block(key, counter+j, nonce)
      block = plaintext[(j*64)..(j*64+63)]
      encrypted_message +=  block ^ key_stream
      end
   if ((len(plaintext) % 64) != 0)
      j = floor(len(plaintext)/64)
      key_stream = chacha20_block(key, counter+j, nonce)
      block = plaintext[(j*64)..len(plaintext)-1]
      encrypted_message +=
(block^key_stream)[0..len(plaintext)%64]
      end
   return encrypted_message
   end
```

The ChaCha20 Decryption Algorithm is similar to the ChaCha20 Encryption Algorithm, with the only difference being the plaintext being replaced with the ciphertext. [9]

F. Poly1305

Poly1305 is a message authentication code (MAC) which is created by D. J. Bernstein. It takes a 32-byte one-time key and message and produces a 16-byte tag. This tag is used to authenticate the message.[9]

The input of Poly1305 includes:

1. An arbitrary length message
2. A 256-bit one-time key, which will be partitioned into two equal parts of 16-octet little endian numbers, “r” and “s”, and the pair (r, s) should be unique and unpredictable for each invocation. “r” will then be modified with a clamp function clamp(r) before being used since:
   - 2.1. r[3], r[7], r[11], and r[15] are required to have their top four bits clear (be smaller than 16)
   - 2.2. r[4], r[8], and r[12] are required to have their bottom two bits clear (be divisible by 4)

The output is a 128-bit tag. [9]

The process is as follows:

1. First, the "r" value should be clamped.
2. Next, set the constant prime "P" be 2^130-5: 3fffffffffffffffffffffffffffffffb. Also set a variable "accumulator" to zero.
3. Next, divide the message into 16-byte blocks. The last one might be shorter:
4. Read the block as a little-endian number.
5. Add one bit beyond the number of octets. For a 16-byte block this is equivalent to adding 2^128 to the number. For the shorter block it can be 2^120, 2^112, or any power of two that is evenly divisible by 8, all the way down to 2^8.

6. If the block is not 17 bytes long (the last block), pad it with zeros. This is meaningless if you are treating the blocks as numbers.
7. Add this number to the accumulator.
8. Multiply by "r"
9. Set the accumulator to the result modulo p. To summarize: Acc = ((Acc+block)*r) % p.
10. Finally, the value of the secret key "s" is added to the accumulator, and the 128 least significant bits are serialized in little-endian order to form the tag. [9]

In pseudocode, the Poly1305 algorithm is shown below.

Poly1305 algorithm in pseudocode [9]

```
clamp(r): r &= 0x0ffffffc0ffffffc0ffffffc0fffffff
    poly1305_mac(msg, key):
      r = (little_endian_bytes_to_num(key[0..15])
      clamp(r)
      s = little_endian_num(key[16..31])
      accumulator = 0
      p = (1<<130)-5
      for i=1 upto ceil(msg length in bytes / 16)
        n = little_endian_bytes_to_num(msg[((i-1)*16)..
(i*16)] | [0x01])
        a += n
        a = (r * a) % p
        end
      a += s
      return num_to_16_le_bytes(a)
      end
```

G. ChaCha20-Poly1305

ChaCha20-Poly1305 is an AEAD algorithm which combines ChaCha20 stream cipher algorithm and Poly1305 MAC algorithm. [9]

The input of encryption with ChaCha20-Poly1305 includes:
1. A 256-bit key
2. A 96-bit nonce - different for each invocation with the same key.
3. An arbitrary length plaintext
4. Arbitrary length additional authenticated data (AAD) [9]

The process of encryption with ChaCha20-Poly1305 is as follows:
1. First, a Poly1305 one-time key is generated from the 256-bit key and nonce
2. Next, the ChaCha20 encryption function is called to encrypt the plaintext, using the same key and nonce, and with the initial counter set to 1.
3. Finally, the Poly1305 function is called with the Poly1305 key calculated above, and a message constructed as a concatenation of the following:
   3.1. The AAD
   3.2. padding1 - the padding is up to 15 zero bytes, and it brings the total length so far to an integral multiple of 16. If the length of the AAD was already an integral multiple of 16 bytes, this field is zero-length.
