Introduction to Cryptology (NS chapter 2)

Encryption: plaintext + key → ciphertext
Decryption: plaintext ← ciphertext + 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 (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 capabilities 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⊕C_1, M_2⊕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 orders 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-session 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 (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

- 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

Final permutation is inverse of initial permutation. Not of security value (why?, what does this mean?)
DES: Generation of K1, K2, ..., K16

- 56-bit key
- permute 56-bit key, split to form two 28-bit parts
- rotate each left by 1 bit in rounds 1, 2, 9, 16
- rotate each left by 2 bits in other rounds
- Each part: permute, drop some bits to form 24-bit chunk. Join to form one 48-bit key K1
- repeat 16 times get Ci, Di, K

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

```
DES_encryption {
  a1: L0 | R0 ← iperm(dblk);
  for n = 0, ..., 15 do
    a2: Ln+1 ← Rn;
    a3: Rn ← Mangler_r(Rn, K_n-1) ⊕ Ln;
        //Yields L16|R16
    a4: L0 | R0 ← swap (L0 | R0);
  }
  // key order: K15, ..., K1
```

DES_encryption {
  b1: R16 | L16 ← iperm(cblk);  //a6 bkw
  b2: for n = 15, ..., 0 do //a2 bkw
    b3: Rn ← L_n+1;
    b4: L_n ← Mangler_r (R_n, K_n) ⊕ R_n-1;
        //a4 bkw
  // sets L_n to X such that
  // R_n+1 ← Mangler_r (R_n-1, K) ⊕ X
  // Yields R0 | L0
  b5: L0 | R0 ← swap (R0 | L0);
    //a5 bkw
  b6: dblk ← ipermInv (R16 | L16);
      // key order K15, ..., K1
```

DES encryption round

- 64-bit input
- 48-bit key K(n+1)
- same as encryption with arrows reversed except for mangler function

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 $C_0 = D_0 = \text{all ones or all zeros}$
- 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 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 $K_1, \ldots, K_{16}$ are all equal to each other.
So the key sequence $K_1, \ldots, K_{16}$ (used in encryption) is the same as the key sequence $K_{16}, \ldots, K_1$ (used in decryption).
So encryption and decryption are the same, i.e., $E_x(b) = D_x(b)$.
So $E_x(E_x(b)) = b$.

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

Proof

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

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
- 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 $<m_1, c_1>, <m_2, c_2>, \ldots$, there is an attack that requires $2^{56}$ storage.
    - Table A with $2^{36}$ entries $<K_1, E(K_1, m_1)>$, sorted by column 2.
    - Table B with $2^{36}$ entries $<K_1, D(K_1, c_1)>$, sorted by column 2.
    - Do join of Table A and Table B.
    - Each match provides candidate $<K_a, K_b>$ for $<key1, key2>$.
    - Use $<m_2, c_2>$, 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

```
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: \( \text{msg} = (M_1, M_2, \ldots) \)
- Use block encryption repeatedly to get ciphertext = \((C_1, C_2, \ldots)\)
  - 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 cipherblocks

Output Feedback Mode (OFB)

64-bit OFB

- Generate stream cipher \( B_0, B_1, \ldots \), 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:
  - \( \text{Attacker with <plaintext, ciphertext> can obtain } B_i \)'s
  - \( \text{and so generate ciphertext for any plaintext} \)

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 \( B_{i-1} \| Y_i \)

Cipher Block Chaining (CBC)

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

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 \( B_{i-1} \| Y_i \)
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.

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 (i.e., if some ciphertext is garbled, everything is lost)

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 (⊕ 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 → 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 m1 and m2 s.t. h(m1)=h(m2)

Assuming hash is random, how long should it be?
- Consider hash space of K (i.e., hash of (log K) bits)
- Consider N randomly chosen messages, m1, m2, ..., mN
- Pr[ there is a pair of distinct msgs < m1, m1 > : H(m1) = H(m1) ]
  - = Pr[ H(m1)=H(m2) or H(m1)=H(m3) or ... or H(mN-1)=H(mN) ]
  - = Sum {over distinct < m1, m1 > pairs} (1/K)
  - = [N(N-1)/2] [1/K]
- So if N= √K then Pr is 1/2
- K should be large enough so that searching through √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 $msg = (m_1, m_2, ..., m_n)$ given shared secret key $K_{AB}$
- Obvious approach: $MAC = H(K_{AB} | msg)$
- Not ok because $H(m_1, m_2, ..., m_n)$ is usually $H(H(m_1, m_2, ..., m_{n-1}) m_n)$
- 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 $msg = (m_1, m_2, ..., m_n)$
- 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 $c_1, c_2, ..., c_n$
- Decryption identical

