University of Chicago
CMSC 28400: Introduction to Cryptography, Autumn 2021
Course Staff: Akshima, David Cash (Intructor), Andrew Chu, Alex Hoover

Project 3: Hash Function Attacks

Due Friday December 3 at 11:59pm Chicago Time

Introduction

This short project will walk you through some basic hash function attacks. We'll start with the basic birthday attack and Floyd's algorithm, and then look at how to extend these attacks to produce "useful" collisions. The project asks you to implement and empirically measure the performance of these algorithms.

Rules

Collaboration policy. Please respect the following collaboration policy: You may discuss problems with up to 3 other students in the class, *but you must write up your own responses and your own code.* At the beginning of your submission write-up, you must indicate the names of your (1, 2, or 3) collaborators, if any.

Sources. Cite any sources you use. You may Google liberally to learn basic Python, and you should use libraries for non-crypto steps like Base64 encoding if needed, but you should not Google for anything related to the cryptographic problem solving. Using Google, or searching for posted solutions from other universities, is not allowed.

Campuswire. We encourage you to post questions on Campuswire, but do not include any significant code in public Campuswire posts. If you have a question that you believe will reveal secrets you have discovered while working on the assignment, post privately to just the instructors. If you have a question that you believe will be of general interest or clarifies the assignment, please post publicly. If you are uncertain, post privately; we will make public posts that we believe are of general interest (with your permission).

Grading. Solutions to problems consist of two parts: Your code and a brief written response to questions. Responses will be graded for correctness and clarity.

Assignment Tech Set-Up and Overview

What and How to Submit

You should submit two files:

1. A file `<YOUR CNETID>-project3.pdf` or `<YOUR CNETID>-project3.txt` that contains responses in English to the questions in the problems (replace `<YOUR CNETID>` with your actual cnetid).

2. A file `<YOUR CNETID>-project3.py` containing all of your code. You should implement functions with names `problem1`, `problem2`, etc here. These functions are stubbed out already in the provided python file, which can serve an initial template. It's fine, and encouraged, to write helper functions and reuse them across problems. You may also copy code from your Projects 1 and 2 if you want, but be sure to cite yourself if and when you do this.

Please upload these two files to Canvas.

Hash Function for this Project

The hash function SHA256[1], takes a bit-string of any length and produces a 32-byte output. It is designed with several security properties in mind, including "looking random". Actually finding collisions in SHA256 is intractable, so the problems in this project will attack a weakened version of SHA256 that only outputs the *first five bytes*.

More precisely, the hash to attack is implemented in the python file with this project as a function called `proj3hash`.

Problem 1: Birthday Attack Warm-up (5 points)

In this problem we will look for collisions in `proj3hash` and observe if they occur with the frequency we'd expect from a random function.

For this problem, run the following experiment 10 times:

1. Create a string `pre` by prepending an ASCII digit between `'0'` and `'9'` to your `cnet_id`. (Use a different digit each time.)

2. Evaluate `proj3hash` on strings `s` that start with `pre` until you find two that collide under `proj3hash`. You can create the strings by appending whatever you like.

3. Count the number of `proj3hash` evaluations in your trial.

Finally, compute the average number of `proj3hash` evaluations of your 10 trials.

What to submit. In your code file, implement a function `problem1()` that will run all ten trials and return the average number of evaluations.

In your write-up file, explain how you generated your inputs. In lecture and the probability notes, we learned that it should take about $2^{n/2}$ evaluations to find a collision in a hash function with n-bit output. How does this relate to your problem, where you are looking for collisions? Are your findings consistent with that claimed behavior?

