

# Computer and Network Security

## 414 Fall 2008

Udaya Shankar  
shankar@cs.umd.edu

### CRYPTO

9/30/2008 shankar

crypto slide 1

#### Issues in security of information in computer networks

- **Confidentiality:** Information should be accessible only to select few.
- **Integrity:** When A receives information supposedly from B, the information originated from B and has not been changed by anybody else.
- **Non-repudiation:** After B receives information from A, A cannot claim it never sent the information and B cannot claim it never received it.
- **Anonymity:** The sender of a message is guaranteed to be from a given set S of users, but the sender cannot be further identified. NOT COVERED.
- **Availability:** A is not prevented from viewing information because of lack of resources (because of malicious users imposing unnatural workload, denial-of-service attack). NOT COVERED.
- **Others??**

9/30/2008 shankar

crypto slide 2

## How were they solved before computer networks

- **Confidentiality:** storing the information in a physically "secure" place (in a bank, in a safe, under the mattress, in your head, etc).
- **Integrity:** putting the information on hard-to reproduce paper (eg, dollar bills), affixing a hard-to reproduce signature, etc.
- **Non-repudiation:** having a trusted third party record the exchange of information (e.g., postal service registered mail, sub-poena, etc).
  - C more trusted than B because
    - C is appointed a government; C employs B; C is bigger than B;
    - C has acted fairly in the past; etc.
- **Unique issues arising with computers**
  - Much higher rate of transactions.
  - Much larger and more dynamic set of users
  - Information is in soft (not hard) copy

## Basic tools of Information security

- **Cryptography:**
  - keys + encryption/decryption algorithms (known)
  - encryption: plaintext + key → ciphertext
  - decryption: plaintext ← ciphertext + same/related key
  - Solves static (trust/communication) situation
- **Key-distribution protocols/distributed algorithms:**
  - use cryptographic functions to manage dynamic users
  - key distribution, authentication, ...
  - Kerberos, PKI, ...
- **Implementation of crypto and key-distribution**
  - OS/PL/runtime protection issues
    - user memory protection, access control, no stack overflow, ...
    - uC vs Java, trojan horses, runtime protection, code review, ...
  - Sys admin issues

## Introduction to Cryptology (NS chapter 3)

Encryption: plaintext + key  $\rightarrow$  ciphertext

Decryption: ciphertext  $\leftarrow$  plaintext + same/related key

- Key is secret. Encryption/decryption algorithms not secret.
- Given plaintext and ciphertext, computationally hard to get key.
- Attacks depend on what is available
  - Ciphertext available: search key/plaintext space, replay, ...
  - Plaintext-ciphertext pairs available: ...
  - Chosen plaintext-ciphertext pairs available: ...
- Types of cryptographic functions:
  - Secret key (symmetric key): DES, AES, ...
  - Public key (asymmetric): RSA, DH, ...
  - Hash functions (of cryptographic kind): MD5, SHA-1, ...

## Secret (symmetric) key crypto

- Single key: used in encryption and in decryption.
- Ciphertext about the same length as plaintext.
- Provides confidentiality over insecure channel/storage.
  - A and B share secret key K
  - A sends K(plaintext).
  - B receives and decrypts using K.
- Provides authentication over insecure channel:
  - A and B share secret key K
  - A sends random number  $r_A$  to B, and expects  $K(r_A)$  back
  - B sends random number  $r_B$  to A, and expects  $K(r_B)$  back
  - This particular one is flawed.

## Secret (symmetric) key crypto (cont.)

- Provides integrity over insecure channel:
  - A and B share secret key K
  - A sends plaintext and **fixed-length** part of  $K(\text{plaintext})$  to B.
  - Fixed-length part of  $K(\text{plaintext})$  can be, e.g., last 128 bits
  - Called MAC (msg authentication code) or MIC (msg integrity code))
  - B receives plaintext, computes  $\text{MAC}(\text{plaintext})$  and checks for match with received MAC
  - This particular one provides attacker with plaintext-ciphertext pairs

## Public key cryptography (asymmetric key)

- Each principal has two keys:
  - private key (not shared)
  - public key (shared with world).
  - Plaintext encrypted with one can only be decrypted with the other.
- Confidentiality:
  - B transmits  $\text{pubkey}_A(\text{plaintext})$ . A decrypts using  $\text{privkey}_A$ .
- Integrity and digital signature (non-repudiation)
  - A transmits  $\text{privkey}_A(\text{plaintext})$
  - Anyone with  $\text{pubkey}_A$  can decrypt and know that it could only have been sent by A.
- Public key crypto can do anything secret key crypto does
  - But orders slower.
  - So use public key crypto to exchange per-session secret key and do secret-key crypto on data
    - eg, B creates per-session key K and sends  $\text{pubkey}_A(K)$  to A

## Hash algorithms (of cryptographic kind)

- Transform plaintext msg of arbitrary length
  - to short fixed-length hash  $h(\text{msg})$  (eg, 128 bits)
    - eg,  $h(\text{msg})$  is  $(\text{msg}+C)$  squared and take middle 128 bits.
- Easy to compute  $h(\text{msg})$  from msg
- Not easy to find msg1 and msg2 such that  $h(\text{msg1})=h(\text{msg2})$
- Applications:
  - password hashing
  - Msg integrity: given secret S, send  $(\text{msg}|S)$  and  $h(\text{msg}|S)$ .
  - Use digital signature on  $h(\text{msg})$  rather than on msg

## Secret Key Crypto (NS chapter 3)

- Consider **fixed-length** message of k bits here.
  - Variable length addressed later.
- Fixed-length message + Fixed-length key  $\rightarrow$  message-length output
  - DES: 64 bit message + 56 bit key
- If key length j is too small, insecure. If j is too large, expensive.
- Want function S mapping k-bit msg to k-bit output such that:
  - For decryption, S must be 1-1 mapping from  $2^k$  to  $2^k$ .
  - For security, S must be “random”:
    - even if msg1 and msg2 differ in just one bit,
    - $S(\text{msg1})$  and  $S(\text{msg2})$  differ in many bits (approx k/2 bits).
  - So S cannot be a “simple” function
    - e.g.,  $S(\text{msg}) = \text{msg} \oplus \text{key}$ . Or  $S(\text{msg}) = \text{msg}$  bits in reverse order

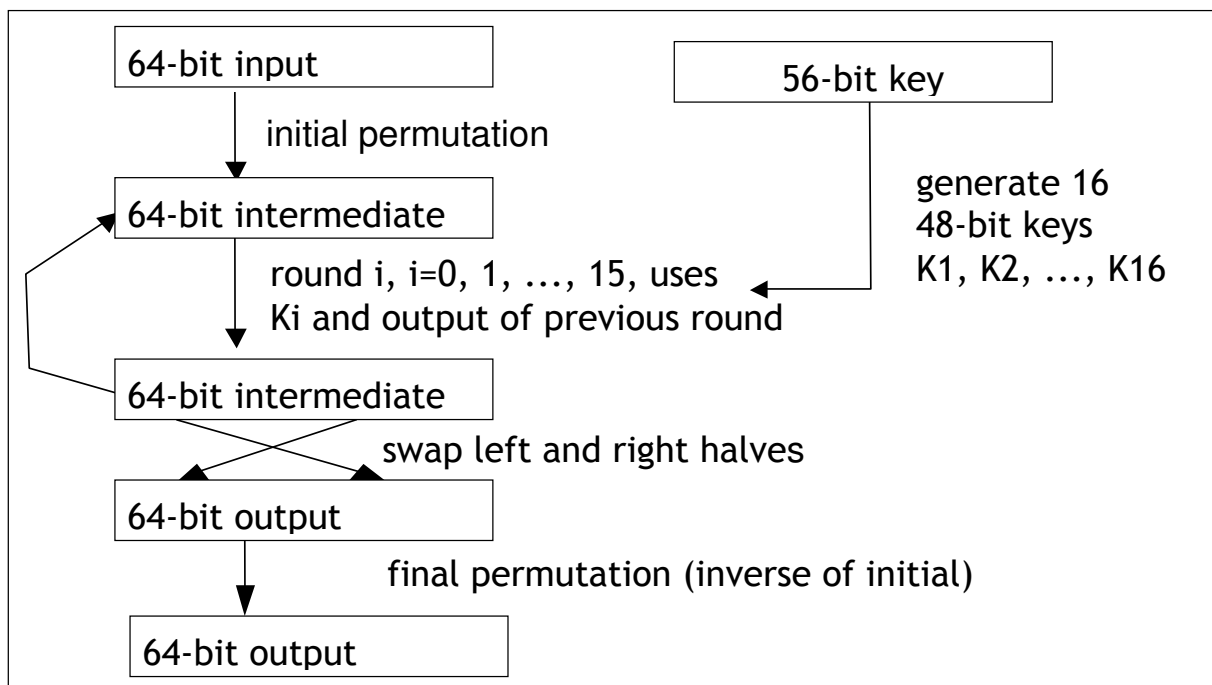
## Secret Key Crypto (contd.)