Encryption and integrity of $msg = (m_1, m_2, ..., m_n)$
- 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 \( m_1, m_2, \ldots \)
  - \( C_i = \) 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:
  - \( C_i = \) encryption of a constant \( C_{0i} \) with \( M_i \) 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.

More Hash Functions

- **MD2: octet-oriented**
  - Message of arbitrary number of octets \( \rightarrow \) 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 \( \rightarrow \) 128-bit digest
  - Like MD4 except four passes and different mangler functions

- **SHA-1: 32-bit word oriented**
  - Message of arbitrary number of bits up to \( 2^{64} \) bits \( \rightarrow \) 160-bit digest
  - Like MD5 except five passes, different mangler functions, and at start of each stage, 512-bit msg chunk \( \rightarrow \) 5 x 512-bit chunk using rotated versions of the msg chunk

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
    \( pmsg \leftarrow msg \mid \) one 1 \mid p 0's \mid (64-bit encoding of p);
    where \([msgsize+1\mid p\mid 64]\) 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: \( m_0, m_1, \ldots, m_{15} \)
    - Initialize \( <d_0, d_3> \rightarrow \) \( <01, 23|\ldots|89|ab|cd|ef|fe|dc|\ldots|10> \);
    - For each 512-bit chunk \( c \) of msg:
      \[ e_0, e_1 \leftarrow d_0, d_3; \]

    - **Pass 1:** mangle \( d_0, d_3 \) using \( m_0, m_{15} \), mangler \( H_1 \), permutation \( J \)
      For \( i = 0, \ldots, 15: \)
      \( d_{j(i)} \leftarrow H_1(i, d_0, d_1, d_2, d_3, m_i); \)

    - **Pass 2:** mangle \( d_0, d_3 \) using \( m_0, m_{15} \), mangler \( H_2 \), permutation \( J \)
      For \( i = 0, \ldots, 15: \)
      \( d_{j(i)} \leftarrow H_2(i, d_0, d_1, d_2, d_3, m_i); \)

    - **Pass 3:** mangle \( d_0, d_3 \) using \( m_0, m_{15} \), mangler \( H_3 \), permutation \( J \)
      For \( i = 0, \ldots, 15: \)
      \( d_{j(i)} \leftarrow H_3(i, d_0, d_1, d_2, d_3, m_i); \)

      \( d_0, d_3 \leftarrow d_0, d_3 \oplus e_0, e_3; \)

      \( md \leftarrow d_0, d_3 \)

HMAC: defacto MAC standard

- Can use any hash function \( H \) (eg, MD2, MD4, SHA-1)
- Variable-sized message and variable-length key \( \rightarrow \) fixed-size MAC of same size as output of \( H \)
- \( p\text{addedKey} \leftarrow \) pad key with 0's to 512 bits
  - If key is larger than 512 bits, first hash key and then pad
- \( h1 \leftarrow H(\text{msg}, p\text{addedKey} \oplus \text{[string of 36, octets]} ) \)
- \( \text{result} \leftarrow H( h1, p\text{addedKey} \oplus \text{[string of 5C, octets]} ) \)
A Bit of Number Theory (NS chapter 7)

