

Palestine
An-Najah National University
Faculty of Information Technology
Network and Information Security



دولة فلسطين
جامعة النجاح الوطنية
كلية تكنولوجيا المعلومات
قسم شبكات وامن المعلومات

Experiment#3

Hash Function and Message Authentication Code

Dyaa Al-dein Ashraf Tummazeh

11924899

Manar Eyad Harb

11924470

Yara Mohammad Sholi

11924207

Supervisor:

Dr. Amjad Hawash

Abstract

In this experiment, the goal was practicing on generating hash values and MAC for files using the openssl tool.

Introduction

A hash function is a mathematical algorithm that maps data of arbitrary size (often called the "message") to a bit array of a fixed size (the "hash value", "hash", or "message digest"). It is a one-way function, that is, a function for which it is practically infeasible to invert or reverse the computation. And there are several algorithms of hashes, SHA1, SHA256, MD5.

The purpose of using hash functions is to protect the data integrity, they are like fingerprints, a unique numerical value is produced and assigned for a specific file, if the data gets modified then the hash value will directly be changed, which means the data isn't the original one.

A message authentication code (MAC) is a short piece of information used for authenticating a message In other words, to confirm that the message came from the stated sender (its authenticity) and has not been changed. The MAC value protects a message's data integrity, as well as its authenticity, by allowing verified (who also possess the secret key) to detect any changes to the message content.

Next is showing how to generate hash and MAC values using OpenSSL commands.

Procedures

I. Message digest generation:

- a. Hashing is used to prove authentication and to protect your messages from changes.
- b. It's a tag that is used to authenticate the origin of the message and protect it from changes.
- c. Text file name Input.txt was created and stored arbitrary data inside it.
- d. Hash value of the text file was generated with the MD5 algorithm, the hash code size is 128 BIT.

openssl dgst -md5 -hex index.txt

- e. Hash value of the text file was generated with the SHA1 algorithm, the hash code size is 160 BIT.

openssl dgst -sha1 -hex index.txt

- f. Hash value of the text file was generated with the SHA256 algorithm, the hash code size is 256 BIT.

openssl dgst -sha256 -hex index.txt

II. Message Authentication Code (MAC) & Keyed Hash:

As talked in introduction, message authentication code (MAC) is a short piece of information used for authenticating a message. In other words, to confirm that the message came from the stated sender (its authenticity) and has not been changed. The MAC value protects a message's data integrity, as well as its authenticity, by allowing verified (who also possess the secret key) to detect any changes to the message content.