- **Conceptually simple secret-key algorithm S**
  - “Substitution” table: random permutation of all N-bit strings.
  - $S(i)$  is  $i$ th row of table
  - Table obtained with physical-world randomness (eg, coin toss).
  - Pro:  $S$  is perfectly random
  - Con: need to store table of size  $k \cdot 2^k$ . Impractical for  $k=64$
- **Basic approach:** mix permutations and substitutions
  - Divide  $k$ -bit block into  $p$ -bit blocks for reasonably small  $p$  (eg,  $p=8$ ).
  - Use  $p \times p$  substitution tables “garble”  $p$ -bit output blocks.
  - Concatenate the  $p$ -bit output blocks to get a  $k$ -bit block
  - and permute to get garbled  $k$ -bit output block.
  - Repeat 1, 2, 3 for  $n$  rounds, where  $n$  is large enough to get good scrambling.
- **Decryption**, ie, reversing, is no more expensive.  
Often can be done with the same algorithm/hardware.

9/30/2008 shankar

crypto slide 11

## DES

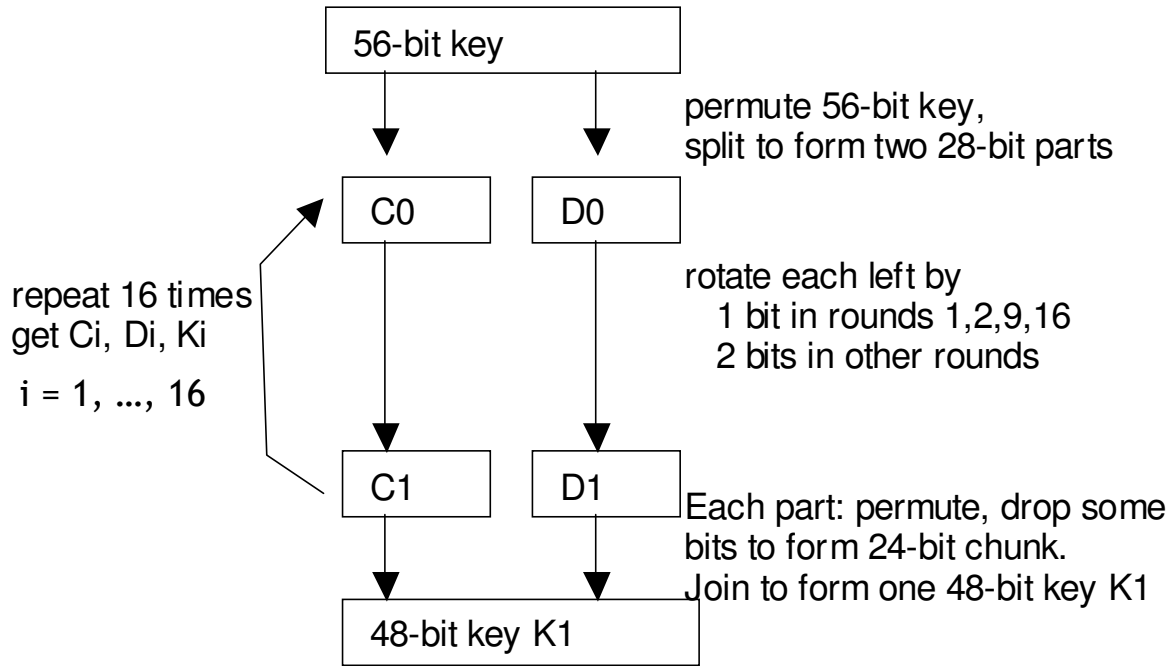


Final permutation is inverse of Initial. Not of security value (why?)

9/30/2008 shankar

crypto slide 12

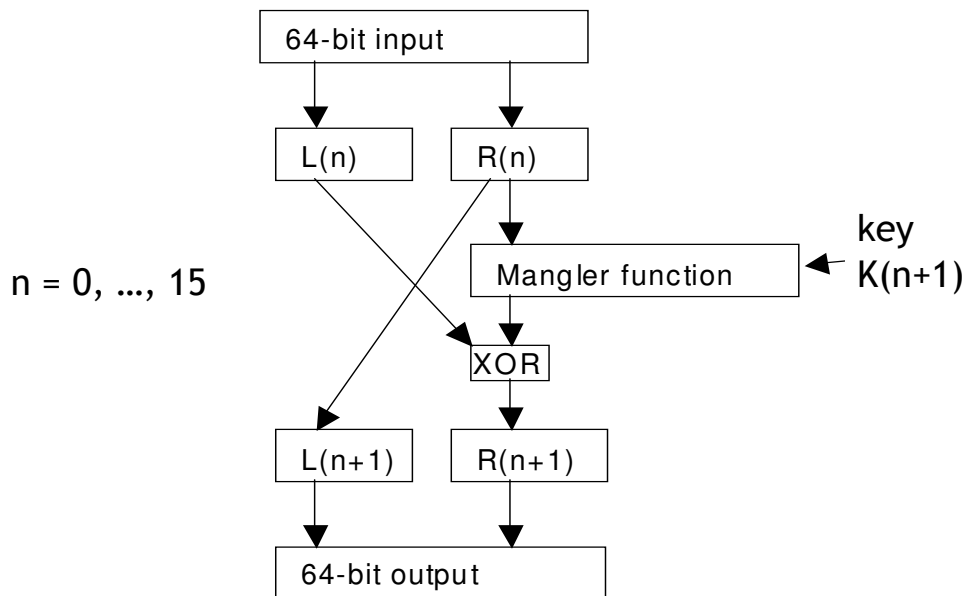
## DES: Generation of K1, K2, ..., K16



9/30/2008 shankar

crypto slide 13

## DES encryption round



- DES decryption round: given  $R(n+1) | L(n+1) \rightarrow R(n) | L(n)$   
same as encryption with arrows reversed except for mangler function

9/30/2008 shankar

crypto slide 14

DES: decryption = encryption with  $K_i$ 's in reverse order

```
DES_encryption {
a1: L0|R0 ← iperm(dblk);
a2: for n= 0, ..., 15 do
a3: Ln+1 ← Rn;
a4: Rn+1 ← Mnglrn(Rn,Kn+1)⊕Ln;
    //Yields L16|R16

a5: L17|R17 ← R16|L16 ;
a6: crblk ← ipermInv( R16|L16 );
}

// key order: K1, ..., K16
```

```
DES_decryption {
b1: R16|L16 ← iperm(cblk); //a6 bkw
b2: for n = 15, ..., 0 do //a2 bkw
b3: Rn ← Ln+1; // a3 bkw
b4: Ln ← Mnglrn(Rn,Kn)⊕Rn+1; //a4 bkw
    // sets Ln to X such that
    // Rn+1 ← Mnglrn(Rn, Kn)⊕ X
    // Yields R0|L0
b5: L0|R0 ← swap(R0|L0 ); // a5 bkw
b6: dblk ← ipermInv(L0|R0 ); // a1 bkw
}

// key order K16, ..., K1
```

## DES: Mangler function

**32-bit R + 48-bit K → 32-bit output**

- 32-bit R is split up into 8 6-bit chunks (duplicating some bits)
- 48-bit K split up into 8 6-bit chunks
- chunk i of R ⊕ chunk i of K
- Put 6-bit result in S box i (different for each round)
- Output of S box is 4-bit chunk
  
- All chunks concatenated and permuted to get 32 bit output

## DES: Weak and semi-weak keys

- **4 weak keys:** generate  $C_0=D_0$ =all ones or all zeros
- **12 semi-weak keys:** generate  $C_0$  and  $D_0$  of alternating 0 and 1

**A weak key  $x$  is its own inverse, i.e., for any block  $b$ :  $E_x(b) = D_x(b)$**

Proof

A weak DES key has each of  $C_0$  and  $D_0$  to be all ones or all zeroes.

Since each  $C_i$  is a permutation of  $C_0$ , each  $C_i$  is the same as  $C_0$ .

Since each  $D_i$  is a permutation of  $D_0$ , each  $D_i$  is the same as  $D_0$ .

Each per-round key  $K_i$  depends only on  $C_i$  and  $D_i$ .

So the per-round keys  $K_1, \dots, K_{16}$  are all equal to each other.

So the key sequence  $K_1, \dots, K_{16}$  (used in encryption) is the same as the key sequence  $K_{16}, \dots, K_1$  (used in decryption).

So encryption and decryption are the same, i.e.,  $E_x(b) = D_x(b)$ .

So  $E_x(E_x(b)) = b$ .

9/30/2008 shankar

crypto slide 17

## DES: Weak and semi-weak keys

**A semi-weak key  $x$  is the inverse of another semi-weak key  $y$ , i.e., for any block  $b$ :  $E_x(\text{block}) = D_y(\text{block})$**

Proof

Let  $\langle K_1(x), \dots, K_{16}(x) \rangle$  be the per-round keys obtained from  $x$ .

Show that there is another semi-weak key  $y$  such that  $y$

$\langle K_1(x), \dots, K_{16}(x) \rangle = \langle K_{16}(y), \dots, K_1(y) \rangle$ .

