Computer and Network Security
CMSC 414
CRYPTO

Udaya Shankar
shankar@cs.umd.edu

Introduction to Cryptology (NS chapter 2)

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

- Key is secret. Encryption/decryption algorithms not secret.
- Given plaintext and cyphertext, 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 (Diffie-Helman), …
  - Hash functions (of cryptographic kind): MD5, SHA-1, …

Secret-key (symmetric) 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.
- Provides integrity over insecure channel:
  - A and B share secret key K
  - A sends plaintext and **fixed-length** part of K(plaintext) to B,eg, last 128 bits
  - Called MAC (msg authentication code) or MIC (msg integrity code))
  - B receives plaintext, computes its MAC and checks against received MAC
  - This particular protocol provides attacker with plaintext-ciphertext pairs

Hashing (of cryptographic kind)

- Hash function H(.) transforms plaintext msg of arbitrary length to fixed-length hash H(msg)
  - Easy to compute H(msg) from msg
  - Not easy to find msg1 and msg2 such that H(msg1) = H(msg2)
- **Keyed hash**: Hash msg along with a shared secret S, e.g., H(msg|S)
- Keyed hashing provides all the capabilites of secret-key crypto.
- Integrity:
  - Send msg and H(msg|S) as MAC.
- Confidentiality:
  - Generate sequence C_0, C_1, C_2, …, where C_0 is random and $C_{i+1} = H(C_i|S)$; to encrypt an arbitrary-length message, XOR it with the sequence.
  - So to send message = [M_0, M_1, M_2, …], send [C_0, $M_1 \oplus C_1$, $M_2 \oplus C_2$, …]

Public key (asymmetric) crypto

- Each principal has two related keys:
  - private key (not shared)
  - public key (shared with world).
  - Plaintext encrypted with one can only be decrypted with the other.
- Confidentiality:
  - B transmits $pubkey_A$(plaintext). A decrypts using $privkey_A$.
- Integrity and digital signature (non-repudiation)
  - A transmits $privkey_A$(plaintext)
  - Anyone with $pubkey_A$ can decrypt it
    and be assured that it could only have been sent by A.
- But public-key crypto is <u>orders</u> slower than secret-key crypto/hashing, so it is used in conjunction with the latter.
- To sign a message: sign the hash of the message.
- To encrypt or integrity-protect a message:
  - First use public-key crypto to establish a per-sesssion secret;
    eg, B creates per-session key K and sends $pubkey_A$(K) to A
  - Then use secret-key crypto or keyed-hashing.

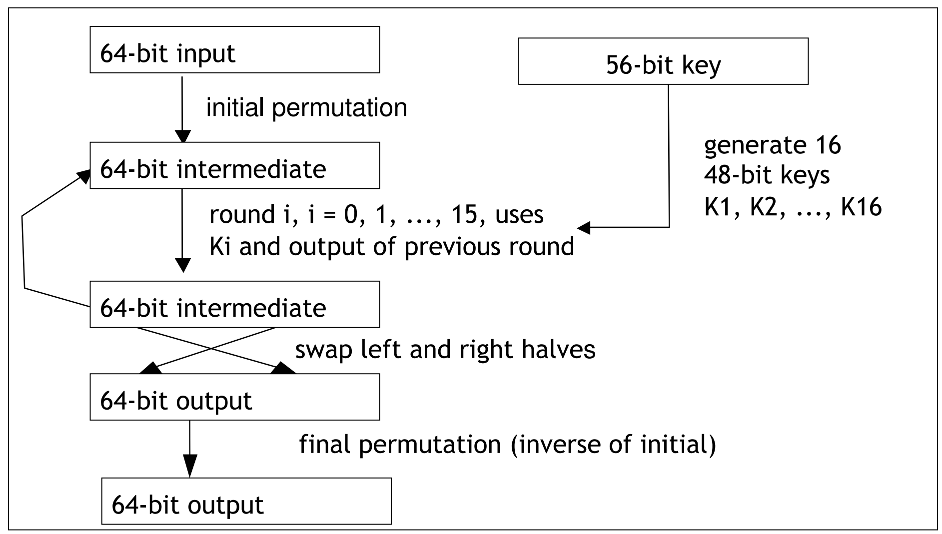
Secret Key Crypto (NS chapter 3)

- Consider **fixed-length** message of k bits here.
  - Variable-length message addressed later.
- Fixed-length message and Fixed-length key → 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(msg1) and S(msg2) differ in many bits (approx k/2 bits).
  - So S cannot be a "simple" function; so following are no good:
    - S(msg) = msg ⊕ key
    - S(msg) = msg bits in reverse order

Secret Key Crypto (contd.)

