Crypto Stuff

Shankar

May 18, 2013

Overview

- Encryption: plaintext + key $\longrightarrow$ ciphertext
- Decryption: plaintext $\longrightarrow$ ciphertext + same/related key
- Key is secret. It is the ONLY secret.
- Not secret: crypto algorithms, protocols, programs,
- Good crypto algorithm:
  - Given cyphertext, hard to get plaintext.
  - Given plaintext and ciphertext, hard to get key.
  - Hard: requires brute-force search of key-space (eg, 2^{128} keys)
- Types of cryptographic functions:
  - Secret-key: DES, AES, ... // aka symmetric, ordinary
  - Hash (of cryptographic kind): MD5, SHA-1, ...
  - Public-key: RSA, DH, DSS, Fiat-Shamir, ... // aka asymmetric

Secret-key (symmetric/ordinary) crypto

- Same key for encryption and decryption
- Ciphertext about the same length as plaintext.
- Achieve confidentiality, integrity, authentication.

- A and B share secret key K and are separated by insecure channel/storage.
- Confidentiality:
  - A sends $enc(plaintext, K)$
  - B receives and $dec(ciphertext, K)$
- Integrity:
  - MAC (aka checksum): fragment of $enc(plaintext, K)$
  - A sends $[plaintext, MAC]$
  - B receives and verifies MAC
- Authentication:
  - A sends random number r_A to B, and expects $enc(r_A, K)$ back
  - B sends random number r_B to A, and expects $enc(r_B, K)$ back

(Cryptographic) Hash functions

- $H(.)$: <arbitrary-length msg> $\longrightarrow$ <fixed-length hash>
- Easy to compute $H(msg)$ from msg
- Hard to find msg_1 and msg_2 such that $H(msg_1) = H(msg_2)$

- Keyed-hash: Hash msg along with a shared secret K
  eg, $H(msg|K)$ // "|" denotes concatenation

- Keyed-hashing provides all the capabilites of secret-key crypto.
- Integrity
  - $MAC = H(msg|K)$
- Confidentiality
  - Get pad $C_0, C_1, \cdots$ where C_0 random and $C_{i+1} = H(C_i|K)$
  - encryption of $[M_0, M_1 \cdots]$ is $[C_0,\ M_1 \oplus C_1,\ M_2 \oplus C_2, \cdots]$

Public-key (asymmetric) crypto

- Each principal has two related keys:
  - private key (not shared)
  - public key (shared with world)
  - text encrypted with one can only be decrypted with the other
- Confidentiality
  - B transmits text encrypted with $pubkey_A$.
  - A decrypts using $privkey_A$.
- Integrity and digital signature (non-repudiation)
  - A sends encryption of *text* with $privkey_A$
  - Anyone with $pubkey_A$ can decrypt and be assured that A generated it
- Public-key crypto is *orders slower* than ordinary crypto
  - To sign *msg*: sign the hash of *msg*
  - To encrypt *msg*:
    - generate secret-key K,
    - send [encryptn *msg* with K, encryptn K with public key]

Secret-Key Crypto

- Consider fixed-length message of k bits for now (eg, 64, 128)
- Fixed-size key of j bits (eg, 128, 256)
- Encryption S : k-bit msg + j-bit key $\longrightarrow$ k-bit output
- S: 1-1 mapping of msgs to outputs, o/w cannot decrypt
- S must be "random", o/w not secure
  - Msgs and keys that differ only slightly should map to outputs that differ greatly (in approx $k/2$ bits)
- Large enough key length j so that searching 2^j hard

- Clearly, S cannot be a "simple" function, eg, $msg \oplus key$

Secret-Key Crypto (cont)

- Simple solution
  - "Substitution table": random permutation of k-bit strings
  - Table is $2^k \times k$ bits
  - Entries obtained via physical-world randomness (eg, coin toss)
  - $S(i)$ is ith row of table
  - Pro: S is perfectly random
  - Con: Table is itself the key! Too large to be practical
- Want a compact deterministic algorithm.
- Approach: mix small-size tables and global permutations