Need some number theory to understand public key cryptology
- Modular addition, multiplication, exponentiation over \( \mathbb{Z}_n = \{ 0, 1, \ldots, n-1 \} \)
- Euclid’s algorithm: gcd and multiplicative inverse
- Chinese remainder theorem: \( (x \mod pq) \iff (x \mod p) \text{ and } (x \mod q) \)
- \( \mathbb{Z}_n^* = \{ j : j > 0 \text{ and relatively prime to } n \} \)
- Euler’s totient function \( \phi(n) = | \mathbb{Z}_n^* | \)
- 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 \( \mathbb{Z}_n \) s.t. \( x = y + k \cdot n \) for some integer \( k \).
- Nonnegative remainder of \( x/n: \)
  - \( 3 \mod 10 = 3 \) \( (3 = 3 + 0 \cdot 10) \)
  - \( 23 \mod 10 = 3 \) \( (23 = 3 + 2 \cdot 10) \)
  - \( -27 \mod 10 = 3 \) \( (-27 = 3 + (-3) \cdot 10) \) (unlike in most prog lang)
- Integers \( u \) and \( v \) are said to be \textit{equal mod-}\( n \) if \( (u \mod n) = (v \mod n) \)
  o 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 \textit{mod-}\( n \) operation
  o \( (3+7) \mod 10 = 10 \mod 10 = 0 \)
  o \( (3−7) \mod 10 = −4 \mod 10 = 6 \)
- Note: \( (u+v) \mod n = (u \mod n) + (v \mod n) \mod n \)
- \textit{Additive inverse} mod-\( n \) of \( x \) is \( y \) st \( (x+y) \mod n = 0 \)
  o denoted \( −x \mod n \)
  o exists for any \( x \) and \( n \)
  o easy to compute: eg, for \( x \) in \( \mathbb{Z}_n \), additive inverse is \( n−x \)

Modulo-\( n \) multiplication and multiplicative inverse
- Mod-\( n \) multiplication is ordinary multiplication followed by \textit{mod-}\( n \) operation
  o \( (3 \cdot 7) \mod 10 = 21 \mod 10 = 1 \)
  o \( (8) \cdot (−7) \mod 10 = −56 \mod 10 = 4 \)
- Note: \( (u \cdot v) \mod n = (u \mod n) \cdot (v \mod n) \mod n \)
- \textit{Multiplicative inverse} mod-\( n \) of integer \( x \) is \( y \) s.t. \( (x \cdot y) \mod n = 1 \)
  o denoted \( x^{-1} \mod n \)
  o \( 3^{-1} \mod 10 \) is \( 7 \) \( (3 \cdot 7 = 21 = 1 \mod 10) \).
  o \( x^{-1} \) exists and is unique iff \( x \) and \( n \) are relatively prime
  - \( \text{ie, } \gcd(x,n) = 1 \)
- 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 \textit{mod-}\( n \)
  o \( 3^2 \mod 10 = 9 \)
  o \( 3^3 \mod 10 = 27 \mod 10 = 7 \)
  o \( (−3)^2 \mod 10 = −27 \mod 10 = 3 \)
- Note: \( (u^x) \mod n \neq (u \mod n)^x \mod n \)
- \textit{Exponentiative inverse} mod-\( n \) of integer \( x \) is \( y \) s.t. \( (x^y \mod n) = 1 \)
  o \( 3^4 = 81 = 1 \mod 10 \), so \( 4 \) is the exponentiative inverse mod-\( 10 \) of \( 3 \)
  o Exists and is unique iff \( x \) and \( n \) are relatively prime
  o 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
  o 25 primes less than 100
  o \( \Pr[ \text{ random 10-digit number is a prime } ] = 1/23 \)
  o \( \Pr[ \text{ random 100-digit number is a prime } ] = 1/230 \)
  o \( \Pr[ \text{ random k-digit number is a prime } ] = 1/(10 \cdot \ln k) \)
Euclid’s algorithm for \( \text{gcd}(x, y) \)
- \([x, y]\) has same divisors/\( \text{gcd} \) as \([x-y, y]\), \([x\mod y, y]\), \([x, x\mod y]\), \([y, \text{remainder}(x/y)]\)
- repeat \([x, y] \rightarrow [y, \text{remainder}(x/y)]\) until first entry is 0; second entry is gcd
- store intermediate remainders in array \( r \)
- \( r = [r_2, r_1, r_0, r_1, r_2, \ldots] \)
- \( x \mod y \) has same \( \text{gcd} \) as \( (x-y) \mod y \), \( [x-y, y] \)

Euclid \((x, y)\) with intermediate remainders
- array \( r = [r_2, r_1, r_0, r_1, r_2, \ldots] \)
- \( r_2 \leftarrow x; r_1 \leftarrow y; \)
- integer \( n \leftarrow 0; \)
- while \( r_{n+1} \neq 0 \)
  - \( r_n \leftarrow \text{remainder}(r_{n-2}/r_{n-1}); \)
  - \( n \leftarrow n+1; \)
- return \( r_{n-2}; \) \( \text{gcd}(x, y) \)

