CSE127, Computer Security

Cryptography II

UC San Diego

Housekeeping

General course things to know

- PA5 released!
  - Focuses on Cryptography!
  - A little delay in release, so extended deadline… now due **3/14 EOD**
- Note, due to travel class is cancelled on **3/10**
- **Final exam time:** Thursday, **3/19** at **8am**
- **Final exam location:** Mosaic Lecture Hall 113 (has 250 seats so we don't need to be so cramped!)
- SETs are now available! We will take some time in the last lecture to fill them out. But please do them, they are very helpful to me.

Previously on CSE 127…

Recap

- Introduction to cryptography
  - Two classes — symmetric (two parties use a shared key) and asymmetric (each party has a public / private key for different things)
  - Two primary functions — encryption (confidentiality) and MAC / signature (integrity)

Recap quiz….

- Alice wants to send Bob an encrypted message with asymmetric cryptography. How can she do this?

Recap quiz….

- Alice wants to send Bob an encrypted message with asymmetric cryptography. How can she do this?
  - Encrypt the message with Bob’s public key! Why?

Recap quiz….

- Alice wants to send Bob an encrypted message with asymmetric cryptography. How can she do this?
  - Encrypt the message with Bob’s public key!
- Bob wants to send Alice message with his signature attached with asymmetric cryptography. How can he do this?

Recap quiz….

- Alice wants to send Bob an encrypted message with asymmetric cryptography. How can she do this?
  - Encrypt the message with Bob’s public key!
- Bob wants to send Alice message with his signature attached with asymmetric cryptography. How can he do this?
  - Sign message with his private key! Why?

Recap quiz….

- Alice wants to send Bob an encrypted message with asymmetric cryptography. How can she do this?
  - Encrypt the message with Bob’s public key!
- Bob wants to send Alice message with his signature attached with asymmetric cryptography. How can he do this?
  - Sign message with his private key!
- Alice wants to send Bob an encrypted message with their shared key *k*. She’s considering using AES with ECB mode. Is this a good idea?

Recap quiz….

- Alice wants to send Bob an encrypted message with asymmetric cryptography. How can she do this?
  - Encrypt the message with Bob’s public key!
- Bob wants to send Alice message with his signature attached with asymmetric cryptography. How can he do this?
  - Sign message with his private key!
- Alice wants to send Bob an encrypted message with their shared key *k*. She’s considering using AES with ECB mode. Is this a good idea?
  - No! Remember the Linux penguin. She should use a different mode.

Today's lecture — More Cryptography

Learning Objectives

- Recap some modular arithmetic properties, understand a little bit of the math behind the cryptography we use
- Discuss the fundamental mechanisms of **key exchange**, and how we do key exchange today (e.g., Diffie-Hellman key exchange)
- Discuss RSA and some of the problems with using RSA in practice

Key Exchange

Modular Arithmetic

- Modular arithmetic is the primary mechanisms of a lot of cryptography
- What does the statement: $a \equiv b \pmod{n}$ mean?

Modular Arithmetic

- Modular arithmetic is the primary mechanisms of a lot of cryptography
- What does the statement: a ≡ b (mod n) mean?
  - Both **a** and **b** are numbers that, when divided by *n*, have the same remainder
- Some other important modular properties:
  - (a mod d) + (b mod d) ≡ (a + b) mod d
  - (a mod d) - (b mod d) ≡ (a - b) mod d
  - (a mod d) * (b mod d) ≡ (a * b) mod d

Modular Inverse: “Division” for mod

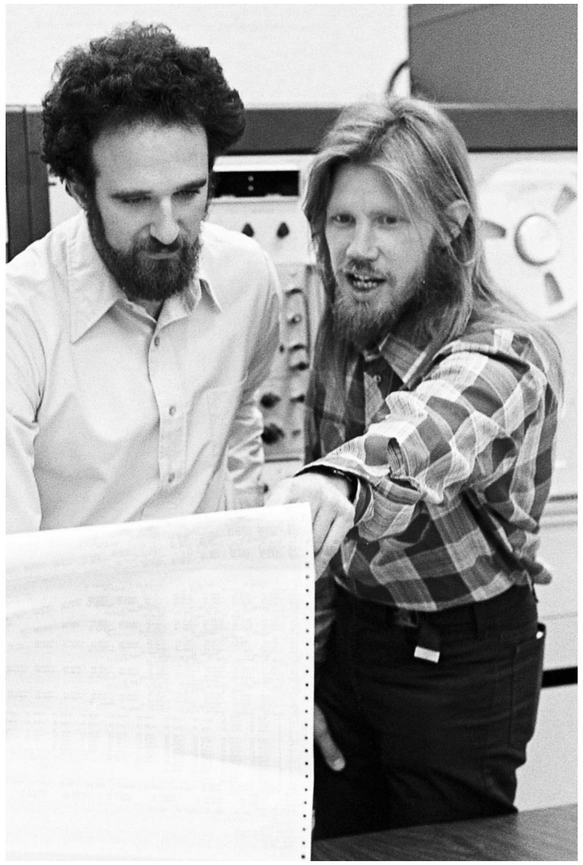
- Modular inverse of integer *a* mod *m* is an integer *x* such that a*x ≡ 1 (mod *m*)
  - *(a*x)* - 1 is divisible by *m*
- **Modular inverses do not always exist;** they only exist when the **gcd(a, m) = 1**
  - Why? You can think of this like division…. you need to get *one unique value* for the inverse; if the gcd(a, x) > 1, then you don’t know how to divide a
- Examples
  - a = 3, m = 7, gcd(3, 7) = 1… we can find 3 * **5** = 15 ≡ 1 (mod 7)
  - a = 4, m = 8; gcd(4, 8) = 4… no multiple of 4 will result in 1 (mod 8)
- **Efficient to compute modular inverse!**