Secret-Key Crypto (cont)

- Practical approach
  - p: reasonably small divisor of k (eg, $p = 8$)
  - $2^p \times p$ substitution tables // aka "S-boxes"
  - k-bit permutation functions

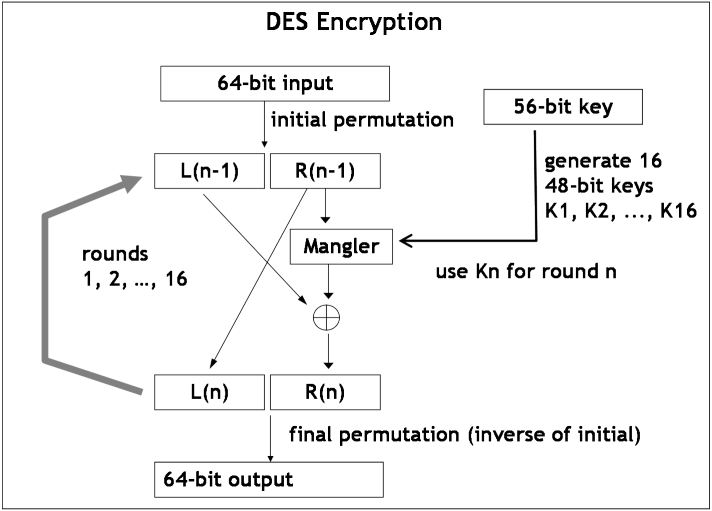
1. Divide k-bit string into p-bit strings
2. Apply S-boxes to p-bit strings // localized scrambling
3. Concatenate the resulting p-bit outputs // k-bit string
4. Apply permutation to get k-bit string // propagate scrambling

  - Repeat 1-4 for n rounds (with 4's output as 1's input)
  - n should be large enough to get good scrambling
    Each output bit is "influenced" by all input bits

- Decryption, ie, reversing, is no more expensive.
  - Often can be done with the same algorithm/hardware.

DES

- Old standard no longer being used: 56-bit keys, 64-bit text



DES (cont)

DES encryption

a1: $L_0 \mid R_0 \leftarrow perm(pt)$
a2: for $n = 0, ..., 15$
a3: $L_{n+1} \leftarrow R_n$
a4: $R_{n+1} \leftarrow mnglr_n(R_n, K_{n+1}) \oplus L_n$
// yields $L_{16} \mid R_{16}$

a5: $L_{17} \mid R_{17} \leftarrow R_{16} \mid L_{16}$
a6: $ct \leftarrow perm^{-1}(R_{16} \mid L_{16})$

// key order: $K_1, \cdots, K_{16}$

DES decryption

b1: $R_{16} \mid L_{16} \leftarrow perm(ct)$ //a6 bw
b2: for $n = 15, ..., 0$ //a2 bw
b3: $R_n \leftarrow L_{n+1}$ //a3 bw
b4: $L_n \leftarrow mnglr_n(R_n, K_n) \oplus R_{n+1}$ //a4 bw
// sets L_n to X such that
// $R_{n+1} \leftarrow mnglr_n(R_n, K_n) \oplus X$
// yields $R_0 | L_0$
b5: $L_0 | R_0 \leftarrow R_0 | L_0$ //a5 bw
b6: pt $\leftarrow perm^{-1}(L_0 | R_0)$ //a1 bw

// key order $K_16, \cdots, K_1$

Multiple Encryption DES (EDE or 3DES)

- Makes DES more secure
  - Encryption: encrypt key1 $\rightarrow$ decrypt key2 $\rightarrow$ encrypt key1
  - Decryption: decrypt key1 $\rightarrow$ encrypt key2 $\rightarrow$ decrypt key1
- encrypt key1 $\rightarrow$ encrypt key1 is not effective
  - Just equivalent to using another single key.
- encrypt key1 $\rightarrow$ encrypt key2 is not so good
- Current standard encryption algorithm: AES
  - different sizes of keys (64, 128, ...)
  - different data block sizes (..., 64, 128, ...)

