CPSC 467: Cryptography and Computer Security

Michael J. Fischer

Lecture 16
October 30, 2017

Properties of Hash Functions

- Hash functions do not always look random
- Relations among hash function properties

Constructing New Hash Functions from Old

- Extending a hash function
- A general chaining method

Common Hash Functions

- SHA-2
- SHA-3
- MD5

Appendix: Birthday Attack Revisited

Properties of Hash Functions

Collision-resistance

Recall the three collision-resistance properties for a hash function $\mathcal{H}$ from lecture 15:

- One-way: Given $y \in \mathcal{H}$, it is hard to find $m \in \mathcal{M}$ such that $h(m) = y$. [1]
- Weakly collision-free: Given $m \in \mathcal{M}$, it is hard to find $m' \in \mathcal{M}$ such that $m' \neq m$ and $h(m') = h(m)$.
- Strongly collision-free: It is hard to find colliding pairs (m, m').

[1] More precisely, no probabilistic polynomial-time algorithm $A(y)$ succeeds with non-negligible probability at finding a pre-image $m \in h^{-1}(y)$, where y is chosen at random from $\mathcal{H}$ with probability proportional to $|h^{-1}(y)|$.

Hash values can look non-random

Intuitively, we like to think of $h(m)$ as being *"random-looking"*, with no obvious pattern.

Indeed, it would seem that obvious patterns and structure in h would provide a means of finding collisions, violating the property of being strong collision-free.

However, hash functions don't necessarily look random or share other properties of random functions, as I now show.

Example of a non-random-looking hash function

Suppose h is a strong collision-free hash function.

Define $H(m) = 0 \cdot h(m)$.

If (m, m') is a colliding pair for H, then (m, m') is also a colliding pair for h.

Hence, if we could find colliding pairs for H, we could find colliding pairs for h, contradicting the assumption that h is strong collision-free.

We conclude that H is strong collision-free, despite the fact that $H(m)$ always begins with 0.

A one-way function that is sometimes easy to invert

Let $h(m)$ be a cryptographic hash function that produces hash values of length n. Define a new hash function $H(m)$ as follows:

$$H(m) = \begin{cases} 0 \cdot m & \text{if } |m| = n \\ 1 \cdot h(m) & \text{otherwise.} \end{cases}$$

Thus, H produces hash values of length $n + 1$.

- $H(m)$ is clearly collision-free since the only possible collisions are for m's of lengths different from n.
- Any colliding pair (m, m') for H is also a colliding pair for h.
- Since h is collision-free, then so is H.

H is one-way

H is one-way, assuming uniformly distributed messages.

This is true, *even though H can be inverted for 1/2 of all possible hash values y*, namely, those that begin with 0.

The reason this doesn't violate the definition of one-wayness is that only 2^n values of m map to hash values that begin with 0, and all the rest map to values that begin with 1.

Since we are assuming $|\mathcal{M}| \gg |\mathcal{H}|$, the probability is small that a uniformly sampled $m \in \mathcal{M}$ has length exactly n.

We see that H is a cryptographic hash function, even though H does not look random.

Strong implies weak collision-free

There are some obvious relationships between properties of hash functions that can be made precise once the underlying definitions are made similarly precise.

Fact

If h is strong collision-free, then h is weak collision-free, assuming uniformly distributed messages.

Proof that strong ⇒ weak collision-free

Proof (Sketch).

Suppose h is *not* weak collision-free. We show that it is not strong collision-free by showing how to enumerate colliding message pairs.

The method is straightforward:

- Pick a random message $m \in \mathcal{M}$.
- Try to find a colliding message m'.
- If we succeed, then output the colliding pair (m, m').
- If not, try again with another randomly-chosen message.

Since h is not weak collision-free, we will succeed in finding m' for a significant number of m. Each success yields a colliding pair (m, m'). □

Speed of finding colliding pairs

How fast the pairs are enumerated depends on how often the algorithm succeeds and how fast it is.

These parameters in turn may depend on how large $\mathcal{M}$ is relative to $\mathcal{H}$.

It is always possible that h is one-to-one on some subset U of elements in $\mathcal{M}$, so it is not necessarily true that every message has a colliding partner.

However, an easy counting argument shows that U has size at most $|\mathcal{H}| - 1$.