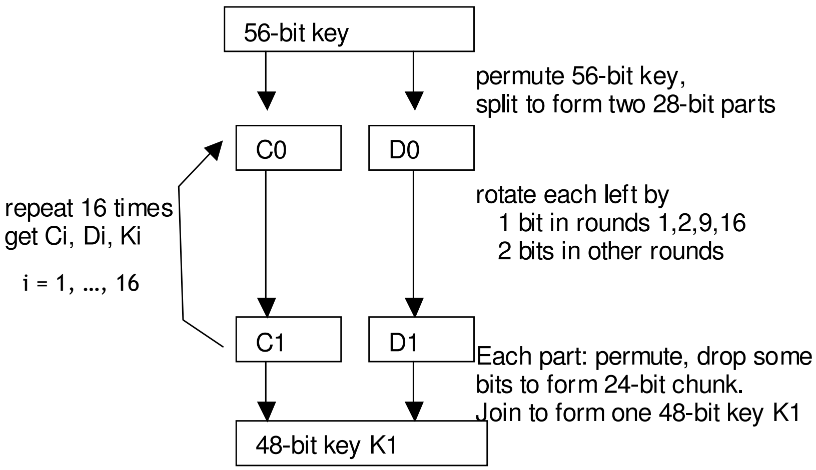
- **Conceptually simple secret-key algorithm S**
  - "Substitution" table: random permutation of all N-bit strings.
  - S(i) is ith row of table
  - Table obtained with physical-world randomness (eg, coin toss).
  - Pro: S is perfectly random
  - Con: need to store table of size $k.2^k$. Impractical for k=64
- **Goal:** Deterministic algorithm that produces "random looking" output.
  Want each output bit to be "influenced" by all input bits.
- **Basic approach**: mix permutations and substitutions
  - Divide k-bit block into p-bit blocks for reasonably small p (eg, p=8).
  - Use p x 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.

DES



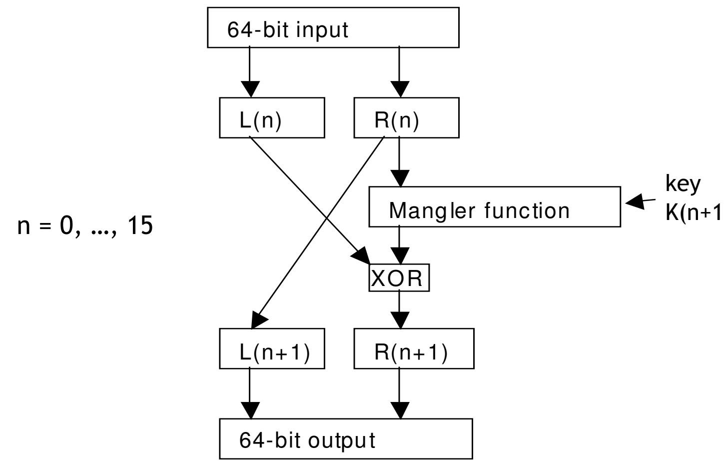
Final permutation is inverse of initial permutation.
Not of security value (why?, what does this mean?)

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



•

DES encryption round



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

DES: decryption = encryption with Ki's in reverse order

DES_encryption {

a_1: $L_0|R_0 \leftarrow$ iperm(dblk);
a_2: for n= 0, …, 15 do
a_3: $L_{n+1} \leftarrow R_n$;
a_4: $R_{n+1} \leftarrow \text{Mangler}_n(R_n,K_{n+1}) \oplus L_n$;
//Yields $L_{16}|R_{16}$

a_5: $L_{17}|R_{17} \leftarrow R_{16}|L_{16}$;
a_6: crblk ← ipermInv($R_{16}|L_{16}$);
}

