7. Cryptographic Hashes

7. Cryptographic Hashes

7.1. Overview

A cryptographic hash function is a function, H, that when applied on a message, M, can be used to generate a fixed-length "fingerprint" of the message. As such, any change to the message, no matter how small, will change many of the bits of the hash value with there being no detectable patterns as to how the output changes based on specific input changes. In other words, any changes to the message, M, will change the resulting hash-value in some seemingly random way.

The hash function, H, is deterministic, meaning if you compute $H(M)$ twice with the same input M, you will always get the same output twice. The hash function is unkeyed, as it only takes in a message M and no secret key. This means anybody can compute hashes on any message.

Typically, the output of a hash function is a fixed size: for instance, the SHA256 hash algorithm can be used to hash a message of any size, but always produces a 256-bit hash value.

In a secure hash function, the output of the hash function looks like a random string, chosen differently and independently for each message—except that, of course, a hash function is a deterministic procedure.

To understand the intuition behind hash functions, let's take a look at one of its many uses: document comparisons. Suppose Alice and Bob both have a large, 1 GB document and wanted to know whether the files were the same. While they could go over each word in the document and manually compare the two, hashes provide a quick and easy alternative. Alice and Bob could each compute a hash over the document and securely communicate the hash values to one another. Then, since hash functions are deterministic, if the hashes are the same, then the files must be the same since they have the same "fingerprint". On the other hand, if the hashes are different, it must be the case that the files are different. Determinism in hash functions ensures that providing the same input twice (i.e. providing the same document) will result in the same hash value; however, providing different inputs (i.e. providing two different documents) will result in two different hash values.

7.2. Properties of Hash Functions

Cryptographic hash functions have several useful properties. The most significant include the following:

- **One-way:** The hash function can be computed efficiently: Given x, it is easy to compute $H(x)$. However, given a hash output y, it is infeasible to find any input x such that $H(x) = y$. (This property is also known as "**preimage resistant**.") Intuitively, the one-way property claims that given an output of a hash function, it is infeasible for an adversary to find *any* input that hashes to the given output.
- **Second preimage resistant:** Given an input x, it is infeasible to find another input x' such that $x' \neq x$ but $H(x) = H(x')$. This property is closely related to *preimage resistance*; the difference is that here the adversary also knows a starting point, x, and wishes to tweak it to x' in order to produce the same hash—but cannot. Intuitively, the second preimage resistant property claims that given an input, it is infeasible for an adversary to find another input that has the same hash value as the original input.
- **Collision resistant:** It is infeasible to find *any* pair of messages x, x' such that $x' \neq x$ but $H(x) = H(x')$. Again, this property is closely related to the previous ones. Here, the difference is that the adversary can

freely choose their starting point, x, potentially designing it specially to enable finding the associated x'—but again cannot. Intuitively, the collision resistance property claims that it is infeasible for an adversary to find *any* two inputs that both hash to the same value. While it is impossible to design a hash function that has absolutely no collisions since there are more inputs than outputs (remember the pigeonhole principle), it is possible to design a hash function that makes finding collisions *infeasible* for an attacker.

By "infeasible", we mean that there is no known way to accomplish it with any realistic amount of computing power.

Note, the third property implies the second property. Cryptographers tend to keep them separate because a given hash function's resistance towards the one might differ from its resistance towards the other (where resistance means the amount of computing power needed to achieve a given chance of success).

Under certain threat models, hash functions can be used to verify message integrity. For instance, suppose Alice downloads a copy of the installation disk for the latest version of the Ubuntu distribution, but before she installs it onto her computer, she would like to verify that she has a valid copy of the Ubuntu software and not something that was modified in transit by an attacker. One approach is for the Ubuntu developers to compute the SHA256 hash of the intended contents of the installation disk, and distribute this 256-bit hash value over many channels (e.g., print it in the newspaper, include it on their business cards, etc.). Then Alice could compute the SHA256 hash of the contents of the disk image she has downloaded, and compare it to the hash publicized by Ubuntu developers. If they match, then it would be reasonable for Alice to conclude that she received a good copy of the legitimate Ubuntu software. Because the hash is collision-resistant, an attacker could not change the Ubuntu software while keeping the hash the same. Of course, this procedure only works if Alice has a good reason to believe that she has the correct hash value, and it hasn't been tampered with by an adversary. If we change our threat model to allow the adversary to tamper with the hash, then this approach no longer works. The adversary could simply change the software, hash the changed software, and present the changed hash to Alice.

7.3. Hash Algorithms

Cryptographic hashes have evolved over time. One of the earliest hash functions, MD5 (Message Digest 5) was broken years ago. The slightly more recent SHA1 (Secure Hash Algorithm) was broken in 2017, although by then it was already suspected to be insecure. Systems which rely on MD5 or SHA1 actually resisting attackers are thus considered insecure. Outdated hashes have also proven problematic in non-cryptographic systems. The `git` version control program, for example, identifies identical files by checking if the files produce the same SHA1 hash. This worked just fine until someone discovered how to produce SHA1 collisions.

Today, there are two primary "families" of hash algorithms in common use that are believed to be secure: SHA2 and SHA3. Within each family, there are differing output lengths. SHA-256, SHA-384, and SHA-512 are all instances of the SHA2 family with outputs of 256b, 384b, and 512b respectively, while SHA3-256, SHA3-384, and SHA3-512 are the SHA3 family members.