Problem 2: Finding Useful Collisions (5 points)

Hash function collisions are usually only useful for attackers when they involve meaningful messages. For instance, suppose messages `M1 = "David owes cnet_id 100 dollars by tomorrow.` and `M2 = "David owes cnet_id a million dollar by tomorrow"` hash to the same value `y`. In this problem you will find such a useful collision in `proj3hash`. More precisely, your goal is to find two

[1] SHA256 is a member of the SHA2 family: `https://en.wikipedia.org/wiki/SHA-2`.

distinct messages `M1`, `M2` such that `proj3hash(M1)` = `proj3hash(M2)`, *and* `M1` can be interpreted as meaning that David owes you $100, while `M2` means he owes you a million bucks.

Suggested approach. To solve this problem you need to generate a large number of strings that have the first meaning ("David owes me 100 dollars") and a large number of strings that have the second meaning ("David owes me 1,000,000 dollars") To do this, write two routines that will produce a lot of such strings. The easiest approach is to come up with several ways to tweak the strings (i.e. adding adding some space or tab characters, using a synonym, formatting the numbers differently, etc) without changing their meaning. Then combine those tweaks in an exponential number of combinations. You can have a little fun with the tweaks.

What to submit. In your code file, implement a function `problem2()` that will return your colliding messages. Don't hard-code the messages; your code should actually find them when run.

In your write-up file, explain how you generated your inputs to the hash, and report how many inputs you had to generate before finding a collision.

Problem 3: Finding Collisions in Small Space (5 points)

Recall that we learned how to find collisions via Floyd's algorithm. This problem is simple: Repeat the first problem, but with Floyd's algorithm. Implementing Floyd's is very easy; Please take a second to do it yourself instead of Googling for it. Seeing that it *actually* works is fun (in my opinion, anyway).

What to submit. In your code file, implement a function `problem3()` that will run all 10 experiments and return the average number of hash evaluations needed by your algorithm. Note that hash evaluations is different that loop iterations, since one loop iteration results in multiple hash evaluations. (And there are two loops.)

In your write-up file, explain how you generated your inputs. In lecture we argued that the "random walk" would collide with itself in about $2^{n/2}$ iterations. As with the first problem, comment (briefly; one sentence is fine) on the theory relates to the number of hashes your algorithm needed.

Problem 4: Finding Useful Collisions in Small Space (10 points)

Now let's use Floyd's algorithm to find *useful* collisions. At first glance, it might not be obvious how this should work. Floyd's algorithm depends on feeding the hash back into itself, and produces random-looking strings that collide.

There is however a trick to adapt Floyd's algorithm to produce useful collisions. To start, we'll need to slightly extend your method in Problem 2 for generating a large number of messages with the same semantic meaning.

The first step of this extension is to build a function f that maps *hash outputs* to *useful strings*, such that f has the following properties:

1. If the first bit of the input x is zero, then $f(x)$ is a message that means "David owes me 100 dollars."

2. If the first bit of x is one, then $f(x)$ is a message that means "David owes me million dollars."

You can design your f to use the bits of x to determine how to tweak the output string without changing its meaning. So, for example, the second bit of x might determine if you put in "David" or "Lil Davey", the third bit might determine if there's an extra space, etc (it's also possible to use multiple bits at once to select from a list, which could reduce the amount of code you have to write). Your code from Problem 2 might already be in the right form to work this way.

The second step is a simple change. Before, we were finding collisions in a function H (called `proj3hash` above). Now, instead define the function $H'(x) = H(f(x))$, and find a collision in H'. You should work out for yourself why a collision in H' is (potentially) useful if your goal is to attack H. The attack might actually get unlucky and fail to find something useful, but after a few tries (from different starting points in Floyd's algorithm) it will tend to succeed.

The idea being exploited here has many applications in cryptographic attacks. In fact, this idea can be extended to give a small-space version of the meet-in-the-middle attack against double encryption, and some public-key encryption algorithms as well!

What to submit. In your code file, implement a function `problem4()` that will find your useful collision against `proj3hash` via the method described above.

In your write-up file, explain how you generated your inputs. Also explain when a collision might not be useful, and roughly estimate how many times the algorithm will have to restart (the answer for the latter part should be very simple).