Cryptography

Hash Functions and Message Authentication Codes (MAC)

Professor: Marius Zimand

Recall that the most important services of a cryptosystem are

- confidentiality (Eve does not get information about the message).
- data integrity (Eve cannot alter the message).
- authentication (Bob knows for sure that the message comes from Alice).
- non-repudiation (Bob can prove to a third party that the message comes from Alice).

So far we have looked at confidentiality.

We start now to look at the other services.

Hash functions

Hash functions provide a basic primitive that is used mainly for

- data integrity
- to make digital signing more efficient

A hash functions is a transformation which maps strings that are arbitrarily long into strings of fixed length.

$$m \longrightarrow h(m)$$

$h(m)$ is called the digest of m (also the fingerprint of m, also the hashcode of m, also the hash value of m, $\ldots$).

In principle, hash functions are used for data integrity.

Alice sends $(m, h(m))$ to Bob.

If m has been corrupted during transmission into m', then Bob receives $(m', h(m))$, and he sees that $h(m) \neq h(m')$. (This works for accidental corruption, not for malicious corruption).

Hash functions should satisfy the following *efficiency properties*:

- extremely fast to compute
- the length of the digest (i.e., length of $h(m)$) is a fixed constant typically much smaller than the length of the message m.

It is desired that crypto hash functions have the following properties:

1. **preimage resistant:** given a digest y it should be infeasible to find messages m with $h(m) = y$.
2. **weak collision resistant:** given message m_1 it should be infeasible to find another message m_2 such that m_1 and m_2 collide, which means that $h(m_1) = h(m_2)$.
3. **strong collision resistant:** it should be infeasible to find two messages m_1 and m_2 that collide, i.e., $h(m_1) = h(m_2)$.

Note that these properties are increasingly stronger: a hash function with property (3) has property (2), and a hash function with property (2) has property (1) with the requierement than it should be unfeasible to find at least two messages having a given digest. Ideally a hash function should have property (3) to be usable in all crypto applications.

Example: $h(m) = m(\mod n)$.

This one has the efficiency properties, but none of (1), (2), or (3). So this is a bad crypto hash function.

Example: The following is an example of a hash function that is theoretically good (has property (3) if DLP is hard), but is not efficient to be used in practice.

- Choose p prime so that $q = \frac{p-1}{2}$ is also prime.
- choose α, β primitive roots of p.
- for $m \in [0, q^2 - 1]$, write $m = x_1 \cdot q + x_0$, with $x_0, x_1 \in \{0, \ldots, q-1\}$.
- define the hash function h by:

$$h(m) = \alpha^{x_0} \cdot \beta^{x_1} (\mod p).$$

- Note that $h(m) < p < 2q + 1 < q^2 - 1$, so "shrinkage" is achieved.

Designing hash functions

The current recommendations are that the value of hash function should be at least 128 bits long (for applications with low security), but preferably 160 bits long.

The input size can be arbitrarily large. In practice designing compression functions with inputs that are arbitrarily long is difficult and most designs start with a fixed compression function (from fixed input length n to fixed output length s) and then use some chaining method.

Most hash functions that are used in practice use the Merkle-Damgard method.

Merkle-Damgard construction.

Suppose f is a compression function from length n to length s.

- take $l = n - s$.
- pad the input message m with zeroes so that its length becomes a multiple of l.
- break the input m into t blocks of l bits, $m_1, m_2, \ldots, m_t$.
- initialize H to some fixed bit string of length s.

```
for $i = 1$ to $t$ do
        $H = f(H||m_i)$
return $H$.
```

This is the basic Merkle-Damgard scheme, but almost always the padding uses the so-called *length strengthening.* In this variant the input message is preprocessed by first padding with zero bits to obtain a length that is a multiple of l - l_1 (for some integer l_1) and then one final block is added which encodes using l_1 bits, the original length of the unpadded message.

