CS 55:
Security and Privacy

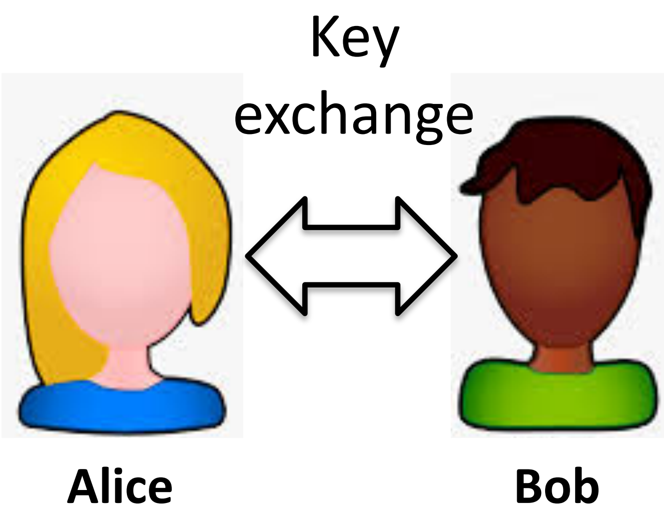
Asymmetric encryption

Source: https://xkcd.com/1323/

Agenda

1. Diffie-Helman key exchange
2. Asymmetric encryption
3. The RSA algorithm
4. Digital Signatures

With symmetric key cryptography, both sender and receiver must know key



Both Alice and Bob must both know the same key for symmetric encryption

Traditionally keys distributed via paper and trusted couriers

Problematic in web age!

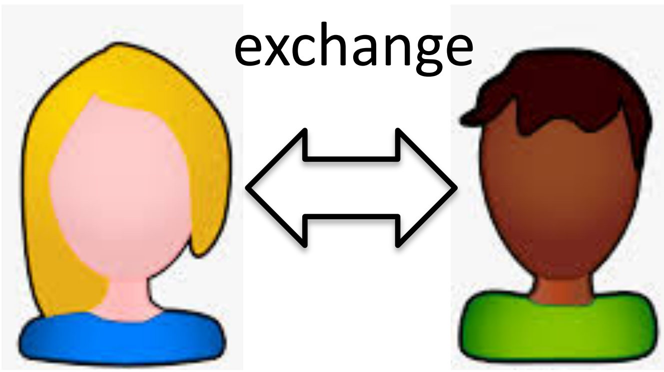
Eve

If Eve learns key, can decrypt messages

We assume Eve is always listening

Diffie-Helman (DH) allows a sender and receiver to develop the same key

Key exchange



Alice **Bob**

Goal: develop the same key on both sides, but prevent Eve from doing the same

Then use the key for symmetric encryption (e.g., AES)

Called key agreement protocol

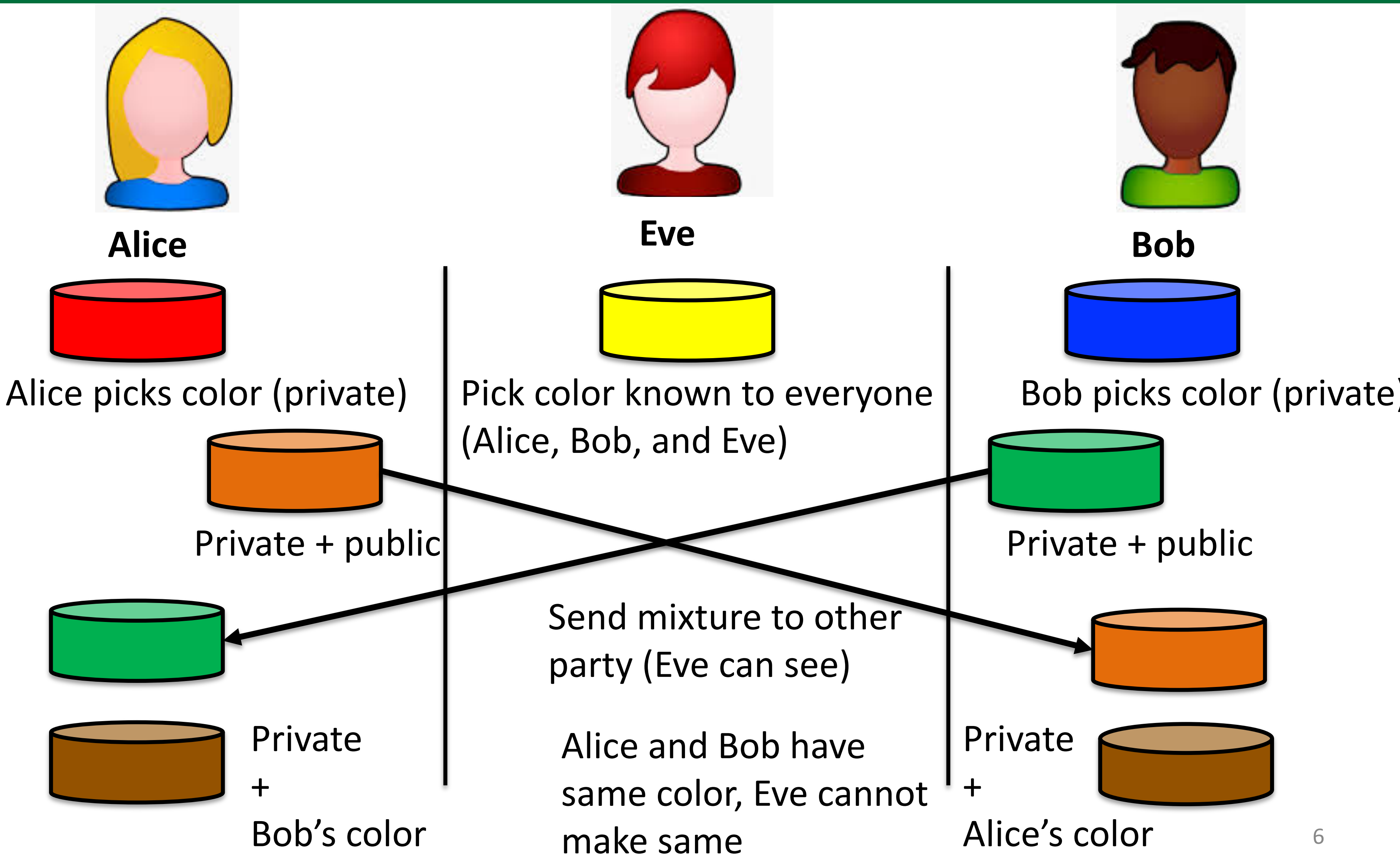
Eve

If Eve learns key, can decrypt messages

We assume Eve is always listening

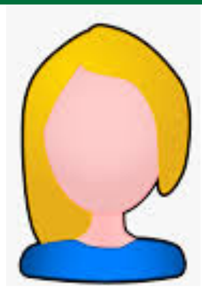
NOTE: Diffie-Helman is non-authenticated (you don't know with whom you are agreeing a key)!

