

# 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 0b011000110111001001110010111000001101000110111
```

Text

Binary

Open File



Paste text or drop text file

crypto



Character encoding (optional)

ASCII/UTF-8

Output delimiter string (optional)

Space

Convert

Reset

Swap

01100011 01110010 01111001 01110000 01110100 01101111

Copy

Save

Save Bin

The obtained output is:

da2f073e06f78938166f247273729dfe465bf7e46105c13ce7cc651047bf0ca4

Let's try with another word like horse:

ruby sha256.rb 0b0110100011011101100100111001101100101

Text

Binary

Open File



Paste text or drop text file

horse



Character encoding (optional)

ASCII/UTF-8

Output delimiter string (optional)

Space

Convert

Reset

Swap

01101000 01101111 01110010 01110011 01100101

Copy

Save

Save Bin

The process is shown iteratively and the final output is:

**fd62862b6dc213bee77c2badd6311528253c6cb3107e03c16051aa15584eca1c**

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

```
final hash value: (H1)
-----
a = 11111101011000101000011000101011 = fd62862b
b = 01101101110000100001001110111110 = 6dc213be
c = 111001110111110000010101110101101 = e77c2bad
d = 11010110001100010001010100101000 = d6311528
e = 00100101001111000110110010110011 = 253c6cb3
f = 000100000111111000000011111000001 = 107e03c1
g = 01100000010100011010101000010101 = 6051aa15
h = 01011000010011101100101000011100 = 584eca1c
```

**fd62862b6dc213bee77c2badd6311528253c6cb3107e03c16051aa15584eca1c**

The different functions used during the process can be individually 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 = 111010011011010111011011110100101  
4 = 00111001010101101100001001011011  
5 = 01011001111100010001000111110001  
6 = 1001001000111111000001010100100  
7 = 10101011000111000101111011010101  
8 = 1101100000001111010101010011000  
9 = 00010010100000110101101100000001  
10 = 00100100001100011000010110111110  
11 = 01010101000011000111110111000011  
12 = 01110010101111100101110101110100  
13 = 10000000110111101011000111111110  
14 = 1001101111011100000011010100111  
15 = 11000001100110111111000101110100  
16 = 11100100100110110110100111100001  
17 = 11101111101111100100011110000110  
18 = 00001111110000011001110111000110  
19 = 0010010000011001010000111001100  
20 = 00101101111010010010110001101111  
21 = 01001010011101001000010010101010  
22 = 01011100101100001010100111011100  
23 = 01110110111110011000100011011010  
24 = 10011000001111100101000101010010  
25 = 10101000001100011100011001101101  
26 = 10110000000000110010011111001000  
27 = 1011111101011001011111111000111  
28 = 1100011011100000000101111110011  
29 = 11010101101001111001000101000111  
30 = 00000110110010100110001101010001  
31 = 000101000010100100101001011000111  
32 = 00100111101101110000101010000101  
33 = 00101110000110110010000100111100  
34 = 01001101001011000110110111111100  
35 = 01010011001110000000110100010011  
36 = 01100101000010100111001101010100  
37 = 011101100110100000101010111011  
38 = 10000001110000101100100100101110  
39 = 10010010011100100010110010000101  
40 = 1010001010111111110100010100001  
41 = 10101000000110100110011001001011  
42 = 11000010010010111000101101110000  
43 = 11000111011011000101000110100011  
44 = 11010001100100101110100000011001  
45 = 110101110100110000011000100100  
46 = 1111010000011100011010110000101  
47 = 00010000011010010101000001110000  
48 = 00011001101001001100000100010110  
49 = 00011110001101110110110000001000  
50 = 00100111010010000111011101001100  
51 = 00110100101100001011110010110101  
52 = 00111001000111000000110010110011  
53 = 01001110110110001010101001001010  
54 = 01011011100111001100101001001111  
55 = 01101000001011100110111111110011  
56 = 01110100100011111000001011101110  
57 = 0111100010100101100011011011111  
58 = 10000100110010000111100000010100  
59 = 10001100110001110000001000001000  
60 = 1001000010111110111111111111010  
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
```



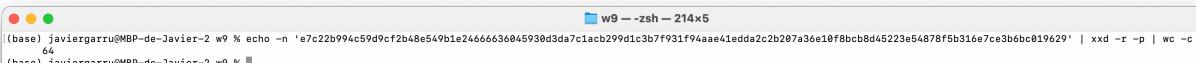
w9 --zsh -- 152x6  
(base) javiergarru@MBP-de-Javier-2 w9 % echo "hello" > small.txt  
cat small.txt  
openssl dgst -sha512 small.txt  
hello  
SHA512(small.txt)= e7c22b994c59d9cf2b48e549b1e24666636045930d3da7c1acb299d1c3b7f931f94aae41edda2c2b207a36e10f8bcb8d45223e54878f5b316e7ce3b6bc019629  
(base) javiergarru@MBP-de-Javier-2 w9 %

```
e7c22b994c59d9cf2b48e549b1e24666636045930d3da7c1acb299d1c3b7f931f94aae41edda2c2b207a36e10f8bcb8d45223e54878f5b316e7ce3b6bc019629
```

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

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

```
echo -n  
'e7c22b994c59d9cf2b48e549b1e24666636045930d3da7c1acb299d1c3b7f931f94aae41edda2c2b207a36e10f8bcb8d45223e54878f5b316e7ce3b6bc019629' | xxd -r -p | wc -c
```



w9 --zsh -- 214x5  
(base) javiergarru@MBP-de-Javier-2 w9 % echo -n 'e7c22b994c59d9cf2b48e549b1e24666636045930d3da7c1acb299d1c3b7f931f94aae41edda2c2b207a36e10f8bcb8d45223e54878f5b316e7ce3b6bc019629' | xxd -r -p | wc -c  
64  
(base) javiergarru@MBP-de-Javier-2 w9 %

### 3.3 SHA512 on a large file

```
w9 -- zsh -- 141x20
(base) javiergarru@MBP-de-Javier-2 w9 % head -c 16000000 /dev/random > large.txt
(base) javiergarru@MBP-de-Javier-2 w9 % ls -la
total 55948
drwxr-xr-x 16 javiergarru staff      512 Apr 30 20:30 .
drwxr-xr-x@ 12 javiergarru staff      384 Apr 30 19:39 ..
-rw-r--r--@ 1 javiergarru staff     8196 Apr 30 19:43 .DS_Store
-rw-r--r--@ 1 javiergarru staff    2272 Apr 30 20:23 Cryptography - Lab 5 - Asymmetric Crypto.md
-rw-r--r--@ 1 javiergarru staff   3900509 Apr 30 19:38 Hash Functions.docx
-rw-r--r--@ 1 javiergarru staff   540287 Apr 30 19:38 MD5.pdf
-rw-r--r--@ 1 javiergarru staff  223901 Apr 30 19:38 Salt_1.pdf
-rw-r--r--@ 1 javiergarru staff  1146348 Apr 30 19:38 Salt_2.pdf
-rw-r--r--@ 1 javiergarru staff    129 Apr 30 20:25 first_sha
-rw-r--r--@ 1 javiergarru staff 16000000 Apr 30 20:30 large.txt
-rw-r--r--@ 1 javiergarru staff   16164 Apr 30 19:38 password (1).lst
-rw-r--r--@ 1 javiergarru staff  11434 Apr 30 19:38 python-md5-collision-master.zip
drwxrwxr-x@ 29 javiergarru staff      928 Aug 18 2021 sha256-animation-master
-rw-r--r--@ 1 javiergarru staff  5464529 Apr 30 19:38 sha256-animation-master.zip
-rw-r--r--@ 1 javiergarru staff    265 Apr 30 19:38 sha512crack.sh
-rw-r--r--@ 1 javiergarru staff      6 Apr 30 20:22 small.txt
(base) javiergarru@MBP-de-Javier-2 w9 %
```

```
w9 -- zsh -- 159x5
(base) javiergarru@MBP-de-Javier-2 w9 % openssl dgst -sha512 large.txt
SHA512(large.txt)= 61f37db1f31363aaea005ece582d8c56819584b0c16ea4160e8170e06320fa25bb0b726068968810ee413d14227161c742b4e5586bba03ddc04199da20bf640c
(base) javiergarru@MBP-de-Javier-2 w9 %
```

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
```

```
w9 -- zsh -- 214x5
(base) javiergarru@MBP-de-Javier-2 w9 % echo -n '61f37db1f31363aaea005ece582d8c56819584b0c16ea4160e8170e06320fa25bb0b726068968810ee413d14227161c742b4e5586bba03ddc04199da20bf640c' | xxd -r -p | wc -c
64
(base) javiergarru@MBP-de-Javier-2 w9 %
```

## 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:

