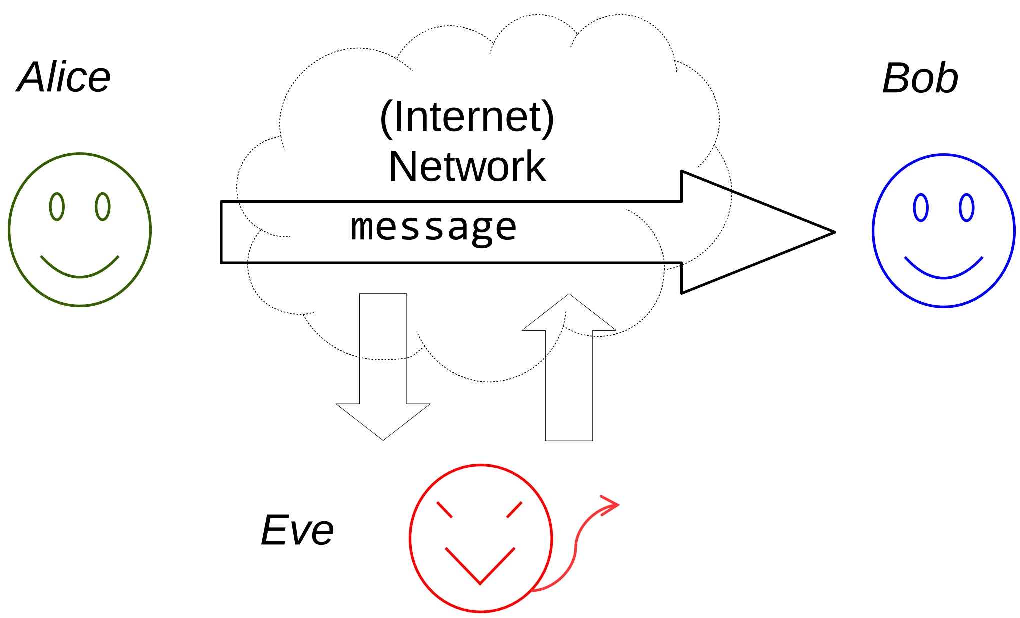
PARTE 9

INTRODUZIONE AI PROTOCOLLI SICURI

Cryptographic settings: Secure communications

- Cryptographic schemes can protect data on <u>unprotected</u> channels
  - whenever the attacker can directly access data
  - “data in motion”, the original historical motivation to develop cryptography

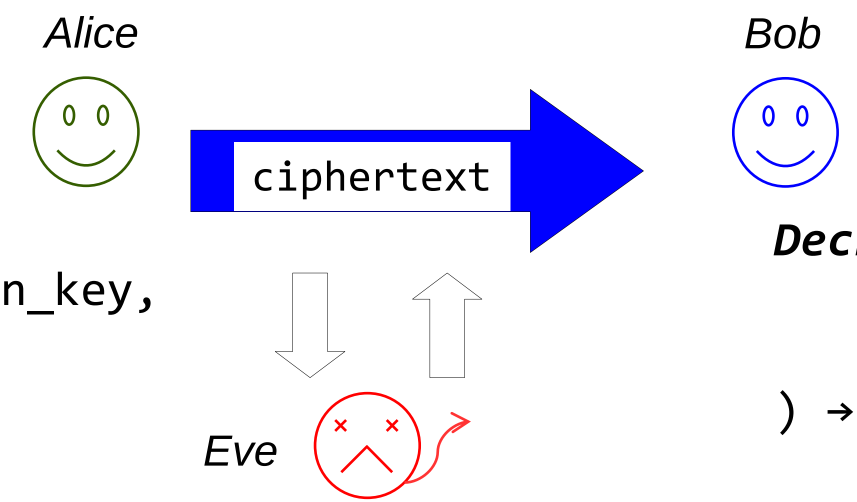


Cryptography

- The adversary has access to the encrypted information
- But the text is transformed in such a way that the adversary cannot make sense of it

Security guarantees

- Cryptographic protocols allow to protect data as a secure enclosure




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

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

- High-level security guarantees given by standard cryptographic protocols:
  - 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 and cryptographic protocols

- Cryptographic primitives are mathematical tools
  - block/stream ciphers, hash functions, …

- Cryptographic protocols are security-oriented communication protocols that leverage cryptographic primitives, but also
  - include details at the application level
  - include details about key management
  - can be deployed in specific scenarios/settings

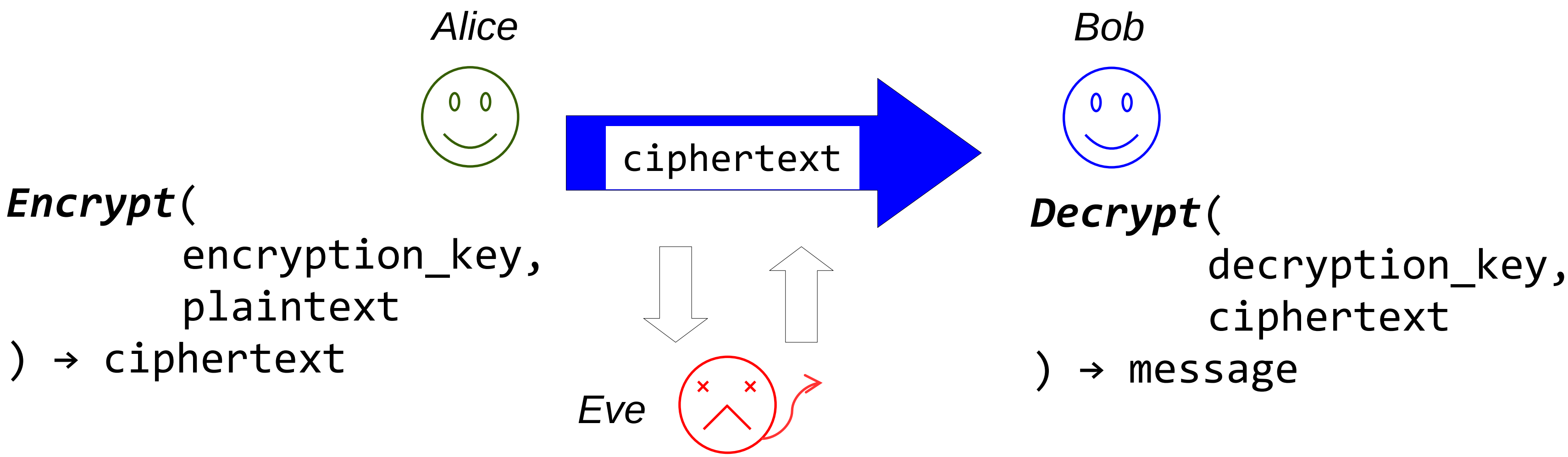
Modern encryption schemes

- Kerckhoffs principle:
  - algorithms are public
  - security relies on the secrecy of the key
  - “A cipher must be **practically**, if not mathematically, indecipherable.”

