Cryptography - Lab 6 - One-Way Hash Functions

Javier Antxon Garrues Apecechea

Student ID: 24647808

In this lab we will get familiar with one-way hash functions.

Part 3: SHA-512 HASH FUNCTION

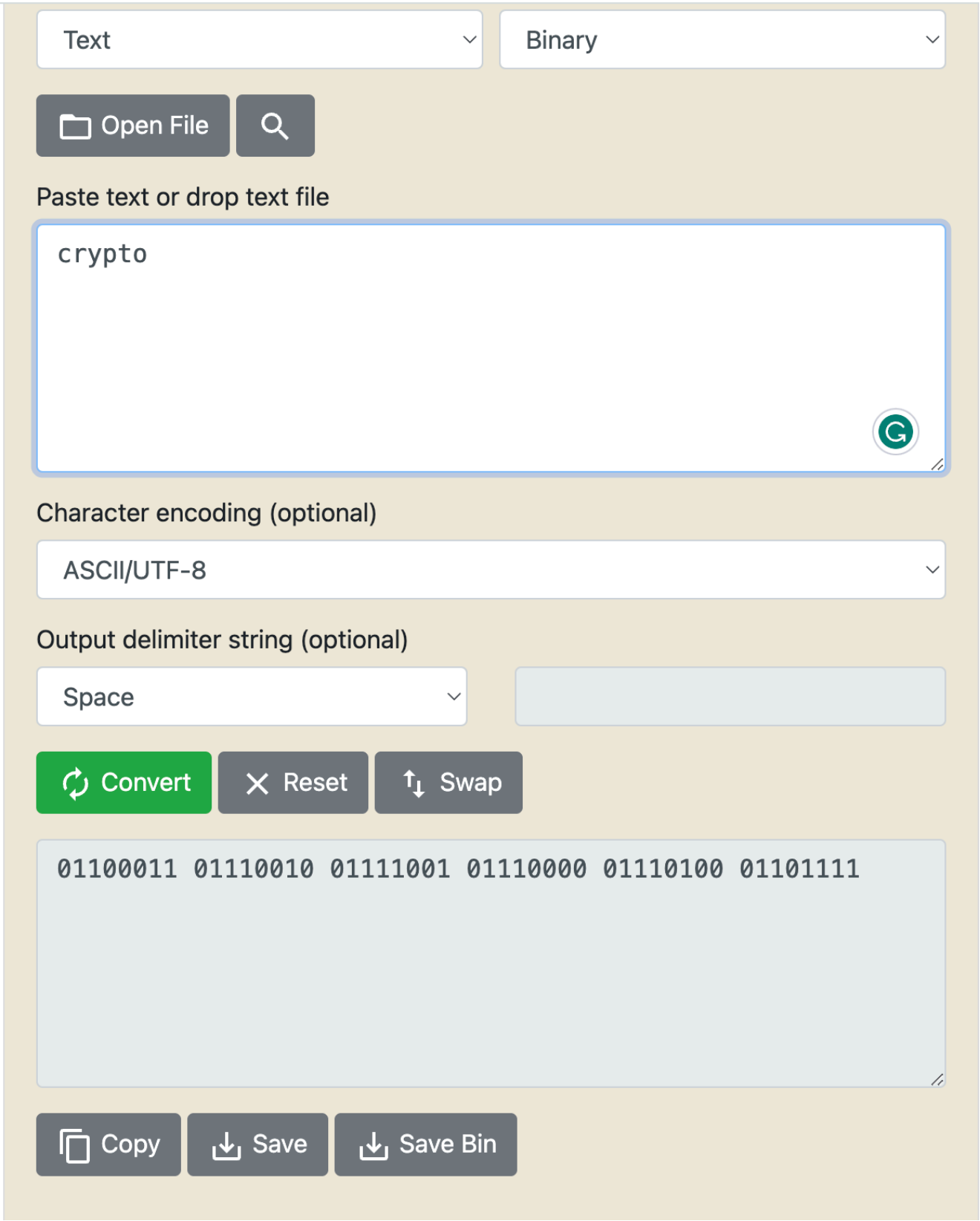
3.1.Details of SHA-256/512 algorithm

First, we install ruby:

```
brew install ruby
```

Then, we convert *crypto* to binary and execute the ruby file sha256.rb with the binary value as input:

```
ruby sha256.rb 0b01100011011100100111100101110000011101000110111
```

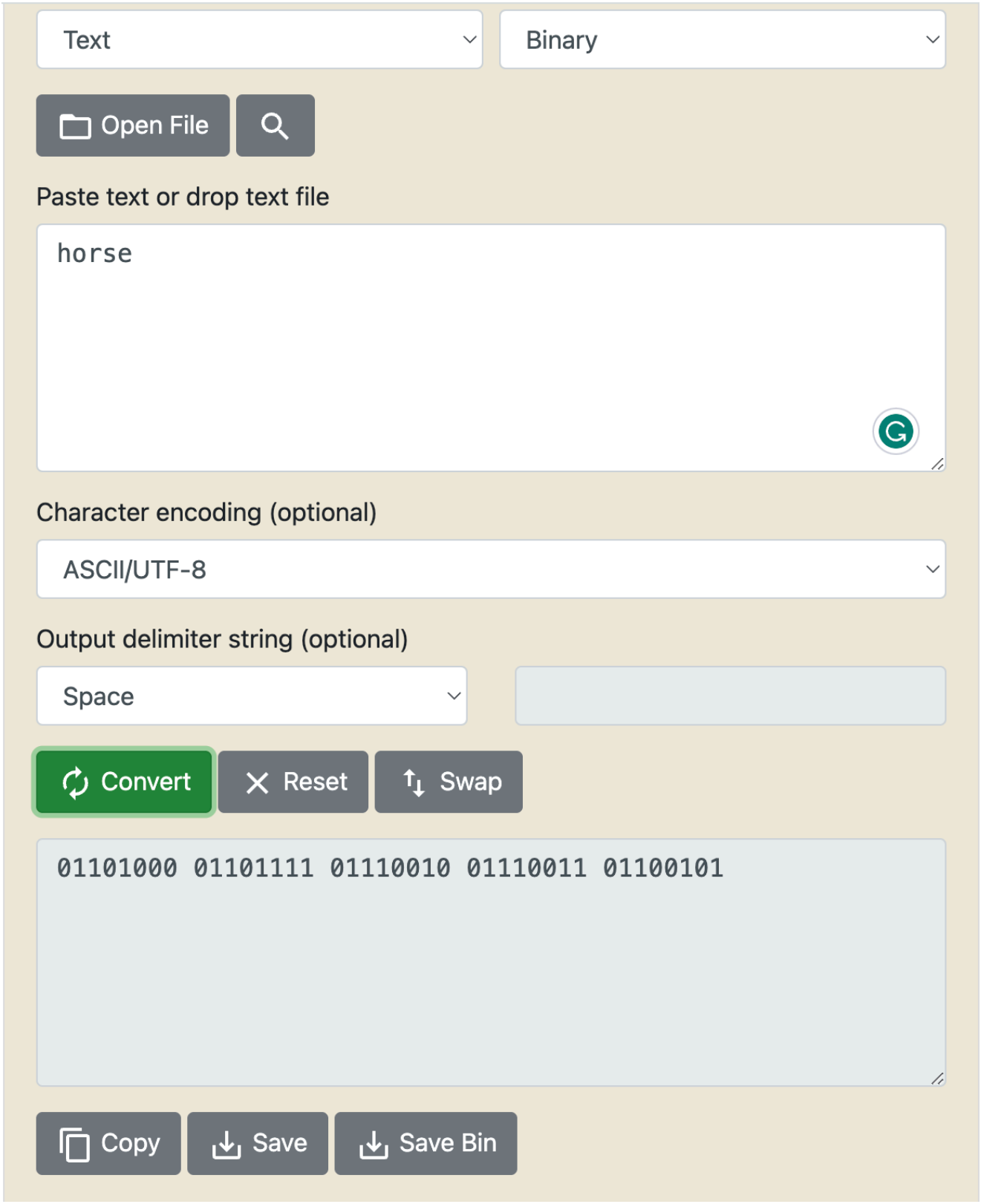


The obtained output is:

```
da2f073e06f78938166f247273729dfe465bf7e46105c13ce7cc651047bf0ca4
```

Let's try with another word like horse:

```
ruby sha256.rb 0b01101000011011110111001001110011011001010l
```



The process is shown iteratively and the final output is:

```
-------
message:
-------
input:   0b0110100001101111011100100111001101100101 (binary)
bytes:   [104, 111, 114, 115, 101]
message: 0110100001101111011100100111001101100101

-------
padding: (512 bits)
-------
message: 0110100001101111011100100111001101100101100000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000000000000000000101000

--------------
message blocks:
--------------
0: 0110100001101111011100100111001101100101100000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000000000000101000

  -------
  block 0:
  -------



  ----------------
  message schedule:
  ----------------
  W47 01101000111010011101111101100011
  W48 10010000000101001111011111101101
  W49 10011110101110001010010111111100
  W50 10101110110011110001000111001100
  W51 01100001011100000000100111011101
  W52 11011110111010011100110110100000
  W53 10000110111001101001001000100010
  W54 11110110101110001001001011000000
  W55 11011001110011100100000010111001
  W56 01110010011100000111010001010011
  W57 10100010000111010011110000011101
  W58 10111111001001101100110111011111
  W59 10100100001011111110010101011010
  W60 00100100001000000001110000100100
  W61 10001000101001100101100101111111
  W62 00001010011000101001111100100000
  W63 10110111111001100100001110011110

  -----------
  compression: (H0 -> H1)
  -----------
  W36 = 01010101110100001011001000100011 (message schedule)
  K36 = 01100101000010100111001101010100 (constant)

  T1 = Σ1(e) + Ch(e, f, g) + h + K36 + W36 = 01011100000100000011101111101110
  T2 = Σ0(a) + Maj(a, b, c)                = 11100011001001000111111011001010

  a = 11110111100011110010110111011001 <- T1 + T2
  b = 10110001100010011110010111110110
  c = 10101010011110000111100101111001
  d = 10101100110111010011111101101110
  e = 00100010111010100111010000100010 + T1
  f = 00101001101100001001100100000111
  g = 01001010001000001001000000111100
  h = 10111110100111101010000011001010
```

```
fd62862b6dc213bee77c2badd6311528253c6cb3107e03c16051aa15584eca1c
```