Encrypting Arbitrary-length Messages

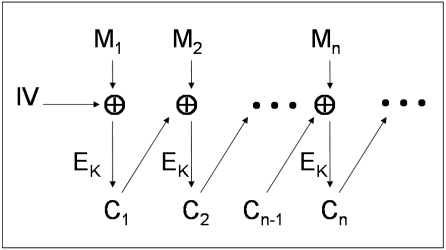
- Encrypting large msg given k-bit block encryption
  - Pad message to multiple of block size:
    $msg \longrightarrow M_1, M_2, \cdots$
  - Use block encryption repeatedly to get ciphertext
    $M_1, M_2, \cdots \quad \longrightarrow \quad C_1, C_2, \cdots$
- Desired
  - $C_j \neq C_k$ even if $M_j = M_k$ // like block encryption
  - Repeated encryptions of *msg* yield distinct *ctxt*
    // unlike block encryption
  - Δ *ctxt* $\not\to$ predictable Δ plaintext // really an integrity issue
- Various methods: ECB, CBC, CFB, OFB, CTR, others

ECB: Electronic Code Book

- Encryption: $M_1, M_2, \cdots \quad \longrightarrow \quad C_1, C_2, \cdots$
- Obvious approach: encrypt each block independently
- Encryption: $C_i = enc_K(M_i)$
- Decryption: $M_i = dec_K(C_i)$
- Not good: repeated blocks get same cipherblock

CBC: Cipher Block Chaining

- Encryption: $M_1, M_2, \cdots \quad \longrightarrow \quad C_1, C_2, \cdots$
- Use C_{i-1} as a "random" pad to M_i before encrypting.
  - $C_0 \leftarrow$ random IV
  - $C_i \leftarrow enc_K(M_i \oplus C_{i-1})$
  - send $C_0, C_1, C_2, \cdots$



- Decryption: $C_1, C_2, \cdots \quad \longrightarrow \quad M_1, M_2, \cdots$
  - $M_i \leftarrow dec_K(C_i \oplus C_{i-1})$, for $i = 1, 2, \cdots$

- "Attacks" on integrity:
  - $X \oplus C_n \longrightarrow M_n$ garbled, $M_{n+1} \leftarrow \oplus X$, other M_i's unchanged.
  - Can somewhat overcome with ordinary checksum (eg, CRC)

OFB: Output Feedback Mode

- Encryption: $M_1, M_2, \cdots \quad \longrightarrow \quad C_1, C_2, \cdots$
- Generate pad $B_0, B_1, \cdots$:
  - B_0 is IV
  - $B_i \leftarrow enc_K(B_{i-1})$
- $C_i \leftarrow B_i \oplus M_i$
- One-time pad that can be generated in advance.
- Attacker with <plaintext, ciphertext> can obtain B_i's. Hence generate ciphertext for any plaintext

CFB: Cipher Feedback Mode

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

MACs from encryption

- MAC: message authentication code, aka cryptographic checksum
- Provides integrity

- Encrypting msg (using CBC, CFB, OFB) does not provide integrity
  - Modified ciphertext yields plaintext that a human or program *may* find fishy
  - But not a MAC

- MAC is usually generated by hash functions

- Standard way to generate MAC with an encryption function
  - residue(msg): last block in CBC encryption of msg
  - MAC = [IV, residue (msg)]

Confidentiality and Integrity with Encryption

- Send enc(msg) | residue(msg)] // not ok
  - Just repeats the last cipherblock
- enc(msg | residue(msg)) // not ok
  - Last block is enc(0) //$\oplus$ of last cipherblock with itself
- enc(msg | ordinary_checksum(msg)) // not ok
  - Almost works. Subtle attacks are known.
- enc_{Key2}(msg | residue_{Key1}(msg) // ok
  - But twice the work.
  - $Key2$ can be related to $Key2$ (eg, $Key1 = Key2 + 1$)
- encrypt(msg | weak_crypto_checksum(msg)) // probably ok

