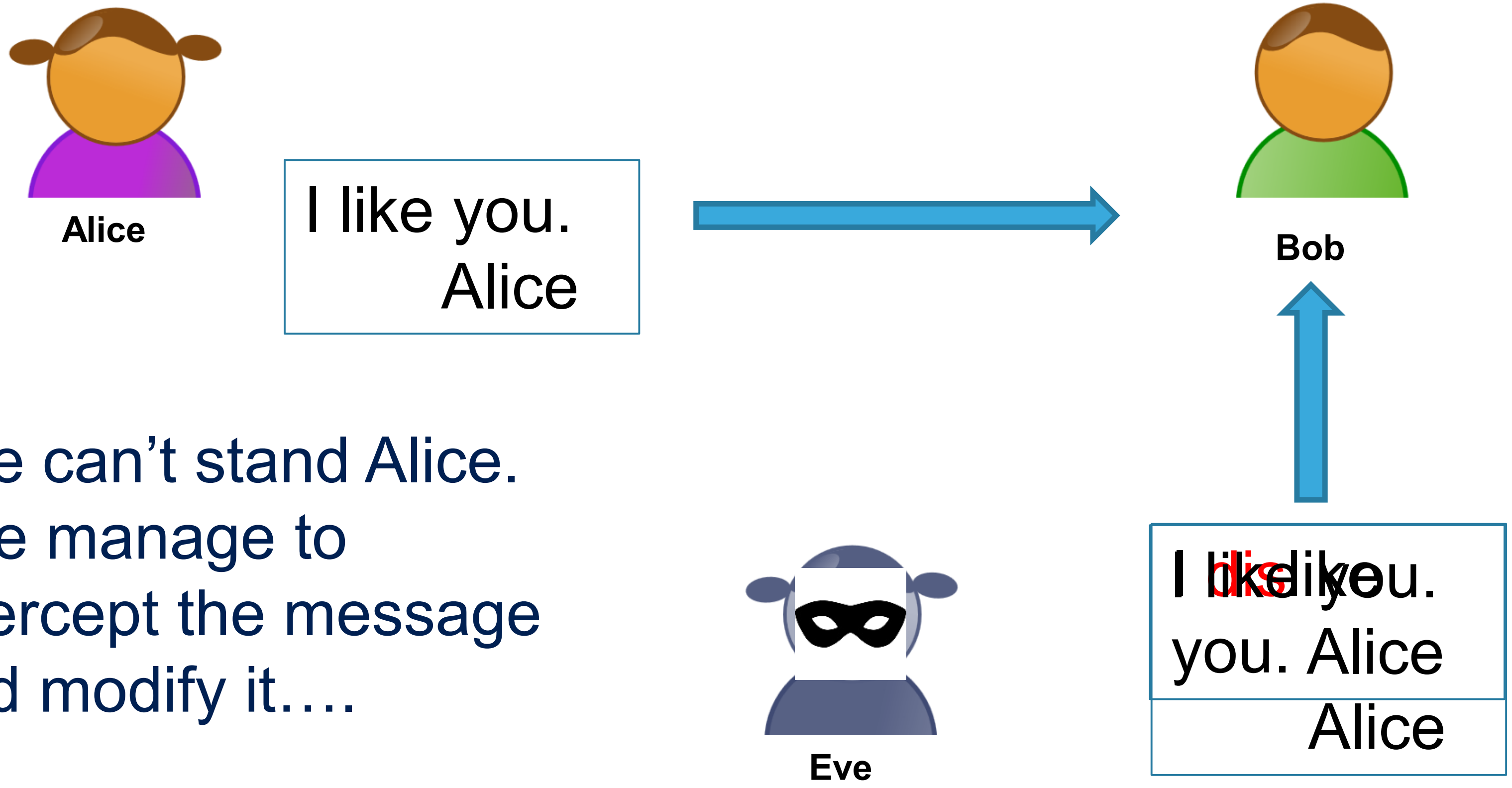
Basics of cryptography

life.augmented

Love story…

- Alice like Bob …. She decided to send him a letter.



I like you.
Alice

I dislike you.
I like you. Alice
Alice

- Eve can’t stand Alice. She manage to intercept the message and modify it….

What do we want to do ?

- Exchange confidential messages: no one can see the message except Bob
- Ensure message integrity : no modification made by Eve
- Authentication: Bob wants to be sure the message comes from Alice

Let’s see how to address this thanks cryptography….

