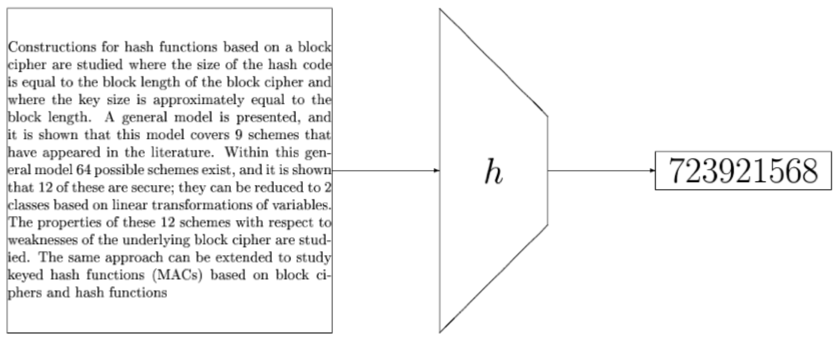
5 Hash Functions

5.1 Hash Functions

Hash Functions

A hash function is an efficient function mapping binary strings of arbitrary length to binary strings of fixed length (e.g. 128 bits), called the hash-value or digest.



Hash Functions

A hash function is many-to-one; many of the inputs to a hash function map to the same digest.

However, for cryptography, a hash function must be one-way.

- Given only a digest, it should be computationally infeasible to find a piece of data that produces the digest (pre-image resistant).

A collision is a situation where we have two different messages M and M' such that $H(M) = H(M')$.

- A hash function should be collision free.
- A hash function is weakly collision-free or second pre-image resistant if given M it is computationally infeasible to find a different M' such that $H(M) = H(M')$.
- A hash function is strongly collision-free if it is computationally infeasible to find different messages M and M' such that $H(M) = H(M')$.

Hash Functions

In theory, given a digest D we can find data M that produces the digest by performing an exhaustive search.

- In fact, we can find as many pieces of such data that we want.
- With a well constructed hash function, there should not be a more efficient algorithm for finding M.

Why do we need hash functions?

- Given any data M we can determine its digest $H(M)$.
- Since it is (computationally) impossible to find another piece of data M' that produces the same digest, in certain circumstances we can use the digest $H(M)$ rather than M.
- We cannot recover M from $H(M)$, but in general, the digest is smaller than the original data and therefore, its use may be more efficient.
- We can think of the digest as a unique fingerprint of the data.

5.2 Collisions

The Birthday Paradox
What is the probability that two people have the same birthday?

People	Possibilities	Different Possibilities
2	365^2	365×364
3	365^3	$365 \times 364 \times 363$
	$\vdots$	
k	365^k	$365 \times 364 \times 363 \times \ldots \times (365-k+1)$

P(no common birthday) = $\dfrac{365 \times 364 \times 363 \times \ldots \times (365-k+1)}{365^k}$

The Birthday Paradox
With 22 people in a room, there is better than 50% chance that two people have a common birthday.
With 40 people in a room there is almost 90% chance that two people have a common birthday.
If there are k people, there are $\frac{k(k-1)}{2}$ pairs.

- The probability that one pair has a common birthday is $\frac{k(k-1)}{2 \times 365}$.
- If $k \geq \sqrt{365}$ then this probability is more than half.

In general, if there are n possibilities then on average $\sqrt{n}$ trials are required to find a collision.

Probability of Hash Collisions
Hash functions map an arbitrary length message to a fixed length digest.

- Many messages will map to the same digest.

Consider a 1000-bit message and 128-bit digest.

- There are 2^{1000} possible messages.

- There are 2^{128} possible digests.
- Therefore there are $2^{1000}/2^{128} = 2^{872}$ messages per digest value.

For a n-bit digest, we need to try an average of $2^{n/2}$ messages to find two with the same digest.

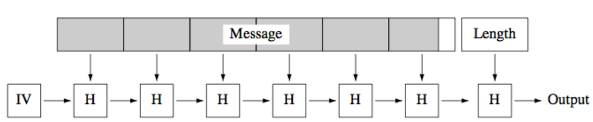
- For a 64-bit digest, this requires 2^{32} tries (feasible)
- For a 128-bit digest, this requires 2^{64} tries (not feasible)