Hence for any block  $b$ :  $E_x(\text{block}) = D_y(\text{block})$

9/30/2008

## Multiple Encryption DES (EDE or 3DES)

- Makes DES more secure
  - Encryption: encrypt key1 -> decrypt key2 --> encrypt key1
  - Decryption: decrypt key1 -> encrypt key2 --> decrypt key1
- EE (encrypting twice) with same key is not effective.
  - Just equivalent to using another single key.
- EE with key1 and key 2 is not so good.
- Given  $\langle m_1, c_1 \rangle, \langle m_2, c_2 \rangle, \dots$ ,
  - there is an attack that requires  $2^{56}$  storage.
    - Table A with  $2^{56}$  entries  $\langle \text{key } K_i, E(K_i, m_1) \rangle$ , sorted by column 2.
    - Table B with  $2^{56}$  entries  $\langle \text{key } K_i, D(K_i, c_1) \rangle$ , sorted by column 2.
    - Do join of Table A and Table B.
    - Each match provides candidate  $\langle K_A, K_B \rangle$  for  $\langle \text{key1}, \text{key2} \rangle$ .
    - Use  $\langle m_2, c_2 \rangle$ , etc. to weed out false candidates.

## IDEA, AES

## RC4 encryption algorithm

- Stream cipher (one time pad), can use variable length key.
- Key stream independent of plaintext
- 8x8 S-box. each entry is a key-permutation of 0..255

### S-box initialization

```
byte S[0..255] ← 0..255; // S[i]=i
byte i := 0; j ← 0; // counters
byte K[0..255] ← key | ... |key;
for i = 0 to 255 do
    j ← ( j + S[i] + K[i] ) mod 256;
    swap S[i] and S[j]
```

### Generate random byte

⊕ with pt/ct for encrypt/decrypt

```
i ← (i+1) mod 256;
j ← (j+S[i]) mod 256;
swap S[i] and S[j];
return S[ ( S[i] + S[j] ) mod 256 ] ;
```

# Encrypting Large Messages (NS chapter 4)

## Encrypting large msg given method to encrypt a k-bit block

- Pad message to multiple number of blocks:  $\text{msg} = (M_1, M_2, \dots)$
- Use block encryption repeatedly to get ciphertext =  $(C_1, C_2, \dots)$ 
  - Same  $M_i$ 's get encrypted to different  $C_i$ 's
  - Repeated encryptions of same msg result in different ciphertexts.
  - Ciphertext cannot be changed to cause predictable change to decrypted plaintext.
- **Various methods:** ECB, CBC, CFB, OFB, CTR, others

---

## Electronic Code Book (ECB)

- Obvious approach: encrypt/decrypt each block independently
- Encryption:  $C_i = E_K(M_i)$
- Decryption:  $M_i = D_K(C_i)$
- not good: repeated blocks get same cipherblock

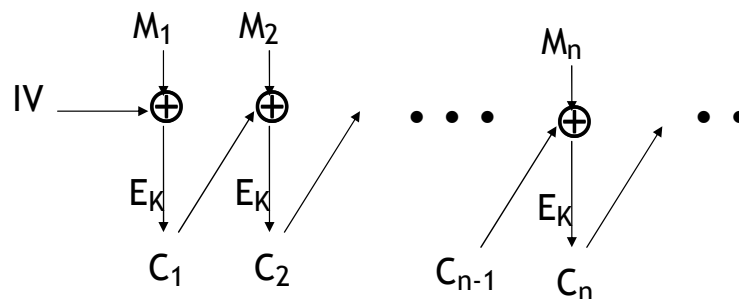
9/30/2008 shankar

crypto slide 21

## Cipher Block Chaining (CBC)

### • Encryption:

$\oplus M_i$  with random  $R_i$   
obtained from  $C_{i-1}$



- $C_i = E_K(M_i \oplus C_{i-1})$ , where  $C_0$  is a random IV (initialization vector)
- Transmit IV and  $C_1, \dots, C_n$
- **Decryption:** reverse arrows; change  $E_K$  to  $D_K$ 
  - $M_i = D_K(C_i \oplus C_{i-1})$ , where  $C_0$  is IV
- **Attack 1:** Modify  $C_n$ : garbles  $M_n$  unpredictably and  $M_{i+1}$  predictably other  $M_i$ 's unchanged. Can use a CRC to overcome this.
- **Attack 2:** Exchanging cipherblocks can counteract CRC to some extent

9/30/2008 shankar

crypto slide 22

## Output Feedback Mode (OFB)

### 64-bit OFB

- Generate stream cipher  $B_0, B_1, \dots$ , where  $B_0$  is IV and  $B_i = E_K(B_{i-1})$
- Then  $C_i = B_i \oplus M_i$
- So a one-time pad that can be generated in advance.
- One-time pad:
  - Attacker with <plaintext, ciphertext> can obtain  $B_i$ 's
    - and so generate ciphertext for any plaintext

### k-bit OFB ( $k < 64$ )

- Generate stream cipher in k-bit chunks, rather than 64-bit chunks.
- Let  $X_i = E_K(B_{i-1})$ , where  $B_0$  is 64-bit IV
- Let  $Y_i$  be k leftmost bits of  $X_i$
- $C_i = Y_i \oplus M_i$
- $B_i$  is rightmost 64 bits of  $B_{i-1} \parallel Y_i$

## Cipher Feedback Mode (CFB)

### 64-bit CFB

- Like OFB except that output  $C_{i-1}$  is used instead of  $B_i$
- $C_i = M_i \oplus E_K(C_{i-1})$  where  $C_0$  is IV
- Cannot generate one-time pad in advance.

### k-bit CFB ( $k < 64$ )

- Generate ciphers in k-bit chunks, rather than 64-bit chunks.
- Let  $X_i = E_K(B_{i-1})$ , where  $B_0$  is 64-bit IV (pad with zeros on left if needed).
- Let  $Y_i$  be k leftmost bits of  $X_i$
- $C_i = Y_i \oplus M_i$
- $B_i$  is rightmost 64 bits of  $B_{i-1} \parallel C_i$

---

## Counter Mode (CTR)

- See text

---

## 3DES on Large Messages

3DES is used with CBC on the “outside” not “inside”

Using with CBC on inside eliminates self-synchronization of received ciphertext (ie, if some ciphertext is garbled, everything is lost)

---

## MACs from encryption/decryption (NS chapter 4)

### Ensuring integrity (but not confidentiality):

- CBC, CFB, OFB, ... do not protect against “undetectable” modifications by attacker knowing the plaintext
- Of course, a human may find something fishy
  - Similarly, a computer can check for structure in plaintext
- Need a cryptographic checksum.
- Standard way: send CBC **residue** (last block in CBC encryption) along with the plaintext message and IV.

## Ensuring confidentiality and integrity of a large message

- Not ok: Send CBC encrypted message and CBC residue.
  - Just repeats the last cipherblock
- Not ok: CBC\_Encrypt[ plaintext, CBC\_residue[ plaintext ] ]
  - Last block is encryption of zero (  $\oplus$  of last cipherblock with itself)
- Not ok: Encrypt[ plaintext, noncryptographic checksum (eg, CRC)]
  - Almost works. Subtle attacks are known.
- Ok: Encrypt\_Key2[ plaintext, CBC\_residue\_Key1[ plaintext] ]
  - But twice the work.
- Key2 can be related to Key1 (eg, key2 = key1  $\oplus$  C), but still same work.
- Probably ok:
  - CBC\_encrypt[ plaintext, weak cryptographic checksum (plaintext) ]
  - Probably ok: CBC\_encrypt[ plaintext, hash[ plaintext] ]
- **Offset Codebook Mode (OCB)**

## Hashes and Message Digests (NS chapter 5)

- msg  $\rightarrow$  fixed-length hash  $H(\text{msg})$ 
  - Not 1-1 since msg space is much larger than hash space
  - secure one-way function:
    - computationally hard to find two msgs  $m_1$  and  $m_2$  s.t.  $h(m_1)=h(m_2)$

### Assuming hash is random, how long should it be?

- Consider hash space of  $K$  (ie, hash of  $(\log K)$  bits)
- Consider  $N$  randomly chosen messages,  $m_1, m_2, \dots, m_N$
- $\text{Pr}[ \text{there is a pair of distinct msgs } \langle m_i, m_j \rangle : H(m_i) = H(m_j) ]$ 
  - =  $\text{Pr}[ H(m_1)=H(m_2) \text{ or } H(m_1)=H(m_3) \text{ or } \dots \text{ or } H(m_{N-1})=H(m_N) ]$
  - $\approx \text{Sum \{over distinct } \langle m_i, m_j \rangle \text{ pairs} \} (1/K)$
  - =  $[N(N-1)/2] [1/K]$
- So if  $N = \sqrt{K}$  then Pr is  $1/2$
- $K$  should be large enough so that searching through  $\sqrt{K}$  is hard.
  - So  $K = 2^{128}$  is ok (assuming search through  $2^{64}$  is hard)

## Keyed Hash: Hash with secret key

### Keyed hash equivalent to secret-key encryption

- confidentiality
- authentication
- integrity

### Authentication with keyed hash

- A and B share secret key  $K_{AB}$
- A sends random number  $r_A$  to B.
- B computes  $H(K_{AB} | r_A)$  and sends it back.
- A computes  $H(K_{AB} | r_A)$  (cannot invert it) and check if received value equals it. Match authenticates B to A.
- Similarly, B sends random number  $r_B$  to A and expects  $H(K_{AB} | r_B)$  back.