- Modern encryption scheme:
  - the key space is large enough to prevent brute force search
  - the scheme is designed to prevent cryptanalysis of the ciphertext
    - no information can be obtained from the ciphertext regardless the type of plaintext data
    - it is said that the ciphertext is indistinguishable from random

Symmetric encryption

Symmetric setting



- In symmetric settings Alice and Bob share the same key

encryption_key = decryption_key

Modern Encryption schemes types

- Computational security
  - Stream ciphers
  - Block ciphers + operation mode

- Perfect secrecy (Unconditional security)
  - One-Time-Pad

XOR

- Modern symmetric ciphers are designed for binary data
- The base operation for symmetric crypto is the XOR

$$out = x \oplus k$$

x	k	out
0	0	0
0	1	1
1	0	1
1	1	0

- Why XOR?
  - it is impossible to infer information about the plaintext or the key from the output

One time pad (Vernam's cipher) [1]

- One-time-pad (OTP)
  - XOR operation between the plaintext and the key
  - size of the key = size of the plaintext
  - the whole key is random

Random key	0	1	1	0	1	0	0	1	
	$\oplus$	$\oplus$	$\oplus$	$\oplus$	$\oplus$	$\oplus$	$\oplus$	$\oplus$	
Message	1	1	0	0	0	0	1	1	
	=	=	=	=	=	=	=	=	
Ciphertext	1	0	1	0	1	0	1	0	

One time pad (Vernam's cipher) [2]

- Perfect secrecy:
  - no algorithms can ever break the encryption scheme
    - no cryptanalysis, no brute-force attacks, no issues regarding future novel computation paradigms or more efficient algorithms

- The size of the key must equal or greater than that of the message
  - the key must be random
    - NB: the key must be shared through secure channels with the other participants. HOW?
  - any reuse of the key allows the attacker to recover the plaintext message

→ OTP cannot be used in most scenarios

Computational security [1]

- An encryption scheme must be <u>practical</u>
  - keys should be “short” (big enough to resist brute-force search)
  - given the key, encryption and decryption functions should be efficient
- Computational security schemes can be broken...
  - e.g., “in a thousand years by using all computers in the world”
  - The security of practical schemes relies on the computational difficulty of breaking them
  - It is unfeasible to break the scheme given <u>bounded</u> amounts of time or resources

Computational security [2]

- Efficient → Polynomial
  - encryption time is polynomial with respect to the size of the key
- Unfeasible → Exponential
  - brute-forcing a ciphertext takes exponential time with respect to the size of the key

Computational security

- Security-level → Parameters/key size

- In modern symmetric ciphers, the key size defines the security level
  - 128-bit key → ~ 128-bit security

- In asymmetric schemes it is more difficult as it depends on the underlying math
  - 1024-bit RSA key → ~80-bit security
  - 2048-bit RSA key → ~112-bit security

Best practices for security parameters

- Software and libraries should implement secure configurations by default
  - and should be updated when necessary

- NIST releases official best practices for security levels, key sizes, parameters size, etc. (p.66)
  - http://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-57pt1r4.pdf

Stream ciphers

- Encryption with stream ciphers is very similar to OTP
  - a bit-wise XOR operation computed between the plaintext and an intermediate key (keystream)

- The encryption routine takes a short random key as input
  - the PRG is a deterministic function that takes a small seed as input and that outputs pseudo-random data that cannot be distinguished from random

- The encryption key and the nonce are used to generate the PRG seed

$$\text{encrypt(key, nonce, m): m} \oplus \text{PRG(key, nonce)}$$

Stream ciphers security

- Why a nonce
  - given the same seed, the PRG produces the same intermediate key
  - re-using the same nonce (and key) completely breaks the cipher
  - famous misuse in MS.Office → http://eprint.iacr.org/2005/007.pdf

- Side note: as OTP, stream ciphers are very weak against manipulation attacks
  - flipping one bit on the ciphertext also flips the bit at the same position in the plaintext
  - No integrity guarantee!

Encryption based on block ciphers

- Block ciphers allow to encrypt by data block-by-block
  - the block size is one of the main characteristics of a Block Cipher
- Block cipher != encryption protocol
  - Block ciphers are a primitive to build a symmetric encryption scheme.

    Example:
    - AES → very famous block cipher with a block size of 128 bits and allowed key sizes of 128,192,256 bits
    - AES-128 → specific implementation of AES with a key size of 128 bits
    - AES-128-CBC → encryption scheme based on the AES-128 block cipher used in combination with the CBC encryption mode

Block cipher operation modes

- A block cipher is not an encryption scheme
  - it only works on data that are as big as its block size
  - it is deterministic (no nonce here)
- A block cipher can be used to build a symmetric encryption scheme by using an encryption mode
- There exist many operation modes: CBC, CTR, GCM, …
- Certain operation modes allow to build a stream cipher out of a block cipher
  - stream cipher modes
- Certain operation modes allow to authenticate the data
  - authenticated modes
    - Note: authenticated modes always expand the ciphertext

Example: AES encryption/decryption from command line

- To encrypt:
  - openssl enc -aes-256-cbc -pbkdf2 -in *plaintext-file* -out *cyphertext-file*
- To decrypt:
  - openssl enc -d -aes-256-cbc -pbkdf2 -in *cyphertext-file* -out *plaintext-file*

Hash functions and Message Authentication Codes

Integrity protocols (not only crypto)

- Protocols to detect modifications on data

- One routine:
  - compute-digest(data) → digest

- Size(digest) constant with respect to the security level (usually half)

data1 != data2 ⟺ digest1 != digest2

Integrity guarantees without security benefits

- Detecting data modifications due to “accidents”
  - Detect transmission errors
  - Detect storage faults

- Popular algorithms: Parity, CRC, Checksums

- More related to message correctness in network protocols against transmission and propagation errors

