Collision Resistance and File Authentication

6.1600 Course Staff

Fall 2023

In the last chapter, we focused on authenticating people—ensuring that a person (or a request on behalf of that person) is likely who they claim to be. In this chapter, we will focus on authenticating files, code, and other data. When we say that we want to authenticate a file, we mean that we want to verify that the file's contents are exactly as they were when we or someone we trust last viewed them. The key new tool we use to do so is a *collision-resistant hash function*.

Disclaimer: This set of notes is a work in progress. It may have errors and be missing citations. It is certainly incomplete. Please let the staff know of any errors that you find.

1 *Intuition: Collision resistance*

For our purposes, a hash function H maps a bitstring of any length onto a fixed-size space of outputs, so the type signature is $H : \{0,1\}^* \to \{0,1\}^\lambda$.

In order for a hash function to be *collision-resistant*, we want it to be the case that for any input, the generated output should be "unique." Of course, it cannot really be unique—we are mapping infinitely many inputs onto finitely many outputs—but we want it to be *computationally infeasible* to find a pair of distinct inputs that have the same hash values (a "collision").

> **Security goal:** A hash function H is *collision resistant* if it is "computationally infeasible" to find two distinct strings x and x' such that $H(x) = H(x')$.

Given a long message m, it's hash $H(m)$ under a collision-resistant hash function is like a short "fingerprint" of the message—the hash essentially uniquely identifies the message m. For that reason, collision-resistant hash functions let you authenticate a long message m by authenticating the short fixed-length string $H(m)$. We often call the hash value $H(m)$ a *digest*.

1.1 *Applications*

Secure File Mirroring. Often a user wants to download large files (e.g., software updates) from a far-away server. To speed up this process, a company or Internet-service provider may set up local *mirrors* of the remote files. Users can then download the files from the nearby mirror instead of the far-away server. However, without additional security measures, the mirror may server users a different

file than the one the mirror fetched from the origin server. If the mirror is malicious, it can, for example, trick the user into installing a backdoored software update. (We saw an attack based on mirrors in **??**.)

To protect against a malicious mirror, we can add some authentication on the file that the mirror hosts. Say that the origin server publishes a large software update f. The origin server will send the file f to its mirrors and the origin server itself will serve the hash digest $d \leftarrow H(f)$ to anyone who asks for it. A user who wants to fetch the update can download d from the origin server directly—this will be fast since the digest is tiny. Then, the client can fetch the update itself from a (potentially untrustworthy) mirror. When the client receives a file $\hat{f}$ from the mirror, it can check that $d = H(\hat{f})$ to ensure that $\hat{f}$ is the true software update. If H is collision resistant, then if the hash value $H(\hat{f})$ matches the origin server's digest d, the files are almost certainly identical.

Subresource Integrity. If a program fetches a file from some content delivery network, it can store the hash of that file locally and use it to verify that the contents of the file did not change since the application was developed.

Outsourced File Storage. If you want to store your files on a cloud provider, you want to be sure that the cloud provider does not maliciously modify the files without you noticing. To make sure of this, you can store $H(\text{files})$ locally, which takes very little storage space. Then, when you redownload your files locally, you can recompute the hash to verify that they were not tampered with.

2 *Defining collision resistance (slightly more formally)*

An adversary's goal in breaking a collision resistant hash function is to find a collision—a pair of values $m_0, m_1 \in \{0,1\}^*$ such that $m_0 \neq m_1$ and $H(m_0) = H(m_1)$.

Definition 2.1 (Collision Resistance)**.** A function $H : \{0,1\}^* \to \{0,1\}^\lambda$ is collision-resistant if for all "efficient" adversaries $\mathcal{A}$, we have that:

$$\Pr[H(m_0) = H(m_1), m_0 \neq m_1 : (m_0, m_1) \leftarrow \mathcal{A}()] \leq \text{“negligible”}$$

In words, this means that the probability of finding a collision is so small that no efficient adversary could hope to do it.

There are two ways of thinking about the terms "efficient" and "negligible" that we use in this definition—one mindset we use in practice and the other mindset we use in theory.

- In theory...
  - All of our cryptographic constructions are parameterized by an integer $\lambda \in \{1, 2, 3, \dots\}$ that we call the *security parameter*. So instead of a single collision-resistant hash function H, we have a family of functions $\{H_1, H_2, H_3, \dots\}$, where the function H_λ has λ-bit output.
  - An "efficient" algorithm is a randomized algorithm that runs in time polynomial in λ.
  - A "negligible" function is one that grows slower than the inverse of every polynomial—a function that is $O(\frac{1}{\lambda^c})$ for all constants $c \in \mathbb{N}$.
- In practice...
  - We use a fixed hash function H with a fixed-length output, which might be as 256 or 512 bits.
  - An "efficient" adversary is one that runs in time $\leq 2^{128}$.
  - A "negligible" probability is some very small constant, like one smaller than 2^{-128}.