```
w9 -- zsh -- 214x22
(base) javierarru@MBP-de-Javier-2 w9 % echo "hello" >> small.txt
(base) javierarru@MBP-de-Javier-2 w9 % head -c 16000000 /dev/random >> large.txt
(base) javierarru@MBP-de-Javier-2 w9 % ls -la
total 85776
drwxr-xr-x 17 javierarru staff 54 Apr 30 20:53 .
drwxr-xr-x 12 javierarru staff 384 Apr 30 19:39 ..
-rw-r--r--@ 1 javierarru staff 891 Apr 30 19:39 DS_Store
-rw-r--r--@ 1 javierarru staff 3756 Apr 30 19:39 Cryptography - Lab 5 - Asymmetric Crypto.md
-rw-r--r--@ 1 javierarru staff 3900659 Apr 30 19:38 Hash Functions.docx
-rw-r--r--@ 1 javierarru staff 540287 Apr 30 19:38 MD5.pdf
-rw-r--r--@ 1 javierarru staff 223981 Apr 30 19:38 Salt_1.pdf
-rw-r--r--@ 1 javierarru staff 114634 Apr 30 19:38 Salt_2.pdf
-rw-r--r--@ 1 javierarru staff 151 Apr 30 20:39 first_sh
-rw-r--r--@ 1 javierarru staff 1401 Apr 30 20:39 first_sh.sh
-rw-r--r--@ 1 javierarru staff 3208000 Apr 30 20:53 large.txt
-rw-r--r--@ 1 javierarru staff 16144 Apr 30 19:38 password (1).lst
-rw-r--r--@ 1 javierarru staff 11434 Apr 30 19:38 python-md5-collision-master.zip
drwxrwxr-x@ 29 javierarru staff 928 Aug 18 2021 sha256-animation-master
-rw-r--r--@ 1 javierarru staff 5464529 Apr 30 19:38 sha256-animation-master.zip
-rw-r--r--@ 1 javierarru staff 264 Apr 30 19:38 sha12crack.sh
-rw-r--r--@ 1 javierarru staff 1 Apr 30 20:53 small.txt
(base) javierarru@MBP-de-Javier-2 w9 %
```

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

```
w9 -- nano small.txt -- 214x22
File: small.txt
Hello
hola

Hello
hola

^C Get Help      ^A WriteOut    ^R Read File   ^V Prev Pg     ^X Cut Text
^X Exit          ^S Justify      ^W Where is     ^N Next Pg     ^U Uncut Text
^T Cur Pos       ^I To Spell
```

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

```
w9 -- zsh -- 214x6
(base) javierarru@MBP-de-Javier-2 w9 % openssl dgst -sha512 small.txt
SHA512(small.txt)= a649a1bad8e015009e5943b924ea2fbd50a28d5d9ce96455fff14a7d27503f506e8317ff1d073af6f98ee887f64bc784556839ad75de49ca1b88070936caa598
(base) javierarru@MBP-de-Javier-2 w9 % openssl dgst -sha512 large.txt
SHA512(large.txt)= 2cf3ad52ef2b0544dea90845aec6509961515a717cd946dfeefb934ffd918b86ccdbc493a7a30f6e0c1acc1
(base) javierarru@MBP-de-Javier-2 w9 %
```

```
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
```

```
w9 -- zsh -- 200x21
(base) javiergarru@MBP-de-Javier-2 w9 % echo -n "1" >> large.txt
(base) javiergarru@MBP-de-Javier-2 w9 % ls -la
total 85784
drwxr-xr-x 17 javiergarru staff 544 Apr 30 20:59 .
drwxr-xr-x@ 12 javiergarru staff 384 Apr 30 19:39 ..
-rw-r--r--@ 1 javiergarru staff 8196 Apr 30 19:43 .DS_Store
-rw-r--r--@ 1 javiergarru staff 5871 Apr 30 20:59 Cryptography - Lab 5 - Asymmetric Crypto.md
-rw-r--r--@ 1 javiergarru staff 3980589 Apr 30 19:38 Hash Functions.docx
-rw-r--r--@ 1 javiergarru staff 540287 Apr 30 19:38 MD5.pdf
-rw-r--r--@ 1 javiergarru staff 223901 Apr 30 19:38 Salt_1.pdf
-rw-r--r--@ 1 javiergarru staff 1146348 Apr 30 19:38 Salt_2.pdf
-rw-r--r--@ 1 javiergarru staff 151 Apr 30 20:39 first_sha
-rw-r--r--@ 1 javiergarru staff 148 Apr 30 20:39 first_sha.txt
-rw-r--r--@ 1 javiergarru staff 3288000 Apr 30 20:39 large.txt
-rw-r--r--@ 1 javiergarru staff 14164 Apr 30 19:38 python-md5-collision-(1).lett
-rw-r--r--@ 1 javiergarru staff 11434 Apr 30 19:38 python-md5-collision-master.zip
drwxrwxr-x@ 29 javiergarru staff 928 Aug 18 2021 sha256-animation-master
-rw-r--r--@ 1 javiergarru staff 5464529 Apr 30 19:38 sha256-animation-master.zip
-rw-r--r--@ 1 javiergarru staff 265 Apr 30 19:38 sha512crack.sh
-rw-r--r--@ 1 javiergarru staff 11 Apr 30 20:53 small.txt
(base) javiergarru@MBP-de-Javier-2 w9 %
```

```
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?

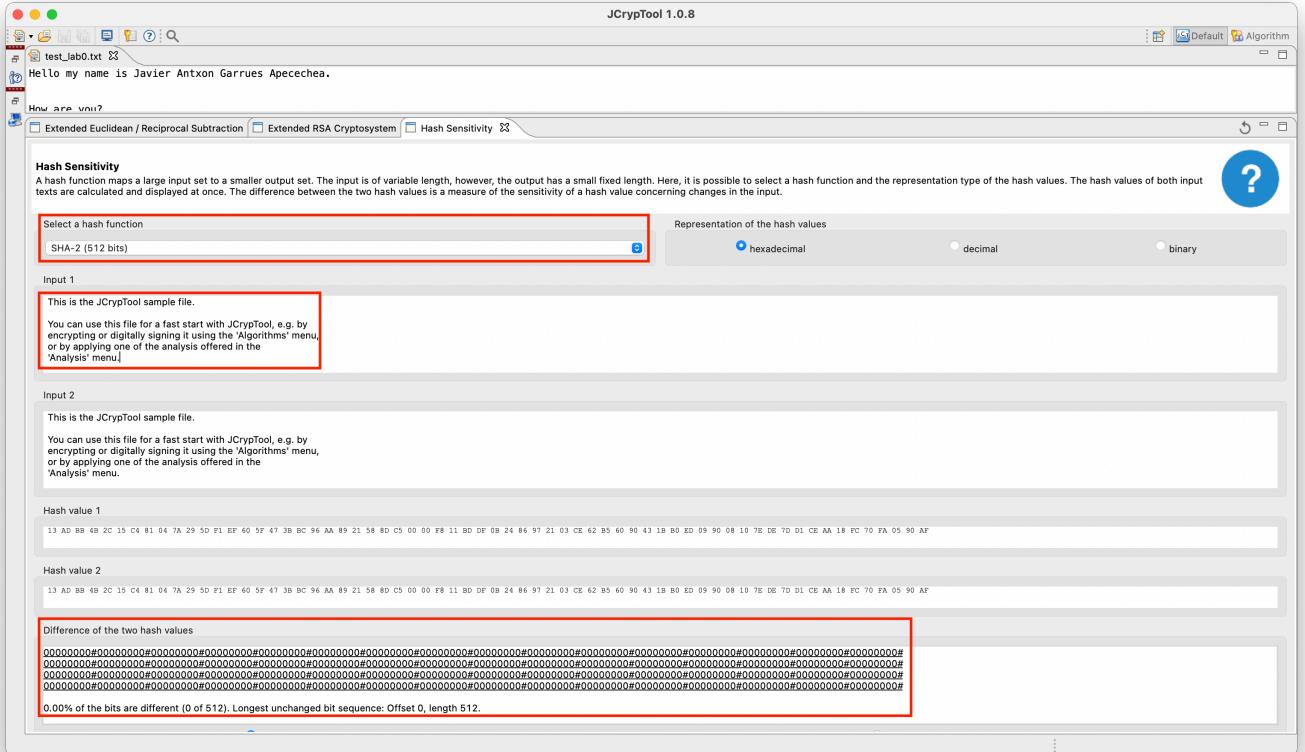
Both hash values are 512 bits again. The new hash value is significantly different from the original one, even though we have only added a bit to the file as the command `ls -la` showed on the file size. This shows how powerful the hash algorithm used is.