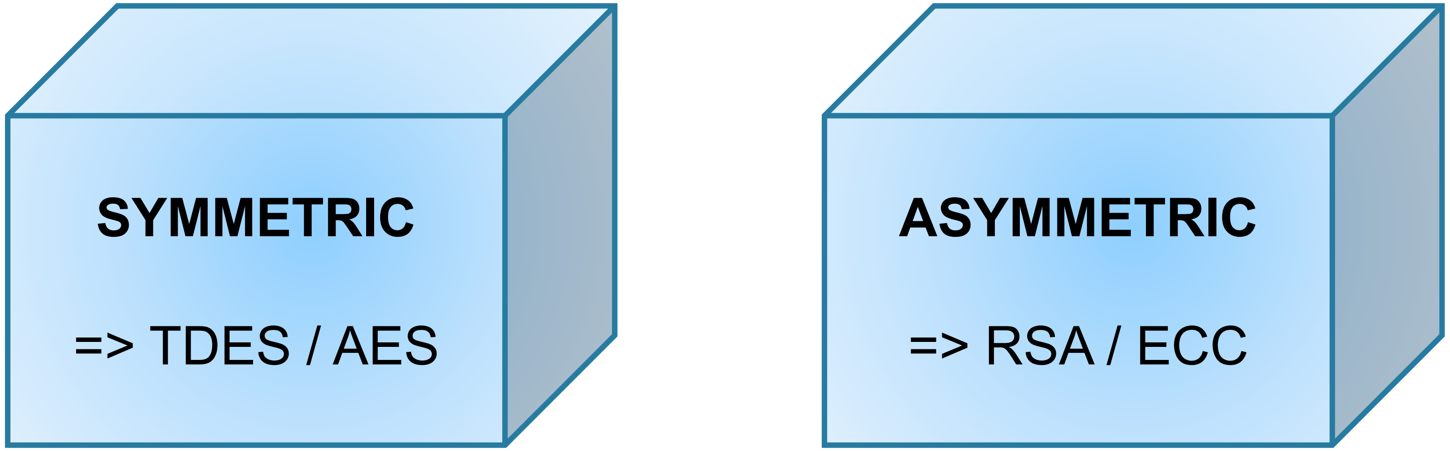
Cryptography is the science and art of transforming messages to make them secure and immune to attack

Exchanging confidential data

Encrypt the message

- Confidentiality of exchange will be addressed with encryption/decryption mechanism.

- 2 main categories for this :



SYMMETRIC

=> TDES / AES

ASYMMETRIC

=> RSA / ECC

- Encryption/Decryption need KEYs.

Agenda

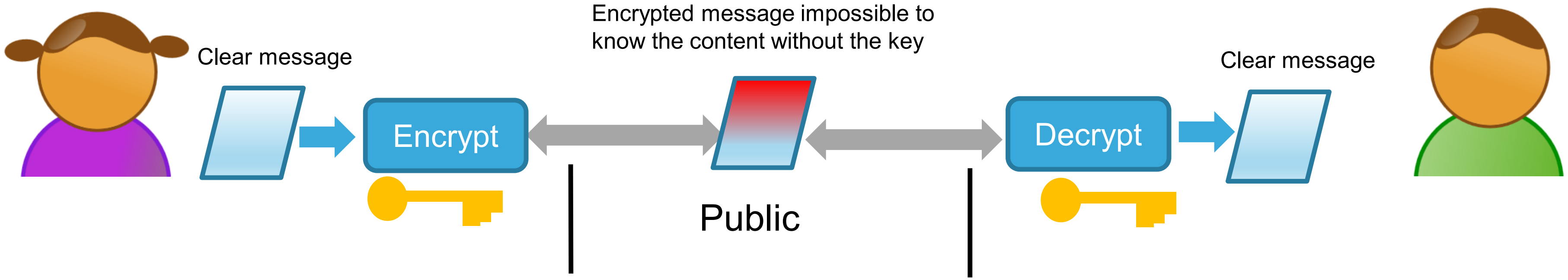
- Principle:
  - Symmetric encryption/decryption theory
  - Asymmetric encryption/decryption theory
  - Combination of Symmetric/Asymmetric
  - Shared secret generation

- Main algorithm
  - Symmetric : TDES, AES
  - Asymmetric : RSA, Elliptic curves, Diffie-Hellman
  - Combination of Symmetric/Asymmetric: IES

Encryption/decryption theory

Symmetric cryptography

- Alice and Bob share a single, common secret, cryptographic key
- The secret key is used with a symmetric encryption algorithm to encrypt and decrypt the message



Clear message

Encrypt

Encrypted message impossible to know the content without the key

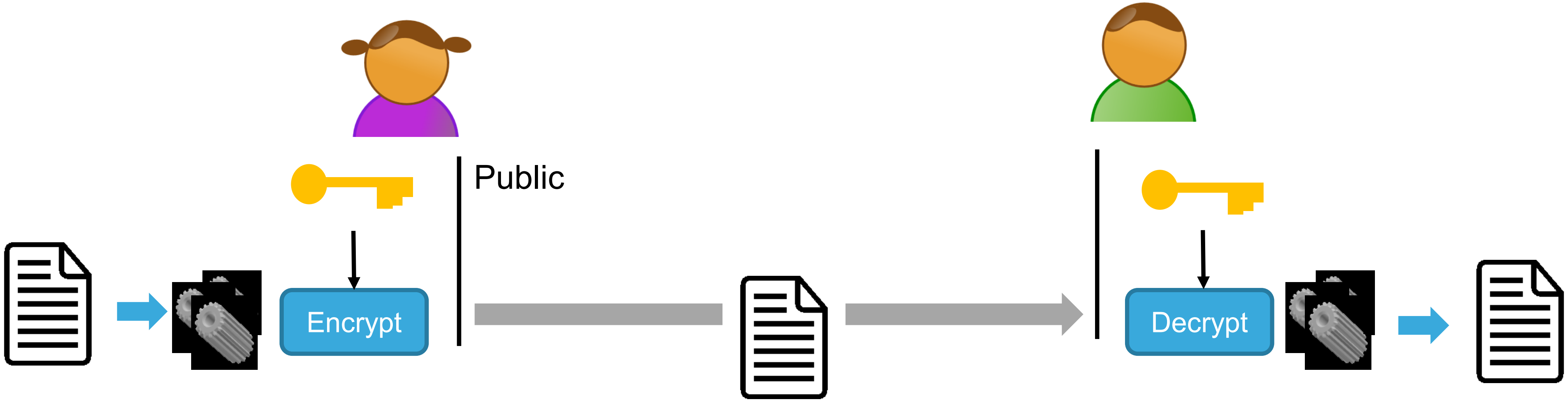
Public

Decrypt

Clear message

- Symmetric-key systems are simple and fast
- Main drawback: the two parties must exchange the key in a secure way

