Chacha20-Blake2b-512 Authenticated Encryption

Course: Cryptography

Group-20

Isha Aggarwal (S20180010067), IIITS
Nitin Chauhan (S2018001119), IIITS

Abstract

This report has been written in reference to the cryptography course project, Monsoon semester 2021. It demonstrates the Chacha20 cipher, Blake2b-512 HMAC, and their combined authenticated encryption mode. We elaborate on the significance of authentication and briefly discuss the implementation of Chacha20-Blake2b-512, and finally present the obtained results along with conclusion.

Index Terms

Chacha20, Blake2b-512, cipher, HMAC, authenticated encryption

I. INTRODUCTION

In any communication system, using encryption alone only provides data confidentiality between sender and receiver. The message can easily be forged by the adversary, without the receiver getting to know about the forgery and he could end up reading the wrong message. In order to verify if the message is authentic and has not been modified by the adversary, we can use message authentication codes (MAC) along with encryption rather than simple encryption.

MAC produces a tag based on the message and a secret key, which is appended to the message by the sender. The receiver again computes a tag on the received message using the shared secret key and verifies if this tag is the same as the tag he has received with the message. If both are same, then he accepts and reads the message, otherwise he gets to know that the message has been forged by the adversary.

Cryptographic hash functions, when used with a key, can be used to produce a tag in a way similar to MAC for message authentication ; this is commonly known as HMAC (Hash based MAC or keyed-hash).
In this paper, we use the Chacha20 cipher for message encryption-decryption, Blake2b-512 hash function as HMAC for message authentication, and combine both to provide authentication encryption (AE). This ensures data confidentiality, integrity and authentication.

In Section II, we present the goals achieved in this project. Then Section IV discusses the Chacha20 and Blake2b-512 algorithms and their implementations in brief. Subsequent section compares Blake2b-512 to other hash functions and also discusses the applications of Chacha20 and Blake2b-512 in short. In Section VI we present the results we have obtained and finally, we summarize our work in Section VII.

II. GOALS

We achieve the following goals in this project:

- Encrypting given plain text to cipher text and decrypting the cipher text back to plain text using ChaCha20 stream cipher. This provides confidentiality in the communication.

- Generating digest using Blake2b-512 HMAC based on the input message and the input key, and the digest verification by the receiver. Receiver would be able to identify if the message has been forged by the adversary, by the digest verification mechanism. This allows for integrity in the communication and helps to authenticate the received message.

- Combining Chacha20 and Blake2b-512 to perform authenticated encryption. This provides both confidentiality and integrity in the communication and helps to authenticate the message.

- Providing a graphical user interface for easy use and nice demonstration.

III. WORK CONTRIBUTION

- Isha Aggarwal:
    - Implementation of Chacha20 cipher
    - Implementation of Blake2b-512 HMAC

- Nitin Chauhan:
    - Implementation of Chacha20 cipher
    - Building GUI

IV. ALGORITHM AND IMPLEMENTATION

A. Chacha20 cipher:

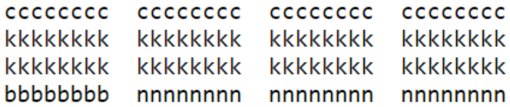
Chacha20 is a 20 round stream cipher (symmetric key cryptosystem) designed by D.J. Bernstein. It is highly recommended by the designer for typical cryptographic applications as it is consistently faster than AES.

The encryption algorithm is as follows:

Inputs to chacha20 are:

- Plaintext: Any arbitrary length
- Key: A 256-bit key, treated as a concatenation of eight 32-bit little-endian integers
- Counter: A 32-bit block count parameter (can be any number, but usually set to 0 or 1), treated as a 32-bit little-endian integer
- Nonce: A 96-bit nonce, treated as a concatenation of three 32-bit little-endian integers.

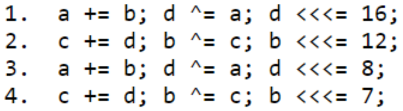
Chacha20 state is a vector of 16 unsigned 32-bit integers which is initialized from the key,nonce and the block counter as follows:



```
cccccccc  cccccccc  cccccccc  cccccccc
kkkkkkkk  kkkkkkkk  kkkkkkkk  kkkkkkkk
kkkkkkkk  kkkkkkkk  kkkkkkkk  kkkkkkkk
bbbbbbbb  nnnnnnnn  nnnnnnnn  nnnnnnnn
```

- c: constants Here first 4 words i.e. 0-3 are constants given by 0x61707865, 0x3320646e, 0x79622d32, 0x6b206574.
- k: key Here words 4-11 are taken from the 256-bit key. Bytes are read in little-endian order, with 4 byte chunks.
- b: block counter Word 12 is a 32-bit block count
- n: nonce Words 13-15 are from nonce, read in little-endian order in 4 bytes chunks. First 32 bits of nonce form the 13th word, while last 32 bits of nonce form the 15th word.

