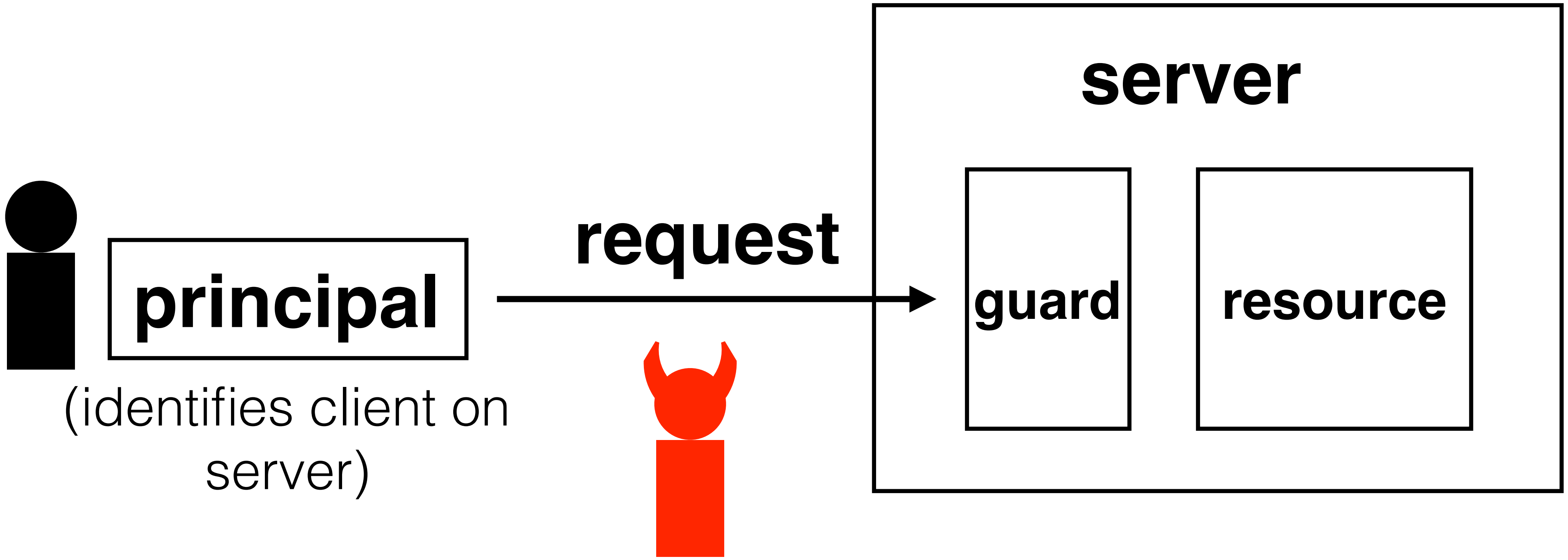
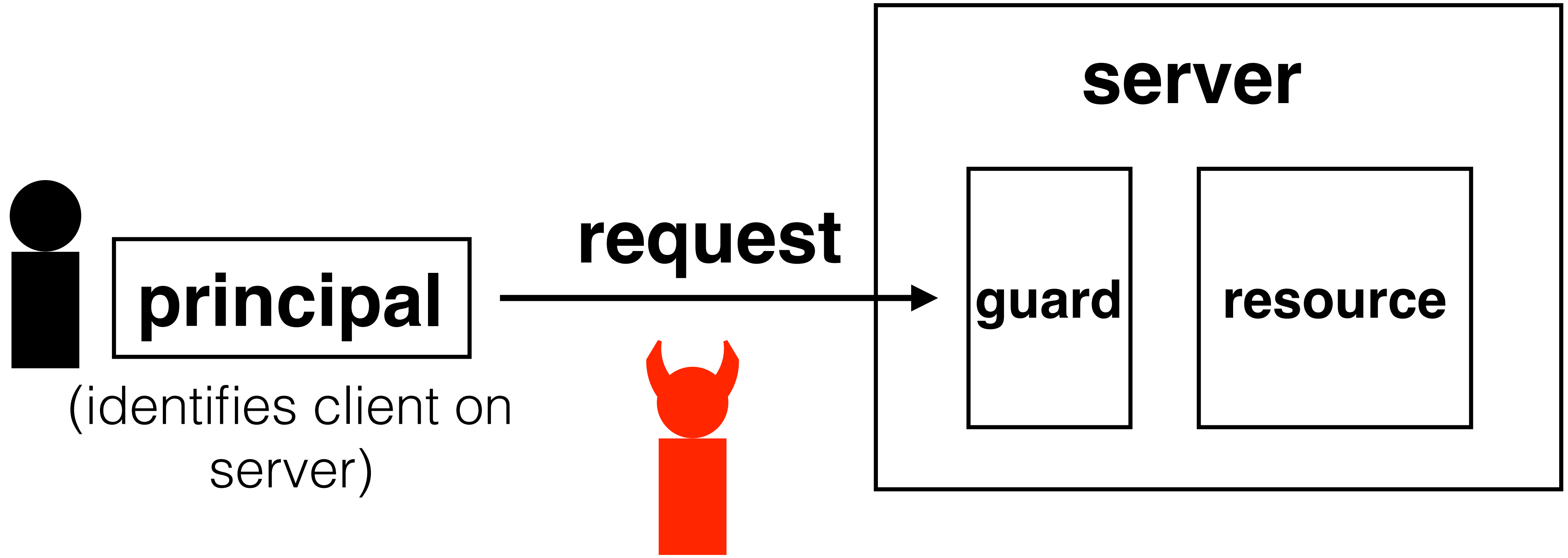
6.033 Spring 2019