### 3.5 Hash Sensitivity Visualisation Using JCrypTools

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

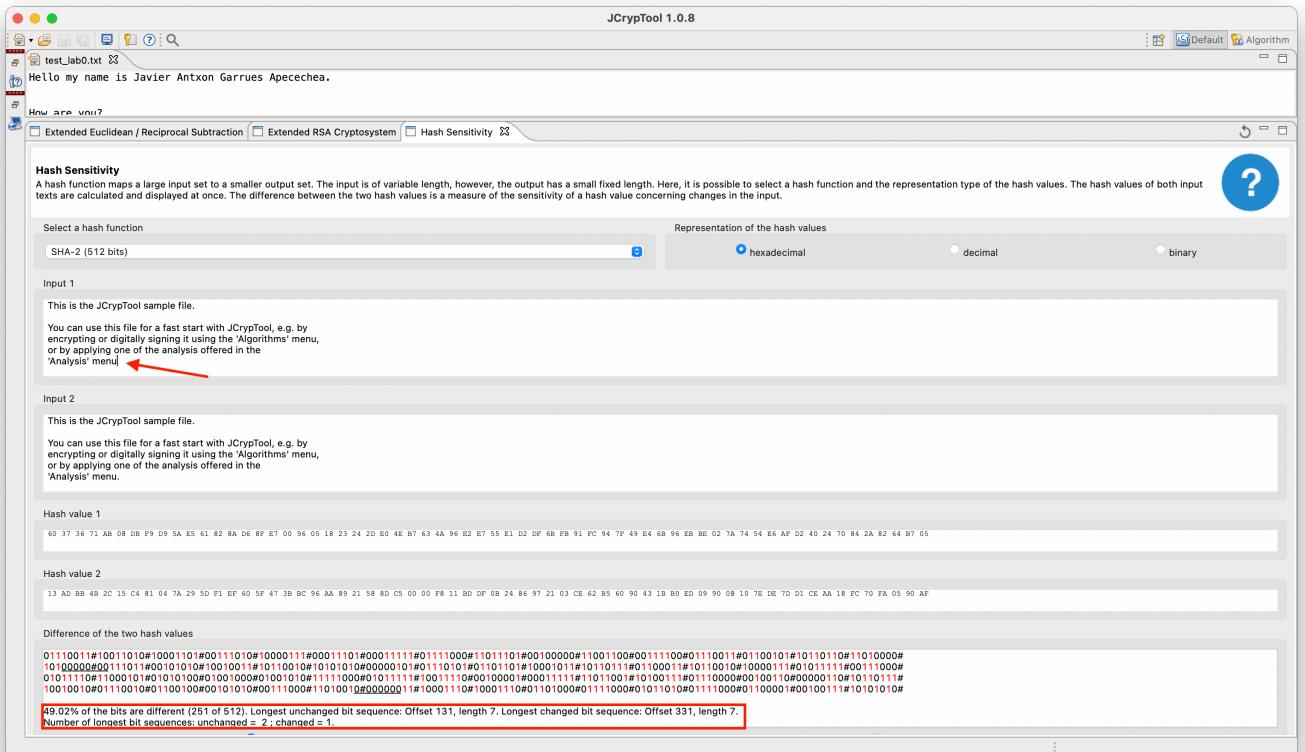
- Android Unlock Pattern (AUP)
- Ant Colony Optimization (ACO)
- ARC4 / Spritz
- Certificate Verification
- Chinese Remainder Theorem (CRT)
- Diffie-Hellman Key Exchange (EC)
- Elliptic Curve Calculations
- Extended Euclidean / Reciprocal Subtraction
- Extended RSA Cryptosystem
- Hash Sensitivity
- Homomorphic Encryption (HE)
- Huffman Coding
- Inner States of the Data Encryption Standard (DES)
- Kleptography
- McEliece Cryptosystem
- Merkle Signatures (XMSS<sup>MT</sup>)
- Merkle-Hellman Knapsack Cryptosystem
- Multipartite Key Exchange (BD II)
- Multivariate Cryptography
- Public-Key Infrastructure
- Redactable Signature Schemes (RSS)
- Shamir's Secret Sharing
- Shanks Babystep-Giantstep
- Signature Demonstration
- Signature Verification
- Simple Power Analysis / Square and Multiply
- SPHINCS Signature
- SPHINCS+ Signature
- SSL/TLS Handshake
- Verifiable Secret Sharing
- Winternitz OT-Signature (WOTS / WOTS+)
- Zero-Knowledge: Feige Fiat Shamir
- Zero-Knowledge: Fiat Shamir
- Zero-Knowledge: Graph Isomorphism
- Zero-Knowledge: Magic Door

And then proceed to modify the original input1:

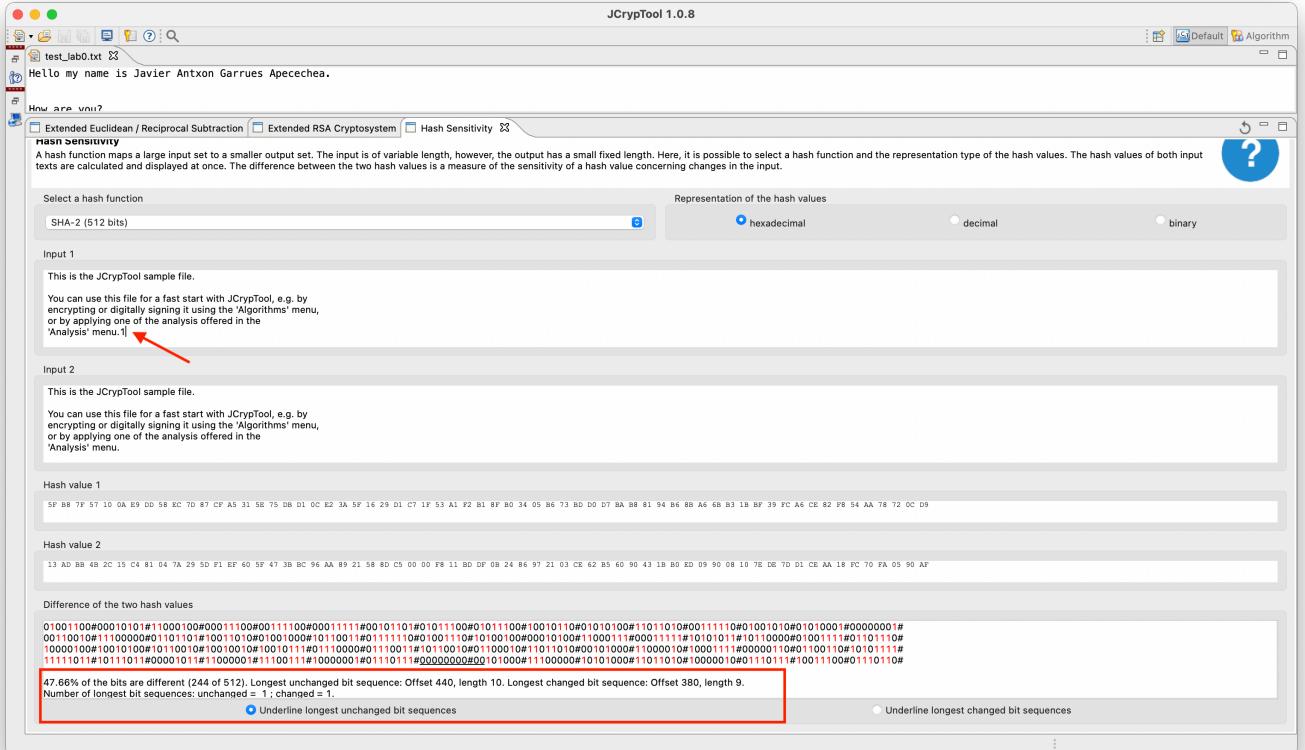


Initially there are no difference between the two hash values.

