Concepts of secure communication protocols

Luca Ferretti
Protocolli e Architetture di Rete
Università degli Studi di Modena e Reggio Emilia
Laurea Informatica A.A. 2024/2025

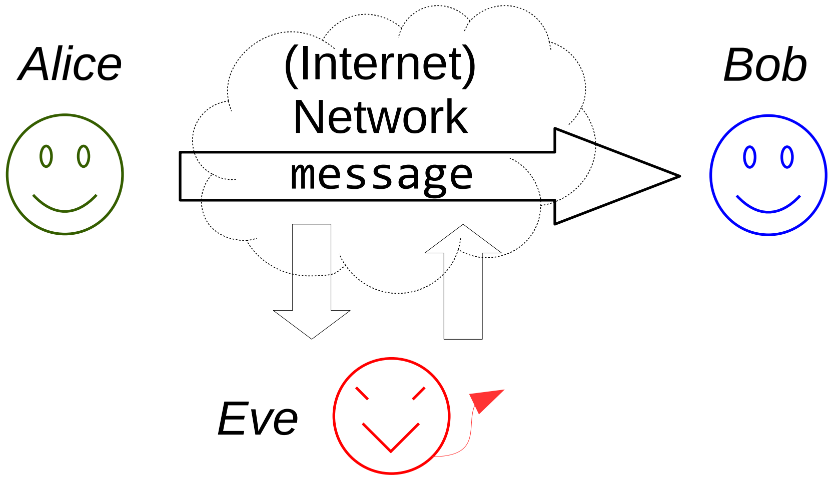
Secure communication protocols

- Standard Internet protocols have been designed by aiming at:
  - Performance
  - Reliability
  - **(almost) no security guarantees**
- In the modern Internet, security is **mandatory**
- **We consider that humans (with the help of computers) try to:**
  - Access our information in transit → violation of **confidentiality**
  - Fake sender identity **→** violation of **data origin authenticity**
- A secure communication protocol
  - Confidentiality: prevents adversaries to access data in motion

Secure communication

Secure communication protocols aim at ***protect*** information when the attacker can ***directly*** **access data (e.g., to the physical communication channel)**

- Communication scenarios
  - *"data in motion"*



- *This is the original <u>historical</u> motivation to develop cryptography*

Secure communications: Goals

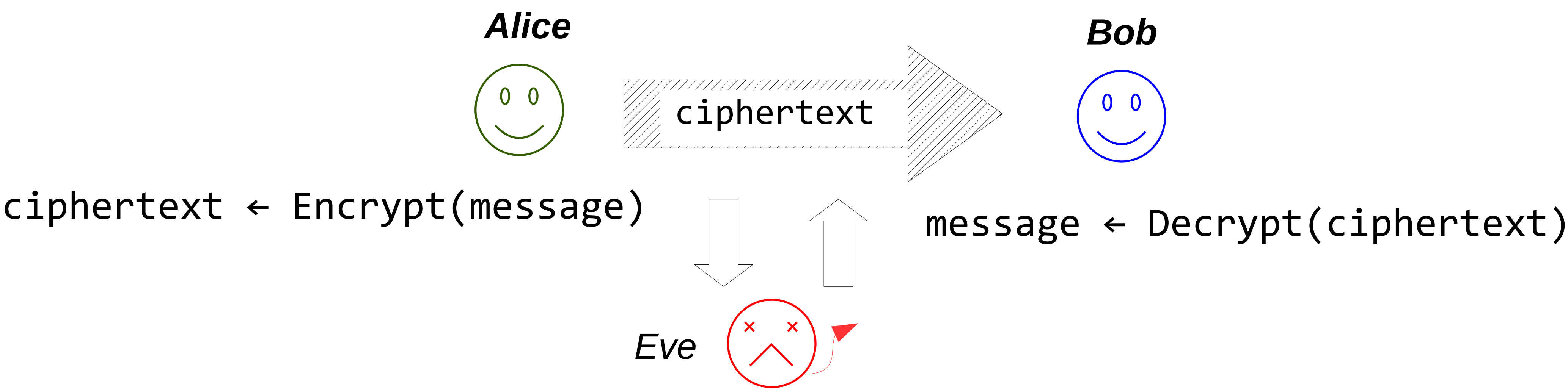
- Deter, prevent, detect, and correct security violations that involve the transmission of information
- Security guarantees
  - <u>Confidentiality</u>
  - <u>Integrity</u>
    - Authenticity
  - <u>Availability</u>
  - Accountability

CIA
(or AIC for disambiguation)

RID
- Riservatezza
- Integrità
- Disponibilità

Security guarantees

- *The simplest setting:* ***secure envelopes***



- ***Confidentiality:*** Eve cannot access any information about the *message*
- ***Integrity:*** Bob can detect if the message has been modified by Eve
  - ***Authenticity:*** Bob can verify if the message has not been sent by Alice

Cryptographic primitives

Overview

- **Symmetric** setting → entities know the **same secret key**
  - Symmetric encryption
  - Hash functions (here there is no key, but still considered symmetric due to internal designs)
  - Message Authentication Codes (MAC)
- **Asymmetric** setting → entities know different keys
  - Asymmetric encryption
  - Digital Signatures
  - Key exchange

Symmetric Encryption

Black-box symmetric encryption, probabilistic framework

`keygen([size]) → key`

p → plaintext
c → ciphertext

`encrypt(key, p) → c`

- *This function is **PROBABILISTIC**. Executing it multiple times with the same input message and key **always outputs different ciphertexts***

`decrypt(key, c) → p`

Note:

- Keys, Messages and Ciphertexts are **binary data**

Black-box symmetric encryption, deterministic framework

`keygen([size]) → key`

p → plaintext
c → ciphertext
n → nonce
iv → initialization vector

`encrypt(key, {n|iv}, p) → c`

- *Note: This function is **DETERMINISTIC***

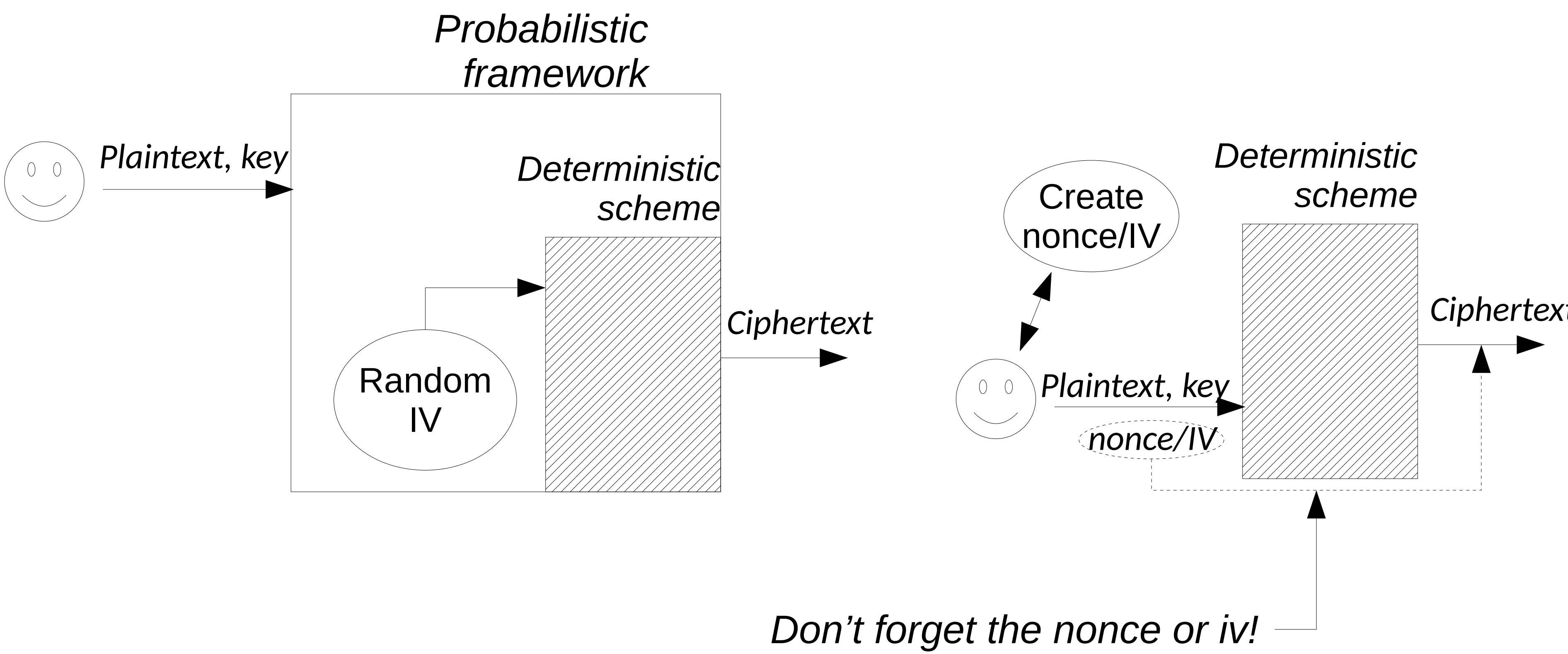
`decrypt(key, {n|iv}, c) → p`

Note: the only secret information is the key, the nonce/iv is not confidential and is typically sent as part of the ciphertext

Hands-on example: Python Cryptography

Probabilistic *vs.* Deterministic framework

- The *only* difference is the *explicit* presence of the ***nonce** or **iv*** in the encryption function inputs
  - different deterministic implementations may or may not put the ***nonce or iv within the ciphertext***
    - ***<u>check the documentation</u>***



Popular standards for non authenticated encryption

Current popular standards

- AES-CBC, AES-CTR
- Chacha20
- See extra material at the end of the slides for additional insights

Legacy/old standards/popular schemes

- 3DES-CBC
- rc4

For *authenticated encryption* see slides after Message Authentication Codes

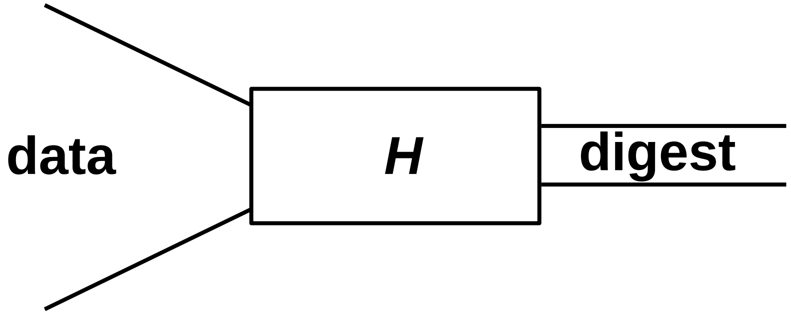
Integrity guarantees and Hash functions

Cryptographic Hash Functions [1]

A hash function *H* maps arbitrary-length strings to fixed-length (small) ones

$$H : \{0,1\}^* \rightarrow \{0,1\}^{\mathbf{n}}$$

hash*(data)* $\rightarrow$ *digest*



Hash functions exist for non-cryptographic settings too.

The size of the digest ***n*** is chosen such that it is *highly improbable* that two different inputs produce the same output → *the digest is a small-sized information that represents the data*

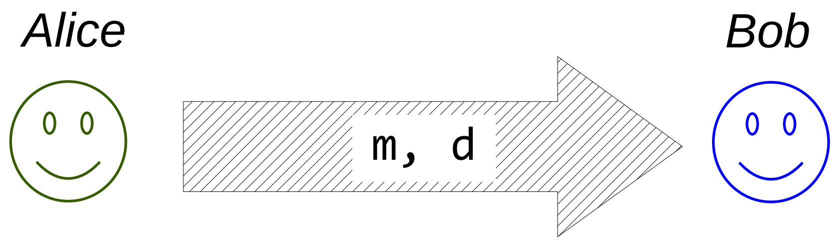
data1 != data2 ⟺ *digest1 != digest2*

A secure cryptographic hash function is **collision resistant**: it is unfeasible to find any m1 != m2 : H(m1) == H(m2)

Cryptographic integrity guarantees VS
<u>Integrity checks against transmission errors</u>

We studied that integrity checks within a message allows a recipient to detect corruption against transmission errors (e.g., CRC, checksum)

`1. Compute d=H(m)`

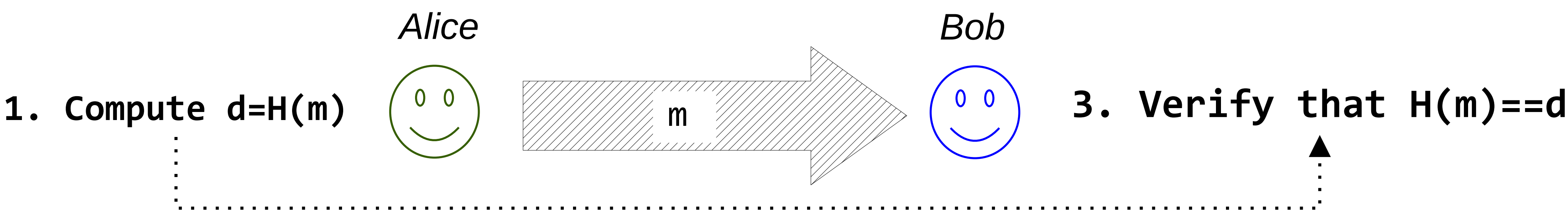


`Verify that H(m)==d`

Hash functions can also be used to detect corruption due to transmission errors, however an adversary could just re-compute the digest