Symmetric cryptography

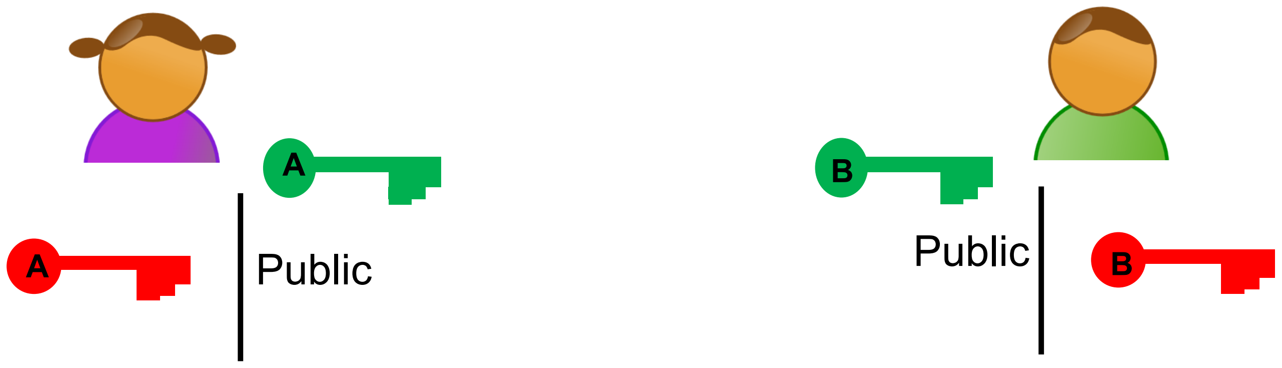


Hands-on

- In the folder Hands-on\01_SymmetricEncryption
  - > Encrypt_sym.bat 123455 message.txt message_encrypted.txt
  - > Decrypt_sym.bat 123455 message_encrypted.txt message_decrypted.txt

Asymmetric cryptography

- The encryption and the decryption are done with different keys…
- Alice and Bob now have a key pair



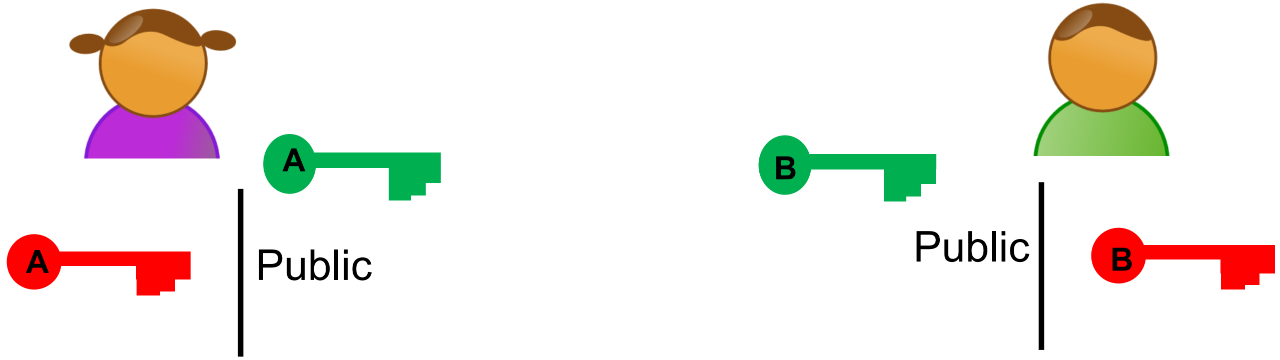
- Main drawback : asymmetric-key systems are complex (HW or SW), generally not used to encrypt/decrypt big data.

Asymmetric cryptography

- The encryption and the decryption are done with different keys….
- A key pair is composed of:
  - a public key available to anybody 🟩
  - a private key that should be keep secret 🟥

- Thanks mathematics properties, keys are linked together so that:
  - If you encrypt something with 🟩 you can only decrypt it with 🟥
  - If you encrypt something with 🟥 you can only decrypt it with 🟩

