SIC Lab
Cryptographic Hash Functions (or Digest Functions)

version 1.0

Authors: André Zúquete, Vitor Cunha

Version log:

- 1.0: Initial version

1 Introduction

In this guide, we will interact with different cryptographic hash functions. We will use and develop some tools in order to analyze their statistical characteristics.

2 Cryptographic hash functions

Open a terminal and use the commands `md5sum`, `sha256sum`, `sha384sum` and `sha512sum` to produce and compare the output for some letters, words and sentences. For example:

```
$ echo -n "You should ONLY use this if you're 100% absolutely sure that you know what you're doing" | sha256sum
```

Note: the command `echo` is used here with `-n` as argument. Can you explain why?

Questions:

- Does the number in command's name have any relation to the number of characters that are printed out?
- Given that cryptographic hash functions generate a relatively short output for an input of any size, how can these functions be used to circumvent text size restrictions enforced by services like X (a.k.a. Twitter)?

2.1 Avalanche effect

An essential requirement for cryptographic hash functions is that a small change in a text must produce a completely different hash – avalanche effect. In this exercise we are going to verify this requirement.

1. Choose 1 of the aforementioned commands;
2. Choose a sentence and produce its hash;
3. Change a single bit in the sentence and produce a new hash;
4. Compare the 2 outputs and take conclusions.

Repeat the procedure for different commands and multiple pairs of sentences.

Hint: To change a bit in a sentence you can use the command `man ascii` to identify characters that differ in a single bit, such as `'0'` and `'1'`, and replace one by another.

2.2 Statistical analysis of avalanche effect

Create a program to calculate the statistical distribution of the differences in the hashes (Hamming distance) of a set of messages that differ in one single bit from an original message. You can either use as a starting point the `avalanche-analysis.c` C program or produce your own code.

If you want to use C you need to install some tools to compile the code:

```
$ sudo apt install libssl-dev libssl-doc
```

Then you can execute:

```
$ gcc -O2 -Wall avalanche-analysis.c -o avalanche-analysis -lcrypto
$ # Remember you need to run this every time you change the code
```

If you want to use `Python` you need to install Python3 Cryptography module. The module can be installed using:

```
$ sudo apt install python3-cryptography
```

The program must receive from the user the following input:

1. Number of bytes used as source message;
2. Number of messages (`N`), differing one bit from the original message, to calculate the hash.

For example:

```
$ ./avalanche-analysis 1024 512
$ # OR
$ python3 avalanche-analysis.py 1024 512
```

should produce an initial source message with 1024 bytes and 512 one-bit altered messages. The creation of single bit altered messages must use a random number generator to calculate the position of the bit to alter (make sure you do not use the same one-bit altered message twice!).

After the calculation of the hashes of all the `N` one-bit altered messages, evaluate the difference between each of these hashes and the hash of the original message, in terms of number of bits (Hamming distance, in bits). Comment the distribution of the differences that you obtained.

If you want to have a graphical output of that distribution, install `gnuplot`:

```
$ sudo apt install gnuplot
```

and, assuming that the output of your application is formed by two numbers per row, where the first is the number of bits modified and the second is the number of occurences, then you can use gnuplot as follows:

```
$ ./avalanche-analysis 1024 512 | gnuplot -p -e "plot '-' using 1:2 with boxes"
```

Repeate the process by changing a byte at once. Compare the results.

Hint: You can use the `XOR` operation to change bits (for some bit `b` the result of `b XOR 1` will be different from `b` and the result of `b XOR 0` will be equal to `b`) and also to detect the number of different bits between two hashes (number of bits with the value 1 in the result of `XOR` operation).

2.3 Proof-of-work

Cryptographic hash functions can be used as proof-of-work (or cryptopuzzle), i.e., to prove to others that some task (usually a time consuming one) was done.

What is the best strategy to create a sentence that includes your student number and has a sha256 that starts with 3 hexadecimal `0`'s, for example:

```
$ echo -n "My student number is 1234 and I tend to introduce a bug for every 2469 lines of code I write!" | sha256sum
```

Follow that strategy and find a solution for that cryptopuzzle, counting the number of attempts until finding one. Then, do the same for other student numbers, and compute the average of the number of attempts. Given the birthday paradox, does that average value matches the expectations? Explain why.