Chacha20 consists of 20 full rounds, each full round has 4 quarter rounds- which operate on diagonal elements of or column elements of chacha state vector alternately. So, in total there are 80 quarter rounds. Each quarter round deals with 4 pre-specified elements (32 bit unsigned integers) of the chacha state; the operations in a quarter round are given as follows:



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

Here addition is modulo 2**32, and ’<<< n’ denotes circular left shift of n bits.

Chacha20 block function - transforms a ChaCha state by running multiple (80) quarter rounds i.e performing 20 full rounds on a state. At the end of 20 rounds, the original input state is added (modulo 2**32) to the transformed state and the words of the resulting state are serialized (in little-endian order) to produce a keystream block.

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

Overall chacha20 encryption is as follows.

The plain text is divided into blocks of size 512 bits and xored with the keystream blocks produced by running chacha block function on chacha state. The initialization of the chacha state is already described above. The block counter is increased by 1 during successive calls. There is no need for the text to be a multiple of 512 bits. If there is an extra key stream from last block, it is discarded.

```
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
```

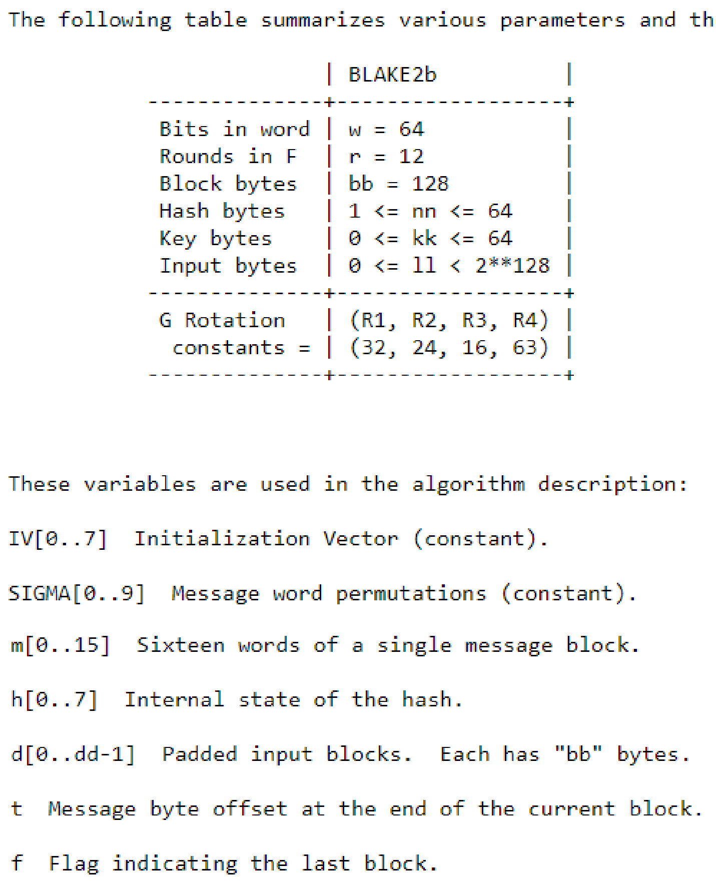
The output is encrypted/cipher text of the same length as that of input plain text. Decryption is done in the same way.

B. Blake2b-512 HMAC

Blake2b is a cryptographic hash function designed by Jean-Philippe Aumasson, released on January 13, 2013. It does not require a special ”HMAC " (Hashed Message Authentication Code) construction for keyed message authentication as it has a built-in keying mechanism. It produces a digest based on the message and the key, which is used to verify data integrity and the authenticity of the message. Receiver can make out if the message has been forged by the adversary.

The Blake-2b-512 algorithm is as follows:

The range of various parameters and the notations used in the algorithm are described below:



The following table summarizes various parameters and their ranges:

	BLAKE2b
Bits in word	w = 64
Rounds in F	r = 12
Block bytes	bb = 128
Hash bytes	1 <= nn <= 64
Key bytes	0 <= kk <= 64
Input bytes	0 <= ll < 2**128
G Rotation constants =	(R1, R2, R3, R4) (32, 24, 16, 63)

These variables are used in the algorithm description:

IV[0..7] Initialization Vector (constant).