// key order: K_1, ..., K_{16}

DES_decryption {

b_1: $R_{16}|L_{16} \leftarrow$ iperm(cblk); //a_6 bkw
b_2: for n = 15, ⋯, 0 do //a_2 bkw
b_3: $R_n \leftarrow L_{n+1}$; // a_3 bkw
b_4: $L_n \leftarrow \text{Mnglr}_n(R_n,K_n) \oplus R_{n+1}$; //$a_4$ bkw
// sets L_n to X such that
// $R_{n+1} \leftarrow \text{Mangler}_n (R_n, K_n) \oplus X$
// Yields $R_0|L_0$
b_5: $L_0|R_0 \leftarrow$ swap($R_0|L_0$); // a_5 bkw
b_6: dblk ← ipermInv($L_0|R_0$); // a_1 bkw
}

// key order K_{16}, …, K_1

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 $K_1, ..., K_{16}$ are all equal to each other.
So the key sequence $K_1, ..., K_{16}$ (used in encryption) is the same as the key sequence $K_{16}, ..., K_1$ (used in decryption).
So encryption and decryption are the same, i.e., $E_x(b) = D_x(b)$.
So $E_x(E_x(b)) = b$.

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 $\langle K_1(x), ..., K_{16}(x)\rangle$ be the per-round keys obtained from x.
Show that there is another semi-weak key y such that y
$\langle K_1(x), ..., K_{16}(x)\rangle = \langle K_{16}(y), ..., K_1(y)\rangle$.
Hence for any block b: $E_x(\text{block}) = D_y(\text{block})$

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 $\langle m_1, c_1\rangle, \langle m_2, c_2\rangle$, ..., there is an attack that requires 2^{56} storage.
  - Table A with 2^{56} entries <key K_i, $E(K_i, m_1)$>, sorted by column 2.
  - Table B with 2^{56} entries <key K_i, $D(K_i, c_1)$>, sorted by column 2.
  - Do join of Table A and Table B.
  - Each match provides candidate $\langle K_A, K_B\rangle$ for <key1, key2>.
  - Use $\langle m_2, c_2\rangle$, etc. to weed out false candidates.

Current standard encryption algorithm: AES
- different sizes of keys (64, 128, …)
- different data block sizes (…, 64, 128, …)

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

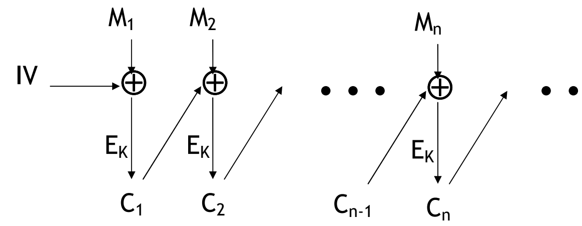
- Pad message to multiple number of blocks: **msg = (M1, M2, …,)**
- Use block encryption repeatedly to get ciphertext = **(C1, C2, …,)**
  - Same Mi's get encrypted to different Ci's
  - Repeated encryptions of same msg result in different ciphertexts.
  - Ciphertext cannot be changed to cause predictable 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

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 $C_1, ..., C_n$
- **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

Output Feedback Mode (OFB)

64-bit OFB

- Generate stream cipher $B_0, B_1, ...,$ 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 $B_{i-1} \mid Y_i$

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 $B_{i-1} \mid C_i$

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.
  So can a computer that checks 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 messsage

- 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 (⊕ 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 Key2 (eg, key1 = key2 ⊕ 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(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 , …, m_N
- Pr[there is a pair of distinct msgs < m_i, m_j > : $H(m_i) = H(m_j)$]
  - = Pr[$H(m_1)=H(m_2)$ or $H(m_1)=H(m_3)$ or … or $H(m_{N-1})=H(m_N)$]
  - ≈ Sum {over distinct < m_i, m_j > 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 = $(m_1, m_2, \ldots, m_n)$ given shared secret key K_{AB}

- Obvious approach: MAC = $H(K_{AB} | msg)$
- Not ok because $H(m_1, m_2, \ldots, m_n)$ is usually $H(H(m_1, m_2, \ldots, m_{n-1})\ m_n)$
- 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 = 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 = $(m_1, m_2, \ldots, m_n)$
- 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 $c_1, c_2, \ldots, c_n$
- Decryption identical