## MAC (message integrity checksum) with keyed hash

### Obtaining MAC for $msg = (m_1, m_2, \dots, m_n)$ given shared secret key $K_{AB}$

- Obvious approach:  $MAC = H(K_{AB} | msg)$
- Not ok because  $H(m_1, m_2, \dots, m_n)$  is usually  $H(H(m_1, m_2, \dots, m_{n-1}) | m_n)$
- So attacker can add any  $m_{n+1}$  and get its MAC as  $H(old\ MAC, m_{n+1})$ .
- Possible fixes:
  - $MAC = H(msg | K_{AB})$
  - $MAC = \text{half the bits of } H(K_{AB} | msg)$
  - $MAC = H(K_{AB} | msg | K_{AB})$
- HMAC (de facto standard):  $MAC = H(K_{AB} | H(K_{AB} | msg))$  (almost)

## Encryption / encryption + integrity with keyed hash

Encryption of msg =  $(m_1, m_2, \dots, m_n)$

- Generate (can be precomputed) one-time pad:
  - $b_i = H(K_{AB} | b_{i-1})$  where  $b_0$  is IV
  - $c_i = b_i \oplus m_i$
- transmit IV and  $c_1, c_2, \dots, c_n$
- Decryption identical

Encryption and integrity of msg =  $(m_1, m_2, \dots, m_n)$

- Encryption with plaintext mixed into one-time pad
- $b_i = H(K_{AB} | c_{i-1})$  where  $c_0$  is IV
- $c_i = b_i \oplus m_i$
- Decryption straightforward (homework)

## Hash from secret-key encryption/decryption

Hashing a block with secret key encryption

- Hash(block) = Encrypt constant (eg, 0) using block as the key

Unix (original) uses a variation to store passwords

- When user sets password
  - Concatenate 7-bit ASCII of first eight chars to get 56-bit secret key
  - Generate 12-bit random number (called salt)
  - Encrypt the number 0 using the key and a salt-modified DES
    - defends against DES-cracking hardware
    - salt indicates duplicated bits in 32-bit R  $\rightarrow$  48-bit mangler input
  - Store salt and ciphertext
- When user enters password,
  - compare stored ciphertext with that computed from password

## Hashing large messages with secret-key encryption (key size $k$ )

- Obvious extension of above approach:
  - Divide large message into  $k$ -bit chunks  $m_1, m_2, \dots$
  - $C_i$  = encryption of  $C_{i-1}$  with  $m_i$  as key, where  $C_0$  is a constant
  - Let the last  $C_i$  be the hash of message
- Not ok if  $C_i$  is usually too small to be a good hash (eg, 64 bits in DES)
- Sufficient fix is to  $\oplus$  each stage's input with previous stage's output:
  - $C_1$  = encryption of a constant  $C0\_$  with  $M_1$  as key
  - For  $i > 1$ :  $C_i$  = encryption of  $C_{i-2} \oplus C_{i-1}$  with  $M_i$  as key
  - Let the last  $C_i$  be the hash of message
- One way to generate 128 bits of hash with DES:
  - Generate 64-bit hash as above.
  - Generate another 64-bit hash with message blocks in reverse order
  - This approach has a flaw (homework)
- Better way to generate 128 bits of hash with DES:
  - Generate two 64-bit hashes as above but with different constants.

## MD4: 32-bit-word-oriented hash function

- message of arbitrary number of bits  $\rightarrow$  128-bit hash
- Step 1: Pad  $msg$  to multiple of 512 bits  
 $pmsg \leftarrow msg \mid \text{one } 1 \mid p \text{ 0's} \mid (64\text{-bit encoding of } p)$ ;  
where  $[msgsize+1+p+64]$  is a multiple of 512 (note:  $p$  in  $1..512$ )
- Step 2: Process  $pmsg$  in 512-bit chunks to obtain 128-bit hash  $md$   
128-bit  $md$  treated as 4 words:  $d_0, d_1, d_2, d_3$ ;  
512-bit  $pmsg$  chunk treated as 16 words:  $m_0, m_1, \dots, m_{15}$ ;  
Initialize  $\langle d_0 \dots d_3 \rangle$  to  $\langle 01 \mid 23 \mid \dots \mid 89 \mid ab \mid cd \mid ef \mid fe \mid dc \mid \dots \mid 10 \rangle$ ;  
For each 512-bit chunk  $c$  of  $msg$ :  
 $e_0 \dots e_3 \leftarrow d_0 \dots d_3$ ; // store current  $md$  for use later  
// Pass 1: mangle  $d_0 \dots d_3$  using  $m_0 \dots m_{15}$ , mangler H1, permutation J  
For  $i = 0, \dots, 15$ :  $d_{J(i)} := H1(i, d_0, d_1, d_2, d_3, m_i)$ ;  
// Pass 2: mangle  $d_0 \dots d_3$  using  $m_0 \dots m_{15}$ , mangler H2, permutation J  
For  $i = 0, \dots, 15$ :  $d_{J(i)} := H2(i, d_0, d_1, d_2, d_3, m_i)$ ;  
// Pass 3: mangle  $d_0 \dots d_3$  using  $m_0 \dots m_{15}$ , mangler H3, permutation J  
For  $i = 0, \dots, 15$ :  $d_{J(i)} := H3(i, d_0, d_1, d_2, d_3, m_i)$ ;  
 $d_0 \dots d_3 \leftarrow d_0 \dots d_3 \oplus e_0 \dots e_3$ ;  
 $md \leftarrow d_0 \dots d_3$ ;

## More Hash Functions

- **MD2: octet-oriented**
  - Message of arbitrary number of octets → 128-bit digest
  - Like MD4 except
    - Step 1: pad to multiple of 16 octets
    - Step 2: append 16-octet checksum (not cryptographic)
    - Step 3: do 18 passes over msg in 16-octet chunks
- **MD5: 32-bit word oriented**
  - Message of arbitrary number of bits → 128-bit digest
  - Like MD4 except four passes and different mangler functions
- **SHA-1: 32-bit word oriented**
  - Message of arbitrary number of bits upto  $2^{64}$  bits → 160-bit digest
  - Like MD5 except five passes, different mangler functions, and at start of each stage, 512-bit msg chunk → 5 x 512-bit chunk using rotated versions of the msg chunk

## HMAC: defacto MAC standard

- Can use any hash function H (eg, MD2, MD4, SHA-1)
- Variable-sized message and variable-length key
  - fixed-size MAC of same size as output of H
- $paddingKey \leftarrow$  pad key with 0's to 512 bits
  - If key is larger than 512 bits, first hash key and then pad
- $h1 \leftarrow H( msg, paddingKey \oplus [string\ of\ 36_{16}\ octets] )$
- $result \leftarrow H( h1, paddingKey \oplus [string\ of\ 5C_{16}\ octets] )$

# A Bit of Number Theory (NS chapter 7)

## Need some number theory to understand public key cryptology

- Modular addition, multiplication, exponentiation over  $Z_n = \{0, 1, \dots, n-1\}$
- Euclid's algorithm: gcd and multiplicative inverse
- Chinese remainder theorem:  $(x \bmod pq) \iff (x \bmod p) \text{ and } (x \bmod q)$
- $Z_n^* = \{j : j > 0 \text{ and relatively prime to } n\}$
- Euler's totient function  $\phi(n) = |Z_n^*|$
- Euler's theorem
  
- **Conventions**
- All variables are integers (positive, zero, negative)
- unless otherwise stated
- $n$  is positive integer

## Numbers modulo-n

- For any  $x$ :  $(x \bmod n)$  equals  $y$  in  $Z_n$  s.t.  $x = y+k \cdot n$  for some integer  $k$ .
- Nonnegative remainder of  $x/n$ :
  - $3 \bmod 10 = 3$  ( $3 = 3 + 0 \cdot 10$ )
  - $23 \bmod 10 = 3$  ( $23 = 3 + 2 \cdot 10$ )
  - $-27 \bmod 10 = 3$  ( $-27 = 3 + (-3) \cdot 10$ ) (unlike in most prog lang)
- Integers  $u$  and  $v$  are said to be **equal mod-n** if  $(u \bmod n) = (v \bmod n)$ 
  - Math books say "equivalent mod-n", denoted  $u \bmod n \equiv v \bmod n$

---

## Modulo-n addition and additive inverse

- Mod-n addition is ordinary addition followed by *mod-n* operation
  - $(3+7) \bmod 10 = 10 \bmod 10 = 0$
  - $(3-7) \bmod 10 = -4 \bmod 10 = 6$
- Note:  $(u+v) \bmod n = (u \bmod n) + (v \bmod n) \bmod n$
- *Additive inverse mod-n* of  $x$  is  $y$  st  $(x+y) \bmod n = 0$ 
  - denoted  $-x \bmod n$
  - exists for any  $x$  and  $n$
  - easy to compute: eg, for  $x$  in  $Z_n$ , additive inverse is  $n-x$

