Cryptography

Public Key Cryptography

("Asymmetric Encryption Systems")

Professor Travis Peters
CSCI 476 - Computer Security
Spring 2020

Some slides and figures adapted from Wenliang (Kevin) Du's
Computer & Internet Security: A Hands-on Approach (2nd Edition).
Thank you Kevin and all of the others that have contributed to the SEED resources!

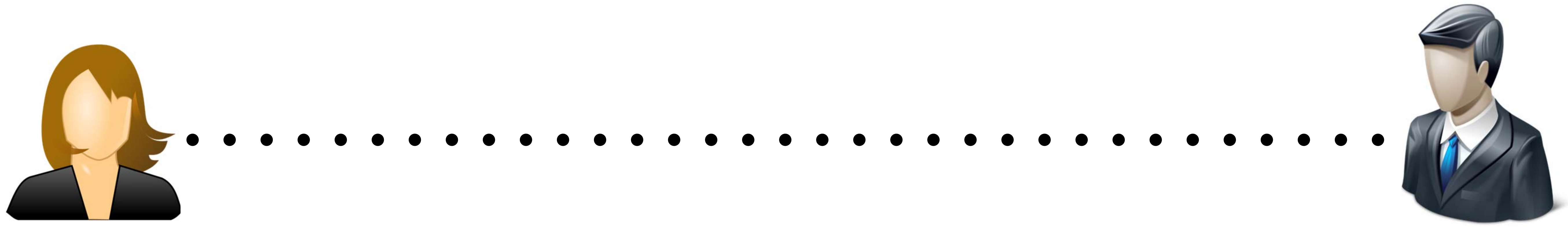
Introduction to Public Key Cryptography

This Video Covers:

- Overview of Public-Key Cryptography
- Overview of where we are going this week

Introduction & Overview

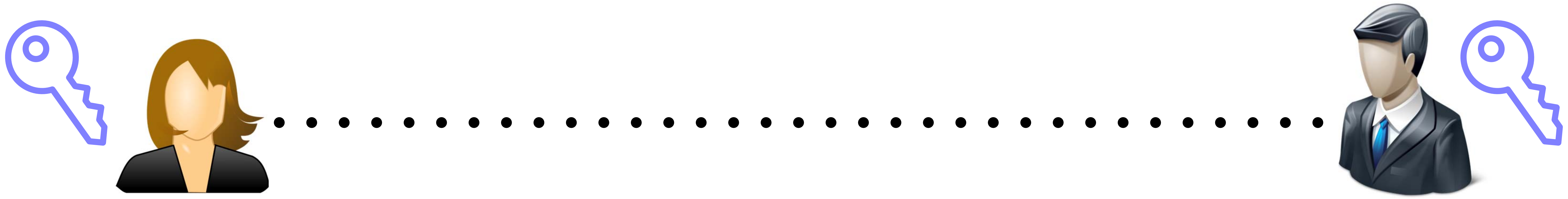
- **Public Key Cryptography** is at the foundation of today's secure communication
- Allows communicating parties to obtain a ***shared secret key***



Introduction & Overview

- **Public Key Cryptography** is at the foundation of today's secure communication
- Allows communicating parties to obtain a ***shared secret key***

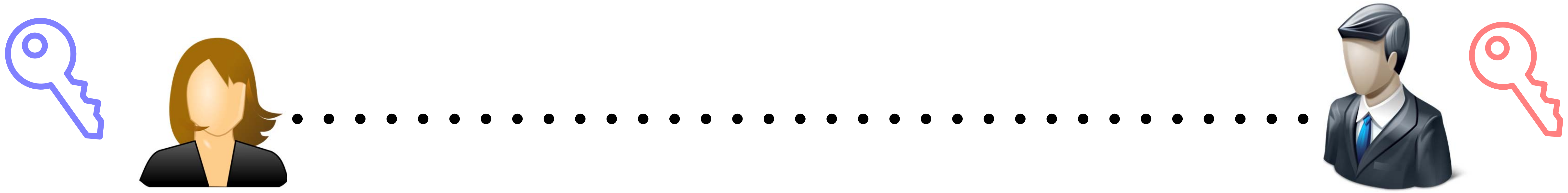
In **secret key crypto**, the **same key** was used for both encryption and decryption



Introduction & Overview

- **Public Key Cryptography** is at the foundation of today's secure communication
- Allows communicating parties to obtain a ***shared secret key***
- Public key (for ***encryption***) and Private key (for ***decryption***)
- Private key (to create ***digital signature***) and Public key (to ***verify signature***)

In **public key crypto**, **different keys** are used for encryption and decryption

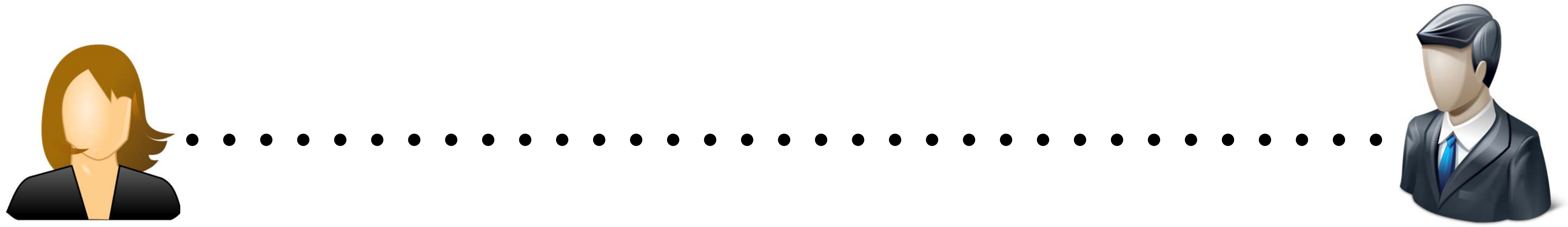


A Brief History Lesson

- Historically same key was used for encryption and decryption
- **Challenge:** exchanging the secret key (e.g. face-to-face meeting)

- 1976: Whitfield Diffie and Martin Hellman
  - DH key exchange protocol
  - proposed a new public-key cryptosystem

- 1978: Ron Rivest, Adi Shamir, and Leonard Adleman (all from MIT)
  - attempted to develop a public-key cryptosystem
  - created RSA algorithm



Outline

- Public-key algorithms
  - Diffie-Hellman key exchange
  - RSA algorithm
  - Digital signatures

- Public-key crypto & Python

- Applications
  - Authentication
  - HTTPS and TLS/SSL
  - Chip Technology Used in Credit Cards

Diffie-Hellman Key Exchange

This Video Covers:

- The DH key exchange protocol
- How to exchange (symmetric) keys

Diffie-Hellman Key Exchange *(High-Level)*

Allows communicating parties *with* ***no prior knowledge*** to ***exchange shared secret keys over an insecure channel***