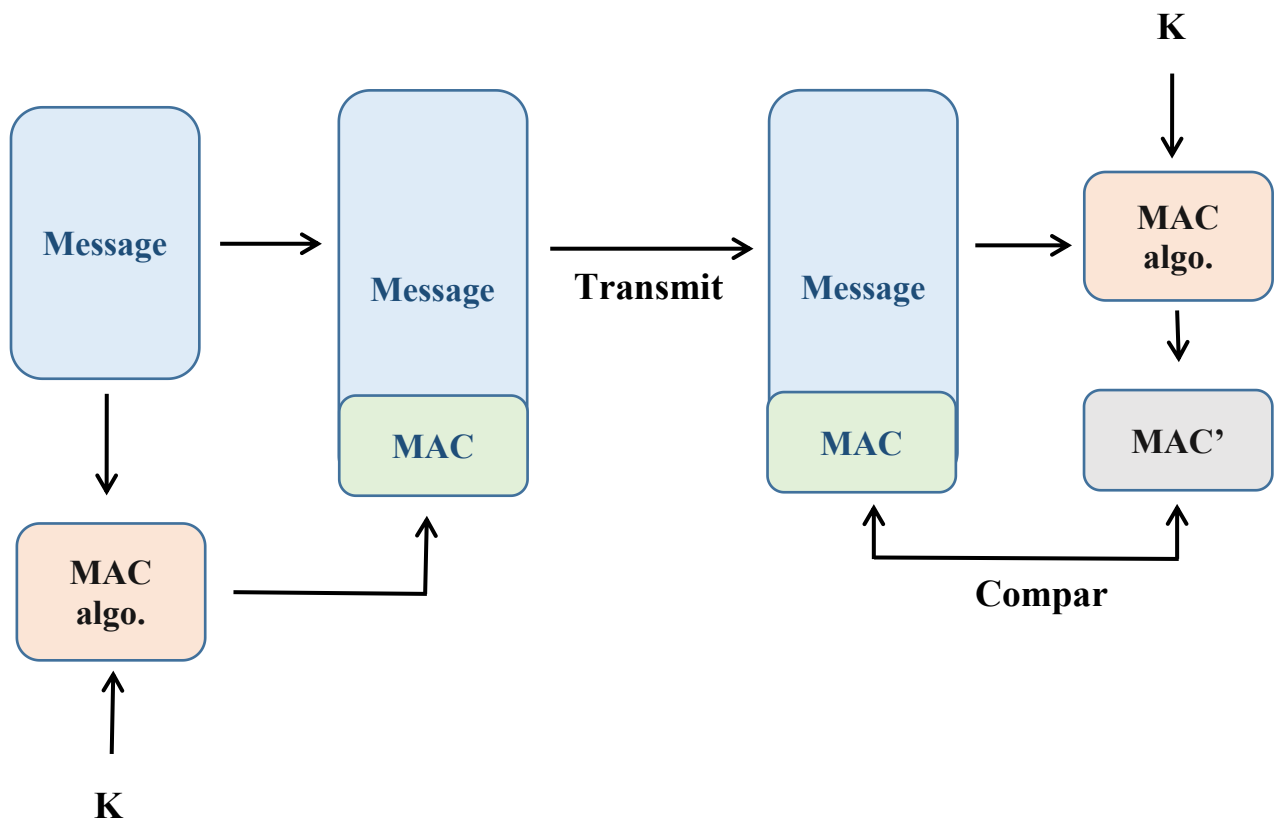


Figure2.1 shows MAC operation.

1. Generate a keyed hash using MD5 for the input file you have created, by following this command:

openssl dgst -md5 -hmac \$(<test_key.key) index.txt

2. Generate a keyed hash using SHA1 for the input file you have created, bu following this command:

openssl dgst -sha1 -hmac \$(<test_key.key) index.txt

3. Generate a keyed hash using SHA256 for the input file you have created, by following this command:

openssl dgst -sha256 -hmac \$(<test_key.key) index.txt

Finally, as you see the hash code change every time the hash algorithm change for the same key and same message. Of course, key size and message size affected the hash code value if they changed, and when it increases hash code becomes more complex.

III. One-Way Hash Functions and Randomness:

1. A file named Input1.txt was created.
2. A hash value for the file was generated using SHA1 algorithm and was saved in a file named H1.txt.
3. A flipping process was applied on the file, flipping one bit using Bless editor.

