Hash Functions

Hashes provide assurance of data integrity, a different property from secrecy.

Hash function: construct a short fingerprint of a message (often called a *message digest*) — e.g. 160 bits in size.

- Given message x, compute $h(x)$, and store in a safe place.
- At later time, check if message has same fingerprint.
- If not, message was tampered with (network error, adversary messed with it)

Why do you need to keep $h(x)$ safe?

- Otherwise, he who modified x could modify the digest accordingly.

Solution: keyed hash function. Really, a family of hash functions, indexed by a key.

Scenario:

- Alice and Bob share a key K (back in the shared key model...)
- Alice wants to send x, computes $y = h_k(x)$.
- Alice sends (x, y). Bob receives it and checks $h_k(x) = y$.
- If so, great; if not, x or y was tampered with.

Formal definition: A *hash family* is a tuple $(\mathcal{X}, \mathcal{Y}, \mathcal{K}, \mathcal{H})$ where:

1. $\mathcal{X}$ is a set of possible messages (not necessarily finite)
2. $\mathcal{Y}$ is a finite set of possible digests
3. $\mathcal{K}$ is a finite set of possible keys (keyspace)

4. For each key $k \in \mathcal{K}$, there is a hash function $h_k : \mathcal{X} \to \mathcal{Y}$ in $\mathcal{H}$.

A pair (x, y) is called a *valid pair under key* k if $h_k(x) = y$.

An unkeyed hash function can be modeled by a hash family with a single fixed key k that is known to everyone.

Security for unkeyed hash functions

Suppose $h : \mathcal{X} \to \mathcal{Y}$ is an unkeyed hash function. The following three problems should be difficult to solve if the hash function is to be considered secure. (Why these problems? Reflect how hash functions are used.)

1. (Preimage problem) Given $h : \mathcal{X} \to \mathcal{Y}$ and $y \in \mathcal{Y}$, find $x \in \mathcal{X}$ such that $h(x) = y$.
2. (Second preimage problem) Given $h : \mathcal{X} \to \mathcal{Y}$ and $x \in \mathcal{X}$, find $x' \in \mathcal{X}$ such that $x' \neq x$ and $h(x') = h(x)$.
3. (Collision problem) Given $h : \mathcal{X} \to \mathcal{Y}$, find $x, x' \in \mathcal{X}$ such that $x \neq x'$ and $h(x) = h(x')$.

The random oracle model

We want to understand the above problems for a "perfect hash function", to get a sense of the best we can do. How do we model a perfect hash function? The random oracle model is used as a mathematical model of a perfect hash function.

Intuitively, it captures the intuition that we should not be able to extract any information from how a hash function computes its hash.

In the random oracle model, a hash function $h : \mathcal{X} \to \mathcal{Y}$ is chosen at random, and we are only permitted oracle access to h:

- We cannot see how h is implemented
- We can only ask the oracle, what's $h(x)$? for some xs, and the oracle answers.

Let $M = |\mathcal{Y}|$.

Theorem 4.1: Suppose $h : \mathcal{X} \to \mathcal{Y}$ is chosen randomly. Let $\mathcal{X}_0 \subset \mathcal{X}$. Suppose $h(x)$ are given for $x \in \mathcal{X}_0$. Then Pr[h(x)=y] = 1/M for all $x \in \mathcal{X} - \mathcal{X}_0$ and all $y \in \mathcal{Y}$.

In other words, even if we query the oracle for some valid pairs, given a message x not part of the queries, the probability that the hash of x yields a particular digest is the same for all digests: we do not gain any information about the function h even if we query for valid pairs.

How hard are the three problems above in the random oracle model?

For the preimage problem, consider the following (randomized) algorithm, for a fixed Q (the number of queries you allow).

1. Choose Q messages at random
2. For each chosen message x, compute $h(x)$
3. If one of the $h(x)$ is the digest you are looking for, return x, otherwise, faily.

This is essentially the best we can do, because the only moves we have available to use is to query the oracle for digest of messages.

The average case success probability (i.e., the probability that this algorithm reports a good x given a random digest y for which you are looking for a preimage) is $1 - (1 - \frac{1}{M})^Q$. If Q is much smaller than M, this is approximately $\frac{Q}{M}$.

For the second preimage problem, consider the following randomized algorithm:

1. Choose Q messages at random, none equal to initial x
2. For each chosen message x', compute $h(x')$
3. If one of the $h(x')$ is $h(x)$, return x'; otherwise, fail.

Again, the average case success probability is $1 - (1 - \frac{1}{M})^Q$.

For the collision problem consider the following randomized algorithm:

1. Choose Q messages at random
2. For each chosen message x, compute $y_x = h(x)$.
3. If any two y_x and $y_{x'}$ are equal, return (x, x'); otherwise, fail.

The average case success probability is $1 - (\frac{M-1}{M})(\frac{M-2}{M}) \ldots (\frac{M-Q+1}{M})$, which is about $1 - e^{\frac{-Q(Q-1)}{2M}}$.

If we care about finding a collision with probability $1/2$, we can solve the equation $1 - e^{\frac{-Q(Q-1)}{2M}} = 1/2$ and get that we need a Q, or number of queries, of about $\sqrt{M}$.

Bottom line: for an "ideal" hash function, we need to make sure M is large enough for the hash function to be safe.

Collision resistance implies second-preimage resistance.

Collision resistance implies preimage resitance (under some conditions).

Iterated hash functions

A method to extend a hash function on a finite domain to an infinite domain.