To get the intuition behind DH, consider mixing colors three colors



Inspired by: https://www.youtube.com/watch?v=NmM9HA2MQGI

In reality DH uses two public prime numbers and two private random numbers



Alice

Elliptic curve DH (ECDH) uses curves instead of p and g

Eve

Bob

Pick random $1<a<p$

Alice and Bob pick public numbers g and p

- p large prime
- g small prime

Pick random $1<b<p$

What is the problem here? Non-authenticated, who did you agree a key with?

$g^a \bmod p \longrightarrow$

$\longleftarrow g^b \bmod p$

Compute
$K = (g^b \bmod p)^a \bmod p$
$= g^{ab} \bmod p$

Eve cannot separate x and y from g and p

Compute
$K = (g^a \bmod p)^b \bmod p$
$= g^{ab} \bmod p$

Now Alice and Bob know same key, can use with symmetric encryption (AES)

Inspired by: https://www.youtube.com/watch?v=Yjrfm_oRO0w

Test program confirms this approach allows Alice and Bob to calculate same key

diffie_helman.c

```
void main() {
        int g = 5;  //prime, known to alice, bob, eve
        int p = 23; //prime, known to alice, bob, eve

        int a = 6; //random private, known only to alice
        int b = 15; //random private, known only to bob

        //alice knows g, p, and a
        //bob knows g, p, and b
        //eve knows g, p

        //alice sends to bob g^a mod p
        //bob sends to alice g^b mod p
        int a_to_b = (long long)(pow(g,a)) % p;
        int b_to_a = (long long)(pow(g,b)) % p;
        //eve sees both a_to_b and b_to_a

        //alice calcualtes b_to_a^a % p
        //bob calculates a_to_b^b %p
        int alice = (long long)(pow(b_to_a,a)) % p;
        int bob = (long long)(pow(a_to_b,b)) % p;

        printf("Alice calculates %i\n",alice);
        printf("Bob calculates %i\n",bob);
}
```

g **(small prime) and** ***p*** **(large prime) to known all**

Alice and Bob each pick private random number

- **Alice does not know** ***b***
- **Bob does not know** ***a***
- ***Eve*** **does not know either a or b**

Alice sends ***g^a mod p*** **to Bob**
Bob sends ***g^b mod p*** **to Alice**

- **Eve sees both numbers**
- ***Discrete log problem*** **says it is difficult for Eve to extract a or b from numbers she sees**
- **No known way to solve quickly**

Alice calculates ***(g^b mod p)^a mod p***
Bob calculates ***(g^a mod p)^b mod p***
= ***g^{ab} mod p***

Each calculate 2 in this example

Alice and Bob have same key, but do not know each other's secret
Eve cannot calculate key

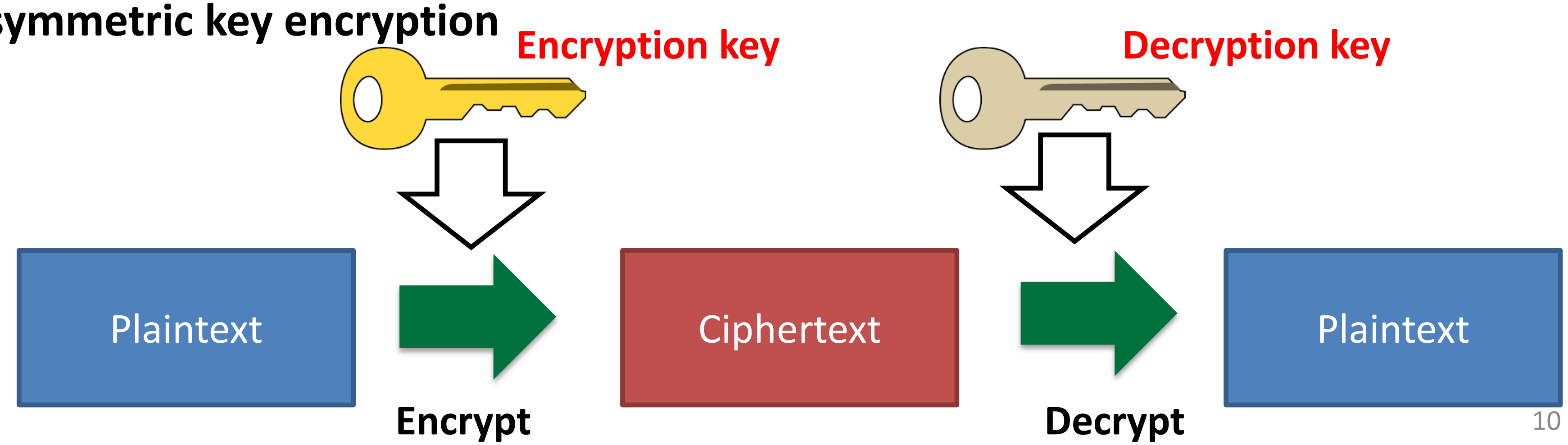
Agenda

1. Diffie-Helman key exchange
2. Asymmetric encryption
3. The RSA algorithm
4. Digital Signatures

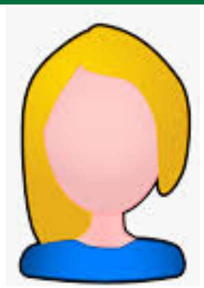
Asymmetric encryption uses a public and private key; symmetric only uses one key

Symmetric key encryption

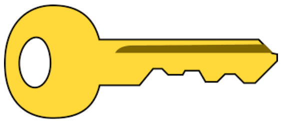
Asymmetric key encryption



Step 1: Alice creates public and private key, publishes public key; keeps private secret