Probability of Hash Collisions
Say B chooses 2^{32} messages M_i which A will accept that differ in 32 words, each of which has two choices:
$A \left\{\begin{array}{l}\text{will}\\ \text{promises to}\end{array}\right\} \left\{\begin{array}{l}\text{give}\\ \text{transfer to}\end{array}\right\} B$ the amount of 100 $\left\{\begin{array}{l}\text{US}\\ \text{American}\end{array}\right\}$ dollars $\left\{\begin{array}{l}\text{before}\\ \text{up to}\end{array}\right\}$ April 2013. $\left\{\begin{array}{l}\text{Then}\\ \text{Later}\end{array}\right\} B$ will $\left\{\begin{array}{l}\text{use}\\ \text{invest}\end{array}\right\}$ this amount for ...
and 2^{32} messages M'_j which A will not accept that also differ in 32 words, each of which has two choices:
$A \left\{\begin{array}{l}\text{will}\\ \text{promises to}\end{array}\right\} \left\{\begin{array}{l}\text{give}\\ \text{transfer to}\end{array}\right\} B$ the amount of $\left\{\begin{array}{l}\text{twenty}\\ \text{forty}\end{array}\right\} \left\{\begin{array}{l}\text{million}\\ \text{billion}\end{array}\right\} \left\{\begin{array}{l}\text{US}\\ \text{American}\end{array}\right\}$ dollars $\left\{\begin{array}{l}\text{which}\\ \text{that}\end{array}\right\}$ is given as a present and $\left\{\begin{array}{l}\text{should}\\ \text{will}\end{array}\right\}$ not be returned ...

Probability of Hash Collisions
By the birthday paradox, there is a high probability that there is some pair of messages M_i and M'_j such that $H(M_i) = H(M'_j)$.
Both messages have the same signature.
B can claim in court that A signed on M'_j.
Alternatively, A can choose such two messages, sign one of them, and later claim in court that she signed the other message.

5.3 Merkle-Damgård Construction

Hash Functions
Most practical hash functions make use of the Merkle-Damgård construction which divides the message M into fixed-length blocks M_1, M_2, etc., pads the last block and appends the message length to the last block.
The resultant last block (after all paddings) is denoted by M_n.
Then, the hash function applies a collision-free function H on each of the blocks sequentially:



The function H takes as input the result of the application of H on the previous block (or a fixed initial value IV in the first block), and the block itself, and results in a hash value.
The hash value is an input to the application of H on the next block.

Hash Functions
The result of H on the last block is the hashed value of the message $h(M)$:

$$\begin{array}{lcl} h_0 & = & IV = \text{a fixed initial value} \\ h_1 & = & H(h_0, M_1) \\ & \vdots & \\ h_i & = & H(h_{i-1}, M_i) \\ h_n & = & H(h_{n-1}, M_n) \\ \\ h(M) & = & h_n \end{array}$$

If H is collision-free, then h is also collision-free.

Hash Functions
Two approaches for the design of hash functions are:

1. To base the function H on a block cipher.
2. To design a special function H, not based on a block cipher.

The first approach was first proposed using DES; however the resulting hash is too small (64-bit).

- Susceptible to direct birthday attack.
- Also susceptible to "meet-in-the-middle" attack.

More modern block ciphers are suitable for implementing hash functions, but the second approach is more popular.

5.4 Commonly Used Hash Functions

Hash Functions
There are a number of widely used hash functions:

- `MD2`, `MD4`, `MD5` (Rivest).
  - Produce 128-bit digests.
  - Analysis has uncovered some weaknesses with these.
- `SHA-1` (Secure Hash Algorithm).
  - Produces 160-bit digests.

- `SHA-2` family (Secure Hash Algorithm).
  - `SHA-224`, `SHA-256`, `SHA-384` and `SHA-512`.
  - These yield digests of sizes 224, 256, 384 and 512 bits respectively.
- `SHA-3` (Secure Hash Algorithm).
  - KECCAK recently announced as winner of NIST competition.
  - Works very differently to SHA-1 and SHA-2.
- `RIPEMD`, `RIPEMD-160` (EU RIPE Project).
  - `RIPEMD` produces 128-bit digests.
  - `RIPEMD-160` produces 160-bit digests.

5.5 Applications of Hash Functions

Applications of Hash Functions
Applications of hash functions:

- Message authentication: used to check if a message has been modified.
- Digital signatures: encrypt digest with private key.
- Password storage: digest of password is compared with that in the storage; hackers can not get password from storage.
- Key generation: key can be generated from digest of pass-phrase; can be made computationally expensive to prevent brute-force attacks.
- Pseudorandom number generation: iterated hashing of a seed value.
- Intrusion detection and virus detection: keep and check hash of files on system

Information Security
Modern cryptography deals with more than just the encryption of data.
It also provides primitives to counteract active attacks on the communication channel.

- Confidentiality (only Alice and Bob can understand the communication)
- Integrity (Alice and Bob have assurance that the communication has not been tampered with)
- Authenticity (Alice and Bob have assurance about the origin of the communication)

Data Integrity
Encryption provides confidentiality.
Encryption does not necessarily provide integrity of data.
Counterexamples:

- Changing order in ECB mode.
- Encryption of a compressed file, i.e. without redundancy.
- Encryption of a random key.