Which is obtained concatenating the hash values of the a-h variables:

```
----------------
final hash value: (H1)
----------------
a = 11111101011000101000011000101011 = fd62862b
b = 01101101110000100001001110111110 = 6dc213be
c = 11100111011111000010101110101101 = e77c2bad
d = 11010110001100010001010100101000 = d6311528
e = 00100101001111000110110010110011 = 253c6cb3
f = 00010000011111100000001111000001 = 107e03c1
g = 01100000010100011010101000010101 = 6051aa15
h = 01011000010011101100101000011100 = 584eca1c

fd62862b6dc213bee77c2badd6311528253c6cb3107e03c16051aa15584eca1c
```

The different functions used during the process can be indivually executed too. For instance, the following code returns the constants used during the process:

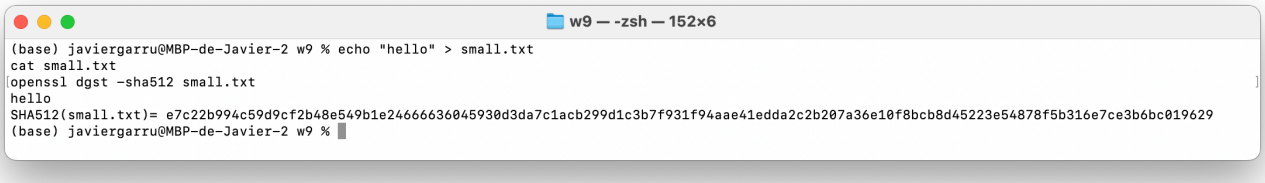
```
ruby constants.rb
```

```
-------------
constants (K)
-------------
0  = 01000010100010100010111110011000
1  = 01110001001101110100010010010001
2  = 10110101110000001111101111001111
3  = 11101001101101011101101110100101
4  = 00111001010101101100001001011011
5  = 01011001111100010001000111110001
6  = 10010010001111111000001010100100
7  = 10101011000111000101111011010101
8  = 11011000000001111010101010011000
9  = 00010010100000110101101100000001
10 = 00100100001100011000010110111110
11 = 01010101000011000111110111000011
12 = 01110010101111100101110101110100
13 = 10000000110111101011000111111110
14 = 10011011110111000000011010100111
15 = 11000001100110111111000101110100
16 = 11100100100110110110100111000001
17 = 11101111101111100100011110000110
18 = 00001111110000011001110111000110
19 = 00100100000011001010000111001100
20 = 00101101111010010010110001101111
21 = 01001010011101001000010010101010
22 = 01011100101100001010100111011100
23 = 01110110111110011000100011011010
24 = 10011000001111100101000101010010
25 = 10101000001100011100011001101101
26 = 10110000000000110010011111001000
27 = 10111111010110010111111111000111
28 = 11000110111000000000101111110011
29 = 11010101101001111001000101000111
30 = 00000110110010100110001101010001
31 = 00010100001010010010100101100111
32 = 00100111101101110000101010000101
33 = 00101110000110110010000100111000
34 = 01001101001011000110110111111100
35 = 01010011001110000000110100010011
36 = 01100101000010100111001101010100
37 = 01110110011010100000101010111011
38 = 10000001110000101100100100101110
39 = 10010010011100100010110010000101
40 = 10100010101111111110100010100001
41 = 10101000000110100110011001001011
42 = 11000010010010111000101101110000
43 = 11000111011011000101000110100011
44 = 11010001100100101110100000011001
45 = 11010110100110010000011000100100
46 = 11110100000011100011010110000101
47 = 00010000011010101010000001110000
48 = 00011001101001001100000100010110
49 = 00011110001101110110110000001000
50 = 00100111010010000111011101001100
51 = 00110100101100001011110010110101
52 = 00111001000111000000110010110011
53 = 01001110110110001010101001001010
54 = 01011011100111001100101001001111
55 = 01101000001011100110111111110011
56 = 01110100100011111000001011101110
57 = 01111000101001010110001101101111
58 = 10000100110010000111100000010100
59 = 10001100110001110000001000001000
60 = 10010000101111101111111111111010
61 = 10100100010100000110110011101011
62 = 10111110111110011010001111110111
63 = 11000110011100010111100011110010
```

3.2.SHA-512 on a small file:

We create a small file and obtain its hash using SHA-512:

```
echo "hello" > small.txt
cat small.txt
openssl dgst -sha512 small.txt
```

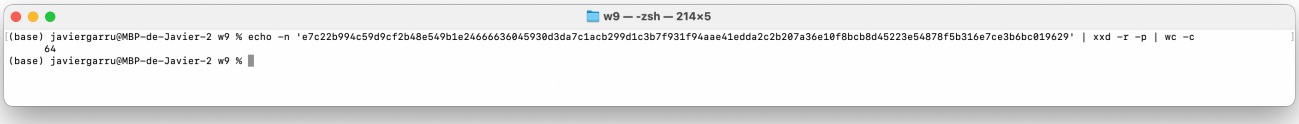



```
e7c22b994c59d9cf2b48e549b1e24666636045930d3da7c1acb299d1c3b7f931f94aae41edda2c2b207a36e
10f8bcb8d45223e54878f5b316e7ce3b6bc019629
```

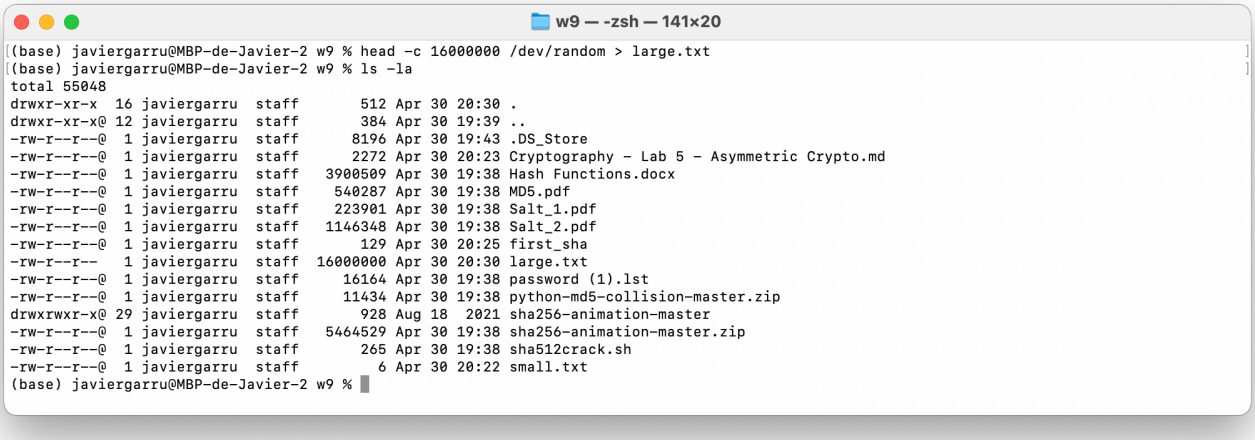
Question: How long is the SHA-512 hash value?

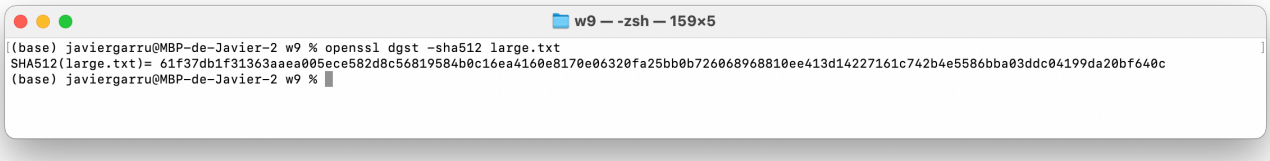
The hash value is 64 bytes long or 512 bits long:

```
echo -n
'e7c22b994c59d9cf2b48e549b1e24666636045930d3da7c1acb299d1c3b7f931f94aae41edda2c2b207a36
e10f8bcb8d45223e54878f5b316e7ce3b6bc019629' | xxd -r -p | wc -c
```



3.3 SHA512 on a large file



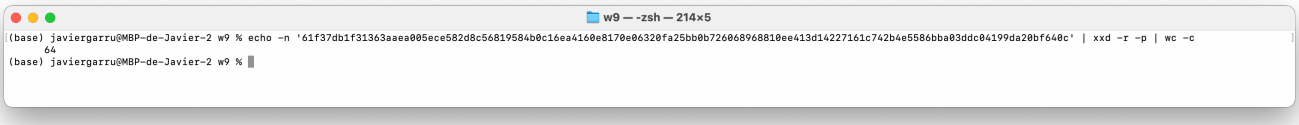



```
61f37db1f31363aaea005ece582d8c56819584b0c16ea4160e8170e06320fa25bb0b726068968810ee413d1
4227161c742b4e5586bba03ddc04199da20bf640c
```

Question: How long is the SHA-512 hash value?