## Modulo-n multiplication and multiplicative inverse

- Mod-n multiplication is ordinary multiplication followed by *mod-n* operation
  - $(3 \cdot 7) \bmod 10 = 21 \bmod 10 = 1$
  - $(8) \cdot (-7) \bmod 10 = -56 \bmod 10 = 4$
- Note:  $(u \cdot v) \bmod n = (u \bmod n) \cdot (v \bmod n) \bmod n$
- *Multiplicative inverse mod-n* of integer  $x$  is  $y$  s.t.  $(x \cdot y) \bmod n = 1$ 
  - denoted  $x^{-1} \bmod n$
  - $3^{-1} \bmod 10$  is  $7$  ( $3 \cdot 7 = 21 = 1 \bmod 10$ ).
  - $x^{-1}$  exists and is unique iff  $x$  and  $n$  are relatively prime
    - ie,  $\gcd(x, n) = 1$
- Euclid's algorithm: efficiently computes  $\gcd(x, n)$  and  $x^{-1}$  (if it exists)

## Modulo-n exponentiation and exponentiative inverse

- Modulo-n exponentiation is ordinary exponentiation followed by *mod-n*
  - $3^2 \bmod 10 = 9$
  - $3^3 \bmod 10 = 27 \bmod 10 = 7$
  - $(-3)^3 \bmod 10 = -27 \bmod 10 = 3$
- Note:  $(u^v) \bmod n \neq (u^{v \bmod n}) \bmod n$
- *Exponentiative inverse mod-n* of integer  $x$  is  $y$  s.t.  $(x^y \bmod n) = 1$ 
  - $3^4 = 81 = 1 \bmod 10$ , so 4 is the exponentiative inverse mod-10 of 3
  - Exists and is unique iff  $x$  and  $n$  are relatively prime
  - Easy to compute if  $n$  has certain structure.

---

## Primes

- Positive integer  $p$  is prime iff it is exactly divisible only by itself and 1
- Infinitely many primes, but they thin out as numbers get larger
  - 25 primes less than 100
  - $\text{Pr}[\text{random 10-digit number is a prime}] = 1/23$
  - $\text{Pr}[\text{random 100-digit number is a prime}] = 1/230$
  - $\text{Pr}[\text{random } k\text{-digit number is a prime}] = 1/(10 \cdot \ln k)$

## Euclid's algorithm for gcd(x, y)

- $[x, y]$  has same divisors/gcd as  $[x-y, y]$ , as  $[x-k \cdot y, y]$ , as  $[x \bmod y, y]$ , as  $[y, x \bmod y]$ , as  $[y, \text{remainder}(x/y)]$
- repeat  $[x, y] \rightarrow [y, \text{remainder}(x/y)]$  until first entry is 0; second entry is gcd

- store intermediate remainders in array r

$$r = [r_{-2} \ r_{-1} \ r_0 \ \quad \quad \quad r_1 \ \quad \quad \quad r_2 \ \quad \quad \quad \dots]$$

$$x \quad y \quad \text{remainder}(x/y) \quad \text{remainder}(y/r_0) \quad \text{remainder}(r_0/r_1) \quad \dots]$$

## Euclid (x,y) with intermediate remainders

array r = [r<sub>-2</sub> r<sub>-1</sub> r<sub>0</sub> r<sub>1</sub> r<sub>2</sub> ...]

r<sub>-2</sub> ← x; r<sub>-1</sub> ← y;

integer n ← 0;

while r<sub>n-1</sub> ≠ 0 do

    r<sub>n</sub> ← remainder(r<sub>n-2</sub>/r<sub>n-1</sub>);

    n ← n+1;

return r<sub>n-2</sub>; // gcd(x,y)

- To get multiplicative inverse, need to keep track of quotients, differences

## Euclid\_Augmented (x,y)

arrays r, q, u, v;

r<sub>-2</sub> ← x; r<sub>-1</sub> ← y;

u<sub>-2</sub> ← 1; v<sub>-2</sub> ← 0;

u<sub>-1</sub> ← 0; v<sub>-1</sub> ← 1;

integer n := 0;

while r<sub>n-1</sub> ≠ 0 do // invariant r<sub>n</sub> = u<sub>n</sub>·x + v<sub>n</sub>·y

    r<sub>n</sub> ← remainder (r<sub>n-2</sub>/r<sub>n-1</sub>);

    q<sub>n</sub> ← quotient ( r<sub>n-2</sub>/r<sub>n-1</sub> );

    u<sub>n</sub> ← u<sub>n-2</sub> - q<sub>n</sub>·u<sub>n-1</sub>;

    v<sub>n</sub> ← v<sub>n-2</sub> - q<sub>n</sub>·v<sub>n-1</sub>;

    n ← n+1;

// Termination: gcd(x,y) = r<sub>n-2</sub> = u<sub>n-2</sub>·x + v<sub>n-2</sub>·y

return r<sub>n-2</sub>, u<sub>n-2</sub>, v<sub>n-2</sub>;

- If gcd(x,y)= 1 then multiplicative inverse mod-y of x = u<sub>n-2</sub>  
multiplicative inverse mod-x of y = v<sub>n-2</sub>  
else multiplicative inverses do not exist

# Chinese remainder theorem

Let  $z_1, z_2, \dots, z_k$  be relatively prime.

Then the mapping  $Z_{z_1, z_2, \dots, z_k} \rightarrow Z_{z_1} \times Z_{z_2} \times \dots \times Z_{z_k}$  where

$x \rightarrow \langle x \bmod z_1, x \bmod z_2, \dots, x \bmod z_k \rangle$  is 1-1 onto (so invertible).

So for  $\langle x_1, x_2, \dots, x_k \rangle$ : exactly one  $x$  in  $Z_{z_1, z_2, \dots, z_k}$  s.t.  $(x \bmod z_i) = x_i$

- For  $k=2$ ,  $(x \bmod z_1 \cdot z_2) = [x_2 \cdot a \cdot z_1 + x_1 \cdot b \cdot z_2] \bmod z_1 \cdot z_2$ , where  $1 = a \cdot z_1 + b \cdot z_2$
- $z_1=3, z_2=4$  (relatively prime)

$Z_{3 \cdot 4}$	0	1	2	3	4	5	6	7	8	9	10	11
$Z_3 \times Z_4$	$\langle 0,0 \rangle$	$\langle 1,1 \rangle$	$\langle 2,2 \rangle$	$\langle 0,3 \rangle$	$\langle 1,0 \rangle$	$\langle 2,1 \rangle$	$\langle 0,2 \rangle$	$\langle 1,3 \rangle$	$\langle 2,0 \rangle$	$\langle 0,1 \rangle$	$\langle 1,2 \rangle$	$\langle 2,3 \rangle$

- $z_1=2, z_2=4$  (not relatively prime)

$Z_{2 \cdot 4}$	0	1	2	3	4	5	6	7
$Z_2 \times Z_4$	$\langle 0,0 \rangle$	$\langle 1,1 \rangle$	$\langle 0,2 \rangle$	$\langle 1,3 \rangle$	$\langle 0,0 \rangle$	$\langle 1,1 \rangle$	$\langle 0,2 \rangle$	$\langle 1,3 \rangle$

- If  $z_1, z_2$  relatively prime, no number in  $[1 .. z_1 \cdot z_2]$  is multiple of  $z_1$  and  $z_2$

## Proof of Chinese remainder theorem for $k = 2$

- Note  $Z_{z_1 \cdot z_2}$  and  $Z_{z_1} \times Z_{z_2}$  have the same number of elements (namely  $z_1 \cdot z_2$ )
- Will show mapping is 1-1 and obtain inverse.
- For any integer  $x$ , let
  - $(x \bmod z_1) = x_1$  and
  - $(x \bmod z_2) = x_2$

- By Euclid: there exist  $a$  and  $b$  such that  $1 = a \cdot z_1 + b \cdot z_2$

- Multiplying both sides by  $x$  and taking mod  $z_1 \cdot z_2$

$$\begin{aligned}
 (x \bmod z_1 \cdot z_2) &= [x \cdot a \cdot z_1 + x \cdot b \cdot z_2] \bmod z_1 \cdot z_2 \\
 &= [(x_2 + k \cdot z_2) \cdot a \cdot z_1 + (x_1 + j \cdot z_1) \cdot b \cdot z_2] \bmod z_1 \cdot z_2 \\
 &= [x_2 \cdot a \cdot z_1 + x_1 \cdot b \cdot z_2] \bmod z_1 \cdot z_2
 \end{aligned}$$

LHS depends only on  $x_1, x_2, a, b$ .