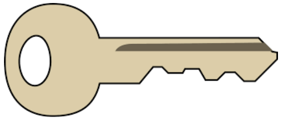
Alice



Alice publishes public key

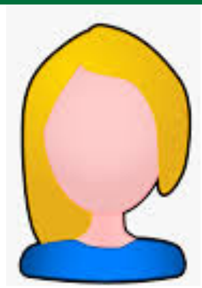
Eve

Bob

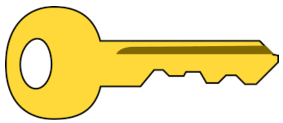


Private key held in secret

Step 2: Bob gets Alice's public key

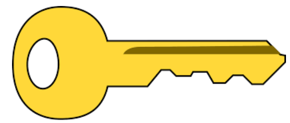


Alice



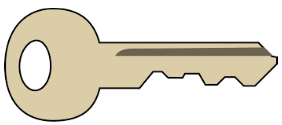
Alice publishes public key

Eve



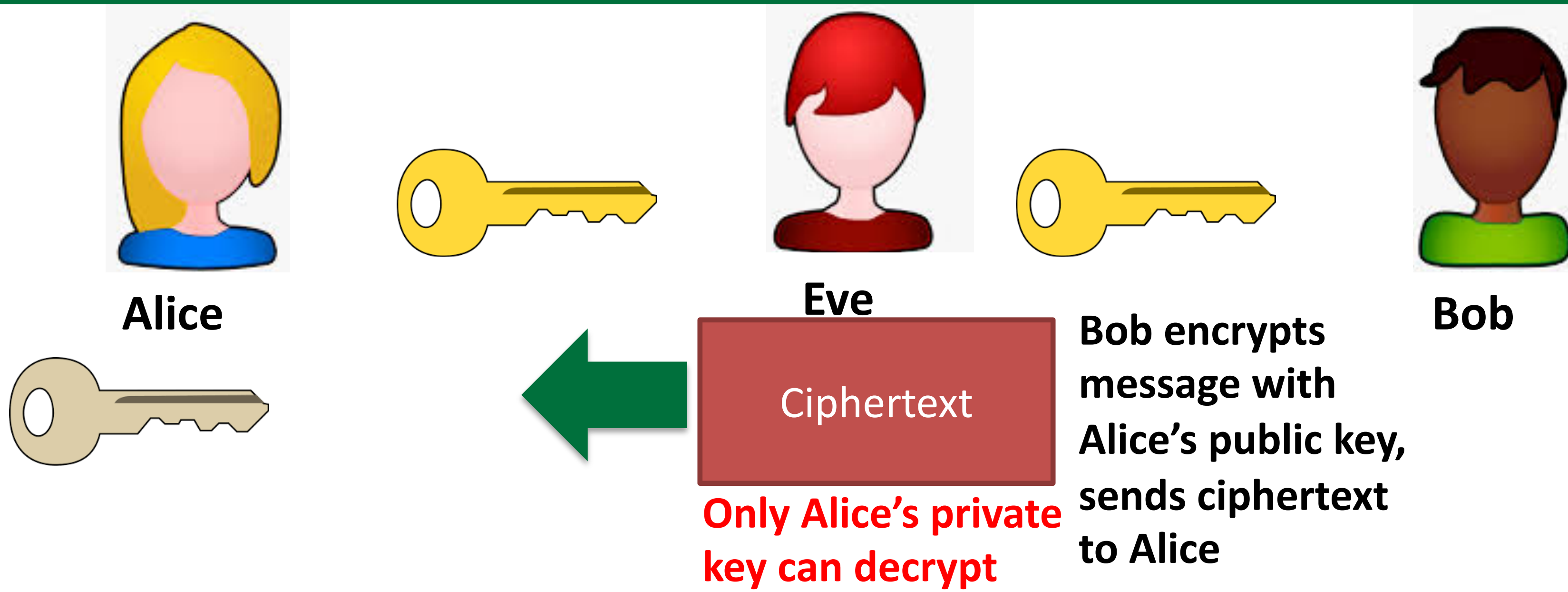
Bob gets Alice's public key

Bob

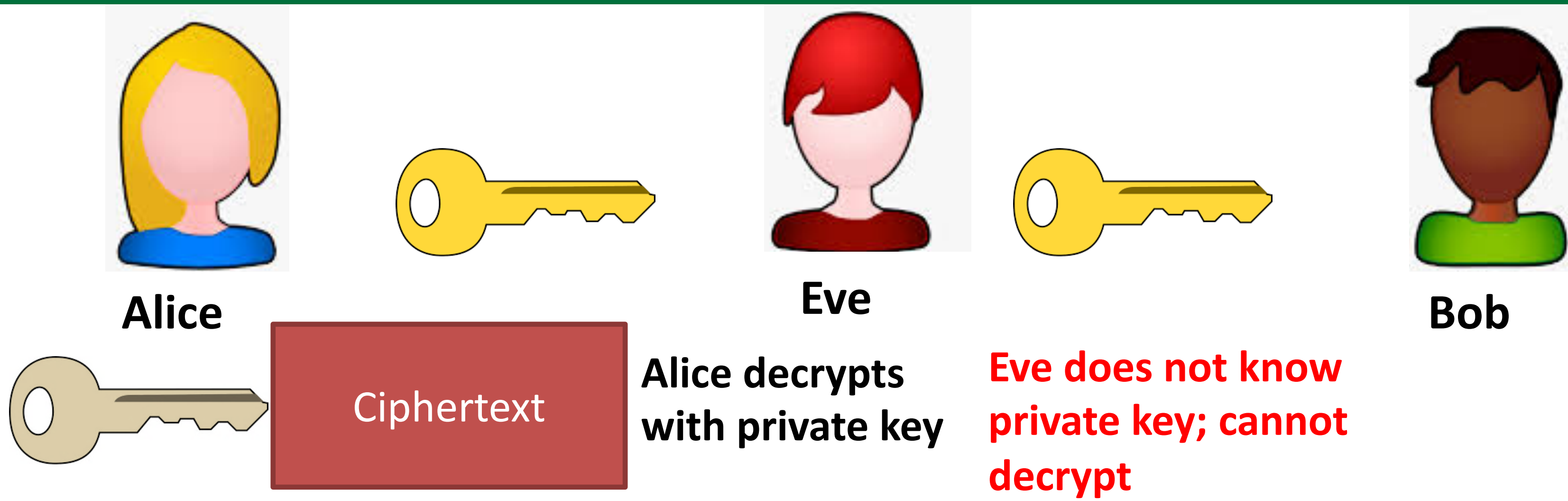


Private key held in secret

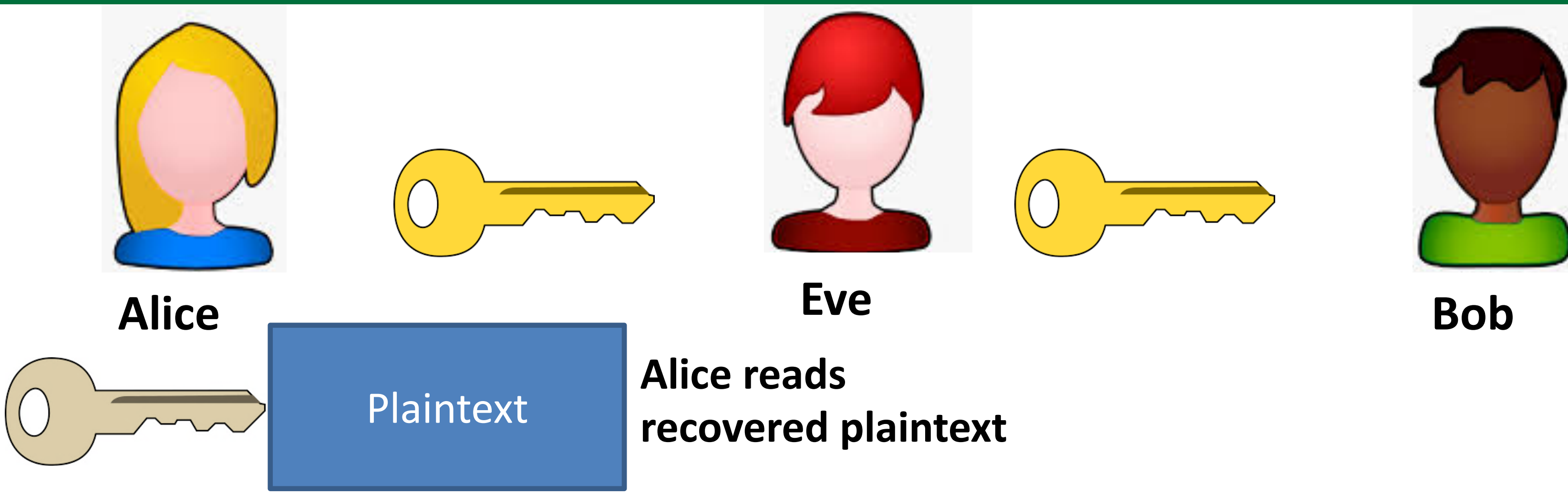
Step 3: Bob encrypts message to Alice using Alice's public key