Alice

Bob

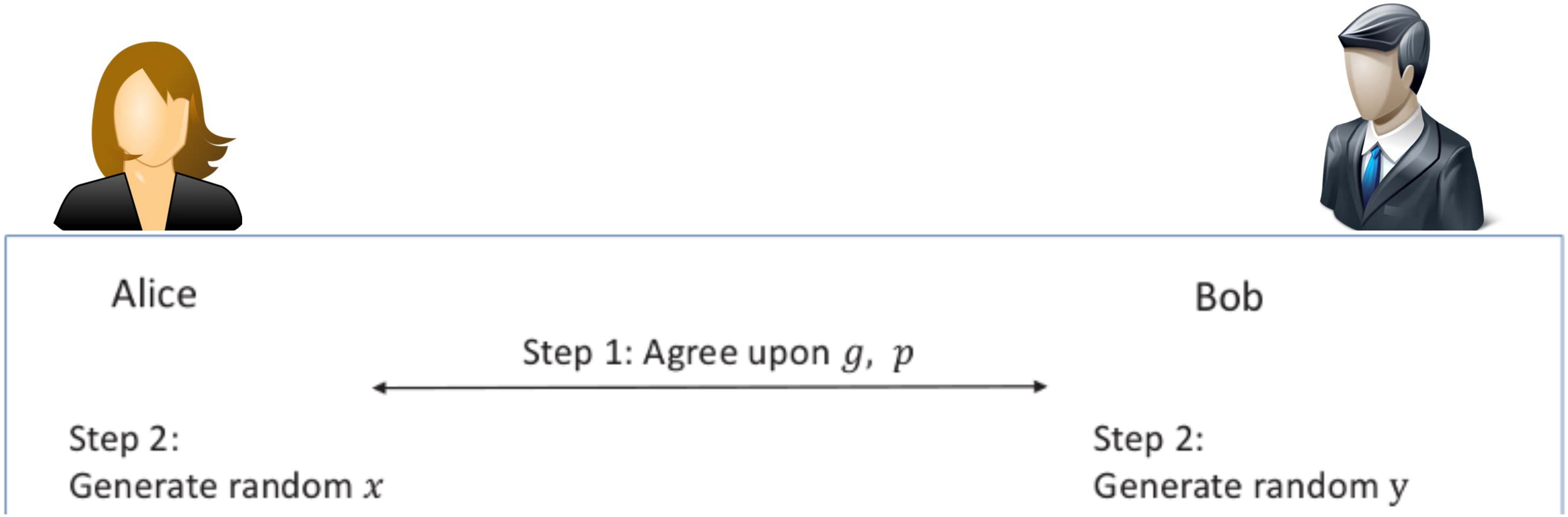
Diffie-Hellman Key Exchange *(cont.)*

Alice

Bob

Step 1: Agree upon $g,\ p$

Diffie-Hellman Key Exchange *(cont.)*



Diffie-Hellman Key Exchange *(cont.)*

Alice

Bob

Step 1: Agree upon $g,\ p$

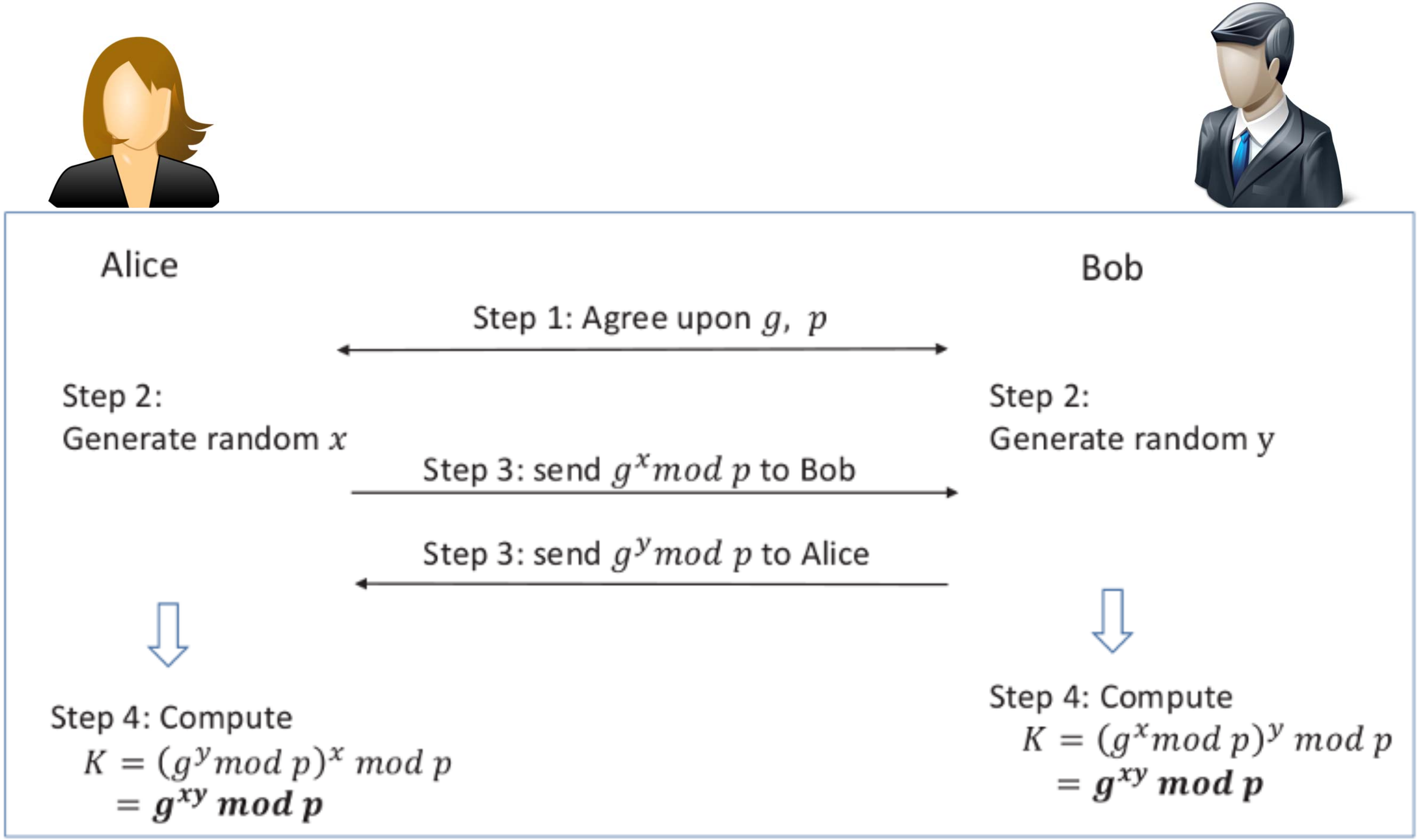
Step 2:
Generate random x

Step 2:
Generate random y

Step 3: send $g^x mod\ p$ to Bob

Step 3: send $g^y mod\ p$ to Alice

Diffie-Hellman Key Exchange *(cont.)*



Turn DH Key Exchange into a Public-Key Encryption Algorithm!

- DH key exchange protocol allows two parties to exchange ***a secret***
- Protocol can be tweaked to turn into a public-key encryption scheme if…
  - **Public key:** known to the public and used for encryption
  - **Private key:** known only to the owner, and used for decryption
  - Establish algorithm(s) for encryption and decryption

Turn DH Key Exchange into a Public-Key Encryption Algorithm!

Alice

Key Generation

Public key: $\boldsymbol{g},\ \boldsymbol{p},\ \boldsymbol{g^x mod\ p}$

Private key: $\boldsymbol{g},\ \boldsymbol{p},\ \boldsymbol{x}$

Alice's Public key →

Bob

Encryption

- Generate a random number y
- Compute $K = (g^x mod\ p)^y\ mod\ p$
  $= g^{xy}\ mod\ p$
- Ciphertext $C = ENCRYPT_K(M)$

← $C,\ g^y\ mod\ p$

Decryption

- Compute $K = (g^y mod\ p)^x\ mod\ p$
  $= g^{xy}\ mod\ p$
- Plaintext $M = DECRYPT_K(C)$

Turn DH Key Exchange into a Public-Key Encryption Algorithm! *(cont.)*

Alice determines $g, p, g^x \bmod p$ — publish!

Alice

Bob

Keep x private

Key Generation

Public key: $\boldsymbol{g,\ p,\ g^x mod\ p}$

Private key: $\boldsymbol{g,\ p,\ x}$

Alice's Public key

Encryption

- Generate a random number y
- Compute $K = (g^x mod\ p)^y\ mod\ p$
  $= g^{xy}\ mod\ p$
- Ciphertext $C = ENCRYPT_K(M)$

$C,\ g^y\ mod\ p$

Decryption

- Compute $K = (g^y mod\ p)^x\ mod\ p$
  $= g^{xy}\ mod\ p$
- Plaintext $M = DECRYPT_K(C)$

Turn DH Key Exchange into a Public-Key Encryption Algorithm! *(cont.)*

Alice

Bob

Key Generation

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

Private key: $g,\ p,\ x$

Alice's Public key →

Encryption

- Generate a random number y
- Compute $K = (g^x \bmod p)^y \bmod p$
  $= g^{xy} \bmod p$
- Ciphertext $C = ENCRYPT_K(M)$

can compute K based on y and params. from Alice's pub. key

use, e.g., AES-CBC to encrypt m

← $C,\ g^y \bmod p$

send to Alice

Decryption

- Compute $K = (g^y \bmod p)^x \bmod p$
  $= g^{xy} \bmod p$
- Plaintext $M = DECRYPT_K(C)$

Turn DH Key Exchange into a Public-Key Encryption Algorithm! *(cont.)*

Alice

Bob

Key Generation
Public key: $g, \ p, \ g^x \bmod p$
Private key: $g, \ p, \ x$

Alice's Public key →

Encryption
- Generate a random number y
- Compute $K = (g^x mod\ p)^y \ mod\ p$
  $= g^{xy} \ mod\ p$
- Ciphertext $C = ENCRYPT_K(M)$

← $C, \ g^y \ mod\ p$

Decryption
- Compute $K = (g^y mod\ p)^x \ mod\ p$
  $= g^{xy} \ mod\ p$
- Plaintext $M = DECRYPT_K(C)$

Given info from Bob, compute K & decrypt C

The RSA Algorithm

This Video Covers:

- **Modulo Operation (no video)**
- Euler's Theorem
- Extended Euclidean Algorithm
- RSA Algorithm
- Examples