Lecture #22

- **Combating network adversaries**
  - **Secure Channels**
  - **Signatures**


server
request
principal
guard
resource
(identifies client on
server)

today we're going to focus on how to protect packet data from an adversary. in a future lecture, we'll talk about how you can protect meta-information (e.g., packet headers) from an adversary



confidentiality: adversary cannot learn message contents

integrity: adversary cannot tamper with message contents
(if they do, client and/or server will detect it)

```
encrypt(key, message) → ciphertext
decrypt(key, ciphertext) → message
```

```
encrypt(34fbcbd1, “hello, world”) = 0x47348f63a67926cd393d4b93c58f78c
decrypt(34fbcbd1, “0x47348f63a67926cd393d4b93c58f78c”) = hello, world
```

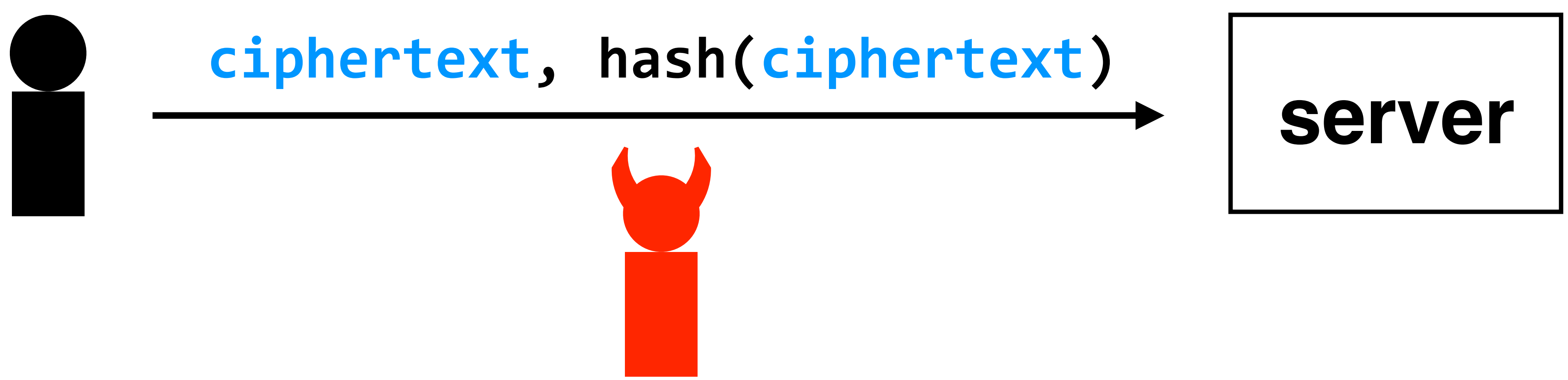
property: given the `ciphertext`, it is (virtually) impossible to obtain the `message` without knowing the `key`

adversary can't determine `message`, *but* might be able to cleverly alter `ciphertext` so that it decrypts to a different message

```
encrypt(key, message) → ciphertext
decrypt(key, ciphertext) → message
```

```
encrypt(34fbcbd1, “hello, world”) = 0x47348f63a67926cd393d4b93c58f78c
decrypt(34fbcbd1, “0x47348f63a67926cd393d4b93c58f78c”) = hello, world
```

