

Collision Resistance and File Authentication

6.1600 Course Staff

Fall 2026

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 $H : \{0,1\}^* \rightarrow \{0,1\}^\lambda$.

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 , its 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 serve 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.

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 $H : \{0, 1\}^* \rightarrow \{0, 1\}^\lambda$ 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 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^{70} hashes/second on the Bitcoin network (as of Fall 2026)
- 2^{95} 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

- 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.

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

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.

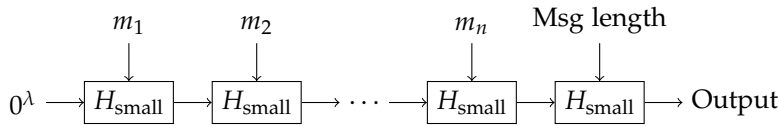
3 Constructing a collision-resistant hash function

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 $H_{\text{small}} : \{0,1\}^{2\lambda} \rightarrow \{0,1\}^\lambda$. 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 $H : \{0,1\}^* \rightarrow \{0,1\}^\lambda$. 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.

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.



The Merkle-Damgård construction first splits the message into λ -sized blocks $[m_1, \dots, m_n]$ 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} .

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 $b \leftarrow m_1$

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.

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.

In practice, standard hash functions have their own 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.

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

- Let $b \leftarrow 0^\lambda$.
- For $i = 1, \dots, n$:
 - Let $b \leftarrow H_{\text{small}}(b, m_i)$.
- Let $b \leftarrow H_{\text{small}}(b, \text{ToBlock}(n))$.
- 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 $m_2, m_3, \dots$. The construction is *not* collision resistant if we omit the length block $\text{ToBlock}(n)$ that we hash in at the end.

Why the length block is necessary. Suppose we delete the final step $b \leftarrow H_{\text{small}}(b, \text{ToBlock}(n))$ from Figure 2, and call the resulting function H^- . Collisions between two messages with the *same* number of blocks still yield collisions in H_{small} : walking backwards from the last block, at each step either the two inputs to H_{small} differ—giving a collision—or they agree and we continue. So any attack must use messages of *different* lengths, and the job of $\text{ToBlock}(n)$ is exactly to keep those apart.

To see that H^- really is broken, it is not enough to exhibit an attack against some particular hash function: we must show that the collision resistance of H_{small} does not *imply* that of H^- . We do this by constructing an H_{small} that is collision resistant and yet makes H^- trivially insecure.

Construction 3.1 (A collision-resistant H_{small} that breaks H^-). Let $G: \{0, 1\}^{2\lambda} \rightarrow \{0, 1\}^{\lambda-1}$ be any collision-resistant function, and define $H_{\text{small}}: \{0, 1\}^{2\lambda} \rightarrow \{0, 1\}^\lambda$ by

$$H_{\text{small}}(b, m) = \begin{cases} 0^\lambda & \text{if } b \parallel m = 0^{2\lambda}, \\ 1 \parallel G(b, m) & \text{otherwise.} \end{cases}$$

Claim. If G is collision resistant, then so is H_{small} .

Proof. Let $x \neq x'$ satisfy $H_{\text{small}}(x) = H_{\text{small}}(x')$. The two outputs have the same leading bit, so either both begin with 1, in which case $G(x) = G(x')$ and we have a collision in G ; or both equal 0^λ , which forces $x = x' = 0^{2\lambda}$ and contradicts $x \neq x'$. $\square$

By construction H_{small} has a *fixed point at the initialization value*:

$$H_{\text{small}}(0^\lambda, 0^\lambda) = 0^\lambda.$$

This is the only property of H_{small} the attack uses. Any H_{small} with $H_{\text{small}}(0^\lambda, m^*) = 0^\lambda$ for some block m^* is equally fatal; prepending copies of m^* then leaves the hash unchanged.

Feeding an all-zeros block into the chaining variable while the chaining variable still holds the initialization value 0^λ leaves it unchanged. Prepending an all-zeros block to a message therefore adds one iteration that does nothing, and so for *every* message m ,

$$H^-(0^\lambda || m) = H^-(m).$$

Taking $n = 1$ and $n = 2$ gives an explicit collision. With $\lambda = 32$ and G instantiated by truncating SHA-256 to its first 31 bits, the two messages

$$M = 41414141 \quad \text{and} \quad M' = 00000000 || 41414141$$

both satisfy $H^-(M) = H^-(M') = \text{d740d7ef}$. Note that M and M' differ in content, not merely in length. The same trick works at any size: for $n = 2$ versus $n = 3$,

$$41414141 || 42424242 \quad \text{and} \quad 00000000 || 41414141 || 42424242$$

both hash to d314a585 . Degenerately, every all-zeros message $0^\lambda, 0^{2\lambda}, 0^{3\lambda}, \dots$ hashes to 0^λ .

