CPSC 467b: Cryptography and Computer Security

Michael J. Fischer

Lecture 16
March 26, 2013

Hash Function Constructions

Extending a hash function

Suppose we are given a strong collision-free hash function

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

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

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

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. Let $M = m_1 m_2$ and 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-bits $\rightarrow$ t-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 $m = m_1 m_2 \ldots m_k$ give state sequence $s_1, \ldots, s_k$.
- Let $m' = m'_1 m'_2 \ldots m'_{k'}$ give state sequence $s'_1, \ldots, s'_{k'}$.

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.

Hash values can look non-random

Intuitively, we like to think of $h(y)$ 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.

But this intuition is faulty, as I now show.

Example of a non-random-looking hash function

Suppose h is a strong collision-free hash function.

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

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

Thus, H is strong collision-free, despite the fact that the string $H(x)$ always begins with 0.

Later on, we will talk about how to make functions that truly do appear to be random (even though they are not).

Birthday Attack on Hash Functions

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

The *birthday paradox* 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.[1].

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.

[1] 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 can be applied to hash functions to yield a much faster way to find colliding pairs 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.

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.

Hash from Cryptosystem

Building hash functions from cryptosystems

We've already seen several cryptographic hash functions as well as methods for making new hash functions from old.

We describe a way to make a hash function from a symmetric cryptosystem with encryption function $E_k(b)$.

Assume the key and block lengths are the same. (This rules out DES but not AES with 128-bit keys.)

The construction

Let m be a message of arbitrary length. Here's how to compute $H(m)$.

- Pad m appropriately and divide it into block lengths appropriate for the cryptosystem.
- Compute the following state sequence:

$$\begin{aligned} s_0 &= \mathrm{I}V \\ s_1 &= f(s_0, m_1) \\ &\vdots \\ s_t &= f(s_{t-1}, m_t). \end{aligned}$$

- Define $H(m) = s_t$.

IV is an initial vector and f is a function built from E.

Possible state transition functions $f(s, m)$

Some possibilities for f are

$$\begin{aligned} f_1(s, m) &= E_s(m) \oplus m \\ f_2(s, m) &= E_s(m) \oplus m \oplus s \\ f_3(s, m) &= E_s(m \oplus s) \oplus m \\ f_4(s, m) &= E_s(m \oplus s) \oplus m \oplus s \end{aligned}$$

You should think about why these particular functions do or do not lead to a strong collision-free hash function.

A bad state transition function

For example, if $t = 1$ and $f = f_1$, then

$$H(m) = f_1(\mathrm{I}V, m) = E_{\mathrm{I}V}(m) \oplus m.$$

$E_{\mathrm{I}V}$ itself is one-to-one (since it's an encryption function), but what can we say about $H_1(m)$?

Indeed, if bad luck would have it that E_{IV} is the identity function, then $H(m) = 0$ for all m, and all pairs of message blocks collide!

Authentication Using Passwords

The authentication problem

The *authentication problem* is to identify whom one is communicating with.

For example, if Alice and Bob are communicating over a network, then Bob would like to know that he is talking to Alice and not to someone else on the network.

Knowing the IP address or URL is not adequate since Mallory might be in control of intermediate routers and name servers.

As with signature schemes, we need some way to differentiate the real Alice from other users of the network.

Possible authentication factors

Alice can be authenticated in one of three ways:

1. By something she knows;
2. By something she possesses;
3. By something she is.

Examples:

1. A secret password;
2. A smart card;
3. Biometric data such as a fingerprint.

Passwords

Assume that Alice possess some secret that is not known to anyone else. She authenticates herself by proving that she knows the secret.

Password mechanisms are widely used for authentication.

In the usual form, Alice authenticates herself by sending her password to Bob.

Bob checks that it matches Alice's password and grants access.

This is the scheme that is used for local logins to a computer and is also used for remote authentication on many web sites.

Weaknesses of password schemes

Password schemes have two major security weaknesses.

1. Passwords may be exposed to Eve when being used.
2. After Alice authenticates herself to Bob, Bob can use Alice's password to impersonate Alice.