The hash value is 64 bytes long or 512 bits long:

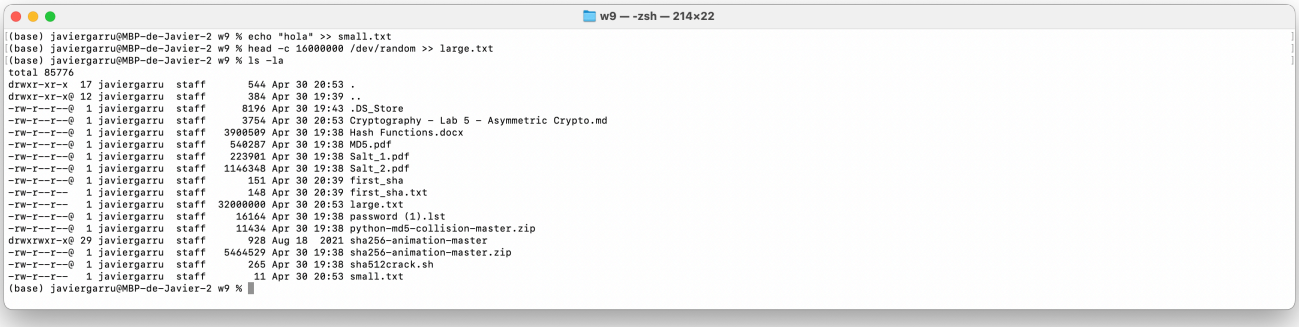
```
echo -n
'61f37db1f31363aaea005ece582d8c56819584b0c16ea4160e8170e06320fa25bb0b726068968810ee413d
14227161c742b4e5586bba03ddc04199da20bf640c' | xxd -r -p | wc -c
```



3.4 Change the file and redo the SHA-512

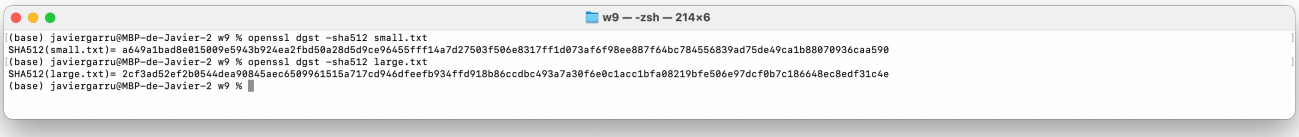
```
echo "hola" >> small.txt
head -c 16000000 /dev/random >> large.txt
nano small.txt
ls -la
```

We can see how the size of large.txt has doubled:



And *small.txt* has a new word underneath *hello*:

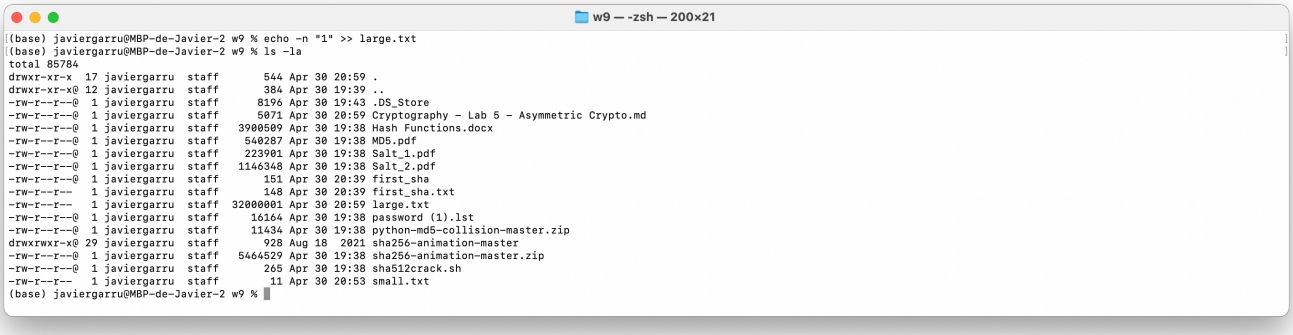

```
openssl dgst -sha512 small.txt
openssl dgst -sha512 large.txt
```




```
SHA512(small.txt)=
a649a1bad8e015009e5943b924ea2fbd50a28d5d9ce96455fff14a7d27503f506e8317ff1d073af6f98ee88
7f64bc784556839ad75de49ca1b88070936caa590

SHA512(large.txt)=
2cf3ad52ef2b0544dea90845aec6509961515a717cd946dfeefb934ffd918b86ccdbc493a7a30f6e0c1acc1
bfa08219bfe506e97dcf0b7c186648ec8edf31c4e
```

Let's now make a slight change the large file and recalculate the hash:

```
echo -n "1" >> large.txt
```



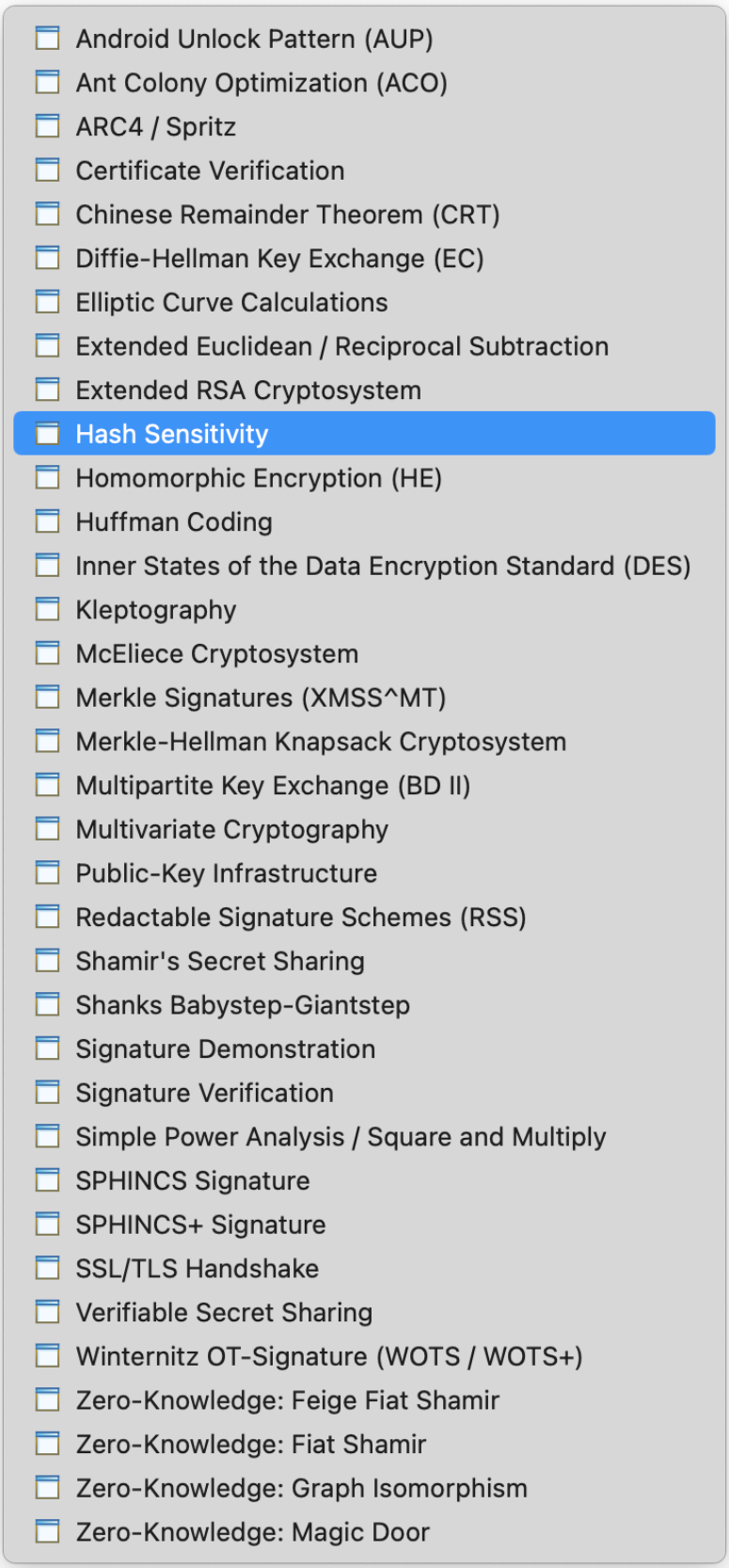

```
SHA512(large.txt)=
0e7dce4090d9c573c8bbadf1e6f92b363a14359c92a26a54554f9f81221536160ce2fd1a8bc4c08d4406a9e
bcb4452fecfdb9c6c057cff0286e97193826f379e

###### OLD HASH BELOW
SHA512(large.txt)=
2cf3ad52ef2b0544dea90845aec6509961515a717cd946dfeefb934ffd918b86ccdbc493a7a30f6e0c1acc1
bfa08219bfe506e97dcf0b7c186648ec8edf31c4e
```

Question: How long is the SHA-512 hash value? Is this hash value significantly different from the last hash value?

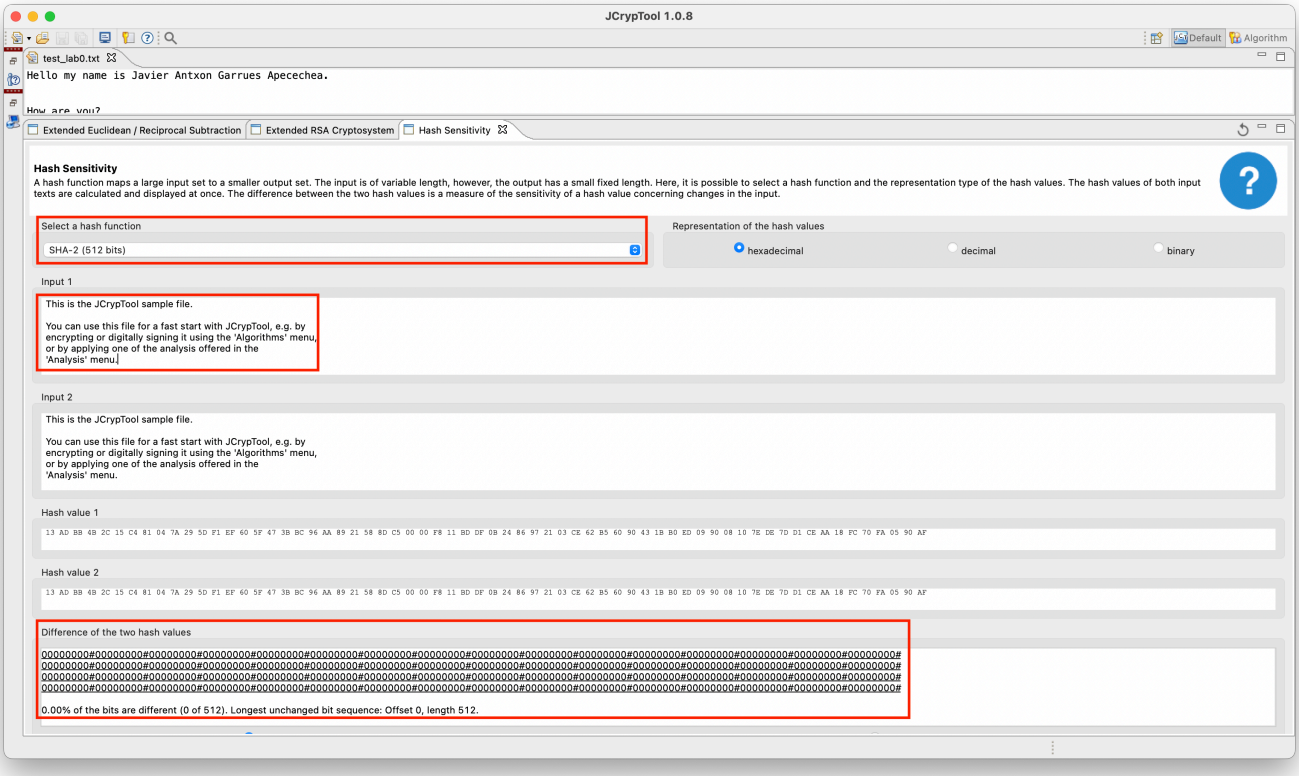
Both has values are 512 bits again. The new hash value is significantly different from the original one, even though we have only added a bit to the file as the command `ls -la` showed on the file size. This shows how powerful the hash algorithm used is.