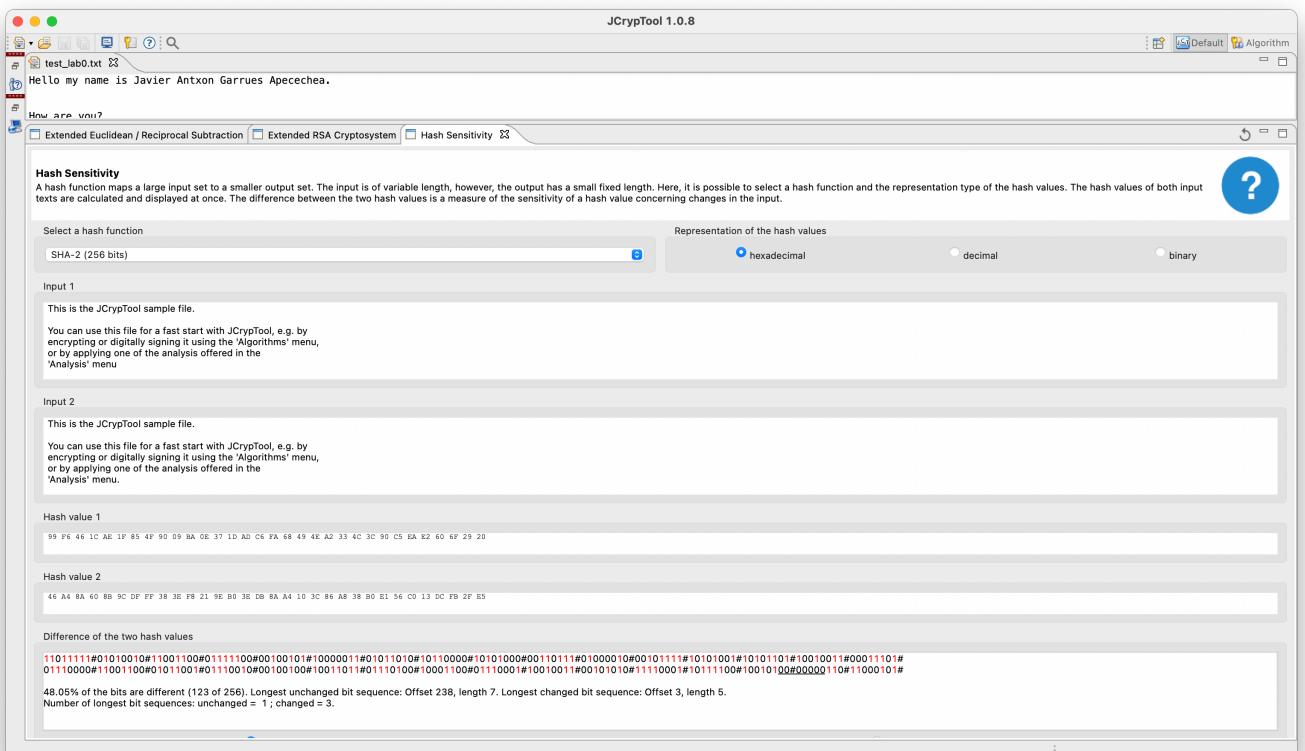
Then, we eliminate the last full stop from input 1 and compare the two hash values:



**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 message. Once the receiver gets the message, its hash is recomputed and compared 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

**ab1c2b83a2d9b0304541de63f302a9580c0b4252dc70d5a6ce0bd5b8829d2b3bc1f727c701  
40113071bcc7f16a737a74a6f0f5166168185dd43bbba690b5041.**

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  
"ab1c2b83a2d9b0304541de63f302a9580c0b4252dc70d5a6ce0bd5b8829d2b3bc1f727c701401130  
71bcc7f16a737a74a6f0f5166168185dd43bbba690b5041"? 
```

```
w9 — openssl - sha512crack.sh ab1c2b83a2d9b0304541de63f302a9580c0b4252cd70d5a6ce0bd5b8829d2b3bc1f727c70140113071bcc7f16...
```

try house ebb82e73fe87f82ff6cc690a8e62badf3318d9bb5d2a1909564b313782c1ba374be4bce7c0cd9548c84fb00f190d89c75c43847366977ed4a6119c0753c3c  
try jackie 83017d8a00f15fc340a9965310a4ea80d3e881e15b3f3e27f5327fa6564406f192f9a908bcd56cb349cea790b0eb79ad06b972758e6dc238b92472e6943c86  
try jean 6ff5abc59eb4b8b4ad9a2ea2e688bx311fa01c4affec037fe1d56f615d1dbb5e0f7d3fbda1b5c130e91d9a212b79b402d0380174eaecb9c62650p09841ee51  
try jenny e285c70bf5a37c45aa276de2440e3b9e1d23d1e25e9972e97be344abe95efb6c2c2a3b295b64bf3d2e1d4a1616a8f0a394bbaeaabe7700f1cb58abada  
try joey 4492d0076b1e7c9b301962f7b45138982273f81af1f631f98fc84fedf6d266f53e240a1cb9a7244681116d3a600b3002bdcce2459748e4c00684f8a9f84e0e4  
try kelly 13450bada8acbf81236f220527a654320b6b379c2b6e29d6cb6a7a9c616e5d54e9c22656ca365a073c767a8c1a7883a78177802e05e09379141c21f3427918  
try laura 4b591f93c712931e3fb91a2d0f9199735b7c9c430c903e8442e19e2bf34a7f5bf3e48714359bf44902d942fc51153992a8d989d6beab7143d2425fa325782  
try lauren 53c41e63aa0f425aa4352336e635daa5d43ab9ff0031f36abd0058228167c954b11fec3618deb4f6be131783368f75b5be203c7bbf1216c1528b10585e90f06b6e  
try lincoln cb76b28ab3c2b34d20269861cccdad57a349d60f0842fd0bc16a837127d0a62cb36d905b5b0c4597e90c6346baad8c43c7d58be358077ff06079d6a549594  
try loveme 5ae28b6f2d0e8bb0d0af8fea20e399fe40141eaec741227f7ddbc2e143c22d026cd0dd1a277f7da8ca02b74b05a98aef82f9c8ead18d4  
try margaret 62815661a1898e37d0d3f18685d32ba989e9f12b1299165db8b3ed224d5c0e48222a6f6c543ba563fc8de4e008950182f9cc86a351887812569d8847850  
try mary 925ece1aeee8468e8921e1a00b889a8895205f2ad3569b38d1f1d68a909156e893ac498c279ac02799fe2ad669d7  
try max f89b9b327584fa388e74b77559f85af9285cd8c48d4c4117e5649514dc236b6c5490ne0be04cc90c7d270d6422f53467da7564631189e2f299fe2ad669d7  
try mercedes 8379ad313ef4d22ec6de652b7db3e13d0e101ac7c8b416de2b0825c69287f9a9ecbf6ba18ba99e0ace77801599a3854ea2c281f97f6fb13b952e89896c4  
try mercury 8e4972d068c57690db1db3098f744cbe8cc376d0eda717d31e48927917bcb09163850c7e87a1635ebfad11584e2583f66db12d52a26cc2a4ebff002681  
try michel 120977defc131474ad6c9d09d97e7ecb9c17ced12c01ca93d700f37d53f5e97b0c26fe9bc48651b525234bcfa1e05c9b8725669e269135f774ee274e4  
try midnight c478c1abbd6a2d9481ad22afe1c9b2a6085c8a30e7b8e9fb5392cf3d7f694132a9bfaece788f689d028292b889db38eee24ae8841594d8b4959befdfcad1c1f  
try mine d27274ae7483db0059b54579723680578804995f6c2068cb84840865b2d066caa99995853c6d38805cef66c286866b0194d064685c769cab10e97461664d3  
try mirror 23651b30dsf44edc975d4c01c4de99308713bd13450e2h1558c94752cbae811287c300fe1f66e8e84c7e128447e9c60401b692ba6b471565e5658e499a43f8  
try mozart 49bb59a77dbfaaf5d5c824626db1c56094888a0fbcc62644e92dc4da2b2e0281fd4d17b38920be3fe3a98420a19511c3010a0e7712b054daef5b5f7ad59ecbd93b3280f210578f547f4aed4d25  
try nicholas 4dfc4f3ca31220a70bfaf9e1a96fb73cfea0d61f664bd73cbfa08086339e984644c31522181a39c93f076e4d489c89988181dd46290f50ab4a7a0c675eb724cb  
try nothing 1d6c61c1f237e2664f242b96d5fae55e0fb323771723d76fa41d0a6e722c45cafefb0951f43309fc6bc852b98a5496d3c2909b606688a882d43c6fb905162b10f  
try oliver b4e6749fe6bc93783b33a14a4b1e795f71fd66ace13f7fe66bf8e6337bbedbc37ecef88355421e88d761e1b9a53e7e95152ba39a2961d5a1f4b3d39c2  
try packard 04e134acb4394037dee7198e89639c47964a60b65e8e5a4136e5705b53c0e460c4e75502b3b967325e149877d9b276b959aa88a40e879351cf9ff5ec83fc732  
try pass 5b722b307fc6e944905d132691054e4a2214b7fe92b738920be3fe3a98420a19511c3010a0e7712b054daef5b5f7ad59ecbd93b3280f210578f547f4aed4d25  
try peace 0598c279b277432b8e04dcfefa7045cd8e4388ab03ca9d50513acc1d954468e2c8a6498755149ad7841f436c4a8d489a46d54e4ddaaeae89d7e856d408a0f6  
try phil 055244bd6fc27c55e1ee33f71a0fd87e32bd9b1d19bd7ed636a6588b375a1e9664a45c3e14707d4e45ed35a13cd95fa7d420fa52cfcbb190e0497c9dbc33fb  
try porsche 9ede80a744a895e8cb26ed1c78f2d7964a3a19845c972897c226285e2858d2b7e40ef9e322cb4920c5c70f8b4a2404744e988357726210e8ed0c9fb6976f  
try psycho 11fa7bae317a093d5cbcbfb1f45ec20624e3de0488e0c1d5afbf02c0a0f0fd5ac37dadf41a90ad7f2993bf85f02d456e1852c0a75b90f50731f9d5808939  
try pumpkin 011aa27183f48c298da63dce16f5c9f9e98e13b9ce5e5a08e58fdb338cc8fc572068c00127d4a13e275b8c9349207f45f360bdc8a98b0f32203291181eb  
try punkin f1020c50b1bd16de57c3880f8ac786942f1fbbeb04694f83a4577fc25214f77f18981160a18a71e93d00ea5b1074954749fc357e0d7d02d3e262bd05b46780  
try puppy123 4d8e4a881412220a4912a1b1a892d42b11705845f945071be8c1953f735e7b953d915c113ef37c4b2e0cde0a1b5c4f39b8e2664a8897fd52a27a3c2565  
try randy 47eab3e02653b3353161466db8a3198b301fa69420a19511c3010a0e7712b054daef5b5f7ad59ecbd93b3280f210578f547f4aed4d25  
try remember 27bf6e889c9e2adebea1f49a7483c84215a26017e2d39fdead92d0446ff3bc9dec098ec9feff55160646ec7889b2d973af2097ed72a047284b1c01c64645f  
try robin 071b38098125700fe49e650f059bfa1f2bd110472a8d7490e0b3af6a8260712f958b9c7f27be61a8ddfe4ecbf719f8002360f894f06dd4b1f383963d71d41  
try rosebud ecf1536b1e65151f47070a01e6ac997aece0a3190c703e8b852cc7efbf8face7b49a1058377d4f913be4b5a40a85f8fb1c3159937eca00982cdf22a500baaf8  
try sadie 5dc32edbe35fbdbab3efcad794da8d13b6f2afc0fd3b5db2ab99580b1b1a80f1283fc882d20710bc271a3137700eec3460e0a2db9bb49c1f17b3bbf5a2a450d7  
try sampson 1a63f5310e3640972a78a6fd10b47dca89530ae296539b7368f6e0999dc68c08bf5feafa3dc0b574911cf7154f4fe1c663a21343b40f491dbbefc447b06764

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.

The screenshot shows a terminal window titled "sha512crack.sh". The window contains a single file named "sha512crack". The code is a bash script that reads from a file named "password.lst" and uses openssl dgst -sha512 to find a matching hash. When a match is found, it prints the plaintext and exits. The script starts with a shebang "#!/bin/bash" and ends with "echo \"search end\"". The terminal status bar at the bottom right indicates "Line: 1 Col: 1".

