CSCI3381-Cryptography

Lecture 9: Cryptographic Hash Functions

November 11, 2014

The relevant sections of the textbook are in Chapter 8. The chapter on DES also includes a description of password security using salted passwords, but with DES encryption of a fixed plaintext replacing the hash function.

1 Basic Properties

A hash function takes a bit string x of arbitrary length and returns a bit string $H(x)$ of fixed length ℓ. That is

$$H : \{0,1\}^* \to \{0,1\}^\ell.$$

Basically, a hash function should be easy to compute, but behave like a randomly-chosen function from bit strings into $\{0,1\}^\ell$. Imagine a function that was defined by taking each bit string x in turn and choosing $H(x)$ uniformly at random from the set of strings of length ℓ. More precisely, we require that H have the following security properties:

- *Collision resistance.* It should be computationally infeasible to find two different bit strings $x \neq x'$ such that $H(x) = H(x')$. Of course, there are infinitely many such collisions, but you should not be able to find one.

- *Second pre-image resistance.* It should be computationally infeasible, given bit strings x, y with $H(x) = y$, to find another string $x' \neq x$ with $H(x') = y$. This property follows from collision resistance, since if we can find such a second preimage, we get a collision.

- *Pre-image resistance.* Given $y \in \{0,1\}^\ell$, there is no efficient algorithm for finding a bit string x such that $H(x) = y$. This also follows from collision resistance: Suppose we had such an algorithm. Pick a random $z \in \{0,1\}^{N\cdot\ell}$, where N is very large, and compute $y = H(z)$. Use your algorithm to find x with $y = H(x)$. There is very little chance that the x produced by the algorithm is identical to the randomly chosen z you started with, so this method will produce a collision.

2 Birthday Attack and Size of Hashes

How big does ℓ have to be? To achieve preimage resistance, H has to be able to resist a brute-force attack that tries out many different x in turn and computes $H(x)$, comparing it to the target value y. This will require computation of 2^ℓ hashes on average, so we would like ℓ to be at least 80.

This is not big enough to resist collisions. In fact, we will show that by computing only slightly more than $2^{\ell/2}$ hashes, we are very likely to find a collision. This is called a *birthday attack,* because it works on the same probabilistic principle as the well-known birthday surprise that with no more than 23 people in a room the probability is more than $1/2$ that two of them share a birthday.

Imagine that we have N target values (the 2^ℓ possible hash values, or the 365 possible birthdays) and we fire k shots (the k hash values we compute, or the k people in the room). The probability that the shots hit all different targets is

$$\frac{N-1}{N}\cdot\frac{N-2}{N}\cdots\frac{N-k+1}{N} = \big(1-\frac{1}{N}\big)\cdot\big(1-\frac{2}{N}\big)\cdots\big(1-\frac{k-1}{N}\big).$$

To estimate this probability, we will use the fact that for x close to 0, e^x is an extremely good approximation to $1+x$. ($y = 1+x$ is the equation of the tangent line to e^x at $x=0$.) Thus we can estimate the probability of no two shots hitting the same target as:

$$\begin{aligned} e^{-1/N}\cdot e^{-2/N}\cdots e^{-(k-1)/N} &= \exp((1+2+\cdots(k-1))/N) \\ &= e^{-\frac{k(k-1)}{2N}}. \end{aligned}$$

As k grows larger, this estimated probability of no two shots hitting the same target grows smaller. It becomes smaller than $1/2$ when

$$e^{-\frac{k(k-1)}{2N}} = 1/2.$$

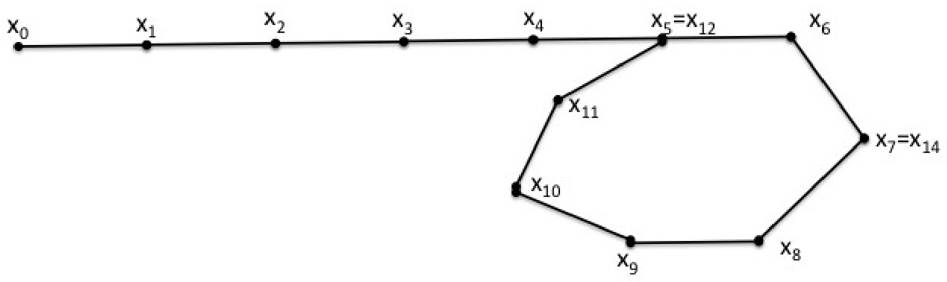

Figure 1: *Floyd's Algorithm Locates the Junction Between the Stem and the Period using Only a Tiny Amount of Memory*