3.5 Hash Sensitivity Visualisation Using JCrypTools

Now, we will use the tool **JCrypTool**. We select *`Visuals-> Hash sensitivity`*:

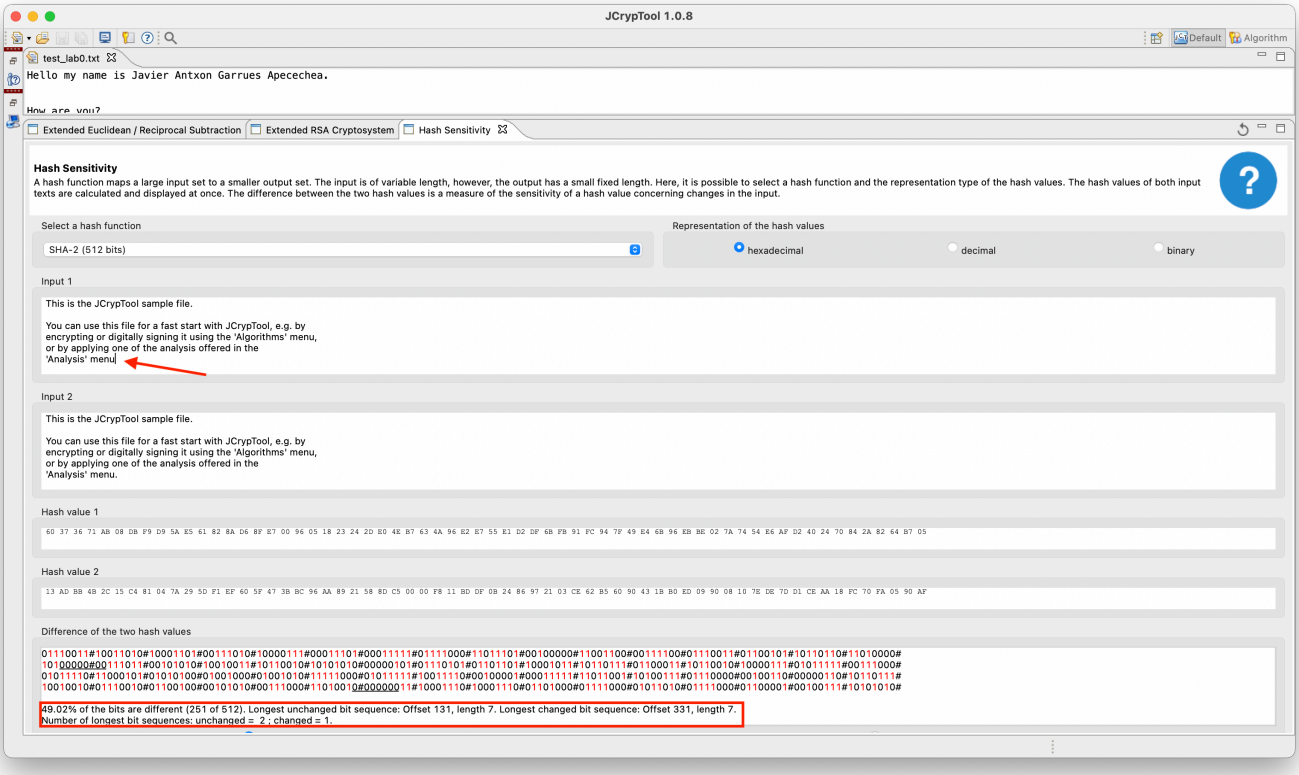


And then proceed to modify the orginal input1:

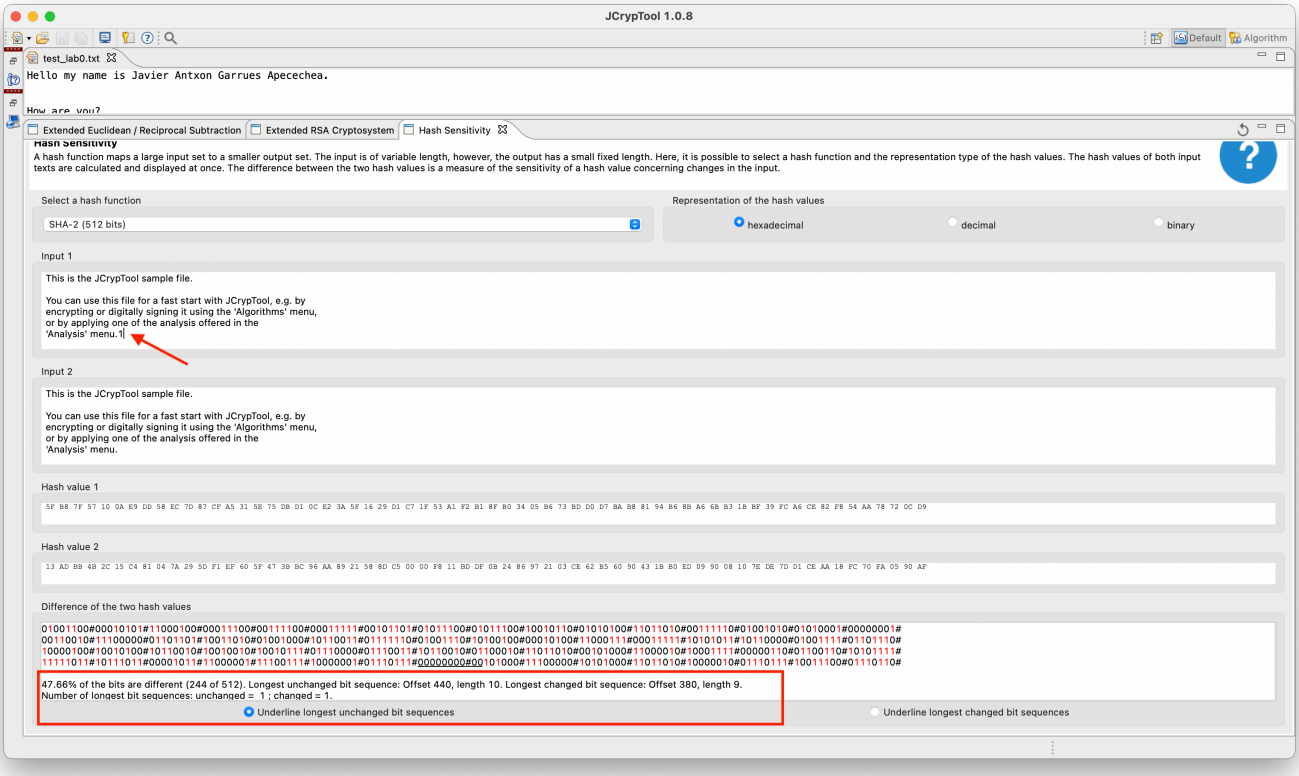


Initially there are no difference between the two hash values.

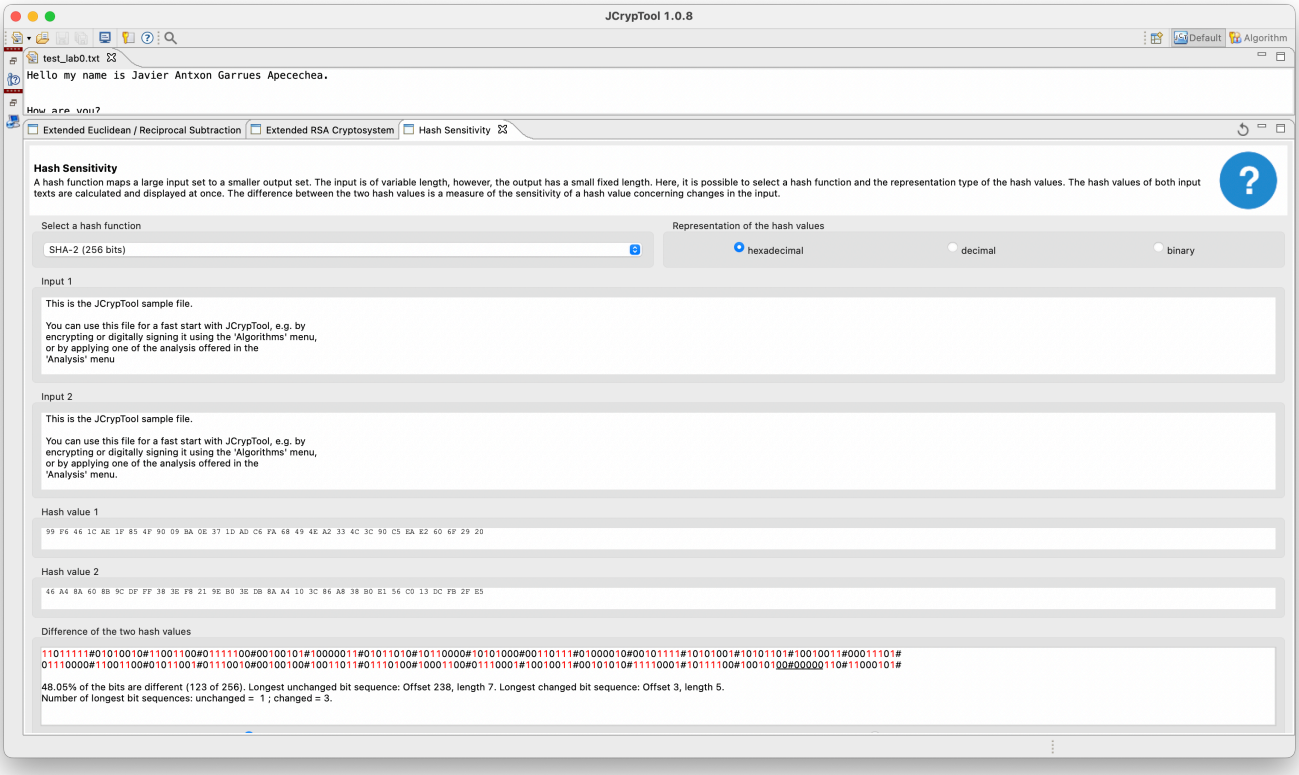
Then, we eliminate the last full spot from input 1 and compare the two hash values:



49.02% of the bits are different after eliminating the full stop. In addition to this, the longest bit sequence unchanged is 7 and there is only one such sequence. This shows that the changes are evenly distributed. Let's see what's the percentage change after adding a bit:



The results are similar with a change of **47.66%**. Apart from this hash function there are several more available. Amongst them we highlight **SHA-2 (256 bits)**:



Again a **48.05%** of the bits are changed after eliminating the full stop.