   3.3. The ciphertext
   3.4. padding2 - the padding is up to 15 zero bytes, and it brings the total length so far to an integral multiple of 16. If the length of the ciphertext was already an integral multiple of 16 bytes, this field is zero-length.
   3.5. The length of the additional data in octets (as a 64-bit little-endian integer).
   3.6. The length of the ciphertext in octets (as a 64-bit little-endian integer). [9]

The output of encryption with ChaCha20-Poly1305 is as follows:
1. A ciphertext of the same length as the plaintext.
2. A 128-bit tag, which is the output of the Poly1305 function. [9]

In pseudocode, the ChaCha20-Poly1305 algorithm is as follows:

ChaCha20-Poly1305 algorithm in pseudocode [9]

```
pad16(x):
      if (len(x) % 16)==0
        then return NULL
        else return copies(0, 16-(len(x)%16))
      end

    chacha20_aead_encrypt(aad, key, iv, constant,
plaintext):
      otk = poly1305_key_gen(key, iv, constant)
      nonce = constant | iv
      ciphertext = chacha_encrypt(key, 1, nonce, plaintext)
      mac_data = aad | pad16(aad)
      mac_data |= ciphertext | pad16(ciphertext)
      mac_data |= num_to_4_le_bytes(aad.length)
      mac_data |= num_to_4_le_bytes(ciphertext.length)
      tag = poly1305_mac(mac_data, otk)
      return (ciphertext, tag)
```

In ChaCha20-Poly1305, the decryption algorithm is similar to the encryption algorithm, with the following differences:
1. The roles of ciphertext and plaintext are reversed, so the
2. ChaCha20 encryption function is applied to the ciphertext, producing the plaintext.
3. The Poly1305 function is still run on the AAD and the ciphertext, not the plaintext.
4. The calculated tag is bitwise compared to the received tag. The message is authenticated if and only if the tags match.

H. XChaCha20-Poly1305

XChaCha20-Poly1305 is an AEAD algorithm, and a variant of ChaCha20-Poly1305 that uses an extended nonce, which is a 192-bit nonce, instead of a 96-bit nonce. It combines the XChaCha20 stream cipher algorithm and the Poly1305 MAC algorithm. [12]

The algorithm for XChaCha20-Poly1305 is as follows

1. Pass the key and the first 16 bytes of the 24-byte nonce to HChaCha20 to obtain the subkey.
2. Use the subkey and remaining 8 bytes of the nonce (prefixed with 4 NUL bytes) with ChaCha20-Poly1305 algorithm as normal. [12]

The HChaCha20 algorithm used in XChaCha20-Poly1305 is an intermediary step towards XChaCha20. HChaCha20 is initialized the same way as the ChaCha cipher, except that HChaCha20 uses a 128-bit nonce and has no counter. Instead, the block counter is replaced by the first 32 bits of the nonce. [8]

The process of HChaCha20 algorithm after initialization is as follows:

1. Proceed through the ChaCha rounds as usual.
2. Once the 20 ChaCha rounds have been completed, the first 128 bits and last 128 bits of the ChaCha state (both little-endian) are concatenated, and this 256-bit subkey is returned.[8]

The XChaCha20 algorithm used in XChaCha20-Poly1305 in pseudocode is as follows:

XChaCha20 encryption algorithm used in XChaCha20-Poly1305 in pseudocode [12]

```
xchacha20_encrypt(key, nonce, plaintext):
        subkey = hchacha20(key, nonce[0:15])
        chacha20_nonce = "\x00\x00\x00\x00" + nonce[16:23]
        return chacha20_encrypt(subkey, 1, chacha20_nonce,
plaintext)
```

III. System Design

The system which is going to be implemented contains the MQTT Protocol with XChaCha20-Poly1305 algorithm. The components of the MQTT protocol are the publisher, broker and subscriber. The publisher will send data and MAC to the broker. The broker will receive data from the publisher. The subscriber will receive data from the broker and validate MAC based on the data received from the broker.

The XChaCha20-Poly1305 algorithm will be applied in the publisher and subscriber, to check the data security and integrity on the publisher and subscriber by encryption/decryption and MAC code generation. The implementation design is shown in the Figure below.

The system is illustrated in the image below.