```
#!/bin/bash
echo "trying password.lst"
file=password.lst
while IFS= read -r line
do
str=$(echo -n $line | openssl dgst -sha512)
echo "try" $line ${str:9}
if [ "${str:9}" = "$1" ];
then
    echo "plaintext is" $line
    break;
fi
done < "$file"
echo "search end"
```

**Question: What is the SHA-512 preimage of  
"ab1c2b83a2d9b0304541de63f302a9580c0b4252dc70d5a6ce0bd5b8829d2b3bc1f72  
7c70140113071bcc7f16a737a74a6f0f5166168185dd43bbba690b5041"?**

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

```

w9 --zsh-- 145x39
try toronto 4e9be86ebde926770923e7b8c3d07e600fa672cc83aaabb1c8d063e8efaf560bd6935ef4c38d5945390e894cf20db82e53268f8d7dbcb190b625cd4962aa3d47c
try tracy 1fc58a0f00da6e8bb6bc6c7d9e1838t60f800aaeb50fe39ca7c7c6d06e9e0c867178795eccc7d5ed7bae7810fa4eacce3bb1835b86c25af80f82473e7b121315d6
try trel a0fd26f71ab7c90859bf759c0e178cce26805f5f8dcd3cae6d2ea6f7a9d9832fb7e53a6c52cbf569089445fb1bb4ccf969ee7598fb9052c2c22f87482fb9fce
try trident d96b83272d363698828a943a66a59d84886ce44320953947766e36c6b60148d67643c9250fd9d3196014ae14d0c24cedb5c6ad5f89a0c68935bec63c9ad40
try trumpet 1f0dc133e1dd50c5aca3c904fdf4552e8291e5abd04729617f78fec8ad36542b4f6b673f15a2db84355fe98afe114dcf5890cf005f5569a11c535f98536
try turbo fbdd2f0c7d6dfb2796d10fcd3649762534274541c1f909f91d8b1057ad2089b1487d1aa22fd36f0a9bcfd833534e92efb22a8f238682bddae811271dcc307ad
try twins 029525e97df0f87f6e515bc66434d709360bed1258121e0b81d98adcba94efffc7c7938e80372c2c8678d6945b5978b1d879192ddde24d62dfe769b9307f87
try user1 9ec62c20118ff506dac139ec30a521d12b9883e55da92b7d9addee09ed4e080d152e2a099339871424263784f8103391f83b781c432f45eccb3e18e28669d2f
try utopia 809668e356940bb9aabfe54b3e17e3aa89a3458683315adad7ca5eb9da19b75b673df2a4f684158a09dac231fc4e629d45a05680e157966887daed165e
try valentin e52a07b501623e5c6faa36a08420004a10f7e5945492d605f8f774766fb85539a2f819db407c3d94d9e13e080a36976e2122b45e1f6727c406e10ee5f531
try valhalla ce73a83b90223fd3c075dee4ff2a62bde05f6d60438f20046a6f3f4dcba341986d7c9f54749cf0e6b5c5f6b488261caea0d7b713492a0f7e1017ab699
try velvet 82c15d0e62767f50f42515601e078d92d5b50507e98a104c4887c5d179e247121c6e4ce267424579b4c88274a853917fb1240c379ad1a6c5f0714f6a32073
try venus cd596c282382d1e246b129b7dcf9028b81e6fe62ac6912f92f74b8f9c22adb9eef49d2a48921026a981e9c07a8aad53f3e936137f5270492275808f721dcdb
try vermont c3d91a45a485cb53f8b5886546995d4f4cc1fce26897c6d724b504abc337f0928a27943b295b4c2146fe5e0509a32a962b40d6a4ac75f7808e46fe4821120
try vicky ff9b999cbe5356b094241465f00311b6cc6d68b7d8e8b1a5f1c20b6b684c723347b18d5ac9e0d4b4f27b6d3a3839dc752929333f9b304b94a6d2a098146
try wendy 656006cdfb59cb1dd65dd4f922b64f9b7be70623d4aaa02a0a5f3df8e66b1ae63ac38a63c1b5f750c11840a090cb864f46447965d1d5bcbad41a9ed801e
try wanker 7de922febcadc4acf7366fe171dea0f78a409964094a20f8e989b2adfa52c8a7fe3718f4c01e5455e03c6e6b604d5635323f7791bc79b9d9abc8d92f91d325
try warriors 150fe5c5b2653ec71341d4e2ea280a7ca0dd8a0618dd0d5b1692b5a6e7f462d62cd5f4d87bb0348370b6e22c51c17e2347e0a008c391c895615f0b954a703
try whitney aaale93b3a091d3e095b7bd4691007627b157119ad6894bfe441ab182406765fb5a085f5c1305b0ca4f15f127a42cdcb3e3d9788c8294946c6c758c8e8ee
try wolfman 4100d0909f0d293ee769e69ed45e07b37a8376464a121f29850a03d8b8a1b48432846a780f2299e083eda961e3a3b317092a3b386deaa9457ebbd33a719a
try yomban 2e5b9917609605a6165a0a85661b7e345d517f83d5713e4491897f7e02d446187fe702d4e6c46b41980a64fe59b3510f3c6683f736bb6f
try wombat 066f79b22d01b59e41e4f881a32f1fffc8a415a3d8ce30dd5f84f090e1e2b8ad080e1f09dd52a6b569b55eb74ade41b9842cbcd651919b19372535ce3562baacc
try wonder 40d8a00ed23a0f8b79b8bb0b017703b86c18a58524e0873caad9706f4bd7278c563e8a5477daca8fc9247eddf112ab982634738e90a2944cd932f76b109f
try Wright c6491a572240d55acae0486ad627b7c94521878c6499a74986fb82ccab03d7420787d2214c02d295dccfa0764ce3f382049a4120f70ff45f11f37c95b367t33
try xxxx d597ad764e8238dedb1684527197c5a39743f805f1d2355a89e62eca5725d62cd545dfa573a6b37c71527a63986bce78a056a2a0c6479391fc2349
try yoda 0b48920fa990311b6cc127146a293ea4b5a89aca9574361499b6fc3355625d8d81e220b045f52658d1d6fc7db3c3f04c2b21b2765e0e78d0d4b785542528
try yomama 289bcd6909412ac3179fa1b24e96c3fe88999ea9eb7f0730bd563a30088cd7d442e6b645947112dbc8543a368e5fee735c76d41f0a6185d2a0207897d74837f
try young 8bc62b96f9c8eb345852e0f20a685f0a4e4bdcda764076ad49688812d6f01d154d1ca564150668c813c76a56192220b56e14b591891c5595fcfd5e9cd4d83c
try yvonne d74b91c779166a0706ce2899b2c794c435fd5524c77b53740e2ac1a12974c392d4bb5f682c36973f364aecf61b37f152fde8bd2444910eacc7ae942ebcb0
try zenith 05ce9803e0e32ea2f619f034a93a1ba07869ce7d371eb99idd5d98574e59bb22f447e2d17a0014a74ba66d1d777b24c4fa95878960a341e1f114f5
try zeppelin 5f4a04af6be4a9623c5423ea8ea7464529a3ea0623b6e195306bc97b0e06e3721f1ffac57ab0d4ff6ed33262fc23d4e3771ef8bc7386a0629f36b443e1
try zhongguo 4a9a904d70439d2b032717a237618636e82ebef56a1bd072372cf5771b6a63d7eb5206cd84d6b1d10470b9e066b0756361c4a26aa4f282
try biscotte 016e0c06cbf79500e0d179b2ba78440045488b15af6d8bd68330d54daic51c81b660687ece6a129bc316c4781bc32e45b350b19e957848a5e95f28d69e
try PRIORITY! 484ca8783a185863e98cbee40f2829b299389219a738f082f7839d6d91aa3cde7cdc3f95252b0505a74f023d83cfad5a472158a9dbbf50a02e9b555d53c2dec9
try diction ab1c2b83a2d9b0304541de63f302a9580c0b4252cd70d5a6ce0bd5b8829d2b3bc1f727c70140113071bcc7f1a737a746f0f516618185dd43bbba690b5041
plaintext is diction
search end
(base) javiergarrru@MBP-de-Javier-2 w9 %

```

## Question: How does the brute-force crack work?

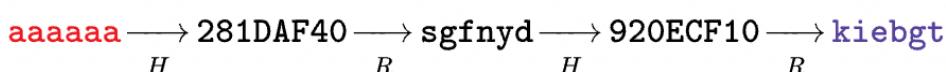
As mentioned before, the program consists of an interative loop that computes the hash of a word in the password.lst at every iteration. The hash is computed using openssl and the loop doesn't stop until the computed hash matches the target one. This brute force attack is successful due to the word diction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



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:

920ECF10 → **kiebgt**  
 $R$

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

aaaaaa → 281DAF40 → sgfnyd → 920ECF10  
 $H$        $R$        $H$

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 problems.

This information has been extracted from: [https://en.wikipedia.org/wiki/Rainbow\\_table](https://en.wikipedia.org/wiki/Rainbow_table).

**Question: What is the SHA-512 preimage of  
 "21fc9c65b173371a479de565a9a529c2aedfea54db72b7b7718e786dbb1265c4517d1  
 c71237f566e98ab961d0b49e5ae4462099735122c61c8f7ab37ac5c389"?**

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

Search in over 26,860,039,705 decrypted MD5 and SHA1 password hashes

Search: 21fc9c65b173371a479de565a9a529c2aedfea54db72b7b7718e786dbb1265c4517d1c71237f566e98ab961d0b49e5ae4462099735122c61c8f7ab37ac5c389 

Algorithm	Hash	Decrypted
sha512	21fc9c65b173371a479de565a9a529c2aedfea54db72b7b7718e786dbb1265c4517d1c71237f566e98ab961d0b49e5ae4462099735122c61c8f7ab37ac5c389 	crypto 

### 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:

```
(base) javiergarru@MacBook-Pro-de-Javier-2 python-md5-collision-master % python3 gen_coll_python.py
Compiling fastcoll
g++ -O3 *.cpp -lboost_filesystem -lboost_program_options -lboost_system -o fastcoll
main.cpp:87:10: fatal error: 'boost/filesystem/operations.hpp' file not found
#include <boost/filesystem/operations.hpp>
^
1 error generated.
make: *** [fastcoll] Error 1
Traceback (most recent call last):
  File "/Users/javiergarru/Desktop/42000/w9/python-md5-collision-master/gen_coll_python.py", line 3, in <module>
    from coll import Collider, md5pad, filter_disallow_binstrings
  File "/Users/javiergarru/Desktop/42000/w9/python-md5-collision-master/coll.py", line 43, in <module>
    raise Exception('could not compile fastcoll')
Exception: could not compile fastcoll
(base) javiergarru@MacBook-Pro-de-Javier-2 python-md5-collision-master %
```

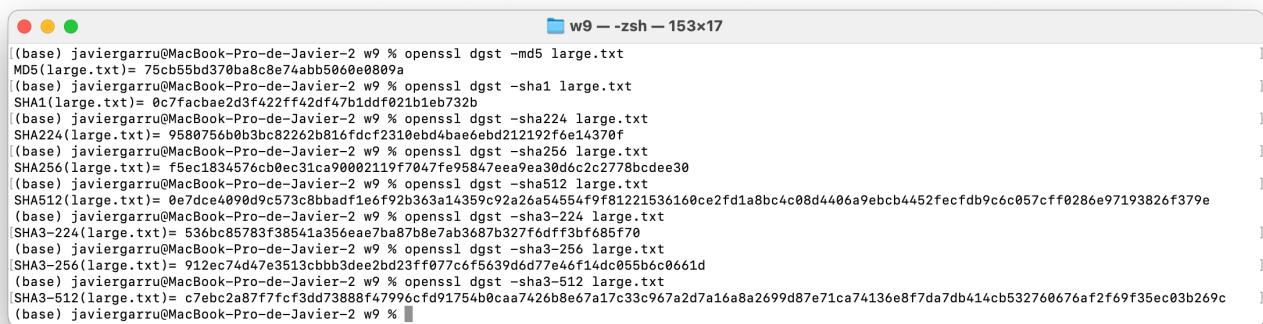
**Question: Suppose an adversary uses brute-force to attack hash functions. For a general hash function with m-bit hash value, how many times on average should an adversary try to find the preimage of a given hash value? How many times on average should an adversary try to find two preimages that generate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



```
w9 -- zsh -- 153x17
(base) javiergarru@MacBook-Pro-de-Javier-2 w9 % openssl dgst -md5 large.txt
MD5(large.txt)= 75cb55bd370ba8c8e74abb50e0e0809a
(base) javiergarru@MacBook-Pro-de-Javier-2 w9 % openssl dgst -sha1 large.txt
SHA1(large.txt)= 0c7facb8e2d3f422ff42df47b1ddf021b1eb732b
(base) javiergarru@MacBook-Pro-de-Javier-2 w9 % openssl dgst -sha224 large.txt
SHA224(large.txt)= 958b0756b0b3bc82262b816fdcf2310ebd4bae6ebd212192f6e14370f
(base) javiergarru@MacBook-Pro-de-Javier-2 w9 % openssl dgst -sha256 large.txt
SHA256(large.txt)= f5ec1834576cb0ec31ca90002119f7047fe95847eea9ea30d6cc22778bcdee30
(base) javiergarru@MacBook-Pro-de-Javier-2 w9 % openssl dgst -sha512 large.txt
SHA512(large.txt)= 0e7dce499d9c573c8bbadff1e6f92b363a14359c92a26a5455af9f81221536160ce2fd1a8bc4c08d4406a9ebcb4452fecfdb9c6c057cff0286e97193826f379e
(base) javiergarru@MacBook-Pro-de-Javier-2 w9 % openssl dgst -sha3-224 large.txt
SHA3-224(large.txt)= 536bc85783f38541a356eae7ba87b8e7ab3687b327fd6ff3bf685f70
(base) javiergarru@MacBook-Pro-de-Javier-2 w9 % openssl dgst -sha3-256 large.txt
SHA3-256(large.txt)= 912ec74d47e3513ccb3dee2bd23ff077c6f5639d6d77e46f14dc055b6c0661d
(base) javiergarru@MacBook-Pro-de-Javier-2 w9 % openssl dgst -sha3-512 large.txt
SHA3-512(large.txt)= c7ebc2a877fcf3dd7388bf47996cf91754b0caa7426b8e67a17c33c967a2d7a16a8a2699d87e71ca74136e8f7da7db414cb532760676af2f69f35ec03b269c
(base) javiergarru@MacBook-Pro-de-Javier-2 w9 %
```

**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
```

```
w9 --zsh-- 153x5
(base) javiergarru@MacBook-Pro-de-Javier-2 w9 % time openssl dgst -md5 superlarge.txt
MD5(superlarge.txt)= 4cabeb2d41b6d86a7e36d76f33d70923
openssl dgst -md5 superlarge.txt 0.96s user 0.11s system 91% cpu 1.169 total
(base) javiergarru@MacBook-Pro-de-Javier-2 w9 %
```