Modular Exponentiation

- Over the integers… $g^a = g * g * g * g \ldots\ldots * g$
- $g^a \pmod d = g^a \pmod d * (((g \bmod d) * g \bmod d) * \ldots. g \bmod d) \bmod d$
- Notably….
  - $(g^a \pmod d)^b = g^{ab} \bmod d$
  - This is used as a building block for Diffie Hellman key exchange (we'll see this in a bit)
- **Efficient to compute modular exponents!**

Discrete Log (Inverse of Modular Exponentiation)

- "Inverse" of modular exponentiation: Discrete log
- What is a logarithm?
  - $b^a = y$; $log_b y = a$
- Discrete log is logarithm in modular arithmetic… if you have *b*, *d*, *y*; discrete log is
  - *a* such that, $b^a \equiv y \bmod d$
  - **There is no known polynomial-time algorithm to compute this when d is a large prime… easy to compute in forward direction, hard to reverse!**



“We stand today on the brink of a revolution in cryptography.”

New Directions in Cryptography

Invited Paper

WHITFIELD DIFFIE AND MARTIN E. HELLMAN, MEMBER, IEEE

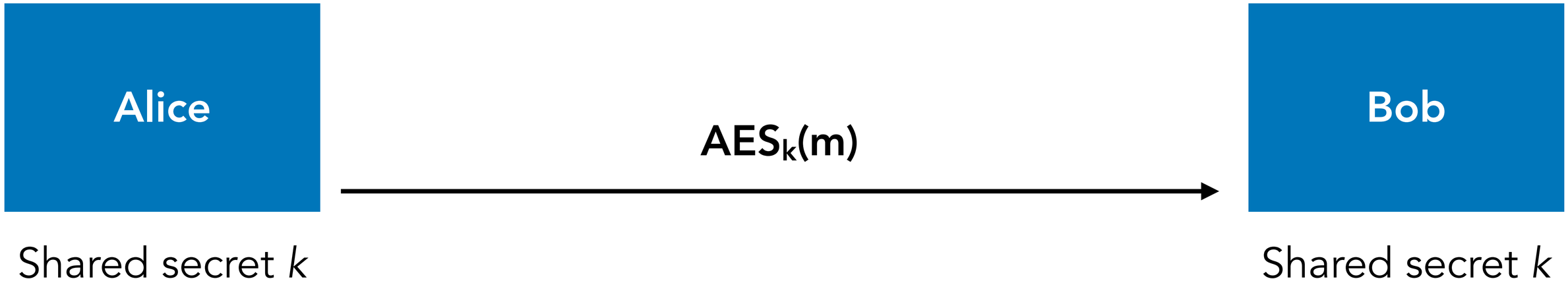
Sharing Symmetric Keys

- What we want to do: **symmetric key encryption** — using a shared secret key for cryptography
- Why it's hard: **We need to figure out how to share a key.** Any ideas?

Sharing Symmetric Keys

- What we want to do: **symmetric key encryption** — using a shared secret key for cryptography
- Why it's hard: **We need to figure out how to share a key.** Any ideas?
  - Trusted third party intermediary [requires trust]
  - Meet offline to agree on key [difficult to do every time you want to establish a new key]

Enter: Key Exchange



Can't do this without pre-sharing *k*.