Password exposure

Passwords sent over the network in the clear are exposed to various kinds of eavesdropping, ranging from ethernet packet sniffers on the LAN to corrupt ISP's and routers along the way.

The threat of password capture in this way is so great that one should *never* send a password over the internet in the clear.

Some precautions

Users of the old insecure Unix tools should switch to secure replacements such as ssh, slogin, and scp, or kerberized versions of telnet and ftp.

Web sites requiring user logins generally use the TSL/SSL (Transport Layer Security/Secure Socket Layer) protocol to encrypt the connection, making it safe to transmit passwords to the site, but some do not.

Depending on how your browser is configured, it will warn you whenever you attempt to send unencrypted data back to the server.

Password propagation

After Alice's password reaches the server, it is no longer the case that only she knows her password.

Now the server knows it, too!

This is no problem if Alice only uses her password to log into that *that particular* server.

However, if she uses the same password for other web sites, the first server can impersonate Alice to any other web site where Alice uses the same password.

Multiple web sites

Users these days typically have accounts with dozens or hundreds of different web sites.

The temptation is strong to use the same username-password pairs on all sites so that they can be remembered.

But that means that anyone with access to the password database on one site can log into Alice's account on any of the other sites.

Typically different sites have very differing sensitivity of the data they protect.

An on-line shopping site may only be protecting a customer's shopping cart, whereas a banking site allows access to a customer's bank account.

Password policy advice

My advice is to use a different password for each account.

Of course, nobody can keep dozens of different passwords straight, so the downside of my suggestion is that the passwords must be written down and kept safe, or stored in a properly-protected password vault.

If the primary copy gets lost or compromised, then one should have a backup copy so that one can go to all of the sites ASAP and change the passwords (and learn if the site has been compromised).

The real problem with simple password schemes is that Alice is required to send her secrets to other parties in order to use them. We will later explore authentication schemes that do not require this.

Secure password storage

Another issue with traditional password authentication schemes is the need to store the passwords on the server for later verification.

- The file in which passwords are store is highly sensitive.
- Operating system protections can (and should) be used to protect it, but they are not really sufficient.
- Legitimate sysadmins might use passwords found there to log into users' accounts at other sites.
- Hackers who manage to break into the computer and obtain root privileges can do the same thing.
- Finally, backup copies may not be subject to the same system protections, so someone with access to a backup device could read everybody's password from it.

Storing encrypted passwords

Rather than store passwords in the clear, it is usual to store "encrypted" passwords.

That is, the hash value of the password under some cryptographic hash function is stored instead of the password itself.

Using encrypted passwords

The authentication function

- takes the cleartext password from the user,
- computes its hash value,
- and checks that the computed and stored hashed values match.

Since the password does not contain the actual password, and it is computationally difficult to invert a cryptographic hash function, knowledge of the hash value does not allow an attacker to easily find the password.

Dictionary attacks on encrypted passwords

Access to an encrypted password file opens up the possibility of a *dictionary attack*.

Many users choose weak passwords—words that appear in an English dictionary or in other available sources of text.

If one has access to the password hashes of legitimate users on the computer (such as is contained in `/etc/passwd` on Unix), an attacker can hash every word in the dictionary and then look for matches with the password file entries.

Harm from dictionary attacks

A dictionary attack is quite likely to succeed in compromising at least a few accounts on a typical system.

Even one compromised account is enough to allow the hacker to log into the system as a legitimate user, from which other kinds of attacks are possible that cannot be carried out from the outside.

Salt

Adding salt is a way to make dictionary attacks more expensive.

- *Salt* is a random number that is stored along with the hashed password in the password file.
- The hash function takes two arguments, the password and salt, and produces a hash value.
- Because the salt is stored (in the clear) in the password file, the user's password can be easily verified.
- The same password hashes differently depending on the salt.
- A successful dictionary attack now has to encrypt the entire dictionary with every possible salt value (or at least with every salt value that appears in the password file being attacked).
- This increases the cost of the attack by orders of magnitude.