Encryption and integrity of msg = $(m_1, m_2, \ldots, m_n)$
- 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 → 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, ...
  - 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 ⊕ 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 → 128-bit hash**
- **Step 1:** Pad *msg* to multiple of 512 bits
  pmsg ← *msg* | one 1 | p 0's | (64-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, …, m_{15};
  Initialize <$d_0 \ldots d_3$> to <01|23|...|89|ab|cd|ef|fe|dc|...10>;
  For each 512-bit chunk c of msg:
  $e_0 \ldots e_3 \leftarrow d_0 \ldots d_3$; // store current md for use later
  // Pass 1: mangle $d_0 \ldots d_3$ using $m_0 \ldots m_{15}$, mangler H1, permutation J
  For i = 0, ..., 15: $d_{J(i)} \leftarrow H1(i, d_0, d_1, d_2, d_3, m_i)$;
  // Pass 2: mangle $d_0 \ldots d_3$ using $m_0 \ldots m_{15}$, mangler H2, permutation J
  For i = 0, ..., 15: $d_{J(i)} \leftarrow H2(i, d_0, d_1, d_2, d_3, m_i)$;
  // Pass 3: mangle $d_0 \ldots d_3$ using $m_0 \ldots m_{15}$, mangler H3, permutation J
  For i = 0, ..., 15: $d_{J(i)} \leftarrow H3(i, d_0, d_1, d_2, d_3, m_i)$;
  $d_0 \ldots d_3 \leftarrow d_0 \ldots d_3 \oplus e_0 \ldots e_3$;
  md ← $d_0 \ldots d_3$;

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
- *paddedKey* ← pad key with 0's to 512 bits
  If key is larger than 512 bits, first hash key and then pad
- *h1* ←H(msg, *paddedKey* ⊕ [string of 36_{16} octets])
- *result* ← H(*h1*, *paddedKey* ⊕ [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 $\mathbf{Z}_n$ = { 0, 1, ..., n−1}
- Euclid's algorithm: gcd and multiplicative inverse
- Chinese remainder theorem: (x mod pq) <=> (x mod p) and (x mod q)
- $\mathbf{Z}_n^*$ = { j : j > 0 and relatively prime to n}
- Euler's totient function $\phi(n) = |\mathbf{Z}_n^*|$
- 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 mod n) equals y in Z_n s.t. $x = y+k \cdot n$ for some integer k.
- Nonnegative remainder of x/n:
  - 3 mod 10 = 3 (3 = 3 + 0·10)
  - 23 mod 10 = 3 (23 = 3 + 2·10)
  - −27 mod 10 = 3 (−27 = 3 + (−3)·10) (unlike in most prog lang)
- Integers u and v are said to be **equal mod-n** if (u mod n) = (v mod n)
  - Math books say "equivalent mod-n", denoted u mod n ≡ v mod n

---

Modulo-n addition and additive inverse

- Mod-n addition is ordinary addition followed by *mod-n* operation
  - (3+7) mod 10 = 10 mod 10 = 0
  - (3−7) mod 10 = −4 mod 10 = 6
- Note: (u+v) mod n = (u mod n)+(v mod n)) mod n
- *Additive inverse mod-n* of x is y st (x+y) mod n = 0
  - denoted −x mod 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
  - (3·7) mod 10 = 21 mod 10 = 1
  - (8)·(−7) mod 10 = −56 mod 10 = 4
- Note: (u·v) mod n = (u mod n)·(v mod n)) mod n
- *Multiplicative inverse mod-n* of integer x is y s.t. (x·y) mod n = 1
  - denoted x^{-1} mod n
  - 3^{-1} mod-10 is 7 (3·7 = 21 = 1 mod 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 mod 10 = 9
  - 3^3 mod 10 = 27 mod 10 = 7
  - $(-3)^3$ mod 10 = −27 mod 10 = 3
- Note: (u^v) mod n ≠ $(u^{v \bmod n})$ mod n
- *Exponentiative inverse mod-n* of integer x is y s.t. (x^y mod n) = 1
  - 3^4 = 81 = 1 mod 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
  - Pr[random 10-digit number is a prime] = 1/23
  - Pr[random 100-digit number is a prime] = 1/230
  - Pr[random k-digit number is a prime] = /(10·ln k)

Euclid's algorithm for gcd(x, y)