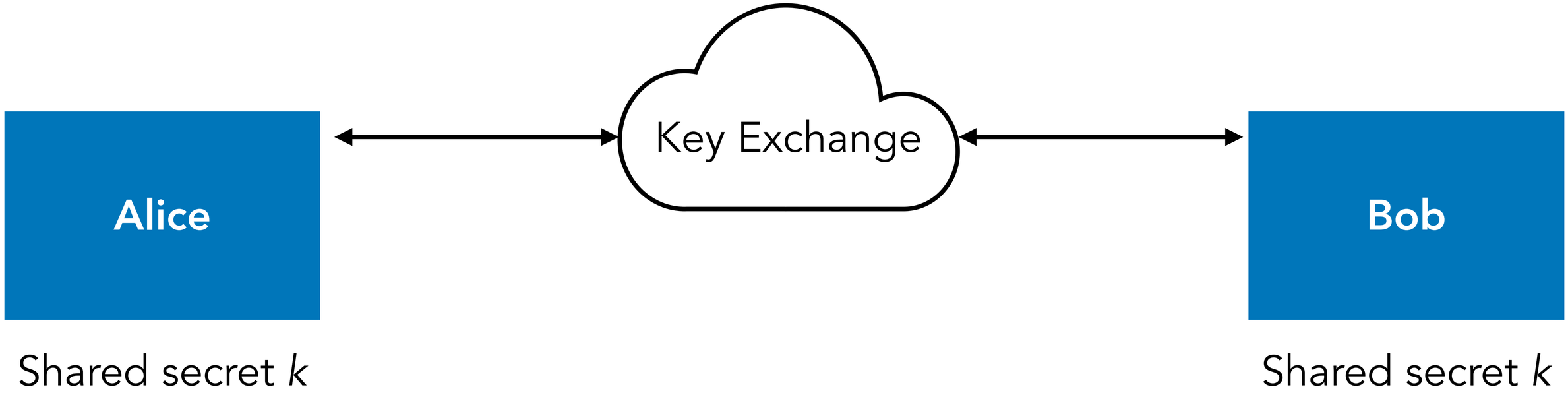
Enter: Key Exchange

Alice

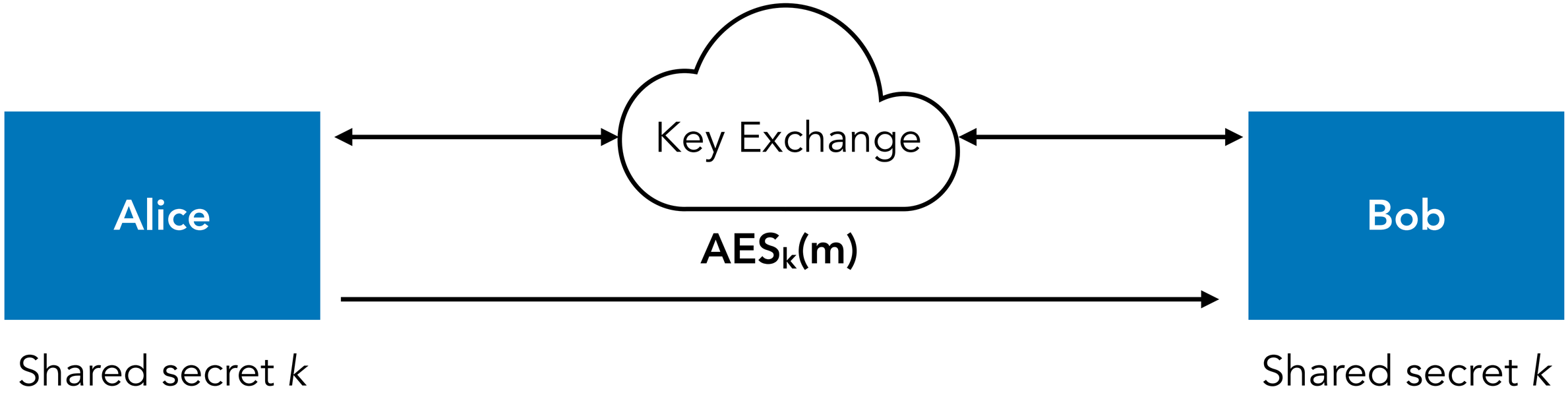
Bob

Enter: Key Exchange

Enter: Key Exchange



Enter: Key Exchange



Textbook Diffie-Hellman Key Exchange

Public Parameters:

p a prime number
g a generator… integer mod p

Alice

Bob

Textbook Diffie-Hellman Key Exchange

Public Parameters:

p a prime number
g a generator… integer mod p

Alice

Bob

Alice picks a random secret number *a*

Textbook Diffie-Hellman Key Exchange

Public Parameters:

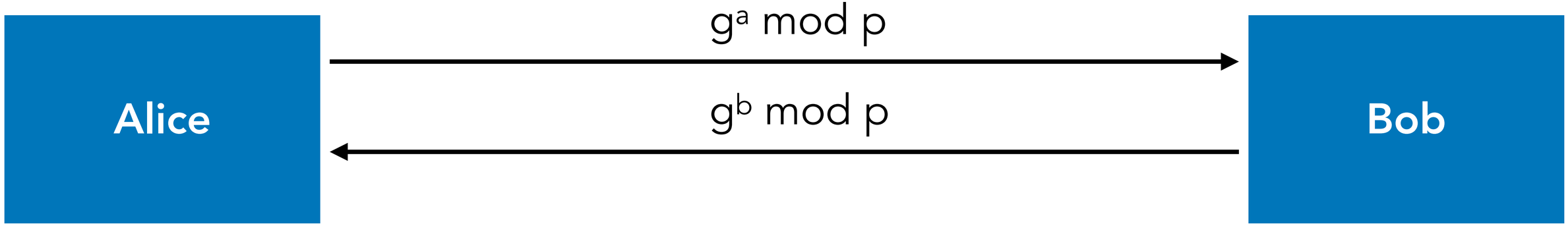
p a prime number
g a generator… integer mod p

Alice picks a random secret number *a*

Textbook Diffie-Hellman Key Exchange

Public Parameters:

p a prime number
g a generator… integer mod p



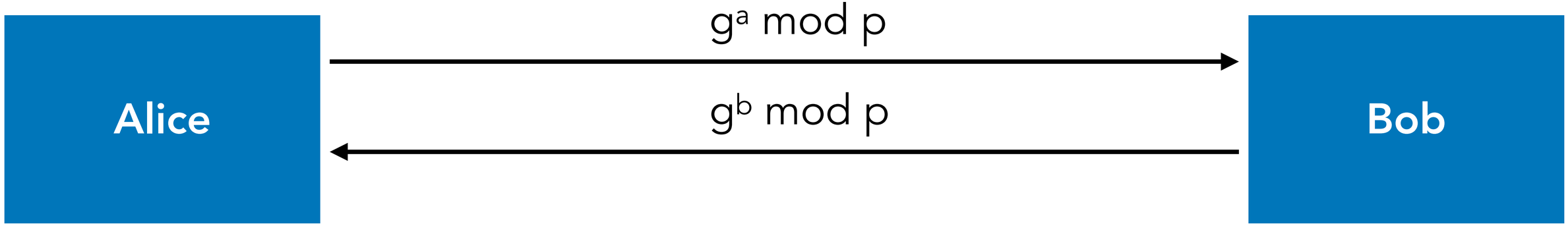
Alice picks a random secret number *a*

Bob picks a random secret number *b*

Textbook Diffie-Hellman Key Exchange

Public Parameters:

p a prime number
g a generator… integer mod p



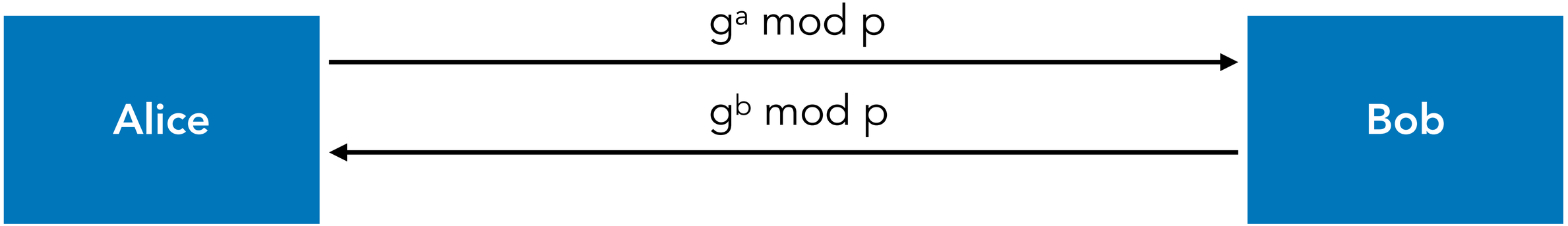
Alice picks a random secret number *a*

Bob picks a random secret number *b*

Alice and Bob both compute g^{ab} (mod p); simply by exponentiating by their secret

$$\mathbf{(g^a \bmod p)^b = (g^b \bmod p)^a = g^{ab} \pmod p}$$

Diffie-Hellman Security



Alice picks a random secret number *a*

Bob picks a random secret number *b*

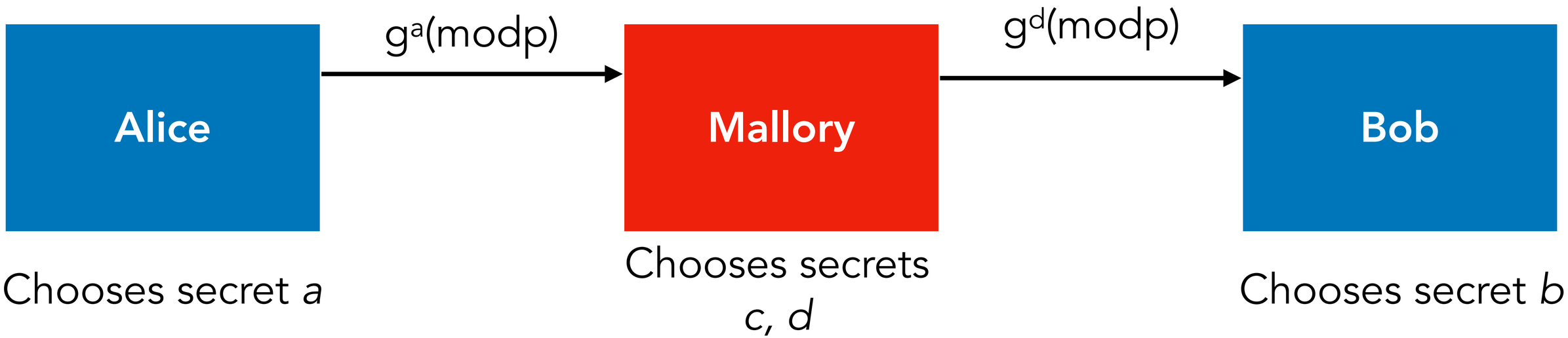
- DH key-exchange is **resistant** against a passive adversary; protected by *discrete-log hardness*
- But, parameter selection is important: *p > 2048-bits*, *a* and *b* roughly the same
- **Do not implement yourself! Discrete log only hard for certain g, p values.**
  - Best implementation choice: x25519 Diffie-Hellman (elliptic curve DH)

Diffie-Hellman Security

- Textbook Diffie-Hellman gives you protection against passive adversary, but…
  - **Weak to an active MiTM attacker.**

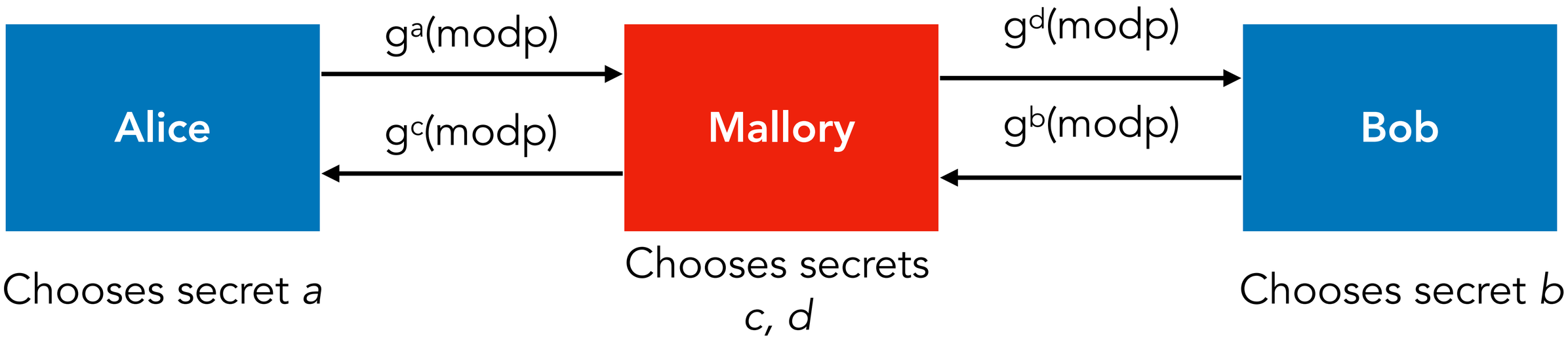
Diffie-Hellman Security

- Textbook Diffie-Hellman gives you protection against passive adversary, but…
  - **Weak to an active MiTM attacker.**



