CS 198 Codebreaking at Cal
Spring 2023 Course Notes

Note 4

Hashing

Hash functions (for our purposes) translate some arbitrary-length input into a fixed-length output that is seemingly random (but deterministic).

For example, the SHA-3 hash of "mypassword" is 0551e8e8f6e8ea3136a21857797b7 355e3e9f9a9b 48b49256104936dbb9fc628. SHA3("mypassword") is the same regardless of who calculates it, and no secret key is required.

These outputs can act as a "tag" to verify two sets of data are the same. If the hashes are equal, then the data is extremely likely to be the same. For the rest of this note, a hash function will be written as $H(m)$, where H is the function and m is the data.

A **cryptographic hash function**, or CHF, is a special class of hash functions that are imbued with three key properties that make them suitable for use in cryptography. These properties are:

1. **Preimage Resistance**
   Given $H(m) = h$, it should be infeasible to recover m. Informally, the hash function should not be reversible except through brute force.

2. **Second-Preimage Resistance**
   Given m_1, it should be infeasible to find m_2 such that $H(m_1) = H(m_2)$. Informally, we should not be able to easily find another value that maps to a given hash value.

3. **Collision Resistance**
   It should be infeasible to find m_1, m_2 such that $H(m_1) = H(m_2)$ and $m_1 \neq m_2$. Informally, we should not be able to easily find two different values to map to the same hash.

Collision resistance turns out to be a stronger version of second-preimage resistance, so we mainly focus on 1 and 3.

Merkle-Damgard Construction

The basic design of cryptographic hash functions can be traced backed to a PhD thesis from Ralph Merkle in 1979. We want to take some input M and produce some fixed-length output h. To do so, we first come up with a *compression function* F that takes in two fixed-length inputs (in our example, 256 bits) and outputs exactly that many bits (256). This compression function is necessarily collision-resistant. These compression functions are out of scope for this class, however, they usually involve encrypting via a block cipher.

Next, we split out input m into 256-bit blocks of data. If m is not perfectly divisible by 256, we add some *padding* to the end of the last block to make it exactly 256 bits long. (More on padding schemes later)

To begin, we define out first initialization vector (IV). The first step is defined as $h_0 = f(IV, m_0)$. At any given step k, we have the previous compression function output (named h_{k-1}) and the next block of data to be hashed m_k. Therefore, $h_k = f(h_{k-1}, m_k)$. We repeat this until we run out of data blocks to hash, and produce h as our final hash value.

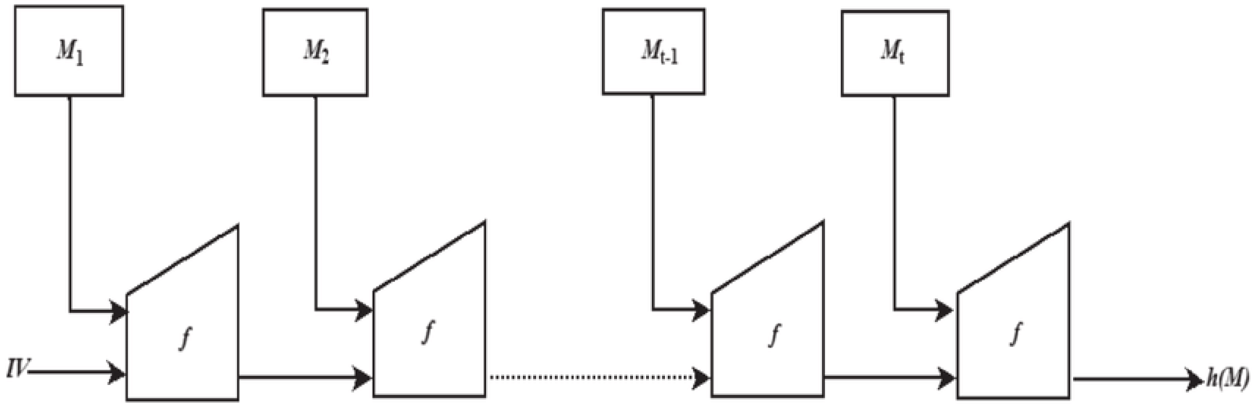

Image credit to Tiwari, Harshvardhan from Merkle-Damgård Construction Method and Alternatives: A Review.

Applications of Hashing

Hashing is extremely important for various applications within computer science. Perhaps the most relevant is *password hashing*, in which websites store your password as its hash rather than its actual value. This way, when a database is leaked, attackers can only see your username and your hashed password, rather than your username and full password.

When the user sends their username and password (p) to the website, the server computes $H(p)$ and checks if password[username] = $H(p)$. If these two values equal, then the user has successfully logged in – otherwise, the password is incorrect.

