Hash Functions

CS 6750     Lecture 5
October 8, 2009
Riccardo Pucella
Hash Functions

Hash functions provide assurance of data integrity
- A different property than secrecy

Idea: construct a short fingerprint of a message
- Often called a message digest (or a hash)
- Size the same for all messages, e.g., 160 bits
Typical Usage Scenario

- Hash function $h(x)$
  - produces digest for message $x$

- Given message $x$:
  - compute $h(x)$ and store in safe place
  - At a later time, check if message still has same digest
  - If not, message was tampered with
    - possibly network error
    - or an attacker messed with it

- Why do you need to keep $h(x)$ safe?
  - otherwise, whomever modified the message could modify the digest accordingly
Keyed Hash Functions

Really, a family of hash functions indexed by a key

Scenario:
- Alice and Bob share a key $K$
- Alice wants to send $x$, computes $y = h_K(x)$
- Alice sends $(x, y)$
- Bob receives it and checks that $h_K(x) = y$
- If not, $x$ or $y$ was tampered with
  - (Or there was a network error)
Formal Definition

A hash family is a tuple \((X,Y,K,H)\) where

- \(X\) is a set of possible messages (could be infinite)
- \(Y\) is a finite set of possible digests
- \(K\) is a finite set of possible keys (the keyspace)
- For each key \(k \in K\), there is a hash function
  \[ h_k : X \rightarrow Y \text{ in } H \]

A pair \((x,y)\) is called a valid pair under key \(k\) if
\[ h_k(x) = y \]

A unkeyed hash function can be modeled as a hash family with a single globally known fixed key \(k\)
Security for Unkeyed Hash Functions

Suppose $h : X \rightarrow Y$ is an unkeyed hash function.

The following three problems should be difficult to solve if the hash function is to be considered secure:

1. **Preimage Problem:**
   - Given $y \in Y$, find $x \in X$ such that $h(x) = y$.

2. **Second Preimage Problem:**
   - Given $x \in X$, find $x' \in X$ such that $x \neq x'$ and $h(x) = h(x')$.

3. **Collision Problem:**
   - Find $x, x' \in X$ such that $x \neq x'$ and $h(x) = h(x')$.
The Random Oracle Model

What is the best we can do for the above problems?

Suppose we had a “perfect hash function”

The random oracle model is a mathematical model of a perfect hash function

Intuition behind a perfect hash function:

we should not be able to extract any information from how a hash function computes the hash

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

We cannot see how $h$ is implement

We can only ask: what’s $h(x)$?
Main Theorem

Let $M = |Y|$

**Theorem:** Suppose $h : X \rightarrow Y$ is chosen randomly. Let $X_0 \subseteq X$. Suppose $h(x)$ are known for all $x \in X_0$. Then $\Pr[h(x)=y] = 1/M$ for all $x \in X-X_0$ and all $y \in Y$.

I.e., 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 $x$ is a particular digest $y$ is the same for all digests.

We do not gain any information about the function $h$ even if we have a set of valid pairs.
Preimage Problem

This algorithm is essentially the best we can do

Let Q be the number of queries we allow
Let y be a digest for which we want a preimage
Choose Q messages at random
For each chosen message x, compute h(x)
If one of the h(x) is y, return x; otherwise fail.

The probability that this algo reports a good x given a random digest y of interest is $1-(1-1/M)^Q$
If Q is much smaller than M, this is $\sim Q/M$
Second Preimage Problem

Again, this algorithm is essentially the best we can do.

Let Q be the number of queries we allow.
Let x be a message for which we want a 2nd preimage.
Choose Q messages at random (none of them x).
For each chosen message x’, compute h(x’).
If one of the h(x’) is h(x), return x’; otherwise fail.

Again, the probability that this returns some x’ is
\[ 1 - \left(1 - \frac{1}{M}\right)^Q \]
If Q is much smaller than M, this is \( \sim \frac{Q}{M} \).
Collision Problem

Again, this algorithm is essentially the best we can do

Let $Q$ be the number of queries we allow

Choose $Q$ messages at random

For each chosen message $x$, compute $y_x = h(x)$

If any two $y_x$ and $y_{x'}$ are equal, return $(x,x')$; otherwise, fail

The probability that we get a pair $(x,x')$ is

$$1 - \left(\frac{M-1}{M}\right) \left(\frac{M-2}{M}\right) \ldots \left(\frac{M-Q+1}{M}\right)$$

which is about $1 - e^{-Q(Q-1)/2M}$
Collision Problem

Again, this algorithm is essentially the best we can do.

Let $Q$ be the number of queries we allow.

Choose $Q$ messages at random.

For each chosen message $x$, compute $y = h(x)$.

If any two $y$s are equal, return $(x, x')$; otherwise, fail.

The probability that we get a pair $(x, x')$ is

$$1 - \left(\frac{M-1}{M}\right) \left(\frac{M-2}{M}\right) \ldots \left(\frac{M-Q+1}{M}\right)$$

which is about $1 - e^{-Q(Q-1)/2M}$.

If we want a collision with probability $1/2$, need $Q$ to be about $\sqrt{M}$. 