Modulo Operation

- The RSA algorithm is based on **modulo operations**

modulus

remainder/residue

$$a \bmod n = r$$

Modulo Operation

- The RSA algorithm is based on **modulo operations**

$$a \bmod \underbrace{n}_{\text{modulus}} = \underbrace{r}_{\text{remainder/residue}}$$

- Examples:
  - $10 \mathtt{\ mod\ } 3 = ?$
  - $15 \mathtt{\ mod\ } 5 = ?$

Modulo Operation

- The RSA algorithm is based on **modulo operations**

$$a \bmod \underbrace{n}_{\text{modulus}} = \underbrace{r}_{\text{remainder/residue}}$$

- Examples:
  - $10 \texttt{ mod } 3 = 1$
  - $15 \texttt{ mod } 5 = 0$

Modulo Operation

- The RSA algorithm is based on **modulo operations**

$$a \bmod \overset{\text{modulus}}{n} = \underset{\text{remainder/residue}}{r}$$

- Examples:
  - $10 \bmod 3 = 1$
  - $15 \bmod 5 = 0$

- Modulo operations are ***distributive***:

$$(a + b) \bmod n = [(a \bmod n) + (b \bmod n)] \bmod n$$
$$a * b \bmod n = [(a \bmod n) * (b \bmod n)] \bmod n$$
$$a^x \bmod n = (a \bmod n)^x \bmod n$$

The RSA Algorithm

This Video Covers:

- Modulo Operation
- **Euler's Theorem (no video)**
- Extended Euclidean Algorithm
- RSA Algorithm
- Examples

Euler's Theorem → easily reduce large powers modulo n

- Euler's totient function φ(n) counts the positive integers up to a given integer n that are *relatively prime* to n
  - $\varphi(n) = n - 1$, if n is a prime number.

- Euler's totient function property:
  - if m and n are relatively prime, $\varphi(mn) = \varphi(m) * \varphi(n)$

- Euler's theorem states:
  - $a^{\varphi(n)} = 1 \pmod{n}$

Euler's Theorem *(cont.)* → easily reduce large powers modulo n

- **Example:** Calculate $4^{100003} \bmod 33$

Euler's Theorem *(cont.)* → easily reduce large powers modulo n

- **Example:** Calculate $4^{100003} \bmod 33$

use $a^{\phi(n)} = 1 \bmod n$
to simplify

$a = 4$
$n = 33$
$\phi(n) = \phi(33) = \cdots$

Euler's Theorem *(cont.)* → easily reduce large powers modulo n

- **Example:** Calculate $4^{100003} \bmod 33$
  - $\varphi(33) = \varphi(3) * \varphi(11) = (3 - 1) * (11 - 1) = 20$
  - $100003 = 5000\varphi(33) + 3$

Euler's Theorem *(cont.)* → easily reduce large powers modulo n

- **Example:** Calculate 4^{100003} mod 33
  - $\varphi(33) = \varphi(3) * \varphi(11) = (3 - 1) * (11 - 1) = 20$
  - $100003 = 5000\varphi(33) + 3$

$$4^{100003} \bmod 33 = 4^{20 \cdot 5000+3} \bmod 33$$

Euler's Theorem *(cont.)* → easily reduce large powers modulo n

- **Example:** Calculate $4^{100003} \bmod 33$
  - $\varphi(33) = \varphi(3) * \varphi(11) = (3 - 1) * (11 - 1) = 20$
  - $100003 = 5000\varphi(33) + 3$

$$
\begin{aligned}
4^{100003} \bmod 33 &= 4^{20 \cdot 5000 + 3} \bmod 33 \\
&= \left(4^{20}\right)^{5000} * 4^{3} \bmod 33
\end{aligned}
$$

Euler's Theorem *(cont.)* → easily reduce large powers modulo n

- **Example:** Calculate 4^{100003} mod 33
  - $\varphi(33) = \varphi(3) * \varphi(11) = (3 - 1) * (11 - 1) = 20$
  - $100003 = 5000\varphi(33) + 3$

$$4^{100003} \bmod 33 = 4^{20\cdot 5000+3} \bmod 33$$
$$= \left(4^{20}\right)^{5000} * 4^{3} \bmod 33$$
$$= \left[\left(4^{20}\right)^{5000} \bmod 33\right] * 4^{3} \bmod 33$$ **(applying distributive rule)**

Euler's Theorem *(cont.)* → **easily reduce large powers modulo n**

- **Example:** Calculate 4^{100003} mod 33
  - $\varphi(33) = \varphi(3) * \varphi(11) = (3 - 1) * (11 - 1) = 20$
  - $100003 = 5000\varphi(33) + 3$

$$4^{100003} \bmod 33 = 4^{20 \cdot 5000+3} \bmod 33$$

$$= \left(4^{20}\right)^{5000} * 4^3 \bmod 33$$

$$= \left[\left(4^{20}\right)^{5000} \bmod 33\right)\right] * 4^3 \bmod 33 \textbf{ (applying distributive rule)}$$

$$= \left[\left(4^{20} \bmod 33\right)\right]^{5000} * 4^3 \bmod 33 \textbf{ (applying distributive rule)}$$

Euler's Theorem *(cont.)* → easily reduce large powers modulo n

- **Example:** Calculate 4^{100003} mod 33
  - $\varphi(33) = \varphi(3) * \varphi(11) = (3 - 1) * (11 - 1) = 20$
  - $100003 = 5000\varphi(33) + 3$

$$
\begin{aligned}
4^{100003} \bmod 33 &= 4^{20\cdot 5000+3} \bmod 33 \\
&= \left(4^{20}\right)^{5000} * 4^{3} \bmod 33 \\
&= \left[\left(\left(4^{20}\right)^{5000} \bmod 33\right)\right] * 4^{3} \bmod 33 \quad \textbf{(applying distributive rule)} \\
&= \left[\left(4^{20} \bmod 33\right)\right]^{5000} * 4^{3} \bmod 33 \quad \textbf{(applying distributive rule)} \\
&= 1^{5000} * 64 \bmod 33 \quad \textbf{(applying Euler's theorem)}
\end{aligned}
$$

Euler's Theorem *(cont.)* → easily reduce large powers modulo n

- **Example:** Calculate 4^{100003} mod 33
  - $\varphi(33) = \varphi(3) * \varphi(11) = (3 - 1) * (11 - 1) = 20$
  - $100003 = 5000\varphi(33) + 3$

$$4^{100003} \bmod 33 = 4^{20 \cdot 5000+3} \bmod 33$$
$$= \left(4^{20}\right)^{5000} * 4^3 \bmod 33$$
$$= \left[\left(\left(4^{20}\right)^{5000} \bmod 33\right)\right] * 4^3 \bmod 33 \text{ (applying distributive rule)}$$
$$= \left[\left(4^{20} \bmod 33\right)\right]^{5000} * 4^3 \bmod 33 \text{ (applying distributive rule)}$$
$$= 1^{5000} * 64 \bmod 33 \text{ (applying Euler's theorem)}$$
$$= 31$$

The RSA Algorithm

This Video Covers:

- Modulo Operation
- Euler's Theorem
- **Extended Euclidean Algorithm (no video)**
- RSA Algorithm
- Examples

RSA and the Extended Euclidean Algorithm

- **Euclid's algorithm:** an efficient method for computing GCD of two #'s
- **Extended Euclidean algorithm:**
  - computes GCD of integers a and b
  - finds integers x and y, such that: $ax + by = g = \gcd(a, b)$

RSA and the Extended Euclidean Algorithm

- **Euclid's algorithm:** an efficient method for computing GCD of two #'s
- **Extended Euclidean algorithm:**
  - computes GCD of integers a and b
  - finds integers x and y, such that: ax + by = g = gcd(a, b)

- RSA uses Extended Euclidean algorithm:
  - e and n are components of public key
  - Find solution to equation:

    $e * x + \varphi(n) * y = \gcd(e, \varphi(n)) = 1$

```
def egcd(a, b):
    if a == 0:
        return (b, 0, 1)
    else:
        g, x, y = egcd(b % a, a)
        return (g, y - (b // a) * x, x)
```

RSA and the Extended Euclidean Algorithm

- **Euclid's algorithm:** an efficient method for computing GCD of two #'s
- **Extended Euclidean algorithm:**
  - computes GCD of integers a and b
  - finds integers x and y, such that: ax + by = g = gcd(a, b)

- RSA uses Extended Euclidean algorithm:
  - e and n are components of public key
  - Find solution to equation:

    $e * x + \varphi(n) * y = \text{gcd}(e, \varphi(n)) = 1$
  - x is private key *(also referred as d)*
  - Equation results: $e * d \bmod \varphi(n) = 1$

```
def egcd(a, b):
    if a == 0:
        return (b, 0, 1)
    else:
        g, x, y = egcd(b % a, a)
        return (g, y - (b // a) * x, x)
```

The RSA Algorithm

This Video Covers:

- Modulo Operation
- Euler's Theorem
- Extended Euclidean Algorithm
- **RSA Algorithm**
- Examples

RSA: Key Generation

Key Generation → *Encryption* → *Decryption*

- **Need to generate:** modulus n, public key exponent e, private key exponent d

- **Approach:**
  - Choose p, q → *large random prime numbers (secret!)*
  - $n = pq$ → *should be LARGE; computationally hard to factor n → Euler's Theorem*
  - Choose e, $1 < e < \varphi(n)$ and e is *relatively prime* to $\varphi(n)$

    → *e is the "public-key exponent" (e.g., e = 65537)*
  - Find d, $ed \bmod \varphi(n) = 1$

    → *solve using the Extended Euclidean Algorithm; d is the "private-key exponent" (secret!)*

    Can be solved in polynomial time if you know p, q, and e!