property: given the `ciphertext`, it is (virtually) impossible to obtain the `message` without knowing the `key`



no good — if the adversary changes `ciphertext`, it can also (correctly) update the hash

```
encrypt(key, message) → ciphertext
decrypt(key, ciphertext) → message
```

```
encrypt(34fbcbd1, “hello, world”) = 0x47348f63a67926cd393d4b93c58f78c
decrypt(34fbcbd1, “0x47348f63a67926cd393d4b93c58f78c”) = hello, world
```

property: given the `ciphertext`, it is (virtually) impossible to obtain the `message` without knowing the `key`

```
MAC(key, message) → token
```

```
MAC(34fbcbd1, “hello, world”) = 0x59cccc95723737f777e62bc756c8da5c
```

property: given the `message`, it is (virtually) impossible to obtain the `token` without knowing the `key`

(it is also impossible to go in the reverse direction)

alice

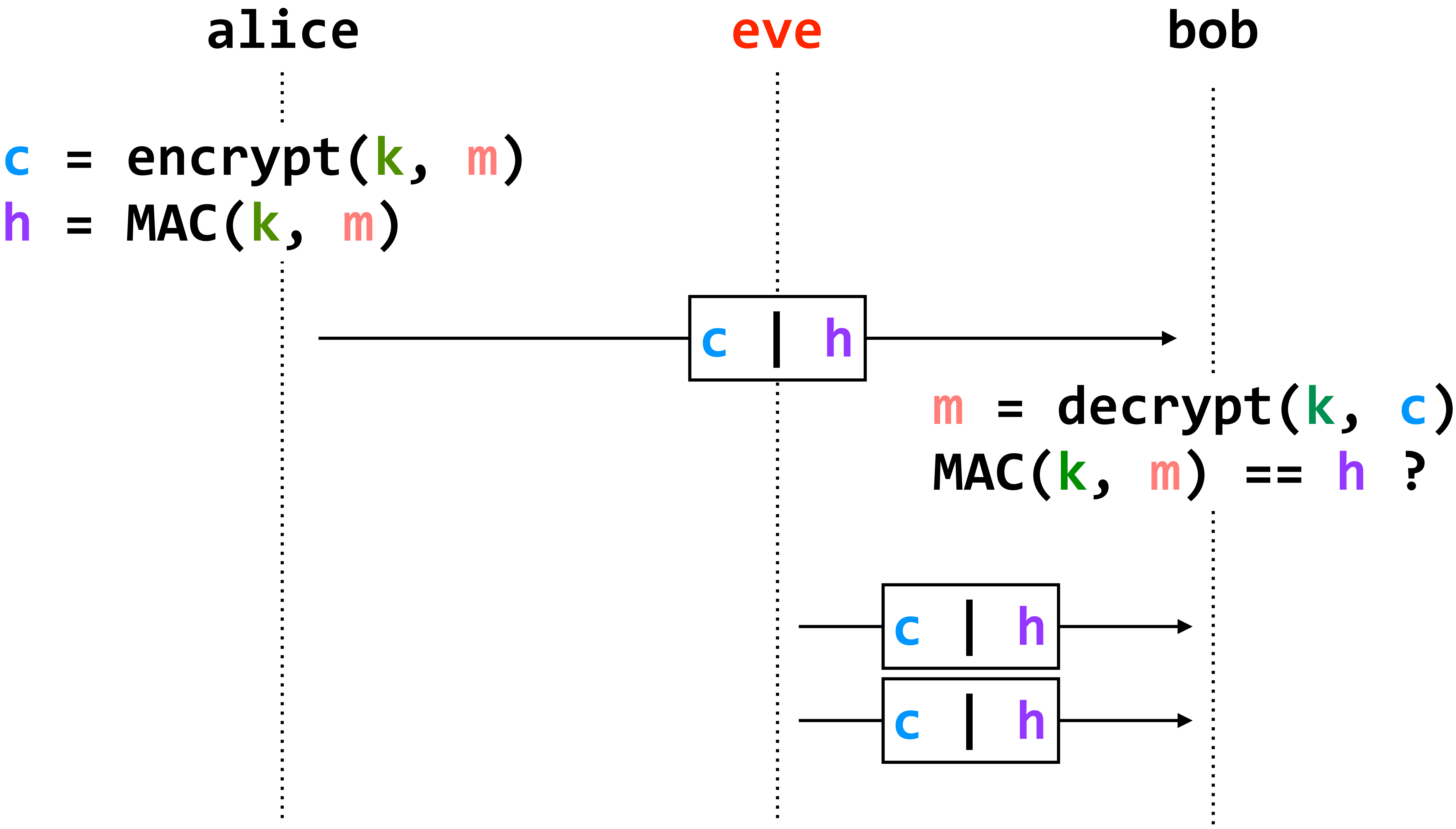
bob

```
c = encrypt(k, m)
h = MAC(k, m)
```