Thus, these iterated hash functions (and most hash functions in the real world are of this type) require that the input length is a multiple of a certain integer called the block size of the hash function.

Real-world hash functions

The three most widely employed hash functions used to be MD5, RIPEMD-160 and SHA-1. The MD5 outputs digests of 128 bits, while RIPEMD-160 and SHA-1 produce digests with 160 bits.

In the Summer of 2005, Wang, Yin and Yu published collisions of MD5 and SHA-0 (a precursor of SHA-1) producing a big turmoil in the crypto community. Since SHA-1 is similar to SHA-0, the utilization of SHA-s is not recommended.

NIST has proposed a new set of hash functions called SHA-256, SHA-384 and SHA-512 with digests of length 256, 384 and 512 respectively. Collectively these algorithms are called SHA-2. These hash functions are derived from an earlier simpler algorithm called MD4.

NIST has organized a competition to produce SHA-3, meant to become the new federal standard. The competition has finished on Oct 2, 2012, and the winner is Keccak, designed by Guido Bertoni, Joan Daemen, Michael Peeters, and Gilles Van Assche.

SHA-2 is still considered secure.

Message Authentication Codes (MACs)

The scheme message + digest (formally, $m||h(m)$) is meant to assure the receiver that the data has not been tampered.

But using a hash function in this way requires that the digest $h(m)$ itself should be protected, as otherwise the digest itself could be tampered.

This problem is solved by using MACs, which are keyed hash functions.

This is a *symmetric cryptography* schema: the sender and the receiver share a secret key.

In symbols we calculate

$$\text{MAC }(m) \; = h_k(m),$$

where

- h is the check/compression/hash function
- k is the secret shared key
- m is the message/data

Typically the sender sends to the receiver

$$m||\mathrm{MAC}_k(m)$$

if only data integrity is needed, or

$$E_{k_1}(m)||\mathrm{MAC}_{k_2}(E_{k_1}(m))$$

if confidentiality + data integrity is needed.

The security requirements for MACs are

- only people having the shared key should be able to produce MACs or verify MACs.
- in particular, it should be unfeasible to produce the MAC of a new message if the MAC of an old message is known.

Constructing MACs

MAC from hash functions

MACs are keyed hash functions, so it is natural to use hash functions in a way that uses secret keys.

The first idea that comes to mind to construct a MAC is to concatenate the key with the message and then apply the hash function.

$$\mathrm{MAC}_k(m) = h(k||m).$$

This idea induces a vulnerability based on the fact that most hash functions use the type of chaining in the Merkle-Damgard schema.

If the adversary somehow obtains the MAC c_1 of the message m_1, she is sometimes able to obtain the MAC c_2 of the message $m_1||m_2$ without knowing the key k. This is so because it can happen that

$$\mathrm{MAC}_k(m_1||m_2) = f(\mathrm{MAC}_k(m_1)||m_2),$$

where f is the compression function used by the hash function.

Thus if someone knows the MAC of m_1, she can construct the MAC of $m_1||m_2$ without knowledge of the key k.

A similar attack (a little bit more sophisticated) works if we append the key k at the end.

To produce a secure MAC from a hash function, the following schemas are recommended. This is called a HMAC and appears in several standards.

$$HMAC = h(k||p_1||h(k||p_2||m)),$$

or

$$HMAC = h(k \oplus p1||h((k \oplus p_2)||m))$$

where p_1 and p_2 are paddings to a multiple of the full block size required by the hash function.

MACs from block ciphers

There are various schemes to obtain MACs from block ciphers. The most widely used (by far) is CBC-MAC.

Using an n-bit block cipher (for example the 64-bit DES) to construct an m-bit MAC, where $m \leq n$, is done as follows:

- the data is padded to form a sequence of n-bit blocks
- the blocks are encrypted using the block cipher in CBC mode
- the MAC is the final block (if $m = n$) or do some post-processing stage and truncation if ($m < n$).

The various standards specify the padding and the post-processing.