CSCI 454/554 Computer and Network Security

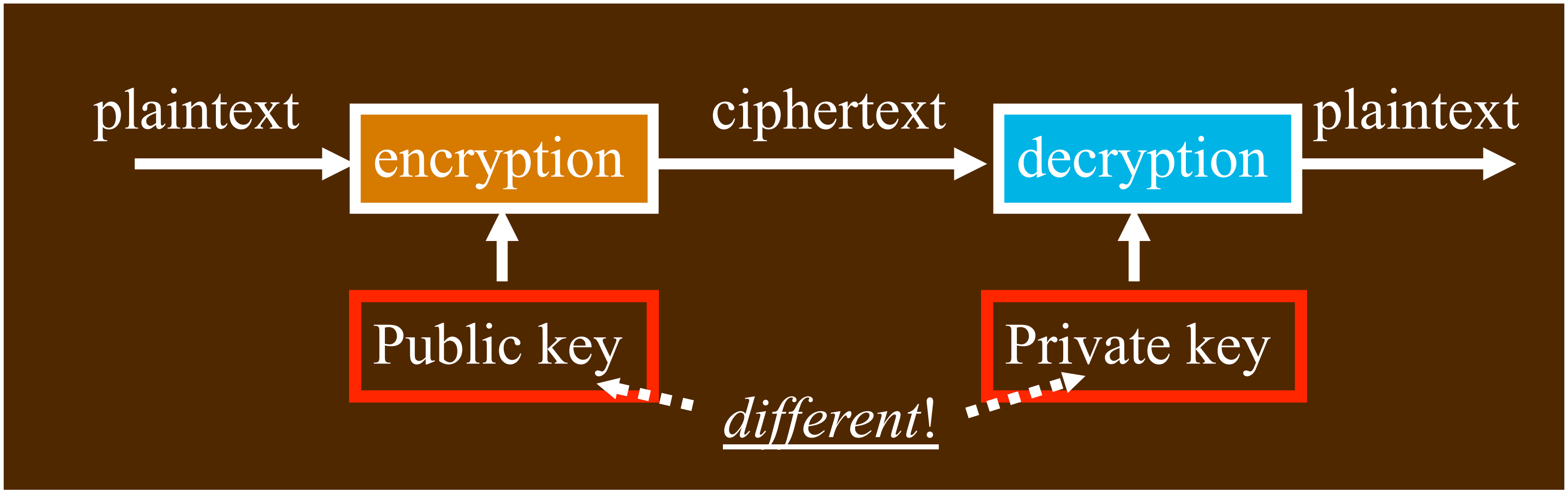
Topic 5.2 Public Key Cryptography

Outline

1. Introduction
2. RSA
3. Diffie-Hellman Key Exchange
4. Digital Signature Standard

Introduction

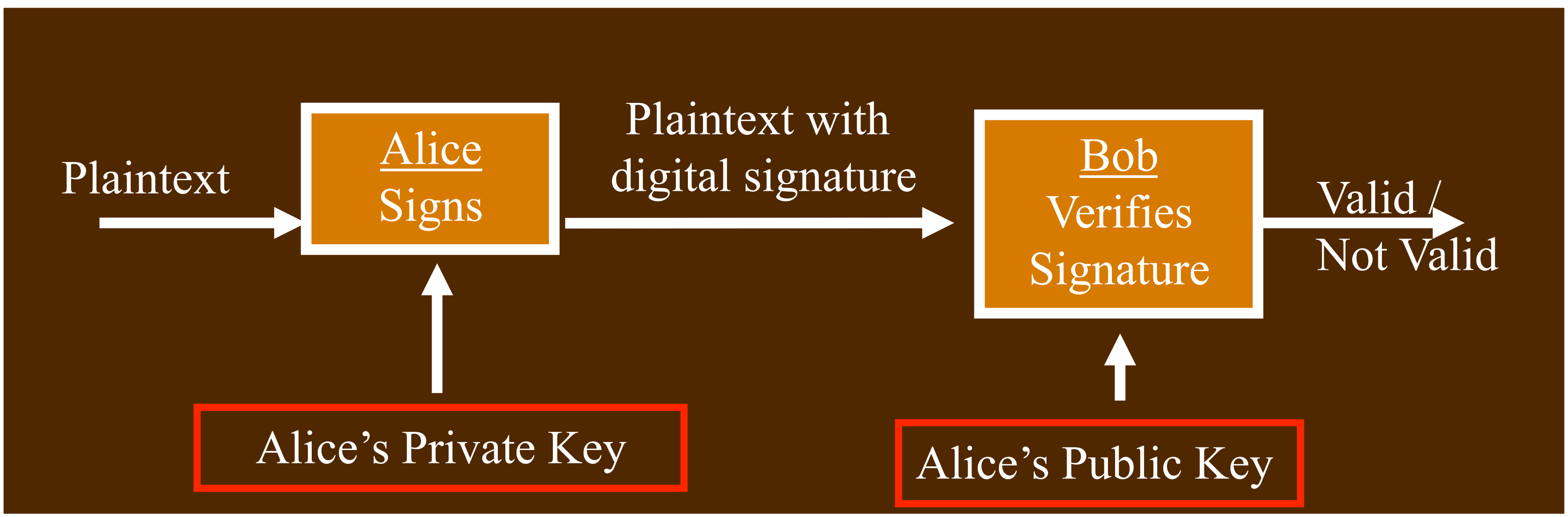
Public Key Cryptography



- Invented and published in 1975
- A *public / private key pair* is used
  - public key can be announced to everyone
  - private key is kept secret by the owner of the key
- Also known as *asymmetric* cryptography
- Much slower to compute than secret key cryptography