Cryptographic integrity guarantees VS Integrity checks against transmission errors

Hash functions allow a recipient to verify message integrity **if he knows the output digest of the hash function**



Cryptographic integrity in communication scenarios

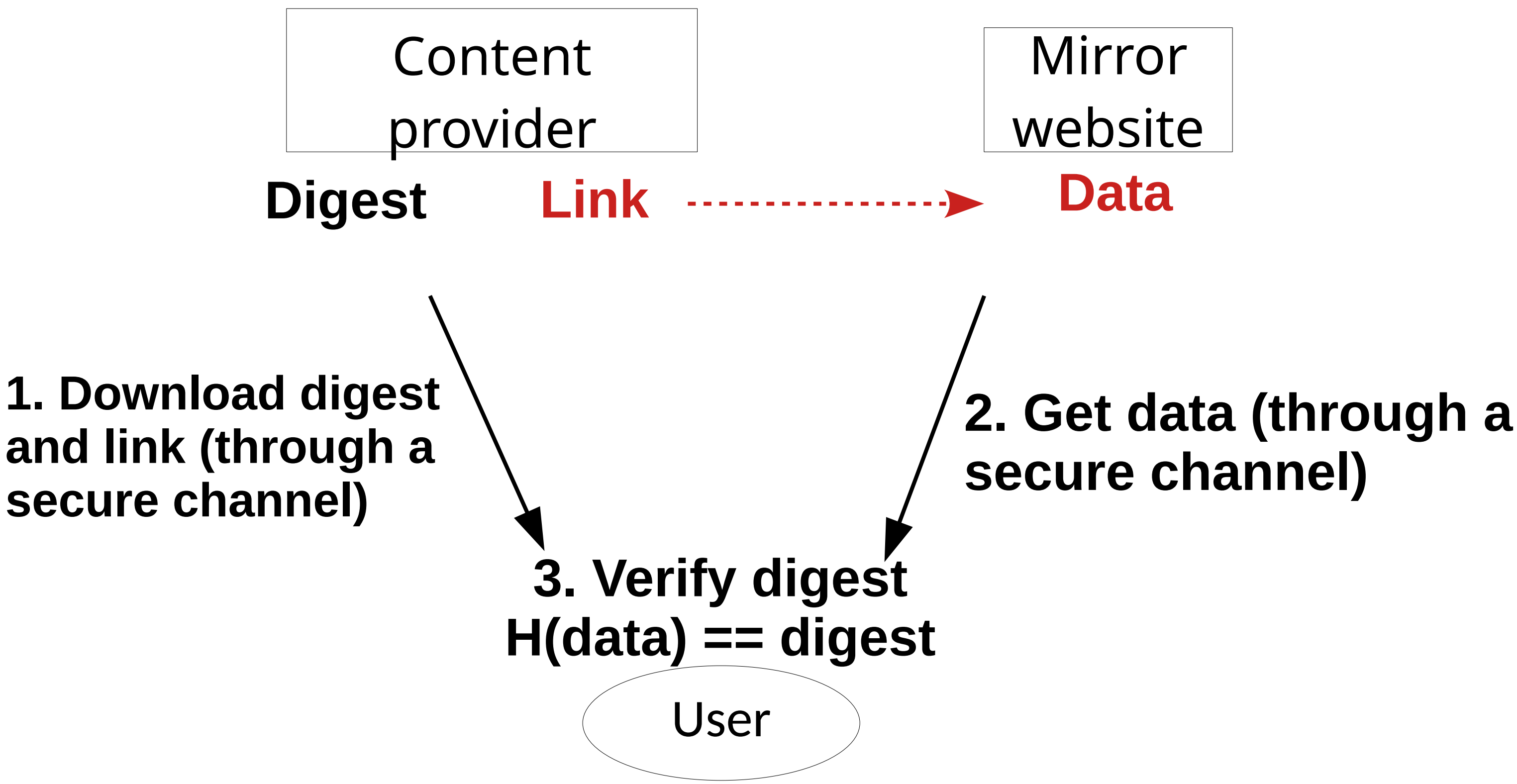
1) Alice computes a ***small-sized digest*** that "represents" the data

2) The digest is distributed to Bob via an <u>out-of-band communication</u> (*orthogonal to the considered protocol*)

3) Bob re-computes the digest by using the same hash function and compares it to that received via the out-of-band communication

Mirror website example

- The content provider computes a **digest = H(data)**, and outsources **data** to the mirror website
- The digest allows user to detect if the mirror modified the **data**
  - **Note that authenticity of the content provider is guaranteed by *out-of-band* secure communication channels**



Popular standard hash functions

Any hash function for which (any) collisions can be found is considered deprecated

- **md5** → unsecure, deprecated, very easy to find collisions
  - Sometimes still used for non-cryptographic integrity guarantees
- **sha1** → deprecated, collisions found
  - 160-bit digest (sha1 is sometimes also called sha160)
- **sha2** → OK, design strengthened design wrt sha1
  - sha224, sha256, sha384, sha512
  - digest size according to the specific implementation
- **sha3** → OK, built on different primitives than sha1 and sha2
  - sha3-224, sha3-256, sha3-384, sha3-512
  - officially standardized in 2015
- **blake/blake2/blake3** → OK, popular in open source contexts, very fast

Message Authentication Codes

Message Authentication Codes

- A message authentication code offers one routine:

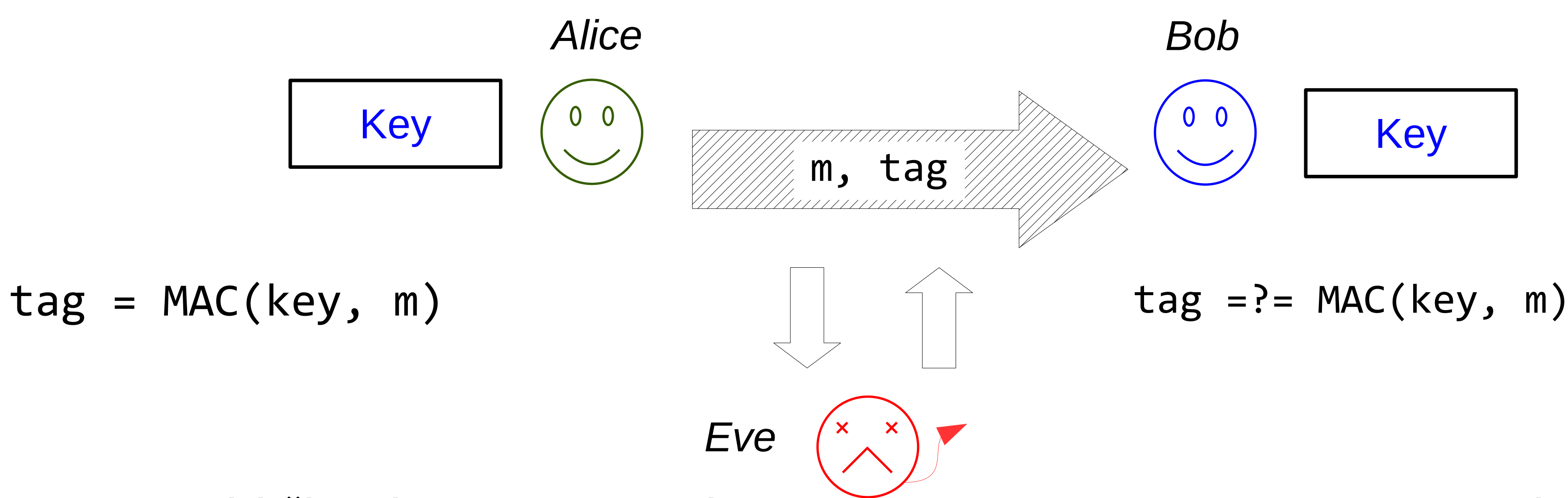
  ```
  MAC(key, message) → tag
  ```

- Verification is similar to hash functions: **the receiver re-computes the tag and compares it to the received one**
- A message authentication code allows Bob to verify whether the message *has ever been generated* by Alice
- *Beware to not mix security settings with Hash function guarantees*
  - *hash functions do not make use of any secret information*
  - *MACs use a secret key*

Integrity and <u>Authenticity</u> Guarantees

The recipient can detect if the **message has not been sent <u>by the legitimate sender</u>**

- *Who is the "legitimate sender"?*
  In the symmetric setting the **legitimate sender is someone who has access to secret key**



- *We could "bind" some metadata to our information to insert an <u>identity information</u> (e.g., "Alice"), but that is useful only in the asymmetric setting*

Popular MAC designs

Most popular, especially when used “standalone”:

- Based on collision resistant hash functions (e.g., **HMAC**)
- Based on block ciphers (e.g., **CMAC**, successor of **CBC-MAC**)

Typically used for authentication in popular authenticated encryption modes:

- Based on Universal hash functions (e.g., **GHASH/GMAC** used with AES-GCM and AES-GCM-SIV)
- Variant of Universal hash functions (e.g., **Poly1305**, used in Chacha20Poly1305, but also exists as Poly1305-AES)

Designed for small tags, specialized for certain usage scenarios:

- Native Keyed hash function (e.g., **SipHash**)

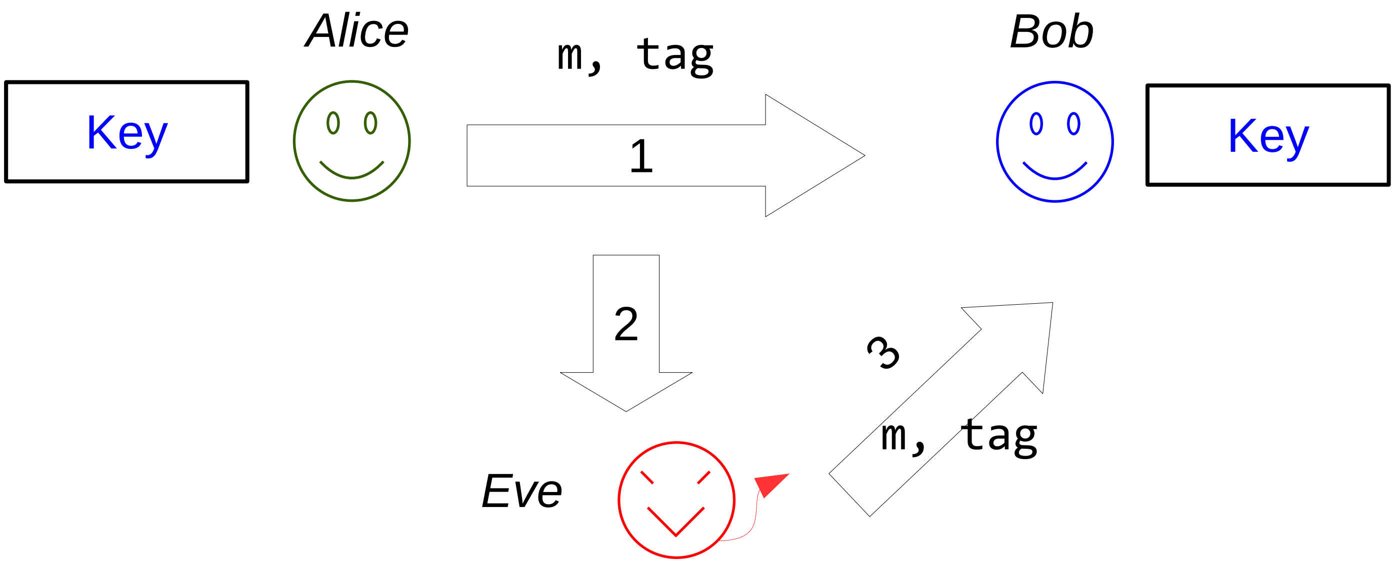
Communication integrity

Distinguish integrity and authenticity with regard to communcation models

- A MAC tag authenticates data included in an envelope (or packet)
- However, guaranteeing **authenticity of more complex communication paradigms** requires additional caveats
  - Byte stream communications composed of multiple packets → **Reply attacks**
  - Full duplex communications → **Reflection attacks**

Message Authentication Codes and replay attacks [1]

