Keeping communications secret

- **Encryption guarantees secrecy**

- **Symmetric encryption**
  - Encryption algorithm consists of two functions $E$ and $D$
  - To communicate secretly, parties share secret key $K$
  - Given message $M$, $E(K, M) \rightarrow C$, $D(K, C) \rightarrow M$
  - $M$ is plaintext, $C$ is ciphertext
  - Attacker cannot derive $M$ from $C$ without $K$
Types of encryption

- **One time pad algorithm**
  - Share a completely random string $P$
  - Encrypt $M$ by XORing with $P$: $C ← M ⊕ P$
  - Decrypt by XORing again: $M ← C ⊕ P$

- **Stream ciphers – pseudo-random pad**
  - Generate pseudo-random stream of bits from short key
  - Encrypt/decrypt by XORing as with one-time pad
  - But **NOT** one-time PAD! (People who claim so are frauds!)

- **Most common algorithm type: Block cipher**
  - Operates on fixed-size blocks (e.g., 64 or 128 bits)
  - Maps plaintext blocks to same size ciphertext blocks
Example stream cipher (RC4)

- **Initialization:**
  - $S[0 \ldots 255] \leftarrow$ permutation $\langle 0, \ldots, 255, \rangle$
    (based on key—specifics omitted)
  - $i \leftarrow 0; j \leftarrow 0$

- **Generating pseudo-random bytes:**
  
  $i \leftarrow (i + 1) \mod 256; \quad j \leftarrow (j + S[i]) \mod 256; \quad \text{swap } S[i] \leftrightarrow S[j];$
  
  $t \leftarrow (S[i] + S[j]) \mod 256; \quad \text{return } S[t];$
Example use of stream cipher

- Pre-arrange to share secret $s$ with web vendor
- Exchange payment information as follows
  - Send: $\text{Encrypt}(s, \text{“Visa card #3273…”})$
  - Receive: $\text{Encrypt}(s, \text{“Order confirmed, have a nice day”})$
- Now an eavesdropper can’t figure out your Visa #
Wrong!

• Let’s say an attacker has the following:
  - $c_1 = \text{Encrypt}(s, \text{“Visa card #3273…”})$
  - $c_2 = \text{Encrypt}(s, \text{“Order confirmed, have a nice day”})$

• Now compute:
  - $m \leftarrow c_1 \oplus c_2 \oplus \text{“Order confirmed, have a nice day”}$

• Lesson: Never re-use keys with a stream cipher
  - Similar lesson applies to one-time pads
    (That’s why they’re called one-time pads.)
Example block cipher (blowfish)

- Derive $F$ and 18 subkeys from Key—$P_1 \ldots P_{18}$
- Divide plaintext block into two halves, $L_0$ and $R_0$
  
  - $R_i = L_{i-1} \oplus P_i$
  - $L_i = R_{i-1} \oplus F(R_i)$
  - $R_{17} = L_{16} \oplus P_{17}$
  - $L_{17} = R_{16} \oplus P_{18}$
  - Output $L_{17}R_{17}$.

(Note: This is just to give an idea; it’s not a complete description)
Using a block cipher

• In practice, message may be more than one block

• Encrypt with ECB (electronic code book) mode:
  - Split plaintext into blocks, and encrypt separately
  - Attacker can’t decrypt any of the blocks
  - Message is secure

• Distributing a secret key can be expensive
  - Want to maximize the use of a shared secret key
  - Can encrypt multiple messages, since each is secure
  - This is okay, since it’s not a stream cipher
Wrong!

- Attacker will learn of repeated plaintext blocks
  - If transmitting sparse file, will know where non-zero regions lie

- Example: Intercepting military instructions
  - Most days, send encryption of “nothing to report.”
  - On eve of battle, send “attack at dawn.”
  - Attacker will know when battle plans are being made

- Solution: Cipher-block chaining
  - Ensures repeated blocks are not encrypted the same
Given a shared key, can you transmit files securely over the Internet if you encrypt them in CBC mode?
Problem: Integrity

- Attacker can tamper with messages
  - E.g., corrupt a block to flip a bit in next

- What if you delete original file after transfer?
  - Might have nothing but garbage at recipient

- Encryption does not guarantee integrity
  - A system that uses encryption alone (no integrity check) is often incorrectly designed.
  - Exception: Cryptographic storage (to protect disk if stolen)
Message authentication codes

• **Message authentication codes (MACs)**
  - Sender & receiver share secret key $K$
  - On message $m$, $\text{MAC}(K, m) \rightarrow v$
  - Attacker cannot produce valid $\langle m, v \rangle$ without $K$