- Offset Codebook Mode (OCB)

Hashes, aka Message Digests

- Hash function H: arbitrary message $\longrightarrow$ k-bit hash
  - Not 1-1: msg space $\gg$ hash space ($= 2^k$)
- Want: hard to find any two msg_1, msg_2 st $H(msg_1) = H(msg_2)$
  - This is stronger than collision for a given msg_1

- Assuming H is random, how large should k be?
- $Pr(\text{collision in } N \text{ random messages}) \approx N^2/K$
  - N random messages, $m_1, m_2, \cdots, m_N$
  - $Pr[\text{collision}]$
    $= Pr[H(m_1) = H(m_2) \text{ or } H(m_1) = H(m_3) \text{ or } \cdots]$
    $= (N(N-1)/2)(1/K)$

- Want searching through $\sqrt{2^k}$ to be hard
  - So $k = 128$ assumes searching through 2^{64} is hard

Keyed Hash: Hash (msg + secret key)

- Keyed-hash $H_K(msg)$:
  - hash H applied to some merge of message msg and key K
- Equivalent to secret-key encryption

- Encryption: $M_1, M_2, \cdots \longrightarrow C_0, C_1, C_2, \cdots$
  - Generate pad: $B_i \leftarrow H_K(B_{i-1})$ where B_0 is IV
  - $C_i \leftarrow B_i \oplus M_i$
  - Transmit IV and $C_1, C_2, \cdots$
  - Decryption identical
- Encryption with plaintext mixed into pad is similar
  - $B_i \leftarrow H_K(C_{i-1})$ where C_0 is IV
  - $C_i \leftarrow B_i \oplus M_i$

- Authentication:
  - A sends random r_A and expects to get $H_K(r_A)$
  - B sends random r_B and expects to get $H_K(r_B)$

Keyed hash: How to merge *msg* and key K

- $H(K|msg)$ NOT OK
  - Because usually $H(msg_1|msg_2)$ is $H(H(msg_1))$
  - So given msg and $H_K(msg)$, attacker can append any m to msg and get $H_K(msg|m)$ by $H(H_K(msg))$

- OK
  - $H(msg|K)$
  - half the bits of $H(K|msg)$
  - $H(K|msg|K)$

- HMAC standard
  - Any hash function H (eg, MD2, MD4, SHA-1) and any key size
  - $paddedKey \leftarrow$ pad key with 0's to 512 bits
    if key is larger than 512 bits, first hash key and then pad
  - $h1 \leftarrow H(msg|paddedKey \oplus$ [string of 36_{16} octets])
  - MAC: $H(h1|paddedKey \oplus$ [string of $5C_{16}$ octets])

MD4: Message Digest 4

- MD4: 128-bit hash, 32-bit architecture
- Step 1: Pad *msg* to multiple of 512 bits
  - $pmsg \leftarrow msg|$one 1$|$ p 0's$|$ (64-bit encodng of p) // p in 1..512

- Step 2: Process *pmsg* in 512-bit chunks to get hash *md*
  - treat 128-bit *md* as 4 words: d_0, d_1, d_2, d_3
    - initialize to 01|23|...|89|ab|cd|ef|fe|dc|...|10

  - For each successive 512-bit chunk of *pmsg*:
    - treat 512-bit chunk as 16 words: $m_0, m_1, \cdots, m_{15}$
    - $e_0..e_3 \leftarrow d_0..d_3$ // save for later
    - pass 1 using mangler $H1$ and permutation J
      // for $i = 0, ..., 15$: $d_{J(i)} \leftarrow H1(i, d_0, d_1, d_2, d_3, m_i)$
    - pass 2: same but with mangler $H2$
    - pass 3: same but with mangler $H3$
    - $d_0..d_3 \leftarrow d_0..d_3 \oplus e_0..e_3$
  - $md \leftarrow d_0..d_3$