Restrict to bit strings for inputs and outputs for simplicity.

$|x|$: length of bit string x

$x \,||\, y$: concatenation of x and y.

Suppose you have a hash function over a finite domain (often called a compression function), **compress** : $\{0,1\}^{m+t} \to \{0,1\}^m$ ($t \geq 1$), we construct $h : (\cup_{i=m+t+1}^{\infty} \{0,1\}^i) \to \{0,1\}^l$.

1. Preprocessing: given x, $|x| \geq m + t + 1$, construct y s.t. $|y| \equiv 0 \pmod{t}$. (Public algorithm)

   Split y into $y_1 \,||\, \ldots \,||\, y_r$, where $|y_i| = t$ for all i.

   Note: Often y is obtained just by padding x $y = x \,||\, pad(x)$.

   Also, need the map $x \mapsto y$ to be injective, otherwise we get collisions.

2. Processing: let IV be a public initial value, $|IV| = m$.

$$
\begin{aligned}
z_0 &\leftarrow IV \\
z_1 &\leftarrow \textbf{compress}(z_0 \,||\, y_1) \\
z_2 &\leftarrow \textbf{compress}(z_1 \,||\, y_2) \\
&\ldots \\
z_r &\leftarrow \textbf{compress}(z_{r-1} \,||\, y_r)
\end{aligned}
$$

3. Output transformation: apply a public $g : \{0,1\}^m \to \{0,1\}^l$. (Optional step; can take identity function and $l = m$.)

Markle-Damgård construction

A particular way to construct an iterated hash function h with good properties, starting from a **compress** hash function with good properties is due to Markle and Damgård. For instance, if **compress** is collision resistant, then h will be collision resistant. (In other words, if we can find a collision in h efficiently, then we can find a collision in **compress** efficiently too.)

Here is the construction. Given **compress** : $\{0,1\}^{m+t} \to \{0,1\}^m$ a hash function. Construct $h : (\cup_{i=m+t+1}^{\infty} \{0,1\}^i) \to \{0,1\}^l$.

Suppose $t \geq 2$.

$x = x_1 \,||\, \ldots \,||\, x_k$.

$|x_1| = \cdots = |x_{k-1}| = t - 1$

$|x_k| = t - 1 - d$

Let $y_1 \leftarrow x_1, \ldots, y_{k-1} \leftarrow x_{k-1}$.

Let $y_k = x_k \,||\, 0^d$; note $|y_k| = t - 1$.

Let y_{k+1} be the binary representation of d (padded on left with 0s) to get size $t - 1$.

Let $y = y_1 \,||\, \ldots \,||\, y_{k+1}$.

$$
\begin{aligned}
z_1 &\leftarrow \textbf{compress}(0^{m+1} \,||\, y_1) \\
z_2 &\leftarrow \textbf{compress}(z_1 \,||\, 1 \,||\, y_2) \\
z_3 &\leftarrow \textbf{compress}(z_2 \,||\, 1 \,||\, y_3) \\
&\ldots \\
z_{k+1} &\leftarrow \textbf{compress}(z_k \,||\, 1 \,||\, y_{k+1})
\end{aligned}
$$

Result of the hash function: z_{k+1}.

Secure Hash Algorithm

SHA-1 algorithm of Rivest.

A finite domain hash function, that can hash messages of length up to $2^{64} - 1$ bits. Outputs a digest of 160 bits.

Series of hash functions: MD4 (1990), MD5 (1992), SHA-0 (1993), SHA-1 (1995).

Message Authentication Codes

A keyed hash function is often used as a message authentication code (MAC).

- A MAC is appended to a sequence of plaintext blocks.
- Used to convince Bob that the given sequence of plaintext originated with Alice and was not tampered with.

Common way to create a MAC: incorporate a secret key into an unkeyed hash function, by including it as part of the message to be hashed.

If one is not careful, this can be easy to break, i.e., adversary can create a MAC with the same key, but without knowing the key:

- Suppose you use an iterated hash function, and use the key as the initial value IV. Suppose there is no pre- or postprocessing.
- Suppose x has length a multiple of t.
- Key k has length m.
- Given x and $h_k(x)$, the MAC of x, the adversary can produce a MAC for another message with k.
- Let x' be a message of length t.
- Take the message $x \,||\, x'$. This is the message the adversary can produce a MAC for.
- The MAC is $h_k(x \,||\, x')$, and working through the definition of iterated is computed as $\mathbf{compress}(h_k(x) \,||\, x')$.
- Thus, since $h_k(x)$ and x' are known, the MAC can be computed directly by the adversary, without knowing k

Two common ways of creating MACs:

1. HMAC : construct a MAC from an unkeyed hash function.

   We describe an HMAC based on SHA-1. Key size: 512 bits.

ipad: 512 bits constant `0x363636..36`.

opad: 512 bits constant `0x5C5C5C..5C`.

$$HMAC_k(x) = SHA1((k \oplus opad) \,||\, SHA1((k \oplus ipad) \,||\, x))$$

(This is a form of *nested MAC*, a composition of two keyed hash functions.)

2. CBC-MAC: use a block cipher in CBC mode.

   Block cipher can be any endomorphic cryptosystem with $\mathcal{P} = \mathcal{C} = \{0,1\}^t$.

   let $IV = 0^t$.

   Take $x = x_1 \,||\, \ldots \,||\, x_n$, where each $|x_i| = t$.

   Compute CBC encryption of x with key k, and keep y_n as the MAC. (Disregard intermediate y_i's.)