Applications of Public Key Crypto

1. Message integrity with *digital signatures*
   Alice computes hash, signs with her private key (no one else can do this without her key)
   Bob verifies hash on receipt using Alice's public key using the verification equation

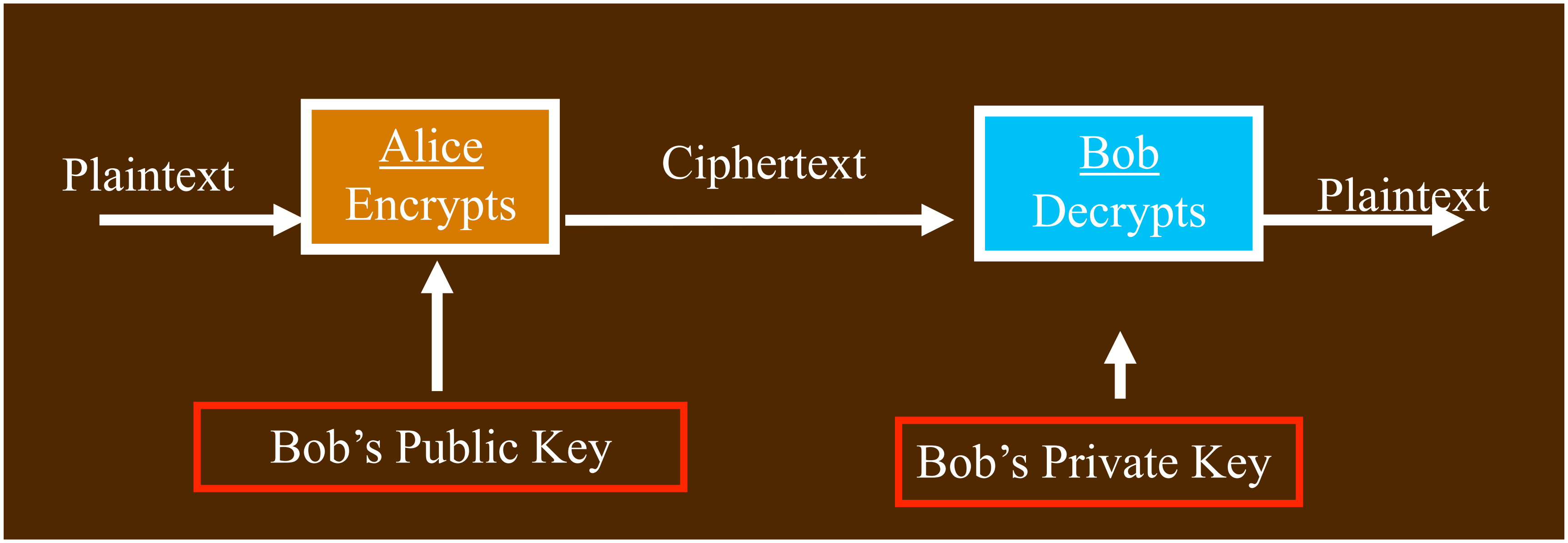


Applications (Cont'd)

- The digital signature is verifiable by anybody
- Only one person can sign the message: *non-repudiation*
  - Non-repudiation is only achievable with public key cryptography

Applications (Cont'd)

2. Communicating securely over an insecure channel
   - Alice encrypts plaintext using Bob's public key, and Bob decrypts ciphertext using his private key
   - No one else can decrypt the message (because they don't have Bob's private key)



Applications (Cont'd)

WILLIAM & MARY

3. Secure storage on insecure medium
   - Alice encrypts data using her public key
   - Alice can decrypt later using her private key
4. *User Authentication*
   - Bob proves his identity to Alice by using his private key to perform an operation (without divulging his private key)
   - Alice verifies result using Bob's public key

Applications (Cont'd)

5. Key exchange for secret key crypto
   - Alice and Bob use public key crypto to negotiate a shared secret key between them

Public Key Algorithms

- Public key algorithms covered in this class, and their applications

System	Encryption / Decryption?	Digital Signatures?	Key Exchange?
RSA	Yes	Yes	Yes
Diffie-Hellman			Yes
DSA		Yes	

Public-Key Requirements

- It must be computationally
  - easy to generate a public / private key pair
  - hard to determine the private key, given the public key
- It must be computationally
  - easy to encrypt using the public key
  - easy to decrypt using the private key
  - hard to recover the plaintext message from just the ciphertext and the public key

Trapdoor One-Way Functions

- *Trapdoor* one-way function
  - $Y=f_k(X)$: easy to compute if k and X are known
  - $X=f^{-1}_k(Y)$: easy to compute if k and Y are known
  - $X=f^{-1}_k(Y)$: hard if Y is known but k is unknown
- Goal of designing public-key algorithm is to find appropriate trapdoor one-way function

