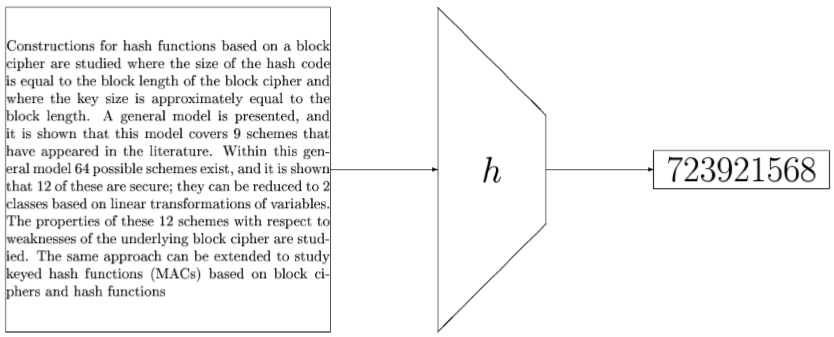
# 8 Hash Functions

## 8.1 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.

## 8.2 Collisions

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

| People | Possibilities | Different Possibilities |
|---|---|---|
| 2 | $365^2$ | $365 \times 364$ |
| 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.

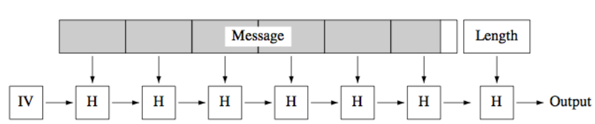
- For a 64-bit digest, this requires $2^{32}$ tries (feasible)
- For a 128-bit digest, this requires $2^{64}$ tries (not feasible)

**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.

## 8.3 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 $h(M)$:

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

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.

## 8.4 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.

**MD5**
Overview:

- Designed by Ron Rivest
- Latest in a series of MD2, MD4
- Produces a 128-bit hash value
- Until recently was the most widely used hash algorithm
- In recent times have both brute-force and cryptanalytic concerns
- Specified as Internet standard RFC1321

**MD5**
Operates as follows:

1. Pad message so its length is 448 mod 512
2. Append a 64-bit length value to message
3. Initialise 4-word (128-bit) MD buffer (A,B,C,D)
4. Process message in 16-word (512-bit) blocks:
   - Using 4 rounds of 16 bit operations on message block and buffer
   - Add output to buffer input to form new buffer value
5. Output hash value is the final buffer value

**MD5**
Compression function operates as follows:

- Each round has 16 steps of the form:

$$A = B + ((B + g(B,C,D) + X[k] + T[i]) <<< s)$$

- $A, B, C, D$ refer to the 4 words of the buffer, but used in varying permutations
  - Note this updates only one word of the buffer
  - After 16 steps each word is updated 4 times
- $g(B,C,D)$ is a different non-linear function in each round
- $T[i]$ is a constant value derived from *sin*

**MD5**
Strength of MD5:

- MD5 hash is dependent on all message bits
- Rivest claims security is good as can be
- Known attacks are:
  - (Berson, 92) attacked any one round using differential cryptanalysis (but cannot extend)
  - (Boer & Bosselaers, 93) found a pseudo-collision (again unable to extend)
  - (Dobbertin, 96) created collisions on MD5 compression function (but initial constants prevent exploit)
- Conclusion is that MD5 looks vulnerable soon

**SHA-1**
Overview:

- The Secure Hash Standard was designed by the NSA, following the structure of Rivest's MD4 and MD5.
- The first standard was SHA (now called SHA-0).
- It was later changed slightly to SHA-1, due to some unknown weakness found by the NSA.
- US standard for use with DSA signature scheme (FIPS 180-1 1995, also Internet RFC3174)
- Produces 160-bit hash values

**SHA-1**
Operates as follows:

1. Pad message so its length is 448 mod 512
2. Append a 64-bit length value to message
3. Initialise 5-word (160-bit) buffer $(A,B,C,D,E)$ to $(67452301, efcdab89, 98badcfe, 10325476, c3d2e1f0)$
4. Process message in 16-word (512-bit) chunks:
   - Expand 16 words into 80 words by mixing and shifting
   - Use 4 rounds of 20 bit operations on message block and buffer
   - Add output to input to form new buffer value
5. Output hash value is the final buffer value

