Review Assignment Due Date: 14th of February

Cooking with Hash: Slicing, Dicing, and Mixing Data

Hash functions are everywhere in computer science and cryptography. We use them to create fingerprints of data for quick comparison, to check file integrity, to organize data in hash tables, and, crucially in cryptography, to commit to a secret (to prove you know something without revealing it).

For many of these applications, especially in cryptography, it's not enough for a hash function to just scramble data. A good hash function must resist several types of attacks:

- Collision resistance: It should be practically impossible to find any two different inputs that produce the same hash output.
- Irreversibility (First pre-image resistance): Given a hash output, it should be infeasible to figure out what input produced it.

If a hash function is vulnerable to these attacks, it can't be safely used for cryptographic purposes, as attackers could break commitments, forge digital fingerprints, or violate the security of protocols that rely on hashes.

In this assignment, you'll experiment with hash functions to discover how and why these properties matter and what can go wrong if they aren't satisfied.

Expected submissions

Your solutions should be in the form of bash scripts in the "**solutions**" folder. They should provide output as specified in the following specifications.

You are supposed to create some brute forcing programs.

Instead, submit text files with your solutions as specified below.

You can use any programming language you see fit and I want to check your source code.

You will probably need to reimplement the given hash functions if you use another language to avoid Python, but they are quite simple to follow. Please submit your source code to the "**implementation**" folder so that an I can provide feedback.

Use blackboard to ask for questions and ask for feedback.

When two bytes collide: cryptanalysis of a weak hash function

Consider the hash function “**xor32_hash**” given in “**functions/hash0.py**” that produces 32-bit output. You are supposed to analyze this hash function and produce the following attacks.

It is simple enough for you do it by hand, you don't need to brute force anything (we are going to do that later on).

Exercise 1

Calculate two different strings that will hash to the same value. Both strings should be composed of ascii characters and be 8 characters long.

Expected output: a text file “**submissions/exercise01.txt**” with both strings separated by a comma (see provided example).

We call this a collision attack. If it's easy to find collisions for a hash function, it can't be trusted as a unique fingerprint of data, i.e., we can't use digests to identify or commit to data.

Exercise 2

Calculate an input string that will have exactly the hash ‘1b575451’.

Expected output: a text file “**submissions/exercise02.txt**” with the required string.

We call this a first pre-image attack, i.e., we can construct the pre-image for a given hash. If a hash function is not pre-image resistant, an attacker could “reverse” a hash value to recover or forge a message. This would break the use of hashes for commitments (proving you knew a value without revealing it) or for securely storing passwords (as a hash could be reversed to recover the password).

Brute forcing can be an option

For the next exercises, consider the function given in “**functions/hash1.py**” that also produces 32-bits digests. This is still an insecure function, but it's algebraic structure is a bit more complicated to render cryptanalysis infeasible for the scope of this class. Yet, we can always brute force it, i.e., try many combinations of inputs and search for a convenient output.

The search space is quite small since 32-bits will render around 4 billion possible outputs, something that any modern computer can generate in a few minutes (if not seconds if properly implemented).

For the next exercises, you are expected to brute force the solutions. Expect to let your program run for a few minutes if you are using a multicore solution or a few hours for single core.

Exercise 3

Calculate two different strings that will hash to the same value (i.e., perform a collision attack). Both strings should be composed of ascii characters and be 8 characters long (32 bits).

Expected output: a text file “**submissions/exercise03.txt**” with both strings separated by a comma (see provided example).

This should be fairly easy (a few seconds to minutes) if you understand how to take advantage of the birthday paradox.

Exercise 4

For this exercise, consider the function given in “**functions/sha256.py**” Compute three partial collision using SHA256 (proof-of-work style). Find three ASCII strings starting with `bitcoin` (e.g. `bitcoin0`, `bitcoinA0b1`, etc) that, when hashed with SHA256, produces a digest starting with the bytes `0xcafe`, `0xfaded`, and `0xdecade`, in this order.

Expected output: a text file `solutions/exercise06.txt` the three strings separated by commas (e.g `bitcoin0,bitcoin1,bitcoin2`).

This exercise mimics Bitcoin's proof-of-work system, where miners search for inputs that produce hashes with specific patterns. Here, you're looking for specific leading bytes with increasing difficulty. This should take seconds on modern hardware, it is intended to give you a feel of how to build an adjustable proof of work system.