Conclusions

For a perfect hash function, to be secure, we need a large $M$

- In an ideal situation

In practice, hash functions are not perfect, but we still need a large $M$

- Note that
- Collision resistance implies second-preimage resistance
- Collision resistance implies preimage resistance (under some conditions)
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)
- SHA-2 (2001) -- similar to SHA-1 but with different digest lengths
Iterated Hash Functions

- A method to extend a hash function on a finite domain to an infinite domain
- For simplicity, consider bit strings as inputs/outputs

Notation:
- $|x|$ = length of bit string $x$
- $x \| y$ = concatenation of bit strings $x$ and $y$
Iterated Hash Functions

Given compress : \( \{0,1\}^{m+t} \rightarrow \{0,1\}^m \)

- a hash function over a finite domain (compression)
- we construct \( h : (\bigcup_{i>m+t} \{0,1\}^i) \rightarrow \{0,1\}^l \), for some \( l \)

Preprocessing:
- given \( x \) with \( |x| > m+t \), construct \( y \) such that 
  \( |y| \equiv 0 \pmod{t} \)
- e.g., using a padding function, \( y = x \| \text{pad}(x) \)
- Make sure map from \( x \) to \( y \) is injective (otherwise, collisions)
- Split \( y \) into \( y_1 \| ... \| y_r \) where \( |y_i| = t \) for all \( i \)
Iterated Hash Functions

Processing:
- Let IV be some public initial value, $|IV| = m$

\[
\begin{align*}
  z_0 &\leftarrow IV \\
  z_1 &\leftarrow \text{compress}(z_0 \parallel y_1) \\
  z_2 &\leftarrow \text{compress}(z_1 \parallel y_2) \\
  \vdots \\
  z_r &\leftarrow \text{compress}(z_{r-1} \parallel y_r)
\end{align*}
\]

Output transformation:
- Apply a public $g : \{0,1\}^m \rightarrow \{0,1\}^l$
- Can take $g$ to be the identity function, and $l=m$
Markle-Damgard Construction

A way to construct an iterated hash function $h$ with good properties from a compress hash function with good properties.

If compress is collision resistant, then $h$ is collision resistant.

Given compress : $\{0,1\}^{m+t} \rightarrow \{0,1\}^m$

a hash function over a finite domain (compression)

we construct $h : \bigcup_{i>m+t} \{0,1\}^i \rightarrow \{0,1\}^l$, for some $l$.
Suppose $t > 1$

Let $x \in (U_{i>m+t} \{0,1\}^i)$, split $x$ into $x_1 \parallel \ldots \parallel x_k$

- $|x_1| = \ldots = |x_{k-1}| = t-1$
- $|x_k| = t-1-d$

Set $y_1 = x_1$, ..., $y_{k-1} = x_{k-1}$

Set $y_k = x_k \parallel 0^d$ \hspace{1cm} (Note: $|y_k| = t-1$)

Set $y_{k+1} = \text{binary representation of } d \text{ padded on the left with 0s to size } t-1$
Markle-Damgård Construction

Processing:

$z_1 \leftarrow \text{compress}(0^{m+1} \parallel y_1)$

$z_2 \leftarrow \text{compress}(z_1 \parallel 1 \parallel y_2)$

$z_3 \leftarrow \text{compress}(z_2 \parallel 1 \parallel y_3)$

$\ldots$

$z_{k+1} \leftarrow \text{compress}(z_k \parallel 1 \parallel y_{k+1})$

Result of the hash function $h(x)$ is $z_{k+1}$
Keyed Hash Functions

- A common way to create keyed hash functions incorporate a secret key into an unkeyed hash function by including the key as part of the message to be hashed.

- If one is not careful, this can be easy to break.
  - The adversary may be able to create a keyed hash with the same key, but without knowing the key.
Example

- Suppose you use an iterated hash function
- Suppose you use the key as initial value IV
- Suppose no pre- or post-processing steps
- Let $|x| \equiv 0 \pmod{t}$
- $|k| = m$

  - Given $x$ and $h_k(x)$, the adversary can produce $h_k(x_{\text{alt}})$ for some other $x_{\text{alt}}$
  - Let $x'$ be a message with $|x'| = t$
  - Take the message $x \parallel x'$ (this will be $x_{\text{alt}}$)
  - $h_k(x_{\text{alt}}) = h_k(x \parallel x') = \ldots = \text{compress}(h_k(x) \parallel x')$
  - Since $h_k(x)$ and $x'$ are known, can compute $h_k(x_{\text{alt}})$
    - Without knowing $k$
A keyed hash function is often used as a message authentication code (MAC)

A MAC can be happened to a sequence of plaintext blocks

Used to convince receiver that the given plaintext originated with Alice and was not tampered with

This is the original scenario that I gave at the beginning of lecture
Common Ways to Create MAC (1)

- HMAC (keyed-Hash Message Authentication Code)
  - Construct MAC from an unkeyed hash function
  - Example based on SHA-1, with key size 512 bits:

  - ipad = 512 bits constant 0x363636..36
  - opad = 512 bits constant 0x5c5c5c5c..5c

  \[
  T = \text{SHA1}((k \oplus \text{ipad}) \| x)
  \]

  \[
  \text{HMAC}_k(x) = \text{SHA1}((k \oplus \text{opad}) \| T)
  \]

  (A form of nested MAC, with two keyed hashes)
CBC-MAC

- Use a block cipher in CBC mode
- Any endomorphism block cipher with \( P=C=\{0,1\}^t \)
- Let \( x = x_1 \| ... \| x_n \)
  where \( |x_i| = t \) for each \( i \)

- Compute CBC encryption with key \( k \)
- Keep \( y_n \) as MAC