That is,

$$\frac{k(k-1)}{2N} = \ln 2 \approx 0.69$$
$$k^2 - k \approx 1.38N$$
$$k \approx \sqrt{1.38N} = 1.17\sqrt{N}.$$

For $N = 365$, this gives $k = 22.4$. Resisting a birthday attack on a hash function will require $1.17 \cdot 2^{\ell/2}$ to be at least 2^{80}, which requires ℓ about 160.

Launching a birthday attack as just described against an 80-bit hash requires computing 2^{40} hashes, but also storage for the 2^{40} values and checking for duplicates, so this requires an enormous amount of memory and bookkeeping. However, there is a clever way to implement the attack that requires very just a small fixed amount of storage, at the cost of computing several times as many hashes. The idea is based on something called *Floyd's Cycle-Finding Algorithm.* Imagine a function $f : X \to X$, where X is a finite set. We choose a starting value x_0 and repeatedly compute

$$x_{i+1} = f(x_i).$$

Since X is finite, these values will eventually wrap around and form a cycle. (You can't do this without a picture! See Figure 1.) The cycle has a stem of length 's' and a 'period' of length p. As a result, if $r > s$, then

$$x_r = x_{s+(r-s) \bmod p}.$$

If we choose r to be a multiple of p that is larger than s, then

$$(r - s) \bmod p = (2r - s) \bmod p,$$

and thus $x_r = x_{2r}$. For instance, in the example illustrated with $s = 5$ and $p = 7$, we have $x_7 = x_{14}$.

Floyd's Algorithm begins by computing the pairs (x_i, x_{2i}) for $i = 1, 2, \ldots$. Note that $x_{2(i+1)} = f(f(x_{2i}))$, so we need three applications of f for each such step. Eventually we will find a pair

$(x_r = x_{2r})$, where $r = s + (kp - s)$ for some k. We now reset the first component of the pair to x_0 and repeatedly apply f to each component, giving

$$(x_0, x_r) \mapsto (x_1, x_{r+1}) \mapsto (x_{s-1}, x_{r+s-1}) \mapsto (x_s, x_{r+s}).$$

Observe $x_{r+s} = x_{s+kp} = x_s$, so this algorithm locates the junction point where the stem joins the period.

If $f = H$ is a hash function and x_0 is a randomly chosen bit string, this algorithm produces essentially random hashes, so the birthday statistics apply, and the cycle should close on itself within about $\sqrt{X}$ steps. Applying Floyd's algorithm finds a collision; the number of hashes we had to compute is about 3 times larger than the straighforward method, but we do not need to manage a very large table.

3 Design of Hash Functions: Hash Functions in Wide Use (and Disuse)

A standard way of creating hash functions is to use an iterative process, based on a *compression function*. A compression function $c : \{0,1\}^{2\ell} \to \{0,1\}^{\ell}$ takes a bit string of length 2ℓ and outputs a bit string of length ℓ. The compression function was supposed to be designed so that it is itself collision resistant. To hash a long message, you would partition it into ℓ-bit blocks

$$m_1, \ldots, m_r$$

and compute

$$h_1 = c(IV, m_1)$$

$$h_2 = c(h_1, m_2),$$

etc., finishing with

$$h_r = c(h_{r-1}, m_r),$$

where IV is a fixed initialization vector used for all hashes. h_r is is used as the final hash value. The idea is supposed to be that any collision in the hash function can be traced back to a collision in the compression function. This is not quite true, because if you have a block m such that $c(IV, m) = IV$, then m and $m||m$ provide a collision, even without a collision in c. To avoid this, we add a final flourish: Instead of taking h_r as the value of $H(m_1, \ldots, m_r)$, we add one more block to the message, the integer r itself encoded in an ℓ-bit block, and set

$$H(m_1, \ldots, m_r) = c(h_r, r).$$

The result really is collision-resistant if the compression function is. (Of course this begs the question of how to design a collision-resistant compression function!)