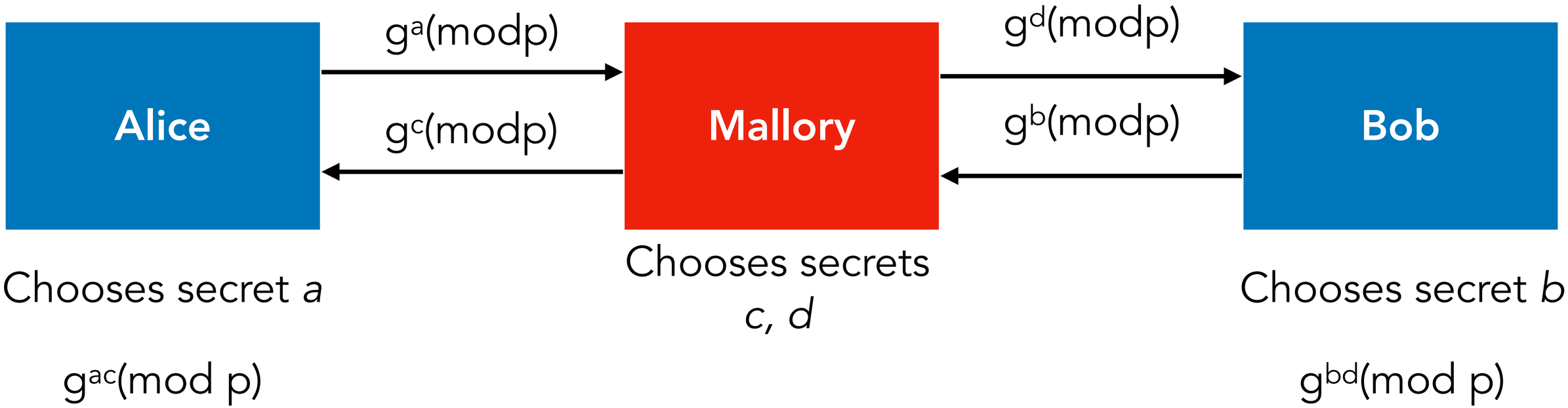
Diffie-Hellman Security

- Textbook Diffie-Hellman gives you protection against passive adversary, but…
  - **Weak to an active MiTM attacker.**



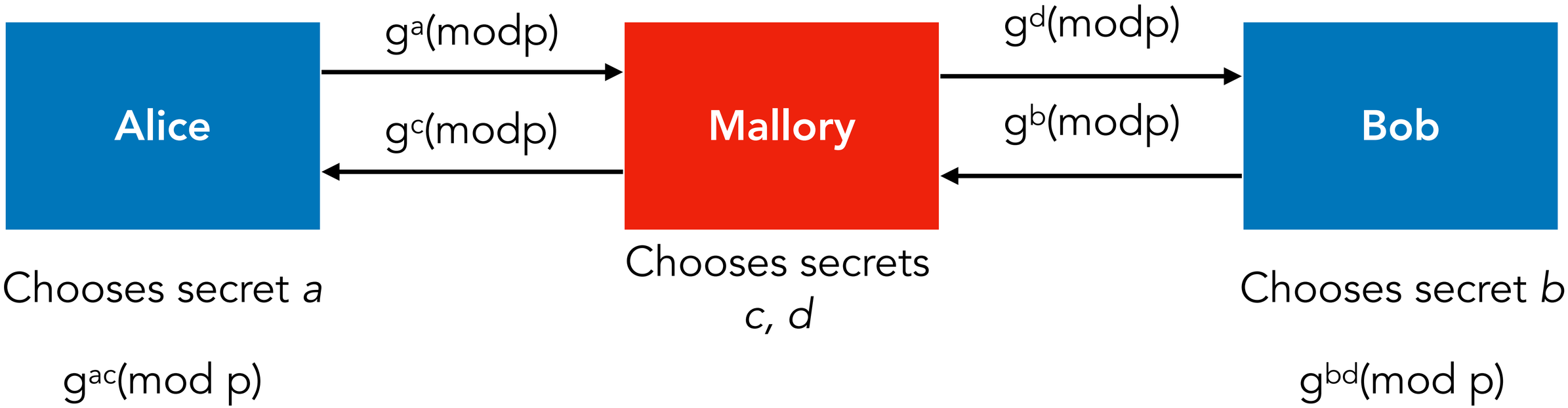
Diffie-Hellman Security

- Textbook Diffie-Hellman gives you protection against passive adversary, but…
  - **Weak to an active MiTM attacker.**