• **To send message securely, append MAC**
  - Send $\{m, \text{MAC}(K, m)\}$, where $m$ could be ciphertext, $E(K', M)$
  - Receiver of $\{m, v\}$ checks $v \overset{?}{=} \text{MAC}(K, m)$

• **Careful of Replay – don’t believe previous $\{m, v\}$**
Cryptographic hashes

- Hash arbitrary-length input to fixed-size output
  - Typical output size 128 or 160 bits
  - Cheap to compute on large input (faster than network)

- Collision-resistant: Computationally infeasible to find \( x \neq y, \ H(x) = H(y) \)
  - Many such collisions exist
  - No one has been able to find one, even after analyzing the algorithm

- Several hashes in common use (SHA-1, MD5)
Applications of cryptographic hashes

- Small hash uniquely specifies large data
  - Hash a file, remember the hash value
  - Recompute hash later, if same value no tampering
  - Hashes often published for software distribution

- $\text{HMAC}(K, m) = H(K \oplus \text{opad}, H(K \oplus \text{ipad}, m))$
  - $H$ is a cryptographic hash like SHA-1
  - ipad is 0x36 repeated 64 times, opad 0x5c repeated 64 times
Order of Encryption and MACs

• Should you Encrypt then MAC, or vice versa?
• MACing encrypted data is always secure
• Encrypting Data+MAC may not be secure!
  - Consider the following secure, but stupid encryption alg
  - Transform $m \rightarrow m'$ by mapping each bit to two bits:
    Map $0 \rightarrow 00$ (always), $1 \rightarrow \{10, 01\}$ (randomly pick one)
  - Now encrypt $m'$ with a stream cipher to produce $c$
  - Attacker flips two bits of $c$—if msg rejected, was 0 bit in $m$
Public key encryption

- Three randomized algorithms:
  - *Generate* – $G(1^k) \rightarrow K, K^{-1}$
  - *Encrypt* – $E(K, m) \rightarrow \{m\}_K$
  - *Decrypt* – $D(K^{-1}, \{m\}_K) \rightarrow m$

- Provides secrecy, like conventional encryption
  - Can’t derive $m$ from $\{m\}_K$ without knowing $K^{-1}$

- Encryption key $K$ can be made public
  - Can’t derive $K^{-1}$ from $K$
  - Everyone can use the same public key to encrypt messages for one recipient.
The RSA algorithm

- **Generation:**
  - Pick two primes, $p$ and $q$, let $N = pq$
  - Pick random $e$ that does not divide $(p - 1)(q - 1)$
  - Compute $d$ such that $de \equiv 1 \pmod{(p - 1)(q - 1)}$
  - Public key: $N, e$, private key $N, d$

- **Facts:**
  - If $m \in \mathbb{Z}_N^*$, then $(m^e \mod N)^d \mod N = m$.
  - For large enough $p, q$ and random $m$, Given $N, e$, and $m^e \mod N$, No one knows how to find $m$ if they don’t already know $p, q, or d$.

- To encrypt a message, just treat bits as number and **computer** $m^e \mod N$. 
Wrong!

• What if message is from a small set (yes/no)?

• What if I want to outbid you in secret auction?
  - I take your encrypted bid $c$ and submit $c(11/10)^e \mod n$.

• What if there’s some protocol in which I can learn other message decryptions?
  - E.g., people escrow ciphertexts, and get them back under certain circumstances (if an employee is fired or dies)
  - I take your ciphertext $c = m^e \mod n$, and escrow $c2^e \mod n$.
  - After I’m fired, my coconspirator gets back $2m$

• Many people make this mistake, including SSL
  - SSL didn’t return decryptions, but error messages had some information
Notions of security

- How do design systems using RSA?
  - You don’t want to think about interactions between your error messages, modular exponentiation, and lattice theory.

- A PKS is **adaptive chosen ciphertext secure** if
  - No attacker $A$ can win the following game with probability more than $1/2 + \text{negligible}$:
    - $A$ can first ask for arbitrary messages to be decrypted
    - $A$ then produces two messages, $m_0$ and $m_1$
    - The good guy flips a coin $b \leftarrow \{0, 1\}$, returns $c = E(K, m_b)$.
    - $A$ can ask for any messages except $c$ to be decrypted
    - $A$ guesses the value of $b$
Practical solution: OAEP+ (Shoup)

- Transforms plaintext $M$ into number $M'$ for RSA:

$$M \xrightarrow{H'} G \xrightarrow{H} M' =$$

- Not provable, but heuristically secure
Digital signatures

• Three (randomized) algorithms:
  - Generate – $G(1^k) \rightarrow K, K^{-1}$
  - Sign – $S(K^{-1}, m) \rightarrow \{m\}_{K^{-1}}$
  - Verify – $V(K, \{m\}_{K^{-1}}, m) \rightarrow \{\text{true, false}\}$