The resulting hash function is limited to a finite domain, since the number of blocks is limited to what we can encode in a single block. In practice this is not serious, since if we take a block size of 160 bits, this will allow r to be as large as $2^{160} - 1$, which is larger than any message we will ever encounter.

This iterative design is called the *Merkle-Damgard* construction. It is essentially the design used in MD5 and SHA1, whose compression function is described in gory detail in the textbook.

In 2004, cryptography researchers in China were able to find a collision in MD5. It is now known that a collision in SHA-1 can in principle be found in under 2^{65} steps, although the attack has not been carried out. As a result, these older hash functions are being retired, as is, it seems, the Merkle-Damgard construction itself. The new standards to replace them are based on something called *sponge functions.* Learning what these are is on my to-do list!

4 Applications

4.1 Password storage

You might think that a password system includes a file on the server with a big list of username-password pairs, something like:

```
straubin:my_cats_name
signoril:my_kids_name
muller:f33bl3*di$gui$3
```

but this leads to disaster if hackers ever get a hold of the file. A better strategy is to use a hash function and store hashes of the passwords:

```
straubin:h(my_cats_name)
signoril:h(my_kids_name)
muller:h(f33bl3*di$gui$3)
```

When someone attempts to log in to the system, the hash of the entered password is computed, and the entered username is used to lookup the hash of the actual password. These two are compared, and the system allows entry if the two values match.

In principle, an attacker who recovers this table will have to solve the preimage problem for these hashes in order to come up with a usable password. For a 160-bit hash like SHA-1 this ought to require computation of about 2^{160} hashes, which is completely infeasible. However, things are much easier for the attacker, because the passwords people actually use form a much smaller set than 2^{160}. For instance, a very common pattern in user passwords is four letters followed by four digits, and the digits are usually a date, so they begin with 1 or 2. The number of such passwords is accordingly

$$52^4 \times 2 \times 10^3 \approx 1.5 \times 10^9.$$

An attacker who obtains a large collection of hashed passwords can compute the hashes of these 1.5 billion likely passwords and compare them against the collected information—there's a good chance that some of them will be broken.

One defense against this kind of 'dictionary attack' is to 'salt' the passwords: concatenate a random string of, say, 20 bits to each password before hashing. Now the each entry of the password file has three fields, one containing the salt:

```
straubin:a24eb:h(my_cats_name || a24eb)
signoril:390c7:h(my_kids_name || 390c7)
muller:1ab83:h(f33bl3⋆di$gui$3 || 1ab83)
```

On login, the authentication software looks up the salt associated with a username in this file, concatenates it to the entered password, and hashes, the compares to stored hashed value. This induces a million-fold slowdown in a dictionary attack that is designed to obtain *some* password from the stolen table. If the intent of the attack is to find one particular password, salt is not much help, since the attacker gets the salt value from the table, so he knows what to concatenate to dictionary entry.

Another way to defend against the attack is to use an algorithm that computes not just a single hash of the password, but iterates the hash a large number (several thousand) times. Again, this induces a significant slowdown in the dictionary attack, but only a barely noticeable delay for a user logging in. (See the posted article on password cracking.)

4.2 Coin-tossing by telephone

Alice and Bob are coaches of opposing football teams, and would like to agree the night before the big game on which team will kick off. As is traditional, they toss a coin for this, but they are conducting the transaction by telephone:

```
Alice:  I'm tossing; call it.

Bob: Heads.

Alice:  Sorry, it was tails, so how about you kick off to us?

Bob:  Wait a second.....

Alice:  All right, I'll take a little selfie of me next to this coin,

Bob: WAIT a second...
```

Alice of course cannot tell Bob how the coin came up before he has to guess, but she cannot *not* tell him, since otherwise she can just make up something after he submits his guess. Somehow Alice has to *commit* to the outcome of the coin-toss and allow Bob to hold this commitment, without actually revealing the result. It's as though she handed Bob a locked box containing the outcome, but only provided the key after he submits his guess.

Here is one way to solve the problem: Alice chooses at random a long (let's say 256-bit) string s whose lowest-order bit is used as the toss (say 1 for heads, 0 for tails). She then reads $H(s)$ to Bob, using a collision-resistant hash function H. After Bob guesses heads or tails, Alice gives him the string s, and Bob verifies that is has matches the committed value.