2.1 *Understanding which attacks are feasible*

Typically, we think of an attack that runs in more than 2^{128} time as infeasible and an event that happens with probability less than 2^{-128} is one that will never happen. These seemingly magic constants come from empirical considerations:

2^{30}	operations/second on a laptop
2^{58}	ops/sec on Fugaku supercomputer
2^{68}	hashes/second on the Bitcoin network (as of Fall 2022)
2^{92}	hashes/yr on the Bitcoin network
2^{114}	hashes required to use enough energy to boil the ocean
2^{140}	hashes required to use one year of the sun's energy

See Lenstra, Kleinjung, and Thomé for an entertaining discussion of these constants.[1]

[1] Arjen K Lenstra, Thorsten Kleinjung, and Emmanuel Thomé. "Universal security". In: *Number Theory and Cryptography*. 2013.

2^{-1}	fair coin lands heads
2^{-13}	probability that a randomly sampled MIT grad is a Nobel Prize winner
2^{-19}	probability of being struck by lightning next year
2^{-28}	probability of winning the Mega Millions jackpot
2^{-128}	will essentially never happen

The takeaway is that if an attacker finds a collision with probability 2^{-128}, we can be extremely sure that a collision will never occur.

For most cryptosystems, there is a tradeoff between the attacker's running time and success probability. For example, an attacker running in time T can find a collision in a hash function with n-bit output with probability $T^2/2^n$. So, as the attack runs for more time, it has a better chance of finding a collision.

3 *Constructing a collision-resistant hash function*

The current standard for fast collision-resistant hashing is SHA256 (a.k.a. SHA2), which was designed by the NSA in 2001. The SHA2 hash functions are designed using the following common two-step approach:

1. Build a small collision-resistant hash function on a fixed-size domain $H_{small} : \{0,1\}^{2\lambda} \to \{0,1\}^{\lambda}$. This step is, to some degree, "more art than science". The standard practice is to design a hash function that defeats all known collision-finding attacks. If no known attack works well, we declare the candidate function to be collision resistant.
2. Use H_{small} to construct $H : \{0,1\}^* \to \{0,1\}^{\lambda}$. This can be done very cleanly using the "Merkle-Damgård" approach described below. This step requires no additional assumptions: we can prove unconditionally that if H_{small} is collision resistant, then H is as well.

We can also build collision-resistant hash functions that are secure under "nice" cryptographic assumptions, such as the assumption that factoring large numbers is hard. Unfortunately, hash functions based on these nice assumptions tend to be very slow and, as a result, are almost never used in practice.

3.1 *Merkle-Damgård*

The Merkle-Damgård construction gives a way to construct a collision-resistant hash function for all bitstrings (i.e., $\{0,1\}^*$) from a collision resistant hash function that maps 2λ-bit strings down to λ-bit strings, sketched out in Figure 1.

Another way to build collision-resistant hash functions is to use the so-called "sponge" construction. It is similar to the approach described here in that we start with a small primitive, which we assume secure in some sense, and then we use the small primitive to build a hash function on a large domain.

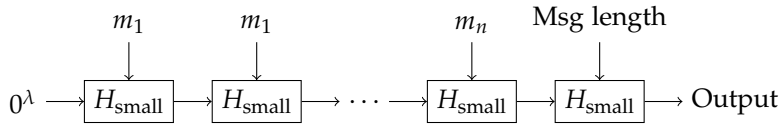

Figure 1: Sketch of the Merkle-Damgård construction for a collision-resistant hash function.

The Merkle-Damgård construction first splits the message into λ-sized blocks $[m_1, \ldots, m_n]$ and successively hashes them together. In the following pseudocode, the function ToBlock converts an integer, representing the length of the input message in blocks, into a λ-bit string. Then the Merkle-Damgård construction is: (Here, we are assuming that the message is at most $\lambda 2^{\lambda}$ bits long.)

We won't prove it here, but we can use the fact that H_{small} is collision-resistant to prove that H must also be collision-resistant. The basic idea of the proof is to show that given a collision in H, we can easily compute a collision in H_{small}.

In practice, standard hash functions have limits on the length of the messages that they can hash. For example, SHA256 can hash messages of length up to $2^{64} - 1$ bits.

Note: In the Merkle-Damgård construction of Fig. 2, we initialize the variable b to the all-zeros string. The construction is collision-resistant if we omit the all-zeros string and start by setting $b \leftarrow m_1$

$H(m_1, \ldots, m_n)$: // Merkle-Damgård construction

- Let $b \leftarrow 0^\lambda$.
- For $i = 1, \ldots, n$:
  - Let $b \leftarrow H_{\text{small}}(b, m_i)$.
- Let $b \leftarrow H_{\text{small}}(b, \mathsf{ToBlock}(n))$.
- Output b.