So for any  $\langle x_1, x_2 \rangle$ , exactly one  $x$  s.t.  $(x \bmod z_1) = x_1$  and  $(x \bmod z_2) = x_2$

- So  $x$  and  $y$  are the same mod  $z_1 \cdot z_2$

## Proof of for $k > 2$ is by induction

- If  $z_1, z_2, \dots, z_k, z_{k+1}$  rel. prime, then  $(z_1 \cdot z_2 \cdots z_k)$  and  $z_{k+1}$  are rel. prime

## $Z_n^*$

$Z_n^* = \{ x : x \text{ is mod-}n \text{ integer relatively prime to } n \}$

- $Z_{10}^* = \{ 1, 3, 7, 9 \}$  whereas  $Z_{10} = \{ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 \}$
- 0 is not an element of  $Z_n^*$  because  $\gcd(0, n) = n$  for any  $n$

**Theorem:**

$Z_n^*$  closed under multiplication mod- $n$ : for  $x, y$  in  $Z_n^*$ ,  $x \cdot y \text{ mod-}n$  in  $Z_n^*$ .  
Also, multiplying elements of  $Z_n^*$  with any  $x$  is a permutation of  $Z_n^*$ .

**Proof**

Let  $a$  and  $b$  be in  $Z_n^*$ . By definition  $\gcd(a, n) = \gcd(b, n) = 1$ .

So there exist  $u_a, v_a, u_b, v_b$  s.t.  $u_a \cdot a + v_a \cdot n = 1$  and  $u_b \cdot b + v_b \cdot n = 1$ .

Multiply the two equations:

$$u_a \cdot u_b \cdot (a \cdot b) + n \cdot (u_a \cdot v_b \cdot a + v_b \cdot u_b \cdot b + u_a \cdot v_b \cdot n) = 1$$

Hence, by Euclid alg,  $a \cdot b$  is relatively prime to  $n$ , and so  $a \cdot b$  is in  $Z_n^*$ .

To show  $x \cdot Z_n^*$  is a permutation of  $Z_n^*$ , show that mapping is 1-1.  
(Work out the details)

## Euler's Totient Function

$\phi(n)$ : number of elements in  $Z_n^*$

- For  $n$  prime:  $\phi(n) = n - 1$
- For  $n = p^a$  where  $p$  is prime and  $a > 0$ :  $\phi(n) = (p-1) \cdot p^{a-1}$
- For  $n = p \cdot q$  where  $p$  and  $q$  are relatively prime:  $\phi(n) = \phi(p) \cdot \phi(q)$
- For  $n = p_1^{a_1} \cdot p_2^{a_2} \cdot \dots \cdot p_k^{a_k}$  where  $p_1, \dots, p_k$  are prime:  
 $\phi(n) = \phi(p_1)^{a_1} \cdot \phi(p_2)^{a_2} \cdot \dots \cdot \phi(p_k)^{a_k}$

**Proof**

**For  $n$  prime:**  $\phi(n) = n - 1$ . Obvious.

**For  $n = p^a$  where  $p$  is prime and  $a > 0$ :**  $\phi(n) = (p-1) \cdot p^{a-1}$

$Z_n = \{0, 1, 2, \dots, p, \dots, 2 \cdot p, \dots, 3 \cdot p, \dots, \dots, (p^{a-1} - 1) \cdot p, \dots, (p^a) - 1\}$ .

Only the multiples of  $p$  can divide  $n$ . There are  $(p^{a-1} - 1)$  of them.

Removing them from the set  $\{1, 2, \dots, n-1\}$  yields  $Z_n^*$

So  $\phi(n) = (n-1) - (p^{a-1} - 1) = (p^a - 1) - (p^{a-1} - 1) = p^a - p^{a-1} = (p-1) \cdot p^{a-1}$

For  $n = p \cdot q$  where  $p$  and  $q$  are relatively prime:  $\phi(n) = \phi(p) \cdot \phi(q)$

Let  $m_p = m \bmod p$  and  $m_q = m \bmod q$ . Abbr “relatively prime to” to rpt.  
First show that  $m$  rpt  $p \cdot q$  iff  $m_p$  rpt  $p$  and  $m_q$  rpt  $q$ .

- Assume  $m$  rpt  $p \cdot q$ . Then there exist  $u$  and  $v$  such that  $u \cdot m + v \cdot p \cdot q = 1$ .

Substituting  $m = m_p + k \cdot p$ , we get  $u \cdot m_p + p \cdot (u \cdot k + v \cdot q) = 1$ , so  $m_p$  rpt  $p$ .

Similarly,  $m_q$  rpt  $q$ .

- Assume  $m_p$  rpt  $p$  and  $m_q$  rpt  $q$ . Then there exist  $u_p, v_p, u_q, v_q$ , such that

$$u_p \cdot m_p + v_p \cdot p = 1 \quad \text{and} \quad u_q \cdot m_q + v_q \cdot q = 1.$$

$$\text{So } u_p \cdot (m - k \cdot p) + v_p \cdot p = 1 \quad \text{for some } k, \text{ or } u_p \cdot m + (v_p - u_p \cdot k) \cdot p = 1$$

Similarly, for some  $j$ ,

$$u_q \cdot m + (v_q - u_q \cdot j) \cdot q = 1$$

Multiplying the two, we get

$$[u_p u_q m + u_p (v_q - u_q j) \cdot q + u_q (v_p - u_p k) \cdot p] \cdot m + (v_p - u_p k) \cdot (v_q - u_q j) \cdot p \cdot q = 1$$

So  $m$  rpt  $n$ .

- So there is a 1-1 correspondence between numbers in  $Z_{p \cdot q}^*$  and  $Z_p^* \times Z_q^*$ .

So  $\phi(n) = \phi(p) \cdot \phi(q)$ .

For  $n = p_1^{a_1} \cdot p_2^{a_2} \cdot \dots \cdot p_k^{a_k}$  where  $p_1, \dots, p_k$  are prime.  
(homework)

**End of proof**

9/30/2008 shankar

crypto slide 47

## Euler's Theorem

For all  $a$  in  $Z_n^*$ :  $a^{\phi(n)} = 1 \bmod n$

**Proof:**

Let  $x$  be the product of all the elements of  $Z_n^*$ .

Because  $Z_n^*$  is closed under multiplication,  $x$  is in  $Z_n^*$  and  $x^{-1}$  exists.

Let  $b_1, b_2, \dots, b_{\phi(n)}$  be the elements of  $Z_n^*$  listed in some order.

Let  $y = (a \cdot b_1) \cdot (a \cdot b_2) \cdot \dots \cdot (a \cdot b_{\phi(n)})$ . So  $y = a^{\phi(n)} \cdot x \bmod n$ .

But  $a \cdot b_1, a \cdot b_2, \dots, a \cdot b_{\phi(n)}$  is also  $Z_n^*$  permuted. So  $y = x \bmod n$ .

Thus  $a^{\phi(n)} \cdot x = x \bmod n$ . Multiplying sides by  $x^{-1}$  yields  $a^{\phi(n)} = 1 \bmod n$ .

**Euler's Theorem Variant:**

For all  $a$  in  $Z_n^*$  and any non-negative integer  $k$ :  $a^{k \cdot \phi(n) + 1} = a \bmod n$

**Proof:**  $a^{k \cdot \phi(n) + 1} = a^{k \cdot \phi(n)} \cdot a = a^{\phi(n) \cdot k} \cdot a = [a^{\phi(n)}]^k \cdot a = 1^k \cdot a = a$

**Question:** Does  $a^{\phi(n)} = 1 \bmod n$  hold for all  $a$  in  $Z_n$  (not just  $Z_n^*$ ) ?

9/30/2008 shankar

crypto slide 48

## Generalization of Euler's Theorem (for $a$ in $Z_n$ and $n=p \cdot q$ )

If  $n=p \cdot q$ , where  $p$  and  $q$  are distinct primes then

$a^{k \cdot \phi(n)+1} = a \pmod{n}$  for all  $a$  in  $Z_n$  and any non-negative integer  $k$ .

Proof: Assume  $a$  not in  $Z_n^*$  (o/w follows from Euler's Theorem Variant).

Also assume  $a$  is not 0 (otherwise result holds trivially).

So  $a$  is a multiple of  $p$  or  $q$  but not both. Suppose  $a$  is a multiple of  $q$ .

Decompose  $(a^{k \cdot \phi(n)+1} \pmod{n})$  into  $\pmod{p}$  and  $\pmod{q}$ , and use CRT.

$$\begin{aligned} a^{k \cdot \phi(n)+1} \pmod{p} &= a^{k \cdot \phi(n)} \cdot a \pmod{p} \\ &= a^{k \cdot \phi(p) \cdot \phi(q)} \cdot a \pmod{p} \quad (\text{because } \phi(n) = \phi(p) \cdot \phi(q)) \\ &= a^{\phi(p) \cdot k \cdot \phi(q)} \cdot a \pmod{p} \\ &= 1^{k \cdot \phi(q)} \cdot a \pmod{p} \quad (a \text{ rpt } p, \text{ so } a^{\phi(p)} = 1 \pmod{p} \text{ by Euler's theorem}) \\ &= a \pmod{p} \end{aligned}$$

Similarly  $a^{k \cdot \phi(n)+1} \pmod{q} = a \pmod{q}$

So by CRT  $a^{k \cdot \phi(n)+1} \pmod{n} = a \pmod{n}$

### Further generalization:

Above is true for any  $n$  that is a product of distinct primes.