- [x, y] has same divisors/gcd as [x−y, y], as [x−k·y, y], as [x mod-y, y], as [y, x mod-y], as [y, remainder(x/y)]
- repeat [x, y] → [y, 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 r_1 r_2 …]

  x y remainder(x/y) remainder(y/r_0) remainder(r_0/r_1) …]

Euclid (x,y) with intermediate remainders

```
     array r = [r-2  r-1  r0  r1  r2 ...]
     r-2 ← x;  r-1 ← y;
     integer n ← 0;
     while r(n-1) ≠ 0 do
          r(n) ← remainder(r(n-2)/r(n-1));
          n ← n+1;
return r(n-2);       // gcd(x,y)
```

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

Euclid_Augmented (x,y)

```
   arrays r, q, u, v;
   r-2 ← x;  r-1 ← y;
   u-2 ← 1; v-2 ← 0;
   u-1 ← 0; v-1 ← 1;
   integer n := 0;
   while r(n-1) ≠ 0 do     // invariant  r(n) = u(n)·x + v(n)·y
        r(n) ← remainder (r(n-2)/r(n-1));
        q(n) ← quotient ( r(n-2)/r(n-1) );
        u(n) ← u(n-2) − q(n)·u(n-1);
        v(n) ← v(n-2) − q(n)·v(n-1);
        n ← n+1;
// Termination: gcd(x,y) = r(n-2) = u(n-2)·x + v(n-2)·y
return r(n-2) , u(n-2) , v(n-2) ;
```

r =	[r_{-2}	r_{-1}	r_0	r_1	r_2	…]	(remainders)
q =	[		q_0	q_1	q_2	…]	(quotients)
u =	[u_{-2}	u_{-1}	u_0	u_1	u_2	…]	(differences)
v =	[v_{-2}	v_{-1}	v_0	v_1	v_2	…]	(differences)

- If gcd(x,y)= 1 then multiplicative inverse mod-y of x = u_{n-2}

  multiplicative inverse mod-x of y = v_{n-2}

  else multiplicative inverses do not exist

Chinese remainder theorem

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

Then the mapping $Z_{z1,z2,\ldots,zk} \rightarrow Z_{z1} \times Z_{z2} \times \ldots \times Z_{zk}$ where

$x \rightarrow <x \bmod z_1, x \bmod z_2, \ldots, x \bmod z_k>$ is 1–1 onto (so invertible).

So for $<x_1, x_2, \ldots, x_k>$: exactly one x in $Z_{z1,z2,\ldots,zk}$ 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)

$Z_{3\cdot4}$	0	1	2	3	4	5	6	7	8	9	10	11
$Z_3 \times Z_4$	⟨0,0⟩	⟨1,1⟩	⟨2,2⟩	⟨0,3⟩	⟨1,0⟩	⟨2,1⟩	⟨0,2⟩	⟨1,3⟩	⟨2,0⟩	⟨0,1⟩	⟨1,2⟩	⟨2,3⟩

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

$Z_{2\cdot4}$	0	1	2	3	4	5	6	7
$Z_2 \times Z_4$	⟨0,0⟩	⟨1,1⟩	⟨0,2⟩	⟨1,3⟩	⟨0,0⟩	⟨1,1⟩	⟨0,2⟩	⟨1,3⟩

- 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 $Z_{z1\cdot z2}$ and $Z_{z1} \times Z_{z2}$ 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 $z_1 \cdot z_2$

  $(x \bmod z_1 \cdot z_2) = [x \cdot a \cdot z_1 + x \cdot b \cdot z_2] \bmod z_1 \cdot z_2$

  $= [(x_2 + k.z_2) \cdot a \cdot z_1 + (x_1 + j.z_1) \cdot b \cdot z_2)] \bmod z_1 \cdot z_2$

  $= [x_2 \cdot a \cdot z_1 + x_1 \cdot b \cdot z_2] \bmod z_1 \cdot z_2$

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

  So for any $<x_1, x_2>$, exactly one x s.t. $(x \bmod z_1) = x_1$ and $(x \bmod z_1) = x_2$
- So x and y are the same mod $z_1 \cdot z_2$

Proof of for k > 2 is by induction

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

Z_n^*

Z_n^* = { x : x is mod-n 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^*, $x \cdot y$ 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 $n = p_1^{a1} \cdot p_2^{a2} \cdots p_k^{ak}$ where $p_1, \ldots, p_k$ are prime: $\phi(n) = \phi(p_1)^{a1} \cdot \phi(p_2)^{a2} \cdots \phi(p_k)^{ak}$