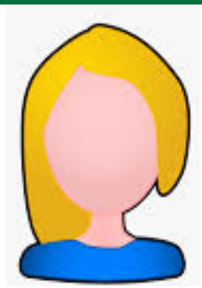
Step 4: Alice decrypts message using private key



Step 5: Alice reads plaintext from Bob



We can tweak DH to turn it into a Public Key encryption scheme



Alice

Eve

Bob

Pick: g, p, a

Public key: g, p, $g^a \bmod p$

Private key: a

Pick: b

Works, but not used this way in practice

We use RSA instead

Send encrypted message to Alice:

- Get Alice public key = $g^a \bmod p$
- Compute $K = (g^a \bmod p)^b \bmod p$
  $= g^{ab} \bmod p$
- Use AES to encrypt $C = E(m,K)$
- Send C and $g^b \bmod p$

To decrypt

- $K = (g^b \bmod p)^a \bmod p$
  $= g^{ab} \bmod p$
- Use AES to decrypt $m = D(C,K)$

Agenda

1. Diffie-Helman key exchange
2. Asymmetric encryption
3. The RSA algorithm
4. Digital Signatures

Step 1: Alice generates a public encryption key (*e,n*) and private decryption key *d*

Alice

Key generation:

- Pick two large primes *p* and *q*
- Compute *n = pq*
- Select public encryption key *e* (often use *e* = 65537 = 0x10001)
- Find private decryption key *d* such that *ed* mod ϕ*(n)* = 1
- Publish (*e*,n)
- Keep *d* secret

Choosing public key *e*

- **Choice of *e* is made public (nothing to hide)**
- **Want *e* that is coprime with $\phi(n) = (p-1)(q-1)$**
- **In practice fix *e* first, often use 65537 (prime)**

Choosing private key *d*

- **Decryption key *d* based on choice of *e, p, and q***
- **Solve *ed* mod $\phi(n) = 1$ (inverse *e*)**

Implementation issues

- **Will be dealing with big numbers, use BigNum package in C**
- **sudo apt-get install libssl-dev**
- **To get *d*, use:**
  BN_mod_inverse(d, e, phi, ctx);

Du, Wenliang. *Computer & Internet Security: A Hands-on Approach*. 2019.

The BIGNUM library in C can create public key *(e,n)* and private key *d*

rsa.c

```
BN_CTX *ctx = BN_CTX_new();
BIGNUM *p, *q, *n, *phi, *e, *d, *m, *c, *res;
BIGNUM *new_m, *p_minus_one, *q_minus_one;
<snip>

printf("Creating public and private keys\n");
// Set the public key exponent e
BN_dec2bn(&e, "65537");
printBN("Public key (hex):",e);

// Check whether e and (*@$\phi(n)$@*) are relatively prime.
do {
  // Generate random p and q.
  BN_generate_prime_ex(p, NBITS, 1, NULL, NULL, NULL); //random prime p
  BN_generate_prime_ex(q, NBITS, 1, NULL, NULL, NULL); //random prime q
  BN_sub(p_minus_one, p, BN_value_one());    // Compute p-1
  BN_sub(q_minus_one, q, BN_value_one());    // Compute q-1
  BN_mul(n, p, q, ctx);                      // Compute n=pq
  //make sure e is relatively prime to phi
  BN_mul(phi, p_minus_one, q_minus_one, ctx); // Compute (*@$\phi(n)$@*)
  BN_gcd(res, phi, e, ctx);
} while (!BN_is_one(res)); // try again if not relatively prime

// Compute the private key exponent d, s.t. ed mod phi(n) = 1
BN_mod_inverse(d, e, phi, ctx);
printBN("Private key (hex):", d);
```

Set up BIGNUM context

Fix public key e as prime number
Need not be 65537, but commonly chosen

Calculate random large primes p and q

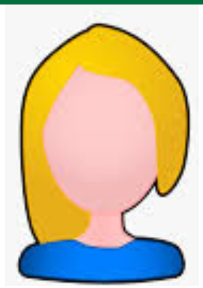
Calculate p-1,q-1, and n

Ensure *e* and $\phi(n) = (p-1)(q-1)$ are coprime (gcd ==1)

Calculate *d* as inverse of *e*
(*d* undoes *e*'s encryption)
Solves $ed \bmod \phi(n) = 1$

Du, Wenliang. *Computer & Internet Security: A Hands-on Approach*. 2019.

Step 2: Create ciphertext by converting message to number, then raise to *e*



Alice

Key generation:

- Pick two large primes p and q
- Compute $n = pq$
- Select public encryption key e (often use e = 65537 = 0x10001)
- Find private decryption key d such that $ed \bmod \phi(n) = 1$
- Publish (e,n)
- Keep d secret

Bob

2) Create ciphertext

- Get Alice's public key e and n
- Convert message to number m (string of hex numbers)
- Calculate ciphertext:

  $c = m^e \bmod n$
- Send c to Alice

BIGNUM can create ciphertext from message using recipient's public key (*e,n*)

rsa.c

```
// Encryption: calculate m^e mod n
  printf("\nEncrypting message: %s\n",msgToSend);
  char *msgHex = convertStringtoHex(msgToSend); //convert msgToSend to hex
  BN_hex2bn(&m, msgHex);
  BN_mod_exp(c, m, e, n, ctx); //encrypt with public key e and n
  printBN("Encryption result:", c);
```

Convert message to send to number *m* (here a hex string based on characters in msgToSend)

Concatenates character hex values into long string *m*
Create *m* in BIGNUM

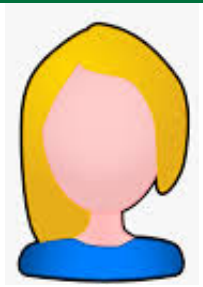
Calculate ciphertext
$c = m^e \bmod n$

- **If *Bob* sending to *Alice* use Alice's published *e* and *n***
- **If *Alice* sending to *Bob*, use Bob's published *e* and *n***

```
char *convertStringtoHex(char *msg) {
  const int len = strlen(msg);
  char *hex = malloc((len+1)*sizeof(char)*2);

  for (int i = 0, j = 0; i < len; ++i, j += 2)
      sprintf(hex + j, "%02x", msg[i] & 0xff);
 return hex;
}
```


Du, Wenliang. *Computer & Internet Security: A Hands-on Approach*. 2019.