Integrity guarantees from a security perspective

- Detect adversarial modifications based on the knowledge of the integrity algorithm
- Consider $H(m1) \rightarrow d$
- collision resistance
  - infeasible to find any m1, m2 such that $H(m1) = H(m2)$
- preimage collision resistance
  - given m1, infeasible find m2 such that $H(m1) = H(m2)$
- Cryptographic hash functions guarantee collision resistance and preimage collision resistance

Hash functions implementations

- md5 → really deprecated and insecure
- sha1 → deprecated, collisions found
  - 160-bit digest (sha1 is sometimes also called sha160)
  - ongoing interesting discussions about sha1 in Git
- sha2 → OK, stronger version of 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
  - slower than sha2
  - officially standardized in 2015
  - not many implementations around

Message Authentication Codes

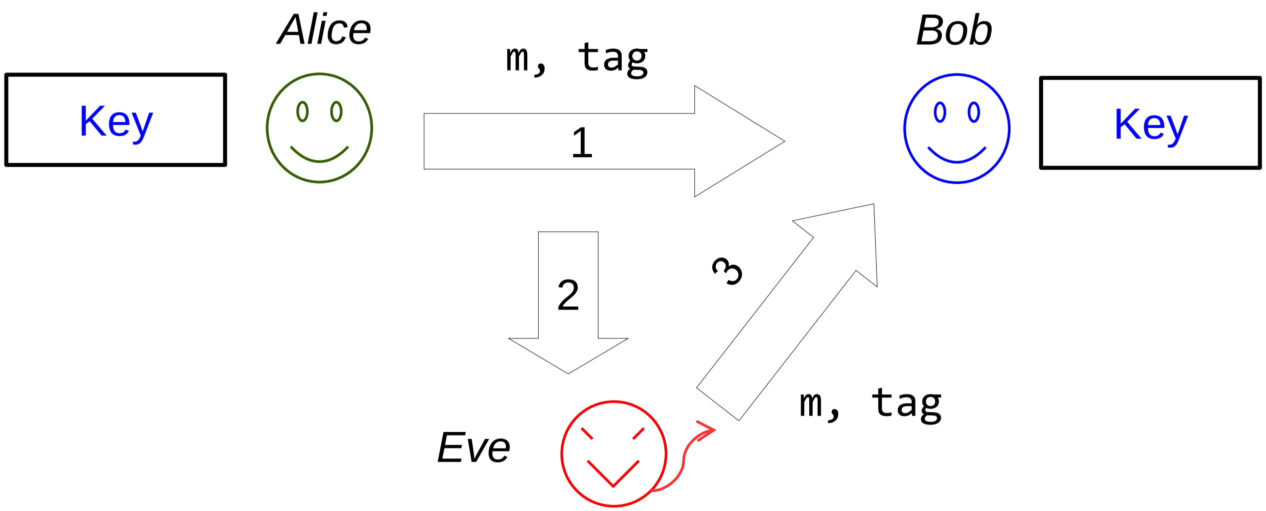
- A message authentication code includes one routine:

  MAC(key, message) → tag

- A message authentication code allows Bob to verify whether the message has ever been generated by Alice
- Beware: Hash != MAC
  - hash functions do not make use of any secret information
  - MACs use a secret key
- Common standard implementation of MAC use:
  - hash functions → HMAC
  - block ciphers → CBC-MAC (old), CMAC (new), Poly1305

Message Authentication Codes and replay attacks [1]

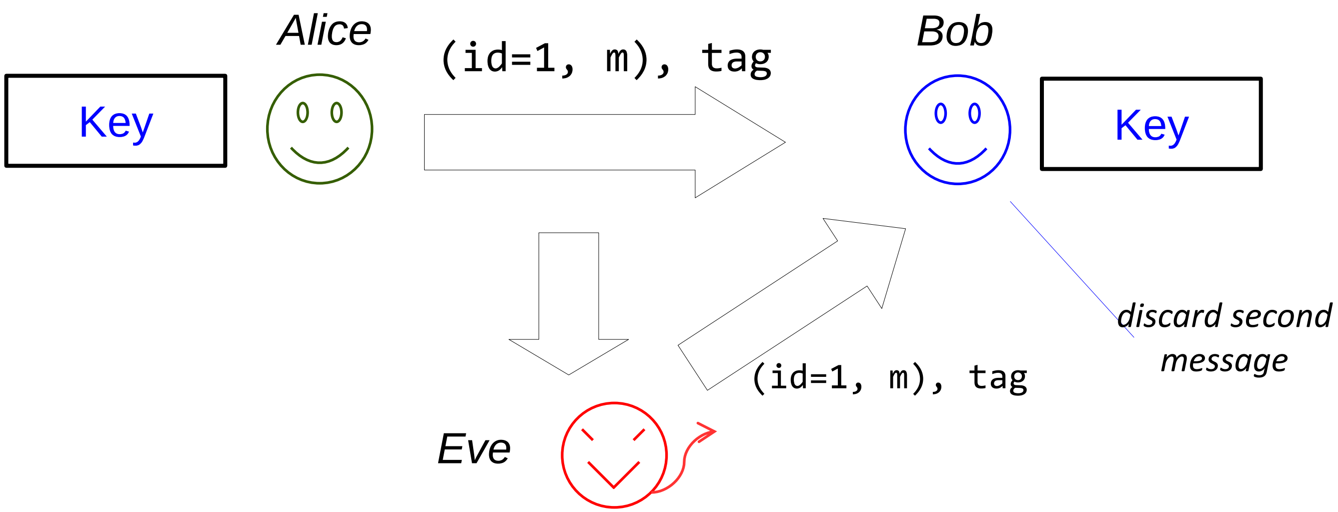
- A message authentication code guarantees that the symmetric key has been used to produce the tag



- 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 “Take 1000$ from my account and put them on Eve’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 from such attacks. However, due design choices at the upper layers (e.g., transport or application layers) can prevent such an attack
  - Without going into details the common defense against such an attack is to implement unique counters/identifiers within the message