Question: Comment the sensitivity of hash functions to the input. Why can hash functions verify data integrity?

The sensitivity to the input of hash functions is really big as we have seen that a small change (or even the addition of one single bit) can make the hash value change almost a 50%. This is the most valuable characteristic of hash functions.

To verify data integrity the hash function of the message must be computed by the sender and sent along the messge. Once the receiver gets the message, its hash is recomputed and compare to the received one. If both hash values match, then the message hasn't been changed during the communication. As we have seen, a small change would have made the hash to change dramatically and be noticed by the receiver after recomputing the hash.

Part 4: HASH CRACK

4.1 Brute-force attack against the preimage resistant

Finding the preimage of a hash is finding an input that when applied the hash function is mapped to the hash value. We are going to try to find the preimage on SHA-512 of **ab1c2b83a2d9b0304541de63f302a9580c0b4252dcd70d5a6ce0bd5b8829d2b3bc1f727c701401130 71bcc7f16a737a74a6f0f5166168185dd43bbba690b5041**.

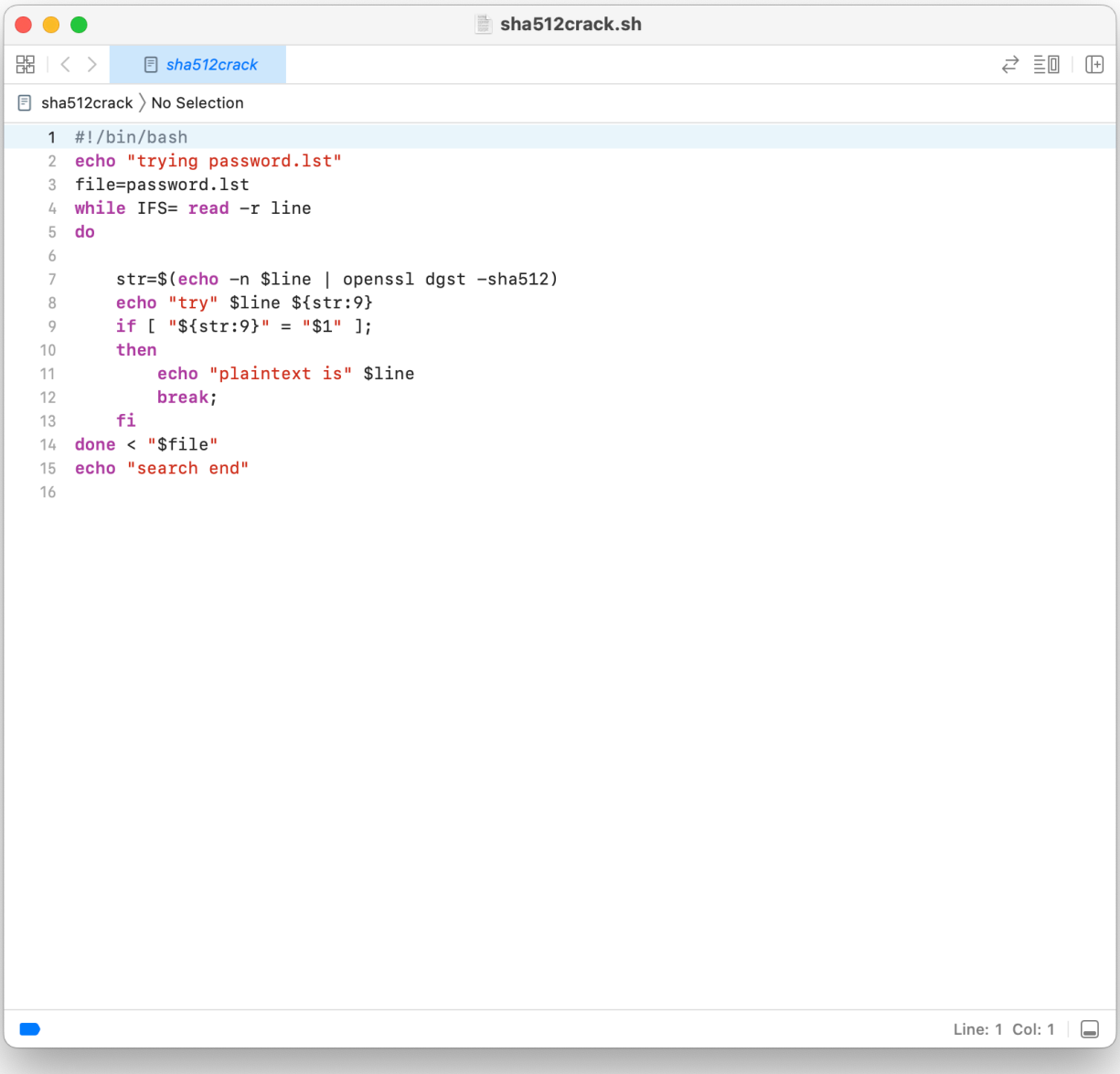
We first give us permission to execute the program sha512crack.sh:

```
chmod +x sha512crack.sh
```

And then execute the program:

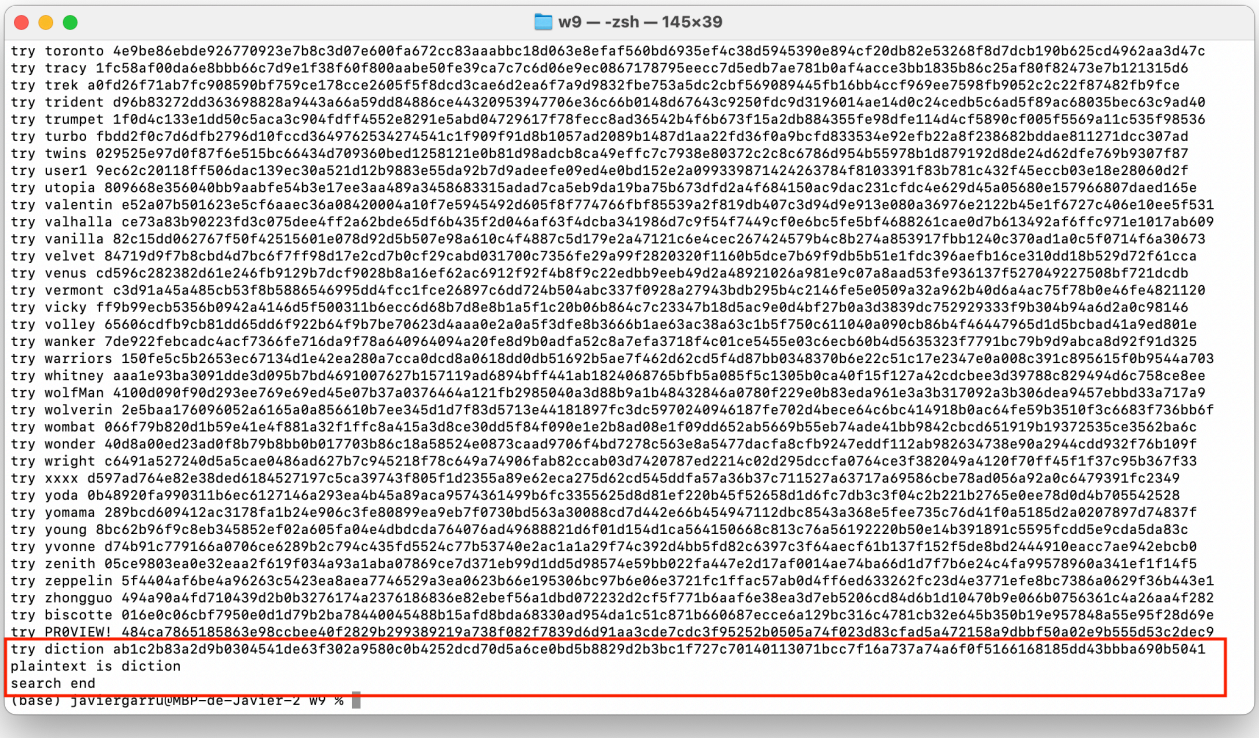
```
Q: What is the SHA-512 preimage of
“ab1c2b83a2d9b0304541de63f302a9580c0b4252dcd70d5a6ce0bd5b8829d2b3bc1f727c701401130
71bcc7f16a737a74a6f0f5166168185dd43bbba690b5041”?
```

The program uses the words in password.lst to compute their hash values using openssl dgst. Then, compares the obtained value to the hash whose preimage we are looking for. The program would not work if the word whose hash matches the target one wouldn't be in the password.lst.



Question: What is the SHA-512 preimage of "ab1c2b83a2d9b0304541de63f302a9580c0b4252dcd70d5a6ce0bd5b8829d2b3bc1f727c70140113071bcc7f16a737a74a6f0f5166168185dd43bbba690b5041"?

The preimage of that hash is: **diction**.



Question: How does the brute-force crack work?

As mentioned before, the program consists of an interative loop that computes the hash of a word in the password.lst at every iteration. The hash is computed using openssl and the loop doesn't stop until the computed hash matches the target one. This brute force attack is successful due to the word diction being found in the password.lst. Otherwise, it wouldn't have succeeded.