SIGMA[0..9] Message word permutations (constant).

m[0..15] Sixteen words of a single message block.

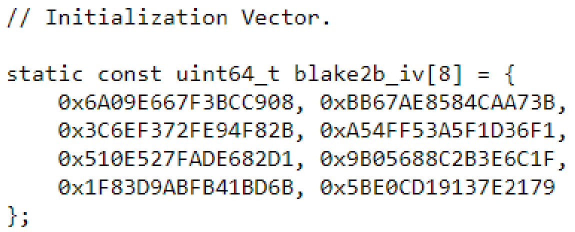
h[0..7] Internal state of the hash.

d[0..dd-1] Padded input blocks. Each has "bb" bytes.

t Message byte offset at the end of the current block.

f Flag indicating the last block.

The IV (initialization vector) constants are:



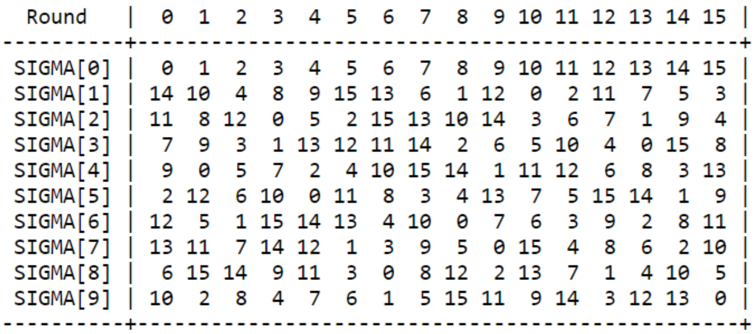
```
// Initialization Vector.

static const uint64_t blake2b_iv[8] = {
    0x6A09E667F3BCC908, 0xBB67AE8584CAA73B,
    0x3C6EF372FE94F82B, 0xA54FF53A5F1D36F1,
    0x510E527FADE682D1, 0x9B05688C2B3E6C1F,
    0x1F83D9ABFB41BD6B, 0x5BE0CD19137E2179
};
```

Blake2b takes in an arbitrary length message (ll) and variable length key (0 <= kk <= 64) and produces a digest tag (1 <= nn <= 64), the number of bytes the user wants in the hash could be set, the maximum is 64. It operates on 64-bit words. The key and the message form data blocks (more details-shortly), each data block has 16 words, i.e. 128 (=bb) bytes.

Note: Little-endian order is followed for reading bytes.

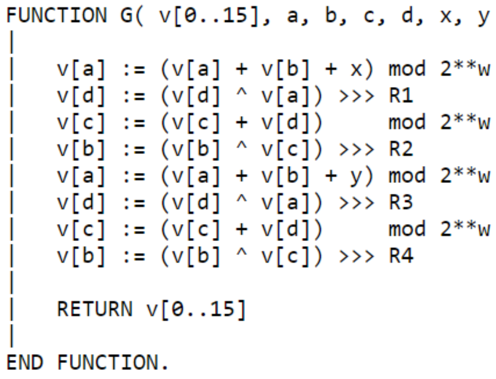
Blake2b consists of 12 rounds, permutations of words in each round is given by sigma:

sigma[round index] = sigma



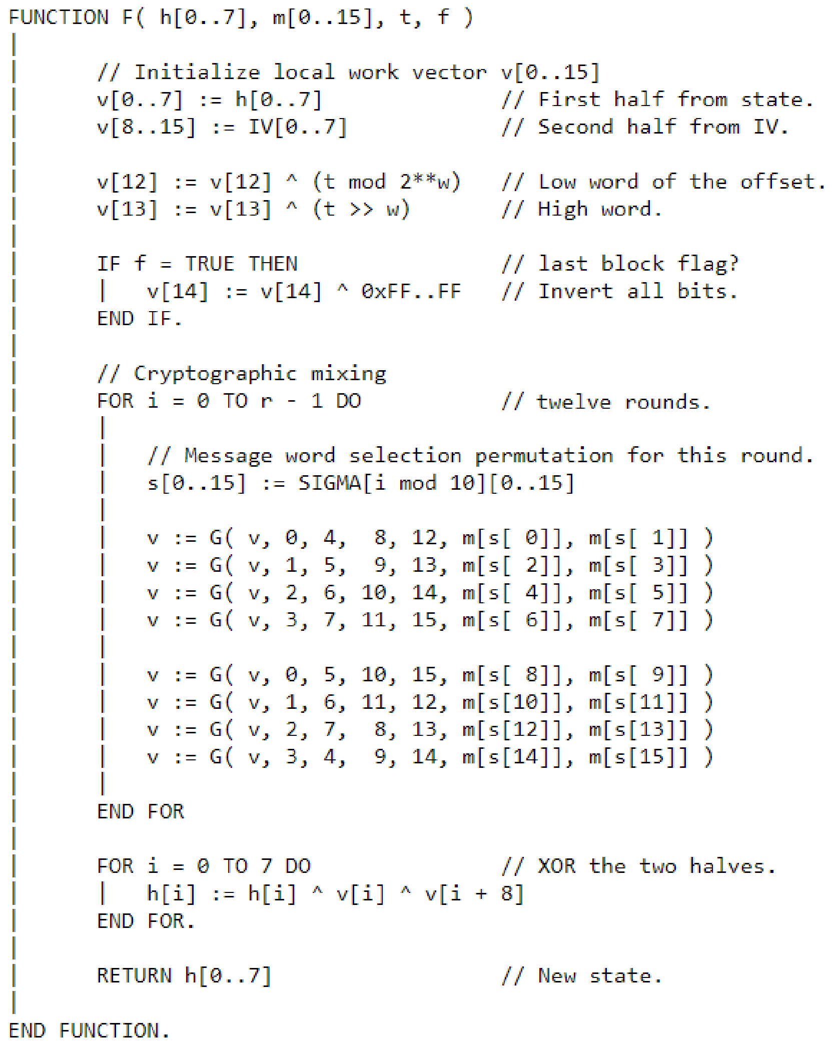
Round	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
SIGMA[0]	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
SIGMA[1]	14	10	4	8	9	15	13	6	1	12	0	2	11	7	5	3
SIGMA[2]	11	8	12	0	5	2	15	13	10	14	3	6	7	1	9	4
SIGMA[3]	7	9	3	1	13	12	11	14	2	6	5	10	4	0	15	8
SIGMA[4]	9	0	5	7	2	4	10	15	14	1	11	12	6	8	3	13
SIGMA[5]	2	12	6	10	0	11	8	3	4	13	7	5	15	14	1	9
SIGMA[6]	12	5	1	15	14	13	4	10	0	7	6	3	9	2	8	11
SIGMA[7]	13	11	7	14	12	1	3	9	5	0	15	4	8	6	2	10
SIGMA[8]	6	15	14	9	11	3	0	8	12	2	13	7	1	4	10	5
SIGMA[9]	10	2	8	4	7	6	1	5	15	11	9	14	3	12	13	0

Mixing function G modifies the working vector v by mixing the 2 input words x and y into 4 words of the working vector. Rotation constants G1, G2, G3, G4 are already specified above.



```
FUNCTION G( v[0..15], a, b, c, d, x, y )
|
|   v[a] := (v[a] + v[b] + x) mod 2**w
|   v[d] := (v[d] ^ v[a]) >>> R1
|   v[c] := (v[c] + v[d])     mod 2**w
|   v[b] := (v[b] ^ v[c]) >>> R2
|   v[a] := (v[a] + v[b] + y) mod 2**w
|   v[d] := (v[d] ^ v[a]) >>> R3
|   v[c] := (v[c] + v[d])     mod 2**w
|   v[b] := (v[b] ^ v[c]) >>> R4
|
|   RETURN v[0..15]
|
END FUNCTION.
```