## Public Key Algorithms (NS chapter 6)

- Public key algorithm: principal has **public key** and **private key**
- Examples:
  - RSA and ECC: encryption and digital signatures.
  - ElGamal and DSS: digital signatures.
  - Diffie-Hellman: establishment of a shared secret
  - Zero knowledge proof systems: authentication
- Most public key algorithms are based on modulo- $n$  arithmetic.

## Recall some modulo-n arithmetic

- Modulo-n addition:  $(a+b) \bmod n$ 
  - Any  $x$  has a unique additive inverse mod- $n$ .
  - Easily computed.
- Modulo-n multiplication:  $(a \cdot b) \bmod n$ 
  - Any  $x$  has a unique multiplicative inverse mod- $n$  iff  $\gcd(x,n)=1$
  - Existence and value easily computed (Euclid's alg)
- $Z_n = \{0, 1, \dots, n-1\}$
- $Z_n^* = \{\text{numbers in } Z_n \text{ that are relatively prime to } n\}$
- $\phi(n) = \text{number of elements in } Z_n^*$  ; easy to get given prime factorization
- Modulo-n exponentiation:  $(a^b) \bmod n$ 
  - Any  $x$  has a unique exponentiative inverse mod- $n$  iff  $\gcd(x,n)=1$ .
  - Easy to compute?
  - For all  $x$  in  $Z_n^*$ :  $x^{\phi(n)} = 1 \bmod n$ . (Euler's Theorem)
  - For all  $x$  in  $Z_n^*$  and non-negative  $k$ :  $x^{k\phi(n)+1} = x \bmod n$ . (Variant)
  - For all  $x$  in  $Z_n$  and non-negative integer  $k$ :  $x^{k\phi(n)+1} = x \bmod n$ 
    - if  $n=p \cdot q$  where  $p$  and  $q$  are distinct primes. (Generalization)

## RSA (Rivest, Shamir, Adleman)

- Key size variable (longer for better security, usually 512 bits, 100 digits).
- Plaintext block size variable but smaller than key length.
- Ciphertext block of key length.
- RSA is much slower to compute than secret key algorithms (e.g., DES)
  - So not used for data encryption

# RSA Algorithm

- Generation of public key and corresponding private key
  - Choose two large primes,  $p$  and  $q$  ( $p$  and  $q$  remain secret).
  - Let  $n = p \cdot q$ .
  - Choose a number  $e$  relatively prime to  $\phi(n) = (p-1) \cdot (q-1)$
  - Public key =  $\langle e, n \rangle$
  - Find multiplicative inverse  $d$  of  $e \pmod{\phi(n)}$  [i.e.,  $e \cdot d = 1 \pmod{\phi(n)}$ ]
  - Private key =  $\langle d, n \rangle$
- Encryption/decryption
  - To encrypt message  $m$  using public key:
    - ciphertext  $c = m^e \pmod{n}$
  - To decrypt ciphertext  $c$  using private key:
    - plaintext  $m = c^d \pmod{n}$
- Signing/Verifying signature
  - To sign a message  $m$  using private key:
    - signature  $s = m^d \pmod{n}$
  - To verify signature  $c$  using public key:
    - plaintext  $m = s^e \pmod{n}$

## Why does the decryption operation work, ie, why is $m^{e \cdot d} = m$

$$\begin{aligned} m^{e \cdot d} &= m^{1 \pmod{\phi(n)}} && \text{[ because } e \cdot d = 1 \pmod{\phi(n)} \text{ ]} \\ &= m^{1 + k \cdot \phi(n)} && \text{[ definition of mod ]} \\ &= m && \text{[Euler's theorem generalization, applicable because} \\ &&& \text{- } m \text{ in } \mathbb{Z}_n \text{ (in RSA)} \\ &&& \text{- } n \text{ is product of distinct primes } p \text{ and } q \text{ ]} \end{aligned}$$

## Why is RSA secure

- Only known way to obtain  $m$  from  $m^e$  is by  $m^{e \cdot d}$  where  $d = e^{-1} \pmod{\phi(n)}$
- Only known way to obtain  $\phi(n)$  is with  $p$  and  $q$
- Factoring a large number is hard, so hard to obtain  $p$  and  $q$  given  $n$

## Efficient modulo exponentiation

- Need to get  $m^e \bmod n$ , for 512-bit (100-digit) numbers  $m$ ,  $e$ ,  $n$
- Consider a small example:  $123^{54} \bmod 678$
- **Naive way:** Multiply  $m$  with itself  $e$  times and then take  $\bmod n$ .
  - $e$  multiplications of increasingly larger numbers ( $m^2, m^3, \dots$ ). Too expensive.
  - $123^{54}$  is approx 100 digits ( $54 \cdot \log_{10} 123$ )
- **Better way:** Multiply  $m$  with itself and take  $\bmod n$ ; repeat  $e$  times.
  - $e$  multiplications of large (100-digit) numbers, and  $e$  divisions.
  - Still expensive.
- **Much better:** Exploit  $m^{2x} = m^x \cdot m^x$  and  $m^{2x+1} = m^{2x} \cdot m$ .
  - Log  $e$  multiplications.

9/30/2008 shankar

crypto slide 55

### ModuloExponentiation( $m, e, n$ )

```
( $x_0, x_1, \dots, x_k$ )  $\leftarrow$   $e$  in binary;           //  $x_0 = 1$ 
initially  $y \leftarrow m$ ;  $j \leftarrow 0$ ;           //  $y = m^{x_0}$ 
while  $j < k$  do           // loop invariant:  $y = m^{(x_0, \dots, x_j)}$  mod- $n$ 
     $y \leftarrow y \cdot y$  mod- $n$ ;           //  $y = m^{(x_0, \dots, x_j, 0)}$  mod- $n$ 
    if  $x_{j+1} = 1$  then  $y \leftarrow y \cdot m$  mod- $n$ ; //  $y = m^{(x_0, \dots, x_j, 1)}$  mod- $n$ 
     $j \leftarrow j + 1$ ;           //  $y = m^{(x_0, \dots, x_j)}$  mod- $n$ 
//  $y = m^e$  mod- $n$ 
```

- **Example:**  $123^{54} \bmod 678$ .  $54 = (1101110)_2$ 
  - $123^{(1)} \bmod 678 = 123$
  - $123^{(10)} \bmod 678 = 123 \cdot 123 \bmod 678 = 15129 \bmod 678 = 213$
  - $123^{(11)} \bmod 678 = 213 \cdot 123 \bmod 678 = 26199 \bmod 678 = 435$
  - $123^{(110)} \bmod 678 = 435 \cdot 435 \bmod 678 = 1889225 \bmod 678 = 63$
  - $123^{(1100)} \bmod 678 = 63 \cdot 63 \bmod 678 = 3969 \bmod 678 = 579$
  - $123^{(1101)} \bmod 678 = 579 \cdot 123 \bmod 678 = 71217 \bmod 678 = 27$
  - $123^{(11010)} \bmod 678 = 27 \cdot 27 \bmod 678 = 729 \bmod 678 = 51$
  - $123^{(11011)} \bmod 678 = 51 \cdot 123 \bmod 678 = 6273 \bmod 678 = 171$
  - $123^{(110110)} \bmod 678 = 171 \cdot 171 \bmod 678 = 29241 \bmod 678 = 87$

9/30/2008 shankar

crypto slide 56

## Generating RSA Keys consists of two parts:

- find big primes  $p$  and  $q$
- finding  $e$  relatively prime to  $\phi(n)$  ( $= (p-1) \cdot (q-1)$ )
  - $d = e^{-1} \text{ mod-}\phi(n)$

## Finding big primes $p$ and $q$ (100-digit numbers)

- Choose random  $n$  and test for prime. If not prime, retry.  
(recall that  $\text{Pr}(100\text{-digit number is prime}) = 1/230$ )
- Testing  $n$  for prime:
  - No practical deterministic way (eg, dividing  $n$  by every  $j < \sqrt{n}$ )
  - Practical probabilistic ways (ie,  $n$  is prime with high prob)
- Probabilistic test 1:  
Generate random  $n$  and  $a$  in  $1..n$ ;  
Treat  $n$  as prime if  $a^{n-1} = 1 \text{ mod-}n$ ;
  - $\text{Prob}[\text{test fails}]$  is low ( $\sim 10^{-13}$  for 100-digit  $n$ ).  
Note: converse holds from Euler's theorem
  - Can make the test stronger by trying several different  $a$ .
  - But *Carmichael numbers*: 561, 1105, 1729, 2465, 2821, 6601, ...
- Probabilistic test 2 (Miller-Rabin): works even for Carmichael numbers.