Step 3: Use *d* to decrypt ciphertext *C*



Alice

Bob

Key generation:

- Pick two large primes p and q
- Compute $n = pq$
- Select public encryption key e (often use e = 65537 = 0x10001)
- Find private decryption key d such that ed mod $\phi(n) = 1$
- Publish (e,n)
- Keep d secret

2) Create ciphertext

- Get Alice's public key e and n
- Convert message to number m (string of hex numbers)
- Calculate ciphertext:
  $c = m^e \bmod n$
- Send c to Alice

3) Decrypt *c* using *d*

- Calculate $m = c^d \bmod n = (m^e \bmod n)^d \bmod n = m^{ed} \bmod n = m$

BIGNUM can decrypt ciphertext *C* using private key *d*

rsa.c

```
// Decryption: calculate c^d mod n
  BN_mod_exp(new_m, c, d, n, ctx); //decrypt with private key d
  printBN("Decryption result:", new_m);
  char *string = BN_bn2hex(new_m);
  char *recovered_message = convertHexToString(string);
  printf("Decrypted message: %s\n",recovered_message);

  // Clear the sensitive data from the memory
  BN_clear_free(p); BN_clear_free(q); BN_clear_free(d);
  BN_clear_free(phi); BN_clear_free(m); BN_clear_free(new_m);
  BN_clear_free(c); BN_clear_free(res);
  BN_clear_free(p_minus_one); BN_clear_free(q_minus_one);

  char *convertHexToString(char *hex) {
    const int len = strlen(hex);
    char *string = malloc((len/2+1)*sizeof(char));
     for (int i = 0, j = 0; j < len; ++i, j += 2) {
       int val[1];
       sscanf(hex + j, "%2x", val);
       string[i] = val[0];
       string[i + 1] = '\0';
     }
    return string; }
```

Use private decryption key *d* to recover message *m*

Convert hex back to string

Don't leave crypto info lying around in memory!!

Use BIGNUM's methods to clear memory

Du, Wenliang. *Computer & Internet Security: A Hands-on Approach*. 2019.

OpenSSL can create public and private keys from the command line instead of in C

```
$ openssl genrsa -aes128 -out privateKey.pem 1024
Generating RSA private key, 2014 bit long modulus
......................+++
.........................................+++
e is 65537 (0x10001)
Enter pass phrase for privateKey.pem:
Verifying - Enter pass phrase for privateKey.pem:
```

Size of private key in bits

Store key in file called privateKey.pem

Encrypt output file with 128-bit AES (using password promoted – here cs55)

If this option not selected, key file is not password protected

.pem format stores data in Base64 encoding

Examine the contents of the key file (.pem) using OpenSSL's rsa command

Do not show encrypted data

$openssl rsa -in privateKey.pem -noout -text
Enter pass phrase for privateKey.pem:
Private-Key: (1024 bit)
modulus:
  00:b3:b8:1e:de:de:2f:91:a1:00:c3:f0:4b:73:5d:...<snip>
publicExponent: 65537 (0x10001)
privateExponent:
  00:8a:c0:ae:64:d7:19:d6:cf:7d:2d:c9:ca:16:e9:...<snip>
prime1:
  00:de:c7:8b:8f:67:45:59:dd:f9:8f:65:dd:8b:87:...<snip>
prime2:
  00:ce:84:c7:93:8b:20:d0:3c:35:9b:d6:01:cf:05:...<snip>
exponent1:
  00:9b:81:0a:47:b5:3c:51:78:82:64:b8:24:26:ea:...<snip>
exponent2:
  00:bd:9f:3f:5c:e2:ff:63:14:15:a9:1b:ec:17:39:...<snip>
coefficient:
  00:a3:c0:5d:0e:5d:52:db:a1:d8:6e:55:63:e5:b3:...<snip>

Show data in plaintext

Output data stored in key .pem file

Public key *n*

Public key *e*

Private key *d*

Other values for performance improvement

DO NOT share the private key file!

Create a public key file using the rsa command also

```
$openssl rsa -in privateKey.pem -pubout > publicKey.pem
Enter pass phrase for privateKey.pem:
writing RSA key
```

Extract publicKey from privateKey.pem, save in publicKey.pem

More shows the value of the public key

```
$openssl rsa -in privateKey.pem -pubout > publicKey.pem
Enter pass phrase for privateKey.pem:
writing RSA key

$ more publicKey.pem
-----BEGIN PUBLIC KEY-----
MIGfMA0GCSqGSIb3DQEBAQUAA4GNADCBiQKBgQCzuB7e3i+RoQDD8EtzXXVHi24C
VpcXWro5BFw1gRHQIkN61A6cGEbta5KhPKTz8eqd9AihflLHwRSB2U7Cna26aYK+
e9tdyMuFt8hAEtlk2MBYmxfpw2p8pdcBznKLiLzLJt4ozsf+Fsw0NUiE2wwYI+ZG
jLW1wJ4NM0iTM1JYjwIDAQAB
-----END PUBLIC KEY-----
```

Extract publicKey from privateKey.pem, save in publicKey.pem

See the entire public key

OpenSSL shows public key (*n, e)* in a more readable form

```
$openssl rsa -in privateKey.pem -pubout > publicKey.pem
Enter pass phrase for privateKey.pem:
writing RSA key

$ more publicKey.pem
-----BEGIN PUBLIC KEY-----
MIGfMA0GCSqGSIb3DQEBAQUAA4GNADCBiQKBgQCzuB7e3i+RoQDD8EtzXXVHi24C
VpcXWro5BFw1gRHQIkN61A6cGEbta5KhPKTz8eqd9AihflLHwRSB2U7Cna26aYK+
e9tdyMuFt8hAEtlk2MBYmxfpw2p8pdcBznKLiLzLJt4ozsf+Fsw0NUiE2wwYI+ZG
jLW1wJ4NM0iTM1JYjwIDAQAB
-----END PUBLIC KEY-----

$ openssl rsa -in publicKey.pem -pubin -text -noout
Public-Key: (1024 bit)
Modulus:
    00:b3:b8:1e:de:de:2f:91:a1:00:c3:f0:4b:73:5d:...<snip>
Exponent: 65537 (0x10001)
```

Extract publicKey from privateKey.pem, save in publicKey.pem

See the entire public key

Get *n* (modulus) and *e*

Encrypt and decrypt data using public and private keys

```
$ echo "We attack at dawn!" > msg.txt

$ openssl rsautl -encrypt -inkey publicKey.pem -pubin -in msg.txt -out msg.enc
```

Message to encrypt/decrypt

Encrypt using public key (assume public key shared with message sender)

Encrypt and decrypt data using public and private keys

```
$ echo "We attack at dawn!" > msg.txt

$ openssl rsautl -encrypt -inkey publicKey.pem -pubin -in msg.txt -out msg.enc

$ xxd msg.enc
00000000: 03a5 e47f 68dc ff98 dc59 c976 ccd1 6351  ....h....Y.v..cQ
00000010: a7ab d475 c336 0530 ac77 dc4b ff87 2ba1  ...u.6.0.w.K..+.
00000020: 6c87 3937 3f76 3220 5359 9242 4c2d 0b1c  l.97?v2 SY.BL-..
00000030: 7335 52f4 3609 1610 dd69 cdf6 2c2d 80de  s5R.6....i..,-..
00000040: df6c ad76 0cb6 0ba8 fe0c 0d75 1c00 853b  .l.v.......u...;
00000050: 5762 ce15 aa5e 3c14 f70b 95b5 a940 c4e3  Wb...^<......@..
00000060: 56bb 4e2f c296 b9fd fb48 d617 17d9 0f05  V.N/.....H......
00000070: 7f74 32e7 ee26 156e 5b90 21f7 617b 3f04  .t2..&.n[.!.a{?.
```

Message to encrypt/decrypt

Encrypt using public key (assume public key shared with message sender)