Compression function F takes in hash state vector h and a data block of size bb bytes (i.e m consisting of 16 words, each of 64 bits) and manipulates the hash state vector h by performing r(= 12) rounds. Here v[0...15] is local working vector, input parameter t is 2w bit offset which indicates the number of bytes that haven been processed and f is a flag which indicates if the data block is a final block:



```
FUNCTION F( h[0..7], m[0..15], t, f )
|
|      // Initialize local work vector v[0..15]
|      v[0..7] := h[0..7]              // First half from state.
|      v[8..15] := IV[0..7]            // Second half from IV.
|
|      v[12] := v[12] ^ (t mod 2**w)   // Low word of the offset.
|      v[13] := v[13] ^ (t >> w)       // High word.
|
|      IF f = TRUE THEN                // last block flag?
|      |   v[14] := v[14] ^ 0xFF..FF   // Invert all bits.
|      END IF.
|
|      // Cryptographic mixing
|      FOR i = 0 TO r - 1 DO           // twelve rounds.
|      |
|      |   // Message word selection permutation for this round.
|      |   s[0..15] := SIGMA[i mod 10][0..15]
|      |
|      |   v := G( v, 0, 4,  8, 12, m[s[ 0]], m[s[ 1]] )
|      |   v := G( v, 1, 5,  9, 13, m[s[ 2]], m[s[ 3]] )
|      |   v := G( v, 2, 6, 10, 14, m[s[ 4]], m[s[ 5]] )
|      |   v := G( v, 3, 7, 11, 15, m[s[ 6]], m[s[ 7]] )
|      |
|      |   v := G( v, 0, 5, 10, 15, m[s[ 8]], m[s[ 9]] )
|      |   v := G( v, 1, 6, 11, 12, m[s[10]], m[s[11]] )
|      |   v := G( v, 2, 7,  8, 13, m[s[12]], m[s[13]] )
|      |   v := G( v, 3, 4,  9, 14, m[s[14]], m[s[15]] )
|      |
|      END FOR
|
|      FOR i = 0 TO 7 DO               // XOR the two halves.
|      |   h[i] := h[i] ^ v[i] ^ v[i + 8]
|      END FOR.
|
|      RETURN h[0..7]                  // New state.
|
END FUNCTION.
```

The overall algorithm is as follows:

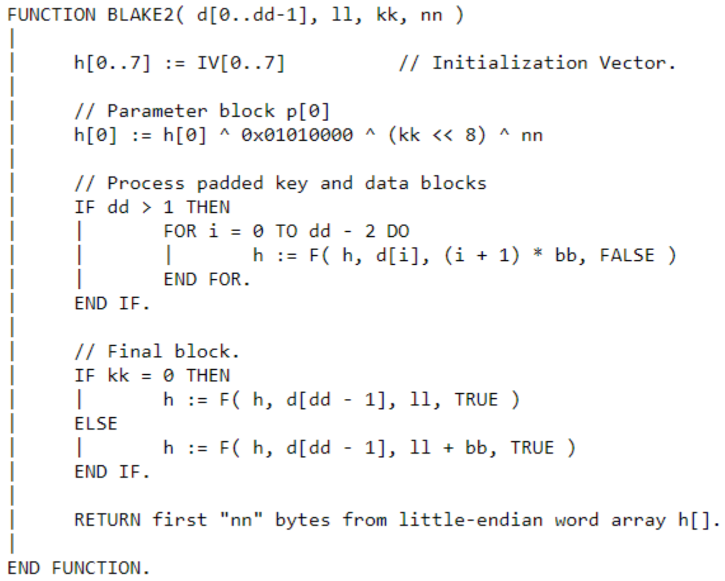
Key and message input are split and padded into ”dd” message blocks d[0..dd-1], each consisting of 16 words (or ”bb” bytes).

In case a secret key is used i.e kk > 0 (and it always is in case of HMAC), then it is padded with zero bytes to form d[0]. Else, d[0] is simply obtained from the message. The final data block d[dd-1] is also padded with zero bytes to get ’bb’ bytes.

Therefore, the total number of data blocks: dd = ceil(kk/bb) + ceil(ll/bb) Note: If both kk = 0 and ll = 0 (i.e. unkeyed empty message), then we still set dd = 1, and d[0] contains all zeros.

The data blocks are then processed to get ’nn’ byte final hash digest.

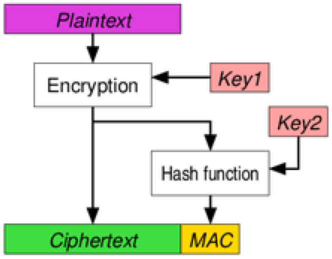
Basically, the compression function F is called for each data block (with other arguments such as: byte offset- the number of bytes that have been processed, last block indicator etc.) and the hash state is modified as a result of the call. Finally, nn bytes from the hash are returned after the last call.



```
FUNCTION BLAKE2( d[0..dd-1], ll, kk, nn )
|
|     h[0..7] := IV[0..7]          // Initialization Vector.
|
|     // Parameter block p[0]
|     h[0] := h[0] ^ 0x01010000 ^ (kk << 8) ^ nn
|
|     // Process padded key and data blocks
|     IF dd > 1 THEN
|     |       FOR i = 0 TO dd - 2 DO
|     |       |       h := F( h, d[i], (i + 1) * bb, FALSE )
|     |       END FOR.
|     END IF.
|
|     // Final block.
|     IF kk = 0 THEN
|     |       h := F( h, d[dd - 1], ll, TRUE )
|     ELSE
|     |       h := F( h, d[dd - 1], ll + bb, TRUE )
|     END IF.
|
|     RETURN first "nn" bytes from little-endian word array h[].
|
END FUNCTION.
```