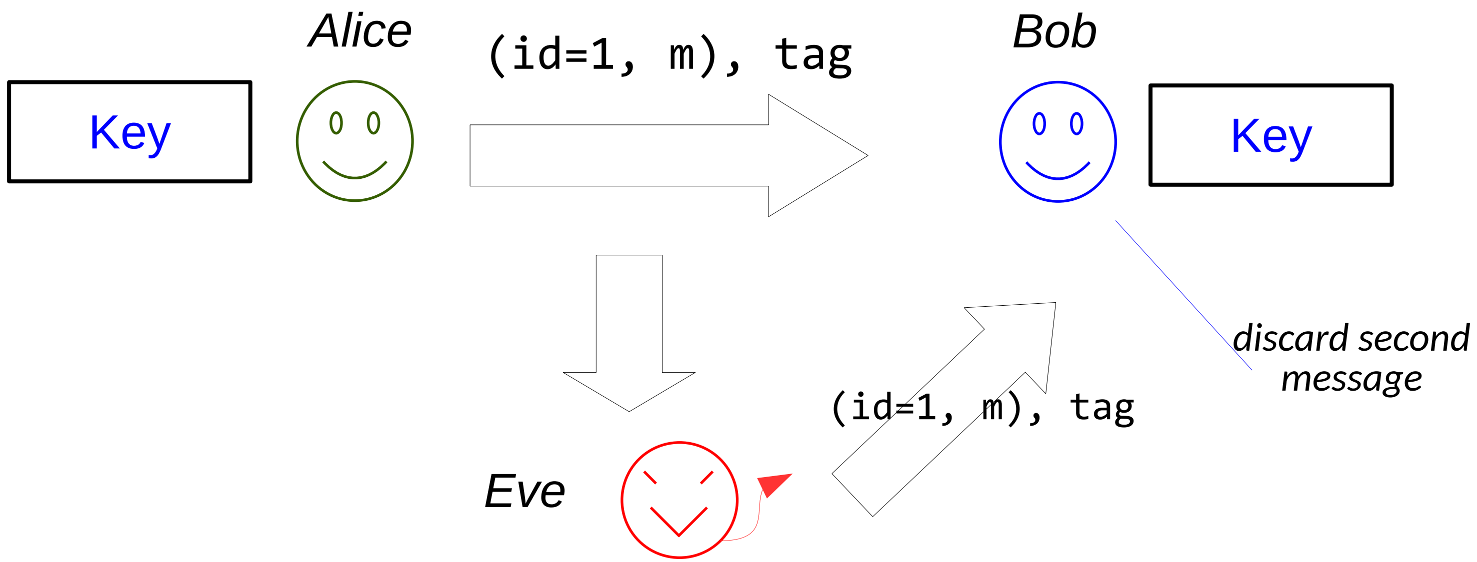
- A message authentication code guarantees that the ***symmetric key*** has been used to produce the ***tag***



- In a ***replay attack***, Eve sends messages that were actually sent by Alice to Bob
  - as an example, imagine that Alice and Bob are bankers, and Alice sent a message that says "Add 1000$ on Carl's account"

Message Authentication Codes and replay attacks [2]

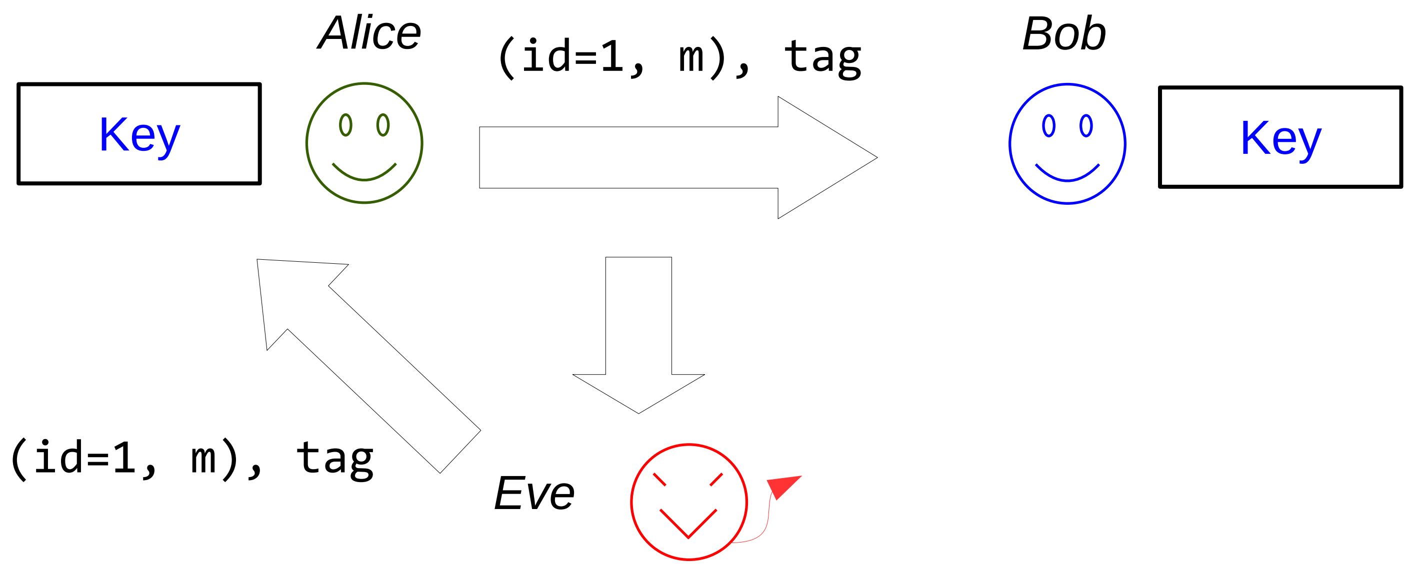
- A message authentication code guarantees that the ***symmetric key*** has been used to produce the ***tag***



- *Crypto alone cannot protect to such attacks. However, due design choices at the upper layers (e.g., **transport** or **application** layers) can prevent them*
  - Without going into details, the common defense against such an attack is to implement ***unique counters*** within the message

Message Authentication Codes and reflection attacks [1]