in practice, we'd use one key to encrypt and a different one to MAC

```
c | h
```

```
m = decrypt(k, c)
MAC(k, m) == h ?
```



problem: replay attacks

(adversary could intercept a message, re-send it at a later time)

alice

bob

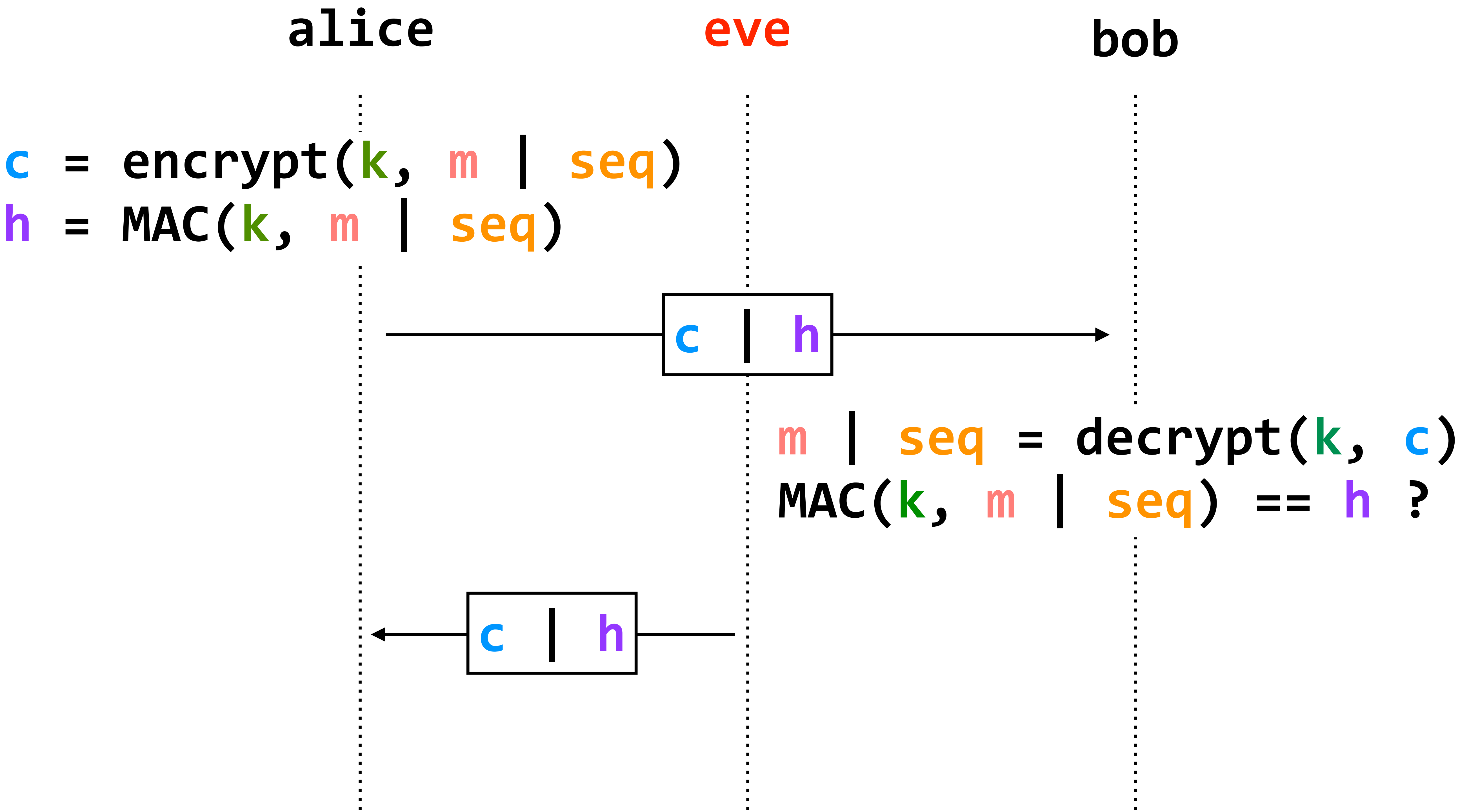
$c = \text{encrypt}(k, m \mid seq)$

$h = \text{MAC}(k, m \mid seq)$

$\boxed{c \mid h}$

$m \mid seq = \text{decrypt}(k, c)$

$\text{MAC}(k, m \mid seq) == h\ ?$



problem: reflection attacks

(adversary could intercept a message, re-send it at a later time in the opposite direction)

alice **bob**

$c_a = \mathrm{encrypt}(k_a,\ m_a \mid seq_a)$

$h_a = \mathrm{MAC}(k_a,\ m_a \mid seq_a)$

$\xrightarrow{\quad \boxed{c_a \mid h_a} \quad}$

$m_a \mid seq_a = \mathrm{decrypt}(k_a,\ c_a)$

$\mathrm{MAC}(k_a,\ m_a \mid seq_a) == h_a\ ?$

$c_b = \mathrm{encrypt}(k_b,\ m_b \mid seq_b)$

$h_b = \mathrm{MAC}(k_b,\ m_b \mid seq_b)$

$\xleftarrow{\quad \boxed{c_b \mid h_b} \quad}$

$m_b \mid seq_b = \mathrm{decrypt}(k_b,\ c_b)$

$\mathrm{MAC}(k_b,\ m_b \mid seq_b) == h_b\ ?$

problem: how do the parties know the keys?

known: p (prime), g

property: given $g^r \bmod p$, it is (virtually) impossible to determine **r** *even if* you know **g** and **p**