Diffie-Hellman Security

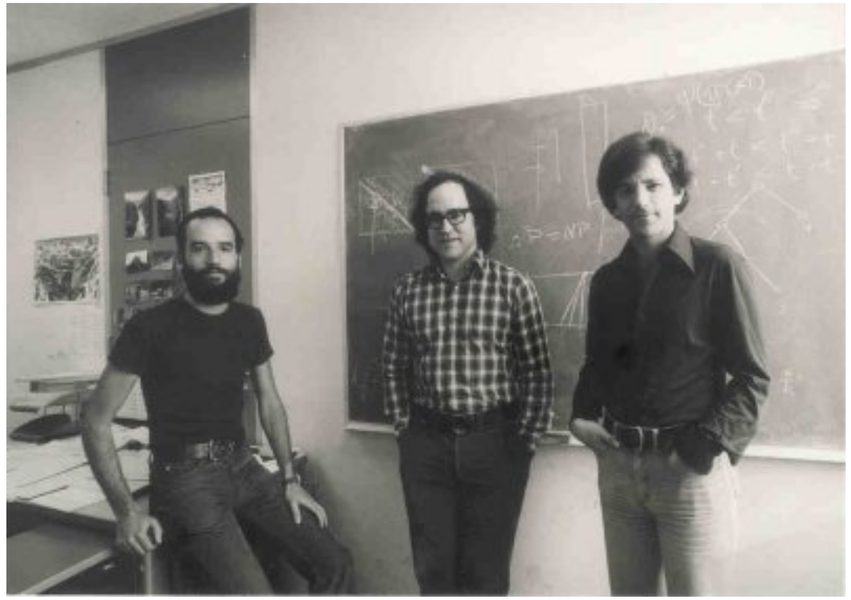
- Textbook Diffie-Hellman gives you protection against passive adversary, but…
  - **Weak to an active MiTM attacker.**



Diffie-Hellman does not provide *any* authentication!

Introducing RSA

- Maybe, if we could *authenticate* what we're sending w/ DH… we could be more sure that the secret *actually* came from Bob



A Method for Obtaining Digital Signatures and Public-Key Cryptosystems

R.L. Rivest, A. Shamir, and L. Adleman*

Factoring is hard

- Here, we're talking about *big integers*
- Integer multiplication is efficient to compute
- **There is no polynomial-time algorithm for general purpose factoring (especially of large numbers; or the product of large primes)**
  - Known as "integer factoring problem"
- Similar "one-way" function… easy to compute, hard to reverse to modular exponentiation

How RSA works

Alice

Bob

How RSA works

Alice

RSA key-generation scheme:

1. Generate two numbers, p and q. Large primes, usually 1024 – 2048 bits

Bob

How RSA works

Alice

RSA key-generation scheme:

1. Generate two numbers, *p* and *q*. Large primes, usually 1024 – 2048 bits
2. Compute N = p * q, this is known as the public modulus. This is known to all parties.

Bob

How RSA works

Alice

RSA key-generation scheme:

1. Generate two numbers, *p* and *q*. Large primes, usually 1024 – 2048 bits
2. Computer N = p * q, this is known as the public modulus. This is known to all parties.
3. Compute λ(N), LCM(p-1,q-1), called the totient function

Bob

How RSA works

Alice

RSA key-generation scheme:

1. Generate two numbers, *p* and *q*. Large primes, usually 1024 – 2048 bits
2. Computer N = p * q, this is known as the public modulus. This is known to all parties.
3. Compute λ(N), LCM(p-1,q-1), called the totient function
4. Choose *e*, such that e and λ(N) are co-prime; meaning gcd(e, λ(N)) = 1

Bob

How RSA works

Alice

RSA key-generation scheme:

1. Generate two numbers, *p* and *q*. Large primes, usually 1024 – 2048 bits
2. Computer N = p * q, this is known as the public modulus. This is known to all parties.
3. Compute λ(N), LCM(p-1,q-1), called the totient function
4. Choose *e*, such that e and λ(N) are co-prime; meaning gcd(e, λ(N)) = 1
5. Compute $d \equiv (e^{-1}) \pmod{\lambda(N)}$, d is the modular multiplicative inverse of *e* mod *λ*(*N*)

Bob

How RSA works

Alice

In the end, Alice has two keys:

Public Key: (e_alice, N)

Private Key: (d_alice, N)

Bob

How RSA works

Alice

In the end, Alice has two keys:
Public Key: (e_alice, N_alice)
Private Key: (d_alice, N_alice)

Bob also has two keys:
Public Key: (e_bob, N_bob)
Private Key: (d_bob, N_bob)

Bob

How RSA works

Alice wants to send an encrypted message to Bob. How would she do that?

Pub:(e_alice,N)
Priv: (d_alice,N)

Alice

$x = m^{e\_bob} \bmod N$

Pub:(e_bob,N)
Priv: (d_bob,N)

Bob

How RSA works

Alice wants to send an encrypted message to Bob. How would she do that?

Pub:(e_alice,N)
Priv: (d_alice,N)

Alice

$x = m^{e\_bob} \bmod N$

Pub:(e_bob,N)
Priv: (d_bob,N)

Bob

Decryption is reverse operation: $\mathbf{x^{d\_bob} \bmod N = m}$

How RSA works

Bob wants to sign a message m to Alice. How would he do that?

Pub:(e_alice,N)
Priv: (d_alice,N)

Alice

m, m^{d_bob} mod N

Pub:(e_bob,N)
Priv: (d_bob,N)

Bob

How RSA works

Bob wants to sign a message m to Alice. How would he do that?