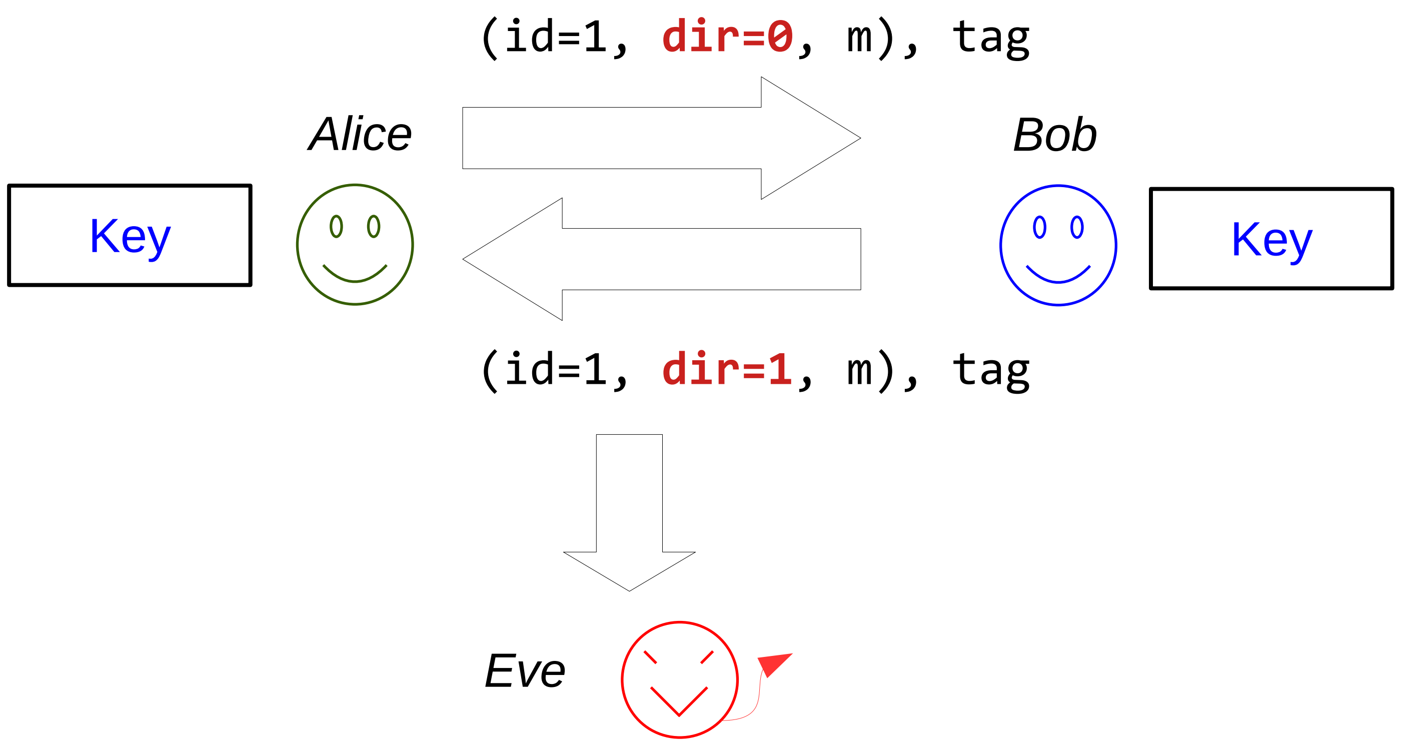
- If we consider **full-duplex communications**, the risk is of having an attacker the returns back the sender's messages



The sender verifies that the tag as valid!

Message Authentication Codes and reflection attacks [2]

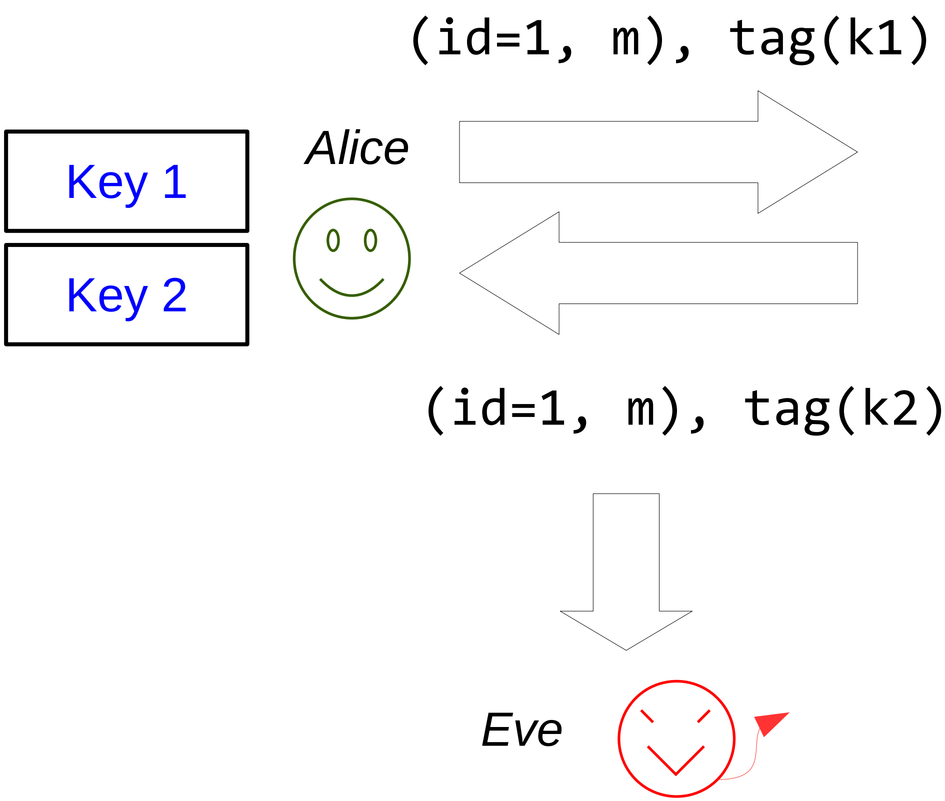
- First defense: add **direction bit** to communications



The direction bit act as an additional metadata that the sender must verify

Message Authentication Codes and reflection attacks [3]

- Second defense: handle the full-duplex communication as **two secure half-duplex → use two different keys**



This is the most popular approach: each party uses different keys **for sending and receiving messages**

Authenticated Encryption and AEAD framework

Encryption and Authenticated encryption

Schemes and Guarantees

- “Normal” (non-authenticated) Encryption → Confidentiality
- Hash → Integrity
- MAC → Authenticity and Integrity

We call **authenticated encryption** those schemes that guarantee confidentiality, integrity and authenticity

- (preferred) We should use a **standardized scheme** that already offers all these guarantees

Authenticated Encryption with Associated Data (AEAD)

The typical framework for general authenticated encryption is

```
keygen([size]) → key
encrypt(key, n, a, m) → c
decrypt(key, n, a, c) → m
```

- The **associated data** are not encrypted nor included in the ciphertext
- However, the decryption operation verifies that these data are the same used at encryption time, otherwise it ***fails*** *(i.e., it* ***detects*** *the violation of the data)*
  - *guarantees confidentiality, authenticity, integrity of plaintext*
  - *guarantees authenticity and integrity of associated data*
- *AEAD schemes are based on encryption schemes and MACs*
- *Hand-on example: Python cryptography*

Standardized authenticated encryption schemes → AEAD

- Benefits of these modes wrt composition methods:
  - *avoid implementation and development* ***errors***
  - *improved* ***efficiency*** (sometimes): *single keys, single-pass, parallel operations, …*
- Examples
  - AES-GCM, AES-GCM-IV (TLS, SSH, WPA2, many others)
    - CTR + GMAC
  - Chacha20Poly1305 (TLS1.3, SSH)
    - Stream + Poly1305
  - CCM (ZigBee)
    - CTR + CBCMAC
  - *Note: all these standards offer AEAD interfaces*

Asymmetric encryption

Issues of symmetric crypto

- Having symmetric keys is a big problem
  - **Distributing symmetric keys without prior knowledge between communicating entities is very hard (or expensive)**
- In asymmetric cryptography sender and recipient have different keys
- Asymmetric crypto introduces the term ***key pair***
  - a key pair is a couple of keys which are mathematically bound to each other