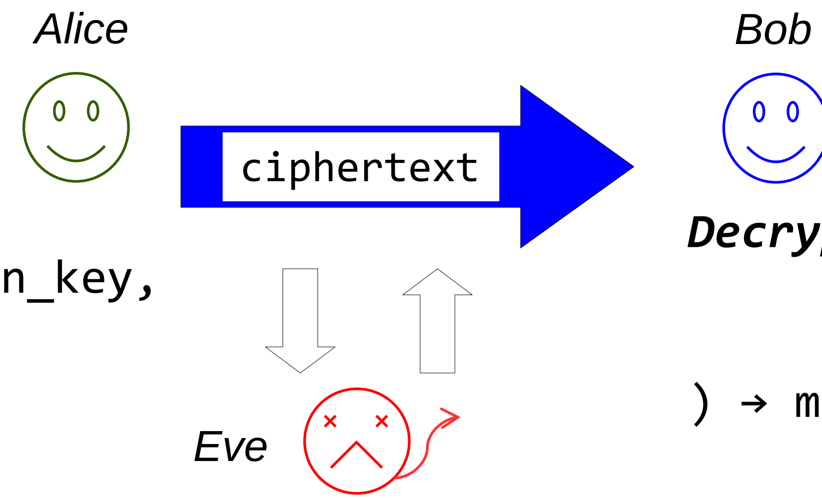
Examples: hashes and HMACs

- Compute the digest of a file
  - openssl dgst --sha256 file
  - sha256sum file

- Compute the HMAC of a file
  - openssl dgst --sha256 --hmac key file

Asymmetric encryption

Asymmetric setting




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

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

- In asymmetric settings Alice and Bob use different keys

encryption_key != decryption_key

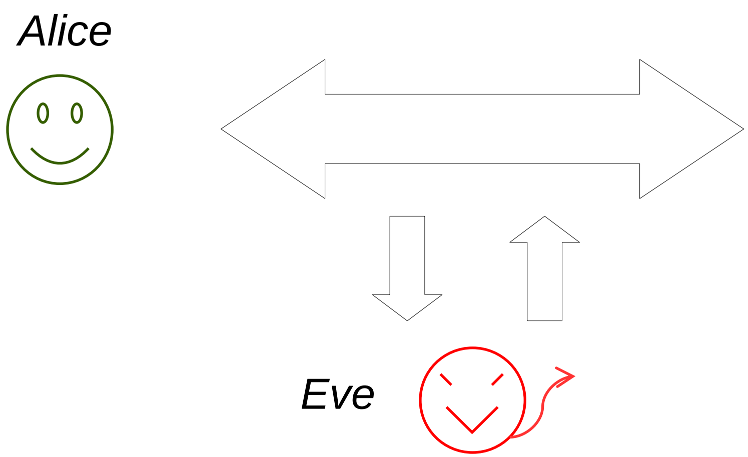
- The encryption key is **public**, the decryption key is **secret**

Protocols

- Most famous protocols based on asymmetric crypto
  - two-party key exchange
  - digital signatures
  - asymmetric encryption
  - authenticated key exchange

Key exchange protocols [1]

- Actually, we do not start with encryption, but with key exchange protocols



- Alice and Bob have no shared key, yet they want to communicate over an insecure channel
- A key exchange protocol allows Alice and Bob to obtain a shared key
  - over an insecure channel
  - without the help of third parties

Trapdoor functions

- Mathematical problems that allow to design trapdoor functions
- Computing the function in one-way is always easy
  - if you know the secret, the inverse is also easy
  - if you don’t, the inverse is hard

- Easy → ~ Polynomial
- Hard → ~ Exponential

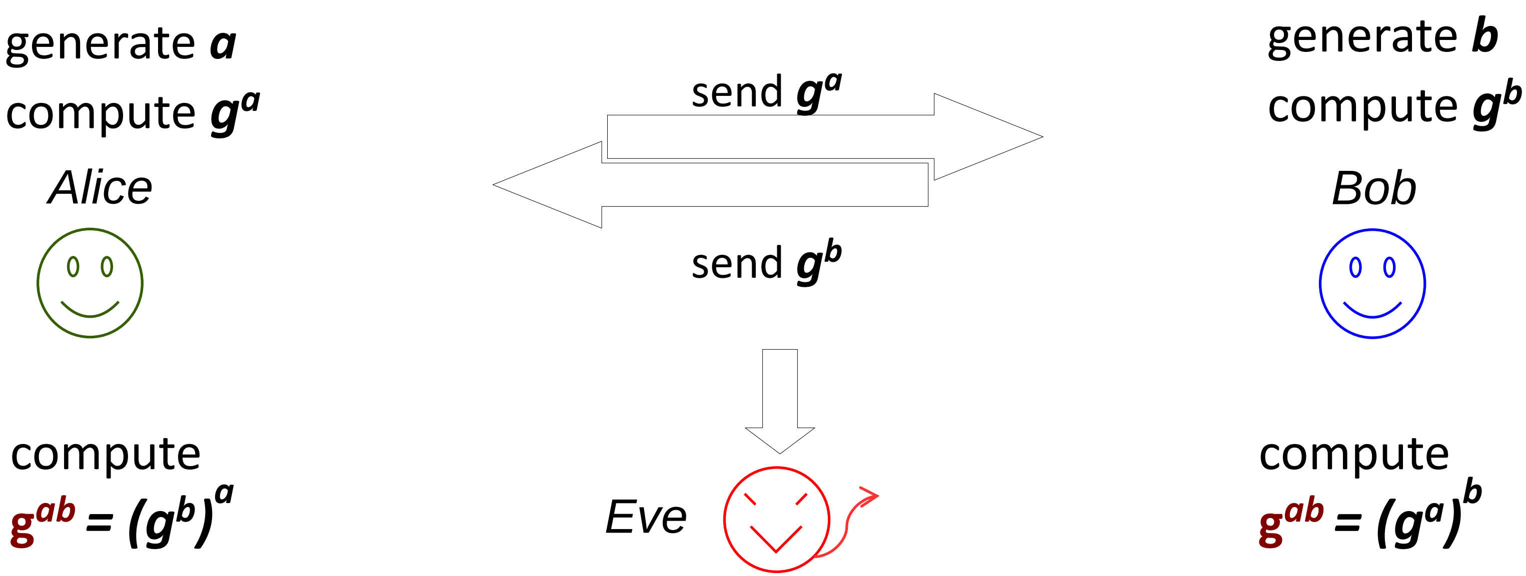
- Factorization — Modular arithmetic
- Discrete logarithm
  - Computational DH
  - Decisional DH

Modular arithmetic, Elliptic curve

Diffie-Hellman Hard Conjectures

- Take generator g of a cyclic group of prime order p
  - Discrete logarithm problem
    - if you know g^a it is hard to compute a
  - Computational Diffie-Hellman
    - given g^a and g^b it is hard to compute g^{ab}
  - Decisional Diffie-Hellman
    - given g^a and g^b it is hard to distinguish g^{ab} from g^r where r is random