The RSA Cipher

RSA (Rivest, Shamir, Adleman)

WILLIAM & MARY

- The most popular public key method
  - provides both public key encryption and digital signatures
- Basis: factorization of large numbers is hard
- Variable key length (1024 bits or greater)
- Variable plaintext block size
  - plaintext block size must be smaller than key size
  - ciphertext block size is same as key size

Generating a Public/Private Key Pair

- Find (using Miller-Rabin) large primes p and q
- Let $n = p*q$
  - do not disclose p and q!
  - $\phi(n) = ???$
- Choose an e that is relatively prime to $\phi(n)$
  - **public** key = $<e,n>$
- Find d = multiplicative inverse of e mod $\phi(n)$ (i.e., $e*d = 1 \bmod \phi(n)$)
  - **private** key = $<d,n>$

RSA Operations

- For plaintext message $\boldsymbol{m}$ and ciphertext $\boldsymbol{c}$

Encryption: $\boldsymbol{c = m^e \bmod n}$, $m < n$

Decryption: $\boldsymbol{m = c^d \bmod n}$

Signing: $\boldsymbol{s = m^d \bmod n}$, $m < n$

Verification: $\boldsymbol{m = s^e \bmod n}$

RSA Example: Encryption and Signing

WILLIAM & MARY

- Choose p = 23, q = 11 (both primes)
  - $n = p*q = 253$
  - $\phi(n) = (p\text{-}1)(q\text{-}1) = 220$
- Choose e = **39** (relatively prime to 220)
  - public key = <**39**, 253>
- Find e^{-1} mod 220 = d = **79**
  (note: $39*79 \equiv 1 \bmod 220$)
  - private key = <**79**, 253>

Example (Cont'd)

- Suppose plaintext **m = 80**

Encryption

$\mathbf{c} = 80^{39} \bmod 253 =$ ____ $\quad (c = m^e \bmod n)$

Decryption

$\mathbf{m} =$ ____ $^{79} \bmod 253 = \mathbf{80} \quad (c^d \bmod n)$

Signing (in this case, for entire message **m**)

$\mathbf{s} = \mathbf{80}^{79} \bmod 253 =$ ____ $\quad (s = m^d \bmod n)$

Verification

$\mathbf{m} =$ ____ $^{39} \bmod 253 = \mathbf{80} \quad (s^e \bmod n)$

Example (Cont'd)

- Suppose plaintext **m = 80**

Encryption

c = 80^{39} mod 253 = **37** ($c = m^e \bmod n$)

Decryption

m = 37^{79} mod 253 = **80** ($c^d \bmod n$)

Signing (in this case, for entire message **m**)

s = **80**^{79} mod 253 = 224 ($s = m^d \bmod n$)

Verification

m = 224^{39} mod 253 = **80** ($s^e \bmod n$)

Using RSA for Key Negotiation

- Procedure
  1. *A* sends random number *R*1 to *B*, encrypted with *B*'s public key
  2. *B* sends random number *R*2 to *A*, encrypted with *A*'s public key
  3. *A* and *B* both decrypt received messages using their respective private keys
  4. *A* and *B* both compute K = H($R1 \oplus R2$), and use that as the shared key

Key Negotiation Example

- For Alice, $e = 39$, $d = 79$, $n = 253$
- For Bob, $e = 23$, $d = 47$, $n = 589$ (=19*31)
- Let **R1 = 15**, **R2 = 55**

1. Alice sends **306** = $\mathbf{15}^{23}$ mod 589 to Bob
2. Bob sends **187** = $\mathbf{55}^{39}$ mod 253 to Alice
3. Alice computes R2 = **55** = $\mathbf{187}^{79}$ mod 253
4. Bob computes R1 = **15** = $\mathbf{306}^{47}$ mod 589
5. A and B both compute $K = H(R1 \oplus R2)$, and use that as the shared key

Proof of Correctness ($D(E(m)) = m$)

- Given
  - public key = $<e, n>$ and private key = $<d, n>$
  - $n = p*q$, $\phi(n) = (p-1)(q-1)$
  - $e*d \equiv 1 \bmod \phi(n)$
- If encryption is $c = m^e \bmod n$, decryption…
  - $= c^d \bmod n$
  - $= (m^e)^d \bmod n = m^{ed} \bmod n = m^{ed \bmod \phi(n)} \bmod n$
  - $= m \bmod n$ (why?)
  - $= m$ (since $m < n$)
- (digital signature proof is similar)

Is RSA Secure?

