UNIT - III

Hash Functions and Digital Signatures:

1. Authentication requirement
2. Authentication function
3. MAC (Message Authentication Code)
4. Security of hash function and MAC
5. MD5 (Message Digest 5)
6. SHA (Secure Hash Algorithm)
7. HMAC
8. CMAC

Digital Signatures and Authentication Protocols:

9. DSS
10. El-Gama
11. Schnorr
UNIT-III  
HASH FUNCTIONS AND DIGITAL SIGNATURES  

Authentication requirements: 

- Disclosure  
- Traffic analysis  
- Masquerade  
- Content modification  
- Sequence  
- Timing  
- Source repudiation  
- Destination  

Message Authentication Function: 

- Hash function  
- Message encryption  
- Message authentication code (MAC)  

Message Encryption: 

Symmetric message encryption: 

- Encryption provides authentication  
- Receiver know sender must have
x. Know content cannot be altered.

x. If msg has suitable structure, redundancy (or) a checksum to detect any changes.

\[ \text{K} \rightarrow \text{Source A} \rightarrow \text{E(K,M)} \rightarrow \text{Destination B} \rightarrow \text{D(K,E(K,M))} \rightarrow \text{M} \]

Confidentiality & authentication

Public key message encryption:

x. Encryption provides no confidentiality.

x. Sender signs msg using their private key. Then encrypts with recipient's public key.

\[ \text{M} \rightarrow \text{E}
\rightarrow \text{E(Pub,E(PrA,M))}
\rightarrow \text{D(Pub,E(PrA,M))}
\rightarrow \text{M} \]

Confidentiality, authenticity & signature.
Message Authentication Code (MAC):

* A small fixed-sized block of data

\[ MAC = C_k(M) \]

Appeared to msg when sent.

**Source** A \[ \rightarrow \] **Destination** B \[ \rightarrow \]

Compare \[ MAC(k, M) \]

**Why use a MAC?**

* Sometimes only authentication is needed
* Sometimes need authentication to persist longer than the encryption.

**Example:** Archival use.

* MAC is not a digital signature.

**Properties:**

* A MAC is a cryptographic checksum

\[ MAC = C_k(M) \]

* Many-to-one function

**Requirements:**

* Knowing a msg & MAC is infeasible to find different msg with same MAC.
* MAC should be uniformly distributed.
* MAC should depend equally on all bits of the msg.

Security of MAC's:

* MAC block ciphers have:
  * brute-force attacks exploiting.

- Strong collision resistance hash have cost $2^{m/2}$.
  - 128 bit hash looks vulnerable,
  - 160 bits better.

- MAC's with known msg-MAC pairs:
  - Can either attack key space (or)
  - MAC at least 128-bit MAC is needed for security.

* Cryptanalytic attacks exploit structure like block ciphers want brute-force attacks to be the best alternative.
* More variety of MACs so harder to guarantee 100% cryptanalytic.
Hash Functions

1. Condenses arbitrary msg to fixed size
   \[ h = H(M) \]

- Assume hash fun is public.
- Used to detect changes to msg.
- Want a cryptographic hash fun.
- Computationally infeasible to find data mapping to specific hash.
  (One-way property).
- Computationally infeasible to find two data to same hash.
  (Collision-free property).

Cryptographic hash function:

- \[ \text{Message (or) data blk, } H(\text{Variable length}) \]
- \[ \text{Hash Value, } H \]
  (Fixed length).
Hash function uses:

1. Message Integrity check (MIC)
2. Message Authentication code (MAC)
3. Digital Signature (non-repudiation)

Hash functions & Message Authentication

Symmetric key

a) Message encrypted:

```
M → [H] → E → D → M
```

b) Message unencrypted:

```
M → [H] → E → [H] → M
```

Keyed Hash:

a) Message unencrypted:

```
M → [H] → E → [H] → M
```
b) Message Encrypted: \[ E(K, [M || H(M || s)]) \]

Hash Functions & Digital Signatures - PKCS:

Other Hash Function Uses:
- Pseudorandom function (PRF)
- Pseudorandom number generator (PRNG)
- To create a one-way password file for intrusion detection & virus detection
MD5 [Message Digest Algorithm]

MD5 hashing alg was developed by Ron Rivest at MIT. It widely used secure hash algorithm.