Since we assume $|\mathcal{M}| \gg |\mathcal{H}|$, the probability that a randomly-chosen message from $\mathcal{M}$ lies in U is correspondingly small.

Strong implies one-way

In a similar vein, we argue that strong collision-free implies one-way.

Fact

If h is strong collision-free, then h is one-way.

Proof that strong ⇒ one-way

Proof (Sketch).

Suppose h is *not* one-way. Then there is an algorithm $A(y)$ for finding m such that $h(m) = y$, and $A(y)$ succeeds with non-negligible probability when y is chosen randomly with probability proportional to the size of its preimage. Assume that $A(y)$ returns $\perp$ to indicate failure.

A randomized algorithm to enumerate colliding pairs:

1. *Choose random m.*
2. *Compute $y = h(m)$.*
3. *Compute $m' = A(y)$.*
4. *If $m' \notin \{\perp, m\}$ then output (m, m').*
5. *Start over at step 1.*

Proof (cont.)

Proof (continued).

Each iteration of this algorithm succeeds with significant probability ε that is the product of the probability that $A(y)$ succeeds on y and the probability that $m' \neq m$.

The latter probability is at least $1/2$ except for those values m which lie in the set of U of messages on which h is one-to-one (defined in the previous proof).

Thus, assuming $|\mathcal{M}| \gg |\mathcal{H}|$, the algorithm outputs each colliding pair in expected number of iterations that is only slightly larger than $1/\varepsilon$. □

Weak implies one-way

These same ideas can be used to show that weak collision-free implies one-way, but now one has to be more careful with the precise definitions.

Fact

If h is weak collision-free, then h is one-way.

Proof that weak $\Rightarrow$ one-way

Proof (Sketch).

Suppose as before that h is *not* one-way, so there is an algorithm $A(y)$ for finding m such that $h(m) = y$, and $A(y)$ succeeds with significant probability when y is chosen randomly with probability proportional to the size of its preimage.

Assume that $A(y)$ returns $\perp$ to indicate failure. We want to show this implies that the weak collision-free property does not hold, that is, there is an algorithm that, for a significant number of $m \in \mathcal{M}$, succeeds with non-negligible probability in finding a colliding m'.

Proof that weak ⇒ one-way (cont.)

We claim the following algorithm works:

Given input m:

1. *Compute $y = h(m)$.*
2. *Compute $m' = A(y)$.*
3. *If $m' \notin \{\perp, m\}$ then output (m, m') and halt.*
4. *Otherwise, start over at step 1.*

This algorithm fails to halt for $m \in U$, but the number of such m is small (= insignificant) when $|\mathcal{M}| \gg |\mathcal{H}|$.

Proof that weak $\Rightarrow$ one-way (cont.)

It may also fail even when a colliding partner m' exists if it happens that the value returned by $A(y)$ is m. (Remember, $A(y)$ is only required to return some preimage of y; we can't say which.)

However, corresponding to each such bad case is another one in which the input to the algorithm is m' instead of m. In this latter case, the algorithm succeeds, since y is the same in both cases. With this idea, we can show that the algorithm succeeds in finding a colliding partner on at least half of the messages in $\mathcal{M} - U$.

Constructing New Hash Functions from Old

Extending a hash function

Suppose we are given a strong collision-free hash function

$$h : 256\text{-bits} \to 128\text{-bits}.$$

How can we use h to build a strong collision-free hash function

$$H : 512\text{-bits} \to 128\text{-bits}?$$

We consider several methods.

In the following, M is 512 bits long.
We write $M = m_1 m_2$, where m_1 and m_2 are 256 bits each.

Method 1

First idea. Define

$$H(M) = H(m_1 m_2) = h(m_1) \oplus h(m_2).$$

Unfortunately, this fails to be either strong or weak collision-free.

Let $M' = m_2 m_1$. (M, M') is always a colliding pair for H except in the special case that $m_1 = m_2$.

Recall that (M, M') is a colliding pair iff $H(M) = H(M')$ and $M \neq M'$.

Method 2

Second idea. Define

$$H(M) = H(m_1 m_2) = h(h(m_1)h(m_2)).$$

m_1 and m_2 are suitable arguments for $h()$ since $|m_1| = |m_2| = 256$.

Also, $h(m_1)h(m_2)$ is a suitable argument for $h()$ since $|h(m_1)| = |h(m_2)| = 128$.

Theorem

If h is strong collision-free, then so is H.