    (secret-key, public-key)
- ***Each key pair is usually associated with an entity***
  - ***the public key of a user is unique, and identifies him***

Asymmetric encryption [1]

`encrypt(pk, message) → ciphertext`

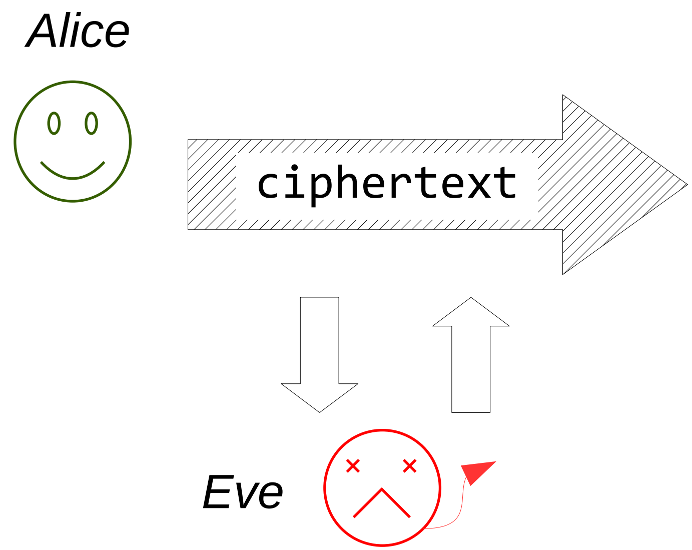
- Anybody that knows the publick key pk of the recipient can encrypt data for him/her
- **Distributing the public key does not introduce any security issues, because…**

`decrypt(sk, ciphertext) → message`

- **…only the entity that knows the corresponding secret key sk can decrypt the ciphertext**

Asymmetric Encryption [2]

```
Encrypt(
    bob_pk,
    plaintext
) → ciphertext
```




```
Decrypt(
    bob_sk,
    ciphertext
) → message
```

Alice knows that ***bob_pk*** *is* ***Bob's public key***

We can assume that Bob distributes it to everybody,
the key is public and does not compromise Bob

KEM – Key Encapsulation Mechanisms

- Intuitively, we call a scheme a ***key encapsulation mechanisms*** **(KEM)** if it only allows plaintext messages of fixed small size, that is, if it is **specialized to encrypt symmetric keys**
- We can combine a KEM with a symmetric scheme to obtain a so-called ***Hybrid schemes***
  - The asymmetric scheme is only applied on the symmetric key through the KEM
    - Note: asymmetric schemes are many orders of magnitude more expensive than symmetric schemes, thus using them for little data is good
  - The symmetric scheme encrypts the actual data

Authenticity in the asymmetric setting: Digital signatures

```
sign(sk, message)  → signature
```

```
verify(pk, message, signature) → {true, false}
```

A secure digital signature is ***unforgeable*** *without knowledge of the secret key*

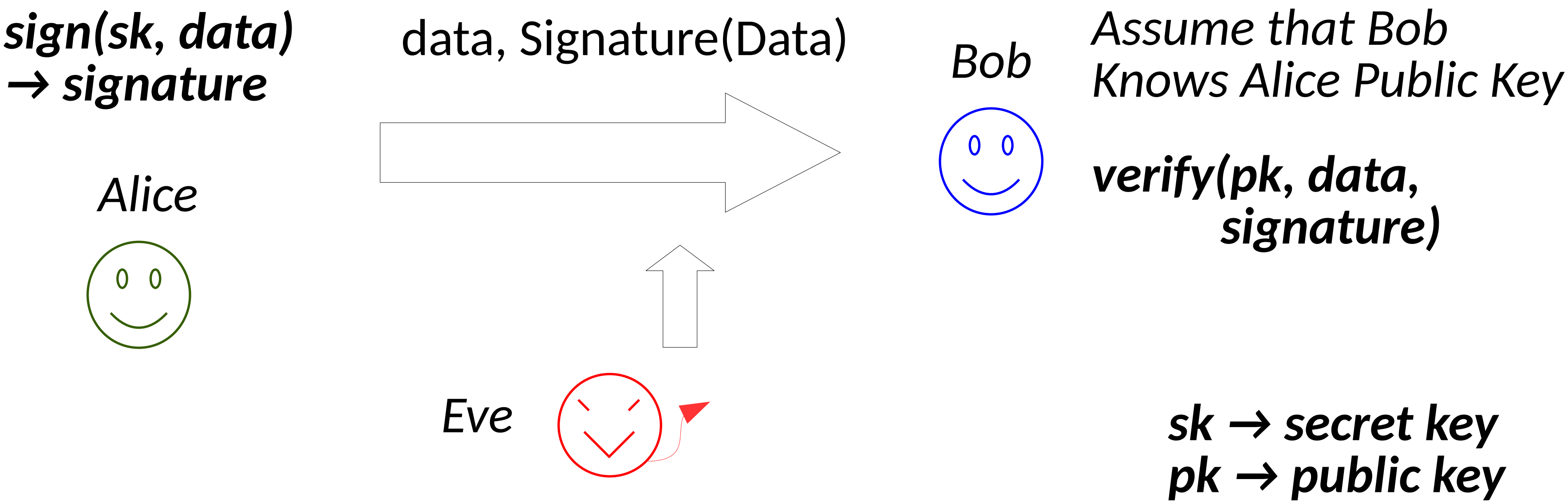
- *cannot create signature of a given message without sk*
- *~similar to MAC, but …*

Everybody can verify the signature → ***public verifiability***

- *assuming knowledge of the public key*

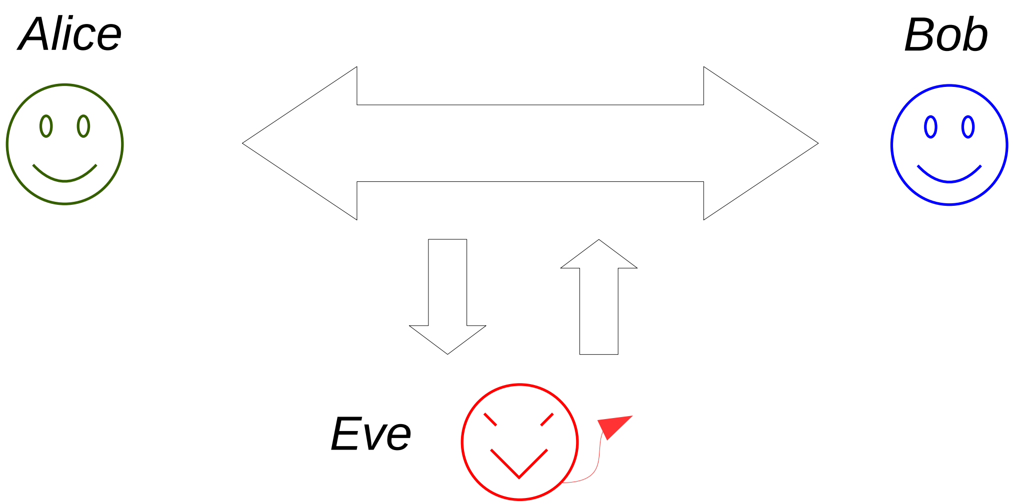
Only one participant knows the secret key → ***non repudiability***