- $<e,n>$ is public information
- If you could factor n into $p*q$, then
  - could compute $\phi(n) = (p-1)(q-1)$
  - could compute $d = e^{-1} \bmod \phi(n)$
  - would know the private key $<d,n>$!
- But: factoring large integers is hard!
  - classical problem worked on for centuries; no known reliable, fast method

Security (Cont'd)

- At present, key sizes of 1024 bits are considered to be secure, but 2048 bits is better
- Tips for making n difficult to factor
  1. p and q lengths should be similar (ex.: ~500 bits each if key is 1024 bits)
  2. both (p-1) and (q-1) should contain a "large" prime factor
  3. gcd(p-1, q-1) should be "small"
  4. d should be larger than $n^{1/4}$

Attacks Against RSA

- Brute force: try all possible private keys
  - can be defeated by using a large enough key space (e.g., 1024 bit keys or larger)
- Mathematical attacks
  1. factor *n* (possible for special cases of n)
  2. determine *d* directly from *e,* without computing $\phi(n)$
     - at least as difficult as factoring *n*

Attacks (Cont'd)

- Probable-message attack (using <*e*,*n*>)
  - encrypt all possible plaintext messages
  - try to find a match between the ciphertext and one of the encrypted messages
  - only works for small plaintext message sizes
- Solution: pad plaintext message with random text before encryption
- PKCS #1 v1 specifies this padding format:

00	02	R1	R2	R3	R4	R5	R6	R7	R8	00	data…

each 8 bits long

Timing Attacks Against RSA

- Recovers the private key from the running time of the decryption algorithm
- Computing $m = c^d \bmod n$ using repeated squaring algorithm:

```
m = 1;
for i = k-1 downto 1
  m = m*m mod n;
  if d_i == 1
      then m = m*c mod n;
return m;
```

Timing Attacks (Cont'd)

- The attack proceeds bit by bit
- Attacker assumed to know **c**, **m**
- Attacker is able to determine bit *i* of *d* because for some ***c*** and ***m***, the highlighted step is extremely slow if $d_i = 1$

Countermeasures to Timing Attacks

1. Delay the result if the computation is too fast
   - disadvantage: ?
2. Add a random delay
   - disadvantage?
3. *Blinding:* multiply the ciphertext by a random number before performing decryption

RSA's Blinding Algorithm

- To confound timing attacks during decryption
  1. generate a random number r between 0 and n–1 such that $\gcd(r, n) = 1$
  2. compute $\mathbf{c'} = \mathbf{c} * r^e \bmod n$
  3. compute $\mathbf{m'} = (\mathbf{c'})^d \bmod n$
  4. compute $\mathbf{m} = \mathbf{m'} * r^{-1} \bmod n$