- **Result:**
  - (e, n) is public key → *without knowledge of p or q, computationally hard to find d*
  - d is private key

RSA: Encryption & Decryption

Key Generation → ***Encryption*** → ***Decryption***

Encryption

- Treat the plaintext as a number
- Assuming $M < n$
- $C = M^e \bmod n$

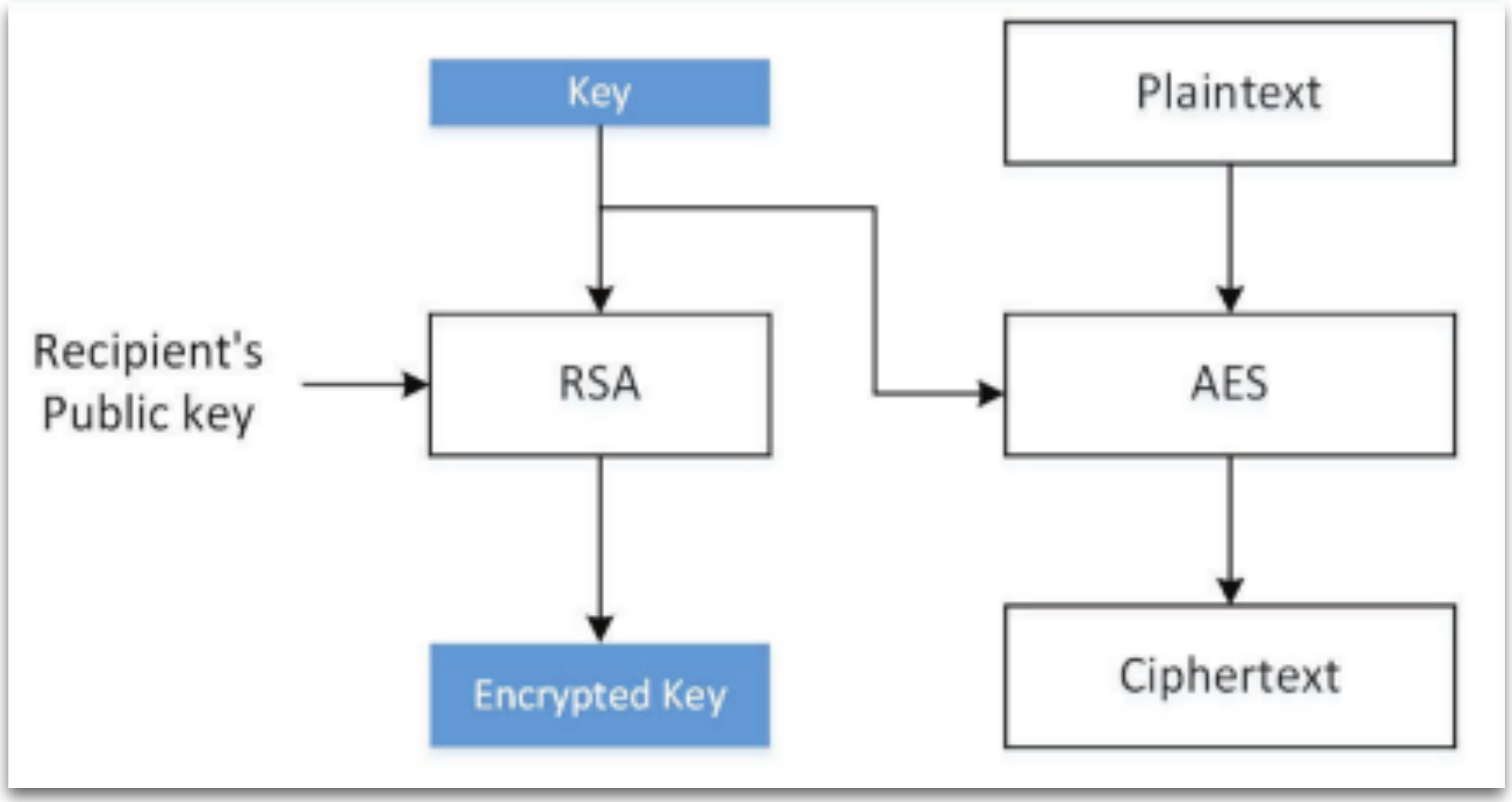
Decryption

- $M = C^d \bmod n$

You can convince yourself (see below) that decryption does indeed yield back the message, M…

$$
\begin{aligned}
M^{ed} \bmod n &= M^{k\phi(n)+1} \bmod n \quad \textbf{(note: } ed = k\phi(n)+1\textbf{)} \\
&= M^{k\phi(n)} * M \quad \bmod n \\
&= \left(M^{\phi(n)} \bmod n\right)^k * M \bmod n \quad \textbf{(applying distributive rule)} \\
&= 1^k * M \bmod n \quad \textbf{(applying Euler's theorem)} \\
&= M
\end{aligned}
$$

Hybrid Encryption



- Public-key encryption is computationally expensive (e.g., large-number multiplications)
- Use public key algorithms to ***exchange a secret session key***
- The key (data-encryption key) used to encrypt data using a symmetric-key algorithm (e.g., AES-128-CBC)

The RSA Algorithm

This Video Covers:

- Modulo Operation
- Euler's Theorem
- Extended Euclidean Algorithm
- RSA Algorithm
- **Examples (no video)**

RSA: Exercise w/ Small Numbers

- Choose two prime numbers $p = 13$ and $q = 17$
- Find e:
  - $n = pq = 221$
  - $\varphi(n) = (p-1)(q-1) = 192$
  - choose $e = 7 \rightarrow$ *7 is relatively prime to* $\varphi(n)$
- Find $\varphi(n)$:
  - $ed \; = 1 \; \texttt{mod} \; \varphi(n)$
- Solving the above equation is equivalent to: $7d + 192y = 1$
- Using Extended Euclidean algorithm, we get $d = 55$ and $y = -2$

RSA: Exercise w/ Small Numbers *(cont.)*

- Encrypt $M = 36$

$$\begin{aligned} M^e \bmod n &= 36^7 \bmod 221 \\ &= \left(36^2 \bmod 221\right)^3 * 36 \bmod 221 \\ &= 191^3 * 36 \ \mathbf{mod}\ 221 \\ &= 179\ \mathbf{mod}\ 221. \end{aligned}$$

- Ciphertext $C = 179$

RSA: Exercise w/ Small Numbers *(cont.)*

$$
\begin{aligned}
C^d \textbf{ mod } n &= 179^{55} \textbf{ mod } 221 \\
&= \left(179^2 \textbf{ mod } 221\right)^{27} * 179 \textbf{ mod } 221 \\
&= 217^{27} * 179 \textbf{ mod } 221 \\
&= \left(217^2 \textbf{ mod } 221\right)^{13} * 217 * 179 \textbf{ mod } 221 \\
&= 16^{13} * 217 * 179 \textbf{ mod } 221 \\
&= \left(16^2 \textbf{ mod } 221\right)^{6} * 16 * 217 * 179 \textbf{ mod } 221 \\
&= 35^{6} * 16 * 217 * 179 \textbf{ mod } 221 \\
&= \left(35^2 \textbf{ mod } 221\right)^{3} * 16 * 217 * 179 \textbf{ mod } 221 \\
&= 120^{3} * 16 * 217 * 179 \textbf{ mod } 221 \\
&= \left(120^2 \textbf{ mod } 221\right) * 120 * 16 * 217 * 179 \textbf{ mod } 221 \\
&= 35 * 120 * 16 * 217 * 179 \textbf{ mod } 221 \\
&= 36 \textbf{ mod } 221
\end{aligned}
$$