Digital signatures



- A digital signature **authenticates** data sent by Alice
- But also enforces ***"non repudiability"***
  - ***Anybody can verify*** that Alice signed the message
  - ***Alice cannot deny to have signed that data***

Secure key exchange protocols



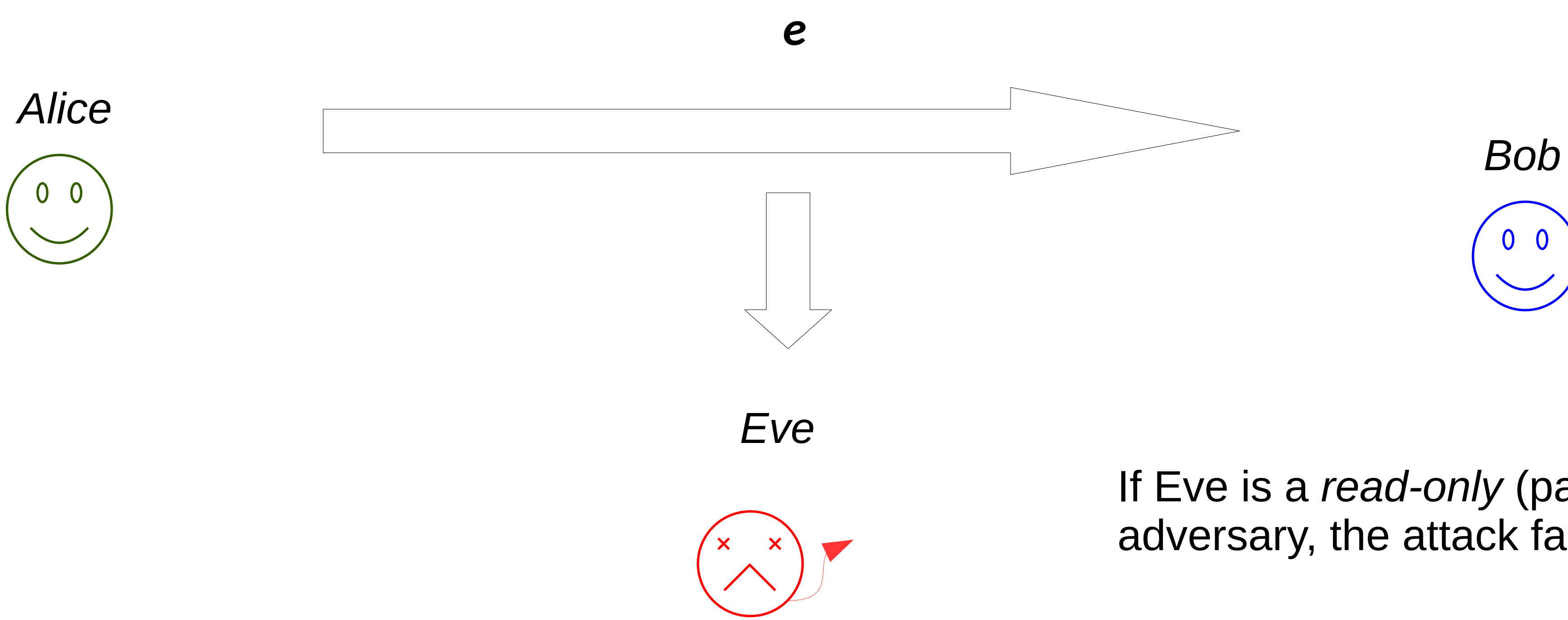
- Alice and Bob have ***no key***
- They want to **obtain a secure shared key *over an insecure synchronous channel***

Secure key exchange with KEM secure against passive adversaries

- Alice can send a symmetric key to Bob by using encrypting it with Bob public key by using a KEM

generate ***k***
compute **e = KEM.encrypt*(pkb, k)***

k = KEM.decrypt*(skb, e)*



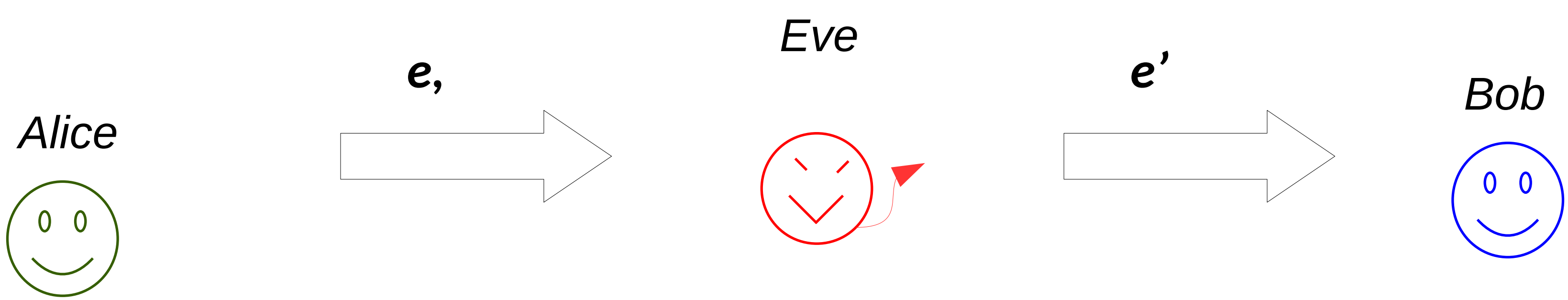
If Eve is a *read-only* (passive) adversary, the attack fails

Man-in-the-Middle

- A Man-in-the-Middle is an attack where the adversary can operate active attacks on the communication channels

generate ***k***
compute **e = KEM.encrypt*(pkb, k)***

k' = KEM.decrypt*(skb, e')*



compute **e' = *E(pkb, k')***

Asymmetric encryption **does not guarantee authenticity**
Bob has no way to verify authenticity of e or e'

Authenticated key exchange with KEM and Digital Signatures

- Alice can send a symmetric key to Bob by using encrypting it with Bob public key by using a KEM

generate ***k***
compute **e = KEM.encrypt(*pkb, k)***
*compute **s = Sign(ska, e)***

verify(pka, e', s)

Alice signs her public material with her secret key

Note:
Bob must know Alice Public Key!