Hashing

Introduction

Hashing has many important uses in cryptography. Strictly speaking, a *hash function* is a function that takes input data and produces an fixed length output called a *hash*. That hash is always the same length, regardless of what the input is. Hash functions are used elsewhere in computer science, but in cryptography, they should satisfy a few properties to be considered secure:

- The output hash should appear random.
- Any small change in the input should drastically change the output hash.
- It should be very difficult to reverse, i.e., to figure out what an input is based on the output hash.
- *Collisions* should be very rare in practice. Collisions are where different inputs have the same output hash.

A common and pretty secure hash is SHA-256. Here is how to use it in Python:

```python
from hashlib import sha256
print(sha256(b'hello').hexdigest())
```

The output is `2cf24dba5fb0a30e26e83b2ac5b9e29e1b161e5c1fa7425e73043362938b9824`. It is 256 bits long, which is 64 hex digits. Below are some more examples:

hello	`2cf24dba5fb0a30e26e83b2ac5b9e29e1b161e5c1fa7425e73043362938b9824`
jello	`187c9bceeb919e1b3e6d20fa50ecabf7d9d50b5343e8f9a3d912abb13929102e`
Hello, this is a test.	`8e128a3aba120f69ee5524ca812987d003ed0c3e31f5a27b0b3010f949b29812`
h	`aaa9402664f1a41f40ebbc52c9993eb66aeb366602958fdfaa283b71e64db123`

Notice that a small change from `hello` to `jello` totally changes the hash. Notice also that the hash length is always the same. That comes from the definition of hashes, that they always produce the same length output. For instance, I have a wordlist file on my computer that is 820,350 bytes in size. I just ran it through SHA-256 and got the hash `5cc92bd6cbf684fe1340b1653f9139fe7e721c95b6aa396b80b06942234dc2d8`, which is the same length as all the hashes above (256 bits, or 32 bytes).

That file was hashed from 820,350 bytes into 32 bytes, so it is mathematically impossible to reverse the hash to recover the contents of the file. Too much data was destroyed in the process. The only thing you could do to “recover” the contents would be to try hashing some files of your own. If you ever do get a file that gives you the same hash as above, then it is almost certainly the same file as I hashed.

Because of the four properties of cryptographic hash functions, hashes can be used as fingerprints of data. When you hash some data, it produces a long, random-looking string that is almost certainly unique to that data. As long as the hash function is secure, the odds of any other piece of data having that same hash (fingerprint) are astronomically small.

This is a little like real-life fingerprints. Fingerprints are more or less unique to a person in that no two people are supposed to have the same fingerprints. You can’t reconstruct the person from the fingerprints, but if you find the fingerprints somewhere, you can be pretty sure who made them.

Uses of hash functions

File integrity — It’s common in security and system administration to want to know if a file’s contents have been changed by someone. Hashing can help with this. You periodically take the hash of the file and store it. If you want to determine if the file has changed, redo the hash and compare it with the stored value. Even a change of one character in the file will totally change the hash. This is often done with important system files to detect if intruders or malware have changed things. You could also detect changes in files by storing copies of the files and comparing things to the copies, but that takes a lot more space than a hash, which is only maybe 32 bytes, and it is also considerably slower.

Websites offering files for downloads also use hashes. They will post the hash of the file on the site. When you download the file, you can compute the hash of the file and compare it to the value posted on the website. That way, you can know if the file was corrupted or possibly tampered with during the download process.

You can compute file hashes at the command line in most operating systems. In Windows PowerShell, `Get-FileHash` is the command. It does SHA-256 by default. On Mac, `shasum -a 256` usually works, and `sha256sum` works on Linux. There are commands for other hash functions as well.

Password storage — If you're in charge of a website that people log onto, it's a really bad idea to store passwords in plain text. If anyone breaks into the server, they will have immediate access to the passwords. Also, any untrustworthy administrators will also have access to them. You could encrypt the passwords, but that's still problematic. The encryption key will need to be stored somewhere, and if an attacker or untrustworthy administrator has access to it, then they have immediate access to the passwords.

Instead, the standard is to hash the passwords and store the hashes. Whenever someone logs into the system, their password is hashed and compared to the stored hash. If they match, then we assume the password is correct since the hash acts like a unique fingerprint of the password. With hashing, attackers and untrustworthy administrators have only the hashes, not the original passwords. They can still figure out some of the passwords using brute-force techniques where they hash common passwords and see if they match any of the hashes stored on the server. But this is considerably more work than if the passwords are stored in plaintext or encrypted.

Blockchain — We will cover this in more detail later, but Bitcoin and blockchain use hashing in multiple places. Bitcoin mining involves trying to brute-force reverse a hash. The chaining in blockchain is accomplished by hashing together blocks of transactions.