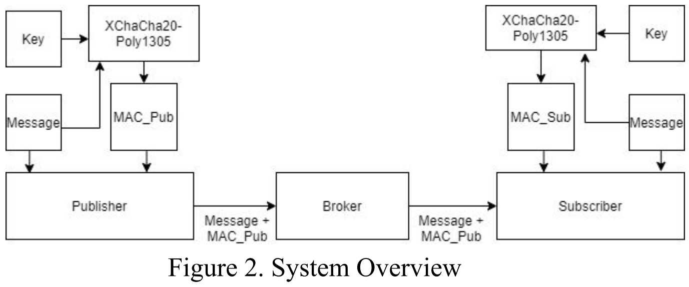
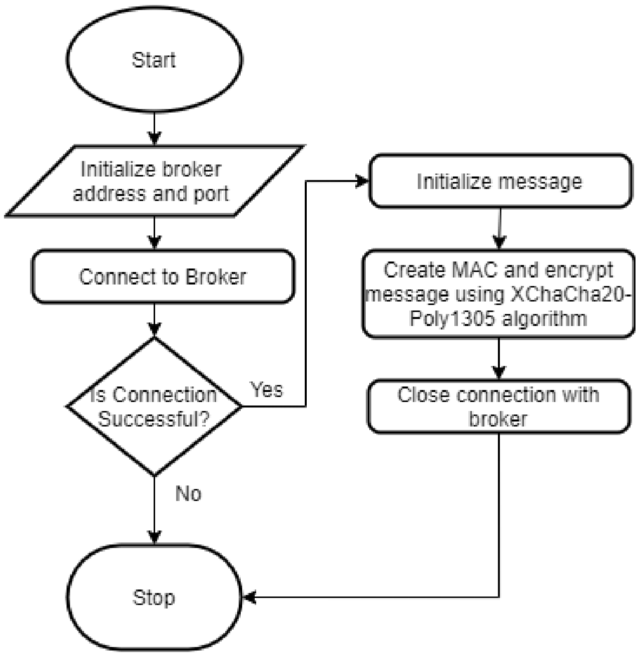
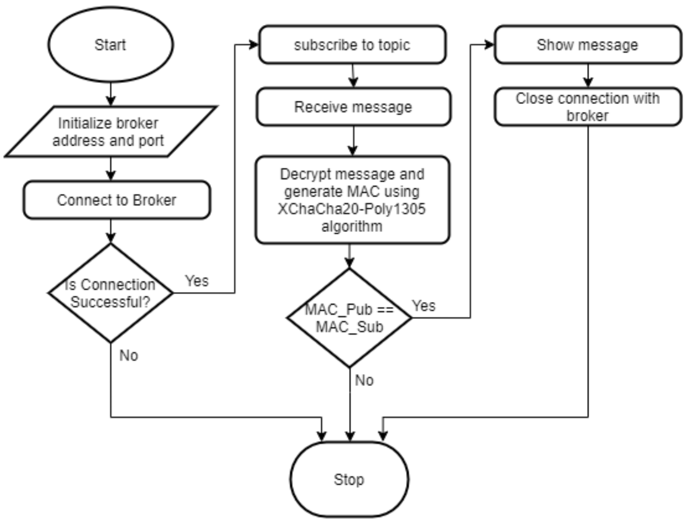


Figure 2. System Overview

The workflow of the components of the system, the publisher and the subscriber, which is going to be implemented, is shown in Figures 3 and 4.


Start
Initialize broker address and port
Connect to Broker
Is Connection Successful?
Yes
No
Initialize message
Create MAC and encrypt message using XChaCha20-Poly1305 algorithm
Close connection with broker
Stop
Figure 3. Publisher Workflow


Start
Initialize broker address and port
Connect to Broker
Is Connection Successful?
Yes
No
subscribe to topic
Receive message
Decrypt message and generate MAC using XChaCha20-Poly1305 algorithm
MAC_Pub == MAC_Sub
Yes
No
Show message
Close connection with broker
Stop
Figure 4. Subscriber Workflow

IV. Testing And Analysis

A. Test Vector Testing

Test Vector Testing is done to make sure the XChaCha20-Poly1305 that is used in this implementation is correct based on the specification of the creator. The encrypted message and MAC generated from the created XChaCha20-Poly1305 will be compared with a test vector which refers to a test vector parameter which has been generated by the designer of the XChaCha20-Poly1305 algorithm [8]. The result of the test is shown below.