Chinese Remainder Theorem

Systems of congruence equations

Theorem (Chinese remainder theorem)

Let $n_1, n_2, \ldots, n_k$ be positive pairwise relatively-prime integers[2], let $n = \prod_{i=1}^{k} n_i$, and let $a_i \in \mathbf{Z}_{n_i}$ for $i = 1, \ldots, k$. Consider the system of congruence equations with unknown x:

$$\begin{aligned} x &\equiv a_1 \pmod{n_1} \\ x &\equiv a_2 \pmod{n_2} \\ &\vdots \\ x &\equiv a_k \pmod{n_k} \end{aligned} \tag{1}$$

(1) has a unique solution $x \in \mathbf{Z}_n$.

---

[2] This means that $\gcd(n_i, n_j) = 1$ for all $1 \le i < j \le k$.

How to solve congruence equations

To solve for x, let

$$N_i = n/n_i = \underbrace{n_1 n_2 \ldots n_{i-1}} \cdot \underbrace{n_{i+1} \ldots n_k},$$

and compute $M_i = N_i^{-1} \bmod n_i$, for $1 \le i \le k$.

$N_i^{-1} \pmod{n_i}$ exists since $\gcd(N_i, n_i) = 1$. (Why?)
We can compute N_i^{-1} by solving the associated Diophantine equation as described in Lecture 10.

The solution to (1) is

$$x = (\sum_{i=1}^{k} a_i M_i N_i) \bmod n \tag{2}$$

Correctness

Lemma

$$M_j N_j \equiv \begin{cases} 1 \ (mod\ n_i) & if\ j = i; \\ 0 \ (mod\ n_i) & if\ j \neq i. \end{cases}$$

Proof.

$M_i N_i \equiv 1 \pmod{n_i}$ since $M_i = N_i^{-1} \bmod n_i$.
If $j \neq i$, then $M_j N_j \equiv 0 \pmod{n_i}$ since $n_i | N_j$. □

It follows from the lemma and the fact that $n_i \mid n$ that

$$x \equiv \sum_{i=1}^{k} a_i M_i N_i \equiv a_i \pmod{n_i} \tag{3}$$

for all $1 \leq i \leq k$, establishing that (2) is a solution of (1).

Uniqueness

To see that the solution is unique in $\mathbf{Z}_n$, let $\chi : \mathbf{Z}_n \to \mathbf{Z}_{n_1} \times \ldots \times \mathbf{Z}_{n_k}$ be the mapping

$$x \mapsto (x \bmod n_1, \ldots, x \bmod n_k).$$

χ is a surjection[3] since $\chi(x) = (a_1, \ldots, a_k)$ iff x satisfies (1).

Since also $|\mathbf{Z}_n| = |\mathbf{Z}_{n_1} \times \ldots \times \mathbf{Z}_{n_k}|$, χ is a bijection, and there is only one solution to (1) in $\mathbf{Z}_n$.

[3] A *surjection* is an onto function.

An alternative proof of uniqueness

A less slick but more direct way of seeing uniqueness is to suppose that $x = u$ and $x = v$ are both solutions to (1).

Then $u \equiv v \pmod{n_i}$, so $n_i | (u - v)$ for all i.

By the pairwise relatively prime condition on the n_i, it follows that $n | (u - v)$, so $u \equiv v \pmod{n}$. Hence, the solution is unique in $\mathbf{Z}_n$.

Quadratic Residues, Squares, and Square Roots

Square roots in $\mathbf{Z}_n^*$

Recall from lecture 13 that to find points on an elliptic curve requires solving the equation

$$y^2 = x^3 + ax + b$$

for $y \pmod{p}$, and that requires computing square roots in $\mathbf{Z}_p^*$.

Squares and square roots have several other cryptographic applications as well.

Today, we take a brief tour of the theory of *quadratic resides*.

Quadratic residues modulo n

An integer b is a *square root* of a modulo n if

$$b^2 \equiv a \pmod{n}.$$

An integer a is a *quadratic residue (or perfect square)* modulo n if it has a square root modulo n.

Quadratic residues in $\mathbf{Z}_n^*$

If $a, b \in \mathbf{Z}_n$ and $b^2 \equiv a \pmod{n}$, then

$$b \in \mathbf{Z}_n^* \text{ iff } a \in \mathbf{Z}_n^*.$$

Why? Because

$$\gcd(b, n) = 1 \text{ iff } \gcd(a, n) = 1$$

This follows from the fact that $b^2 = a + un$ for some u, so if p is a prime divisor of n, then

$$p \mid b \text{ iff } p \mid a.$$

Assume that all quadratic residues and square roots are in $\mathbf{Z}_n^*$ unless stated otherwise.