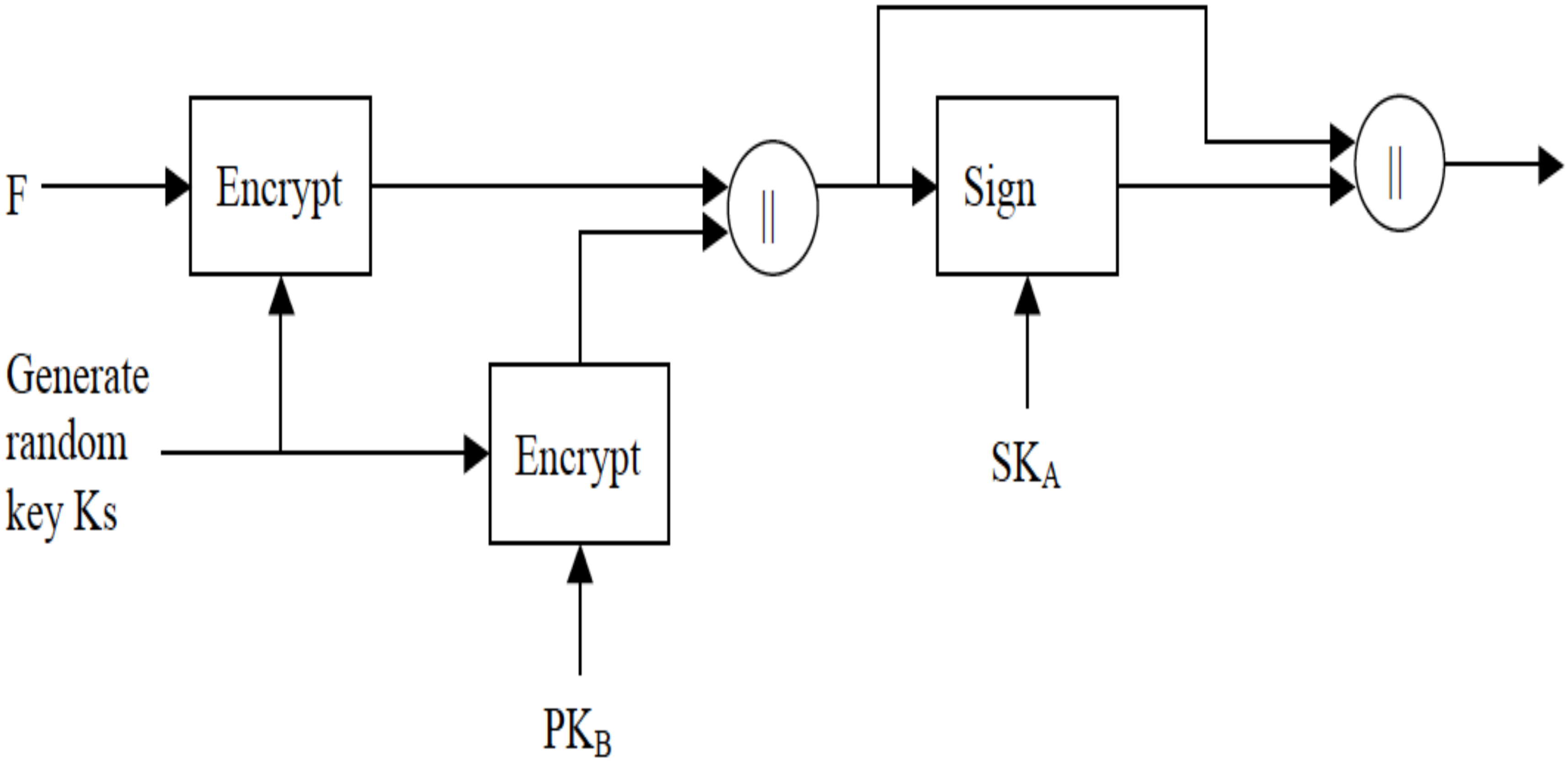
this is where timing attack would occur

- Attacker will not know what the bits of **c'** are
- Performance penalty: < 10% slowdown in decryption speed

File Encryption and Authentication

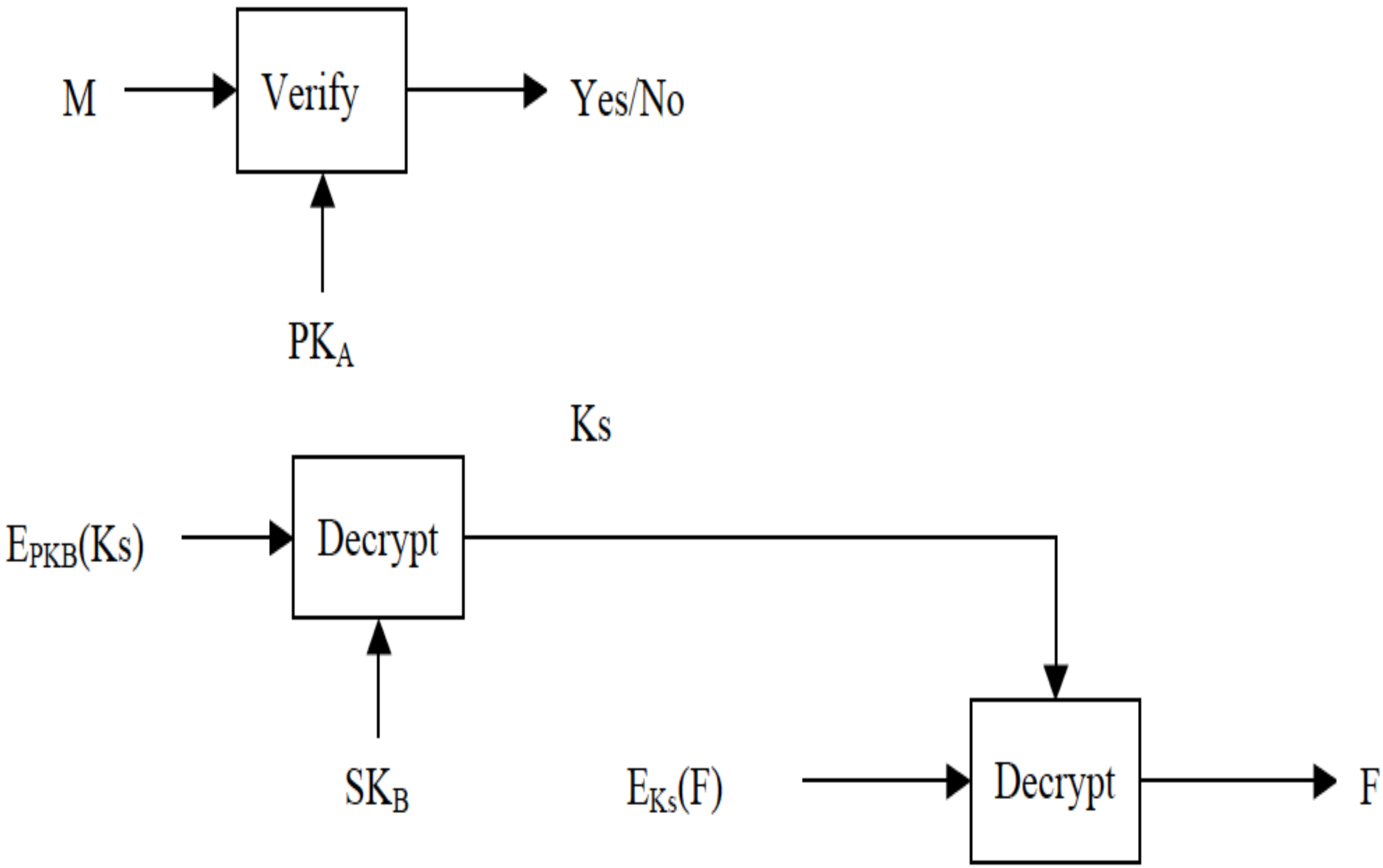
- Alice sends a large file to Bob without disclosing the content of the file to anybody else.
- Also make sure no other people can modify the message without being noticed.
- Conditions:
  - No secret key shared between Alice and Bob.
  - Alice and Bob know each other's RSA public key. ($SK_{A,}$ PK_A) and ($SK_{B,}$ PK_B)