**SHA-1: The Function *H***
Compression function operates as follows:

- Each round has 20 steps which replaces the 5 buffer words $(A,B,C,D,E)$ with:

$$(E + f(t,B,C,D) + (A \ll 5) + W_t + K_t), A, (B \ll 30), C, D)$$

- $t$ is the step number
- $f(t,B,C,D)$ is nonlinear function for round
- $W_t$ is derived from the message block
- $K_t$ is a constant value derived from *sin*

**SHA-1 versus MD5**

- Brute force attack is harder (160 versus 128 bits for MD5)
- Not vulnerable to any other known attacks (compared to MD4/5)
- A little slower than MD5 (80 versus 64 steps)
- Both designed as simple and compact
- Optimised for big endian CPUs (versus MD5 which is optimised for little endian CPUs)

**The SHA-2 Family**
Overview:

- NIST have issued a revision FIPS 180-2
- Adds 3 additional hash algorithms: SHA-256, SHA-384, SHA-512
- Designed for compatibility with increased security provided by the AES cipher
- Structure and detail is similar to SHA-1
- Hence analysis should be similar

**RIPEMD-160**
Overview:

- RIPEMD-160 was developed in Europe as part of RIPE project in 1996
- By researchers involved in attacks on MD4/5
- Initial proposal strengthened following analysis to become RIPEMD-160
- Somewhat similar to MD5/SHA
- Uses 2 parallel lines of 5 rounds of 16 steps
- Creates a 160-bit hash value
- Slower, but probably more secure, than SHA-1

**RIPEMD-160**
Operates as follows:

1. Pad message so its length is 448 mod 512
2. Append a 64-bit length value to message
3. Initialise 5-word (160-bit) buffer $(A,B,C,D,E)$ to $(67452301, efcdab89, 98badcfe, 10325476, c3d2e1f0)$
4. Process message in 16-word (512-bit) chunks:
    - Use 10 rounds of 16 bit operations on message block and buffer in 2 parallel lines of 5
    - Add output to input to form new buffer value
5. Output hash value is the final buffer value

**RIPEMD-160 versus MD5 and SHA-1**

- Brute force attack harder (160 bits like SHA-1 versus 128 bits for MD5)
- Not vulnerable to known attacks (like SHA-1), though stronger (compared to MD4/5)
- Slower than MD5 (more steps)
- All designed as simple and compact
- SHA-1 optimised for big endian CPUs versus RIPEMD-160 and MD5 optimised for little endian CPUs

**SHA-3 (Keccak)**
Overview:

- Alternate, different hash function to MD5, SHA-0 and SHA-1
- Design : block permutation + sponge construction
- Not meant to replace SHA-2
- Efficient hardware implementation.
- Sponge construction:
  - Message blocks XORed with the state which is then permuted (one-way one-to-one mapping)
  - State is $5 \times 5$ matrix with 64 bit words = 1600 bits
  - Reduced versions with words of 32, 16, 8,4,2 or 1 bit

**SHA-3 (Keccak)**
Block permutation:

- Defined for $w = 2^l$ bit ($w$=64, $l$= 6 for SHA-3)
- State = $5 \times 5 \times w$ bits array: notation: $a[i, j, k]$ is the bit with index $(i \times 5 + j) \times w + k$
- Block permutation function = $12 + 2 \times l$ iterations of 5 subrounds ($f = \iota \circ \chi \circ \pi \circ \rho \circ \theta$):
  - $\theta$: xor each of the $5 \times w$ columns of 5 bits parity of its two neighbours
  - $\rho$: bitwise rotate each of the 25 words by a different number, except a[0][0]
  - $\pi$: Permute the 25 words in a fixed pattern
  - $\chi$: Bitwise combine along rows
  - $\iota$: xor a round constant into one word of the state.

**SHA-3 (Keccak)**
Sponge construction = absorption + squeeze

- To hash variable-length messages by $r$ bits blocks
- Absorption:
  - The $r$ input bits are XORed with the $r$ leading bits of the state
  - Block function $f$ is applied
- Squeeze:
  - $r$ first bits of the states produced as outputs
  - Block permutation applied if additional output required

**Cryptanalysis of Hash Functions**
Recall differential cryptanalysis of block ciphers:

- Look at difference of inputs and difference of outputs after each round.
- Never have different inputs producing same outputs.