MD5 Logic:

\[ K \]  \[ \times 512 \ \text{bits} = N + 3 \ \text{pad bits} \]

\[ K \ \text{bits} \] + \[ \bar{M} (16 \ 512 \ \text{bits}) \] \rightarrow \text{msg length k mod 2^64}

Input: IV, 128 bits

Processing: 512 bit blocks

Output: 128 bit msg digest

Steps:
1. Append padding bits
2. Append length
3. Initialize the MD5 buffer
**MD5 compression function:**

A round to one 512 bit block

```
<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
</table>
```

```
\[
A' = A + \text{a function of } B
\]
```

```
\[
B' = B + \text{a function of } C
\]
```

```
\[
C' = C + \text{a function of } D
\]
```

```
\[
D' = D + \text{a function of } A
\]
```

Strength of MD5:

* Every bit of the hash code is a function of every bit in the input.
* When two messages are chosen at random, they will not have the same hash code.

Attacks on MD5:

* Differential cryptanalysis
* Pseudo-collision
SHA (Secure Hash Algorithm):

- Designed by NIST & NSA in 1993.
- Revised in 1995 as SHA-1.
- Based on design of MD4, with key differences. Produces 160-bit hash values.

SHA Versions:

<table>
<thead>
<tr>
<th>SHA</th>
<th>SHA-224</th>
<th>SHA-256</th>
<th>SHA-384</th>
<th>SHA-512</th>
</tr>
</thead>
<tbody>
<tr>
<td>digest size</td>
<td>160</td>
<td>224</td>
<td>256</td>
<td>384</td>
</tr>
<tr>
<td>msg size</td>
<td>&lt;64</td>
<td>&lt;64</td>
<td>&lt;64</td>
<td>&lt;128</td>
</tr>
<tr>
<td>blk size</td>
<td>512</td>
<td>512</td>
<td>512</td>
<td>1024</td>
</tr>
<tr>
<td>word size</td>
<td>32</td>
<td>32</td>
<td>32</td>
<td>64</td>
</tr>
<tr>
<td>no. of steps</td>
<td>80</td>
<td>64</td>
<td>64</td>
<td>80</td>
</tr>
</tbody>
</table>

SHA-512 compression fun:
- Processing msg in 1024-bit blocks.
- Consists of 80 rounds.
  - Updating a 512-bit buffer.
  - Using a 64-bit value & a round constant based on cube root of first 80 prime nos.
SHA-512 Overview

- Message
  - 2^L bits
  - 1024 bits

- H0, H1, H2, H3

- F

- H0 = H0 + 512 bits
- H1 = H1 + 512 bits
- H2 = H2 + 128 bits
- H3 = H3 + 128 bits

- Word-by-word addition
  - + = word-by-word addition mod 264

Keyed Hash Functions as MACs:

- MAC based on a hash fun,
  - hash fun + faster
  - crypto hash fun code is widely available.

- KeyedHash = Hash(key | Message)

- Eventually led to development of HMAC.
HMAC

* HMAC objective is to use, without modifications, hash funs.
* Allow for easy replaceability of embedded hash fun.
* Preserve original performance of hash fun without significant degradation.
* Use & handle keys in a simple way.
* Have well understood cryptographic analysis of authentication mechanism strength.
* HMAC specified as Internet Standard RFC 2104.

HMAC(N) = Hash [(k^t XOR opad) ∥ Hash [(k^t XOR ipad) ∥ N]]

| k^t - is the key padded out to size

Hash function can be used,

Ex: MD5, SHA-1, RIPEMD-160, Whirlpool.
HMAC overview:

\[ \text{HMAC} (k, m) \]

\[ \text{IV} \rightarrow \text{Hash} \]

\[ k + \]

\[ \text{iv} \rightarrow \text{b bits} \rightarrow \text{Hash} \]

\[ \text{H} (k || m) \]

HMAC security:

* It is related with hash algorithm.
* Attacking HMAC requires,
  * brute force attack on key used
  * birthday attack.