• Provides integrity, like a MAC
  - Cannot produce valid $\langle m, \{m\}_{K^{-1}} \rangle$ pair without $K^{-1}$

• Many keys support both signing & encryption
  - But Encrypt/Decrypt and Sign/Verify different algorithms!
  - Common error: Sign by “encrypting” with private key
## Cost of cryptographic operations

<table>
<thead>
<tr>
<th>Operation</th>
<th>msec</th>
</tr>
</thead>
<tbody>
<tr>
<td>Encrypt</td>
<td>0.18</td>
</tr>
<tr>
<td>Decrypt</td>
<td>6.60</td>
</tr>
<tr>
<td>Sign</td>
<td>6.71</td>
</tr>
<tr>
<td>Verify</td>
<td>0.03</td>
</tr>
</tbody>
</table>

[1,280-bit Rabin-Williams keys on 3 GHz Pentium IV]

- **Cost of public key algorithms significant**
  - Encryption only on small messages (< size of key)
  - Signature cost relatively insensitive to message size

- **In contrast, symmetric algorithms much cheaper**
  - Symmetric can encrypt+MAC faster than 100Mbit/sec LAN
Hybrid schemes

• Use public key to encrypt symmetric key
  - Send message symmetrically encrypted: \( \{ \text{msg} \}_{K_S}, \{K_S\}_{K_P} \)

• Use PK to negotiate secret session key
  - E.g., Client sends server \( \{K_1, K_2, K_3, K_4\}_{K_P} \)
  - Client sends server: \( \{\{m_1\}_{K_1}, \text{MAC}(K_2, \{m_1\}_{K_1})\}\)  
  - Server sends client: \( \{\{m_2\}_{K_3}, \text{MAC}(K_4, \{m_2\}_{K_3})\}\)

• Often want mutual authentication (client & server)
  - Or more complex, user(s), client, & server
Server authentication

• An approach: Use public key cryptography
  - Give client public key of server
  - Lets client authenticate secure channel to server

• Problem: Key management problem
  - How to get server’s public key?
  - How to know the key is really server’s?
The danger: Attackers impersonating servers

- File system example:
  - Attacker pretends to be server, gives its own public key
  - Attacker substitutes modified data for file
  - User writes sensitive file to fake server
Man in the middle attacks

- Attacker might not look like server
  - User would notice if file system didn’t contain right files

- Man in the middle attack foils user:
  - Attacker emulates server when talking to client
  - Attacker emulates client when talking to server
  - Attacker passes most messages through unmodified
  - Attacker substitutes own public key for client’s & server’s
  - Attacker records secret data, or tampers to cause damage
Key management

• Put public keys in the phone book
  - How do you know you have the real phone book?
  - How is a program supposed to use phone book
    www.phonebook.com? (are you talking to real web server)

• Exchange keys with people in person

• “Web of trust” – get keys from friends you trust
Certification authorities

- Everybody trusts some certification authority
- Everybody knows authority's public key
  - E.g., built into web browser
Hierarchy with local trust

• To get from cs.nyu.edu to mit.edu:
  - cs.nyu.edu knows key for nyu.edu
  - nyu.edu knows key for edu/root
  - root knows key for mit.edu

• To get within cs.nyu.edu:
  - No need to trust outside authorities
Passwords

• Many systems grant access through a password

• How to implement? Example:
  - Server stores user’s password
  - Client connects to server, sends username, password
  - Server compares password to stored version
  - Grants access if they match

• Is this a good approach?
Weaknesses

• How do you know you are talking only to server
  - Attacker might be eavesdropping on Ethernet
  - Attacker might mess with DNS (or IP routing) so you talk to wrong machine

• Eavesdropper will just read your password off the network

• Server knows your plaintext password
  - Attacker or server operator can modify login record it
  - Bad because people re-use passwords on multiple machines
s/key password authentication

- **Goal:** Protect against passive eavesdroppers
  - Also: Minimize harm of bad clients (e.g., public terminals)

- **Idea:** One time passwords, not valid after snooped

- **Algorithm** takes user's real password, $p$, a random "salt" $s$, and server machine name $m$
  - First one time password is $H^{(100)}(p, m, s)$ (for one way hash $H$)
  - Next password is $H^{(99)}(p, s)$, etc.
  - After 99 logins, must change salt or password

- **Benefit:** Very convenient
  - Carry list of one-time passwords, calculate on palm pilot
Weaknesses

- **Still vulnerable to man-in-the-middle**
  - Attacker impersonates server
  - Re-sends one-time password to real server