Table 2. Test Vector for XChaCha20-Poly1305

Test Vector			
No	Input	Output	Validity

1	Plaintext = 4c616469657320616e642047656e746c656d656e206f662074686520636c617373206f66202739393a204966204920636f756c64206f6666657220796f75206f6e6c79206f6e652074697020666f7220746865206675747572652c2073756e73637265656e20776f756c642062652069742e	Ciphertext = bd6d179d3e83d43b9576579493c0e939572a1700252bfaccbed2902c21396cbb731c7f1b0b4aa6440bf3a82f4eda7e39ae64c6708c54c216cb96b72e1213b4522f8c9ba40db5d945b11b69b982c1bb9e3f3fac2bc369488f76b2383565d3fff921f9664c97637da9768812f615c68b13b52e
		
		Tag = c0875924c1c7987947deafd8780acf49
			Valid

	AAD = 50515253c0c1c2c3c4c5c6c7		
	key = 808182838485868788898a8b8c8d8e8f909192939495969798999a9b9c9d9e9f		
	nonce = 404142434445464748494a4b4c4d4e4f5051525354555657		

Based on the results, in which all scenarios that have been carried out show the same results between the comparison of the test parameter input with the output vector, the XChaCha20-Poly1305 algorithm has been created correctly.

B. Algorithm Execution Time Testing

Algorithm Execution Time Testing is used to find out how long it took the XChaCha20-Poly1305 algorithm to encrypt/decrypt the message and generate MAC codes. The algorithm execution test scenario is done by running the XChaCha20-Poly1305 program code using the test vector input which is implemented in the publisher and subscriber sections 30 times with 10 iterations. The execution time calculation will start when the key and message are received as algorithm input until the message is encrypted and the MAC code is generated.

The results of the algorithm execution time test is the average time of 1.7317851384480794 ms , maximum time of 6.967306137084961 ms, and minimum time of 0.98419189453125 ms in the publisher, and the average time of 0.99968973 ms , maximum time of 1.0013580322265625 ms, and minimum time of 0.9984970092773438 ms in the subscriber. This shows that there is an insignificant difference in execution time from using the XChaCha20-Poly1305 algorithm for the publisher and the subscriber.

C. System Performance Testing

System performance testing is conducted to compare system performance and network performance in algorithmic processing based on system memory usage, time and memory data integrity checking. The results of the memory usage test are obtained from 30 publish-subscribe with 10 iterations when the system uses the encryption algorithm and without the encryption algorithm on the test vector. The results of the memory usage test are shown in the table below

Table 3. Memory Usage

Memory	Publisher		Subscriber	
	Using XChaCha20-Poly1305	Without XChaCha20-Poly1305	Using XChaCha20-Poly1305	Without XChaCha20-Poly1305
Average (MB)	0.0451584	0.03496408462524414	0.0328704	0.0318464

This can be seen from the test results using the XChaCha20-Poly1305 algorithm memory which uses 0.01019431537 MB of memory from the publisher and 0.001024 MB of memory from the subscriber.

Time and memory tests on data privacy by encryption and decryption and data integrity checks, on the test vector, are obtained from published messages on concurrent use of 30, 60, 90, 120, and 150 publishers. The results of the time and memory checks for data integrity are shown in the table below.

Table 4. Time and Memory on Data Privacy by Encryption and Decryption and Data Integrity Checks

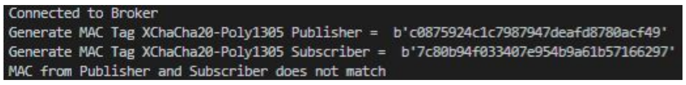
Number of Publishers	30	60	90	120	150
Time(s)	0.01190328598022461	0.023905038833618164	0.03590583801269531	0.0491888999938965	0.060906171798706055
Memory (MB)	0.1850368	0.2301952	0.2866496	0.3319744	0.3861728

It can be concluded from the table that the average increase in time when checking data integrity is 0.012250721454620361 seconds at each multiple of 30 publisher usage and the average increase in memory usage when checking data integrity is 0.050283999999999995 MB for each multiple of 30 publisher usage

D. Security Testing

Security testing is carried out to ensure the security of the system that is being built. The attack that will be used to test security is the Man in The Middle Attack (MITM) attack against a broker with the arp spoofing method that allows message change, insertion and substitution. The message that will be used is the test vector. This arp spoofing attack uses the Ettercap application.