More Hash Functions

- MD2: octet-oriented
- message of arbitrary number of octets $\longrightarrow$ 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 $\longrightarrow$ 128-bit digest
  - Like MD4 except four passes and different mangler functions

- SHA-1: 32-bit word oriented
  - Message of size upto 2^{64} bits $\longrightarrow$ 160-bit digest
  - Like MD5 except five passes, different mangler functions, at each stage, 512-bit msg chunk $\longrightarrow 5 \times 512$-bit chunk

- $\cdots$

Public-Key Crypto

- Prinicpal has a key-pair: [public key, private key]
  - private key: secret shared with no other
  - public key: disclosed to every one
  - text encrypted with one key can be decrypted only with the other key

- Public-key crypto algorithms and typical usage
  - RSA, ECC: encryption and digital signatures
  - ElGamal, DSS: digital signatures
  - Diffie-Hellman: establishment of a shared secret
  - Zero-knowledge proof systems: authentication

- Public-key algorithms involve
  - Prime numbers
  - Modulo-n addition, multiplication, exponentiation
  - A brief review follows.

Prime numbers

- Integer p is prime iff it is exactly divisible only by itself and 1.
- $gcd(p, q)$: greatest common denominator of integers p and q
  - Largest integer that divides both exactly.
- p and q are relatively prime iff $gcd(p, q) = 1$

- 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] $= 1/(10 \cdot \ln k)$

Modulo-n arithmetic

- $Z_n = \{0, 1, \cdots, n-1\}$

- Modulo-n operation: integers $\longrightarrow Z_n$
- x mod-n, for any integer x (including negative)

  $= y$ in Z_n st $x = y + k{\cdot}n$ for some integer k

  $=$ non-negative remainder of x/n
- Examples
  - 3 mod-$10 = 3$ // $3 = 3 + 0{\cdot}10$
  - 23 mod-$10 = 3$ // $23 = 3 + 2{\cdot}10$
  - -27 mod-$10 = 3$ // $-27 = 3 + (-3){\cdot}10$

    Note: mod-n of negative number is non-negative

Modulo-n addition

- $(a + b)$ mod-n, for any integers a and b
- Examples
  - $(3 + 7)$ mod-10 $=$ 10 mod-10 $=$ 0
  - $(3 - 7)$ mod-10 $=$ -4 mod-10 $=$ 6

- Additive-inverse-mod-n of x
  - y st $(x + y)$ mod-$n = 0$
  - Denoted $-x$ mod-n
  - Exists for every x
  - Easily computed: $(n - x)$ mod-n

Modulo-n multiplication

- $(a \cdot b)$ mod-n, for any integers a and b
- Examples
  - $(3 \cdot 7)$ mod-10 $=$ 21 mod-10 $=$ 1
  - $8 \cdot (-7)$ mod-10 $=$ -56 mod-10 $=$ 4

- Multiplicative-inverse-mod-n of x
  - y st $(x \cdot y)$ mod-$n = 1$
  - Denoted x^{-1} mod-n
  - Exists iff $gcd(x, n) = 1$ // x relatively prime to n
  - Euclid's algorithm computes
    - $gcd(x, n)$
    - u, v st $gcd(x, n) = u \cdot x + v \cdot n$
    - if $gcd(x, n) = 1$:
      $u = x^{-1}$ mod-n
      $v = n^{-1}$ mod-x

Modulo-n exponentiation

- (a^b) mod-n, for any integers a and $b > 0$
- Examples
  - 3^2 mod-10 $=$ 9
  - 3^3 mod-10 $=$ 27 mod-10 $=$ 7
  - $(-3)^3$ mod-10 $=$ $-$27 mod-10 $=$ 3

- Exponentiative-inverse-mod-n of x
  - y st (x^y) mod-$n = 1$
  - Exists iff $gcd(x, n) = 1$
  - Easy to compute if prime factors of n are known. Otherwise not.