4.2 SHA-512 Crack with Rainbow table against the preimage resistant

A rainbow table is a precomputed table for caching the output of cryptographic hash functions, usually for cracking passwords. How does it really work?

In this table a function R will be used. This function is called a reduction function and maps hash values to the original domain of the hash function. It is not an inverse, as the only condition that must satisdy is that it maps hash values to a predefined length output in its domain.

The trick is to use the hash and reduction functions sequently creating chains that start from the passwords belonging to the table. For instance, let's assum the domain of the hash function is lowercase alphabetic 6-character passwords, and the hash values were 32 bits long. Then applying the functions we could obtain the following chain from the password *aaaaaa*:

$$\texttt{aaaaaa} \xrightarrow[H]{} \texttt{281DAF40} \xrightarrow[R]{} \texttt{sgfnyd} \xrightarrow[H]{} \texttt{920ECF10} \xrightarrow[R]{} \texttt{kiebgt}$$

Once this is computed, the first (*aaaaaa*) and last (*kiebgt*) values of the chain are saved. Then, once we are given a hash value whose preimage we want to find, we apply the reduction function to this hash value. Then, we compare the obtained value to the end of the chains and if there is a match there is a high probability that the chain will contain the hash. For instance:

$$\texttt{920ECF10} \xrightarrow[R]{} \texttt{kiebgt}$$

Since "kiebgt" is one of the endpoints in our table, the corresponding starting password "aaaaaa" allows to follow its chain until 920ECF10 is reached:

$$\texttt{aaaaaa} \xrightarrow[H]{} \texttt{281DAF40} \xrightarrow[R]{} \texttt{sgfnyd} \xrightarrow[H]{} \texttt{920ECF10}$$

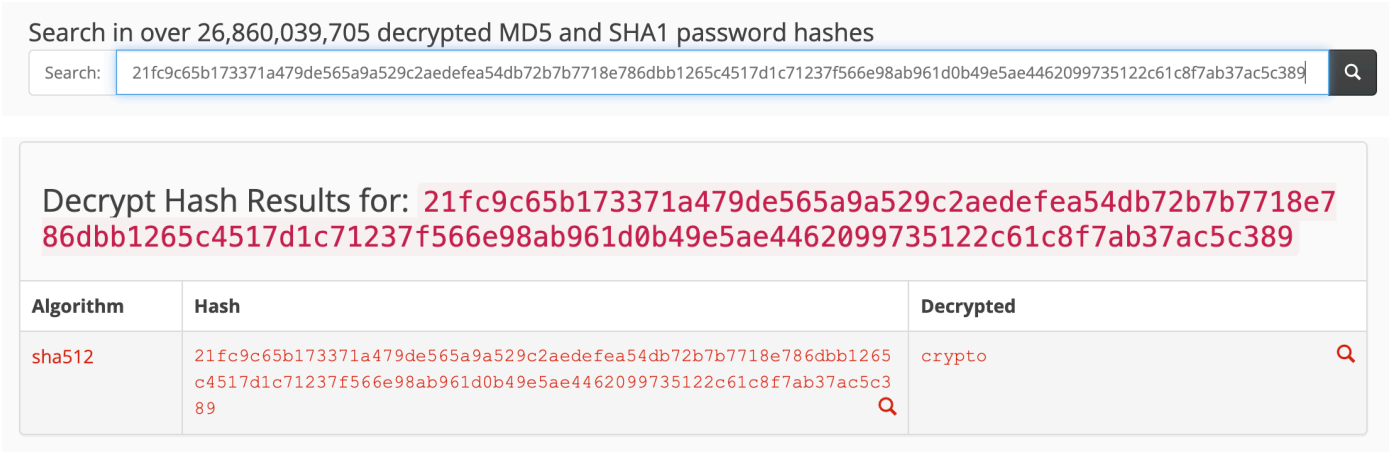
Thus, the password is "sgfnyd" (or a different password that has the same hash value).

The difficulties and not great effectiveness of this method rely on the collisions of chains and the choice of R. Using a sequence of related functions R can help to solve those probelms.

This information has been extracted from: `https://en.wikipedia.org/wiki/Rainbow_table`.

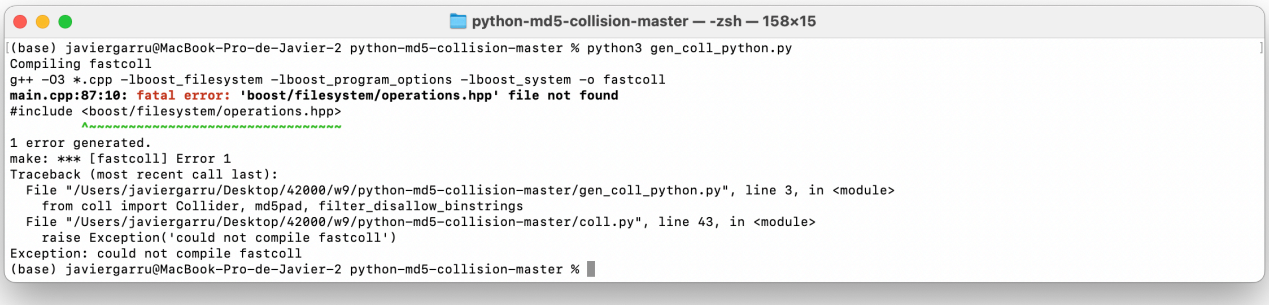
Question: What is the SHA-512 preimage of "21fc9c65b173371a479de565a9a529c2aedefea54db72b7b7718e786dbb1265c4517d1c71237f566e98ab961d0b49e5ae4462099735122c61c8f7ab37ac5c389"?

The SHA-512 preimage is **crypto**. It was obtained using `https://hashtoolkit.com`.



4.3 MD5 collision attack against the strong collision resistant

Due to mac incompatibility I couldn't execute this part of the lab in my computer:



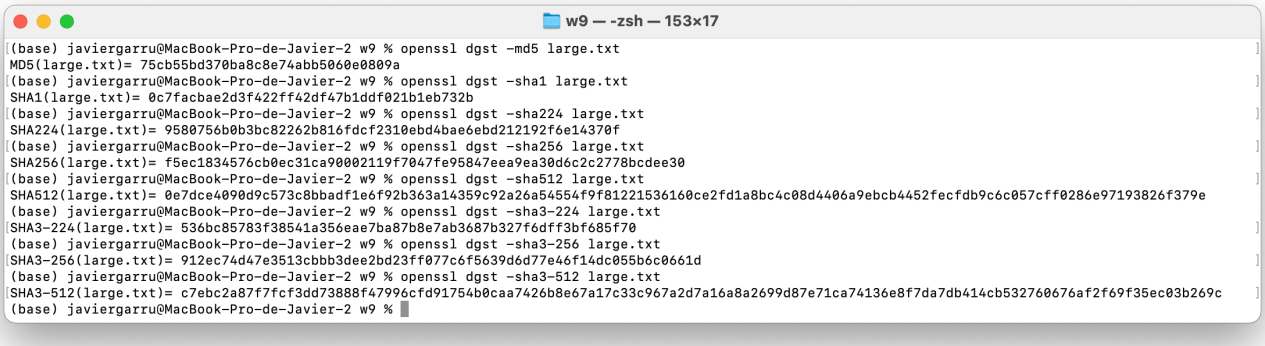
Question: Suppose an adversary uses brute-force to attack hash functions. For a general hash function with m-bit hash value, how many times on average should an adversary try to find the preimage of a given hash value? How many times on average should an adversary try to find two preimages that generate the same hash value?

For a general hash function with m-bit hash value an adversary should on average try to find the preimage 2^(m - 1) times. The adversary should try half of the values on the codomain. Due to the hash function having m-bits with two possible values 0 or 1 the size of this space is 2^m. As the question asks *on average*, the adversary would have a 50% chance of finding the preimage in the 2^(m-1) first tries. The values being tried would be the messages in the domain space of the hash function but, due to the properties of the hash functions (no collisions), we assume this is the same as trying different hash values every time.

To find two preimages that have the same hash values, an adversary would have to try on average sqrt(2^m) values. This number is obtained based on the birthday paradox explained in the slides.

Part 5: OTHER HASH FUNCTIONS

We use openssl dgst and the different options available:



Question: What are the lengths of the hash values?

Hash function	Length
MD5	128 bits
SHA1	160 bits
SHA224	224 bits
SHA256	256 bits
SHA512	512 bits
SHA3-224	224 bits
SHA3-256	256 bits
SHA3-512	512 bits

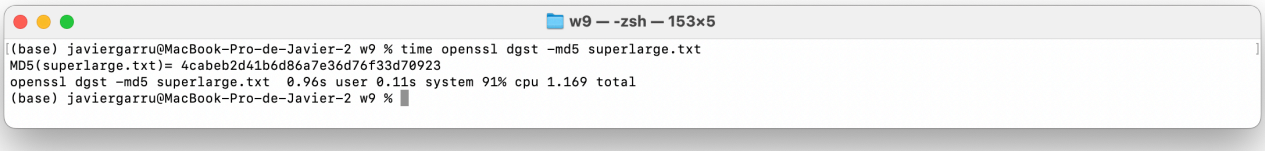
Now, we create a large file (50MB) and compare the times every function takes to compute the hash value:

```
head -c 500000000 /dev/random >> superlarge.txt
```

We will compare the total times for simplicity reasons:

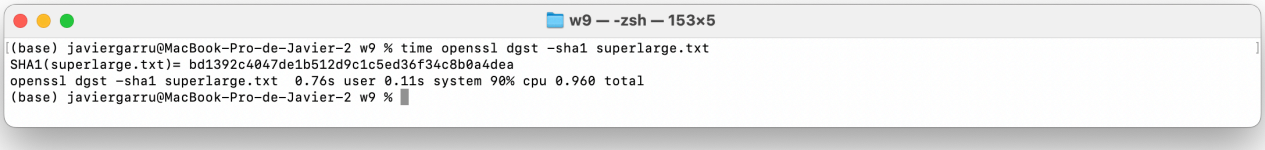
Time MD5: 1.169s

```
time openssl dgst -md5 superlarge.txt
```



