Cryptographic Hash Functions: Exploring Collision Resistance and Pre-Image Resistance

Based on the SEED lab “Crypto Lab -- One-Way Hash Functions. Modified for use in COMP 435 at The University of North Carolina at Chapel Hill.

1 Overview

The learning objective of this lab is for students to get familiar with pre-image resistant and collision resistant hash functions. After finishing the lab, in addition to gaining a deeper understanding of the concepts, students should be able to use tools and write programs to generate hash values for a given message.

2 Lab Environment

Installing a hex editor

In this lab, we need to be able to view and modify files of binary format. You can install any hex editor, but here are some suggestions by operating system:

- Mac: `HexFiend - https://ridiculousfish.com/hexfiend/`
- Windows: `HxD - https://mh-nexus.de/en/hxd/`
- Linux: `GHex - https://wiki.gnome.org/Apps/Ghex`

The hex editor allows the user to load data from any file, and view and edit it in either hex or ASCII.

3 Lab Tasks

Task 1: Generating a Message Digest

In this task, we will play with various hash algorithms. To generate the hash values for a message, you will invoke the message digest functions in Python's built in `hashlib` library. Sample code and documentation can be found at https://docs.python.org/3/library/hashlib.html. You can use the following constructor to generate a specific hash algorithm.

```
m = hashlib.<dgsttype>()
```

Replace `<dgsttype>` with a specific hash algorithm, such as `sha1`, `sha256`, or `md5`. You should then update the algorithm with a message of your choice, and finally, digest the message to see its hash value (note: you should use the `hexdigest` method when trying to view hash values).

In this task, you are encouraged to try at least 3 different algorithms.

Task 2: The Randomness of a Hash

A required property of a cryptographic hash function is that it is difficult to predict the original input given only the resulting hash digest. This means that two input messages that differ only by a single bit should produce hash digests that differ considerably. To see how this works, do the following exercise for both the MD5 and SHA256 hash functions:

1. Create a text file of any length.
2. Generate the hash value H1 for this file using a hash function.
3. Flip one bit of the input file. You can achieve this modification using your hex editor.
4. Generate the hash value H2 for the modified file using the same hash function you used in step 2.
5. Observe whether H1 and H2 are similar or not.

Hamming Distance is a measure of difference between two strings. It is defined for two strings of equal length as the number of positions at which the two strings differ. For the first program you will submit on Gradescope, you will write a function that calculates the Hamming distance between two different binary strings, given their hexadecimal values. For example, take the hash values `ab123` and `ab122`. These two hexadecimal values are represented in binary by:

```
10101011000100100011

10101011000100100010
```

The Hamming Distance between the two binary strings is 1, because the strings differ in only one position (the least significant bit). The input for this function will be two hexadecimal strings. Your function signature should match the following:

```
def hammingdistance(hex1,hex2):
```

The output of your function should be a single integer representing the hamming distance. **Note that you only need to write the function.** The grader will import and test your function automatically. **Name the python file with your function a2_hamming.py**.

6. Use your Hamming Distance function to count the number of bits that differ in your generated H1 and H2.

Task 3: Exploring Pre-image Resistance

In this task, we will investigate the difference between two properties of a cryptographic hash function: pre-image resistance and collision resistance. We will use a brute-force method to see how long it takes to break each of these properties.

Since cryptographic hash functions are quite strong against a brute-force attack on those two properties, it would us years to break them using a brute-force method. To make the task feasible, we will reduce the length of the hash value.
Now, let us say that we have the SHA-256 sum:

```
d7a8fbb307d7809469ca9abcb0082e4f8d5651e46d3cdb762d02d0bf37c9e592
```

(This SHA-256 sum happens to be of the string ”The quick brown fox jumps over the lazy dog.”)

To test the pre-image resistance property, your goal is to find a string which collides with the first n bits of a given SHA-256 sum. For example, a 24 bit collision for the above hash would be `0xd7a8fb`. Write a Python function which will take as input a SHA-256 digest and a hash length parameter n (**hash length is in bytes**), and produce as output a string whose hash digest collides (for the first n bytes) with the given hash digest. Use the hashlib library to generate the hashes. As soon as a colliding pre-image (the string) is found, your program should return that string. Name your file **a2_preimage.py**. Your function signature should be the following:

```
def find_preimage(target, n):
```

Important Note: Input will be provided as a python bytes object. This is the same type of object that sha256.digest() outputs.

For example our input may look like:

```
d7a8fbb307d7809469ca9abcb0082e4f8d5651e46d3cdb762d02d0bf37c9e592 3
```

For the above input, output is the following. Note that the SHA-256 sum of the below string collides in the first 24 bits (3 bytes).

```
19503334fhjasbfjhebw
```

For fun: When you have completed the assignment, try to find a collision up to 28 bits or 32 bits. How much longer did it take?

Task 4: Exploring Collision Resistance

Next you will be testing collision resistance. Now, write a python script which takes a match length parameter n as before, except this time you choose both m1 and m2. You will search for a pair of messages whose hashes collide in the first n bytes. Output a tuple containing the two strings whose hashes collide in the first n bytes. See below for a sample output; n is 3 bytes in this case.

```
(12856fhjasbfjhebw 739fhjasbfjhebw)
```

Your program must not be deterministic and should take less than 10 seconds to run when n is equal to 3 bytes. If your program takes much longer, consider using a certain data structure to optimize.

Answer the following questions:

1. What is the average number of trials it took you to break pre-image resistance based on 15 experiments where n=2?

2. What is the average number of trials it took you to break collision-resistance based on 15 experiments where n=2?

3. Is it easier to find a collision or find a pre-image using brute force?

4 Submission

Submit your code on Gradescope. You will submit the following files:

1. a2_hamming.py
2. a2_preimage.py
3. a2_collision.py
4. YourOnyen_report.txt (The answers to the above questions)