Figure 2: The Merkle-Damgård construction of a large-domain collision-resistant hash function H from a small-domain collision-resistant hash function H_{small}.

and then continue by hashing $m_2, m_3, \ldots$. The construction is *not* collision resistant if we omit the length block $\mathsf{ToBlock}(n)$ that we hash in at the end.

3.2 *The Birthday Paradox*

If you sample $2^{\lambda/2}$ random 10λ-bit strings and hash them with a hash function that has λ-bit outputs, you will find a collision among these inputs with constant probability.

An important thing to understand when dealing with hash functions is the "Birthday Paradox," which states that given a hash function with λ-bit output, you can always find a collision in time $O(\sqrt{2^\lambda}) = O(2^{\lambda/2})$. So, if you want to force an attacker to use at least 2^{128} to find a collision, you must use a hash function with at least 256 bits of output.

3.3 *Domain Separation*

In many applications, we have a one-input CRHF (such as SHA256) $H : \{0,1\}^* \to \{0,1\}^{256}$ and we need to construct a two-input CRHF $H_2(x, y)$.

Bad idea. An obvious solution to construct the two-input hash function H_2 is to concatenate the two values, so that $H_2(x, y) = H(x||y)$. However, this construction allows two different pairs of messages to hash to the same value:

$$H_2(\text{"key"}, \text{"value"}) = H_2(\text{"ke"}, \text{"yvalue"}).$$

Both Amazon and Flickr had a bug arising from this—they concatenated all parameters before hashing, and had parameters such that two different intents had the same concatenation.[2]

[2] Thai Duong and Juliano Rizzo. *Flickr's API Signature Forgery Vulnerability*. `https://vnhacker.blogspot.com/2009/09/flickrs-api-signature-forgery.html`. Sept. 2009.

3.4 *Length-Extension*

Recall the concept of Message Authentication Codes (MAC) from the last lecture—a code that can be sent along with a message to verify

that the message was not changed. (We will see the formal definition in **??**.)

Bad idea. Poorly designed software uses $\mathsf{MAC}(k, m) = H(k\|m)$ as a very simple construction of a MAC. However, this construction has an easy attack—given $\mathsf{MAC}(k, m)$, it is easy to compute $\mathsf{MAC}(k, m\|m')$ without knowing the key k if H is a hash function built with the Merkle-Damgård construction. To do this, the attacker hashes the output of $\mathsf{MAC}(k, m)$ with two more blocks—a new message m'' and another length block. Now, we have computed $\mathsf{MAC}(k, m\|m')$ where m' is the original length block plus some custom message without knowing the key k.

This problem here is that we were using a hash function that was *only* guaranteed to be collision resistant, but we assumed that it had other properties (such as that it is guaranteed to be difficult to compute the hash of an extension of the original message). Figure 3 sketches out the length-extension attack on the Merkle-Damgård construction from Figure 1.

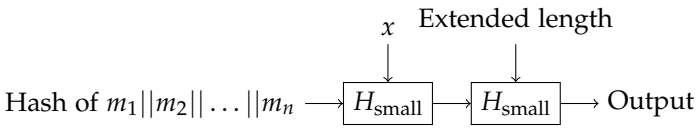

Figure 3: Sketch of the extension attack on the Merkle-Damgård construction, starting with a hash of $m_1||m_2||\ldots||m_n$ to compute the hash of $m_1||m_2||\ldots||m_n||\mathsf{ToBlock}(n)||x$.

4 *Applications: Merkle Trees*

In many settings, an origin server has N files (e.g., Android app binaries) and wants to serve these files from potentially untrustworthy mirror servers (e.g., Akamai servers) distributed around the globe.

To do this, the origin server can put the N files at the leaves of a binary tree. Then the server hashes together pairs of files, then hashes each pair of hashes and so on until it eventually ends up with a single root hash h. The client fetches the root hash h from the origin server directly.

Later on, the client can download any one of the N files from the untrustworthy mirror server. The mirror can produce the file, along with $O(\log N)$ hashes—the sibling nodes of each node on every path from the file's leaf to the root. The client can use the root hash h it got from the origin server, along with the additional hashes from the mirror server, to be convinced that the mirrored file it downloaded was authentic.

TODO: Add diagram from lecture.

References

Duong, Thai and Juliano Rizzo. *Flickr's API Signature Forgery Vulnerability*. `https://vnhacker.blogspot.com/2009/09/flickrs-api-signature-forgery.html`. Sept. 2009.

Lenstra, Arjen K, Thorsten Kleinjung, and Emmanuel Thomé. "Universal security". In: *Number Theory and Cryptography*. 2013.