Use cryptographic function to get a check-value and send it with data. Two types:

- Manipulation Detection Codes (MDC).
- Message Authentication Codes (MAC).

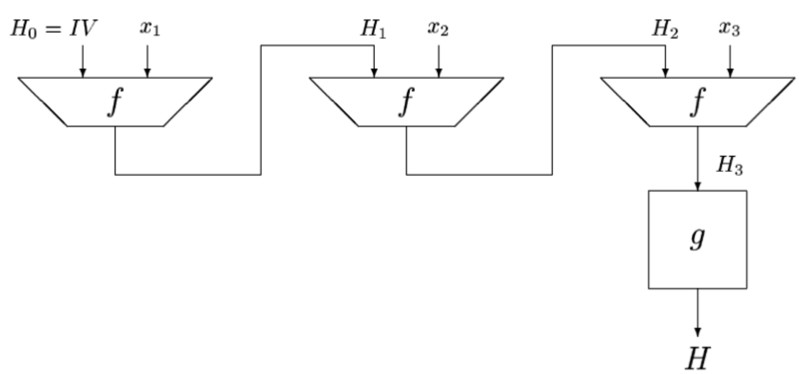
Manipulation Detection Code (MDC)
MDC: hash function without key.
The MDC is concatenated with the data and then the combination is encrypted/signed (to stop tampering with the MDC). $MDC = e_k(m||h(m))$, where:

- e is the encryption function.
- k is the secret key.
- h is the hash function.
- m is the message.
- $||$ denotes concatenation of data items.

Two types of MDC:

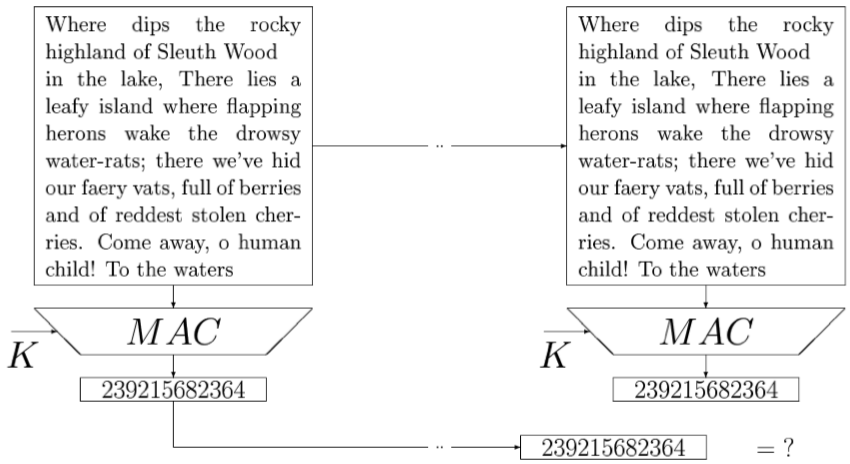
- MDCs based on block ciphers.
- Customised hash functions.

Manipulation Detection Code (MDC)
Most MDCs are constructed as iterated hash functions.



Compression/hash function f.
Output transformation g.
Unambiguous padding needed if length is not multiple of block length.

Message Authentication Code (MAC)
MAC: hash function with secret key.



Message Authentication Code (MAC)
$MAC = h_k(m)$, where:

- h is the hash function.
- k is the secret key.
- m is the message.

Transmit $m||MAC$, where $||$ denotes concatenation of data items.
Description of hash function is public.
Maps string of arbitrary length to string of fixed length (32-160 bits).
Computing $h_k(m)$ easy given m and k.
Computing $h_k(m)$ given m, but not k should be very difficult, even if a large number of pairs $\{m_i, h_k(m_i)\}$ are known.

MAC Mechanisms
There are various types of MAC scheme:

- MACs based on block ciphers in CBC mode.
- MACs based on MDCs.

- Customized MACs.

Best known and most widely used by far are the CBC-MACs.
CBC-MACs are the subject of various international standards:

- US Banking standards ANSIX9.9, ANSIX9.19.
- Specify CBC-MACs, date back to early 1980s.
- The ISO version is ISO 8731-1: 1987.

Above standards specify DES in CBC mode to produce a MAC.