Euler's Theorem

- $Z_n^* = \{x\text{:}\ \ x \text{ in } Z_n,\ \ \ gcd(x,n) = 1\}$
  - Z_{10} : $\{0,1,2,3,4,5,6,7,8,9\}$
  - Z_{10}^* : $\{1,3,7,9\}$
- $\phi(n)$: number of elements in Z_n^*
- Euler's Totient Function

$$\phi(n) = \begin{cases} n-1 & \text{if } n \text{ prime} \\ (p-1)\cdot p^{a-1} & \text{if } n = p^a, p \text{ prime, } a > 0 \\ \phi(p)\cdot\phi(q) & \text{if } n = p\cdot q \text{ and } gcd(p,q) = 1 \\ \phi(p_1^{a1})\cdots\phi(p_K^{aK}) & \text{if } n = p_1^{a1}\cdots p_K^{aK} \end{cases}$$

- Euler's Theorem

  If $n = p \cdot q$, where p and q are distinct primes then
  $a^{k\cdot\phi(n)+1} \ = \ a$ mod-n for all a in Z_n and any $k > 0$.

RSA

- RSA: Rivest, Shamir, Adleman
- Key size variable and much longer than secret keys
  - usually greater than 512 bits (100 decimal digits)
- Plaintext block size variable but smaller than key
- Ciphertext block of key length.
- Orders slower than secret-key algorithms (eg, AES)
  - So not used for data encryption

RSA: Generating [public key, private key] pair

- Choose two large primes, p and q // p and q remain secret
- Let $n = p \cdot q$
- Choose e relatively prime to $\phi(n)$ // $\phi(n) = (p-1) \cdot (q-1)$
- Public key $= [e, n]$ // disclosed to the world
- Find d, mult-inverse-mod-$\phi(i)$ of e // $e \cdot d = 1$ mod-$\phi(n)$
- Private key $= [d, n]$ // do not share

RSA: Encryption and Signing

- Encryption of msg m usin public key
  - ciphertext $c \leftarrow m^e$ mod-n
- Decryption of ciphertext c using private key
  - plaintext $m \leftarrow c^d$ mod-n
- Works because $m^{e \cdot d} \;=\; m$

- Signing message m using private key
  - signature $s \leftarrow m^d$ mod-n
- Verifying signature s using public key
  - plaintext $m \leftarrow s^e$ mod-n
- Works because $m^{e \cdot d} \;=\; m$

Why is $m^{e \cdot d}$ equal to m

- $m^e \cdot d \;=\; m^1$ mod-$\phi(n)$ // because $e \cdot d$ mod-$\phi(n) \;=\; 1$
  $= m^{1+k \cdot \phi(n)}$ for some k // definition of mod
  $= m$ // Euler's theorem, m in Z_n,
  // n is product of distinct primes p and q

Why is RSA secure

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

Efficient modulo exponentation

- Need to get m^e mod-n for large (eg, 100-digit) numbers m, e, n
  - 3-digit example: 123^{54} mod-678
- Naive: Multiply m by itself e times, then take mod n.
  - e multiplications of increasingly larger numbers
  - 123^{54} is approx 100 digits // $54 \cdot \log_{10} 123$
- Better: Multiply m with itself, take mod n; repeat e times.
  - e multiplications and divisions of large numbers.
- Much better: Exploit $m^{2x} = m^x \cdot m^x$ and $m^{2x+1} = m^{2x} \cdot m$.
  - $\log e$ multiplications.

Modulo_Exponentiation(m, e, n)

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

    $y \leftarrow y \cdot m$ mod-n // $y = m^{(x0, \cdots, x_j, 1)}$ mod-n
  - $j \leftarrow j + 1$ // $y = m^{(x_0, \cdots, x_j)}$ mod-n
- // $y = m^e$ mod-n

Example: 123^{54} mod-678

- 54 in binary is $(110110)_2$
- $123^{(1)}$ 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 $= 189225$ 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 has two parts

- Finding big primes p and q
- Finding e relatively prime to $\phi(p{\cdot}q)$ // $= (p-1){\cdot}(q-1)$
  - Given e, easy to obtain $d = e^{-1}$ mod-$\phi(n)$