- **Vulnerable to off-line password guessing**
  - Attacker sees \( s \) and \( H^{(n)}(p, s) \)
  - Can verify guesses of \( p \) off-line – check against dictionary of common passwords
  - \( H \) not very expensive (should be \( H^{(n)}(G(p, s)) \) for expensive \( G \)—or maybe users should crank \( n \))

- **Bad client can compromise session**
  - Before logout, can insert command to create back door:
    
    ```bash
echo 1024 35 145...3 badguy >> .ssh/authorized_keys
    ```
Password-derived public keys

- Derive public key from user’s password
  - E.g., use password as seed for pseudo-random generator
  - Client can regenerate private key given password

- Server stores public key for each user

- Use public key authentication, E.g.:
  - $S \rightarrow C : N_S$
  - $C \rightarrow S : K_u, \{C, S, K_S, u, session, N_S\}_{K_u^{-1}}$
  - $K_S$ is server public key, used to authenticate server
  - Server checks sig, looks up pubkey $K_u$ for credentials
Weaknesses

• No salt
  - Users with same password will have same public key

• No cost parameter
  - Can’t take too long to log in
  - But over time generating key will get faster

• Public key is just like password hash
  - Eavesdropper will see key, can mount off-line attack

• No authentication of server to user
  - W/o accessing server, attacker can pretend server is giving bad answers
MAC of server public key

- User chooses password $p$. Server stores:
  - $h \leftarrow H(p, \text{servername})$ (where $H$ is hash with parameterizable cost)

- Server authenticates its public key to user with secret $h$
  - $S \rightarrow C : \text{MAC}(h, K_S)$
  - Client can establish secure channel using $K_S$
Weaknesses

- Still vulnerable to offline password guessing
- No forward secrecy
  - Attacker can record encrypted communications
  - One year later, breaks into server, gets private key
  - Can now decrypt last year’s messages!
Secure password protocols

• **Purpose of secure password protocol:**
  - Generate strong shared secret from user-chosen password
  - Use secret as key for mutually authenticated session

• **Goals of the SRP protocol**
  - Network attacker should be unable learn anything that will allow an off-line guessing attack
  - Forward secrecy
  - Server knows nothing plaintext-equivalent

• **Warning:** SRP has not been proven correct
  - No known bugs in SRP, but provably correct protocols exist
  - What follows is an *informal* argument about why SRP might be secure
Background: Diffie-Hellman key exchange

- An unauthenticated key exchange protocol
  - Provides forward secrecy against a passive adversary

- The discrete log problem:
  - Let $p$ be a prime and $g$ a generator of $\mathbb{Z}_p^*$
  - Given $x \in \mathbb{Z}_{p-1}$, easy to compute $g^x$ from $x$, inverse hard

- Diffie-Hellman protocol, given $g$ and $p$
  - $A$ picks random $x$, sends $B g^x$
  - $B$ picks random $y$, sends $A g^y$
  - $A$ and $B$ use $H(g^{xy})$ as session key

- Note: Breaking DH may be easier than discrete log
SRP protocol

- User knows $p$, Server knows $s, g^{H(s,p)}$
- Get parameters. $U \rightarrow S : U; S \rightarrow U : s, N, g$
- User picks $a$. $U \rightarrow S : U, g^a$
- Server picks $b, u$. $S \rightarrow U : s, 3g^{H(s,p)} + g^b, u$
- User computes: $K = H \left( \left( g^b \right)^{a + uH(s,p)} \right)$
- Server computes: $K = H \left( g^a \left( g^{H(s,p)} \right)^u \right)^b$
- In either case: $K = H \left( g^{ab} g^{buH(s,p)} \right)$
Informal analysis

• No obvious way for $U$ to get $K$ without $p$

• $S$ doesn’t know $p$
  - Even if $S$’s secret $g^{H(s,p)}$ stolen, thief can’t impersonate user without mounting off-line guessing attack

• No obvious way to mount off-line guessing attack
  - Suppose attacker impersonates user, then learns $K$
    Will know $a, s, u, 3g^{H(s,p)} + g^b, \text{ and } H(g^{ab}g^{b H(s,p)})$
    No obvious way to validate guess of $p$
  - Suppose attacker impersonates server and learns $K$
    Will know $s, B, g^a, u, \text{ and } H((B - 3g^{H(s,p)})^{a+u H(s,p)})$
    No obvious way to validate guess of $p$
Secure password protocols in practice

• **Passwords are sufficient for mutual authentication**
  - A distributed system in which a user types a password should never be susceptible to a man in the middle attack!
  - If user establishes a password in person, no need for PKI to contact server

• **Consider adding cost parameter $c$ to algorithms**
  - For example hash password $2^c$ times
  - $U \rightarrow S : U, g^a$
  - $S \rightarrow U : s, c, g^{H^2}(s, p) + g^b, u$