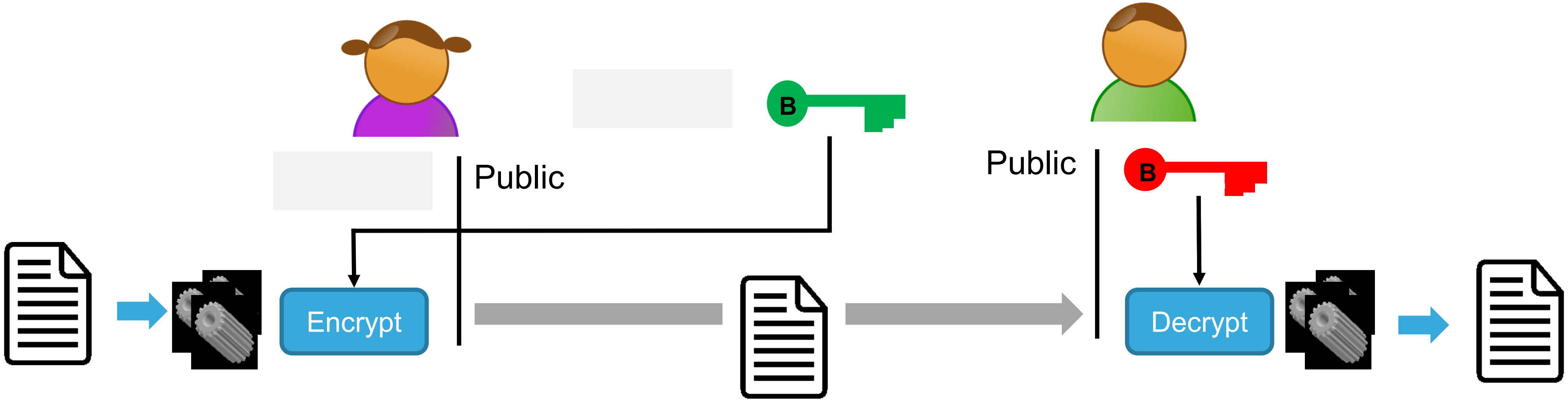
Which key should I used ?



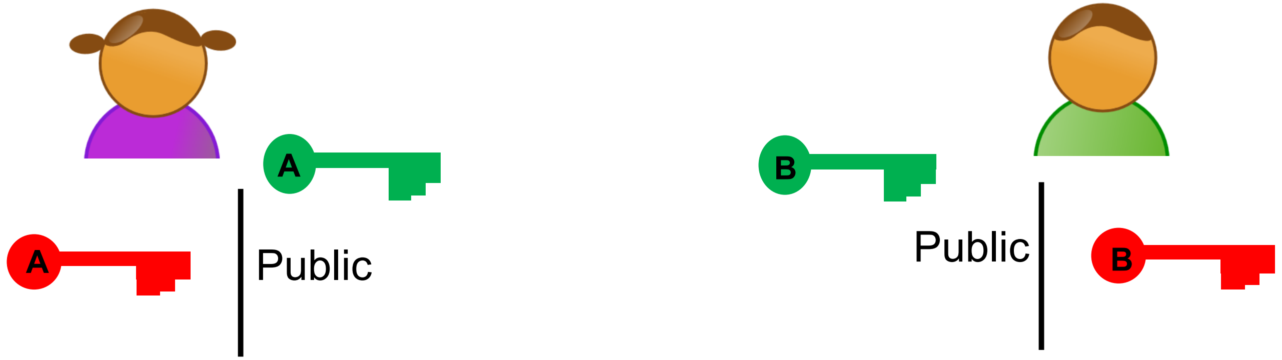
- Alice want to send a secret message to Bob…Which key should she use for encryption ?

  - If Alice use her private key 🟥A ….So everybody could decrypt this message with 🟩A
  - If Alice use her public key 🟩A ….Only Alice could decrypt it with 🟥A
  - If Alice use Bob public key 🟩B …..Only Bob could decrypt it with 🟥B

Asymmetric cryptography



To keep in mind…



- In asymmetric, when you want to send an encrypted message, you must encrypt it with the public key of the recipient

- Public key of Bob is public…So anybody can send an encrypted data to Bob

Hands-on

In the folder Hands-on\AsymmetricEncryption

```
> Encryt_asym.bat BobPublicKey.pem .\Alice\message.txt message_encryted.txt
> Decryt_asym.bat .\Bob\BobPrivKey.pem message_encryted.txt
.\Bob\message_decrypted.txt
```