**Time SHA1:** 0.960s

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

```
w9 --zsh-- 153x5
(base) javiergarru@MacBook-Pro-de-Javier-2 w9 % time openssl dgst -sha1 superlarge.txt
SHA1(superlarge.txt)= bd1392c4047de1b612d9c1c5ed36f34cbb0a4dea
openssl dgst -sha1 superlarge.txt 0.76s user 0.11s system 90% cpu 0.960 total
(base) javiergarru@MacBook-Pro-de-Javier-2 w9 %
```

**Time SHA224:** 1.465s

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

```
w9 -- zsh -- 153x5
(base) javiergarru@MacBook-Pro-de-Javier-2 w9 % time openssl dgst -sha224 superlarge.txt
SHA224(superlarge.txt)= a04dc6b827f3154090d3f3e13b82b202bdef60f4ae232ff1ad2202bc
openssl dgst -sha224 superlarge.txt 1.35s user 0.07s system 97% cpu 1.465 total
(base) javiergarru@MacBook-Pro-de-Javier-2 w9 %
```

**Time SHA256:** 1.445s

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

```
w9 -- zsh -- 153x5
(base) javiergarru@MacBook-Pro-de-Javier-2 w9 % time openssl dgst -sha256 superlarge.txt
SHA256(superlarge.txt)= f59a8009ab55b922d1648204df28435adb17f57dc1419a60c98866e2e1bfa30a
openssl dgst -sha256 superlarge.txt 1.37s user 0.06s system 98% cpu 1.445 total
(base) javiergarru@MacBook-Pro-de-Javier-2 w9 %
```

**Time SHA512:** 0.941s

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

```
w9 -- zsh -- 153x5
(base) javiergarru@MacBook-Pro-de-Javier-2 w9 % time openssl dgst -sha512 superlarge.txt
SHA512(superlarge.txt)= 49f4bd03a4b3177c58e70be4e3ff2243b72bb98bd378028f4e52ac76fb2121a452d3a36ff60ac4eefc43e2f9fb9869a68c26ff7d3d5aed9e73fef3ba6cb2e6
openssl dgst -sha512 superlarge.txt 0.82s user 0.08s system 95% cpu 0.941 total
(base) javiergarru@MacBook-Pro-de-Javier-2 w9 %
```

**Time SHA3-224:** 1.227s

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

```
w9 -- zsh -- 153x5
(base) javiergarru@MacBook-Pro-de-Javier-2 w9 % time openssl dgst -sha3-224 superlarge.txt
SHA3-224(superlarge.txt)= 6156ed7440c1f97c40eb9663aded7fc1c42b449807e68887f2a15375
openssl dgst -sha3-224 superlarge.txt 1.16s user 0.07s system 99% cpu 1.227 total
(base) javiergarru@MacBook-Pro-de-Javier-2 w9 %
```

**Time SHA3-256:** 1.295s

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

```
w9 --zsh -- 153x5
(base) javiergarru@MacBook-Pro-de-Javier-2 w9 % time openssl dgst -sha3-256 superlarge.txt
SHA3-256(superlarge.txt)= 603ade6d660d44c315120d065ec6c0aefbc0937b28a63d9b961180b60fcc64a1
openssl dgst -sha3-256 superlarge.txt 1.21s user 0.07s system 98% cpu 1.295 total
(base) javiergarru@MacBook-Pro-de-Javier-2 w9 %
```

**Time SHA3-512:** 2.306s

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

```
w9 --zsh -- 154x5
(base) javiergarru@MacBook-Pro-de-Javier-2 w9 % time openssl dgst -sha3-512 superlarge.txt
SHA3-512(superlarge.txt)= bd213457cef84b97616c00c29b2f91fb892033ca658942ea266abd19ddaddbf673c6218987b3cec1635999c1b95247a186444c61cae9b46522778584c12
openssl dgst -sha3-512 superlarge.txt 2.21s user 0.08s system 99% cpu 2.306 total
(base) javiergarru@MacBook-Pro-de-Javier-2 w9 %
```

**Question: Compare the running time and rank the speed of hash functions, i.e., md5, sha1, SHA-2(sh224, sha256 and sha512) and SHA-3(sh3-224, 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*:

```
(base) javiergarru@MacBook-Pro-de-Javier-2 python-md5-collision-master % openssl speed sha512 sha1 md5 sha256 sha512
Doing md5 for 3s on 16 size blocks: 23208594 md5's in 2.99s
Doing md5 for 3s on 64 size blocks: 12134560 md5's in 2.99s
Doing md5 for 3s on 256 size blocks: 5031074 md5's in 2.99s
Doing md5 for 3s on 1024 size blocks: 1480800 md5's in 2.98s
Doing md5 for 3s on 8192 size blocks: 199842 md5's in 3.00s
Doing md5 for 3s on 16384 size blocks: 100257 md5's in 3.00s
Doing sha1 for 3s on 16 size blocks: 24472007 sha1's in 3.00s
Doing sha1 for 3s on 64 size blocks: 13715607 sha1's in 3.00s
Doing sha1 for 3s on 256 size blocks: 6140476 sha1's in 3.00s
Doing sha1 for 3s on 1024 size blocks: 1899638 sha1's in 2.99s
Doing sha1 for 3s on 8192 size blocks: 254292 sha1's in 2.99s
Doing sha1 for 3s on 16384 size blocks: 127344 sha1's in 2.98s
Doing sha256 for 3s on 16 size blocks: 15709311 sha256's in 2.98s
Doing sha256 for 3s on 64 size blocks: 8235842 sha256's in 2.98s
Doing sha256 for 3s on 256 size blocks: 3453521 sha256's in 2.99s
Doing sha256 for 3s on 1024 size blocks: 1035751 sha256's in 2.98s
Doing sha256 for 3s on 8192 size blocks: 138000 sha256's in 2.98s
Doing sha256 for 3s on 16384 size blocks: 69744 sha256's in 3.00s
Doing sha512 for 3s on 16 size blocks: 13831269 sha512's in 2.99s
Doing sha512 for 3s on 64 size blocks: 13594257 sha512's in 2.98s
Doing sha512 for 3s on 256 size blocks: 4865714 sha512's in 2.99s
Doing sha512 for 3s on 1024 size blocks: 1650670 sha512's in 2.99s
Doing sha512 for 3s on 8192 size blocks: 231378 sha512's in 2.98s
Doing sha512 for 3s on 16384 size blocks: 116693 sha512's in 2.96s
OpenSSL 1.1.1n 15 Mar 2022
built on: Mon Mar 21 08:17:25 2022 UTC
```

```
The 'numbers' are in 1000s of bytes per second processed.
type      16 bytes    64 bytes   256 bytes  1024 bytes   8192 bytes  16384 bytes
md5      124193.15k  259736.40k  430754.16k  508838.66k  545701.89k  547536.90k
sha1     130517.37k  292599.49k  523987.29k  650578.37k  696709.05k  700135.60k
sha256    84345.29k  176877.14k  295686.08k  355909.07k  379361.07k  388895.23k
sha512    74913.48k  291957.20k  416596.25k  565313.07k  636056.57k  645911.52k
(base) javiergarru@MacBook-Pro-de-Javier-2 python-md5-collision-master %
```