We know that it is hard to recover m given $H(m)$, so it is similarly difficult for the attacker to know your password from the actual hash. However, storing a hash instead of a password brings up a new issue as well – **hash collisions**. As previously mentioned, a *collision* occurs when $H(m_1) = H(m_2)$ for some $m_1 \neq m_2$. Looking back to our user login example, if the attacker were able to find some m_2 such that $H(p) = H(m_2)$, they could send m_2 as the "password" and it would be accepted by the server, *despite not actually being the user's password.*

If this is alarming to you, don't worry. Modern hash functions have strong evidence in favor of their collision resistance – for example, finding a collision with the popular SHA-3 hash function is $2^{\frac{b}{2}}$ for *any* collision, and 2^b for a specific collision (like the password example). Since hashes are often 128 bits, this is completely impractical to attack.

Length-Extension Attacks

A large flaw of the Merkle-Damgard construction comes in the form of *length-extension attacks*, in which attacks can append arbitrary amounts of data to the end of a hash. Formally, given $H(m_1)$, they can find $H(m_1||k)$, for arbitrary k (where || means concatentation, or adding k to the end of m_1). This can pose a large problem to various hashing schemes. Suppose you knew the hash of some signature ($H(key||data)$), a hash that is created with some secret key. We want to add some data afterwards while maintaining the signature's validity. Normally, you'd have to hash every possible input until one matches to figure out the key. Since we can append arbitrary data, however, we can turn this into adding data to the end of an existing hash, without knowing the contents of the earlier hash.

This is derived from the fact that the output h of a Merkle-Damgard construction hash function is basically just the internal state after the input ran out. We can "pretend" the hash function hasn't finished and keep on adding new data, even if we didn't know the previous data. Say we know $H(key||data) = h_k$ – we know from the previous section that $h_{k+1} = f(h_k, m_{k+1})$. We can therefore add another data block m_{k+1} to create $H(m_k||m_{k+1})$ **without knowing** m_k! We then repeat this to add whatever data we want – in the signature example, we add a string like "...pay Bob 5 thousand".

Birthday Paradox

Since we've mentioned this before, now is a good time to cover the Birthday Paradox, a very important concept in cryptography as a whole (especially hashing).

The premise is as follows: given a group of n people, the probability of at least one pair sharing the same birthday is $\approx 50\%$ when $n = 23$, and quickly increases beyond that. The following graph demonstrates the probabilities for any given group size:

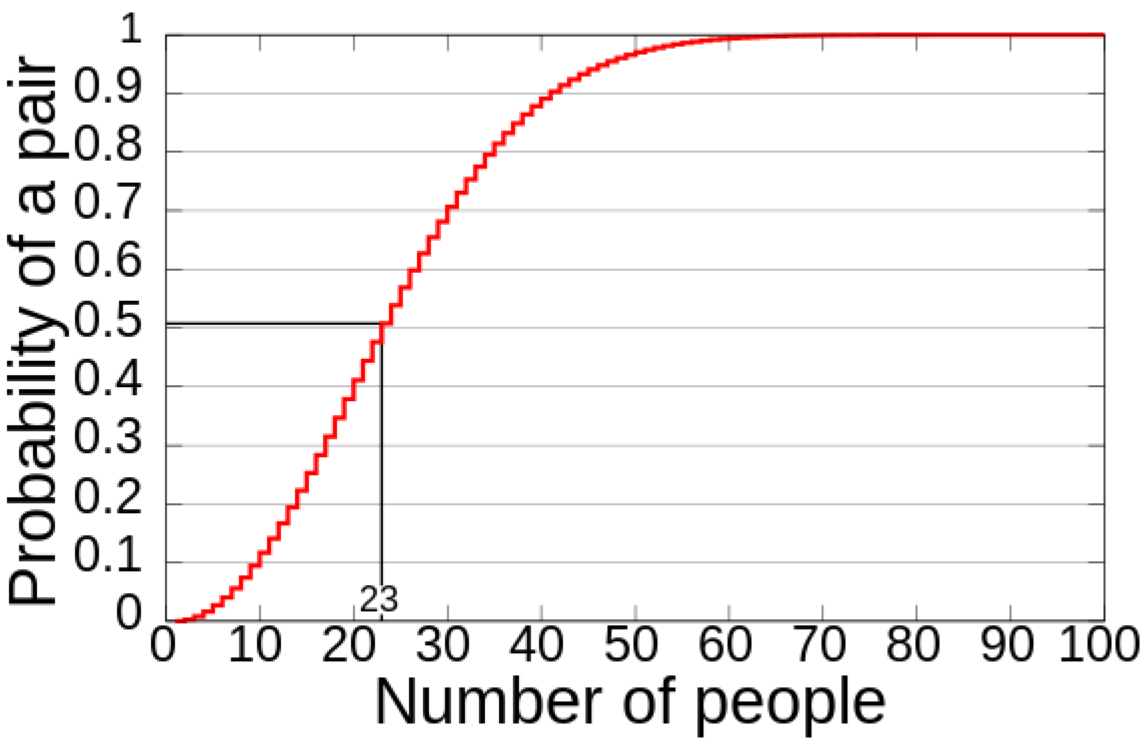

Image credit to https://en.wikipedia.org/wiki/Birthday_problem

The "paradox" nomenclature comes from the unintuitive nature of this phenomenon – there are 365 days in a year, so how is it so likely that two people share a birthday with only 23 people? This comes from the fact that we must consider *every possible pair* of people, instead of one person compared with 22 others. There are exactly $\binom{n}{2} = \frac{n!}{2!(n-2)!} = \frac{n(n-1)}{2}$ pairs. If you're unfamiliar with the notation, that's OK – $\binom{n}{2}$ means "how many ways can we choose two different people from n total people. Each pair has a $\frac{363}{365}$ chance of not having the same birthday. Therefore, the probability that no pair shares a birthday is:

$$\left(\frac{364}{365}\right)^{\frac{n(n-1)}{2}}$$

To find the probability that at least one person shares a birthday, we can subtract this from -1 to get

$$1 - \left(\frac{364}{365}\right)^{\frac{n(n-1)}{2}}$$

There are many ways of approximating the probability of a birthday collision, most of which require some probability theory out of scope. I recommend the CS 70 Note 18 if you've taken some probability theory before, or Googling Stirling's approximation for factorials. For now, we will take it that you need $\approx 1.17\sqrt{n}$ people to have a 50% chance of collision with sample space n. Sample space in this instance refers to how many possibilities each person can have (i.e. days in the year). We further approximate this to $O(\sqrt{n})$.

Returning to hash collisions – the birthday paradox states we will only need to try $\approx \sqrt{n}$ inputs on n total values until we find a collision. This means for a b-bit hash, we only have to try $\sqrt{2^b} = 2^{\frac{b}{2}}$ inputs! While named a "birthday attack", this is not an attack anybody can defend against – it comes as a fundamental consequence of the nature of hash functions having a fixed output length. It acts as the "brute force" cost needed to find *any* collision.

Message Authentication Codes

So far, we've been largely focused on the concept of **confidentiality**, or the property of protecting information from being read by eavesdropper. Now we will focus on **integrity** and **authenticity**, which entail ensuring the message wasn't edited mid-transmission and ensuring it was sent by the person we are expecting it from.

First, for integrity: we need to construct some function that lets us verify that the message wasn't tampered with. This function will output some tag which we add to the end of the message, and the receiver can then check for themselves whether it matches. Call this function $MAC(K,M)$. There will be some secret key K that all parties in the conversation share (like a normal symmetric key).

If Alice wanted to send a message M to Bob (not a secret message), she could send $(M, MAC(K,M))$. Mallory (an eavesdropper who can also modify messages) can read M, but she cannot change M. This is because Bob will then compute $MAC(K,M')$ and see that $MAC(K,M') \neq MAC(K,M)$, and reject the message. Even if Mallory tries to change $MAC(K,M)$, she cannot do so such that it matches M' without knowing K.

Formally, we expect the following properties:

1. **Deterministic**
   The MAC must be determinsitic given the same set of arguments, so that anyone can compute it and end up with the same result.

2. **Unforgeable**
   Without knowing the secret key K, it should be impractical to create a MAC on any message. Formally, given any number of past pairs of $(M, MAC(K,M))$, it is practically impossible to find $MAC(K,M')$ for some $M' \neq M$.

With this scheme, Alice and Bob can always ensure the integrity of their messages! But how is this accomplished?

Constructing MACs

It turns out there exists a convenient and secure method of constructing MACs using cryptographic hash functions, named HMAC. First, let's revisit some key concepts from hash functions: they are **one-way**, meaning we can't easily find m from $H(m)$, and computationally indistinguishable from random noise. The latter part means the output of a hash function looks very similar to random bits, so we can't tell how close two messages are by comparing their hashes.

First, let's try constructing something known as the NMAC. Say Alice and Bob share two symmetric, 128 bit keys K_1 and K_2. If we just did $NMAC(K_1, K_2, M) = H(K_1||M)$, then we could run into length-extension attacks as seen in the last week's note. We can fix this by adding a second layer of hash: $NMAC(K_1, K_2, M) = H(K_2||H(K_1||M))$, where $K_1 \neq K_2$. Why we need different keys is a outside of the scope of this course, and has to do with reducing the security of NMAC to its underlying hash function. You can read this paper for more information if you are interested.

If we tried to add some x' to the end of this new MAC, we could only find $H(K_2||H(K_1||M)||x')$, which will not evaluate to the intended MAC (which would need to be $H(K_2||H(K_1||M||x')))$

With that, we are equipped to define the HMAC:

$$\text{HMAC}(K, M) = H(K \oplus opad||H((K \oplus ipad)||M))$$

where

$$\begin{aligned} ipad &= \text{0x363636......} \\ opad &= \text{0x5C5C5C...} \end{aligned}$$

That formula looks a bit complicated at first glance, but let's break it down piece by piece. In the NMAC we needed two different keys K_1 and K_2, but it turns out that they don't need to be completely unrelated, just different in at least one bit. By XOR-ing with these two pads, we can make $K_2 = K \oplus opad$ and $K_1 = K \oplus ipad$, and only have to use one key K.

Why these specific constants? They are chosen to provide very different bit patterns, but theoretically we only need 1 bit difference. Cryptographers tend to be a bit paranoid, however, and for good reason!