Ciphertext unreadable

Encrypt and decrypt data using public and private keys

```
$ echo "We attack at dawn!" > msg.txt

$ openssl rsautl -encrypt -inkey publicKey.pem -pubin -in msg.txt -out msg.enc

$ xxd msg.enc
00000000: 03a5 e47f 68dc ff98 dc59 c976 ccd1 6351  ....h....Y.v..cQ
00000010: a7ab d475 c336 0530 ac77 dc4b ff87 2ba1  ...u.6.0.w.K..+.
00000020: 6c87 3937 3f76 3220 5359 9242 4c2d 0b1c  l.97?v2 SY.BL-..
00000030: 7335 52f4 3609 1610 dd69 cdf6 2c2d 80de  s5R.6....i..,-..
00000040: df6c ad76 0cb6 0ba8 fe0c 0d75 1c00 853b  .l.v.......u...;
00000050: 5762 ce15 aa5e 3c14 f70b 95b5 a940 c4e3  Wb...^<......@..
00000060: 56bb 4e2f c296 b9fd fb48 d617 17d9 0f05  V.N/.....H......
00000070: 7f74 32e7 ee26 156e 5b90 21f7 617b 3f04  .t2..&.n[.!.a{?.

$ openssl rsautl -decrypt -inkey privateKey.pem -in msg.enc
Enter pass phrase for privateKey.pem:
We attack at dawn!
```

Message to encrypt/decrypt

Encrypt using public key (assume public key shared with message sender)

Ciphertext unreadable

Decrypt with private key (also asks for password to access private key)

NOTE: does not use AES, instead uses RSA to encrypt entire message – slow
Also make sure len(msg) > len(e)

Using RSA to encrypt a large message is slow; use it with AES

Computing ciphertext $c = m^e \bmod n$ takes a long time

- m is generally a large number (two hex digits per plaintext character), long if message is long
- Raising *m* to a large number *e* takes some time
- Result is normally too big to use *int* or *long*, need special library (BIGNUM) to calculate and store value

Computing plaintext $m = c^d \bmod n$ is also slow

Use RSA with AES for long messages *m*

- Generate random key *K* as message and exchange *K* using RSA
- Encrypt plaintext *m* with AES using random key *K*
- Send AES encrypted ciphertext to recipient
- Recipient decrypts using key K

There are some “gotchas” to be aware of when using RSA

RSA is deterministic

- Same plaintext always gives same ciphertext (with same keys)
- Can tell if two messages are the same
- Use AES random IV (say in CBC mode) once keys exchanged
- OpenSSL uses random padding

Poorly chosen *e* with short *m* encryptions can be easily decrypted

- If e is small (say $e=3$) and message m is short, m^e can be $< n$
- Because $c = m^e \bmod n$
    - If $m^e < n$ then mod does not matter
    - Ciphertext is simply m^e
    - Decrypt by just taking e^{th} root of m
- **Choose large *e* and pad *m* if needed!**

If same plaintext is encrypted more than *e* times using same *e* but different n, can decrypt plaintext using Chinese Remainder Theorem

Practice

1) Given

p = F7E75FDC469067FFDC4E847C51F452DF

q = E85CED54AF57E53E092113E62F436F4F

e = 0D88C3

Compute d

2) Given

n = DCBFFE3E51F62E09CE7032E2677A78946A849DC4CDDE3A4D0CB81629242FB1A5

e = 010001 (this hex value is decimal 65537)

M = Attack at dawn!

Compute C

3) Given

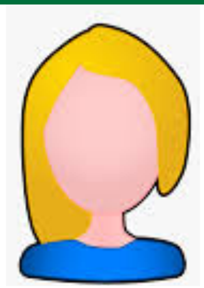
d = 74D806F9F3A62BAE331FFE3F0A68AFE35B3D2E4794148AACBC26AA381CD7D30D

Decrypt C from part 2

Agenda

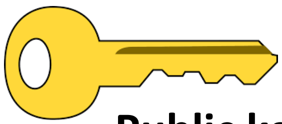
1. Diffie-Helman key exchange
2. Asymmetric encryption
3. The RSA algorithm
4. Digital Signatures

If Alice gets an encrypted message, how does she authenticate the sender?



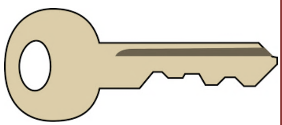
Alice

Public key available to anyone



Public key

Bob



Private key

Ciphertext

Bob could get public key and encrypt message ("We attack at dawn! – Bob")

Alice gets a message encrypted with her public key

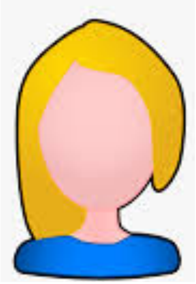
How does she know who sent the message?

We can use digital signatures to show the message must have come from a specific sender

Eve

Alice could also get public key and encrypt message ("We attack at sunset! – Bob")

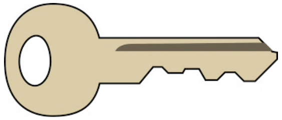
Digital signatures can be used to ensure a message came from a specific sender



Alice

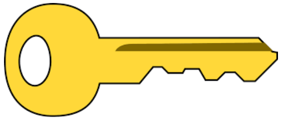
m

H(m)



Private key

Bob

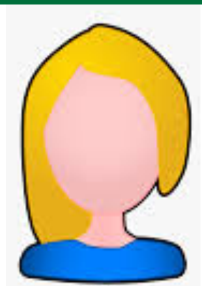


Public key

Bob

- Plaintext message *m* may be long
- Hash message to get small size (e.g., 256 bits)

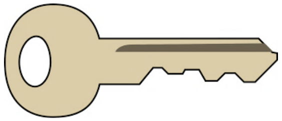
Digital signatures can be used to ensure a message came from a specific sender



Alice

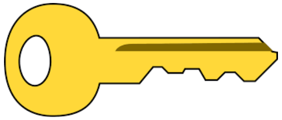
m

Sign(H(m))



Private key

Bob



Public key

Remember, the public and private keys are inverses of each other

Order they are applied doesn't matter

$[m^e \bmod n]^d \bmod n$ (first e, then d)

$= [m^d \bmod n]^e \bmod n$ (first d, then e)