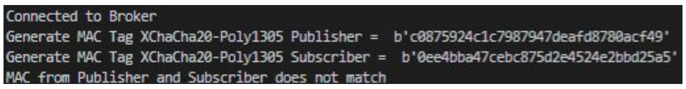
The result of the first attack, which is the message change attack, which involves changing the hex plaintext from "4c616469657320616e642047656e746c656d656e206f662074686520636c617373206f66202739393a204966204920636f756c64206f6666657220796f75206f6e6c79206f6e6520746970206 66f7220746865206675747572652c2073756e73637265656e20776f756c642062652069742**e**" from the test vector, to "4c616469657320616e642047656e746c656d656e206f662074686520636c617373206f66202739393a204966204920636f756c64206f6666657220796f75206f6e6c79206f6e6520746970206 66f7220746865206675747572652c2073756e73637265656e20776f756c642062652069742**f**", with the last hex is changed from "2**e**" to 2**f**", is as follows.



```
Connected to Broker
Generate MAC Tag XChaCha20-Poly1305 Publisher =  b'c0875924c1c7987947deafd8780acf49'
Generate MAC Tag XChaCha20-Poly1305 Subscriber =  b'7c80b94f033407e954b9a61b57166297'
MAC from Publisher and Subscriber does not match
```

Figure 5. Result from the message change attack

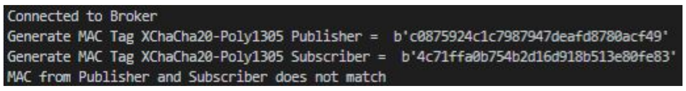
The result of the second attack, which is message insertion attack, so that the hex plaintext "4c616469657320616e642047656e746c656d656e206f662074686520636c617373206f66202739393a204966204920636f756c64206f6666657220796f75206f6e6c79206f6e6520746970206 66f7220746865206675747572652c2073756e73637265656e20776f756c642062652069742e" becomes "4c616469657320616e642047656e746c656d656e206f662074686520636c617373206f66202739393a204966204920636f756c64206f6666657220796f75206f6e6c79206f6e6520746970206 66f7220746865206675747572652c2073756e73637265656e20776f756c642062652069742e**4c616469657320616e642047656e**", with an insertion of "**4c616469657320616e642047656e**", is as follows.



```
Connected to Broker
Generate MAC Tag XChaCha20-Poly1305 Publisher =  b'c0875924c1c7987947deafd8780acf49'
Generate MAC Tag XChaCha20-Poly1305 Subscriber =  b'0ee4bba47cebc875d2e4524e2bbd25a5'
MAC from Publisher and Subscriber does not match
```

Figure 6. Result from the message insertion attack

The result of the third attack, which is message substitution, so that the hex plaintext "4c616469657320616e642047656e746c656d656e206f662074686520636c617373206f66202739393a204966204920636f756c64206f6666657220796f75206f6e6c79206f6e6520746970206 66f7220746865206675747572652c2073756e73637265656e20776f756c642062652069742e" is substituted using the ROT13 algorithm, is as follows.



```
Connected to Broker
Generate MAC Tag XChaCha20-Poly1305 Publisher =  b'c0875924c1c7987947deafd8780acf49'
Generate MAC Tag XChaCha20-Poly1305 Subscriber =  b'4c71ffa0b754b2d16d918b513e80fe83'
MAC from Publisher and Subscriber does not match
```

Figure 7. Result from the message substitution attack

It can be seen from the results that the subscriber has received the message from the broker and the subscriber displays that the MAC from the publisher and the subscriber does not match because the subscriber has a data integrity check by comparing the value of the MAC from the publisher and the subscriber.

V. Conclusion

The subscriber is able to identify message integrity by calculating the MAC value based on the received message value. MAC can be calculated by inputting key, integer, nonce and message received. If the MAC output created by the subscriber is the same as the MAC generated by the publisher, then the integrity of the message received from the broker can be guaranteed and has not undergone changes, insertions and/or substitutions during message transmission by the broker.