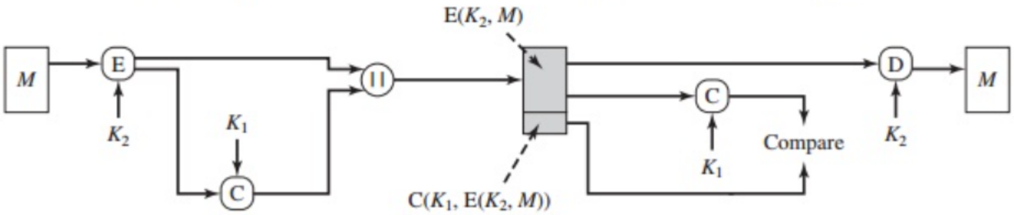
C. Chacha20-Blake2b-512 authenticated encryption

ChaCha20 and Blake2b-512 in combined mode results in an authenticated encryption scheme. We use encrypt-then-MAC mode.



This helps to provide confidentiality, integrity in the communication and helps to identify if the message is authentic.

AE is shown below:



The input plain text message is first encrypted to produce a cipher text and then a tag is computed on the cipher text using MAC; this tag is appended to the cipher text message. When the sender receives the message, he again computes the tag on the cipher text. If both the tags match, then he decrypts the cipher text and reads the message, else he gets to know that the message is not authentic, it has been modified by an adversary.

D. Implementation

We have implemented both the algorithms from scratch in C++17. We provide both command-line and graphical interface to the users. The details on how to use the setup are in README.md file attached with the project code. Github link to repo: https://github.com/aggarwal2000/ChaCha20-Blake2b-Cryptography/tree/Add_GUI

V. COMPARISON AND APPLICATIONS

- Chacha20 stream cipher is much faster than AES. It is an improved variant of Salsa cipher.

- The BLAKE2 hash function is more secure than SHA-3 and faster than MD5, SHA-1, and SHA-2. BLAKE2 is commonly used in cloud storage, intrusion detection, and version control systems. BLAKE2 algorithms are 32% faster than BLAKE hash algorithms. There are two different forms of the Blake2: BLAKE2b (optimized for 64-bit performance) and BLAKE2s (optimized for smaller software architectures). In this project, we have implemented Blake2b.

- Chacha20 cipher, Blake2b HMAC are used in many software projects.For instance: Chacha20: Open SSH, Google's SPDY; Blake2b HMAC: Open SSL, WolfSSL etc.

VI. RESULTS

This section presents the results (as stated in the goals section):

1) Chacha20 encryption (Without authentication):

Inputs

Plaintext:
Ladies and Gentlemen of the class of '99: If I could offer you only one tip for the future, sunscreen would be it.

Chacha20 key: 00:01:01:03:04:05:06:08:09:0a:0b:0c:0d:0e:10:11:12:13:14:15:16:17:18:19:1a:1b:1c:1d:1e:1f

Chacha20 counter:
1

Chacha20 nonce:
00:00:00:00:00:00:00:4a:00:00:00:00

Outputs

A. CASE: No forging by adversary

Cipher text generated by Alice:
6e:2e:35:9a:25:68:f9:80:41:ba:07:28:dd:0d:69:81:e9:7e:7a:ec:1d:43:60:c2:0a:27:af:cc:fd:9f:ae:0b:f9:1b:65:c5:52:47:3
3:ab:8f:59:3d:ab:cd:62:b3:57:16:39:d6:24:e6:51:52:ab:8f:53:0c:35:9f:08:61:d8:07:ca:0d:bf:50:0d:6a:61:56:a3:8e:08:8
a:22:b6:5e:52:bc:51:4d:16:cc:f8:06:81:8c:e9:1a:b7:79:37:36:5a:f9:0b:bf:74:a3:5b:e6:b4:0b:8e:ed:f2:78:5e:42:87:4d

Cipher text reaching Bob:
6e:2e:35:9a:25:68:f9:80:41:ba:07:28:dd:0d:69:81:e9:7e:7a:ec:1d:43:60:c2:0a:27:af:cc:fd:9f:ae:0b:f9:1b:65:c5:52:47:3
3:ab:8f:59:3d:ab:cd:62:b3:57:16:39:d6:24:e6:51:52:ab:8f:53:0c:35:9f:08:61:d8:07:ca:0d:bf:50:0d:6a:61:56:a3:8e:08:8
a:22:b6:5e:52:bc:51:4d:16:cc:f8:06:81:8c:e9:1a:b7:79:37:36:5a:f9:0b:bf:74:a3:5b:e6:b4:0b:8e:ed:f2:78:5e:42:87:4d

Decrypted text:
Ladies and Gentlemen of the class of '99: If I could offer you only one tip for the future, sunscreen would be it.