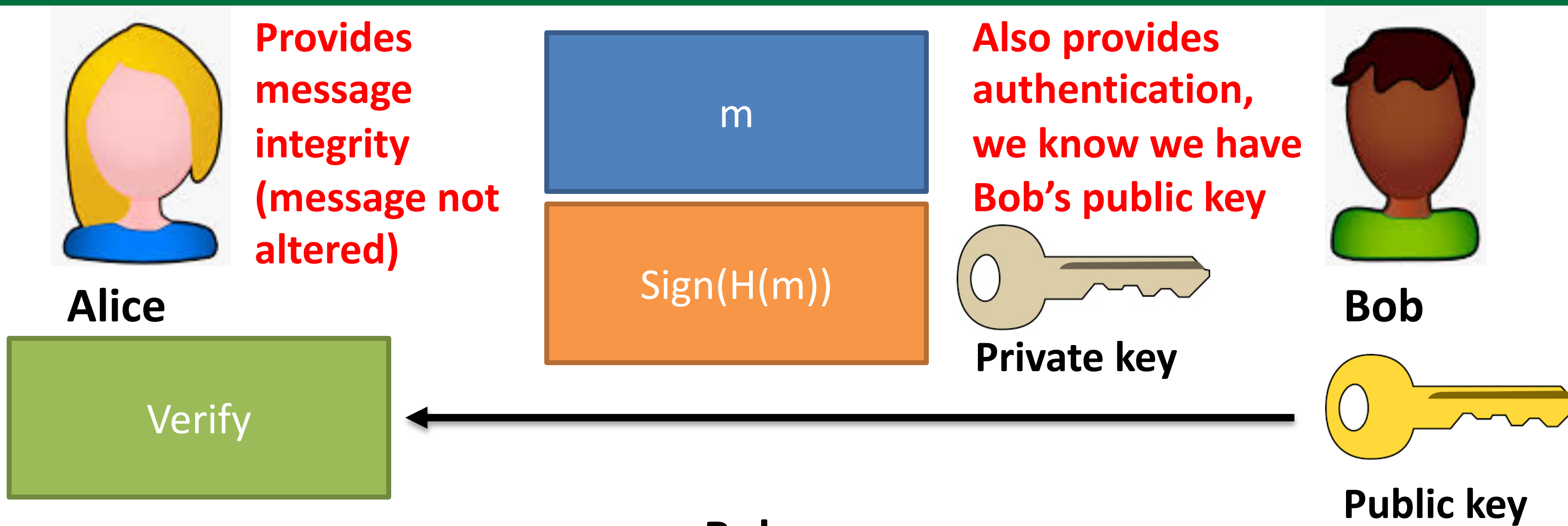
$= m^{ed} \bmod n$ (both orderings give this)

= m (which is the plaintext)

Bob

- Plaintext message *m* may be long
- Hash message to get small size (e.g., 256 bits)
- Encrypt (sign) hash using **<u>private</u>** key *d*
- Signature $s = (H(m))^d \bmod n$
- Everyone can get back *H(m)* using Bob's public key *e*, but only Bob could create *s* from *H(m)*
- Send plaintext message and signature to Alice
- Note: message not encrypted in this example (but it could be)

Digital signatures can be used to ensure a message came from a specific sender



Alice

- Gets plaintext message and signature from Bob
- Verify:
  - Decrypt signature using Bob's public key to get Bob's hash of plaintext (anyone can do this)
  - Alice hashes plaintext
  - See if hashes match
- Verifies unchanged msg from Bob

Bob

- Plaintext message *m* may be long
- Hash message to get small size (e.g., 256 bits)
- Encrypt (sign) hash using **<u>private</u>** key *d*
- Signature $s = (H(m))^d \bmod n$
- Everyone can get back *H(m)* using Bob's public key *e*, but only Bob could create *s* from *H(m)*
- Send plaintext message and signature to Alice
- Note: message not encrypted in this example (but it could be)

Demo: Alice signs message, Bob can be assured message came from Alice

#Bob creates message

$ echo –n "We attack at dawn!" > msg.txt

$ cat msg.txt

We attack at dawn!

#Bob hashes msg.txt using sha256

$ openssl sha256 -binary msg.txt > msg.sha256

$ xxd msg.sha256

00000000: cdc1 866c 2088 9e65 f705 fc42 e556 7da8 ...l ..e...B.V}.

00000010: a310 dd80 fc9f 1853 84e7 8abb a669 95b0 ....S.....i..

Hash message -> fixed size fingerprint of message

-binary leaves in binary form (signature in hex form if not included)

Demo: Alice signs message, Bob can be assured message came from Alice

```
#Bob creates message
$ echo –n "We attack at dawn!" >  msg.txt
$ cat msg.txt
We attack at dawn!
```

Hash message -> fixed size fingerprint of message
-binary leaves in binary form (signature in hex form if not included)

```
#Bob hashes msg.txt using sha256
$ openssl sha256 -binary msg.txt > msg.sha256
$ xxd msg.sha256
00000000: cdc1 866c 2088 9e65 f705 fc42 e556 7da8  ...l ..e...B.V}.
00000010: a310 dd80 fc9f 1853 84e7 8abb a669 95b0  .......S.....i..
```

Sign message using private key
Signature in msg.sig

```
#Bob signs hash
$ openssl rsautl -sign -inkey privateKey.pem  -in msg.sha256 -out msg.sig
Enter pass phrase for privateKey.pem:
```

Demo: Alice signs message, Bob can be assured message came from Alice

#Bob creates message

$ echo –n “We attack at dawn!” > msg.txt

$ cat msg.txt

We attack at dawn!

Hash message -> fixed size fingerprint of message

-binary leaves in binary form (signature in hex form if not included)

#Bob hashes msg.txt using sha256

$ openssl sha256 -binary msg.txt > msg.sha256

$ xxd msg.sha256

00000000: cdc1 866c 2088 9e65 f705 fc42 e556 7da8 ...l ..e...B.V}.

00000010: a310 dd80 fc9f 1853 84e7 8abb a669 95b0 ....S.....i..

Sign message using private key

Signature in msg.sig

#Bob signs hash

$ openssl rsautl -sign -inkey privateKey.pem -in msg.sha256 -out msg.sig

Enter pass phrase for privateKey.pem:

If msg changed, hash will not match

Verify signature using public key

If sig changed, will not match hash

#Alice verifies hash

$ openssl rsautl -verify -inkey publicKey.pem -in msg.sig -pubin -raw | xxd

00000000: 0001 ffff ffff ffff ffff ffff ffff ffff ....

00000010: ffff ffff ffff ffff ffff ffff ffff ffff ....

<snip>

00000050: ffff ffff ffff ffff ffff ffff ffff ff00 ....

00000060: cdc1 866c 2088 9e65 f705 fc42 e556 7da8 ...l ..e...B.V}.

00000070: a310 dd80 fc9f 1853 84e7 8abb a669 95b0 ....S.....i..

Match after padding

Message must have been sent by Bob because only his public key can undo the private key encryption

We can use Python for encryption, decryption, and signatures also

rsa_encrypt.py

```
#run: python3 rsa_encrypt.py publicKey.pem ciphertext.bin
message = b'We attack at dawn!'

if __name__ == '__main__':
    #read public key and set padding to OAEP
    print("Reading public key from",sys.argv[1])
    key = RSA.importKey(open(sys.argv[1],'rb').read())
    cipher = PKCS1_OAEP.new(key)   #use OAEP padding

    #encrypt message using public key
    ciphertext = cipher.encrypt(message)

    #write output file
    print("Writing output to",sys.argv[2])
    f = open(sys.argv[2],'wb')
    f.write(base64.b64encode(ciphertext))
    f.close()
```

Encrypt a message using message ("We attack at dawn!") using publicKey.pem

Store results in ciphertex.bin

We can use Python for encryption, decryption, and signatures also

rsa_decrypt.py

```
#run: python3 rsa_decrypt.py privateKey.pem cs55 ciphertext.bin

if __name__ == '__main__':
	print("Reading ciphertext from",sys.argv[3])
	ciphertext = open(sys.argv[3], 'rb').read()

	print("Reading private key from",sys.argv[1])
	private_key_pem = open(sys.argv[1],'rb').read()
	private_key = RSA.importKey(private_key_pem, passphrase=sys.argv[2])
	cipher = PKCS1_OAEP.new(private_key)

	print("Decrypting ciphertext")
	message = cipher.decrypt(base64.b64decode(ciphertext))

	print("Message:",message)
```