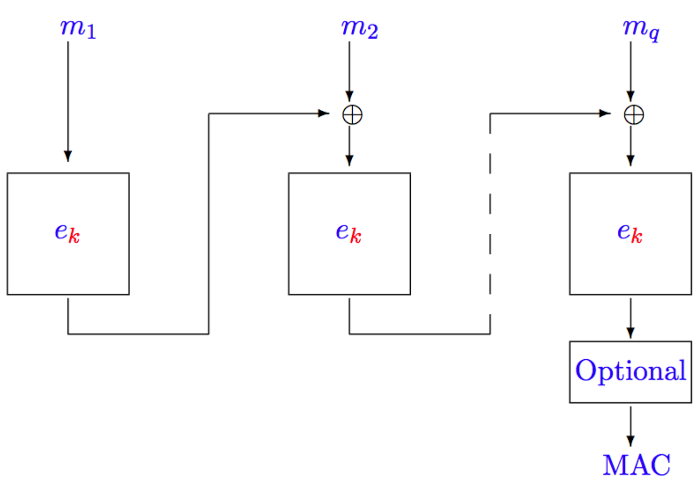
CBC-MAC
Given an n-bit block cipher, one constructs an m-bit MAC ($m \leq n$) as:

- Encipher the blocks using CBC mode (with padding if necessary).
- Last block is the MAC, after optional post-processing and truncation if $m < n$.

If the n-bit data blocks are $m_1, m_2, \ldots, m_q$ then the MAC is computed by:

- Put $I_1 = m_1$ and $O_1 = e_k(I_1)$.
- Perform the following for $i = 2, 3, \ldots, q$:
  - $I_i = m_i \oplus O_{i-1}$.
  - $O_i = e_k(I_i)$.
- O_q is then subject to an optional post-processing.
- The result is truncated to m bits to give the final MAC.

CBC-MAC



CBC-MAC: Padding
There are three possible padding methods proposed in the standards:

- Method 1: Add as many zeroes as necessary to make a whole number of blocks.
- Method 2: Add a single one followed by as many zeroes as necessary to make a whole number of blocks.
- Method 3: As for method 1, but also add an extra block containing the length of the unpadded message.

The first method does not allow detection of additional or deletion of trailing zeroes.

- Unless message length is known by the recipient.

CBC-MAC: Post-Processing
Two specified optional post-processes:

- Choose a key k_1 and compute:

$$O_q = e_k(d_{k_1}(O_q))$$

- Choose a key k_1 and compute:

$$O_q = e_{k_1}(O_q)$$

The optional process can make it more difficult for a cryptanalyst to do an exhaustive key search for the key k.

MACs based on MDCs
Given a key k, how do you transform a MDC h into a MAC?
Secret prefix method: $MAC_k(m) = h(k||m)$

- Can compute $MAC_k(m||m') = h(k||m||m')$ without knowing k.

Secret suffix method: $MAC_k(m) = h(m||k)$

- Off-line attacks possible to find a collision in the hash function.

Envelope method with padding: $MAC_k(m) = h(k||p||m||k)$

- p is a string used to pad k to the length of one block.

None of these is very secure, better to use HMAC:

$$HMAC_k(m) = h(k||p_1||h(k||p_2||m))$$

with p_1, p_2 fixed strings used to pad k to full block.

MACs versus MDCs
Data integrity without confidentiality:

- MAC: compute $MAC_k(m)$ and send $m||MAC_k(m)$.
- MDC: send m and compute $MDC(m)$, which needs to be sent over an authenticated channel.

Data integrity with confidentiality:

- MAC: needs two different keys k_1 and k_2.
  - One for encryption and one for MAC.
  - Compute $c = e_{k_1}(m)$ and then append $MAC_{k_2}(c)$.
- MDC: only needs one key k for encryption.
  - Compute $MDC(m)$ and send $c = e_k(m||MDC(m))$.

Password Storage
Storing unencrypted passwords is obviously insecure and susceptible to attack.
Can store instead the digest of passwords.

- They need to be easy to remember.
- They should not be subject to a dictionary attack.

Can make use of a salt, which is a known random value that is combined with the password before applying the hash.

- The salt is stored alongside the digest in the password file: $\langle s, H(p||s)\rangle$.
- By using a salt, constructing a table of possible digests will be difficult, since there will be many possible for each password.
- An attacker will thus be limited to searching through a table of passwords and computing the digest for the salt that has been used.

Key Generation
We can generate a key at random.

- Most cryptographic APIs have facilities to generate keys at random.
- These facilities normally avoid weak keys.

We can also derive a key from a passphrase by applying a hash and using a salt.

- There are a number of standards for deriving a symmetric key from a passphrase e.g. PKCS#5.

This key generation may also require a number of iterations of the hash function.

- This makes the computation of the key less efficient.
- An attacker performing an exhaustive search will therefore require more computing resources or more time.

Pseudorandom Number Generation
Hash functions can be used to build a computationally-secure pseudo-random number generator as follows:

- First we seed the PRNG with some random data S.
- This is then hashed to produce the first internal state $S_0 = H(S)$.
- By repeatedly calling H we can generate a sequence of internal states $S_1, S_2, \ldots,$ using $S_i = H(S_{i-1})$.
- From each state S_i we can extract bits to produce a random number N_i.
- This PRNG is secure if the sequence of values $S, S_0, S_1, \ldots$ is kept secret and the number of bits of S_i used to compute N_i is relatively small.