Message integrity — When you encrypt something, an eavesdropper can't read the message, but they can still modify the encrypted bits, which might have devastating effects on the resulting plaintext. For instance, the binary encoding of the letters B and C in ASCII are 01000010 and 01000011. Below on the left is a simple stream cipher encryption of the letter B. The eavesdropper would only be able to see the ciphertext line. Suppose she flips the last bit of the ciphertext to a 1 and sends the ciphertext along. When it's decrypted, shown on the right, the resulting plaintext now corresponds to the letter C.

plaintext (letter B)	0 1 0 0 0 0 1 0
keystream	0 1 0 1 0 1 0 0
ciphertext	0 0 0 1 0 1 1 0

modified ciphertext	0 0 0 1 0 1 1 **1**
keystream	0 1 0 1 0 1 0 0
new plaintext (letter C)	0 1 0 0 0 0 1 **1**

In short, an eavesdropper who knows what they are doing can flip bits of the ciphertext in ways that change the meaning of the plaintext, even if she can't read the plaintext. To defend against this, a *message authentication code* (MAC) is used. If Alice and Bob are sending encrypted messages to each other, a MAC is sent with each message. That MAC is computed by Alice and sent to Bob, who uses it to tell if the message has been tampered with.

Here is one way of doing this that uses hashing (often called an HMAC). Alice and Bob first have to establish a shared secret key K, perhaps by Diffie-Hellman. If Alice is sending message M, she combines M and K and then hashes the result. This is the MAC that she sends to Bob. When Bob gets everything, he decrypts the message, recomputes the MAC, and makes sure it agrees with what Alice sent. If it does, then he can be very sure the message is the same as what Alice sent, and otherwise he knows tampering has occurred.

If the message had been tampered with, then when he combines the message and K, the result will be different from what Alice did, and therefore the hashes won't match. The reason hashing comes into things is we need to disguise the combination of M and K before sending in such a way that the eavesdropper can't recover M from it. The reason for the shared key K is because it adds a level of authentication. Not only does Bob know the message was not tampered with, but he also knows it must have come from Alice because she's the only other one with a copy of K. No one else would be able to produce the same MAC without having a copy of K.

Collisions

Recall that collisions are when two inputs have the same output hash. Collisions are bad for hash functions. For instance, if we use hashes to store passwords, a collision would mean someone might be able to log on with a different password if it happened to have the same hash as the real one. A more common problem is when hashes are used for file integrity. There could be two different versions of a file that both have the same hash. In fact, hash collisions are most commonly used to insert malware into other files or create fake certificates.

The former standard hash function MD5 was broken in the early 2000s when people discovered how to create hash collisions with only a few seconds of computing time. This is used to mess with the file integrity application of hashes. For instance, suppose a document says "Alice agrees to pay $500 to Bob." If Alice wants to defraud Bob, she can produce a ton of small variations on the original document that differ in small details like punctuation, spacing, or whatever. For each of those variations, she also replaces $500 with $5. She then computes the hash of all those variations until one of them has a hash matching the original $500 version. For a broken hash function, like MD5, it's possible to do this quickly. She can then pass the $5 document off as the real one because it has the same hash.

The birthday problem makes another appearance here. Remember that the birthday problem says that if you generate random numbers in the range from 1 to n, after about $\sqrt{n}$ numbers are generated, repeats are likely. For a hash function, repeats are collisions. If a hash function has an output of 64 bits, then there are 2^{64} potential hashes, and repeats are likely after $\sqrt{2^{64}} = 2^{32}$ hashes, which is only a few billion. So that's part of the reason why hashes are typically at least 256 bits in length.

Collisions are inevitable for hash functions. There are infinitely many possible data inputs, and because the output is a fixed size, there are only finitely many outputs. For a good cryptographic hash function, however, collisions between any two real-world inputs should be vanishingly small. For instance, SHA-256 produces 256-bit hashes. There are $2^{256} \approx 10^{77}$ possible output hashes. On the other hand, if all 7 billion people on earth had, say, 10,000 devices that produced 1,000,000 strings a second every second for the next 1000 years, the number of strings produced would still only be around 10^{30}, which is far less than 10^{77}. Even with the birthday problem reducing things to $10^{38.5}$, we're still pretty far off. So the odds that any two strings ever produced in the course of human history having the same SHA-256 hash is extremely small. This, of course, assumes that SHA-256 does a good job of randomizing things. It seems to, but it's not been proven.

Common hash functions

- MD5 — This was a standard for a long time and is well known. For that reason, it's still in wide use, but it should not be. It's been badly broken since the early 2000s, with easily-available software able to produce collisions in seconds. MD5 should never be used in security applications, though it's fine for other things.
- SHA-1 — This was another standard that has been deprecated. It's not as badly broken as MD5, however. As of 2020, it's estimated to cost about $45,000 in computing power to create a collision.
- SHA-2 — This is a family of hash functions, including SHA-256, SHA-384, and SHA-512, which produce 256-bit, 384-bit, and 512-bit hashes. As of 2020, SHA-2 is still considered secure, but because of similarities with SHA-1, people think it might eventually be broken.
- SHA-3 — When people realized SHA-2 could eventually be broken, a contest, similar to the one for AES, was held to create a replacement. The winner, called Keccak, was renamed SHA-3. It's a new standard, and it isn't yet widely used.
- Blake2 — This is another good hash function.
- Password hashing functions — All of the above hash functions should *never* be used for hashing passwords. The reason is that they are too fast. We'll look at password cracking a little later, but the basic idea is given a password hash to try to guess what the password is, hash that guess, and compare with the given hash. The faster the hash function, the more guesses can be done in a given amount of time. So there are hash functions deliberately designed to be slow. Some good ones are bcrypt, scrypt, and argon2.