For the purposes of the class, we don't care which of SHA2 or SHA3 we use, although they are in practice very different functions. The only significant difference is that SHA2 is vulnerable to a *length extension attack*. Given $H(M)$ and the length of the message, but not M itself, there are circumstances where an attacker can compute $H(M\|M')$ for an arbitrary M' of the attacker's choosing. This is because SHA2's output reflects all of its internal state, while SHA3 includes internal state that is never outputted but only used in the calculation of subsequent hashes. While this does not violate any of the aforementioned properties of hash functions, it is undesirable in some circumstances.

In general, we prefer using a hash function that is related to the length of any associated symmetric key algorithm. By relating the hash function's output length with the symmetric encryption algorithm's key length, we can ensure that it is equally difficult for an attacker to break either scheme. For example, if we are using AES-128, we should use SHA-256 or SHA3-256. Assuming both functions are secure, it takes 2^{128} operations to brute-force a 128 bit key and 2^{128} operations to generate a hash collision on a 256 bit hash function. For longer key lengths, however, we may relax the hash difficulty. For example, with 256b AES, the NSA uses SHA-384, not SHA-512, because, let's face it, 2^{192} operations is already a hugely impractical amount of computation.

7.4. Lowest-hash scheme

Cryptographic hashes have many practical applications outside of cryptography. Here's one example that illustrates many useful properties of cryptographic hashes.

Suppose you are a journalist, and a hacker contacts you claiming to have stolen 150 million (150 million) records from a website. The hacker is keeping the records for ransom, so they don't want to present all 150 million files to you. However, they still wish to prove to you that they have actually stolen 150 million different records, and they didn't steal significantly fewer records and exaggerate the number. How can you be sure that the hacker isn't lying, without seeing all 150 million records?

Remember that the outputs of cryptographic hashes look effectively random–two different inputs, even if they only differ in one bit, give two unpredictably different outputs. Can we use these random-looking outputs to our advantage?

Consider a box with 100 balls, numbered from 1 to 100. You draw a ball at random, observe the value, and put it back. You repeat this n times, then report the lowest number you saw in the n draws. If you drew 10 balls (n=10), you would expect the lowest number to be approximately 10. If you drew 100 balls (n=100), you might expect the lowest number to be somewhere in the range 1-5. If you drew 150 million balls (n=150,000,000), you would be pretty sure that the lowest number was 1. Someone who claims to have drawn 150 million balls and seen a lowest number of 50 has either witnessed an astronomically unlikely event, or is lying about their claim.

We can apply this same idea to hashes. If the hacker hashes all 150 million records, they are effectively generating 150 million unpredictable fixed-length bitstrings, just like drawing balls from the box 150 million times. With some probability calculations (out of scope for this class), we can determine the expected range of the lowest hash values, as well as what values would be astronomically unlikely to be the lowest of 150 million random hashes.

With this idea in mind, we might ask the hacker to hash all 150 million records with a cryptographic hash and return the 10 lowest resulting hashes. We can then check if those hashes are consistent with what we would expect the lowest 10 samples out of 150 million random bitstrings to be. If the hacker only hashed 15 million records and returned the lowest 10 hashes, we should notice that the probability of getting those 10 lowest hashes from 150 million records is astronomically low and conclude that the hacker is lying about their claim.

What if the hacker tries to cheat? If the hacker only has 15 million records, they might try to generate 150 million fake records, hash the fake records, and return the lowest 10 hashes to us. We can make this attack much harder for the attacker by requiring that the attacker also send the 10 records corresponding to the lowest hashes. The hacker won't know which of these 150 million fake records results in the lowest hash, so to guarantee that they can fool the reporter, all 150 million fake records would need to look valid to the reporter. Depending on the setting, this can be very hard or impossible: for example, if we expect the records to be in a consistent format, e.g. `lastname, firstname`, then the attacker would need to generate 150 million fake records that follow the same format.

Still, the hacker might decide to spend some time *precomputing* fake records with low hashes before making a claim. This is called an *offline attack*, since the attacker is generating records offline before interacting with the reporter. We will see more offline attacks when we discuss password hashing later in the notes. We can prevent the offline attack by having the reporter choose a random word at the start of the interaction, like "fubar," and send it to the hacker. Now, instead of hashing each record, the hacker will hash each record, concatenated with the random word. The reporter will give the attacker just enough time to compute 150 million hashes (but no more) before requesting the 10 lowest values. Now, a cheating hacker cannot compute values ahead of time, because they won't know what the random word is.

A slight variation on this method is to hash each record 10 separate times, each with a different reporter-chosen random word concatenated to the end (e.g. "fubar-1," "fubar-2," "fubar-3," etc.). In total, the hacker is now hashing 1.5b (150 million times 10) records. Then, instead of returning the lowest 10 hashes overall, the hacker returns the record with the lowest hash for each random word. Another way to think of this variation is: the hacker hashes all 150 million records with the first random word concatenated to each record, and returns the record with the lowest hash. Then the hacker hashes all 150 million records again with the second random word concatenated to each record, and returns the record with the lowest hash. This process repeats 10 times until the hacker has presented 10 hashes. The math for using the hash values to estimate the total number of lines is slightly different in this variation (the original uses random selection without substitution, while the variant uses random selection with substitution), but the underlying idea is the same.