Modern Cryptography, Part V

Hashing

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:

```
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 also that the hash length is always the same. Notice that a small change from `hello` to `jello` totally changes the hash. 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 command 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 replaced 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.

Digital signatures

We often physically sign documents. Digital signatures are the electronic analog of that. They are a way for someone to verify a document. They are done using public key cryptography.

Let's say Alice wants to digitally sign a document so that everyone knows that it was she that signed it and not someone else. She creates a public/private key pair. She signs the document by encrypting a hash of the document with her private key. People can then verify the signature by decrypting it with the public key and comparing it with the hash of the document. Because the public and private keys are tied together, only Alice's private key could have produced something that would decrypt back into the actual hash. So that's how we know the document was signed by Alice.

For instance, RSA is often used for this. Suppose Alice uses the public key (n, e) and private key d. She hashes the document to create a hash H. She then does $H^d \bmod n$, which is the digital signature D. People then verify this by doing $D^e \bmod n$ and making sure they get H.

In theory, we could skip the hash and just encrypt the document itself, but public-key cryptographic methods are slow and are best used with small amounts of info. RSA was for years the main digital signature algorithm, and its still widely used for certificates, but another algorithm, ECDSA (elliptic curve digital signature algorithm) is taking its place in newer applications.

Certificates

The last piece of modern cryptography concerns authentication. When we see a public key belonging to Alice, how do we know it belongs to Alice and not to someone impersonating her? This is the weak point of modern cryptography. Everything up to this point involves some pretty solid mathematics, but this part relies on humans, who are sometimes weak and lazy.

When Alice first generates a public/private key pair, she goes to an entity called a *certificate authority* (CA). The CA is charged with verifying that Alice is who she claims to be, possibly by checking her documentation. They then certify that Alice's public key is valid by digitally signing it with their own private key. This is called a *certificate*. From there, we would then have to wonder if that CA really is who they say they are. So that CA itself has to be verified, possibly by another higher-level CA. This process could go on and on, but it stops eventually at root certificates. These are built into browsers and operating systems.

The weak point of the system is the document verification process. Not all CAs are careful about this, and even if they are, a smart attacker could trick them. In fact, certificates are the main weak point of modern cryptography. Everything else we have looked at is build on some pretty solid mathematics, but certificates rely on humans to do the verification.

Further, the certificate system ends at the root certificates built into browsers and operating systems, so you also have to trust your browser and OS, but you also have to trust them on a lot of other things anyway. An interesting instance of this was the Lenovo Superfish scandal. Lenovo wanted a way to inject adds into HTTPs traffic. Because of certificates, they couldn't do that, so they had a special root certificate installed on some of the computers they made in 2014-5 which allowed them to man-in-the-middle HTTPs traffic to inject adds. However, the private-key, which was the same on all the computers, was protected by a weak password. Once attackers had that password, they could man-in-the-middle connections as well.

Most browsers allow you to see a site's certificate. Often this works by clicking the icon next to the URL bar. Your browser does the job of verifying the certificates for the sites you visit. Occasionally, you'll get a certificate warning if something is off. A lot of times, it just comes from a site that misconfigured something with their certificate, but it could also come from someone trying to hijack a connection. Most users, unfortunately, will blindly click through the warning message. You may occasionally see self-signed certificates where the site's private key was not signed by a CA. Often this is okay, but it's something to be aware of.

TLS

The protocol for downloading web pages, HTTP, is unencrypted. User names, passwords, credit numbers, and whatever else are all visible to any network traffic. In the mid 1990s, a technology called SSL (Secure Sockets Layer) was developed to add security to HTTP. SSL went through a few versions and was eventually renamed TLS (Transport Layer Security). People still use terms SSL and TLS interchangeably. The current version of TLS is 1.3, though both versions 1.2 and 1.3 are in common use.

TLS uses much of the cryptography we've covered. It starts with a handshake. The client sends a Hello message that contains, among other things, the *ciphersuites* it supports. This includes what key exchange, encryption, and message authentication types it can handle. The server will look at that list and choose the strongest one that it also supports. Early on in the handshake, the server also presents its certificate, which the client verifies. Then a key exchange happens, usually using Diffie-Hellman or the elliptic curve variation on it. After that, data transfer can begin. The command-line utility `curl` can be used to show the handshake when used with the `-v` option. Here is an example:

```
* TLSv1.2 (OUT), TLS handshake, Client hello (1):
* TLSv1.2 (IN), TLS handshake, Server hello (2):
* TLSv1.2 (IN), TLS handshake, Certificate (11):
* TLSv1.2 (IN), TLS handshake, Server key exchange (12):
* TLSv1.2 (IN), TLS handshake, Server finished (14):
* TLSv1.2 (OUT), TLS handshake, Client key exchange (16):
* TLSv1.2 (OUT), TLS change cipher, Client hello (1):
* TLSv1.2 (OUT), TLS handshake, Finished (20):
* TLSv1.2 (IN), TLS handshake, Finished (20):
* SSL connection using TLSv1.2 / ECDHE-RSA-CHACHA20-POLY1305
```

Below are a few of the couple dozen ciphersuites that my web browser supports. The first two are the strongest ones, and the last one is the weakest one.

```
TLS_ECDHE_RSA_WITH_CHACHA20_POLY1305_SHA256
TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384
TLS_RSA_WITH_3DES_EDE_CBC_SHA
```

One type of attack on TLS is called a *downgrade attack*. This is where an attacker man-in-the-middles a TLS handshake and convinces both sides to use the weakest ciphersuite they both support. The attacker then uses some known attacks to break the encryption. For instance, 3DES in CBC mode is subject to something called a padding oracle attack.

There have been several other attacks on SSL/TLS over the years. The worst was called Heartbleed. It involved a buffer overflow exploit in code from the openSSL library that allowed attackers to read a server's memory to extract key information. It's estimated that around 17% of the HTTPs servers were vulnerable to the attack by the time it was found in 2014.

TLS is used to secure HTTP traffic. When HTTP is run over TLS, it is known as HTTPs (the "s" is for "secure"). TLS is also used to secure email protocols like SMTP, and it's beginning to be used to secure DNS.