Proof

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

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

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

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

Removing them from the set {1, 2, ..., n-1} yields $\mathbf{Z}_n^*$

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 mod p and m_q = m mod 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
  $u_p \cdot m_p + v_p \cdot p = 1$ and $u_q \cdot m_q + v_q \cdot q = 1$.
  So $u_p \cdot (m - k \cdot p) + v_p \cdot p = 1$ for some k, 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 $\mathbf{Z}_{p.q}^*$ and $\mathbf{Z}_p^* \times \mathbf{Z}_p^*$. So $\phi(n) = \phi(p) \cdot \phi(q)$.

For $n = p_1^{a1} \cdot p_2^{a2} \cdots p_k^{ak}$ where $p_1, \ldots, p_k$ are prime.
(homework)

End of proof

Euler's Theorem

For all a in Z_n^*: $a^{\phi(n)} = 1$ mod-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 $b_1, b_2, \cdots, b_{\phi(n)}$ be the elements of Z_n^* listed in some order.
Let $y = (a \cdot b_1) \cdot (a \cdot b_2) \cdots (a \cdot b_{\phi(n)})$. So $y = a^{\phi(n)} \cdot x$ mod-n.
But $a \cdot b_1, a \cdot b_2, \cdots, a \cdot b_{\phi(n)}$ is also Z_n^* permuted. So y = x mod-n.
Thus $a^{\phi(n)} \cdot x = x$ mod-n. Multiplying sides by x^{-1} yields $a^{\phi(n)} = 1$ mod-n.

Euler's Theorem Variant:
For all a in $\mathbf{Z}_n^*$ and any non-negative integer k: $a^{k \cdot \phi(n)+1} = a$ mod-n

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

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

Generalization of Euler's Theorem (for a in Z_n and n=p·q)

If n=p·q, where p and q are distinct primes then
$a^{k\cdot\phi(n)+1}$ = a mod-n for all a in $\mathbf{Z}_n$ and any non-negative integer k.

Proof: Assume a not in $\mathbf{Z}_n$* (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 ($a^{k\cdot\phi(n)+1}$ mod-n) into mod-p and mod-q, and use CRT.

$a^{k\cdot\phi(n)+1}$ mod-p = $a^{k\phi(n)}\cdot a$ mod-p
= $a^{k\phi(p)\cdot\phi(q)}\cdot a$ mod-p (because $\phi(n) = \phi(p)\cdot\phi(q)$)
= $a^{\phi(p)\cdot k\cdot\phi(q)}\cdot a$ mod-p
= $1^{k\cdot\phi(q)}\cdot a$ mod-p (a rpt p, so $a^{\phi(p)}$ = 1 mod-p by Euler's theorem)
= a mod-p

Similarly $a^{k\cdot\phi(n)+1}$mod-q = a mod-q
So by CRT $a^{k\cdot\phi(n)+1}$ mod-n = a mod-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: prinicpal 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) mod-n
  - Any x has a unique additive inverse mod-n.
  - Easily computed.
- Modulo-n muliplication: (a·b) mod-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, ..., n-1}
- Z_n* = {numbers in Z_n that are relatively prime to n}
- $\phi(n)$ = number of elements in Z_n* ; easy to get given prime factorization
- Modulo-n exponentiation: (a^b) mod-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 mod-n. (Euler's Theorem)
  - For all x in Z_n* and non-negative k: $x^{k\phi(n)+1}$ = x mod-n. (Variant)
  - For all x in Z_n and non-negative integer k: $x^{k\phi(n)+1}$ = x mod-n
    - if n=p·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 = <e, n>
  - Find multiplicative inverse d of e mod-$\phi(n)$ [i.e., $e \cdot d = 1$ mod-$\phi(n)$]
  - Private key = <d, n>
- Encryption/decryption
  - To encrypt message m using public key:
    - ciphertext $c = m^e$ mod-n
  - To decrypt ciphertext c using private key:
    - plaintext $m = c^d$ mod-n
- Signing/Verifying signature
  - To sign a message m using private key:
    - signature $s = m^d$ mod-n
  - To verify signature c using public key:
    - plaintext $m = s^e$ mod-n