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

---

**Chinese remainder theorem**

Let \( z_1, z_2, \ldots, z_k \) be relatively prime.
Then the mapping \( Z_{z_1, z_2, \ldots, z_k} \) \( \rightarrow \) \( \mathbb{Z}/z_1 \times \mathbb{Z}/z_2 \times \ldots \times \mathbb{Z}/z_k \) where \( x \rightarrow \langle x \mod z_1, x \mod z_2, \ldots, x \mod z_k \rangle \) is 1-1 (so invertible). So for \( \langle x_1, x_2, \ldots, x_k \rangle \): exactly one \( x \) in \( Z_{z_1, z_2, \ldots, z_k} \) s.t. \( x \mod z_i = x_i \)
- For \( k=2 \), \( (x \mod z_1; z_2) = [x_2 \cdot a \cdot z_1 + x_1 \cdot b \cdot z_2] \mod z_1; z_2 \), where \( 1 = a \cdot z_1 + b \cdot z_2 \)

\( z_1=3, z_2=4 \) (relatively prime)

\[
\begin{array}{ccccccccccc}
  z_1 & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 \\
  z_2 & 0 & 0 & 0 & 0 & 1 & 1 & (1,0) & (2,1) & (2,2) & (0,1) & (1,2) & (2,3) \\
  z_1 \times z_2 & (0,0) & (1,1) & (2,2) & (0,3) & (1,0) & (2,1) & (0,2) & (1,3) & (2,0) & (0,1) & (1,2) & (2,3) \\
\end{array}
\]

\( z_1=2, z_2=4 \) (not relatively prime)

\[
\begin{array}{cccccccc}
  z_1 & 0 & 1 & 2 & 3 & 4 & 5 & 6 \\
  z_2 & 0 & 0 & 0 & 0 & 1 & 1 & (1,1) \\
  z_1 \times z_2 & (0,0) & (1,1) & (0,2) & (1,3) & (0,0) & (1,1) & (0,2) \\
\end{array}
\]

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 \( Z_{z_1 \times z_2} \) 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
  - \( x \mod z_1 = x_1 \) and \( x \mod z_2 = x_2 \)
  - 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 \( z_1; z_2 \)
  - \( (x \mod z_1; z_2) = [x \cdot a \cdot z_1 + x \cdot b \cdot z_2] \mod z_1; z_2 \)
  - \( = [(x_2 + k \cdot z_1) \cdot a \cdot z_1 + (x_1 + j \cdot z_2) \cdot b \cdot z_2] \mod z_1; z_2 \)
  - \( = (x_2 \cdot a \cdot z_1 + x_1 \cdot b \cdot z_2) \mod z_1; z_2 \)
- LHS depends only on \( x_1, x_2, a, b \).
  - So for any \( \langle x_1, x_2 \rangle \), exactly one \( x \) s.t. \( (x \mod z_1) = x_1 \) and \( (x \mod z_2) = x_2 \)
  - So \( x \) and \( y \) are the same mod \( z_1; z_2 \)

**Proof of for \( k > 2 \) is by induction**
- If \( z_1, z_2, \ldots, z_k, z_{k+1} \) rel. prime, then \( (z_1; z_2; \ldots; z_k) \) and \( z_{k+1} \) are rel. prime
Z_n^*

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

- \[ Z_{10}^* = \{ 1, 3, 7, 9 \} \] 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: \( x, y \in Z_n^* \), \( xy \mod n \) 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. \( u_a a + v_a n = 1 \) and \( u_b b + v_b n = 1 \).

Multiply the two equations: \( u_a u_b (a b) + n (u_a v_b a + v_b b u_b + u_b v_b n) = 1 \)
Hence, by Euclid alg, \( a b \) is relatively prime to n, and so \( a b \) is in \( Z_n^* \).