B. CASE: Message forged by adversary

Cipher text generated by Alice:

```
6e:2e:35:9a:25:68:f9:80:41:ba:07:28:dd:0d:69:81:e9:7e:7a:ec:1d:43:60:c2:0a:27:af:cc:fd:9f:ae:0b:f9:1b:65:c5:52:47:3
3:ab:8f:59:3d:ab:cd:62:b3:57:16:39:d6:24:e6:51:52:ab:8f:53:0c:35:9f:08:61:d8:07:ca:0d:bf:50:0d:6a:61:56:a3:8e:08:8
a:22:b6:5e:52:bc:51:4d:16:cc:f8:06:81:8c:e9:1a:b7:79:37:36:5a:f9:0b:bf:74:a3:5b:e6:b4:0b:8e:ed:f2:78:5e:42:87:4d
```

Cipher text reaching Bob:

```
6e:2e:35:9a:25:68:f9:80:41:ba:07:28:dd:0d:69:81:e9:7e:7a:ec:1d:43:60:c2:0a:27:af:cc:fd:9f:ae:0b:f9:1b:65:c5:52:47:3
3:ab:8f:59:3d:ab:cd:62:b3:57:16:39:d6:24:e6:51:52:ab:8f:53:0c:35:9f:08:61:d8:07:ca:0d:bf:50:0d:6a:61:56:a3:8e:08:8
a:22:b6:5e:52:bc:51:4d:16:cc:f8:06:81:8c:e9:1a:b7:79:37:36:5a:06:0b:bf:74:a3:5b:e6:b4:0b:8e:ed:f2:78:5e:42:87:4d
```

Decrypted text (as read by Bob):
Ladies and Gentlemen of the class of '99: If I could offer you only one tip for the future, sunsc�een would be it.

So, in this case, the receiver (Bob) can not find out if the message has been forged by the adversary. He decrypts and reads the wrong message.

Therefore, only confidentiality exists in such communication.

Note: Above are the results produced by our implementation of Chacha20. Correctness of our implementation is tested by comparing the above results to those mentioned in the RFC.

2) Blake-2b HMAC Demonstration:

Producing a hash digest from any given message:

Inputs

Message: << i.e. empty message>>

Blake 2b key: << i.e empty key>>

Blake 2b digest size: 64

Output

Hash digest:

```
78:6a:02:f7:42:01:59:03:c6:c6:fd:85:25:52:d2:72:91:2f:47:40:e1:58:47:61:8a:86
:e2:17:f7:1f:54:19:d2:5e:10:31:af:ee:58:53:13:89:64:44:93:4e:b0:4b:90:3a:68:5b:14:48:b7:55:d5:6f:70:1a:fe:9b:e2:ce
```

Inputs

Message:
Say hi to crypto!

Blake 2b key:
crypto

Blake 2b digest size:
64

Output

Hash digest:

```
5b:9f:20:4e:ec:a4:bd:e4:8e:f0:bb:8f:94:8d:93:bc:6d:f3:a8:17:2b:d4:e0:29:81:2b:31:9c:51:9a:11:87:e9:be:09:43:d7:a9:
32:3b:90:1d:f4:5c:71:6a:61:d9:7f:f9:d8:5a:98:b0:07:07:33:95:d5:30:a0:da:cb:67
```

Inputs

Message:
Say hi to arypto!

Blake 2b key:
crypto

Blake 2b digest size: 64

Output

Hash digest:
82:41:57:f8:fd:f8:2e:1a:c8:12:06:65:41:e5:21:83:9f:d1:7c:90:17:83:17:e3:8a:41:ae:02:a2:c1:67:55:84:e7:4e:96:14:60:
58:ff:14:21:b5:4b:37:fe:0d:68:1e:8c:60:96:f1:c2:34:ab:fb:f0:c2:25:27:95:11:81

It could be observed that even for an unkeyed empty message, hash could be produced.
Also, even for a very little change in the message i.e from “Say hi to crypto!” to
“Say hi to arypto!”, there is considerable change in the hash digest.