Pub:(e_alice,N)
Priv: (d_alice,N)

Alice

m, s = m^{d_bob} mod N

Pub:(e_bob,N)
Priv: (d_bob,N)

Bob

Alice can verify that **m == s^{e_bob} mod N**; message must've come from Bob

Breaking RSA

- Best algorithm to break RSA: Factor *N* and compute *d*
- In general, factoring is not computationally efficient so this is quite hard to do
- Current key size recommendations: N >= 4096 bits; *p* and *q* are ~2048-bits each
- **Do <u>not</u> ever implement this yourself! Factoring is only hard for some integers… leave it to the experts.**

Textbook RSA is insecure

- Two problems with textbook RSA
- *Malleability* — attacker can modify input values and produce valid ciphertext
  - Because RSA is multiplicative:
    - $E(m_1) * E(m_2) = E(m_1 * m_2)$
    - Given a ciphertext *c* = Enc(m) = m^e mod N… attacker can forge ciphertext Enc(m*a) = $c*a^e$ mod N for any *a*
- *Chosen ciphtertext attack* — decrypting arbitrary ciphertext leaks information about the plaintext
  - Given a ciphertext *c* = Enc(m) for unknown *m*…
  - Attacker chooses *r;* computes $c * r^e = (mr)^e$ mod n
  - Attacker gets a decryption; which is m’ = m*r…. compute modular inverse and retrieve *m*

What do we do instead?

- Fundamental problem is RSA is *multiplicatively homomorphic*
  - Means the operation (multiplication) on encrypted data *corresponds to the same operation on hidden data*
- Worse, RSA is *deterministic* — the same inputs produce the same outputs (potentially leaking information in a chosen ciphertext attack)
- How do we prevent this from happening?
  - **Add random padding!**

A first attempt: Public-Key Cryptography Standards (PKCS#1) v1.5

- Basic idea; for a message *m* to be operated with RSA, pad message with the following scheme:

```
0x00 0x02 [random, non-zero bytes] 0x00 m
```

- Because the padding is random every time, exponentiation is *non-deterministic* and you *prevent the multiplicative homomorphism* that enables chosen-ciphertext attacks!

A first attempt: Public-Key Cryptography Standards (PKCS#1) v1.5

- Basic idea; for a message *m* to be operated with RSA, pad message with the following scheme:

```
0x00 0x02 [random, non-zero bytes] 0x00 m
```

- Because the padding is random every time, exponentiation is *non-deterministic* and you *prevent the multiplicative homomorphism* that enables chosen-ciphertext attacks!

- **Broken by Daniel Bleichenbacher in 1998…** showed that *implementation flaws* can break padding scheme without factoring RSA key!

Bleichenbacher's Attack

```
0x00 0x02 [random, non-zero bytes] 0x00 m
```

- If attacker can choose ciphertext and learn if padding is "correct or incorrect" from a server, **they can recover the entire plaintext.**
  - *Padding oracle attack* — if you have an oracle that tells you bit of info about the padding, you can learn the plaintext
- How does it work?
  - $c = m^e \bmod n$; goal is to learn *m*
  - Attacker chooses *s*, computes $c' \equiv c * s^e \pmod n$
  - Send c' to decryption oracle
    - If oracle says c' is PKCS conforming, then attacker knows that first two bytes of m*s are 00 and 02… tells you that m*s is bounded.

Bleichenbacher's Attack Continued

`0x00 0x02 [random, non-zero bytes] 0x00 m`

- Define B = $2^{8(k - 2)}$ where *k* is the length of the padded message
  - In essence, B is just the number of bits in the message *minus* the first two padding bytes 0x00 0x02
- If padding oracle tells you that m*s is valid PKCS padding, that means….
  - 2B <= m*s < 3B
- Why?

Bleichenbacher's Attack Continued

`0x00 0x02 [random, non-zero bytes] 0x00 m`

- Define B = $2^{8(k - 2)}$ where *k* is the length of the padded message
  - In essence, B is just the number of bits in the message *minus* the first two padding bytes 0x00 0x02
- If padding oracle tells you that m*s is valid PKCS padding, that means….
  - 2B <= m*s < 3B
- Why?
  - If you treat padding as an integer, the value of the integer is
    - 0x00 * $2^{8(k-1)}$ + 0x02 * $2^{8(k-2)}$ + …… —> 2*B + remainder
  - m >= 2*B, m < 3B [because 0x00 0x03 is the next highest value]

Bleichenbacher's Attack Continued

```
0x00 0x02 [random, non-zero bytes] 0x00 m
```

- Basic attack strategy….
  - Compute m*s; figure out if the padding is valid, get a bound
  - Keep choosing different s values and get tighter and tighter bounds on m*s; eventually, you'll have enough information to know *exactly* the value of *m*
- Bleichenbacher says this usually takes about 2^{20} trials; can do pretty fast with modern computing!

So what do people do today?

- Avoid padding oracles (PKCS #1.5 still used in practice)
- Today, use RSA-OAEP — optimal asymmetric encryption padding…. actually invented by UCSD faculty member **Mihir Bellare** and UC Davis faculty member Phillip Rogaway
  - Details are left for 107 :)

Signatures are also busted!