RSA: Exercise w/ Large Numbers

Example w/ larger numbers discussed in the text

+

`rsa.c`

Using OpenSSL Tools to Conduct RSA Operations

This Video Covers:

- Generating RSA keys
- Extracting the public key
- Encryption and decryption

OpenSSL Tools: Generating RSA Keys

Example: generate a 1024-bit public/private key pair

- Use **`openssl genrsa`** to generate a file, **`private.pem`**
- private.pem is a Base64 encoding of DER generated binary output

OpenSSL Tools: Generating RSA Keys

Example: generate a 1024-bit public/private key pair

- Use **`openssl genrsa`** to generate a file, **`private.pem`**
- private.pem is a Base64 encoding of DER generated binary output

```
$ openssl genrsa -aes128 -out private.pem 1024 # passphrase csci476
```

OpenSSL Tools: Generating RSA Keys

Example: generate a 1024-bit public/private key pair

- Use **`openssl genrsa`** to generate a file, **`private.pem`**
- private.pem is a Base64 encoding of DER generated binary output

```
$ openssl genrsa -aes128 -out private.pem 1024 # passphrase csci476
$ more private.pem
-----BEGIN RSA PRIVATE KEY-----
Proc-Type: 4,ENCRYPTED
DEK-Info: AES-128-CBC,C30BF6EB3FD6BA9A81CCB9202B95EC1A

sLIQ7Fs5j5zOexdWkZUoiv2W82g03gNERmfG+fwnVnbsIZAuW8E9wiB7tqz8rEL+
xfL+U20lyQNxpmOTUeKlN3qCcJROcGYSNd1BeNpgLWV1bN5FPYce9GRb4tFr4bhK
...
RPtJNKUryhVnAC4a3gp0gcXk1IQLeHeyKQCPQ1SckQRdrBzHjjCNN42NlCVEpcsF
WJ8ikqDd9FslGHc1PT6ktW5oV9cB8G2wfo7D85n91SQfSzuwAcyx7Ecir1o4PfKG
-----END RSA PRIVATE KEY——
```

OpenSSL Tools: Generating RSA Keys *(cont.)*

The ***actual*** content of **`private.pem`**:

```
$ openssl rsa -in private.pem -noout -text
```

OpenSSL Tools: Generating RSA Keys *(cont.)*

The ***actual*** content of **`private.pem`**:

```
$ openssl rsa -in private.pem -noout -text
Enter pass phrase for private.pem: csci476
Private-Key: (1024 bit)
modulus:
    00:b8:52:5c:25:cc:7c:f2:ef:a6:35:9d:de:3d:5d: ...
publicExponent: 65537 (0x10001)
privateExponent:
    4b:0d:ce:53:dd:e6:6b:0d:c6:82:42:9c:42:24:a7: ...
prime1:
    00:ef:14:46:57:9c:d0:4c:98:de:c3:0b:aa:d8:72: ...
prime2:
    00:c5:5d:f8:0b:f9:75:dc:88:ea:d4:d0:56:ee:f9: ...
exponent1:
    00:e6:49:9a:44:14:19:94:5e:7f:dc:52:65:bb:5d: ...
exponent2:
    7c:ad:77:dc:58:a2:13:c6:8a:52:15:aa:55:1c:22: ...
coefficient:
    3a:7c:b9:a0:12:e8:fa:88:b8:6f:38:4a:ed:bc:17: ...
```

OpenSSL Tools: Generating RSA Keys *(cont.)*

The ***actual*** content of **`private.pem`**:

```
$ openssl rsa -in private.pem -noout -text
Enter pass phrase for private.pem: csci476
Private-Key: (1024 bit)
modulus:
    00:b8:52:5c:25:cc:7c:f2:ef:a6:35:9d:de:3d:5d: ...
publicExponent: 65537 (0x10001)
privateExponent:
    4b:0d:ce:53:dd:e6:6b:0d:c6:82:42:9c:42:24:a7: ...
prime1:
    00:ef:14:46:57:9c:d0:4c:98:de:c3:0b:aa:d8:72: ...
prime2:
    00:c5:5d:f8:0b:f9:75:dc:88:ea:d4:d0:56:ee:f9: ...
exponent1:
    00:e6:49:9a:44:14:19:94:5e:7f:dc:52:65:bb:5d: ...
exponent2:
    7c:ad:77:dc:58:a2:13:c6:8a:52:15:aa:55:1c:22: ...
coefficient:
    3a:7c:b9:a0:12:e8:fa:88:b8:6f:38:4a:ed:bc:17: ...
```

n

e

d

p

q

Chinese remainder theorem

OpenSSL Tools: Extracting the Public Key

The ***actual*** content of **`public.pem`**:

```
$ openssl rsa -in private.pem -pubout > public.pem
Enter pass phrase for private.pem: csci476
writing RSA key
$ more public.pem
-----BEGIN PUBLIC KEY-----
MIGfMA0GCSqGSIb3DQEBAQUAA4GNADCBiQKBgQC4UlwlzHzy76Y1nd49XakNUwqJ
Ud3ph0uBWWfnLnjIYgQL/spg9WE+1Q1YPp2t3FBFljhGHdWMA8abfNXG4jmpD+uq
Ix0WVyXg12WWi1kY2/vs8xI1K+PumWTtq8R8ueAq7RzETc3873DO1vjMxXWqau7k
zIkUuJ/JCjzjYfbsDQIDAQAB
-----END PUBLIC KEY-----
```

OpenSSL Tools: Extracting the Public Key

The ***actual*** content of **`public.pem`**:

```
$ openssl rsa -in private.pem -pubout > public.pem
Enter pass phrase for private.pem: csci476
writing RSA key
$ more public.pem
-----BEGIN PUBLIC KEY-----
MIGfMA0GCSqGSIb3DQEBAQUAA4GNADCBiQKBgQC4UlwlzHzy76Y1nd49XakNUwqJ
Ud3ph0uBWWfnLnjIYgQL/spg9WE+1Q1YPp2t3FBFljhGHdWMA8abfNXG4jmpD+uq
Ix0WVyXg12WWi1kY2/vs8xI1K+PumWTtq8R8ueAq7RzETc3873DO1vjMxXWqau7k
zIkUuJ/JCjzjYfbsDQIDAQAB
-----END PUBLIC KEY-----
```

```
$ openssl rsa -in public.pem -pubin -text -noout
Public-Key: (1024 bit)
Modulus:
    00:b8:52:5c:25:cc:7c:f2:ef:a6:35:9d:de:3d:5d: ...
Exponent: 65537 (0x10001)
```

n

e

→ (e,n) = public key!

OpenSSL Tools: Encryption and Decryption

- Create a plaintext message:

```
$ echo "This is a secret." > msg.txt
```

- Encrypt the plaintext:

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

OpenSSL Tools: Encryption and Decryption

- Create a plaintext message:

```
$ echo "This is a secret." > msg.txt
```

- Encrypt the plaintext:

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

- Decrypt the ciphertext:

```
$ openssl rsautl -decrypt -inkey private.pem -in msg.enc
Enter pass phrase for private.pem: csci476
This is a secret.
```

RSA and Padding

This Video Covers:

- *Why* and *how* padding is done in RSA
- Examples with OpenSSL

RSA and Padding

- Secret-key encryption uses encryption modes to encrypt plaintext longer than block size.
- RSA used in hybrid approach (Content key length << RSA key length)

- To encrypt:
  - **short plaintext:** treat it as a number, raise it to the power of e (modulo n)
  - **large plaintext:** use hybrid approach; treat the "content key" as a number and raise it to the power of e (modulo n)

Treating plaintext as a number and directly applying RSA is called plain RSA or textbook RSA

Attacks Against Textbook RSA

- RSA is a ***deterministic*** encryption algorithm
  - The same plaintext encrypted using the same public key gives the same ciphertext
  - secret-key encryption uses randomized IV →different ciphertexts for same plaintext

- For ***small*** e and m
  - if $m^e <$ modulus n
  - e-th root of ciphertext gives plaintext

- If same plaintext is encrypted e times or more using the same e but different n, then it is easy to decrypt the original plaintext message via the Chinese remainder theorem

Padding Schemes: **PKCS#1 v1.5** and **OAEP**

- The simple fix to defend against previous attacks is to ***add randomness*** to the plaintext ***before encryption*** → ***padding!***

- **Types of padding:**
  - **PKCS#1** (up to version 1.5); weakness discovered since 1998
  - Optimal Asymmetric Encryption Padding (**OAEP**); prevents attacks on PKCS

- `rsautl` command provides options for both types of paddings *(PKCS#1 v1.5 is the default… why? IDK…)*

PKCS Padding

```
$ cat msg.txt
This is a secret.

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

PKCS Padding

```
$ cat msg.txt
This is a secret.