The XChaCha20-Poly1305 algorithm, when used for encryption / decryption and data integrity checking in the MQTT protocol, has an average time of 1.7317851384480794 ms to generate MAC for the publisher and the subscriber of 0.99968973 ms. Meanwhile, the increase in memory usage when the XChaCha20-Poly1305 algorithm is applied to the system is 0.01019431537 MB for the publisher and 0.001024 MB for the subscriber. The time performed for data integrity checks performed by subscribers is increased by 0.012250721454620361 seconds for every 30 publishers, and memory usage is increased by 0.050283999999999995 MB for every 30 publishers.

VI. Acknowledgment

The author is grateful to The One Almighty God for the blessing that has been given so that this paper can be finished successfully. The author is also grateful to Dr. Ir. Rinaldi Munir, MT., our lecturer in IF4020 Cryptography - 1st Semester of Academic Year 2020/2021, for the knowledge and the time which are shared with the college students attending the course. The author is also grateful for the support from the author's family and friends.

References

[1] Rouse, M. (2020, February 11). What is IoT (Internet of Things) and How Does it Work? Retrieved December 21, 2020, from https://internetofthingsagenda.techtarget.com/definition/Internet-of-Things-IoT

[2] Yuan, M. (2020, January 7). What is MQTT? Why use MQTT? Retrieved December 21, 2020, from https://developer.ibm.com/technologies/messaging/articles/iot-mqtt-why-good-for-iot/

[3] S. Andy, B. Rahardjo and B. Hanindhito, "Attack scenarios and security analysis of MQTT communication protocol in IoT system," 2017 4th International Conference on Electrical Engineering, Computer Science and Informatics (EECSI), Yogyakarta, 2017, pp. 1-6, doi: 10.1109/EECSI.2017.8239179.

[4] Buchanan, B. (2018, December 28). AES Is Great ... But We Need A Fall-back: Meet ChaCha and Poly1305. Retrieved December 21, 2020, from https://medium.com/asecuritysite-when-bob-met-alice/aes-is-great-but-we-need-a-fall-back-meet-chacha-and-poly1305-76ee0ee61895

[5] Popov, A. (2015, February). Prohibiting RC4 Cipher Suites. Retrieved December 21, 2020, from https://tools.ietf.org/html/rfc7465

[6] MQTT Version 5.0. Edited by Andrew Banks, Ed Briggs, Ken Borgendale, and Rahul Gupta. 07 March 2019. OASIS Standard.

https://docs.oasis-open.org/mqtt/mqtt/v5.0/os/mqtt-v5.0-os.html. Latest version: https://docs.oasis-open.org/mqtt/mqtt/v5.0/mqtt-v5.0.html.

[7] McGrew, D. (2008, January). An Interface and Algorithms for Authenticated Encryption. Retrieved December 17, 2020, from https://tools.ietf.org/html/rfc5116

[8] Tutorialspoint. (n.d.). Message Authentication. Retrieved December 17, 2020, from https://www.tutorialspoint.com/cryptography/message_authentication.htm

[9] Nir, Y., & Langley, A. (2015, May). ChaCha20 and Poly1305 for IETF Protocols. Retrieved December 18, 2020, from https://tools.ietf.org/html/rfc7539

[10] Bernstein, D. J. (n.d.). ChaCha, a variant of Salsa20. Retrieved December 18, 2020, from https://cr.yp.to/chacha/chacha-20080128.pdf

[11] Neves, S. (2009, October 7). Faster ChaCha implementations for Intel processors. Retrieved December 18, 2020, from https://web.archive.org/web/20161128095454/https://eden.dei.uc.pt/~sneves/chacha/chacha.html

[12] Arciszewski, S. (2018, December 18). XChaCha: EXtended-nonce ChaCha and AEAD_XChaCha20_Poly1305. Retrieved December 18, 2020, from https://tools.ietf.org/html/draft-arciszewski-xchacha-03

PERNYATAAN

Dengan ini saya menyatakan bahwa makalah yang saya tulis ini adalah tulisan saya sendiri, bukan saduran, atau terjemahan dari makalah orang lain, dan bukan plagiasi.

Bandung, 19 Desember 2020

[illegible]

Ignatius Timothy Manullang
13517044