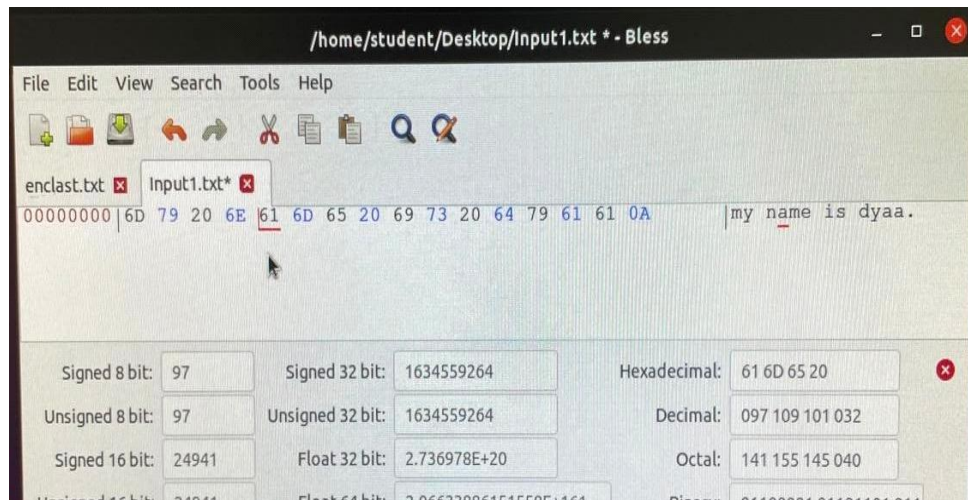


Figure3.1 shows the text in hex editor before editing.

The 4th bit “n” was represented in hexadecimal with the value “6E” and this is equal to “01101110”, we flipped the last bit from 0 to 1 to have the binary value “01101111” which is represented in the hexadecimal with the value 6F to give us the letter “o” instead of “n”.

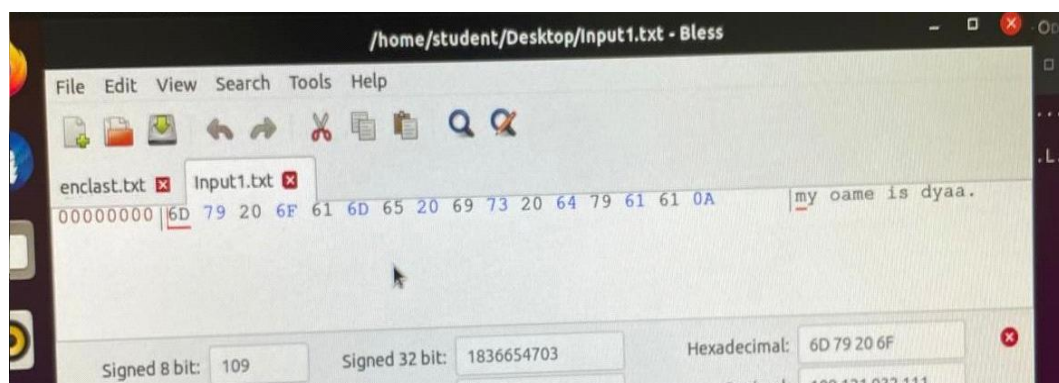


Figure3.2 shows the text in hex editor after editing.

4. After that, a hash value for the modified file was generated with the same algorithm and stored in a file named H2.txt.

5. these were the hash values for the 2 files which are completely different:

```
student@o1J2FP3RnMk4S:~/Desktop$ cat H1.txt
SHA1(Input1.txt)= ca3fc2f14283434cc5a736d74cd32464b22f65e5
student@o1J2FP3RnMk4S:~/Desktop$ cat H2.txt
SHA1(Input1.txt)= 8d728173ff11d28ece8071cab283c8939bd58d7
student@o1J2FP3RnMk4S:~/Desktop$
```

Figure3.3 shows hash code for two files.