To show \( x Z_n^* \) 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 \( n = p_1^{a_1} \cdot p_2^{a_2} \cdot \ldots \cdot p_k^{a_k} \) where \( p_1, \ldots, p_k \) are prime:
  \( \phi(n) = \phi(p_1)^{a_1} \cdot \phi(p_2)^{a_2} \cdot \ldots \cdot \phi(p_k)^{a_k} \)

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

For \( n = p^a \) where p is prime and \( a > 0 \):
\( \phi(n) = (p-1) \cdot p^{a-1} \)
\( Z_n = \{ 0, 1, 2, \ldots, p, \ldots, 2p, \ldots, 3p, \ldots, \ldots, (p^{a-1} - 1) p, \ldots, (p^a - 1) \} \).
Only the multiples of p can divide n. There are \( (p^{a-1} - 1) \) of them.
Removing them from the set \( \{ 1, 2, \ldots, n-1 \} \) yields \( Z_n^* \).
So \( \phi(n) = (n-1) - (p^{a-1} - 1) = (p^a - 1) - (p^{a-1} - 1) = p^a - p^{a-1} = (p-1) \cdot p^{a-1} \)

\[ \phi(n) = \phi(n)+1 \]

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

**Euler’s Theorem Variant:**
For all \( a \in Z_n^* \) and any non-negative integer \( k \):
\( a^{k \phi(n)+1} = a \mod n \)

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

**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 \( \mathbb{Z} \) and \( n=p\cdot\cdots\cdot q \))

If \( n=p\cdot q \), where \( p \) and \( q \) are distinct primes then
\[
    a^{\phi(n)+1} \equiv 1 \pmod{n}
\]
for all \( a \) in \( \mathbb{Z} \) and any non-negative integer \( k \).

Proof: Assume \( a \) not in \( \mathbb{Z}^* \) (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 \( (a^{\phi(n)+1} \mod n) \) into \( \mod p \) and \( \mod q \), and use CRT.

\[
    a^{\phi(q)+1} \mod p = a^{\phi(p)\cdot\phi(q)} \cdot a \mod p
    = a^{\phi(p)} \cdot a \mod p
    = 1 \cdot a \mod p
    \quad \text{(a rpt p, so } a^{\phi(p)} = 1 \pmod p \text{ by Euler’s theorem)}
    = a \mod p
\]
Similarly \( a^{\phi(q)+1} \mod q = a \mod q \)
So by CRT \( a^{\phi(n)+1} \mod n = a \mod n \)

Further generalization:
Above is true for any \( n \) that is a product of distinct primes.

Public Key Algorithms (NS chapter 6)

- Public key algorithm: 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: \( (a+b) \mod n \)
  - Any \( x \) has a unique additive inverse \( \mod n \).
  - Easily computed.

- Modulo-\( n \) multiplication: \( (a\cdot b) \mod n \)
  - Any \( x \) has a unique multiplicative inverse \( \mod n \) iff \( \gcd(x,n)=1 \)
  - Existence and value easily computed (Euclid’s alg)

- \( \mathbb{Z}_n = \{0, 1, \ldots, n-1\} \)
- \( \mathbb{Z}_n^* = \{ \text{numbers in } \mathbb{Z}_n \text{ that are relatively prime to } n \} \)
- \( \phi(n) = \text{number of elements in } \mathbb{Z}_n^* \); easy to get given prime factorization

- Modulo-\( n \) exponentiation: \( (a^k) \mod n \)
  - Any \( x \) has a unique exponentiative inverse \( \mod n \) iff \( \gcd(x,n)=1 \).
  - Easy to compute?
    - For all \( x \) in \( \mathbb{Z}_n^* \): \( x^{\phi(n)} = 1 \mod n \). (Euler’s Theorem)
    - For all \( x \) in \( \mathbb{Z}_n^* \) and non-negative \( k \): \( x^{\phi(n)+1} = x \mod n \). (Variant)
    - For all \( x \) in \( \mathbb{Z}_n \) and non-negative integer \( k \): \( x^{\phi(n)+1} = x \mod n \)
      - 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
  o Choose two large primes, p and q (p and q remain secret).
  o Let \( n = p \cdot q \).
  o Choose a number \( e \) relatively prime to \( \phi(n) = (p-1) \cdot (q-1) \)
  o Public key is \( (e, n) \).
  o Find multiplicative inverse \( d \) of \( e \mod \phi(n) \) [i.e., \( e \cdot d = 1 \mod \phi(n) \)]
  o Private key is \( (d, n) \).

• Encryption/decryption
  o To encrypt message \( m \) using public key:
    - ciphertext \( c = m^e \mod n \)
  o To decrypt ciphertext \( c \) using private key:
    - plaintext \( m = c^d \mod n \).