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 ##HASH-FUNCTION-NAME :`

**SHA3-224:**

```
[base] javiergarru@MacBook-Pro-de-Javier-2 python-md5-collision-master % openssl speed -evp sha3-224
Doing sha3-224 for 3s on 16 size blocks: 534.8320 sha3-224's in 3.00s
Doing sha3-224 for 3s on 64 size blocks: 5499622 sha3-224's in 3.00s
Doing sha3-224 for 3s on 256 size blocks: 3488354 sha3-224's in 2.99s
Doing sha3-224 for 3s on 1024 size blocks: 1064027 sha3-224's in 2.99s
Doing sha3-224 for 3s on 8192 size blocks: 164642 sha3-224's in 2.97s
Doing sha3-224 for 3s on 16384 size blocks: 82990 sha3-224's in 2.99s
OpenSSL 1.1.1n 15 Mar 2022
built on: Mon Mar 21 08:17:25 2022 UTC
options:bn(64,64) rc4(16x,partial) des(partial) idea(int) blowfish(ptr)
compiler: x86_64-apple-darwin13.4.0-clang -D_FORTIFY_SOURCE=2 -isystem /Users/javiergarru/opt/anaconda3/include -mmacosx-version-min=10.9 -march=core2 -mtune=haswell -msse3 -fthread-vectorize -fPIC -fPIE -fstack-protector-strong -O2 -pipe -isystem /Users/javiergarru/opt/anaconda3/include -fdebug-prefix-map=/opt/concourse/worker/volumes/live/6532e873-130f-47ac-7c5a-68f742465f72/volume/openssl_1447850602712/work=/usr/local/src/conda/openssl-1.1.1n -fdebug-prefix-map=/Users/javiergarru/opt/anaconda3=/usr/local/src/conda-prefix -fPIC -arch x86_64 -march=core2 -mtune=haswell -msse3 -fthread-vectorize -fPIC -fPIE -fstack-protector-strong -O2 -pipe -isystem /Users/javiergarru/opt/anaconda3/include -fdebug-prefix-map=/opt/concourse/worker/volumes/live/6532e873-130f-47ac-7c5a-68f742465f72/volume/openssl_1447850602712/work=/usr/local/src/conda/openssl-1.1.1n -fdebug-prefix-map=/Users/javiergarru/opt/anaconda3=/usr/local/src/conda-prefix -DL_ENDIAN -DOPENSSL_PIC -DOPENSSL_L_CPIUID_0B3 -DOPENSSL_IAS2_SSE2 -DOPENSSL_BN_ASM_MONT -DOPENSSL_BN_ASM_GF2m -DSHA1_ASM -DSHA256_ASM -DSHA512_ASM -DKECCAK1600_ASM -DRC4_ASM -DMD5_ASM -DAESNI_ASM -DPVAES_ASM -DGHASH_ASM -DECOP_NISTZ256_ASM -DX25519_ASM -DPOLY1305_ASM -D_REENTRANT -DNDEBUG -D_FORTIFY_SOURCE=2 -isystem /Users/javiergarru/opt/anaconda3/include -mmacosx-version-min=10.9
The 'numbers' are in 1000s of bytes per second processed.
type          16 bytes   64 bytes   256 bytes  1024 bytes  16384 bytes
sha3-224      28524.37k  117325.27k  298668.44k  364402.56k  454123.66k  454751.89k
(base) javiergarru@MacBook-Pro-de-Javier-2 python-md5-collision-master %
```

## SHA3-256:

## SHA3-512:

```
[base] javiergarru@MacBook-Pro-de-Javier-2 python-md5-collision-master % openssl speed -evp sha3-512
Doing sha3-512 for 3s on 16 size blocks: 5554691 sha3-512's in 2.98s
Doing sha3-512 for 3s on 64 size blocks: 5538567 sha3-512's in 2.99s
Doing sha3-512 for 3s on 256 size blocks: 2001642 sha3-512's in 2.99s
Doing sha3-512 for 3s on 1024 size blocks: 605827 sha3-512's in 2.99s
Doing sha3-512 for 3s on 8192 size blocks: 83471 sha3-512's in 3.00s
Doing sha3-512 for 3s on 16384 size blocks: 41916 sha3-512's in 2.99s
OpenSSL 1.1.1n 15 Mar 2022
built on: Mon Mar 21 08:17:25 2022 UTC
options:bn(64,64) rc4(16x,int) desint() aes(partial) idea(partial) blowfish(ptr)
compiler:x86_64-apple-darwin13.4.0-clang -D_FORTIFY_SOURCE=2 -isystem /Users/javiergarru/opt/anaconda3/include -mmacosx-version-min=10.9 -march=core2 -mtune=haswell -msse3 -fthread-vectorize -fPIC -fPIE -fstack-protector-strong -O2 -pipe -isystem /Users/javiergarru/opt/anaconda3/include -fdebug-prefix-map=/opt/concourse/worker/volumes/live/6532e873-130f-47ac-7c5a-68f741265f72:/user/local/src/conda/openssl-1.1.1n -fdebug-prefix-map=/Users/javiergarru/opt/anaconda3/usr/local/src/conda-prefix -fPIC -arch x86_64 -march=core2 -mtune=haswell -msse3 -fthread-vectorize -fPIC -fPIE -fstack-protector-strong -O2 -pipe -isystem /Users/javiergarru/opt/anaconda3/include -fdebug-prefix-map=/opt/concourse/worker/volumes/live/6532e873-130f-47ac-7c5a-68f741265f72/volume/openssl_1647850602712/work/usr/local/src/conda/openssl-1.1.1n -fdebug-prefix-map=/Users/javiergarru/opt/anaconda3/usr/local/src/conda-prefix -DL_ENDIAN -DOPENSSL_PIC -DOPENSSL_CPUID_OBJ -DOPENSSL_IA32_SSE2 -DOPENSSL_BN_ASM_MONT -DOPENSSL_BN_ASM_GF2m -DSHA1_ASM -DSHA256_ASM -DSHA512_ASM -DKECCAK1600_ASM -DRC4_ASM -DMD5_ASM -DAESNI_ASM -DPAESASM -DGHASH_ASM -DECP_NISTZ256_ASM -DX25519_ASM -DPOLY1305_ASM -D_REENTRANT -DNDEBUG -D_FORTIFY_SOURCE=2 -isystem /Users/javiergarru/opt/anaconda3/include -mmacosx-version-min=10.9
The 'numbers' are in 1000s of bytes per second processed.
type          16 bytes    64 bytes   256 bytes  1024 bytes   8192 bytes  16384 bytes
sha3-512      29823.84k  118561.27k  171378.04k  207480.55k  227931.48k  229682.86k
(base) javiergarru@MacBook-Pro-de-Javier-2 python-md5-collision-master %
```

SHA224:

```
[base] javiergarruo@MacBook-Pro-de-Javier-2 python-md5-collision-master % openssl speed -evp sha224
Doing sha224 for 3s on 16 size blocks: 8826800 sha224's in 2.99s
Doing sha224 for 3s on 64 size blocks: 5978211 sha224's in 2.98s
Doing sha224 for 3s on 256 size blocks: 2975625 sha224's in 2.98s
Doing sha224 for 3s on 1024 size blocks: 979568 sha224's in 2.97s
Doing sha224 for 3s on 8192 size blocks: 136076 sha224's in 2.99s
Doing sha224 for 3s on 16384 size blocks: 68669 sha224's in 2.98s
OpenSSL 1.1.1n 15 Mar 2022
built on: Mon Mar 21 08:17:25 2022 UTC
options:bn(64,64) rc4(16x,int) des(int) aes(partial) idea(int) blowfish(ptr)
compiler: x86_64-apple-darwin13.4.0-clang -D_FORTIFY_SOURCE=2 -isystem /Users/javiergarruo/opt/anaconda3/include -mmacosx-version-min=10.9 -march=core2 -mtune=haswell -msse3 -fthread-vectorize -fPIC -fPIE -fstack-protector-strong -O2 -pipe -isystem /Users/javiergarruo/opt/anaconda3/include -fdebug-prefix-map=/opt/concourse/worker/volumes/live/6532e873-130f-47ac-7c5a-68f741265f72/volume /usr/local/src/conda/include/_openssl-1.1.1n -fdebug-prefix-map=/Users/javiergarruo/opt/anaconda3/usr/local/src/conda-prefix -fPIC -arch x86_64 -march=core2 -mtune=haswell -msse3 -fthread-vectorize -fPIC -fPIE -fstack-protector-strong -O2 -pipe -isystem /Users/javiergarruo/opt/anaconda3/include -fdebug-prefix-map=/opt/concourse/worker/volumes/live/6532e873-130f-47ac-7c5a-68f741265f72/volume/_openssl_1.1.1n -fdebug-prefix-map=/Users/javiergarruo/opt/anaconda3/usr/local/src/conda-prefix -DL_ENDIAN -DOPENSSL_PIC -DOPENSSL_CPUID_OBJ -DOPENSSL_IA32_SSE2 -DOPENSSL_BN_ASM_MONT -DOPENSSL_BN_ASM_MONT5 -DOPENSSL_BN_ASM_GF2m -DSHA1_ASM -DSHA256_ASM -DSHA512_ASM -DKECCAK1600_ASM -DRC4_ASM -DMD5_ASM -DAESNI_ASM -DPVAES_ASM -DHASHASM -DCRYPT_NISTZ256_ASM -DX25519_ASM -DPOLY1305_ASM -D_REENTRANT -DNDEBUG -D_FORTIFY_SOURCE=2 -isystem /Users/javiergarruo/opt/anaconda3/include -mmacosx-version-min=10.9
The 'numbers' are in 1000s of bytes per second processed.
type          16 bytes    64 bytes   256 bytes   1024 bytes   8192 bytes   16384 bytes
sha224        47233.71k  128391.11k  255624.16k  337336.58k  372820.93k  377541.24k
(base) javiergarruo@MacBook-Pro-de-Javier-2 python-md5-collision-master %
```

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 possession 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 be 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 it in such way that its hash matches the hash of a real certificate from a well known 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 conceive.

### 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](mailto: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 is 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.