6. The same experiment but this time with flipping 5 bits instead of 1bit.

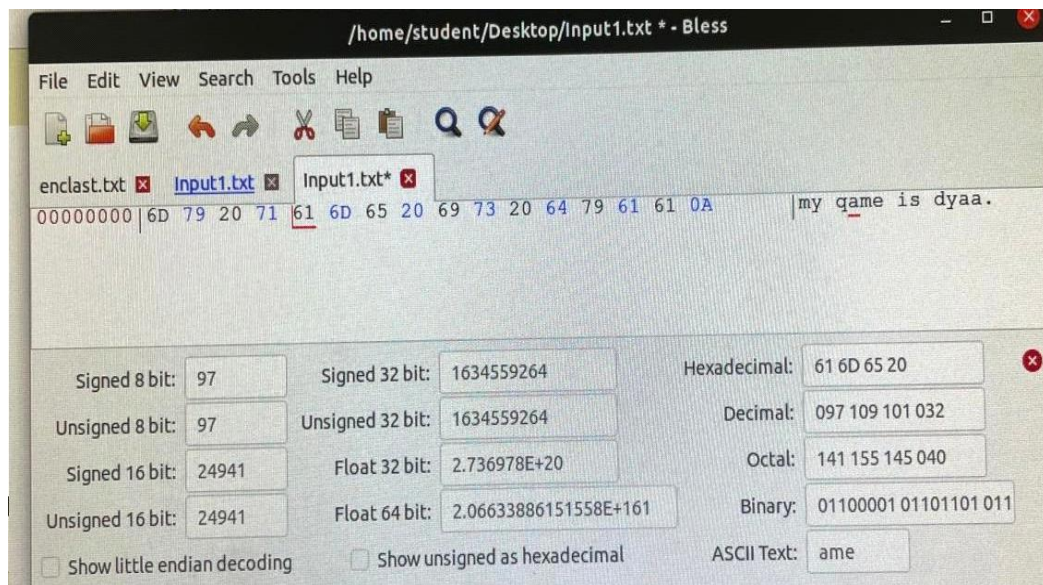


Figure3.4 shows the file before editing.

The 4th bit “n” was represented in hexadecimal with the value “6E” and this is equal to “01101110”, we flipped the last 5 bits to have the binary value “01101111” which is represented in the hexadecimal with the value 71 to give us the letter “q” instead of “n”.

After flipping, a generation process of a new hash value was done using the following command :

openssl dgst -sha1 Input1.txt >> H3.txt

dgst: generate and verify digital signatures using message digests, you can add an option to specify the hash algorithm you want to use, as we have done above following the dgst with sha1 which specifies the hashing algorithm, the default hash algorithm for dgst is SHA256.

sha1: The used hash algorithm

Input1.txt : The file to hash

H3.txt : The file to store the hash value inside.

After applying the above command, a completely different hash value was generated for the modified file:



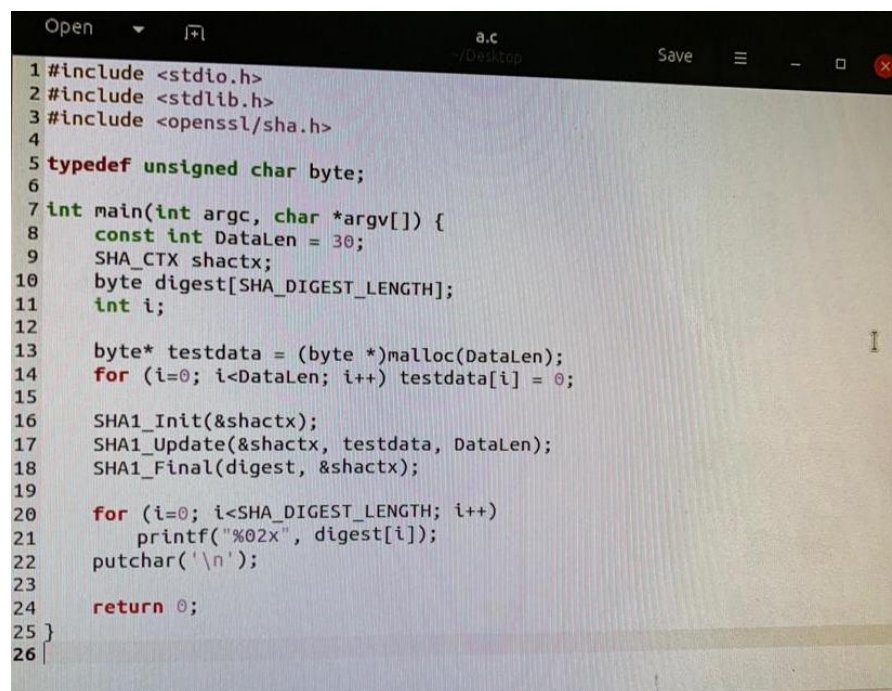
```
student@o1J2FP3RnMk4S:~/Desktop$ cat H1.txt
SHA1(Input1.txt)= ca3fc2f14283434cc5a736d74cd32464b22f65e5
student@o1J2FP3RnMk4S:~/Desktop$ cat H2.txt
SHA1(Input1.txt)= 8d728173ff11d28ece8071cab283c8939bd58d7
student@o1J2FP3RnMk4S:~/Desktop$
```