Can Bob cheat? He would have to determine a preimage of $H(s)$, or at any rate the lowest-order bit of a preimage. But if H behaves as a hash function should, a preimage of s is just as likely to have 1 as its low-order bit as 0.

Can Alice cheat? She has a more interesting strategy: Generate a collision $s' \neq s$ where s' ends in 1 and s ends in 0, but $H(s) = H(s')$, and give Bob this hashed value before he guesses. If Bob guesses 0, Alice reads him s'. If he guesses 1, Alice reads him s. If H is collision-resistant, Alice cannot cheat like this.

4.3 Proof of work: hashcash and Bitcoin mining

Some spam filters have the option of implementing a protocol called **hashash.** If your mailer supports it, the header of an e-mail message you send me will include a tag such as the following:

```
X-Hashcash: 0:141111:straubin@bc.edu:9c04847139af68a
```

The field ‘141111’ is today’s date (November 11, 2014), and the next field, of course, is my e-mail address. What about those 16 hex digits at the end? To see what they are for, let’s look at the SHA-1 hash of the hashcash string:

```
>>> hashlib.sha1('0:141111:straubin@bc.edu:9c04847139af68a').hexdigest
'0000059a4558110ce31ca3a46b7fadf48f6c8163'
```

The first five hex digits (20 bits) of the hash are 0. For your computer to generate the tag, it had to hunt for values at random to append to the date and e-mail address that would make this hash start out with 20 zeros. That requires computation of about one million hashes, and it took my computer a few seconds to do this.

The idea is that a spammer cannot send a mass e-mailing to thousands, or millions, of recipients, without computing a correct tag for each recipient—it would just take too long. But the effort to generate a single message is not too great in the context of personal e-mail, and the filter can verify a message very quickly. The threshold of 20 bits can be adjusted up or down. Hashcash is just one component of the filters that use it, and a lot of the rationale is to prevent legitimate messages from being tagged as spam.

While hashcash is used, it has not been very widely adapted. However, the proof-of-work concept got a new life with the creation of the Bitcoin digital currency. New Bitcoins come into existence by being ‘mined’, and the miner is required to append random data to certain fixed data in order to obtain hashes with lots of leading 0s. As more coins are mined, the threshold of effort required is increased (meaning more and more zeros), so that now the only people who can mine coins are using application-specific chips to test many random strings in parallel.

4.4 Authentication and MACs

One of the big important applications of hash functions is for *message authentication:* How can a recipient of a message be assured that the message he is receiving is the one that was really sent, and has not been tampered with en route? It might be thought that encryption alone should guarantee this, since tampering with ciphertext should reveal itself by a message that, upon decryption, is unreadable. The problem with this approach is that it assumes there is some structure to the plaintext which will enable us to identify any tampering. But the plaintext might consist of a large amount of numerical data used to control some physical device: tampering with the ciphertext produces a ‘meaningful’ result that is likely to be harmful even if the attacker does not know what effect his changes will have. For that matter, we sometimes require *only* authentication and not privacy. For instance, an order from a retail store to a supplier for one hundred widgets may not be confidential, but both parties need to be sure that the order has not been maliciously changed to one thousand!

A *message authentication code* of MAC is basically a hash function together with a secret shared key k that the parties have previously agreed upon. If Bob wants to send an authenticated, but unencrypted, message m to Alice, he might send something like

$$(m, H(k||m)).$$

Alice extracts the first component $m,$ concatenates with the shared key $k,$ and hashes, comparing the result to the second component of the message. If an attacker alters the message $m,$ the hash will change, and the message will not be authenticated. The attacker should not be able to forge a message that will be verified even if he is allowed to see a large number of legitimate messages.

Using $(m, H(k||m))$ as the MAC is insecure if it is used in conjunction with iterated hash functions like those described above. For this reason, it is often recommended to compute the MAC as something like

$$H(k||m, H(k||m)).$$

However, it appears that the Merkle-Damgard model of hash functions is being phased out, and the creators of the newer 'sponge' family of hash functions claim that $(m, H(k||m))$ is a safe way to construct a MAC.

We leave what is probably the most important application of hash functions – digital signatures – to the next unit.