• Signing/Verifying signature
  o To sign a message \( m \) using private key:
    - signature \( s = m^d \mod n \).
  o To verify signature \( c \) using public key:
    - plaintext \( m = c^e \mod n \).

Why does the decryption operation work, i.e., why is \( m^{ed} = m \)?

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

\[ m^{1 \mod \phi(n)} \] [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 \( m^{ed} \) 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 exponentiation

• 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 \( m^{2x+1} = m^x \cdot m \).

- Log \( e \) multiplications.

ModuloExponentiation( \( m \), \( e \), \( n \) )

\[(x_0, x_1, \ldots, x_k) \leftarrow e \text{ in binary}; \quad // \quad x_0 = 1\]

initially \( y \leftarrow m; \quad j \leftarrow 0; \quad // \quad y = m^0\)

while \( j < k \) do

\quad y \leftarrow y \cdot y \mod n; \quad // \quad y = m^{x_0 \cdot \ldots \cdot x_j}\)

\quad if \( x_{j+1} = 1 \) then \( y \leftarrow y \cdot m \mod n; \quad // \quad y = m^{x_0 \cdot \ldots \cdot x_j \cdot 1}\)

\quad j \leftarrow j + 1; \quad // \quad y = m^{x_0 \cdot \ldots \cdot x_j}\)

/ / \quad y = m^e \mod n\]

Example: \( 123^{54} \mod 678 \).

- \( 54 = (1101110)_2 \)

- \( 123 \mod 678 = 123 \)

- \( 123^{10} \mod 678 = 123 \cdot 123 \mod 678 = 15129 \mod 678 = 213 \)

- \( 123^{11} \mod 678 = 213 \cdot 123 \mod 678 = 26199 \mod 678 = 435 \)

- \( 123^{100} \mod 678 = 435 \cdot 435 \mod 678 = 1889225 \mod 678 = 63 \)

- \( 123^{1000} \mod 678 = 63 \cdot 63 \mod 678 = 3969 \mod 678 = 579 \)

- \( 123^{1001} \mod 678 = 579 \cdot 123 \mod 678 = 71217 \mod 678 = 27 \)

- \( 123^{1010} \mod 678 = 27 \cdot 27 \mod 678 = 729 \mod 678 = 51 \)

- \( 123^{1011} \mod 678 = 51 \cdot 123 \mod 678 = 6273 \mod 678 = 171 \)

- \( 123^{10110} \mod 678 = 171 \cdot 171 \mod 678 = 29241 \mod 678 = 87 \)
Generating RSA Keys consists of two parts:
- find big primes p and q
- finding e relatively prime to φ(n) (= (p-1)·(q-1))
  - d = e⁻¹ mod φ(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 < √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 aⁿ⁻¹ = 1 mod n;
  - Prob[test fails] is low (-10⁻¹² 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 φ(n)

Finding e (approach 2):
- Fix e such that mᵉ easy to compute (i.e., few 1's in binary)
- Choose primes p and q such that e relatively prime to (p−1)·(q−1)
- One choice: e=3  [so mᵉ needs 2 multiplications]
  - Need to pad small m.
    - If m < n¹/³ then mᵉ mod n = m³, so attacker can get m by (mⁿ)¹/³
  - Need to use different pads if m is sent to 3 principals with public keys (3,n₁), (3,n₂), (3,n₃).
    - Attacker has m³ mod n₁, m³ mod n₂, m³ mod n₃
    - CRT yields m³ mod n₁·n₂·n₃
  - Because m<n₁, m<n₂, m<n₃, attacker has m³ < n₁·n₂·n₃ and so (mⁿ mod n₁·n₂·n₃)¹/³ yields m.
- Another choice: e = 2¹⁶⁺¹ = 65537  [so mᵉ requires 17 multiplications]
  - No need for pad since unlikely that mⁿ₅₅₃₇ < 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  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 Sa, sends Ta = g⁵ᵃ mod p to B.
  - B chooses random 512-bit number Sb, sends Tb = g⁵ᵇ mod p to A.
  - A computes Tᵃᵇ mod p = g⁵ᵃ·⁵ᵇ mod p.
  - B computes Tᵇᵃ mod p = g⁵ᵇ·⁵ᵃ mod p.
  - A and B now share g⁵ᵃ·⁵ᵇ mod p, which can serve as a key.
  - Attacker knowing Ta and Tb and p and g cannot obtain g⁵ᵃ·⁵ᵇ mod p, because logarithm modulo-n is hard.
  - Does not provide authentication:
    A does not know whether it is talking to B or C.