Asymmetric vs Symmetric

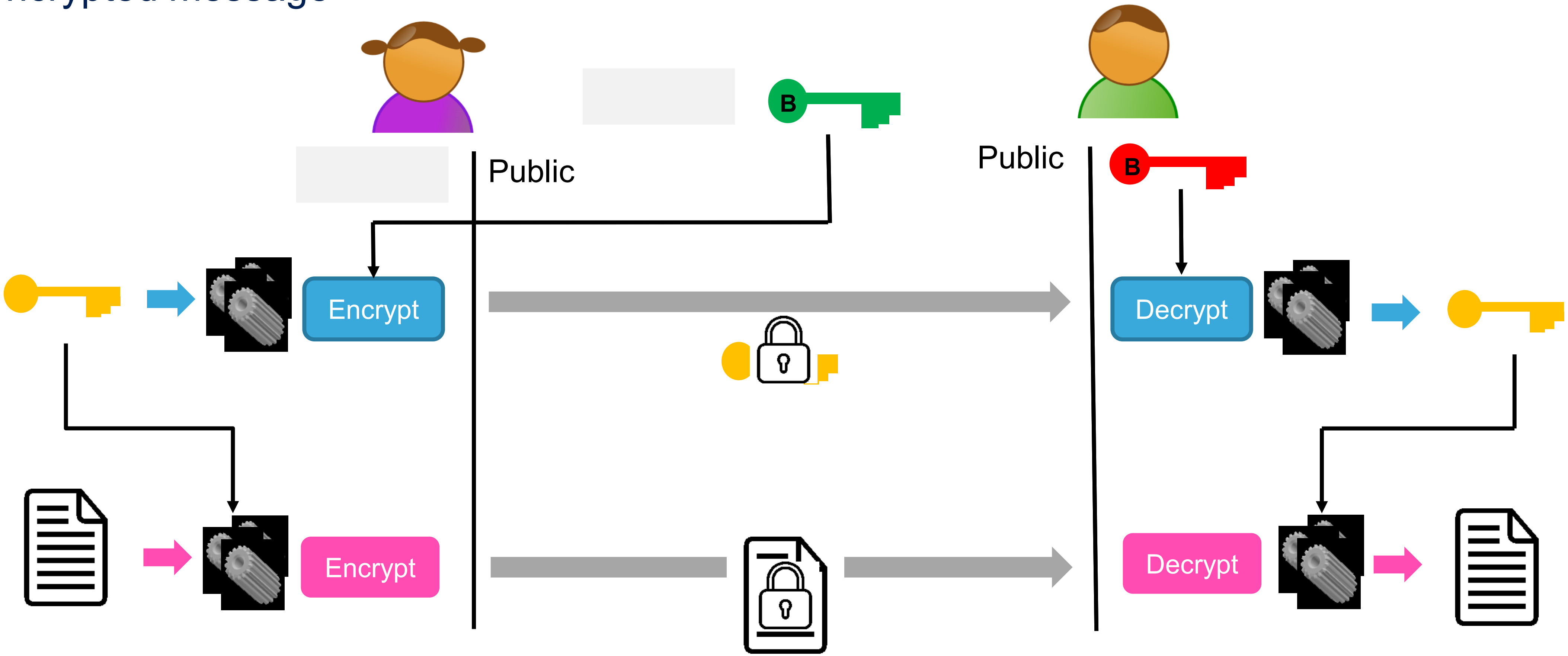
- Symmetric encryption advantage: fast computation
- Symmetric encryption draw back: you need to share a secret (key)
- Asymmetric encryption advantage: no need to share a secret
- Asymmetric encryption draw back: slow computation so can’t encrypt large data in an efficient way.

Let’s see how to combine them

Asymmetric cryptography usage

- Combine symmetric and asymmetric cryptography :

Transmit symmetric key through asymmetric encryption, then transmit symmetric encrypted message



Hands-on

- In the folder Hands-on\Sym_and_AsymmetricEncrytion

```
>type Alice\symetric_key_value.txt
>Encryt_asym.bat BobPublicKey.pem .\Alice\symmetric_key_value.txt secret_key_encrypted.txt
>Decryt_asym.bat .\Bob\BobPrivKey.pem secret_key_encrypted.txt .\Bob\secret_key_received.txt
>type Bob\secret_key_received.txt
>Encryt_sym.bat 12345 .\Alice\message.txt .\message_encrypted.txt
>Decryt_sym.bat 12345 message_encrypted.txt .\Bob\message_decoded.txt
```

Generate a shared secret

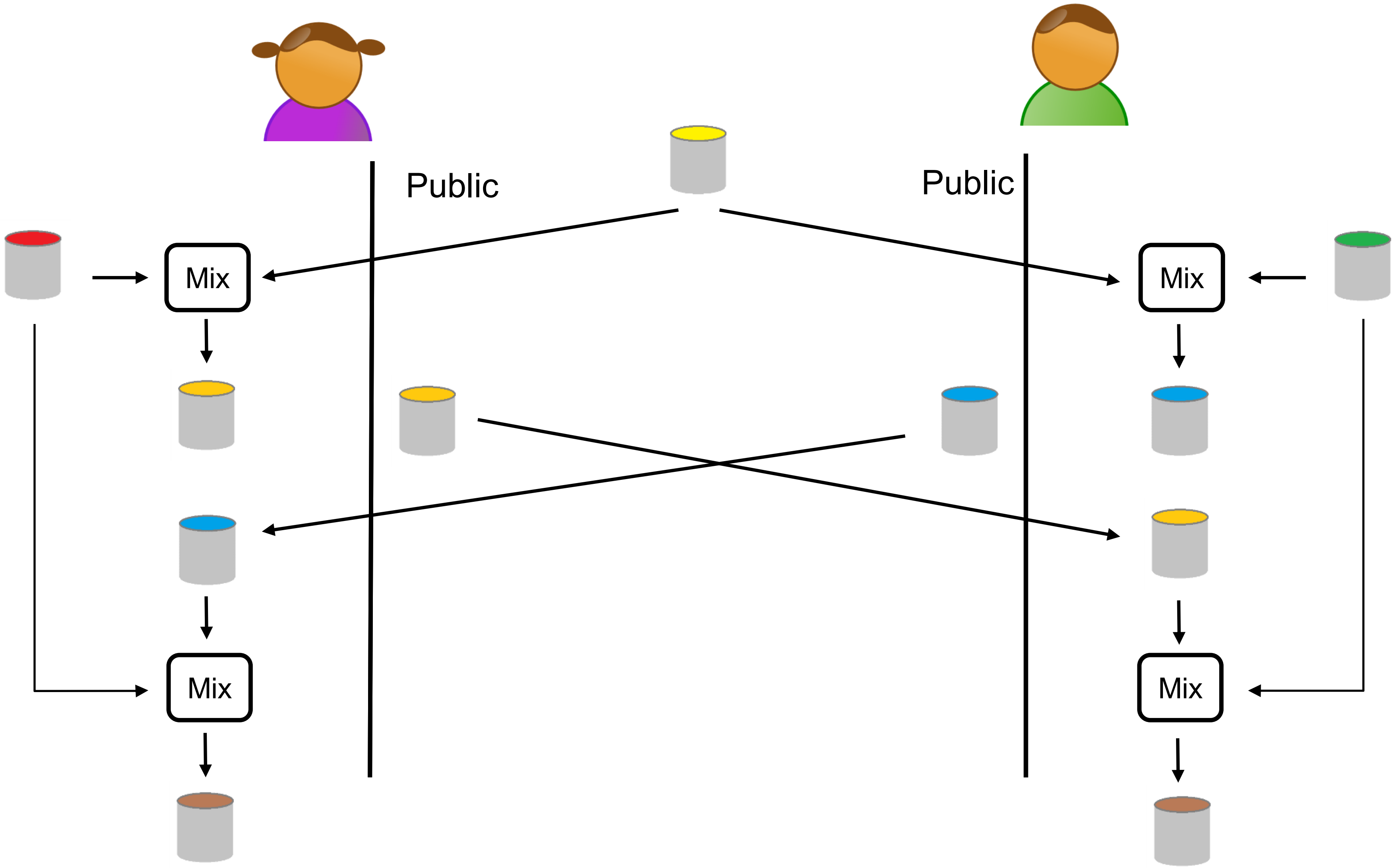
- An other solution would be to create a shared secret between Bob and Alice in a secure way