Diffie-Hellman Key Exchange



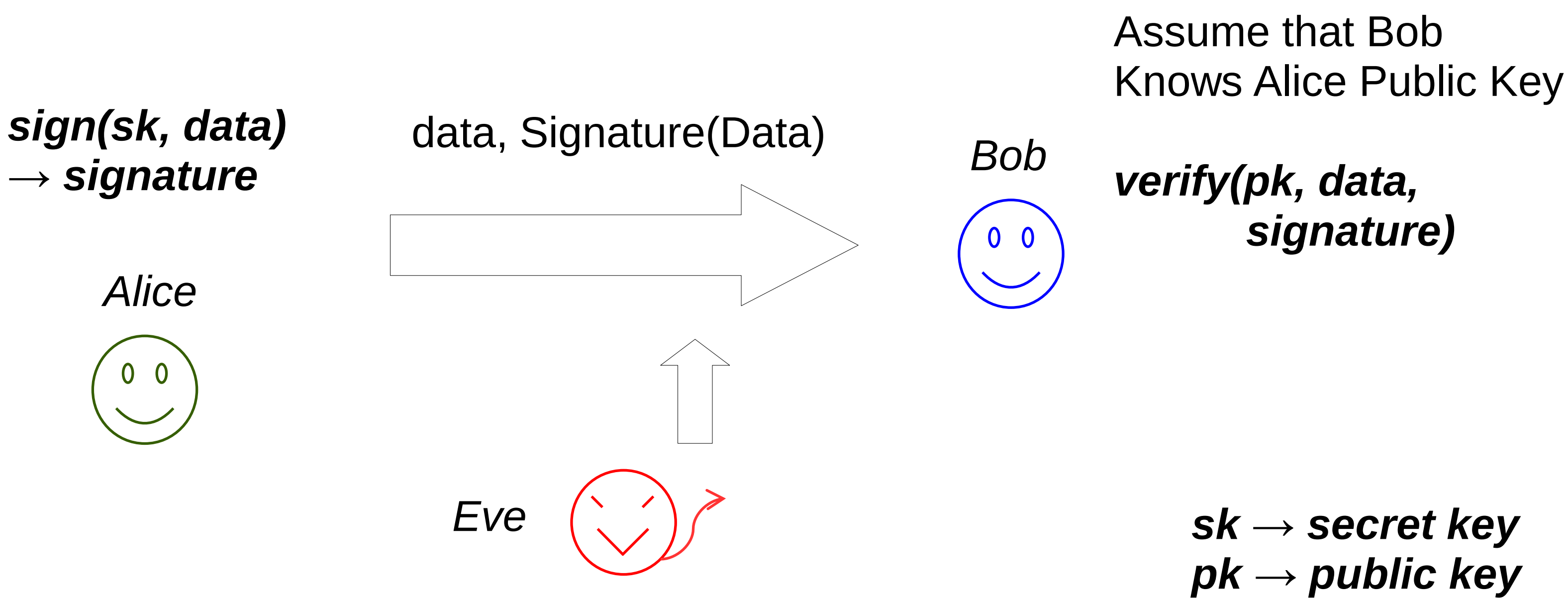
Only Alice and Bob can compute g^{ab}
(and derive a symmetric key, see TLS)
Eve cannot compute it and cannot "distinguish it from random"

Digital signatures [1]

- 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 [2]

- A digital signature authenticate 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



Signature and Encryption Key pairs [1]

- We said that Alice and Bob have different keys

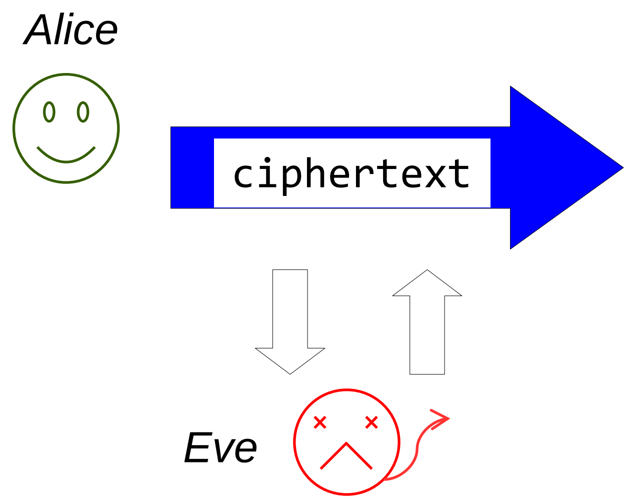
- Asymmetric crypto introduces the term “key pair”
  - a key pair is a couple of keys
  - (secret-key, public-key)

- Each key pair is usually associated with a User
  - the public key of a user is unique, and identifies him

- The protocols presented here assume that other users know the association between public keys and users

Signature and Encryption Key pairs [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

- Not really viable in large Networks

Asymmetric encryption [1]

- encrypt(pk, message) → ciphertext
  - we assume the probabilistic interface (i.e. implicit IV)
- decrypt(sk, ciphertext) → message
- Asymmetric encryption is much slower then symmetric encryption
- Usually, asymmetric encryption is used in combination with symmetric schemes
  - → Hybrid schemes (also, see key encapsulation mechanisms - KEMs)

Asymmetric encryption [2]

- Key encapsulation and hybrid encryption schemes
- To encrypt:
  - Generate a symmetric key
  - Encrypt symmetric with public key of recipient
  - Symmetric key encryption on data

EP(pk, k)	ES(k, data)

- To decrypt:
  - Decrypt symmetric key with own private key
  - Decrypt data with symmetric key

Example: RSA encryption/decryption from command line

- Create keypair
  - openssl genrsa -aes128 -out alice_private.pem 1024
  - openssl rsa -in alice_private.pem -pubout > alice_public.pem
- Encrypt a file
  - openssl rsautl -encrypt -inkey bob_public.pem -pubin -in top_secret.txt -out top_secret.enc
- Decrypt a file
  - openssl rsautl -decrypt -inkey bob_private.pem -in top_secret.enc > top_secret.txt
- Sign a file
  - openssl dgst -sha256 -sign alice_private.pem -out signature file.txt
- Verify a signature
  - openssl dgst -sha256 -verify alice_public.pem -signature signature file.txt