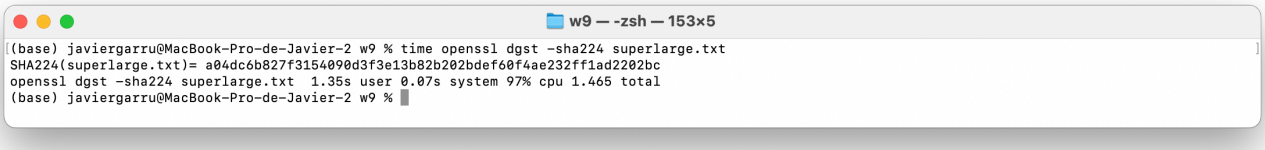
Time SHA1: 0.960s

```
time openssl dgst -sha1 superlarge.txt
```



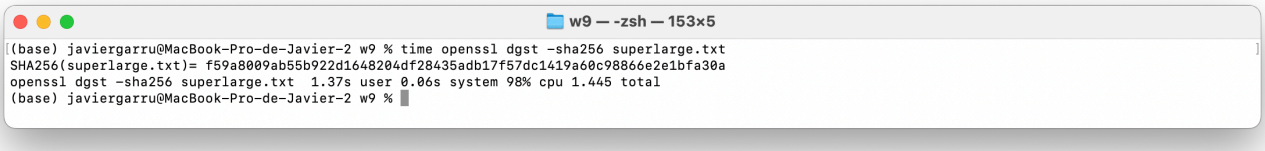
Time SHA224: 1.465s

```
time openssl dgst -sha224 superlarge.txt
```



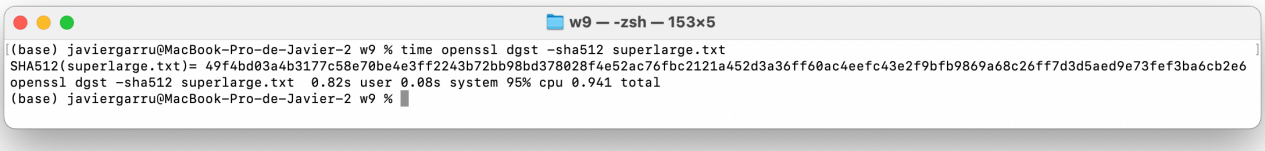
Time SHA256: 1.445s

```
time openssl dgst -sha256 superlarge.txt
```



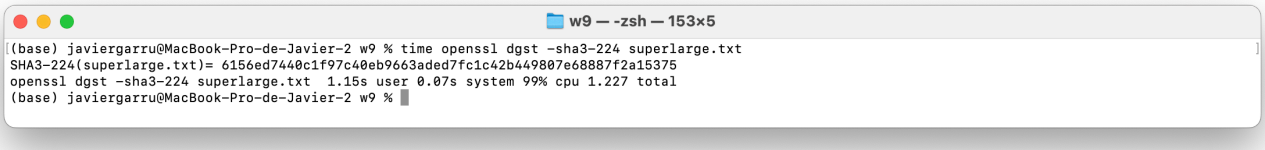
Time SHA512: 0.941s

```
time openssl dgst -sha512 superlarge.txt
```



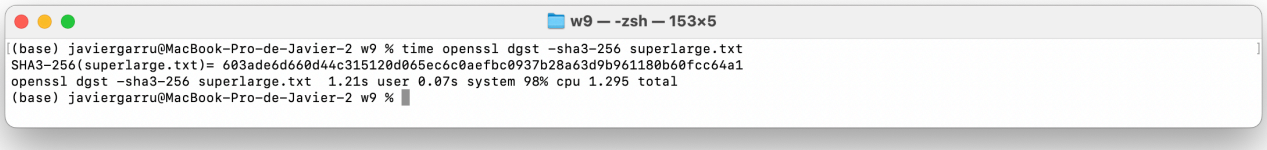
Time SHA3-224: 1.227s

```
time openssl dgst -sha3-224 superlarge.txt
```



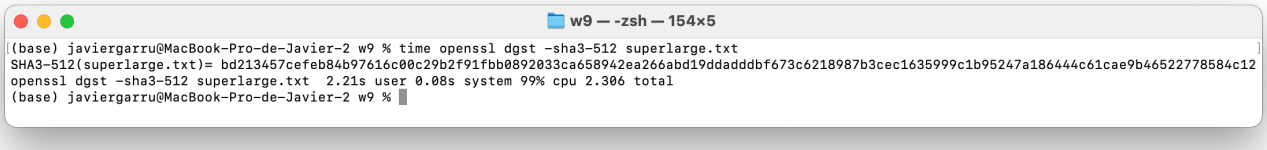
Time SHA3-256: 1.295s

```
time openssl dgst -sha3-256 superlarge.txt
```



Time SHA3-512: 2.306s

```
time openssl dgst -sha3-512 superlarge.txt
```

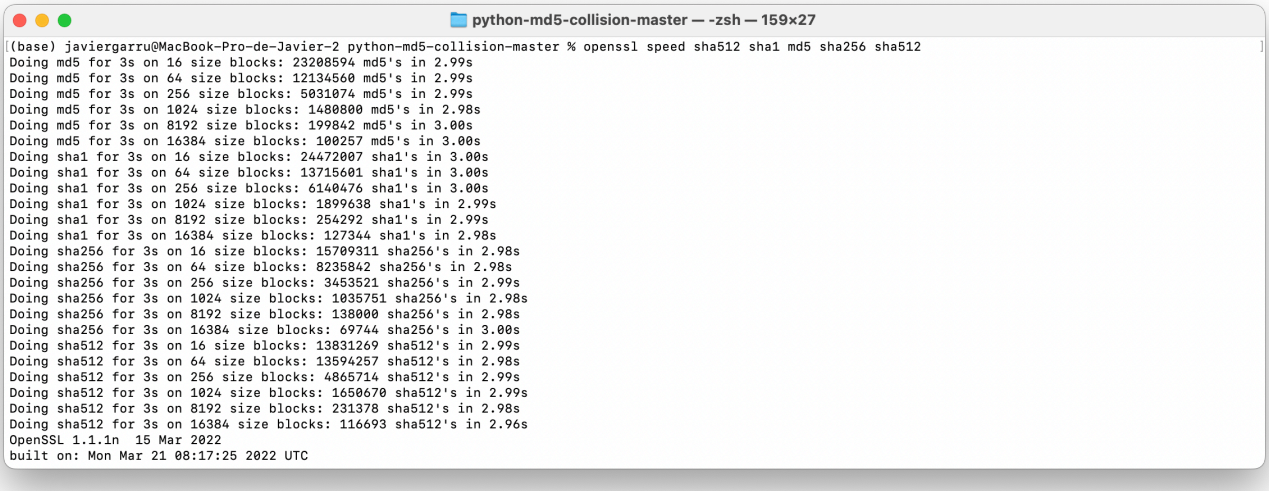


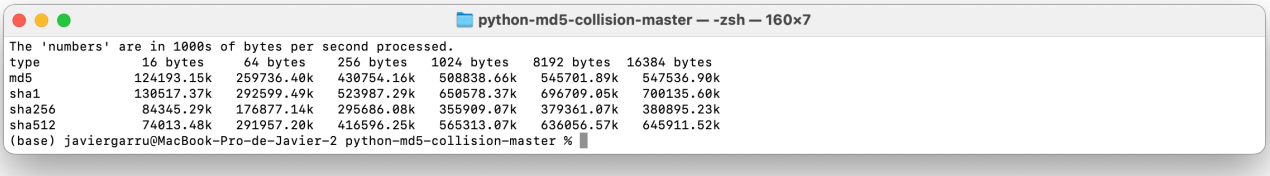
Question: Compare the running time and rank the speed of hash functions, i.e., md5, sha1, SHA-2(sha224, sha256 and sha512) and SHA-3(sha3-224, sha3-256 and sha3-512)? Which one is the fastest?

Function	Speed	Speed rank
SHA512	=50MB/0.941s=53.125 MB/s	1
SHA1	=50MB/0.960s=52.083 MB/s	2
MD5	=50MB/1.169s=42.77 MB/s	3
SHA3-224	=50MB/1.227s=40.749 MB/s	4
SHA3-256	=50MB/1.295s=38.61MB/s	5
SHA256	=50MB/1.445s=34.60 MB/s	6
SHA224	=50MB/1.465s= 34.12969 MB/s	7
SHA3-512	=50MB/2.306s= 21.6825 MB/s	8

Let's have a look at the results given by *time*. Based the fastest one is sha512 closely followed by sha1 and md5.

On the other hand, using the results of *openssl speed*:



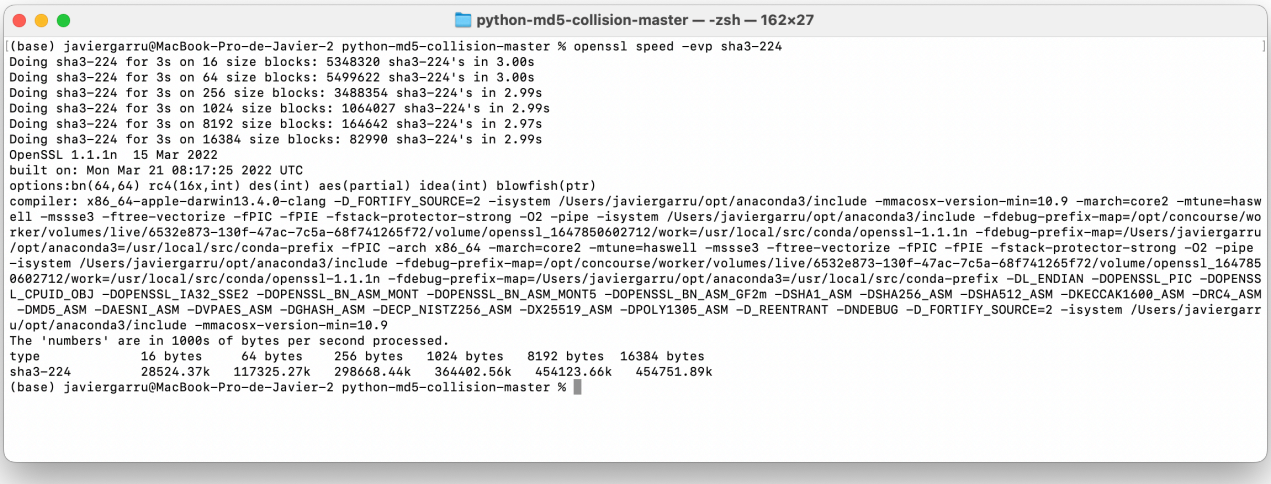


These results show that the speed depends on the size of the file, while in the original test we were using the same file everytime, the new command uses different file sizes and compare the speeds.