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

$m^{e \cdot d} = m^{1 \text{ mod-}\phi(n)}$ [because $e \cdot d = 1$ mod-$\phi(n)$]

$= m^{1 + k \cdot \phi(n)}$ [definition of mod]

$= m$ [Euler's theorem generalization, applicable because
- m in Z_n (in RSA)
- n is product of distinct primes p and q]

Why is RSA secure

- Only known way to obtain m from m^e is by $m^{e \cdot d}$ where $d = e^{-1}$ mod-$\phi(n)$
- 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 exponentation

- Need to get m^e mod-n, for 512-bit (100-digit) numbers m, e, n
- Consider a small example: 123^{54} mod 678
- **Naive way**: Multiply m with itself e times and then take mod-n.
  - e multiplications of increasingly larger numbers (m^2, m^3,…). Too expensive.
  - 123^{54} is approx 100 digits ($54 \cdot \log_{10} 123$)
- **Better way**: Multiply m with itself and take mod-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.

ModuloExponentiation(m, e, n)

$(x_0, x_1, \cdots, x_k) \leftarrow$ e in binary; // $x_0 = 1$
initially $y \leftarrow m$; $j \leftarrow 0$; // $y = m^{x0}$
while $j < k$ do // loop invariant: $y = m^{(x0,\cdots,xj)}$ mod-n
    $y \leftarrow y \cdot y$ mod-n; // $y = m^{(x0,\cdots,xj,0)}$ mod-n
    if $x_{j+1} = 1$ then $y \leftarrow y \cdot m$ mod-n; // $y = m^{(x0,\cdots,xj,1)}$ mod-n
    $j \leftarrow j + 1$; // $y = m^{(x0,\cdots,xj)}$ mod-n
// $y = m^e$ mod-n

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

Generating RSA Keys consists of two parts:

- find big primes p and q
- finding e relatively prime to $\phi(n)$ (= $(p-1)\cdot(q-1)$)
    - $d = e^{-1}$ 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 Pr(100-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 $a^{n-1} = 1$ mod-n;
    - Prob[test fails] is low (-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 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 mod-n_1, m^3 mod-n_2, m^3 mod-n_3
        - CRT yields m^3 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 $(m^3$ 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 $9FF_{16}$	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 $T_A = g^{SA}$ mod-p to B.
  B chooses random 512-bit number S_B, sends $T_B = g^{SB}$ mod-p to A.
  A computes T_B^{SA} mod-p [= $g^{SB\cdot SA}$ mod-p = $g^{SA\cdot SB}$ mod-p].
  B computes T_A^{SB} mod-p [= equals $g^{SA\cdot SB}$ mod-p].
  A and B now share $g^{SA\cdot SB}$ mod-p, which can serve as a key.
  Attacker knowing T_A and T_B and p and g cannot obtain $g^{SA\cdot SB}$ 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^{SA} mod-p]	
	C sends [sender id B, g^{SC} mod-p]
A and C share secret $g^{SA\cdot SC}$ 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} mod-p)
- Then A can encrypt info with ($g^{SA \cdot SB}$ 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 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 $\{G_1, \cdots, G_k\}$, 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 mod-n (so only A knows square root mod-n of v)
- Authentication
  - A chooses k random numbers, $r_1, \cdots, r_k$
  - A sends r_1^2 mod-n, ⋯, r_k^2 mod-n, to B
  - B randomly splits these into subset 1 and subset 2, and informs A
  - A sends
    - $s \cdot r_i$ mod-n for each r_i^2 mod-n in subset 1
    - r_i mod-n for each r_i^2 mod-n in subset 2
  - B checks whether
    - for each entry in subset 1: $(reply_i)^2 = v \cdot r_i^2$ mod-n
    - for each entry in subset 2: $(replyi)^2 = r_i^2$ mod-n
    - If so, A is authenticated

- Why does it work?
  - Finding square root mod-n is at least as hard as factoring.
    - Knowing $s \cdot r_i$ mod-n does not help obtain r_i 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, $r_1, \cdots, r_k$
  - A forms msg [data to be signed | r_1^2 mod-n, ⋯, r_k^2 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 $s \cdot r_i$ mod-n in position i
    - if hash bit i is 0 then the reply vector has r_i^2 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.