Public Key Infrastructure

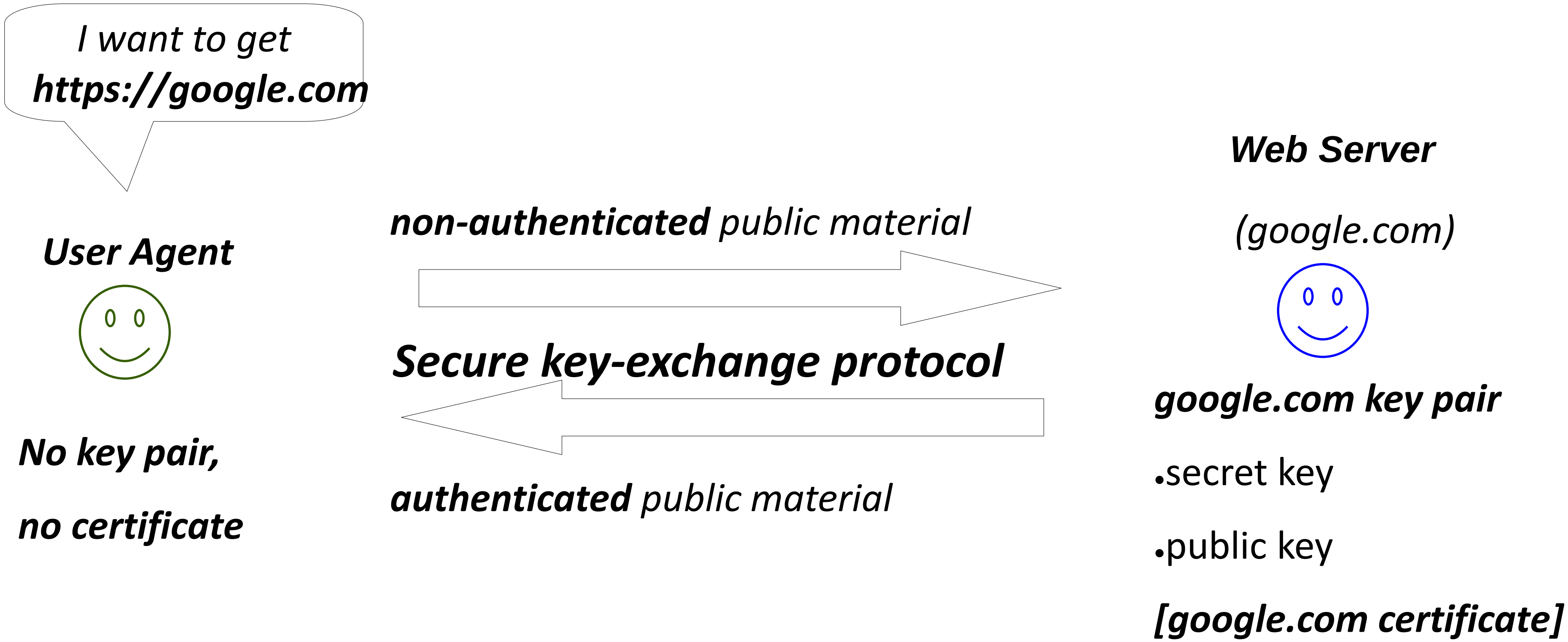
Public Key Infrastructure [1]

- Each participant uses a Key Pair
  - Encryption
    - Secret Key → to decrypt
    - Public Key → to allow others to encrypt for you
  - Signature
    - Secret Key → to sign
    - Public Key → to allow others to verify your messages
- The PKI define how to securely distribute the public keys by using intermediate trusted parties

Public Key Infrastructure [2]

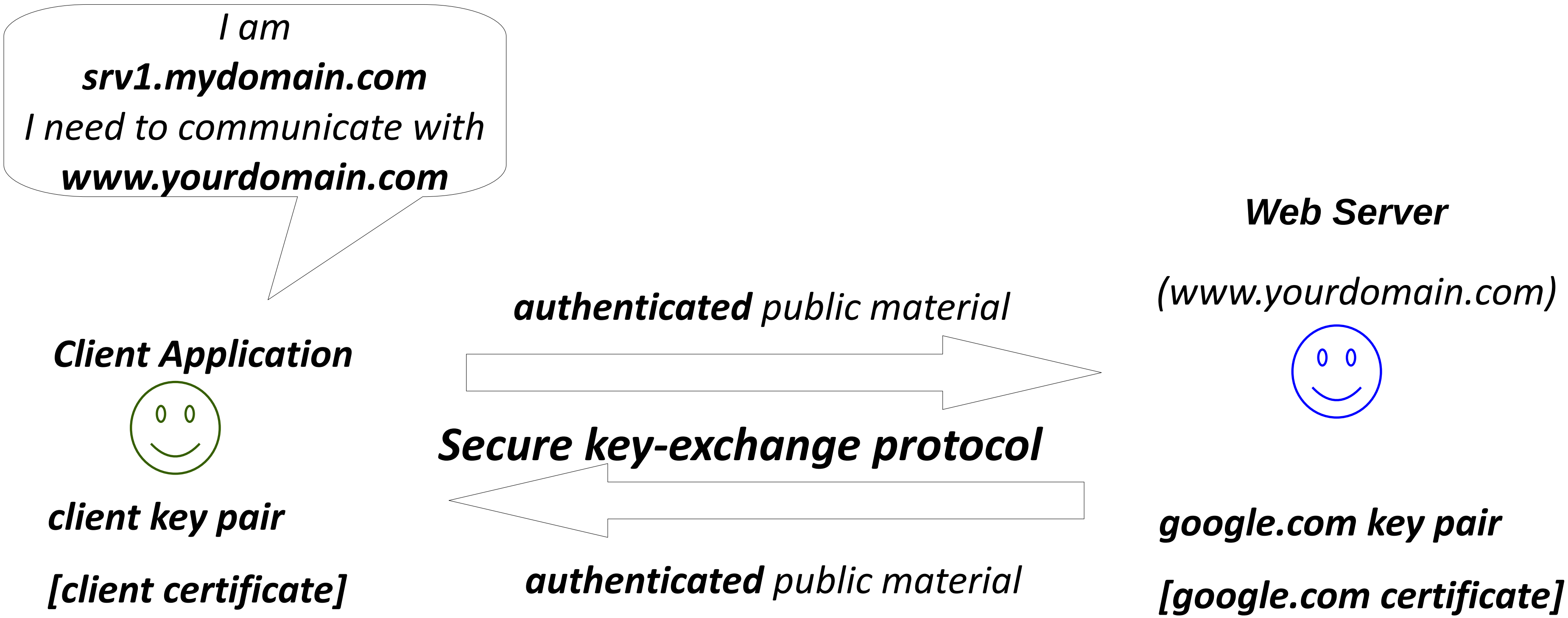
- Key pairs need metadata. Examples:
  - what are the protocols and the crypto parameters?
  - when have they been issued?
  - when do they expire?
  - **who is the owner**?
  - ...
- PKI is based on CERTIFICATES
  - A certificate is a digitally signed document that binds some information to cryptographic material
  - x509 standard