- This is possible thanks an algorithm called : Diffie-Hellman

- It is part of asymmetric crypto as we will have some private data and public data associated.

- To ease the understanding, we will expose this principle with the example of color mixing…

Generate a shared secret



Hands-on

- In the folder Hands-on\03_Diffie-Hellman

```
>Generate_a_common_color.bat common_color.pem
>Chose_and_Mix_color.bat Alice_color.pem common_color.pem
>Chose_and_Mix_color.bat  Bob_color.pem common_color.pem
>Generate_secret.bat Private_Bob_color.pem Melting_Alice_color.pem secret_bob
>Generate_secret.bat Private_Alice_color.pem Melting_Bob_color.pem secret_alice
```

Agenda

- Principle:
  - Symmetric encryption/decryption theory
  - Asymmetric encryption/decryption theory
  - Combination of Symmetric/Asymmetric
  - Shared secret generation

- Main algorithm
  - Symmetric : TDES, AES
  - Asymmetric : RSA, Elliptic curves, Diffie-Hellman, ECDSA

Encryption/decryption main algorithm

Symmetric cryptography

- Symmetric encryption is defined by :
  - Algorithm (TDES / AES…)
  - Block size (padding to be done if needed)
  - Key size

- Main symmetric encryption algorithm :
  - TDES : Triple data encryption standard
  - AES : Advance encryption standard

- Fast in software/hardware as both using internally only permutation/substitution/shift and XOR

TDES

- TDES (Triple data encryption standard) :
  - Based on the DES encryption (Feistel cipher) : block size 64 bits, key size 56 bits
  - Running 3 times the DES algorithm with 3 Keys :

ciphertext = Encrypt $_{K3}$(Decrypt $_{K2}$(Encrypt $_{K1}$(plaintext)))

-> if 3 keys different, TDES key size = 3* 56 = 168 bits

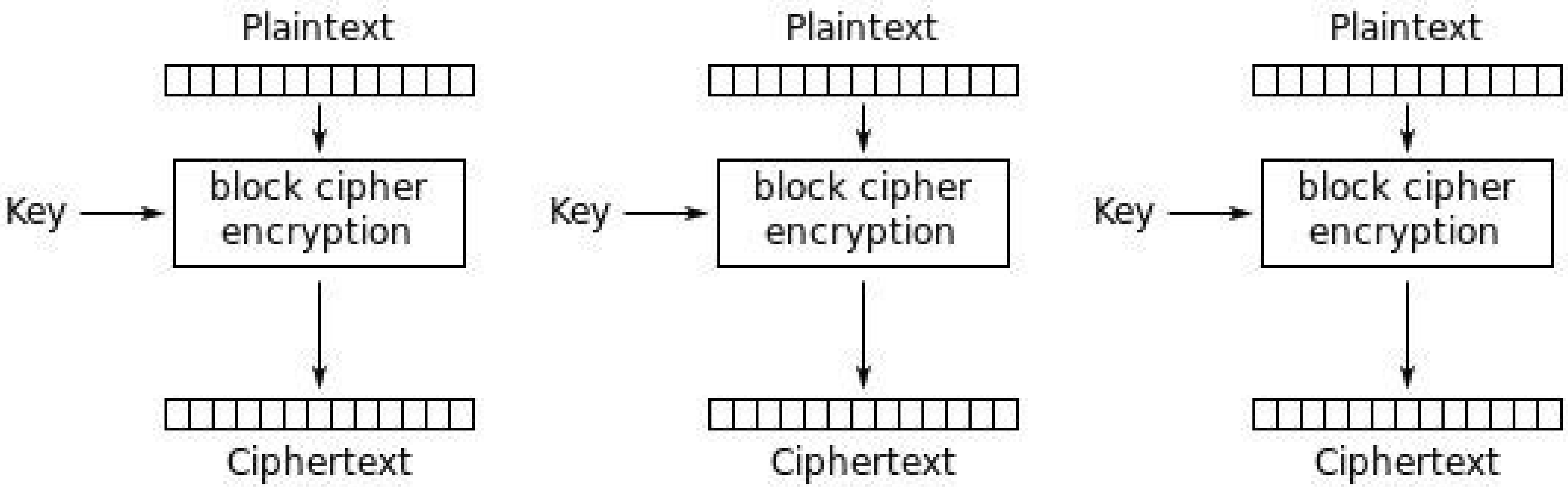
ciphertext = Encrypt $_{K1}$(Decrypt $_{K2}$(Encrypt $_{K1}$(plaintext)))