alice

pick random a

$g^a \bmod p$

$g^b \bmod p$

calculate

$(g^b \bmod p)^a \bmod p$

bob

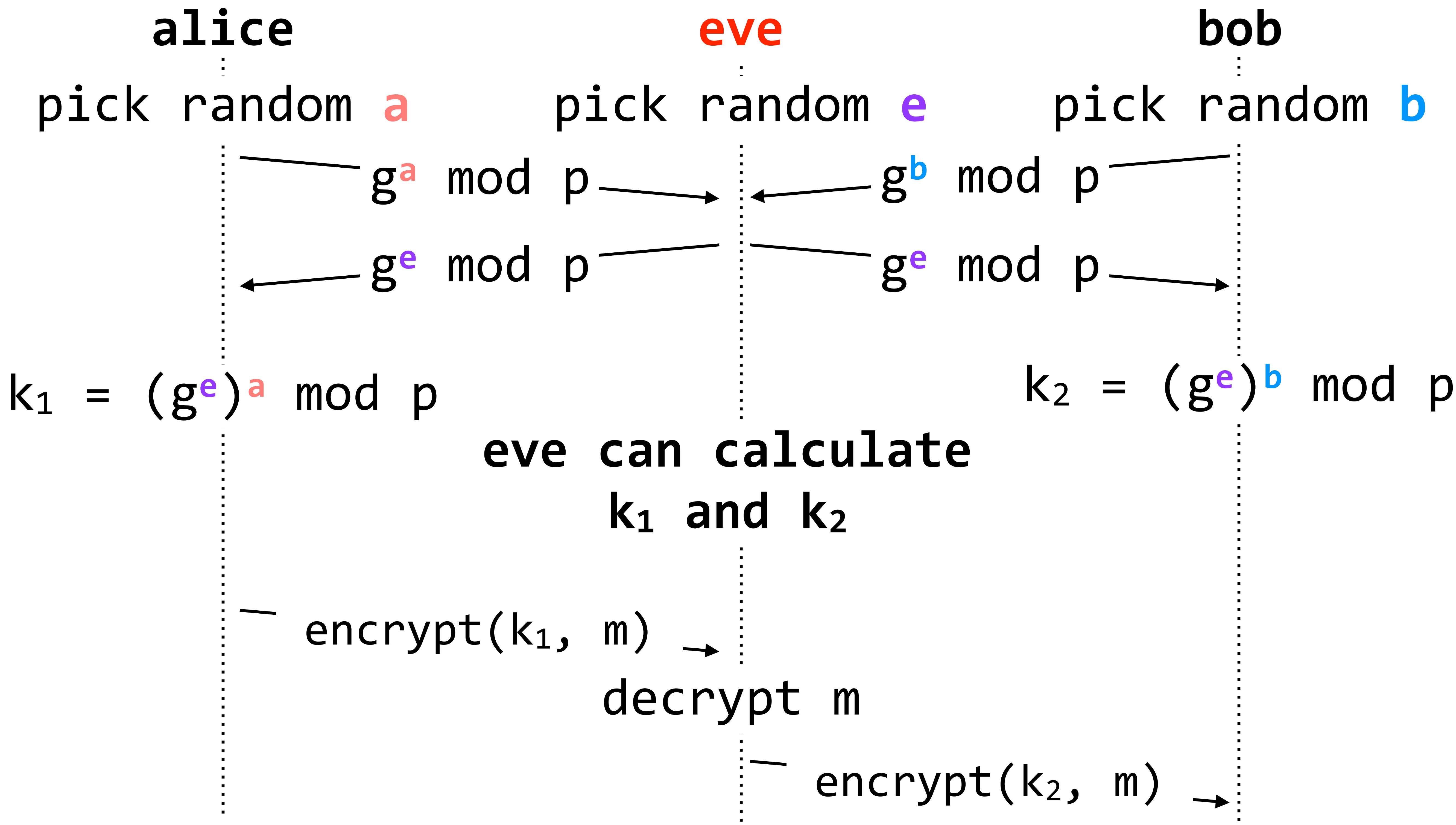
pick random b

calculate

$(g^a \bmod p)^b \bmod p$

key = $g^{ab} \bmod p$

$(g^x \bmod p)^y \bmod p = (g^y \bmod p)^x \bmod p = g^{xy} \bmod p$



problem: alice and bob don't know they're not communicating directly

cryptographic signatures

allow users to verify identities using public-key cryptography

users generate key pairs

the two keys in the pair are related mathematically

```
{public_key, secret_key}
```

```
sign(secret_key, message) → sig
verify(public_key, message, sig) → yes/no
```

property: it is (virtually) impossible to compute `sig` without `secret_key`

alice **bob**

`m = original message`
`c = encrypt(k_a, m | seq_a)`
`h = MAC(k_a, m | seq_a)`
`sig = sign(secret_key_a, m | seq_a)`

→ `c | h | sig` →

`m | seq_a = decrypt(k_a, c)`
`MAC(k_a, m | seq_a) == h ?`
`verify(m | seq_a, public_key_a, sig) == yes?`

verify will not check out if **sig** was not created with **secret_key$_a$**, the corresponding secret key to **public_key$_a$**

alice

bob

m = original message

c = encrypt(k_a, m | seq_a)

h = MAC(k_a, m | seq_a)

sig = sign($secret\_key_a$, m | seq_a)

c | h | sig

m | seq_a = decrypt(k_a, c)

MAC(k_a, m | seq_a) == h ?