Sender



$$\mathbf{M = E_{Ks}(F) \parallel E_{PKB}(Ks) \parallel Sig_{SKA}\,(E_{Ks}(F) \parallel E_{PKB}(Ks))}.$$

Receiver



Diffie-Hellman Key Exchange

Diffie-Hellman Protocol

- For negotiating a shared secret key using only public communication
- Does not provide authentication of communicating parties
- What's involved?
  - p is a large prime number (about 512 bits)
  - g is a primitive root of p, and $g < p$
  - p and g are publicly known

D-H Key Exchange Protocol

<u>Alice</u>	<u>Bob</u>
Publishes or sends g and p	Reads g and p
Picks random number S_A *(and keeps private)*	Picks random number S_B *(and keeps private)*
Computes public key $T_A = g^{S_A} \bmod p$	Computes public key $T_B = g^{S_B} \bmod p$
Sends T_A to Bob, reads T_B from Bob	Sends T_B to Alice, reads T_A from Alice
Computes $T_B^{S_A} \bmod p$ =	Computes $T_A^{S_B} \bmod p$

Key Exchange (Cont'd)

WILLIAM & MARY

- Alice and Bob have now both computed the same secret $g^{S_A S_B}$ mod p, which can then be used as the shared secret key K

- S_A is the discrete logarithm of g^{S_A} mod p and S_B is the discrete logarithm of g^{S_B} mod p

D-H Example

- Let p = 353, g = 3
- Let random numbers be S_A = 97, S_B = 233
- Alice computes T_A = ___ mod __ = 40 = g^{S_A} mod p
- Bob computes T_B = ___ mod ___ = 248 = g^{S_B} mod p
- They exchange T_A and T_B
- Alice computes K = __ mod __ = **160** = $T_B^{S_A}$ mod p
- Bob computes K = __ mod ___ = **160** = $T_A^{S_B}$ mod p

D-H Example

- Let $p = 353$, $g = 3$
- Let random numbers be $S_A = 97$, $S_B = 233$
- Alice computes $T_A = 3^{97} \bmod 353 = 40 = g^{S_A} \bmod p$
- Bob computes $T_B = 3^{233} \bmod 353 = 248 = g^{S_B} \bmod p$
- They exchange T_A and T_B
- Alice computes $K = 248^{97} \bmod 353 = \mathbf{160} = T_B^{S_A} \bmod p$
- Bob computes $K = 40^{233} \bmod 353 = \mathbf{160} = T_A^{S_B} \bmod p$

Why is This Secure?

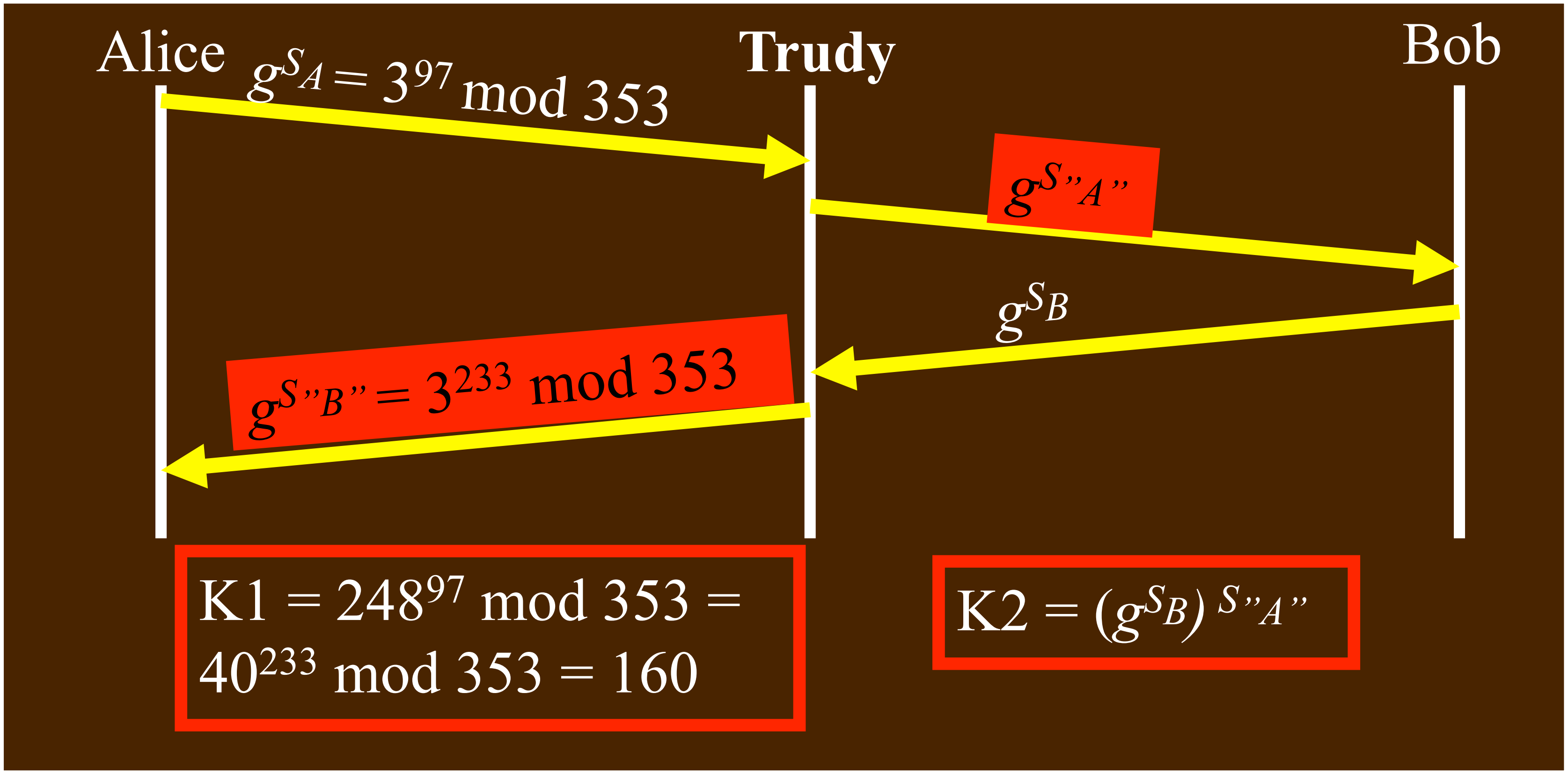
- Discrete log problem:
  - given T_A (= g^{S_A} mod p), g, and p, it is computationally infeasible to compute S_A
  - (note: as always, to the best of our knowledge; doesn't mean there isn't a method out there waiting to be found)
  - same statement can be made for T_B, g, p, and S_B