- Same problems with malleability weaken signatures too…
- Attack:
    1. Attacker wants Sign(x)
    2. Attack computes $z = x*y^e \bmod N$ for some random value y
    3. Attacker asks signer to $s = \text{Sign}(z) = z^d \bmod N$
    4. Attacker computes $\text{Sign}(x) = s*y^{-1} \bmod N$
- **Must use padding, even when signing!**
- **Most applications will sign *hash of m;* not *raw message m*; attacker cannot control the hash function**

Side note…

```
0x00 0x02 [random, non-zero bytes] 0x00 m
```

- Bleichenbacher also found an attack on many RSA implementations of signature checking…. called **low exponent signature forgery attack**
- Basic idea — many implementers were not checking the *length* of padding, just checking the *format*
  - Attacker could arbitrarily shorten the padding length; e.g.,

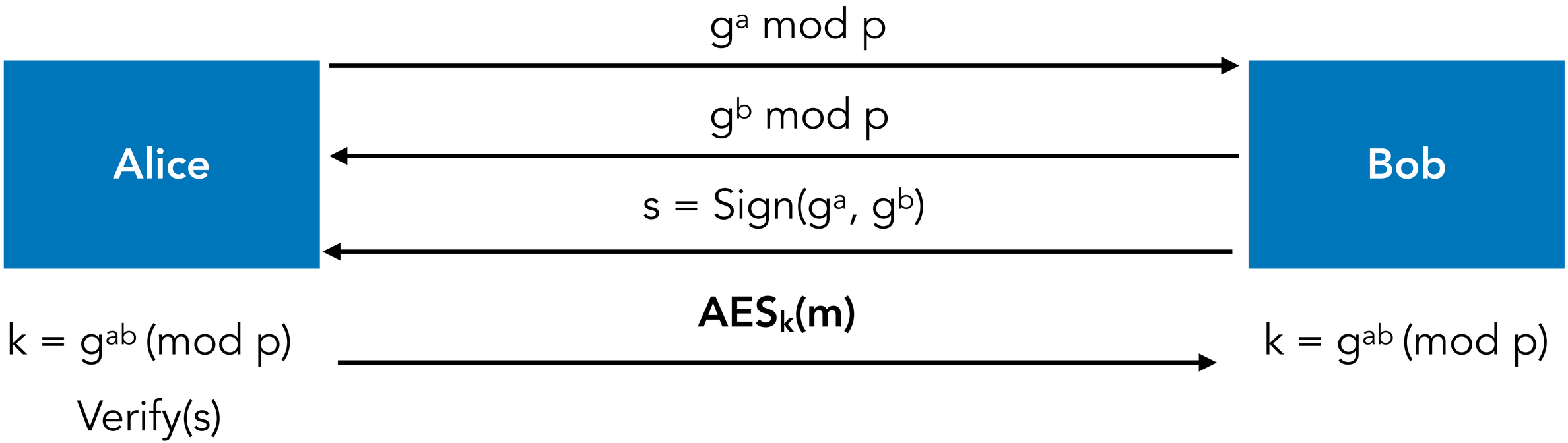
```
x = 0x0002[FFFF…..FF]00[hash of forged message][garbage]
```

  - If attacker can get *garbage* to match public exponentiation, they win
  - Many implementations used e = 3…. so attack was as simple as finding a cube root

Today’s best practice

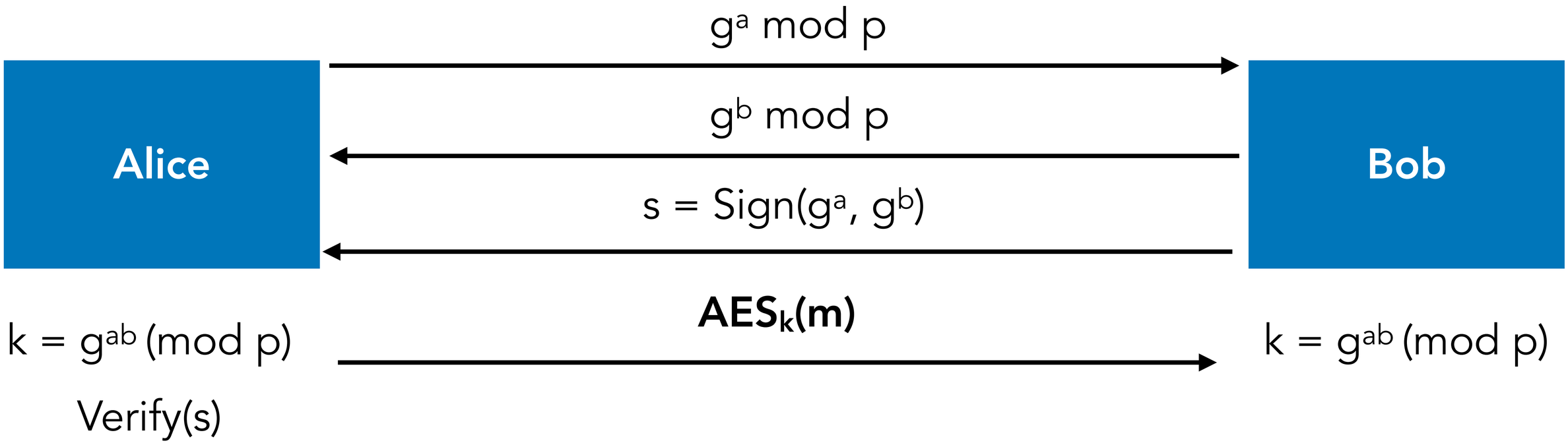
- Most people use elliptic curve signatures today, or even post-quantum algorithms like Falcon, CRYSTALS-Dilithium, SPHINCS+
- My research group uses ed25519 keys for all authentication

Putting it all together…



DH used to negotiate session key; Alice verifies Bob's signature; use shared key to symmetrically encrypt data.

Putting it all together…



DH used to negotiate session key; Alice verifies Bob's signature; use shared key to symmetrically encrypt data. **But how does Alice know Bob's public key….?**

Next time!

- Public-Key Infrastructure — how Alice knows Bob's public key
  - Certificate authorities, third-party trust, roots of trust, and how we actually do this for real on the web
- TLS + HTTPS —> how it works under the hood!