Read ciphertext.bin
Decrypt using privateKay.pem
(using password cs55)
Print recovered message

We can use Python for encryption, decryption, and signatures also

rsa_sign.py

```
#run: python3 rsa_sign.py privateKey.pem cs55 signature.bin
message = b'An important message' #same message used in rsa_veryify.py

if __name__ == '__main__':
	print("Reading private key",sys.argv[1])
	key_pem = open(sys.argv[1]).read()
	key = RSA.import_key(key_pem, passphrase=sys.argv[2])

	print("Message to sign is:",message)
	print("Hashing message")
	h = SHA256.new(message)

	print("Signing hash")
	signer = pss.new(key)
	signature = signer.sign(h)

	print("Writing signature to",sys.argv[3])
	open(sys.argv[3], 'wb').write(signature)
```

Sign message ("An important message") using privateKey.pem (password cs55)

Store signature in signature.bin

Note: pss is a padding scheme that adds random input

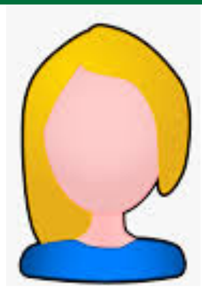
We can use Python for encryption, decryption, and signatures also

rsa_verify.py

```
#run: python3 rsa_verify.py publicKey.pem signature.bin
message = b'An important message' #same message used in rsa_sign.py

if __name__ == '__main__':
        print("Reading signature from",sys.argv[2])
        signature= open(sys.argv[2], 'rb').read()

        print("Reading public key from",sys.argv[1])
        key = RSA.import_key(open(sys.argv[1]).read())

        print(”Hashing and veryifying hash")
        h = SHA256.new(message)
        verifier = pss.new(key)
        try:
                verifier.verify(h, signature)
                print("The signature is valid.")
        except (ValueError, TypeError):
                print("The signature is NOT valid.")
```

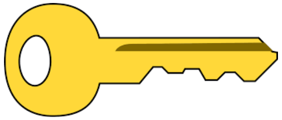
Verify signature file signature.bin using publicKey.pem

A common use of public keys is for authentication (better than passwords)



Alice

Bob



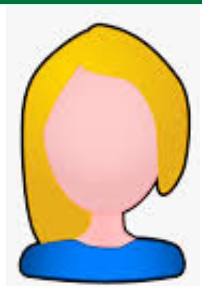
Alice
Public key

Bob wants to authenticate Alice

But anyone who knows Alice's password could appear to be Alice

Assume Bob knows Alice's public key

A common use of public keys is for authentication (better than passwords)

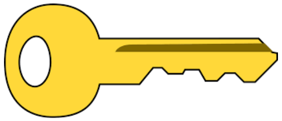


Alice

Challenge R

Bob issues random challenge R to Alice

Bob



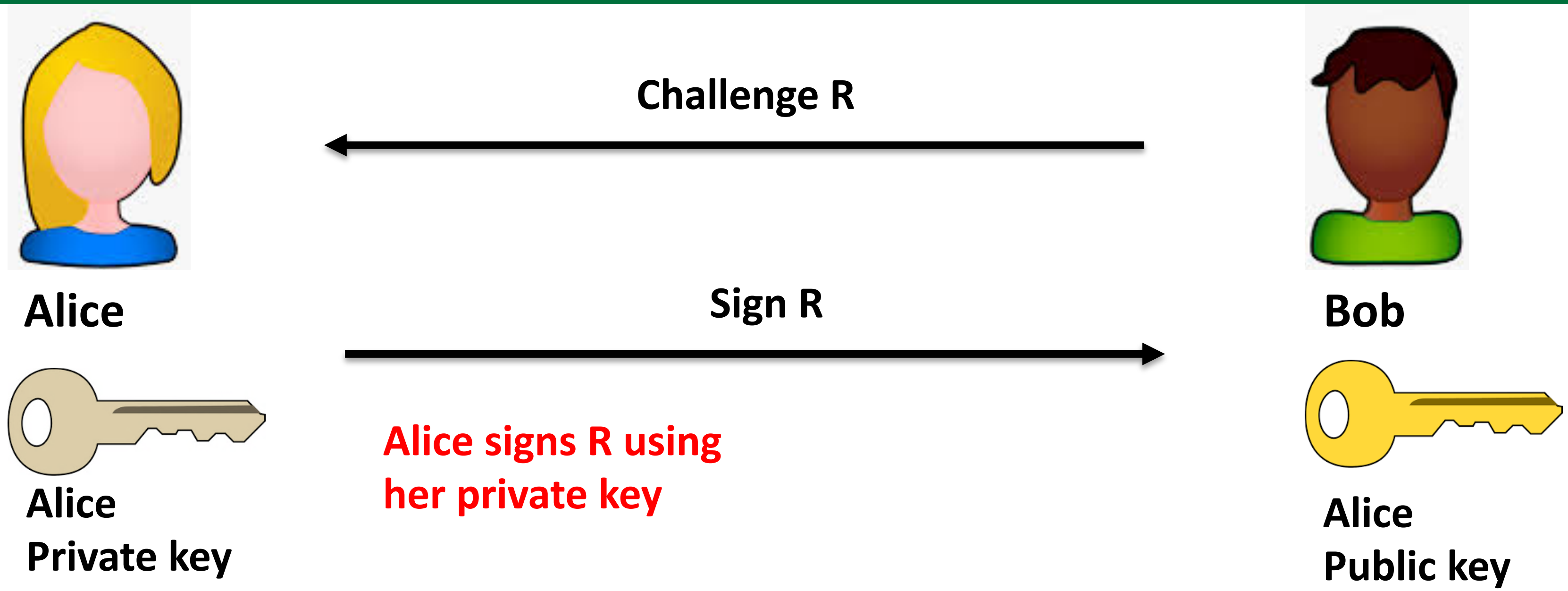
Alice
Public key

Bob wants to authenticate Alice

But anyone who knows Alice's password could appear to be Alice

Assume Bob knows Alice's public key

A common use of public keys is for authentication (better than passwords)

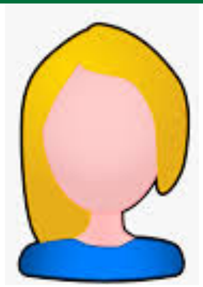


Bob wants to authenticate Alice

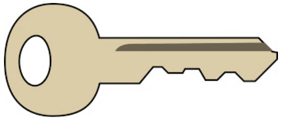
But anyone who knows Alice's password could appear to be Alice

Assume Bob knows Alice's public key

A common use of public keys is for authentication (better than passwords)



Alice

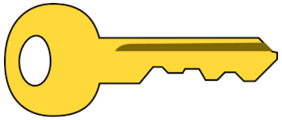


Alice
Private key

Challenge R

Sign R

Bob



Alice
Public key

Bob verifies challenge using Alice's public key

Now simply knowing a static password is not enough

Adversary needs the private key

Bob wants to authenticate Alice

But anyone who knows Alice's password could appear to be Alice

Assume Bob knows Alice's public key