Figure3.5 shows the hash codes after and before editing files.

This means that SHA1 algorithm is avalanche, whenever a single bit changes, a completely new value gets generated.

IV. C Code Compilation for Digest Generation using SHA:

1. Download the C code, here the c code was downloaded called a.c.



```
Open  a.c  Save  -  +  x
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <openssl/sha.h>
4
5 typedef unsigned char byte;
6
7 int main(int argc, char *argv[]) {
8     const int DataLen = 30;
9     SHA_CTX shactx;
10    byte digest[SHA_DIGEST_LENGTH];
11    int i;
12
13    byte* testdata = (byte *)malloc(DataLen);
14    for (i=0; i<DataLen; i++) testdata[i] = 0;
15
16    SHA1_Init(&shactx);
17    SHA1_Update(&shactx, testdata, DataLen);
18    SHA1_Final(digest, &shactx);
19
20    for (i=0; i<SHA_DIGEST_LENGTH; i++)
21        printf("%02x", digest[i]);
22    putchar('\n');
23
24    return 0;
25 }
26
```

Figure4.1 shows the code that will be used.

2. Compile the code you have downloaded to convert it in machine language that can be understood by process.

3. Run the code using this command:

```
sudo gcc code2.cpp -o md6 -lcrypto  
./md6
```

gcc: its a C compiler.

code2.cpp: the code that downloaded.

md6: the object where the code output saved.

-lcrypto: this command link crypto library with our code.

./md6: to show the output of the file.

The output of this code will save in object file md6, so to show the output do:



```
student@linux:/usr/lib/ssl$ sudo gcc code2.cpp -o md6 -lcrypto  
student@linux:/usr/lib/ssl$ ls  
ca.crt  certs      crt      md6      newcerts  opensslcopy1.txt  private  server.key  
ca.key  code2.cpp  demoCA  misc    openssl.cnf  opensslcopy.cnf  server.csr  server.pen  
student@linux:/usr/lib/ssl$ sudo chmod 777 md6  
student@linux:/usr/lib/ssl$ ./md6  
deb6c11e1971aa61dbbcb76e5ea7553a5bea7b7  
student@linux:/usr/lib/ssl$
```

Figure4.2 shows the output of the code.

This code finds the hash code of the message depending on its size and the key length.

Now, will change the message from 0 to 10, and run the code using the same command with different object file.

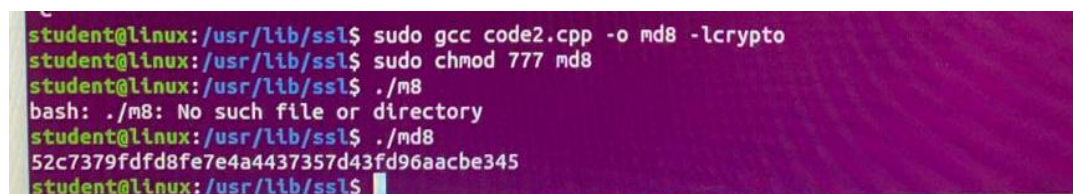


```
student@linux:/usr/lib/ssl$ sudo gcc code2.cpp -o md7 -lcrypto  
student@linux:/usr/lib/ssl$ sudo chmod 777 md7  
student@linux:/usr/lib/ssl$ ./md7  
856b7533aa6cc8b9b5dd9149a895b1fb3d33f803  
student@linux:/usr/lib/ssl$
```

Figure4.3 shows the output of the code when message was changed.