verify(m | seq_a, $public\_key_a$, sig) == yes?

how do we distribute public keys?

TLS handshake

client **server**

ClientHello {version, seq_c, session_id, cipher suites, compression func}

ServerHello {version, seq_s, session_id, cipher suite, compression func}

{server certificate, CA certificates}

ServerHelloDone

client verifies authenticity of server

ClientKeyExchange {encrypt(server_pub_key, pre_master_secret)}

compute

master_secret = PRF(pre_master_secret, “master secret”, seq_c | seq_s)

key_block = PRF(master_secret, “key expansion”, seq_c | seq_s)

= {client_MAC_key,
server_MAC_key,
client_encrypt_key,
server_encrypt_key,
...}

Finished {sign(client_MAC_key, encrypt(client_encrypt_key, MAC(master_secret, previous_messages)))}

Finished {sign(server_MAC_key, encrypt(server_encrypt_key, MAC(master_secret, previous_messages)))}

- **Secure channels** protect us from adversaries that can observer and tamper with packets in the network.

- Encrypting with **symmetric keys** provides secrecy, and using **MACs** provides integrity. **Diffie-Hellman key exchange** lets us exchange the symmetric key securely.

- To verify identities, we use **public-key cryptography** and cryptographic **signatures**. We often distribute public keys with **certificate authorities**, though this method is not perfect.