Finding big prime n

- Choose random n and test for prime. If not prime, retry.
- No practical deterministic test.
- Simple probabilistic test
  - Generate random n and random a in $1..n$
  - Pass if $a^{n-1} = 1$ mod-n // converse to Euler's theorem
  - Prob failure is low // -10^{-13} for 100-digit n
  - Can improve by trying different a's.
  - But Carmichael numbers: $561, 1105, 1729, 2465, 2821, 6601, \cdots$
- Miller-Rabin probabilistic test: better and handles Carmichael

Finding e

- Approach 1
  - Choose random primes p and q as described above
  - Choose e at random until e relatively prime to $\phi(p.q)$
- Approach 2
  - Fix e st m^e easy to compute (i.e., few 1's in binary)
  - Choose random primes p and q st e relatively prime to $\phi(p.q)$

Approach 2 with $e = 3$

- m^3 requires 2 multiplications
- Need pad for small m:
  - If $m < n^{1/3}$ then m^3 mod-$n = m^3$
  - Attacker gets m by $(m^e)^{1/3}$
- Need 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-$n_1 \cdot n_2 \cdot n_3$, which equals m^3 because $m < n_1, n_2, n_3$.

Approach 2 with $e = 2^{16} + 1 = 65537$

- m^e requires 17 multiplications
- No need for pad since unlikely that $m^{65537} < n$
- Need different pads if m sent to 65537 recipients with same e. Unlikely.

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 $e = 3$
  - $\cdots$

- Encryption (fields are octets)

msb	0	2	$\geq$ eight random non-zero octets	0	data	lsb

  Note that the data is usually small (key, hash, etc)

- Signing (fields are octets)

msb	0	1	$\geq$ eight octets of $9F_{16}$	0	digest type and digest (in ASN.1)	lsb

Basic Diffie-Helman

- Share key over open channel using public prime p and g $(< p)$

A	B
choose random S_A	
$T_A \leftarrow g^{S_A}$ mod-p	
send T_A	choose random S_B
	$T_B \leftarrow g^{S_B}$ mod-p
	$K_B \leftarrow T_A^{S_B}$ mod-p
	send T_B
$K_A \leftarrow T_B^{S_A}$ mod-p	

- $K_A \;=\; K_B \;=\; g^{S_A \cdot S_B}$ mod-p // shared key
- Hard to get $g^{S_A \cdot S_B}$ mod-p from T_A and T_B
- DH by itself does not provide authentication

Diffie-Helman with Published Numbers

- Let a set of principals share public DH parameters p and g
- Let every principal X generate random S_X and $T_X = g^{S_X}$ mod-p
  - S_X: X's private key // held secret
  - $[X, g, p, T_X]$: X's public key // made public

- Assume PKI (public-key infrastructure) that publishes $[X, g, p, T_X]$ for every principal X.
- Then any two principals X, Y share key $g^{S_X \cdot S_Y}$ mod-p

Authenticated Diffie-Helman

- DH that incorporates a pre-shared key to provide authentication.
- Authenticated DH when A and B share a secret key K
  - Encrypt (messages of) basic DH exchange with K
    - A sends $enc_K(g^{S_A} \text{ mod-}p)$
    - B sends $enc_K(g^{S_B} \text{ mod-}p)$
    - shared key: $g^{S_A \cdot S_B} \text{ mod-}p$
  - Following basic DH exchange, exchange keyed-hashes of shared DH key and sender names.
  -
- Authenticated DH when A and B have each other's public key.
  - Encrypt basic DH exchange with receiver's public key.
  - Sign basic DH exchange with sender's private key.
  -

Why DH given pre-shared secret

- Get strong key ($g^{S_A \cdot S_B}$ mod-p) even if pre-shared secret is weak
- Perfect-forward secrecy:
  - Suppose A and B forget S_A and S_B after their session
  - Then session data is safe even if pre-shared secret later exposed