(But since keyed would need to observe a very large number of msgs.)

* hash function used based on speed versus security constraints.
CMAC Cipher-based Message Authentication Code

- CMAC is widely used in government and industry.
- However, it has a size limitation.
- CMAC can overcome using 2 keys and padding. This is adopted by NIST SP 800-38B.

Overview:

(a) Message length is an integer multiple of block size.

(b) Message length is not an integer multiple of block size.
Authenticated Encryption:

* Simultaneously protect confidentiality and authenticity of communications.

Approaches:

1. Hash-then-encrypt: \( E(k_1, (M1 \oplus H(M)) \)
2. MAC-then-encrypt: \( E(k_2, (M1 \oplus MAC(k_1, M)) \)
3. Encrypt-then-MAC:
   \[
   (C = E(k_2, M), T = MAC(k_1, C))
   \]
4. Encrypt-and-MAC:
   \[
   (C = E(k_2, M), T = MAC(k_1 \oplus M))
   \]

* Decryption/verification straightforward.

Digital signatures and authentication protocols:

Digital Signatures:

* Digital signatures provide the ability to verify author, date, and time of signature, authenticate message content, and to be used by third parties to resolve disputes.
Digital Signature Model:

Bob

Transmit

Alice

Bob's Public Key

Digital Signature Generation Algorithm

Digital Signature Algorithm

Bob's Private Key

Signature for M

Valid (or) not Valid

Example:

Bob

1. Hash M
2. Encrypt h using Bob's Private Key
3. Bob's Signature for M

Alice

1. Receive M
2. Hash M
3. Decrypt h
4. Compare
5. Validate Signature (Valid or not Valid)
Attacks:

* Key-only attack
* Known msg
* Generic chosen msg
* Directed chosen
* Adaptive

Break success levels:

* Total break
* Selective forgery
* Existential

Digital signature requirements:

* It depends on the msg signed & use info unique to sender. Because to prevent both forgery & denial.
* Must be relatively easy to produce and recognize & verify.
* Must be computationally infeasible to forge.
Direct digital signatures:

1. Involve only sender & receiver.
2. Receiver has sender's public key.
3. It made by sender signing entire msg (or) hash with private key.
5. Sign first then encrypt msg & signature.

DSS (Digital Signature Standard):

1. DSS is US Govt. approved signature scheme. Designed by NIST & NSA in early 90's.
2. Published as FIPS-186 in 1991.
4. DSA is digital signature only, unlike RSA is a public-key technique.
Digital Signature Algorithm (DSA):

- Creates a 320-bit signature with 512-1024 bit security, faster and smaller than RSA.
- Security depends on difficulty of computing discrete logarithms.
- Variant of ElGamal and Schnorr schemes.
**DSA Key Generation:**

* Shared Global public key values \((p, q, g)\):
  
  1. Choose 160-bit prime number \(q\).
  2. Choose a large prime \(p\) with \(2^{l-1} < p < 2^l\).
  3. Choose \(g = h^{(p-1)/q} \mod p\), where \(1 < h < p-1\).

* Uses choose private & compute public key:
  
  1. Choose random private key \(x < q\).
  2. Compute public key \(y = g^x \mod p\).

**DSA Signature Creation:**

* To sign a msg \(M\) the sender:
  
  1. Generates a random signature key \(k\), \(k < q\).
  2. Then compute,
     
     \[
     r = (g^k \mod p) \mod q
     \]
     
     \[
     s = [k^{-1} \cdot [H(M) + x \cdot r]] \mod q
     \]
  3. Sends signature \((r, s)\) with msg.
DSA Signature Verification

> Having received \( r_s \) and signature \( (r_1, r_2) \)