Note: Above are the results produced by our implementation of Blake2b-512. Correctness of our implementation is tested by comparing the above results to those produced by: https://www.toolkitbay.com/tkb/tool/BLAKE2b_512

3) Chacha20-Blake2b Authenticated Encryption:

Inputs

Plaintext:
Hello Bob, how are you?

Chacha20 key:
00:01:01:03:04:05:06:08:09:0a:0b:0c:0d:0e:10:11:12:13:14:15:16:17:18:19:1a:1b:1c:1d:1e:1f

Chacha20 counter:
1

Chacha20 nonce:
00:00:00:00:00:00:00:4a:00:00:00:00

Blake2b-512 key:
crypto

Blake2b-512 digest size:
64

Outputs

A. CASE: No forging by adversary

Cipher text generated by Alice:
6a:2a:3d:9f:2f:3b:9b:8e:4d:f2:07:07:d7:14:3d:8c:fe:76:3f:fb:52:59:39

Hash generated by Alice (on cipher text):
4e:90:67:1b:02:36:bf:df:ae:af:4d:61:c6:b5:aa:ce:7a:9f:87:c5:4a:51:b7:96:f3:83:f1:ef:80:54:1e:c1:3d:3a:c0:11:3e:40:75:
82:92:bc:1f:f6:17:e5:a0:38:a0:fd:f1:2c:e3:04

```
:3d:e1:57:64:99:c7:e2:39:ae:1e
```

Cipher text reaching Bob:

```
6a:2a:3d:9f:2f:3b:9b:8e:4d:f2:07:07:d7:14:3d:8c:fe:76:3f:fb:52:59:39
```

Hash generated by Bob (on cipher text):

```
4e:90:67:1b:02:36:bf:df:ae:af:4d:61:c6:b5:aa:ce:7a:9f:87:c5:4a:51:b7:96:f3:83:f1:ef:80:54:1e:c1:3d:3a:c0:11:3e:40:75:
82:92:bc:1f:f6:17:e5:a0:38:a0:fd:f1:2c:e3:04
:3d:e1:57:64:99:c7:e2:39:ae:1e
```

Tag verification result:
Verified over hash tag

Decrypted text:
Hello Bob, how are you?

B. CASE: Message is forged by the adversary

Cipher text generated by Alice:

```
6a:2a:3d:9f:2f:3b:9b:8e:4d:f2:07:07:d7:14:3d:8c:fe:76:3f:fb:52:59:39
```

Hash generated by Alice (on cipher text):

```
4e:90:67:1b:02:36:bf:df:ae:af:4d:61:c6:b5:aa:ce:7a:9f:87:c5:4a:51:b7:96:f3:83:f1:ef:80:54:1e:c1:3d:3a:c0:11:3e:40:75:
82:92:bc:1f:f6:17:e5:a0:38:a0:fd:f1:2c:e3:04
:3d:e1:57:64:99:c7:e2:39:ae:1e
```

Cipher text reaching Bob:

```
6a:2a:3d:9f:2f:3b:9b:8e:4d:f2:07:f8:d7:14:3d:8c:fe:76:3f:fb:52:59:39
```

Hash generated by Bob (on cipher text):

```
43:52:30:f0:5a:07:1b:99:ed:27:8d:8b:96:3b:8a:1f:ea:6d:73:05:4a:c2:de:0a:de:c3:c5:71:a5:46:50:b3:0f:78:e6:d0:39:b4:
1e:44:2d:0b:f0:9e:cc:6b:00:c7:48:2b:d1:73:30:32:b0:d7:e8:f9:83:46:15:9e:b1:71
```

Tag verification result:
Tag mismatch, message has been forged by adversary

Decrypted text:
Message has been forged by the adversary(found by HMAC- hash tag verification), no use of decryption.

Thus, there is confidentiality, integrity and authentication in such communication. The receiver can make out if the message has been forged by tag verification mechanism, and refuse to decrypt the message in case of tag mismatch.

VII. CONCLUSION

In our cryptography project, we demonstrated the significance of authentication, implemented Chacha20 stream cipher, Blake2b-512 HMAC, and combined both to provide Chacha20-Blake2b-512 authenticated encryption. We demonstrated Blake2b HMAC, presented the results for cases when authentication is turned on or off and considered what happens on eavesdropping by the adversary.

REFERENCES

[1] Bernstein D., ”ChaCha, a variant of Salsa20”
[2] RFC 7539
[3] Jean-Philippe Aumasson et al. “BLAKE2: Simpler, Smaller, Fast as MD5: Springer paper, Conference: 11th international conference of Applied Cryptography and Network Security”
[4] RFC 7693