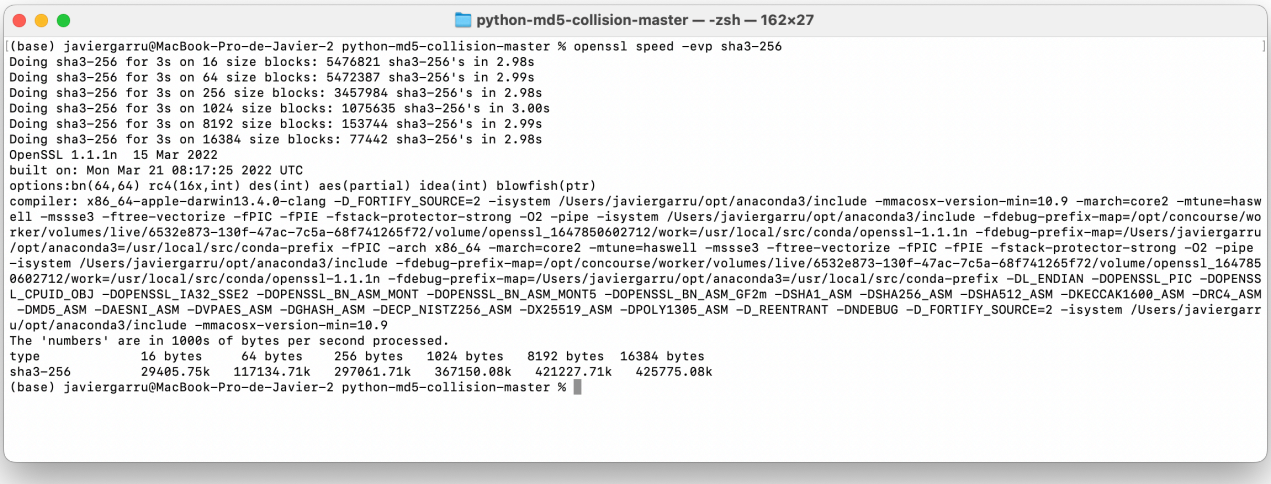
HASH FUNCTION	RANK - 16B	RANK - 64B	RANK - 256B	RANK - 1024B	RANK - 8192B	RANK - 16384B
MD5	1	3	2	3	3	3
SHA1	2	1	1	1	1	1
SHA256	3	4	4	6	6	6
SHA512	4	2	3	2	2	2
SHA3-224	8	7	5	5	4	4
SHA3-256	7	8	6	4	5	5
SHA3-512	6	6	8	8	8	8
SHA224	5	5	7	7	7	7

The rest of components from the table were extracted using `openssl speed -evp ##HASH-FUNCTION-NAME`:

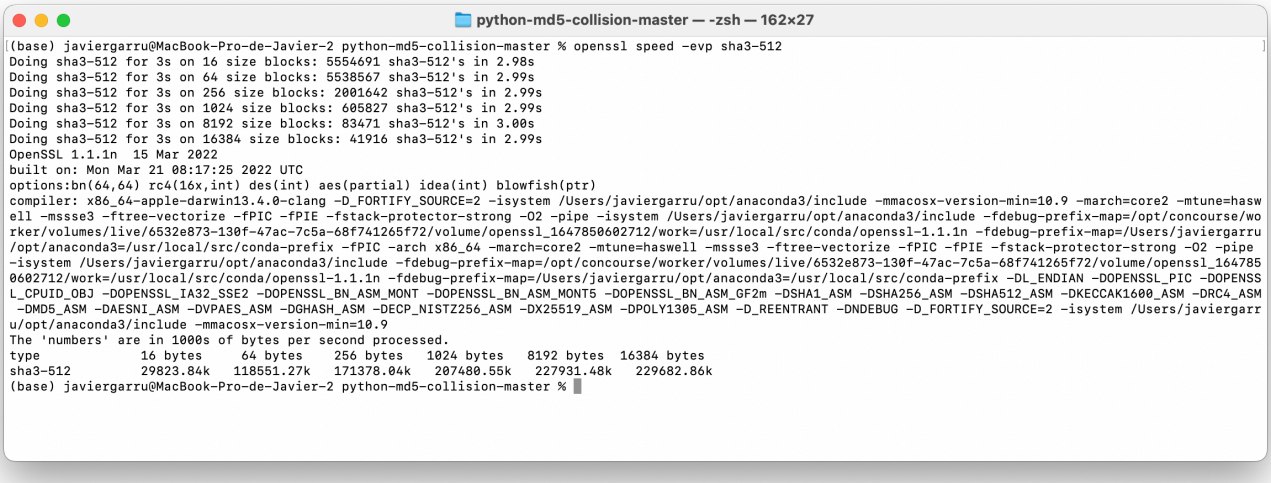
SHA3-224:



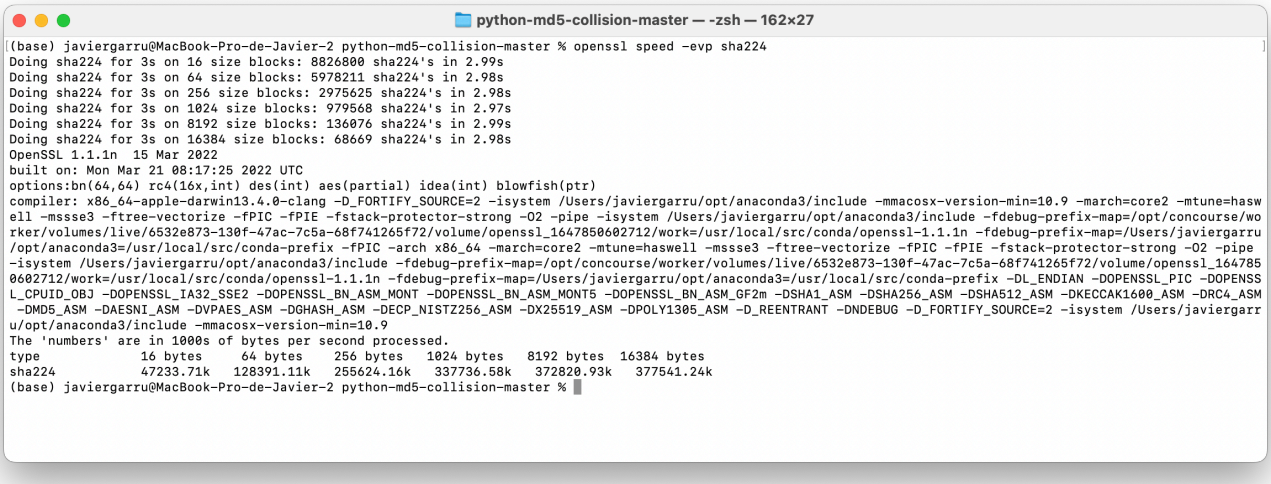
SHA3-256:



SHA3-512:



SHA224:



In general, sha1, sha512 and md5 would be the top three fastest hash functions. Depending on the file size the order could change slightly. For instance, sha512 performs better in larger file sizes.

Part 6: SECURING HASH FUNCTIONS

6.1 Salt

Question: How does the salt scheme improve the security of hash algorithms?

The salt scheme makes brute force attacks and rainbow table attacks considerably more computationally infeasible by attaching a random string at the end of the input of the hash function. This small piece of information ensures that hashing the same input at two different moments will not yield the same result. If this wasn't the case, commonly hashed values would be easily recognizable and used to defeat the purpose of the hashing function. The salt scheme defeats the most common attacks and ensures that being in possesion of a large number of hashed password tables of pairs input-hash doesn't provide any extra power to adversaries when trying to crack new passwords or input values of the hash functions given their associated hash.

Question: How to use the salt scheme?

A salt is randomly generated everytime the hash of a password (or any valuable text input) is going to he hashed. This hash is later concatenated to the password and hashed together. It is important not to reuse the same salt several times as it would demolish its original purpose. Furthermore, the origin of the randomness used to generate the salt must be through a proper source of entropy as if the salt could be guessed it would lose its purpose too. Another important aspect of the salt is its lenght. Using a short salt won't be nearly as helpful as using a longer one. The difficulty of attacking hashed values with longer salts is exponentially increased.

6.2 Strong Hash Functions

Question: What is the vulnerability of MD5 hash function?

The two main vulnerabilities of MD5 hash functions are: collision and preimage attacks. In the first kind of attacks, two different plaintexts can be hashed to generate the same hash. A fraudulent and a normal file could be modified in such way that both their hashes would match implying an adversary could create a fraudulent certificate and modify in such way that its hash matches the hash of a real certificate from a well know CA. This dismantles the whole integrity preservation that supports the PKI.

The other attack consists in finding an input value or plain text to a given hash value. This attack has only been proven theoretically. Although the computational required effort could be achievable in the future the memory requirements are still unfeasible to concieve.

Question: Which hash functions are recommended now?

The most recommended hash function nowadays is SHA-3 in any of its versions. SHA-2 is also a good option although the NIST strongly recommends to leave SHA-1 behind:

> **NIST's Policy on Hash Functions - December 15, 2022**
>
> *December 15, 2022*
>
> NIST is announcing a timeline for a transition for SHA-1. See this announcement for details. After 12/31/2030, any FIPS 140 validated cryptographic module that has SHA-1 as an approved algorithm will be moved to the historical list. NIST recommends that federal agencies transition away from SHA-1 for all applications as soon as possible. Federal agencies should use SHA-2 or SHA-3 as an alternative to SHA-1.
>
> Further guidance will be available soon. Send questions on the transition to sha-1-transition@nist.gov.

Other hash functions like BLAKE2 and Argon2 are recently in popularity as alternatives to the more known SHA standarized functions.

Part 7: LAB SUMMARY

In this lab we have understood the vulnerabilities of the MD5 hash function and how the use of salts its primordial to preserve hash functions' security. We have attacked some hash functions with the use of a brute-force attack and a rainbow table. The different lengths, speed and recommended use cases of the hash functions has been studied and evaluated too.