-> if 2 keys equal, TDES key size = 2*56 = 112 bits

AES

- AES (Advance Encryption Standard) :
  - Based on the substitution–permutation network (SPN) : block size 128 bits, key size 128, 192, 256 bits
  - AES operates on 2 dimensional array 4*4 bytes which is our 128 bits block input
  - The key size is link to the number of transformation done on the input (round)
    - 10 rounds for 128-bit keys.
    - 12 rounds for 192-bit keys.
    - 14 rounds for 256-bit keys.

AES ECB



Electronic Codebook (ECB) mode encryption

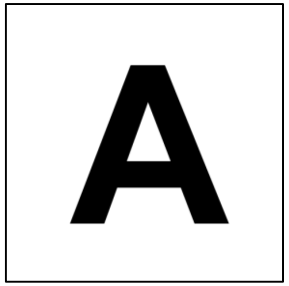
Hands-on

- In the folder Hands-on\Tools

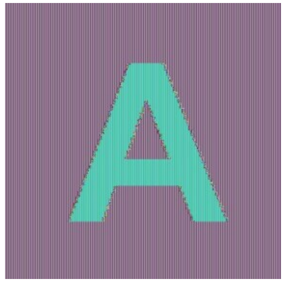
```
>hexdump  Example_AA_BB.bin
>openssl enc  -aes-128-ecb -k deadbeefdeadbeef -nosalt -nopad -in Example_AA_BB.bin -out Example_AA_BB.enc
>hexdump Example_AA_BB.enc
>openssl enc  -d  -aes-128-ecb -k deadbeefdeadbeef -nosalt -nopad -in Example_AA_BB.enc -out Example_AA_BB_clear.bin
>hexdump Example_AA_BB_clear.bin
```

Block ciphering mode

- With block ciphering ECB mode, same block input give same cyphered block output…Some pattern could be visible.