As you show, the hash code was changed, so, that's means changing message will change the hash code value and each message has its own hash code.

Now, will change the message size from 30 to 1000, and run the code using the same command with different object file.



```
student@linux:/usr/lib/ssl$ sudo gcc code2.cpp -o md8 -lcrypto  
student@linux:/usr/lib/ssl$ sudo chmod 777 md8  
student@linux:/usr/lib/ssl$ ./m8  
bash: ./m8: No such file or directory  
student@linux:/usr/lib/ssl$ ./md8  
52c7379fd8fe7e4a4437357d43fd96aacbe345  
student@linux:/usr/lib/ssl$
```

Figure4.4 shows the output of the code when message size was changed.

Also, when the message size was changed the hash code will change.

Finally, change the message size many times and find the hash code with the time that is needed to find it.

```
student@linux: /usr/lib/ssl $ time ./md12
8cf069190d5d3a754162cf9bd5470dadb7cbd7ae

real    0m0.002s
user    0m0.002s
sys      0m0.000s
student@linux: /usr/lib/ssl $ gedit code2.cpp
student@linux: /usr/lib/ssl $ sudo gcc code2.cpp -o md2 -lcrypto
student@linux: /usr/lib/ssl $ sudo chmod 777 md2
student@linux: /usr/lib/ssl $ time ./md2
b93ff5b0b17a50e4234d3e04cb64892bdee2c218

real    0m0.001s
user    0m0.001s
sys      0m0.000s
```

Figure4.4 shows the hash code to different message size.

Here in md12 file the message size equal 30000, and in md2 file the message size equal 300.

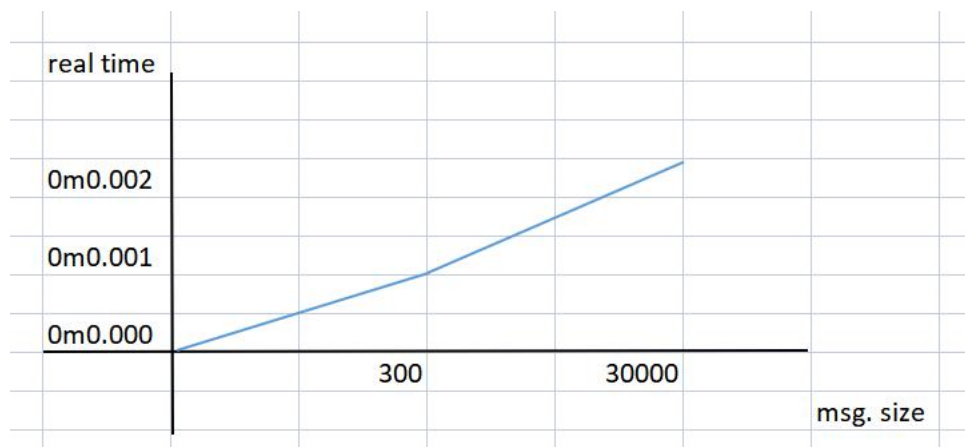


Figure4.5 shows the relation between runtime and file size.

As it shows in last figure, when file size increase the runtime needed will increase.

V. C code compilation for digest generation using MD5:

1. The C code was download.
2. The code was compiled using the following command:

sudo g++ md5.cpp -out md6.cpp -lcrypto

3. To run the code we used the command: `./md6.cpp input.txt` , “input.txt” file is used to store the output of the code.

```
student@o1J2FP3RnMk4S:/usr/lib/ssl$ time ./md6 Input.txt
MD5(Input.txt)= b1c2cf4d71b87346d2ded125fc675f5f

real    0m0.001s
user    0m0.000s
sys     0m0.001s
student@o1J2FP3RnMk4S:/usr/lib/ssl$
```

Figure5.1 shows the time needed to find hash code.