Correctness proof for Method 2

Assume H has a colliding pair $(M = m_1 m_2,\ M' = m_1' m_2')$.
Then $H(M) = H(M')$ but $M \neq M'$.

Case 1: $h(m_1) \neq h(m_1')$ or $h(m_2) \neq h(m_2')$.
Let $u = h(m_1)h(m_2)$ and $u' = h(m_1')h(m_2')$.
Then $h(u) = H(M) = H(M') = h(u')$, but $u \neq u'$.
Hence, (u, u') is a colliding pair for h.

Case 2: $h(m_1) = h(m_1')$ and $h(m_2) = h(m_2')$.
Since $M \neq M'$, then $m_1 \neq m_1'$ or $m_2 \neq m_2'$ (or both).
Whichever pair is unequal is a colliding pair for h.

In each case, we have found a colliding pair for h.

Hence, H not strong collision-free $\Rightarrow$ h not strong collision-free.
Equivalently, h strong collision-free $\Rightarrow$ H strong collision-free.

A general chaining method

Let $h : r\text{-bits} \to t\text{-bits}$ be a hash function, where $r \geq t + 2$.
(In the above example, $r = 256$ and $t = 128$.)
Define $H(m)$ for m of arbitrary length.

- Divide m after appropriate padding into blocks $m_1 m_2 \ldots m_k$, each of length $r - t - 1$.
- Compute a sequence of t-bit states:

$$\begin{aligned} s_1 &= h(0^t 0 m_1) \\ s_2 &= h(s_1 1 m_2) \\ &\vdots \\ s_k &= h(s_{k-1} 1 m_k). \end{aligned}$$

Then $H(m) = s_k$.

Chaining construction gives strong collision-free hash

Theorem

Let h be a strong collision-free hash function. Then the hash function H constructed from h by chaining is also strong collision-free.

Correctness proof

Assume H has a colliding pair (m, m').
We find a colliding pair for h.

- Let $m = m_1 m_2 \dots m_k$ give state sequence $s_1, \dots, s_k$.
- Let $m' = m'_1 m'_2 \dots m'_{k'}$ give state sequence $s'_1, \dots, s'_{k'}$.

Assume without loss of generality that $k \leq k'$.

Because m and m' collide under H, we have $s_k = s'_{k'}$.
Let r be the largest value for which $s_{k-r} = s'_{k'-r}$.

Let $i = k - r$, the index of the first such equal pair $s_i = s'_{k'-k+i}$.

We proceed by cases.

(continued. . .)

Correctness proof (case 1)

Case 1: $i = 1$ and $k = k'$.

Then $s_j = s'_j$ for all $j = 1, \ldots, k$.

Because $m \neq m'$, there must be some ℓ such that $m_\ell \neq m'_\ell$.

If $\ell = 1$, then $(0^t 0 m_1, 0^t 0 m'_1)$ is a colliding pair for h.

If $\ell > 1$, then $(s_{\ell-1} 1 m_\ell,\ s'_{\ell-1} 1 m'_\ell)$ is a colliding pair for h.

(continued...)

Correctness proof (case 2)

Case 2: $i = 1$ and $k < k'$.

Let $u = k' - k + 1$.

Then $s_1 = s'_u$.

Since $u > 1$ we have that

$$h(0^t 0 m_1) = s_1 = s'_u = h(s'_{u-1} 1 m'_u),$$

so $(0^t 0 m_1,\ s'_{u-1} 1 m'_u)$ is a colliding pair for h.

Note that this is true even if $0^t = s'_{u-1}$ and $m_1 = m'_u$, a possibility that we have not ruled out.

(continued...)

Correctness proof (case 3)

Case 3: $i > 1$.

Then $u = k' - k + i > 1$.

By choice of i, we have $s_i = s'_u$, but $s_{i-1} \neq s'_{u-1}$.

Hence,

$$h(s_{i-1}1m_i) = s_i = s'_u = h(s'_{u-1}1m'_u),$$

so $(s_{i-1}1m_i,\ s'_{u-1}1m'_u)$ is a colliding pair for h.

(continued...)

Correctness proof (conclusion)

In each case, we found a colliding pair for h.

The contradicts the assumption that h is strong collision-free.

Hence, H is also strong collision-free.

Common Hash Functions

Popular hash functions

Many cryptographic hash functions are currently in use.