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

$ openssl rsautl -decrypt -inkey private.pem -in msg.enc -out newmsg.txt -raw
Enter pass phrase for private.pem: csci476
```

PKCS Padding

```
$ cat msg.txt
This is a secret.

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

$ openssl rsautl -decrypt -inkey private.pem -in msg.enc -out newmsg.txt -raw
Enter pass phrase for private.pem: csci476

$ xxd newmsg.txt
00000000: 0002 a6dc c092 9a2e 4a8e 3849 c14f cf0b  .........J.8I.O..
00000010: b036 de51 b222 28ab 1b98 6018 5e04 b084  .6.Q."(...`.^...
00000020: 31fc c2ef 680f a4f7 07c9 2b04 8d84 089d  1...h.....+.....
00000030: a2f3 5bbc 2f82 2969 18a1 6c09 2762 82a6  ..[./.)i..l.'b..
00000040: 7d26 b7e0 1a41 077b 86a8 4459 9a0d 6b61  }&...A.{..DY..ka
00000050: af55 a61d 0101 8f26 1ed1 cc3b 33c9 74db  .U.....&...;3.t.
00000060: bad1 38a4 dd0e 59b5 8097 4d93 a400 5468  ..8...Y...M...Th
00000070: 6973 2069 7320 6120 7365 6372 6574 2e0a  is is a secret..

$ ls -al msg.enc
-rw-rw-r-- 1 seed seed 128 Mar 18 14:29 msg.enc
```

OAEP Padding

- Original plaintext is not directly copied into the encryption block
- Plaintext is first XORed with a value derived from random padding data

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

OAEP Padding

- Original plaintext is not directly copied into the encryption block
- Plaintext is first XORed with a value derived from random padding data

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

$ openssl rsautl -decrypt -inkey private.pem -in msg.enc -out newmsg_oaep.txt -raw
Enter pass phrase for private.pem: csci476

$ xxd newmsg_oaep.txt
00000000: 00cd 119c 1376 6ea4 bb17 cd2e 5462 52a1  .....vn.....TbR.
00000010: 4dd1 2031 f446 c3ea f000 55b2 785d 86ba  M. 1.F....U.x]..
00000020: 97af dba7 4ee1 cd02 5fa3 4752 488d f523  ....N..._.GRH..#
00000030: 9d7c c69b f1a8 dba2 c4d1 9c14 f0f1 4abe  .|............J.
00000040: 3c1c e904 711d 0944 2f0b 8b72 7f82 06dc  <...q..D/..r....
00000050: 50af bf94 cac1 b402 7522 7d17 6fc8 699d  P.......u"}.o.i.
00000060: e4ab fff9 952a fb47 673e 7bf5 729f 96bb  .....*.Gg>{.r...
00000070: c282 b678 15c5 2a22 5ae6 bcf1 51be 1a2e  ...x..*"Z...Q...
```

OAEP Padding

- Original plaintext is not directly copied into the encryption block
- Plaintext is first XORed with a value derived from random padding data

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

$ openssl rsautl -decrypt -inkey private.pem -in msg.enc -out newmsg_oaep.txt -raw
Enter pass phrase for private.pem: csci476

$ xxd newmsg_oaep.txt
00000000: 00cd 119c 1376 6ea4 bb17 cd2e 5462 52a1  .....vn.....TbR.
00000010: 4dd1 2031 f446 c3ea f000 55b2 785d 86ba  M. 1.F....U.x]..
00000020: 97af dba7 4ee1 cd02 5fa3 4752 488d f523  ....N..._.GRH..#
00000030: 9d7c c69b f1a8 dba2 c4d1 9c14 f0f1 4abe  .|............J.
00000040: 3c1c e904 711d 0944 2f0b 8b72 7f82 06dc  <...q..D/..r....
00000050: 50af bf94 cac1 b402 7522 7d17 6fc8 699d  P.......u"}.o.i.
00000060: e4ab fff9 952a fb47 673e 7bf5 729f 96bb  .....*.Gg>{.r...
00000070: c282 b678 15c5 2a22 5ae6 bcf1 51be 1a2e  ...x..*"Z...Q...