D-H Limitations

- Expensive exponential operation is required
  - possible timing attacks??
- Algorithm is useful for key negotiation only
  - i.e., not for public key encryption
- Not for user authentication
  - In fact, you can negotiate a key with a complete stranger!

Man-In-The-Middle Attack

- Trudy impersonates as Alice to Bob, and also impersonates as Bob to Alice



MITM Attack (Cont'd)

- Now, Alice thinks K1 is the shared key, and Bob thinks K2 is the shared key
- Trudy intercepts messages from Alice to Bob, and
  - decrypts (using K1), substitutes her own message, and encrypts for Bob (using K2)
  - likewise, intercepts and substitutes messages from Bob to Alice
- Solution???

Authenticating D-H Messages

- That is, you know who you're negotiating with, and that the messages haven't been modified
- Requires that communicating parties already share some kind of a secret
- Then use encryption, or a MAC (based on this previously-shared secret), of the D-H messages

Using D-H in "Phone Book" Mode

1. Alice and Bob each choose a semi-permanent secret number, generate T_A and T_B
2. Alice and Bob *publish* T_A, T_B, i.e., Alice can get Bob's T_B at any time, Bob can get Alice's T_A at any time
3. Alice and Bob can then generate a semi-permanent shared key without communicating
   - but, they must be using the same p and g

- Essential requirement: reliability of the published values (no one can substitute false values)
  - how accomplished???

Encryption Using D-H?

WILLIAM & MARY

- How to do key distribution + message encryption in one step
- Everyone computes and publishes their own individual <p_i, g_i, T_i>, where $T_i = g_i^{S_i} \bmod p_i$
- For Alice to communicate with Bob…
  1. Alice picks a random secret S_A
  2. Alice computes $g_B^{S_A} \bmod p_B$
  3. Alice uses $K_{AB} = T_B^{S_A} \bmod p_B$ to encrypt the message
  4. Alice sends encrypted message along with (unencrypted) $g_B^{S_A} \bmod p_B$

Encryption (Cont'd)

WILLIAM & MARY

- For Bob to decipher the encrypted message from Alice
  1. Bob computes $K_{AB} = (g_B^{S_A})^{S_B} \bmod p_B$
  2. Bob decrypts message using K_{AB}

Example

- Bob publishes $<p_B, g_B, T_B> = <401, 5, 51>$ and keeps secret $S_B = 58$
- Steps
  1. Alice picks a random secret $S_A = 17$
  2. Alice computes $g_B^{S_A} \bmod p_B =$ ___ mod ___ = 173
  3. Alice uses $K_{AB} = T_B^{S_A} \bmod p_B =$ ___ mod ___ = **360** to encrypt message M
  4. Alice sends encrypted message along with (unencrypted) $g_B^{S_A} \bmod p_B = 173$
  5. Bob computes $K_{AB} = (g_B^{S_A})^{S_B} \bmod p_B =$ ___ mod ___ = **360**
  6. Bob decrypts message M using K_{AB}

Example

- Bob publishes <p_B, g_B, T_B> = <401, 5, 51> and keeps secret $S_B = 58$
- Steps
  1. Alice picks a random secret $S_A = 17$
  2. Alice computes $g_B^{S_A} \bmod p_B = 5^{17} \bmod 401 = 173$
  3. Alice uses $K_{AB} = T_B^{S_A} \bmod p_B = 51^{17} \bmod 401 =$ **360** to encrypt message M
  4. Alice sends encrypted message along with (unencrypted) $g_B^{S_A} \bmod p_B = 173$
  5. Bob computes $K_{AB} = (g_B^{S_A})^{S_B} \bmod p_B = 173^{58} \bmod 401 =$ **360**
  6. Bob decrypts message M using K_{AB}