Scenario: MITM protection in Web communications



- The user agent verifies server’s identity
- The Web server does not authenticate the User Agent

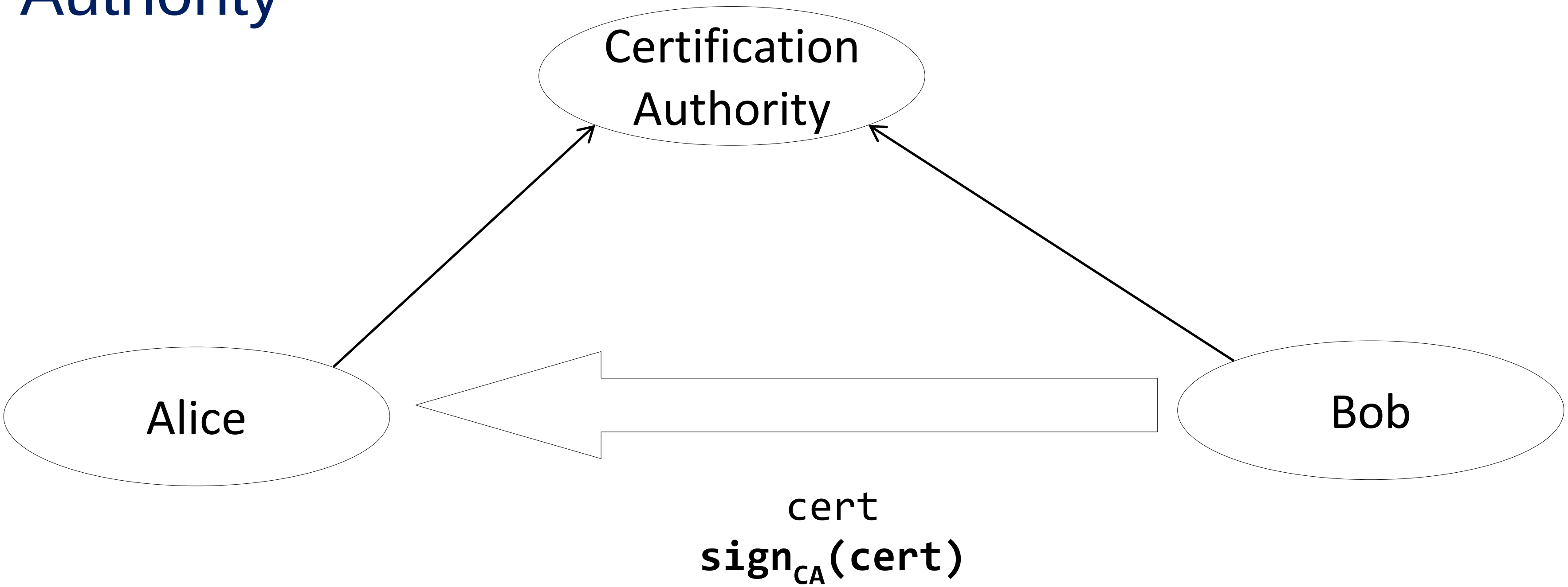
Scenario: client authentication for distributed services communications



- The User agent authenticates the server's identity
- The Web server uses crypto to authenticate the "Client"
  - often adopted for B2B Web services

Trusted third party approach

- Hi Alice, I’m Bob
  - here is my Public key
  - here is the certificate where our mutual friend confirms this information
- The trusted third party in PKI is the Certification Authority