For example, the openssl library includes implementations of MD2, MD4, MD5, MDC2, RIPEMD, SHA, SHA-1, SHA-256, SHA-384, and SHA-512.

The SHA-xxx methods (otherwise known as SHA-2) are recommended for new applications, but these other functions are also in widespread use.

SHA-2

SHA-2 is a family of hash algorithms designed by NSA known as SHA-224, SHA-256, SHA-384, SHA-512, SHA-512/224, SHA-512/256.

They produce message digests of lengths 224, 256, 384, or 512 bits.

They comprise the current Secure Hash Standard (SHS) and are described in FIPS 180–4. It states,

> *"Secure hash algorithms are typically used with other cryptographic algorithms, such as digital signature algorithms and keyed-hash message authentication codes, or in the generation of random numbers (bits)."*

SHA-1 broken

SHA-1 was first described in 1995. It produces a 160-bit message digest.

It was broken in 2005 by Xiaoyun Wang, Yiqun Lisa Yin, and Hongbo Yu: "Finding Collisions in the Full SHA-1". *CRYPTO 2005*: 17-36.

Wang and Yu did their work at Shandong University; Yin is listed on the paper as an independent security consultant in Greenwich, CT.

SHA-1 still in use

SHA-1 is still in widespread use despite its known vulnerabilities.

Google is taking steps in its Chrome browser to alert users to web sites still using SHA-1 based certificates.

See "Gradually sunsetting SHA-1".

A new secure hash algorithm

On Nov. 2, 2007, NIST announced a public competition for a replacement algorithm to be known as SHA-3.

The winner, an algorithm named Keccak, was announced on October 2, 2012 and standardized in August 2015. See http://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.202.pdf.

From the SHA-3 standard

Now that the standards document is out, it seems that SHA-3 is considered to be a supplement to the previous standard, not a replacement for it. The quote below is from the abstract of FIPS PUB 202.

> *"Hash functions are components for many important information security applications, including 1) the generation and verification of digital signatures, 2) key derivation, and 3) pseudorandom bit generation. The hash functions specified in this Standard supplement the SHA-1 hash function and the SHA-2 family of hash functions that are specified in FIPS 180-4, the Secure Hash Standard."*

MD5

MD5 is an older algorithm (1992) devised by Rivest.

Weaknesses were found as early as 1996. It was shown not to be collision resistant in 2004.[2]

Subsequent papers show that MD5 has more serious weaknesses that make it no longer suitable for most cryptographic uses.

We present an overview of MD5 here because it is relatively simple and it illustrates the principles used in many hash algorithms.

[2] How to Break MD5 and Other Hash Functions by Xiaoyun Wang and Hongbo Yu.

MD5 algorithm overview

MD5 generates a 128-bit message digest from an input message of any length. It is built from a basic block function

$$g : \text{128-bit} \times \text{512-bit} \to \text{128-bit}.$$

The MD5 hash function h is obtained as follows:

- The original message is padded to length a multiple of 512.
- The result m is split into a sequence of 512-bit blocks $m_1, m_2, \ldots, m_k$.
- h is computed by chaining g on the first argument.

We next look at these steps in greater detail.

MD5 padding

As with block encryption, it is important that the padding function be one-to-one, but for a different reason.

For encryption, the one-to-one property is what allows unique decryption.

For a hash function, it prevents there from being trivial colliding pairs.

For example, if the last partial block is simply padded with 0's, then all prefixes of the last message block will become the same after padding and will therefore collide with each other.

MD5 chaining

The function h can be regarded as a state machine, where the states are 128-bit strings and the inputs to the machine are 512-bit blocks.

The machine starts in state s_0, specified by an initialization vector IV.

Each input block m_i takes the machine from state s_{i-1} to new state $s_i = g(s_{i-1}, m_i)$.

The last state s_k is the output of h, that is,

$$h(m_1 m_2 \ldots m_{k-1} m_k) = g(g(\ldots g(g(IV, m_1), m_2) \ldots, m_{k-1}), m_k).$$

MD5 block function

The block function $g(s, b)$ is built from a scrambling function $g'(s, b)$ that regards s and b as sequences of 32-bit words and returns four 32-bit words as its result.

Suppose $s = s_1 s_2 s_3 s_4$ and $g'(s, b) = s'_1 s'_2 s'_3 s'_4$.