<table>
<thead>
<tr>
<th>A sends [sender id A, g⁵ᵃ mod p]</th>
<th>C sends [sender id B, g⁵ᵇ mod p]</th>
</tr>
</thead>
<tbody>
<tr>
<td>A and C share secret g⁵ᵃ·⁵ᵇ mod p, but A thinks it is talking to B</td>
<td></td>
</tr>
</tbody>
</table>
Diffie-Helman with Published Numbers

• Assume PKI (public key infrastructure) that publishes for every principal \(X\): \((X, g, p, g^X \mod p)\)
• 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:
  o Encrypt Diffie-Helman exchange with pre-shared secret.
  o Encrypt Diffie-Helman exchange with other’s public key.
  o Sign Diffie-Helman value with your private key.
  o Following Diffie-Helman exchange, transmit hash of shared Diffie-Helman value, sender name, and pre-shared secret.
  o 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?
  o 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 \(g^X\))

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>C</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>send</td>
<td>([A, g^{SA}]) to B</td>
<td>alter msg to ([A, g^{SC}])</td>
<td></td>
</tr>
<tr>
<td></td>
<td>alter msg to ([B, g^{SC}])</td>
<td>send ([B, g^{SB}]) to A</td>
<td></td>
</tr>
<tr>
<td>--------</td>
<td>---------------------------------------------</td>
<td>------------------------------</td>
<td>---------------------------------------------</td>
</tr>
<tr>
<td>(&lt;---)</td>
<td>A and C share (g^{SC \cdot SA})</td>
<td>(\leq C) and B share (g^{SC \cdot SB}) (\rightarrow)</td>
<td></td>
</tr>
<tr>
<td>send</td>
<td>([g^{SC \cdot SA}, pw_{AB}])</td>
<td>decrypt with (g^{SC \cdot SA}), alter to</td>
<td>(g^{SC \cdot SB}, pw_{AB}])</td>
</tr>
<tr>
<td></td>
<td>decrypt using (g^{SC \cdot SB})</td>
<td>decrypt using (g^{SC \cdot SB})</td>
<td>A authenticated (error)</td>
</tr>
</tbody>
</table>

Zero-knowledge proof systems

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

Classic example is based on graph isomorphism

• “Key” generation
  o A chooses a large graph (eg, 500 vertices) \(G_{A1}\).
  o A renames the vertices to produce an isomorphic graph \(G_{A2}\).
  o Graphs \(G_{A1}\) and \(G_{A2}\) are A’s “public key”.
  o The vertex renaming transforming \(G_{A1}\) to \(G_{A2}\) is A’s “private key”.
• A authenticates to B as follows:
  o A sends B a new set of graphs \(\{G_1, \ldots, G_k\}\), each isomorphic to \(G_{A1}\).
  o B randomly divides the graphs into subset 1 and subset 2.
  o 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 \(G_{A2}\)
  o A supplies the vertex-renamings, thereby authenticating itself.
• Why does it work?
  o Graph isomorphism is a hard problem: knowing a renaming to \( G_1 \) does not help obtain a renaming to \( G_2 \).
  o So renamings could only have been generated by A originally.
  o 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
  o A’s private key: a large random number \( s \)
  o 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
  o A chooses \( k \) random numbers, \( r_1, \ldots, r_k \)
  o A sends \( r_1^2 \mod n, \ldots, r_k^2 \mod n \), to B
  o B randomly splits these into subset 1 and subset 2, and informs A
  o 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
  o B checks whether
    • for each entry in subset 1: \((\text{reply}_i)^2 = v \cdot r_i^2 \mod n\)
    • for each entry in subset 2: \((\text{reply}_i)^2 = r_i^4 \mod n\)
  o If so, A is authenticated

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.
  o Suppose hash function chosen provides \( k \)-bit hash (e.g., \( k=128 \)).
  o A chooses \( k \) random numbers, \( r_1, \ldots, r_k \)
  o A forms msg [data to be signed | \( r_1^2 \mod n, \ldots, r_k^2 \mod n \)]
  o 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
  o Why does it work?
    • Forging a signature on a message requires having both possible replies for all the \( r_i \)’s.