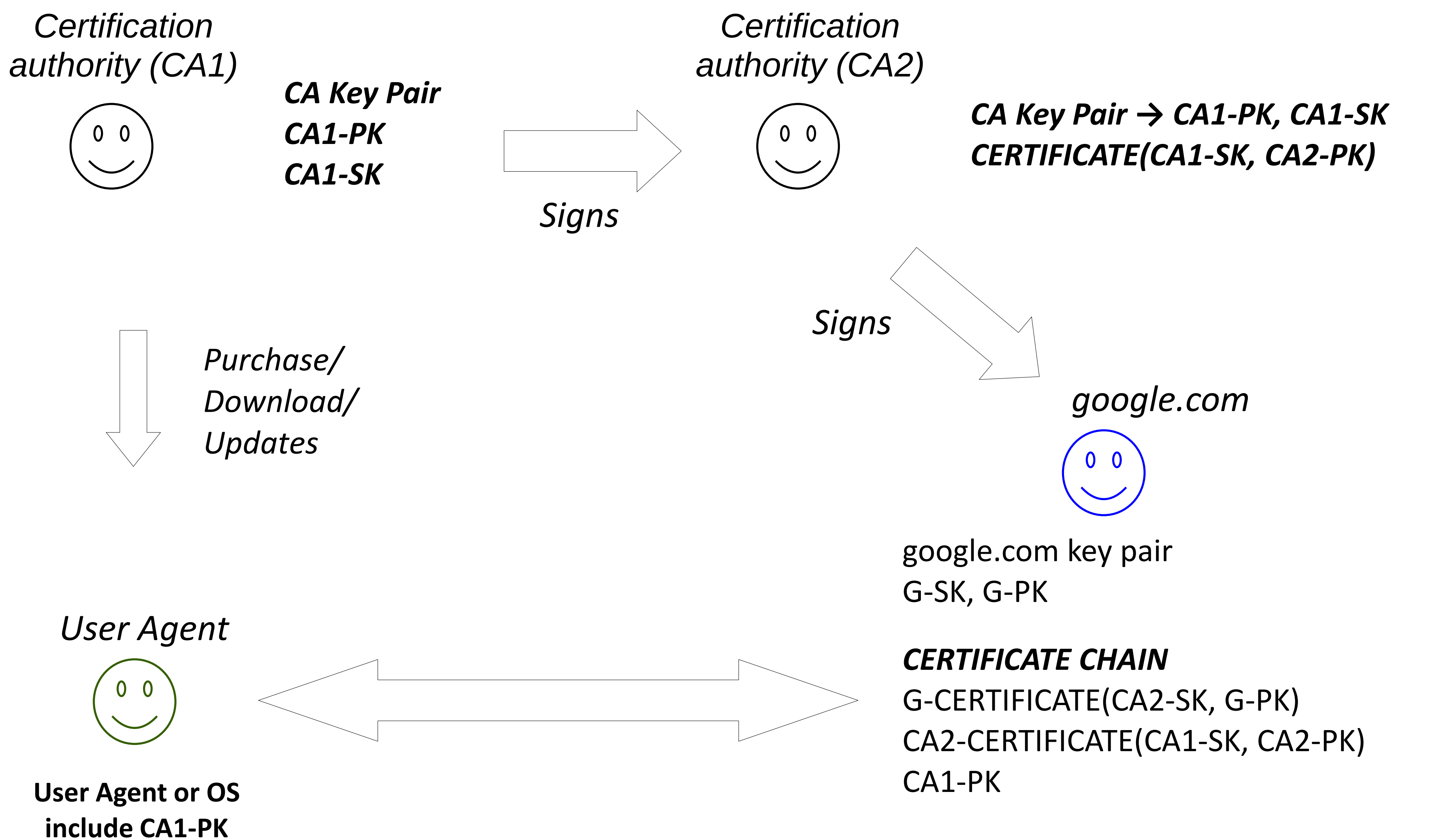
Certification authority

- The Certification Authority (CA) releases certificates that bind the public key to an entity
  - persons
  - role
  - organizations
  - devices
- Entities might include identification information
  - common name, country, state, city, …
- The certificates include additional metadata and information

Delegating certificates

- Requiring a few authorities to sign all certificates is not scalable
  - point of failures
  - political and economic conflicts
  - complex configurations
- A hierarchical approach is a viable trade-off
  - the root CA certificates a CA that certificates a CA that….
  - PKI trust model
- A few CAs PK included in the software allow to verify a huge amount of certificates

Example of certificate chain



Certificate chain

- The server has a certificate, issued by an intermediate CA
  - The intermediate CA has a certificate, issued by another intermediate CA or a root CA
  - Root CAs are known and installed in operating systems and Web browsers
    - https://www.mozilla.org/en-US/about/governance/policies/security-group/certs/
- To verify the end-user certificate, a client needs to verify all certificates in the chain, until it finds a known trusted certificate
- A server may return the full certificate chain
  - or assume that the client knows many “famous” intermediate servers and only return the end certificates
    - Web browsers often store more certificates then the OS

Example: HTTPS

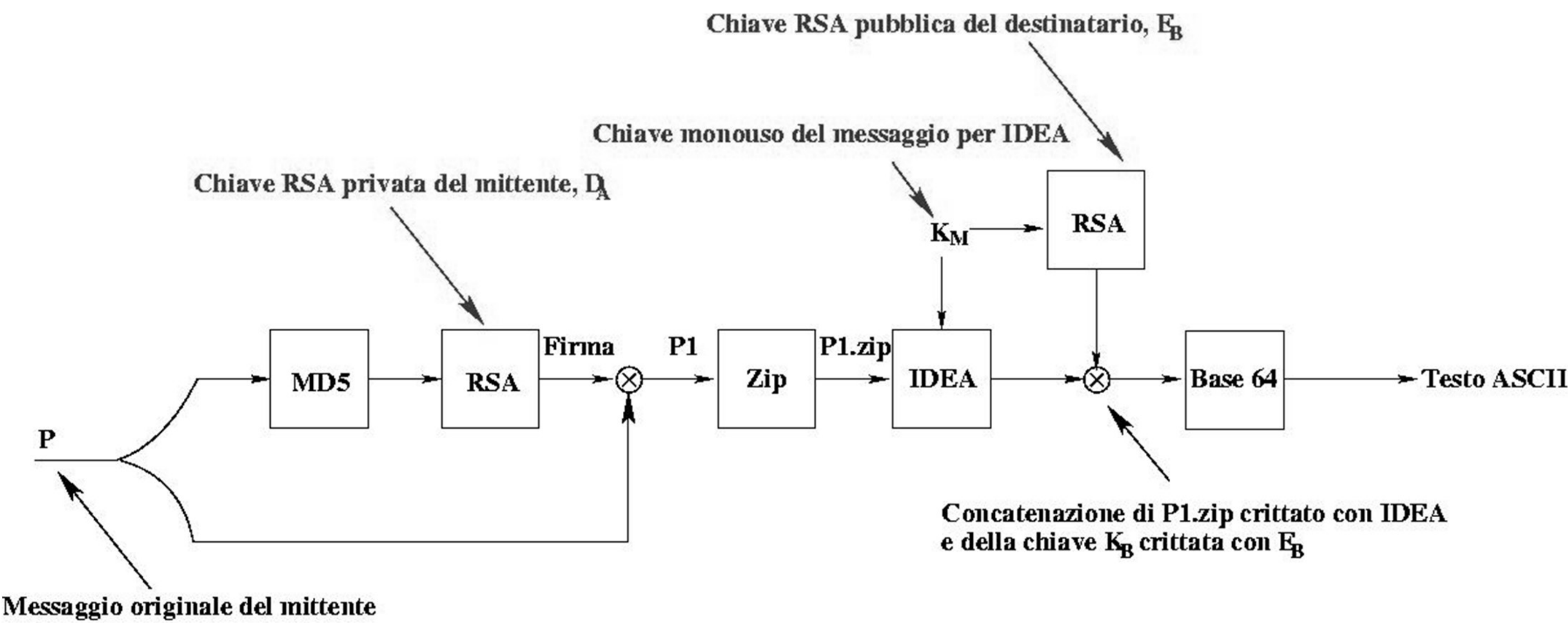
- HTTPS is «just» HTTP over TLS
- Standard port number: 443

- Step 1 → establish TCP connection with server
- Step 2 → carry out the TLS handshake
  - Allows client to authenticate server
  - Server and client use Diffie-Hellman to establishe a symmetric session key
- Step 3 → send normal HTTP requests/responses inside the secure TLS channel

- Test with
  - Openssl s_client --connect www.unimore.it:443

Example: Secure email

- PGP/GPG scheme



Example: Secure email

- Example with encrypted and signed message

- Step 1: extract encrypted file from email
- Step 2: decrypt file
  - gpg2 --decrypt encrypted.asc > decrypted
- Step 3: extract signature
- Step 4: remove signature envelope to recover the original message
- Step 5: verify signature against original message
  - gpg2 --verify signature.asc original-message