We define

$$g(s, b) = (s_1 + s'_1) \cdot (s_2 + s'_2) \cdot (s_3 + s'_3) \cdot (s_4 + s'_4),$$

where "+" means addition modulo 2^{32} and "$\cdot$" is concatenation of the representations of integers as 32-bit binary strings.

MD5 scrambling function

The computation of the scrambling function $g'(s, b)$ consists of 4 stages, each consisting of 16 substages.

We divide the 512-bit block b into 32-bit words $b_1 b_2 \ldots b_{16}$.

Each of the 16 substages of stage i uses one of the 32-bit words of b, but the order they are used is defined by a permutation π_i that depends on i.

In particular, substage j of stage i uses word b_ℓ, where $\ell = \pi_i(j)$ to update the state vector s.

The new state is $f_{i,j}(s, b_\ell)$, where $f_{i,j}$ is a bit-scrambling function that depends on i and j.

Further remarks on MD5

We omit further details of the bit-scrambling functions $f_{i,j}$,

However, note that the state s can be represented by four 32-bit words, so the arguments to $f_{i,j}$ occupy only 5 machine words. These easily fit into the high-speed registers of modern processors.

The definitive specification for MD5 is RFC1321 and errata. A general discussion of MD5 along with links to recent work and security issues can be found on Wikipedia.

Appendix: Birthday Attack Revisited

Bits of security for hash functions

MD5 hash function produces 128-bit values, whereas the SHA-xxx family produces values of 160-bits or more.

How many bits do we need for security?

Both 128 and 160 are more than large enough to thwart a brute force attack that simply searches randomly for colliding pairs.

However, the *Birthday Attack* reduces the size of the search space to roughly the square root of the original size.

MD5's effective security is at most 64 bits. ($\sqrt{2^{128}} = 2^{64}$.)

SHA-1's effective security is at most 80-bits. ($\sqrt{2^{160}} = 2^{80}$.)

Xiaoyun Wang, Yiqun Lisa Yin, and Hongbo Yu describe an attack that reduces this number to only 69-bits (Crypto 2005).

Birthday Paradox

We described a *birthday attack* in lecture 7, based on the *birthday paradox*.

The problem is to find the probability that two people in a set of randomly chosen people have the same birthday.

This probability is greater than 50% in any set of at least 23 randomly chosen people.[3].

23 is far less than the 253 people that are needed for the probability to exceed 50% that at least one of them was born on a specific day, say January 1.

[3] See Wikipedia, "Birthday paradox".

Birthday Paradox (cont.)

Here's why it works.

The probability of *not* having two people with the same birthday is is

$$q = \frac{365}{365} \cdot \frac{364}{365} \cdots \frac{343}{365} = 0.492703$$

Hence, the probability that (at least) two people have the same birthday is $1 - q = 0.507297$.

This probability grows quite rapidly with the number of people in the room. For example, with 46 people, the probability that two share a birthday is 0.948253.

Birthday attack on hash functions

The birthday paradox gives a much faster way to find colliding pairs of a hash function than simply choosing pairs at random.

> Method: Choose a random set of k messages and see if any two messages in the set collide.

Thus, with only k evaluations of the hash function, we can test $\binom{k}{2} = k(k-1)/2$ different pairs of messages for collisions.

Birthday attack analysis

Of course, these $\binom{k}{2}$ pairs are not uniformly distributed, so one needs a birthday-paradox style analysis of the probability that a colliding pair will be found.

The general result is that the probability of success is at least $1/2$ when $k \approx \sqrt{n}$, where n is the size of the hash value space.

Practical difficulties of birthday attack

Two problems make this attack difficult to use in practice.

1. One must find duplicates in the list of hash values. This can be done in time $O(k \log k)$ by sorting.
2. The list of hash values must be stored and processed.

For MD5, $k \approx 2^{64}$. To store k 128-bit hash values requires 2^{68} bytes $\approx$ 250 exabytes = 250,000 petabytes of storage.

To sort would require $\log_2(k) = 64$ passes over the table, which would process 16 million petabytes of data.

A back-of-the-envelope calculation

Google was reportedly processing 20 petabytes of data per day in 2008. At this rate, it would take Google more than 800,000 days or nearly 2200 years just to sort the data.

This attack is still infeasible for values of k needed to break hash functions. Nevertheless, it is one of the more subtle ways that cryptographic primitives can be compromised.