QR_n and QNR_n

We partition $\mathbf{Z}_n^*$ into two parts.

$$\mathrm{QR}_n = \{a \in \mathbf{Z}_n^* \mid a \text{ is a quadratic residue modulo } n\}.$$
$$\mathrm{QNR}_n = \mathbf{Z}_n^* - \mathrm{QR}_n.$$

QR_n is the *set of quadratic residues* modulo n.

QNR_n is the *set of quadratic non-residues* modulo n.

For $a \in \mathrm{QR}_n$, we sometimes write

$$\sqrt{a} = \{b \in \mathbf{Z}_n^* \mid b^2 \equiv a \pmod{n}\},$$

the *set of square roots* of a modulo n.

Quadratic residues in $\mathbf{Z}_{15}^*$

The following table shows all elements of $\mathbf{Z}_{15}^* = \{1, 2, 4, 7, 8, 11, 13, 14\}$ and their squares.

b		$b^2 \bmod 15$
1		1
2		4
4		1
7		4
8	$= -7$	4
11	$= -4$	1
13	$= -2$	4
14	$= -1$	1

Thus, $\mathrm{QR}_{15} = \{1, 4\}$ and $\mathrm{QNR}_{15} = \{2, 7, 8, 11, 13, 14\}$.

Quadratic residues modulo an odd prime p

Fact

For an odd prime p,

- *Every $a \in QR_p$ has exactly two square roots in $\mathbf{Z}_p^*$;*
- *Exactly 1/2 of the elements of $\mathbf{Z}_p^*$ are quadratic residues.*

In other words, if $a \in \mathrm{QR}_p$,

$$|\sqrt{a}| = 2.$$

$$|\mathrm{QR}_n| = |\mathbf{Z}_p^*|/2 = \frac{p-1}{2}.$$

Quadratic residues in $\mathbf{Z}_{11}^*$

The following table shows all elements $b \in \mathbf{Z}_{11}^*$ and their squares.

b	$b^2 \bmod 11$
1	1
2	4
3	9
4	5
5	3

b	$-b$	$b^2 \bmod 11$
6	-5	3
7	-4	5
8	-3	9
9	-2	4
10	-1	1

Thus, $\mathrm{QR}_{11} = \{1, 3, 4, 5, 9\}$ and $\mathrm{QNR}_{11} = \{2, 6, 7, 8, 10\}$.

Proof that $|\sqrt{a}| = 2$ modulo an odd prime p

Let $a \in \mathrm{QR}_p$.

- It must have a square root $b \in \mathbf{Z}_p^*$.
- $(-b)^2 \equiv b^2 \equiv a \pmod{p}$, so $-b \in \sqrt{a}$.
- Moreover, $b \not\equiv -b \pmod{p}$ since $p \nmid 2b$, so $|\sqrt{a}| \geq 2$.
- Now suppose $c \in \sqrt{a}$. Then $c^2 \equiv a \equiv b^2 \pmod{p}$.
- Hence, $p \mid c^2 - b^2 = (c-b)(c+b)$.
- Since p is prime, then either $p \mid (c-b)$ or $p \mid (c+b)$ (or both).
- If $p \mid (c-b)$, then $c \equiv b \pmod{p}$.
- If $p \mid (c+b)$, then $c \equiv -b \pmod{p}$.
- Hence, $c = \pm b$, so $\sqrt{a} = \{b, -b\}$, and $|\sqrt{a}| = 2$.

Proof that half the elements of $\mathbf{Z}_p^*$ are in QR_p

- Each $b \in \mathbf{Z}_p^*$ is the square root of exactly one element of QR_p.
- The mapping $b \mapsto b^2 \bmod p$ is a 2-to-1 mapping from $\mathbf{Z}_p^*$ to QR_p.
- Therefore, $|\mathrm{QR}_p| = \frac{1}{2}|\mathbf{Z}_p^*|$ as desired.