> To verify a signature, recipient computes:

\[
W = s^{-1} \mod q
\]

\[
u_1 = [H(N) \cdot w] \mod q
\]

\[
u_2 = (rw) \mod q
\]

\[
V = \left[ (g^{u_1} \cdot g^{u_2}) \mod p \right] \mod q
\]

> If \( V = r_1 \) then signature is verified.

\[
S = f_1(H(M), k_1, k_2, r_1, \tilde{r}_1, q)
\]

\[
= k_1^{-1}(H(M) + x_1 r_1) \mod q
\]

\[
Y = f_2(k_1, p, q, g) = (g^{k_1} \mod p) \mod q
\]
ElGamal Digital Signatures

- Signature variant of ElGamal, related to D-H. So uses exponentiation in a finite (Galois) with security based difficulty of computing discrete logarithms, as in D-H.

- Use private key for encryption (signing) & uses public-key for decryption (verification).
Each user (ex: A) generates their key:

1. Choose a secret key (number): \( \text{ } 24 \)
   \[ 1 < X_A < q - 1 \]

2. Compute their public key:
   \[ Y_A = a^{X_A} \mod q \]

3. Alice signs a msg \( M \) to Bob by computing the hash \( m = H(M), \ 0 \leq M \leq (q - 1) \)
   * Choose random integer \( k \) with
   \[ 1 \leq k \leq (q - 1), \ \& \ \gcd(k, q - 1) = 1. \]

4. Compute temporary key:
   \[ S_1 = a^k \mod q \]

5. Compute \( k^{-1} \) the inverse of \( k \mod (q - 1) \)

6. Compute the value:
   \[ S_2 = k^{-1} (m - X_A S_1) \mod (q - 1) \]

7. Signature is \( (S_1, S_2) \).

Any user B can verify the signature by computing...
\[ v_1 = a^k \mod q \]
\[ v_2 = y_A s_1 s_3 \mod q \]

> Signature is valid if \( v_1 = v_2 \)

**ElGamal Signature Example:**

> Use field \( \mathbb{GF}(19) \) \( q = 19 \) so \( a = \_ \).

> Alice computes her key:

| \( y_A = 10^{16} \mod 19 \) | \( = 4 \) |

> A choose \( x_A = 16 \) and computes

> Alice signs msg with hash \( m = 14 \) as \( (3, 14) \)

> choosing random \( k = 5 \) which has \( \text{gcd}(18, 5) = 1 \)

> computing \( s_1 = 10^5 \mod 19 \) = 8

> finding \( k^{-1} \mod (q - 1) = 5^{-1} \mod 18 \)

> computing \( s_2 = 11(14 - 16 \cdot 3) \mod 18 \)

> any user B can verify the by computing \( v_1 = 10^{14} \mod 19 \)

\[ v_3 = 4^3 \cdot 3^5 \mod 5184 \]

Since \( y_1 = 16 \) agreed.
Schnorr Digital Signatures:

* Uses exponentiation in a finite (Galois) field based on discrete logarithms, as in D-H.

* It minimizes msg dependent computation - multiplying a 2n-bit integer with an n-bit integer.

* Using a prime modulus p
  \[ p - 1 \] has a prime factor q of appropriate size.

  Typically, p 1024-bit and q 160-bit each.

Schennor key setup:

* choose suitable primes, p, q
* choose a such that \( a^q = 1 \) mod p
* \( (a, p, q) \) are global parameters for all
* each user (ex: A) generates a key
  * choose a secret key (number) \( 0 < s_A < q \)
  * compute their public key \( y_A = a^{s_A} \mod q \)
User signs msg by:

- Choosing random \( r \) with \( 1 \leq r < p \) and computing \( x = a^r \mod p \)
- Concatenate msg with \( x \) and hash result to computing \( e = H(M \| x) \)
- Computing \( y = (x + 8e) \mod q \)
- Signature is pair \( (e, y) \)

Any other user can verify the signature as follows:

- Computing \( x' = a^y \mod p \)
- Verifying that \( e = H(M \| x') \).