In hash functions, output is shorter than input:

- There are different inputs which do produce the same outputs.
- Need to find these inputs.

Using the Merkle-Damgård construction, we need to find messages which produce the same value for the chaining variable $h_i$ with high probability, and set all of the remaining blocks to be the same.

## 8.5 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).

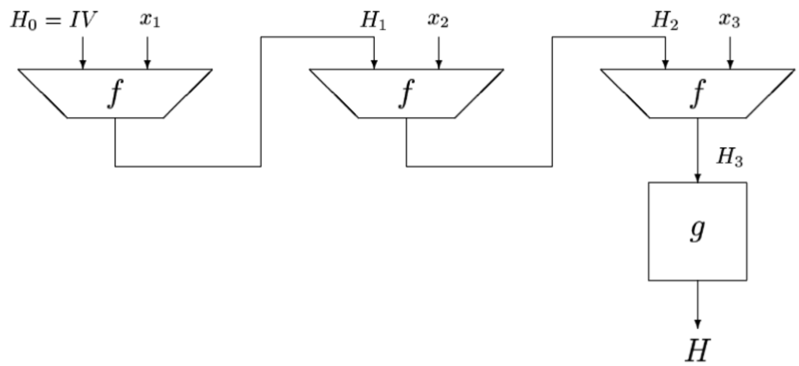
**Manipulation Detection Code (MDC)**
MDC: hash function without key.
The MDC is concatenated with the data and then the combination is encrypted/signed (to stop tampering with the MDC). $MDC = e_k(m||h(m))$, where:

- $e$ 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:

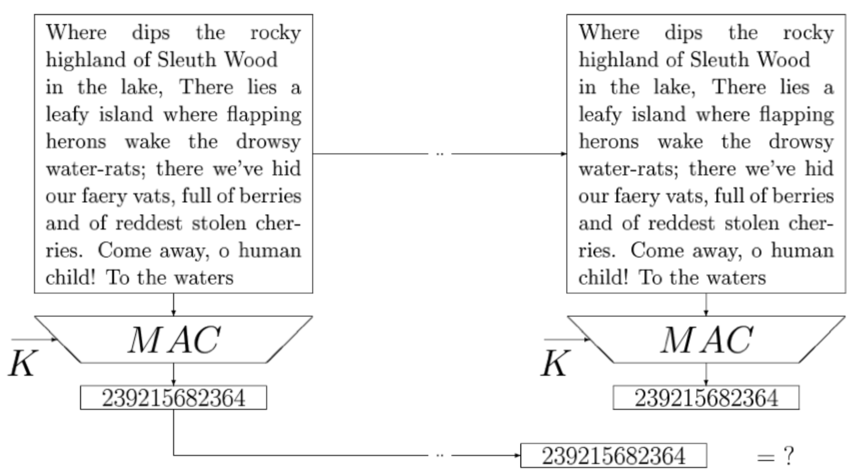
- MDCs based on block ciphers.
- Customised hash functions.

**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.

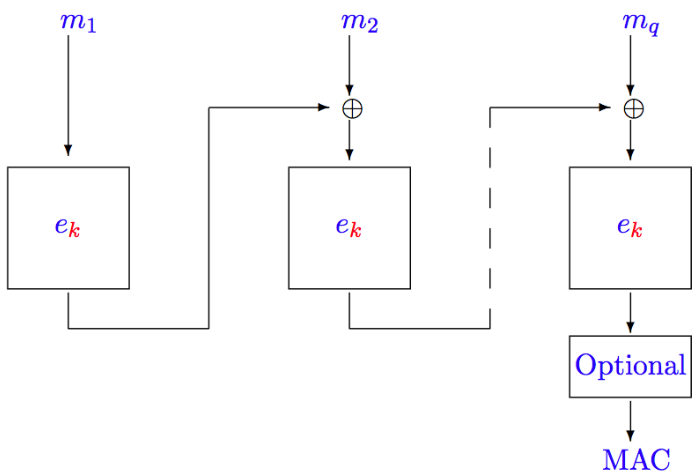
**CBC-MAC**
Given an $n$-bit block cipher, one constructs an $m$-bit MAC ($m \leq n$) as:

- 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 appends $MAC_{k_2}(c)$.
- 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.