# NOTE: decrypt without -raw to recover the original data (need -oaep flag!)
$ openssl rsautl -decrypt -inkey private.pem -in msg.enc -out newmsg_oaep.txt -oaep
Enter pass phrase for private.pem: csci476
This is a secret.
```

Digital Signatures

This Video Covers:

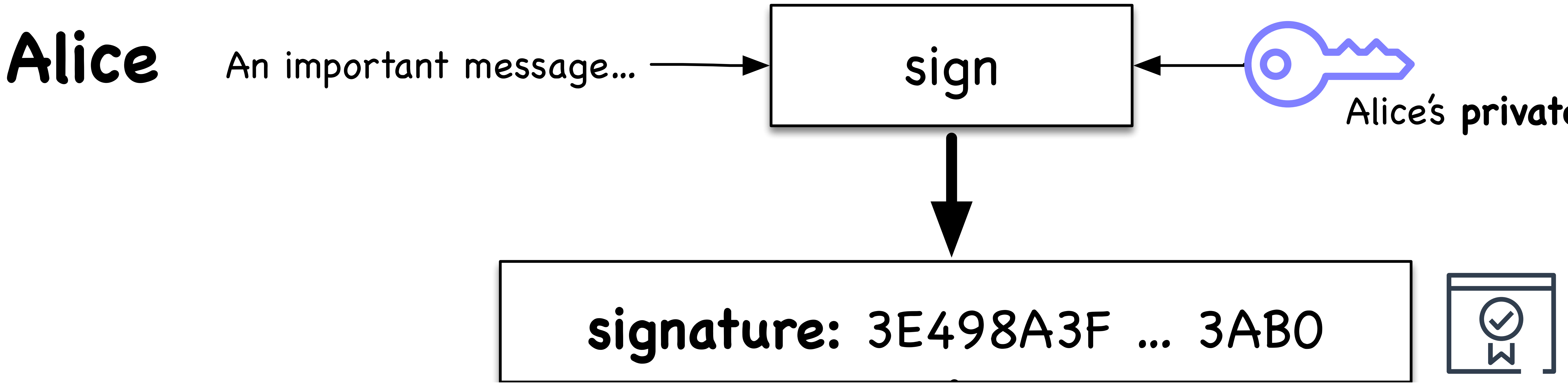
- Digital signatures: *what are they?* and *how do they work?*
- Examples with OpenSSL
- Experiment: attacks on digital signatures

Digital Signatures

- **Goal:** provide *proof of authenticity* by signing digital documents
    - Diffie-Hellman authors proposed the idea, but no concrete solution
    - RSA authors developed the first digital signature algorithm

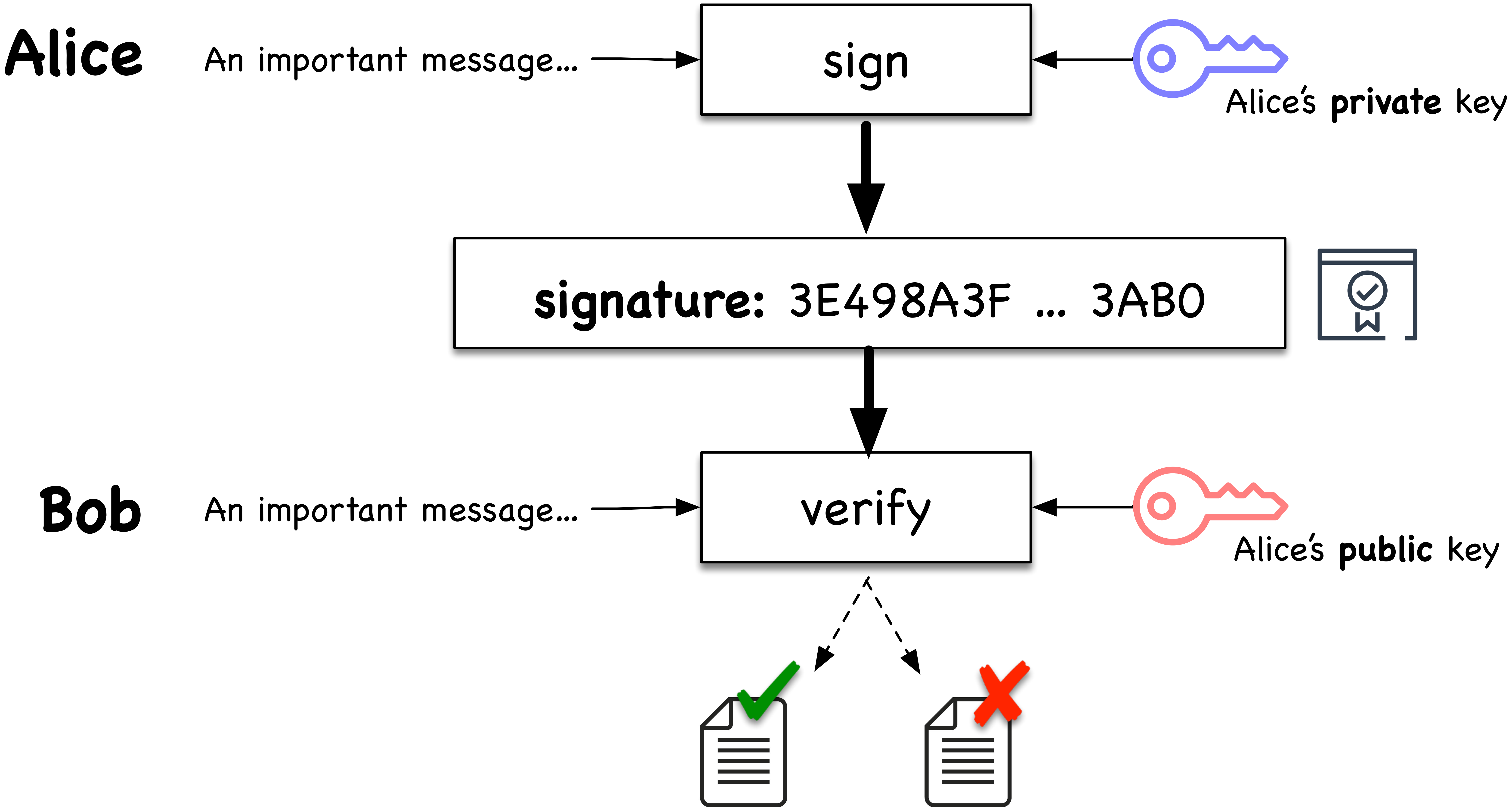
Digital Signatures

- **Goal:** provide *proof of authenticity* by signing digital documents
  - Diffie-Hellman authors proposed the idea, but no concrete solution
  - RSA authors developed the first digital signature algorithm



Digital Signatures

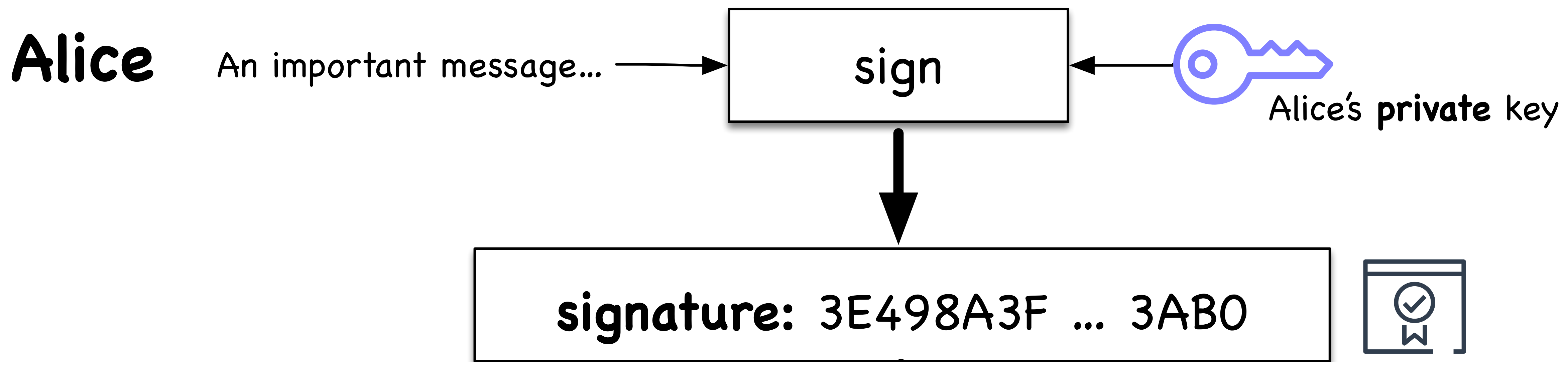
- **Goal:** provide *proof of authenticity* by signing digital documents
    - Diffie-Hellman authors proposed the idea, but no concrete solution
    - RSA authors developed the first digital signature algorithm



Digital Signature using RSA

- Apply private-key operation on m using private key, and get a number s (anyone can get m back from s using the public key)

- To sign a message m:
  - Digital signature = $m^d \bmod n$

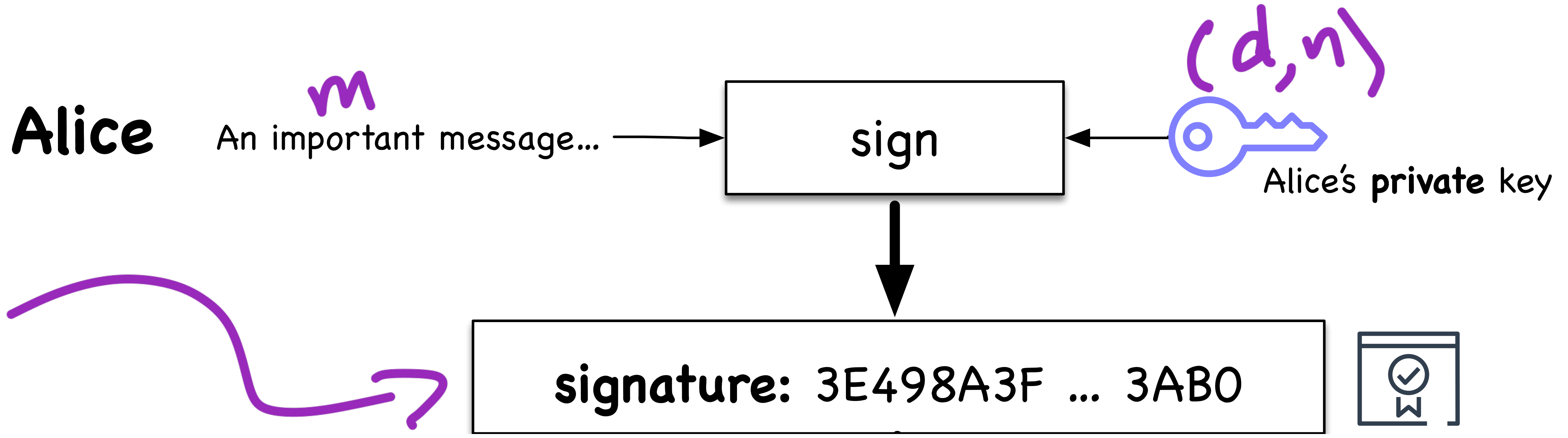


- In practice, a message may be long resulting in a long signature and more computing time… → Instead, generate a cryptographic hash value from the original message, and ***only sign the hash***

Digital Signature using RSA

- Apply private-key operation on m using private key, and get a number s (anyone can get m back from s using the public key)

- To sign a message m:
  - Digital signature = $m^d \ \mathtt{mod}\ n = s$



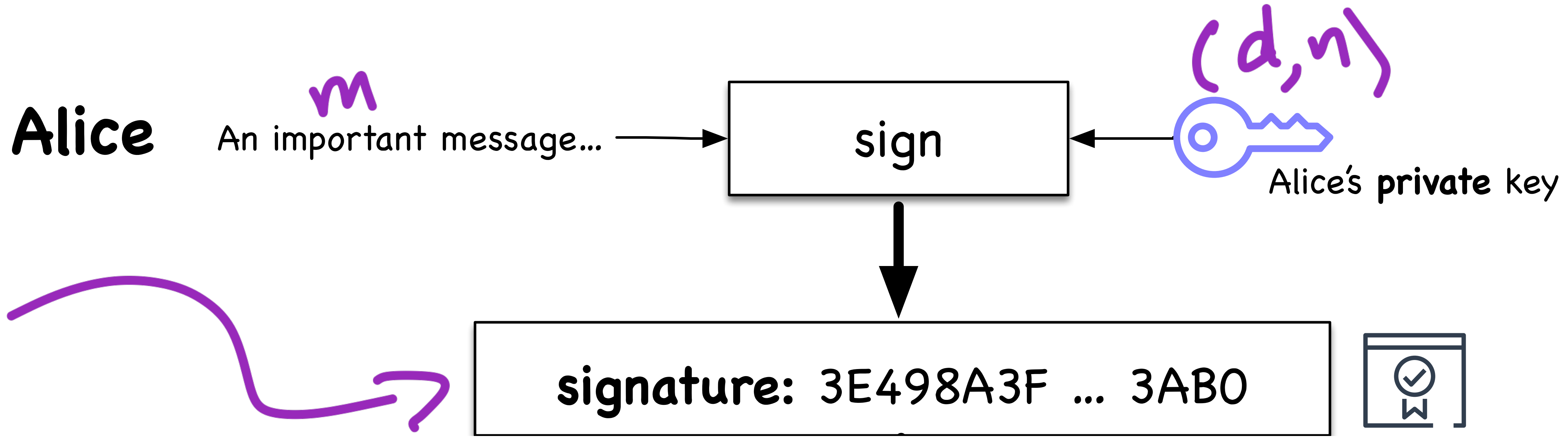
- In practice, a message may be long resulting in a long signature and more computing time… → Instead, generate a cryptographic hash value from the original message, and ***only sign the hash***

Digital Signature using RSA

- Apply private-key operation on m using private key, and get a number s (anyone can get m back from s using the public key)

- To sign a message m:
  - Digital signature = $m^d \bmod n = s$

to verify: $s^e \bmod n = m$

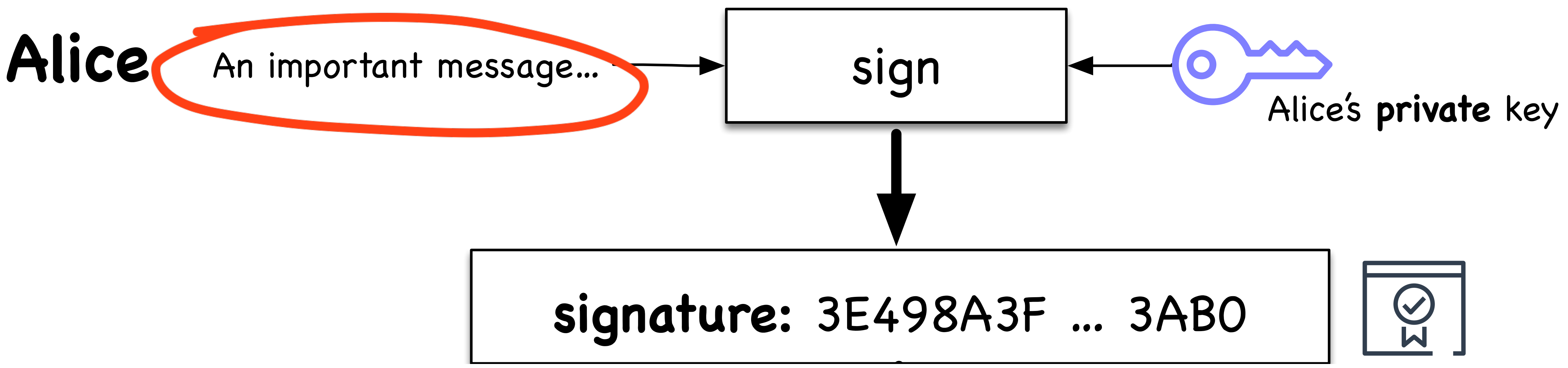


- In practice, a message may be long resulting in a long signature and more computing time… → Instead, generate a cryptographic hash value from the original message, and ***only sign the hash***

Digital Signature using RSA

- Apply private-key operation on m using private key, and get a number s (anyone can get m back from s using the public key)

- To sign a message m:
  - Digital signature = $m^d \bmod n$



- In practice, a message may be long resulting in a long signature and more computing time… → Instead, generate a cryptographic hash value from the original message, and ***only sign the hash***

Digital Signature using RSA *(cont.)*

To generate a hash of the message:

```
# Generate a sha256 hash of the secret message
$ openssl sha256 -binary msg.txt > msg.sha256

$ xxd msg.sha256
00000000: 8272 61ce 5ddc 974b 1b36 75a3 ed37 48cd  .ra.]..K.6u..7H.
00000010: 83cd de93 85f0 6aab bd94 f50c db5a b460  ......j......Z.`
```

Digital Signature using RSA *(cont.)*

To generate and verify the signature:

```
# Sign the hash
$ openssl rsautl -sign -inkey private.pem -in msg.sha256 -out msg.sig

# Verify the signature
$ openssl rsautl -verify -inkey public.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  ................
00000020: ffff ffff ffff ffff ffff ffff ffff ffff  ................
00000030: ffff ffff ffff ffff ffff ffff ffff ffff  ................
00000040: ffff ffff ffff ffff ffff ffff ffff ffff  ................
00000050: ffff ffff ffff ffff ffff ffff ffff ff00  ................
00000060: 8272 61ce 5ddc 974b 1b36 75a3 ed37 48cd  .ra.]..K.6u..7H.
00000070: 83cd de93 85f0 6aab bd94 f50c db5a b460  ......j......Z.`
```

Digital Signature using RSA *(cont.)*

To generate and verify the signature:

```
# Sign the hash
$ openssl rsautl -sign -inkey private.pem -in msg.sha256 -out msg.sig

# Verify the signature
$ openssl rsautl -verify -inkey public.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  ................
00000020: ffff ffff ffff ffff ffff ffff ffff ffff  ................
00000030: ffff ffff ffff ffff ffff ffff ffff ffff  ................
00000040: ffff ffff ffff ffff ffff ffff ffff ffff  ................
00000050: ffff ffff ffff ffff ffff ffff ffff ff00  ................
00000060: 8272 61ce 5ddc 974b 1b36 75a3 ed37 48cd  .ra.]..K.6u..7H.
00000070: 83cd de93 85f0 6aab bd94 f50c db5a b460  ......j......Z.`
```

```
$ xxd msg.sha256
00000000: 8272 61ce 5ddc 974b 1b36 75a3 ed37 48cd  .ra.]..K.6u..7H.
00000010: 83cd de93 85f0 6aab bd94 f50c db5a b460  ......j......Z.`
```

Attack Experiment on Digital Signatures

- Attackers cannot generate a valid signature from a modified message because they do not know the private key
- If an attacker modifies the message, the hash will change, and therefore the signature of the hash will change, so the signature verification should fail

Experiment:

Modify 1 bit of the signature file `msg.sig`
and verify the signature…

Attack Experiment on Digital Signatures

After applying the RSA public key on the signature,
we get a block of data that is significantly different

```
# Verify the signature
$ openssl rsautl -verify -inkey public.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  ................
00000020: ffff ffff ffff ffff ffff ffff ffff ffff  ................
00000030: ffff ffff ffff ffff ffff ffff ffff ffff  ................
00000040: ffff ffff ffff ffff ffff ffff ffff ffff  ................
00000050: ffff ffff ffff ffff ffff ffff ffff ff00  ................
00000060: 8272 61ce 5ddc 974b 1b36 75a3 ed37 48cd  .ra.]..K.6u..7H.
00000070: 83cd de93 85f0 6aab bd94 f50c db5a b460  ......j......Z.`
```

After modifying only 1 bit…

```
$ openssl rsautl -verify -inkey public.pem -in msg.sig -pubin -raw | xxd
00000000: 07a4 8d1c cfb8 b36c 17af e821 a9ea 8c80  .......l...!....
00000010: c654 74b0 afb1 c1d8 616c 9dca 5138 3b9d  .Tt.....al..Q8;.
00000020: 8111 234e d20f 033f 07f2 7f7c a88e 4fb1  ..#N...?...|..O.
00000030: 14e0 8132 6b6e ae1e 2a4c be54 ff61 f2e6  ...2kn..*L.T.a..
00000040: 965e 492c 428a 2cd3 8c07 7764 480d 2697  .^I,B.,...wdH.&.
00000050: db36 f2a4 7916 27aa 8a07 17c4 d94a 1f06  .6..y.'......J..
00000060: 2632 cf4b fb2c e98f fb68 cbe1 b084 3bb1  &2.K.,...h....;.
00000070: bb98 651c 0469 14f5 2f92 0e91 93d7 2d09  ..e..i../.....-.
```

Summary: *Public Key Cryptography*

- The basics of public key cryptography
- Both theoretical and practical sides of public key cryptography
- RSA algorithm and the Diffie-Hellman Key Exchange
- Tools and programming libraries to conduct public-key operations
- How public key is used in real-world applications

You Try!

Exam-like problems that you can use for practice!

- In the Diffie-Hellman key exchange, Alice sends $g^x \texttt{ mod } p$ to Bob, and Bob sends $g^y \texttt{ mod } p$ to Alice. How do they get a common secret?
- In the Diffie-Hellman key exchange protocol, attackers know g, p, and $g^x \texttt{ mod } p$. Why can't attackers figure out x from this data?
- In RSA, decryption is much more expensive than encryption, why?
- Why do we use hybrid encryption? Why can't we use public key to encrypt everything?
- What is the benefit of public-key based authentication scheme, compared to the password-based scheme?