## Finding $e$ (approach 1):

- Choose  $p$  and  $q$  as described above
- Choose  $e$  at random until it is relatively prime to  $\phi(n)$

## Finding $e$ (approach 2):

- Fix  $e$  such that  $m^e$  easy to compute (i.e., few 1's in binary)
- Choose primes  $p$  and  $q$  such that  $e$  relatively prime to  $(p-1) \cdot (q-1)$
- **One choice:  $e=3 = (11)_2$**  [so  $m^e$  needs 2 multiplications]
  - Need to pad small  $m$ .
    - If  $m < n^{1/3}$  then  $m^e \text{ mod-}n = m^3$ , so attacker can get  $m$  by  $(m^e)^{1/3}$
  - Need to use different pads if  $m$  is sent to 3 principals with public keys  $(3, n_1), (3, n_2), (3, n_3)$ .
    - Attacker has  $m^3 \text{ mod-}n_1, m^3 \text{ mod-}n_2, m^3 \text{ mod-}n_3$
    - CRT yields  $m^3 \text{ mod-}n_1 \cdot n_2 \cdot n_3$
    - Because  $m < n_1, m < n_2, m < n_3$ ,  
attacker has  $m^3 < n_1 \cdot n_2 \cdot n_3$  and so  $(m^3 \text{ mod-}n_1 \cdot n_2 \cdot n_3)^{1/3}$  yields  $m$ .
- **Another choice:  $e = 2^{16} + 1 = 65537$**  [so  $m^e$  requires 17 multiplications]
  - No need for pad since unlikely that  $m^{65537} < n$ .
  - No need for random pad when  $m$  sent more than once since unlikely that  $m$  would be sent to 65537 different recipients.

## Public Key Cryptography Standard (PKCS)

- Standard encoding of information to be signed/encrypted in RSA
- Takes care of
  - encrypting guessable messages
  - signing smooth numbers
  - multiple encryptions of same message with  $e=3$
  - ...

### Encryption (fields are octets)

- msb 

0	2	at least eight random non-zero octets	0	data	lsb
---	---	---------------------------------------	---	------	-----

- Note that the data is usually small (DES/3DES/AES key, hash, etc)

### Signing (fields are octets)

- msb 

0	1	at least eight octets of 9FF <sub>16</sub>	0	ASN.1 encoded digest type and digest	lsb
---	---	--	---	--------------------------------------	-----

## Diffie-Helman (Basic)

- Allows any two principals that do not have already have a shared secret to establish a shared secret over an open channel.
- Initially A and B share: (large) prime  $p$  and  $g < p$  (publicly known).
  - A chooses random 512-bit number  $S_A$ , sends  $T_A = g^{S_A} \text{ mod-} p$  to B.
  - B chooses random 512-bit number  $S_B$ , sends  $T_B = g^{S_B} \text{ mod-} p$  to A.
  - A computes  $T_B^{S_A} \text{ mod-} p [ = g^{S_B \cdot S_A} \text{ mod-} p = g^{S_A \cdot S_B} \text{ mod-} p ]$ .
  - B computes  $T_A^{S_B} \text{ mod-} p [ = \text{equals } g^{S_A \cdot S_B} \text{ mod-} p ]$ .
  - A and B now share  $g^{S_A \cdot S_B} \text{ mod-} p$ , which can serve as a key.
  - Attacker knowing  $T_A$  and  $T_B$  and  $p$  and  $g$  cannot obtain  $g^{S_A \cdot S_B} \text{ mod-} p$ , because logarithm modulo- $n$  is hard.
- Does not provide authentication:
  - A does not know whether it is talking to B or C.

A sends [sender id A, $g^{S_A} \text{ mod-} p$ ]	
	C sends [sender id B, $g^{S_C} \text{ mod-} p$ ]
A and C share secret $g^{S_A \cdot S_C} \text{ mod-} p$ , but A thinks it is talking to B	

## Diffie-Helman with Published Numbers

- Assume PKI (public key infrastructure) that publishes for every principal X:  $(X, g, p, g^{SX} \text{ mod-}p)$
- Then A can encrypt info with  $(g^{SA \cdot SB} \text{ mod-}p)$  and only B can decrypt it.
- Note that initial handshake is not needed either.

## Authenticated Diffie-Helman

- If A and B know a secret (eg, shared secret key, public key), there are various ways for A and B to authenticate each other:
  - Encrypt Diffie-Helman exchange with pre-shared secret.
  - Encrypt Diffie-Helman exchange with other's public key.
  - Sign Diffie-Helman value with your private key.
  - Following Diffie-Helman exchange, transmit hash of shared Diffie-Helman value, sender name, and pre-shared secret.
  - Following Diffie-Helman exchange, transmit hash of initially transmitted Diffie-Helman value and pre-shared secret.
- But if A and B have pre-shared secret, why resort to Diffie-Helman?
  - Perfect-forward secrecy

## Man-in-the-middle attack possible even if A and B share passwords

Let  $pw_{AB}$  be A's password to B, and  $pw_{BA}$  be B's password to A

(below  $g^x \text{ mod-p}$  abbreviated to  $g^x$ )

A	C	B
send $[A, g^{SA}]$ to B	alter msg to $[A, g^{SC}]$	
	alter msg to $[B, g^{SC}]$	send $[B, g^{SB}]$ to A
<--- A and C share $g^{SC \cdot SA}$ --->		<--- C and B share $g^{SC \cdot SB}$ --->
send $[g^{SC \cdot SA} \{ pw_{AB} \}]$	decrypt with $g^{SC \cdot SA}$ , alter to $[g^{SC \cdot SB} \{ pw_{AB} \}]$	
		decrypt using $g^{SC \cdot SB}$ A authenticated (error)

## Zero-knowledge proof systems

- Allows you to prove that you know a secret without revealing it.
  - RSA is an example (secret is private key)

### Classic example is based on graph isomorphism

- “Key” generation
  - A chooses a large graph (eg, 500 vertices)  $G_{A1}$ .
  - A renames the vertices to produce an isomorphic graph  $G_{A2}$ .
  - Graphs  $G_{A1}$  and  $G_{A2}$  are A's “public key”.
  - The vertex renaming transforming  $G_{A1}$  to  $G_{A2}$  is A's “private key”.
- A authenticates to B as follows:
  - A sends B a new set of graphs  $\{G_1, \dots, G_k\}$ , each isomorphic to  $G_{A1}$ .
  - B randomly divides the graphs into subset 1 and subset 2.
  - B challenges A to provide vertex-renamings establishing that
    - every graph in subset 1 is isomorphic to  $G_{A1}$
    - every graph in subset 2 is isomorphic to  $G_{A2}$
  - A supplies the vertex-renamings, thereby authenticating itself.

- Why does it work?
  - Graph isomorphism is a hard problem: knowing a renaming to  $G_{A1}$  does not help obtain a renaming to  $G_{A2}$ .
  - So renamings could only have been generated by A originally.
  - Unlikely that they were generated by C (having eavesdropped on many previous authentications of A), because the choice of the subsets 1 and 2 is random.

## Fiat-Shamir variant

- Key generation
  - A's private key: a large random number  $s$
  - A's public key:  $(n, v)$ ,
    - $n$  is product of two large primes (as in RSA)
    - $v$  is  $s^2 \pmod{n}$  (so only A knows square root mod- $n$  of  $v$ )
- Authentication
  - A chooses  $k$  random numbers,  $r_1, \dots, r_k$
  - A sends  $r_1^2 \pmod{n}, \dots, r_k^2 \pmod{n}$ , to B
  - B randomly splits these into subset 1 and subset 2, and informs A
  - A sends
    - $s \cdot r_i \pmod{n}$  for each  $r_i^2 \pmod{n}$  in subset 1
    - $r_i \pmod{n}$  for each  $r_i^2 \pmod{n}$  in subset 2
  - B checks whether
    - for each entry in subset 1:  $(\text{reply}_i)^2 = v \cdot r_i^2 \pmod{n}$
    - for each entry in subset 2:  $(\text{reply}_i)^2 = r_i^2 \pmod{n}$
    - If so, A is authenticated

- Why does it work?
  - Finding square root mod-n is at least as hard as factoring.
    - Knowing  $s \cdot r_i \text{ mod-n}$  does not help obtain  $r_i \text{ mod-n}$ , and vice versa.
  - So replies could only have been generated by A originally.
  - Unlikely that they were generated by C (having eavesdropped on many previous authentications of A), because the choice of the subsets 1 and 2 is random.

## Zero-knowledge signatures

- A zero-knowledge system can be transformed to a public key signature, but performance is poor.
- Note that authentication is interactive but signature is not.
- Trick: use a hash to provide a “random” choice of subset 1 and subset 2.
  - Suppose hash function chosen provides k-bit hash (e.g., k=128).
  - A chooses k random numbers,  $r_1, \dots, r_k$
  - A forms msg [data to be signed |  $r_1^2 \text{ mod-n}, \dots, r_k^2 \text{ mod-n}$ ]
  - A obtains hash of msg, and provides a reply vector in which the 1's in the hash correspond to subset 1 and the 0's correspond to subset 2:
    - if hash bit i is 1 then the reply vector has  $s \cdot r_i \text{ mod-n}$  in position i
    - if hash bit i is 0 then the reply vector has  $r_i^2 \text{ mod-n}$  in position i
  - Why does it work?
    - Forging a signature on a message requires having both possible replies for all the  $r_i$ 's.