The result of running the code was generating an MD5 hash value for the message, and this value was stored in the input.txt file.

Figures that show the changing in the hash value with every change in the data inside the file :

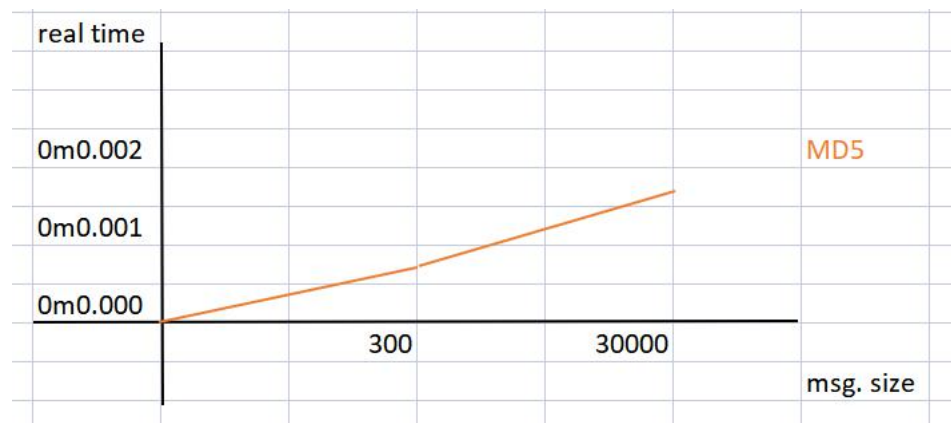


Figure5.2 shows the relation between runtime and file size.

When compare between SHA and MD5 find for the same file size in MD5 need less run time than SHA,because MD5 less CPU-intensive than SHA. It shown in this figure:

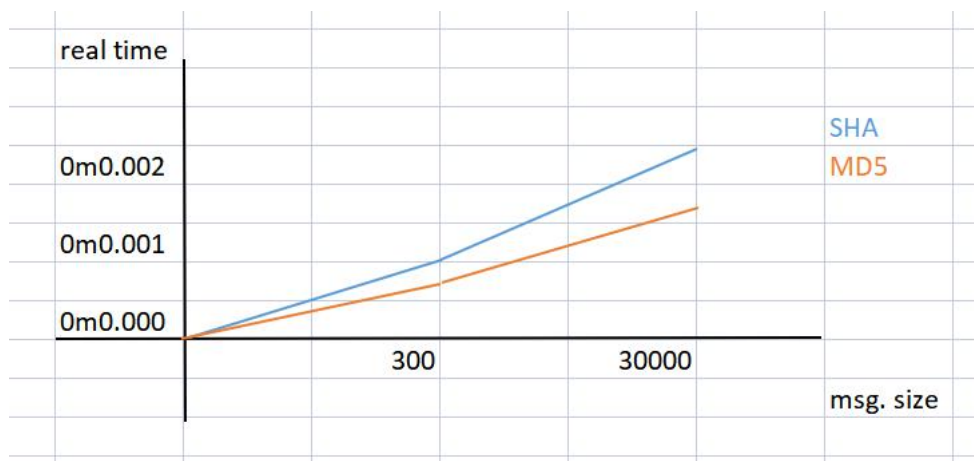


Figure5.3 shows difference between MD5 and SHA.

For the avalanche effect, MD5 algorithm meets the avalanche effect, when a small change happens, this results in a mostly different hash, to be sure of that, we made several experiments on the same file, every time changing a bit of the file, which was resulting in a completely new hash value.

Conclusion

In the end, this experiment talked about hash code, what a message authentication code is, how to generate a hash code in different hash algorithm, generate message authentication code, and what is the difference between these algorithm.

References

- **Wikipedia**
- **openssl.org**