Restoring the length block repairs all of this. The two messages above now reach the final step with the same chaining value d740d7ef but distinct second arguments $\text{ToBlock}(1)$ and $\text{ToBlock}(2)$, so their hashes ca06ae43 and dae0f489 could only coincide if H_{small} itself had a collision. In short: a fixed point lets messages with different block counts converge on a common chaining value, and $\text{ToBlock}(n)$ separates them again.

3.2 The Birthday Paradox

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.

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.

3.3 Domain Separation

In many applications, we have a one-input CRHF (such as SHA256) $H : \{0, 1\}^* \rightarrow \{0, 1\}^{256}$ 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

A better though more expensive way is

$$H_2(x, y) = H(H(x)||H(y)).$$

² Thai Duong and Julian Rizzo. *Flickr's API Signature Forgery Vulnerability*. <https://vnhacker.blogspot.com/2009/09/flickr-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 the lecture on message authentication codes.)

Bad idea. Poorly designed software uses $\text{MAC}(k, m) = H(k||m)$ as a very simple construction of a MAC. If H is built with the Merkle-Damgård construction, this has an easy attack: an adversary who sees a single valid tag can forge a tag on a *longer* message, without ever learning the key k .

The reason is that a Merkle-Damgård hash is its own chaining variable. Write n for the number of blocks in $k||m$, and let

$$t = \text{MAC}(k, m) = H(k||m).$$

By Figure 2, t is exactly the value of the chaining variable b after the construction has absorbed the n message blocks and the final length block $\text{ToBlock}(n)$. Nothing about t is secret or opaque: it is a snapshot of the entire internal state. An attacker who sets $b \leftarrow t$ and keeps iterating H_{small} is simply resuming the hash computation where the honest party stopped.

The attack. Let m'' be any suffix the attacker wishes to append, padded to a whole number of blocks. Given m and t , the attacker computes

1. $b \leftarrow H_{\text{small}}(t, m'')$, and
2. $b \leftarrow H_{\text{small}}(b, \text{ToBlock}(n + 2))$,

and outputs b . This is a valid tag on a message the attacker never had the key to authenticate.

Which message was forged? Here is the subtle point. The attacker resumed the chain *after* the length block $\text{ToBlock}(n)$ had already been absorbed, so that block is now an unavoidable part of the string being hashed. What the attacker has computed is

$$b = H\left(\underbrace{k\|m}_{\text{original}} \parallel \underbrace{\text{ToBlock}(n)}_{\text{forced "glue"}} \parallel \underbrace{m''}_{\text{attacker's choice}}\right) = \text{MAC}(k, m\|m'),$$

where

$$m' = \text{ToBlock}(n) \parallel m''.$$

So m'' is not an alternative name for m' : it is the tail of it. The attacker chooses m'' freely but cannot avoid the glue block that precedes it, which is why the forgery is a tag on $m\|m'$ rather than on the cleaner $m\|m''$. In practice this is rarely an obstacle—the glue is a short run of bytes, and many formats (query strings, log lines, serialized records) will happily ignore or tolerate it.

Two details are worth noting. First, the attacker must know $|k|$, though not k itself, since n depends on the key's length; in practice the key length is public, or the attacker simply tries each plausible value. Second, the attack needs no collision in H and no weakness in H_{small} whatsoever—it works against a perfectly good compression function.

The 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.

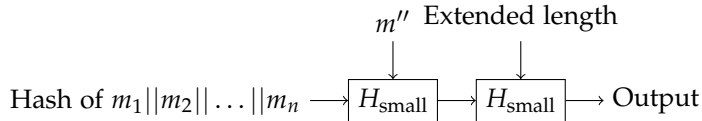


Figure 3: Sketch of the extension attack on the Merkle-Damgård construction, starting with a hash of $m_1\|m_2\|\dots\|m_n$ to compute the hash of $m_1\|m_2\|\dots\|m_n\|\text{ToBlock}(n)\|m''$.

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.

References

- Duong, Thai and Juliano Rizzo. *Flickr’s API Signature Forgery Vulnerability*. <https://vnhacker.blogspot.com/2009/09/flickr-api-signature-forgery.html>. Sept. 2009.
- Lenstra, Arjen K, Thorsten Kleinjung, and Emmanuel Thomé. “Universal security”. In: *Number Theory and Cryptography*. 2013.