Picking g and p

- Advisable to change g and p periodically
  - the longer they are used, the more info available to an attacker
- Advisable not to use same g and p for everybody
- For "obscure mathematical reasons"…
  - $(p-1)/2$ should be prime
  - $g^{(p-1)/2}$ should be $\equiv -1 \bmod p$

Digital Signature Standard (DSS)

Digital Signature Standard (DSS)

- Useful only for digital signing (no encryption or key exchange)
- Components
  - SHA-1 to generate a hash value (some other hash functions also allowed now)
  - Digital Signature Algorithm (DSA) to generate the digital signature from this hash value
- Designed to be fast for the signer rather than verifier
  - e.g., for use in *smart cards*

Digital Signature Algorithm (DSA)

1. Announce public parameters used for signing
   - pick p (a prime with >= 1024 bits)

     ex.: $p = 103$
   - pick q (a 160 bit prime) such that $q|(p\text{-}1)$

     ex.: $q = 17$ (divides 102)
   - choose $g \equiv h^{(p\text{-}1)/q} \bmod p$, where $1 < h < (p - 1)$, such that $g > 1$

     ex.: if $h = 2$, $g = 2^6 \bmod 103 = 64$
   - note: g is of order q mod p

ex.: powers of 64 mod 103 =
64 79 9 61 93 81 34 13 8 100 14 72 76 23 30 66 1

17 values

DSA (Cont'd)

WILLIAM & MARY

2. User Alice generates a long-term private key x_M
   - random integer with $0 < x_M < q$

ex.: $x_M = 13$

3. Alice generates a long-term public key y_M
   - $y_M = g^{x_M} \bmod p$

ex.: $y_M = 64^{13} \bmod 103 = 76$

DSA (Cont'd)

ex.: $p = 103$, $q = 17$, $g = 64$, $x_M = 13$, $y_M = 76$

4. Alice randomly picks a private key k such that $0 < k < q$, and generates $k^{-1} \bmod q$

ex.: $k = 12$, $12^{-1} \bmod 17 = 10$

5. Signing message M

ex.: $H(M) = 75$

- public key $r = (g^k \bmod p) \bmod q$

ex.: $r = (64^{12} \bmod 103) \bmod 17 = 4$

- signature $s = [k^{-1}(H(M)+x_M r)] \bmod q$

ex.: $s = [10 * (75 + 13*4)] \bmod 17 = 12$

- transmitted info = M, r, s

ex.: M, 4, 12

Verifying a DSA Signature

- Known : g, p, q, y_M

ex.: p = 103, q = 17, g = 64, y_M = 76, H(M) = 75

- Received from signer: M, r, s

ex.: M, **4**, 12

1. $w = (s)^{-1} \bmod q$

ex.: $w = 12^{-1} \bmod 17 = 10$

2. $u_1 = [H(M)w] \bmod q$

ex.: $u_1 = 75*10 \bmod 17 = 2$

3. $u_2 = (r*w) \bmod q$

ex.: $u_2 = 4*10 \bmod 17 = 6$

4. $v = [(g^{u1} * y_M^{u2}) \bmod p] \bmod q$

ex.: $v = [(64^2 * 76^6) \bmod 103] \bmod 17 = \mathbf{4}$

5. If $v = r$, then the signature is verified

Verifying DSA Signature

- Received: M, r=**13**, s=24

1. $w = (s)^{-1} \bmod q = 24$
2. $u_1 = [H(M)w] \bmod q = 22*24 \bmod 25 = 3$
3. $u_2 = (r)w \bmod q = 13 * 24 \bmod 25 = 12$
4. $v = [(g^{u1} y_A^{u2}) \bmod p] \bmod q =$
   $[5^3 * 56^{12} \bmod 101] \bmod 25 =$ **13**
5. If $v = r$, then the signature is verified

Why Does it Work?

WILLIAM & MARY

- Correct? The signer computes
- $s = k^{-1} * (H(m) + x*r) \bmod q$
- so $k \equiv H(m)*s^{-1} + x*r*s^{-1}$
- $\equiv H(m)*w + x*r*w \bmod q$
- Since g has order q:
- $g^k \equiv g^{H(m)w} * g^{xrw}$
- $\equiv g^{H(m)w} * y^{rw}$
- $\equiv g^{u1} * y^{u2} \bmod p$, and
- $r = (g^k \bmod p) \bmod q = (g^{u1}*y^{u2} \bmod p) \bmod q = v$

Is it Secure?

- Given y_M, it is difficult to compute x_M
  - x_M is the discrete log of y_M to the base g, mod p
- Likewise, given r, it is difficult to compute k
- Cannot forge a signature without x_M
- Signatures are not repeated (only used once per message) and cannot be replayed

Assessment of DSA

- Slower to verify than RSA, but faster signing than RSA
- Key lengths of 2048 bits and greater are also allowed