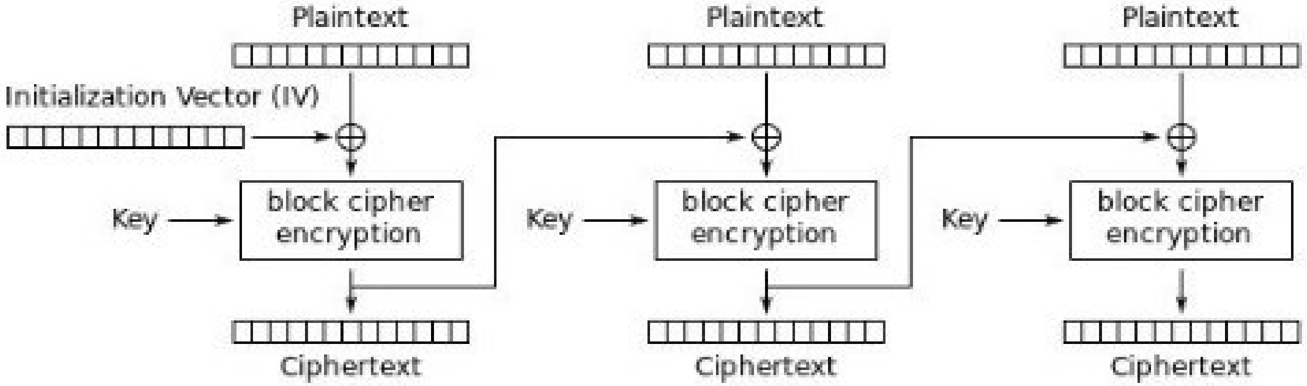
Clear message



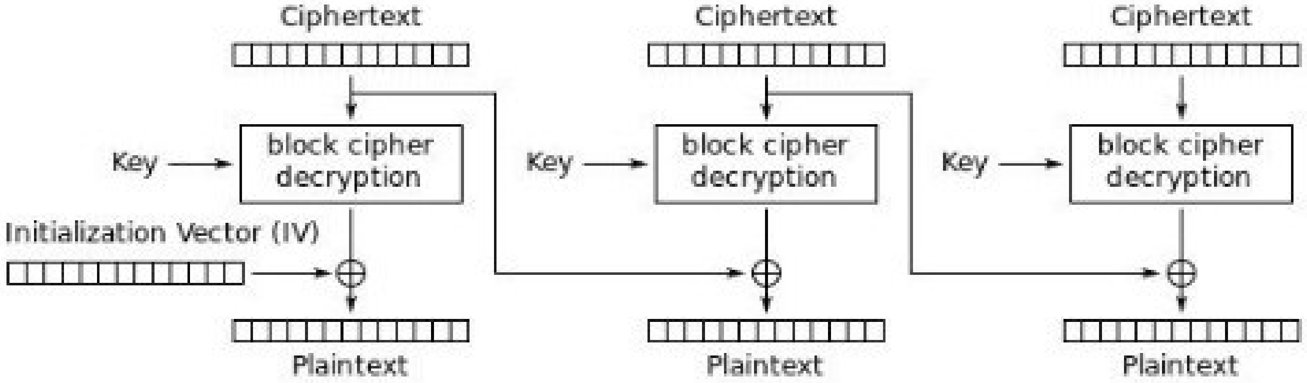
Encrypted using AES-128
ECB

- To avoid this problem we could :
  - Use some data from a previous block to encrypt a block . This is chaining mode (CBC / CFB / OFB)
  - Combine each block data with a counter. This is counter mode (CTR / CCM / GCM)

AES CBC



Cipher Block Chaining (CBC) mode encryption



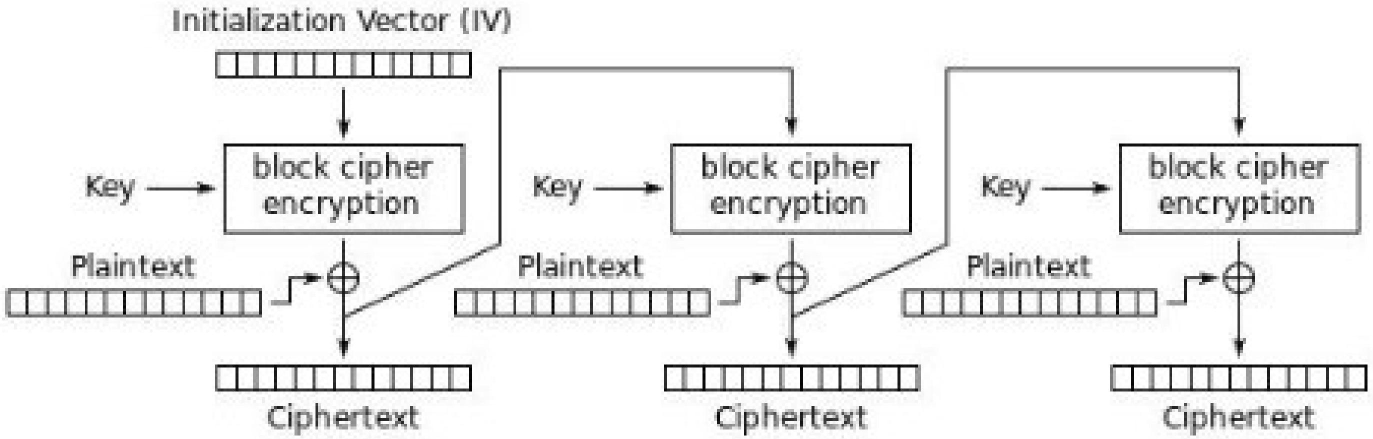
Cipher Block Chaining (CBC) mode decryption

Hands-on

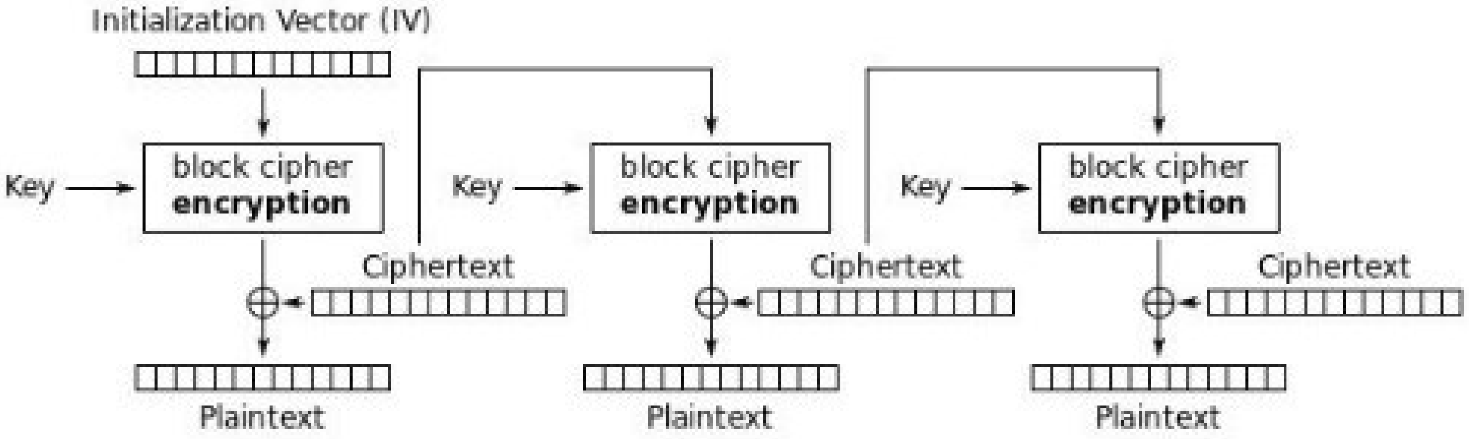
- In the folder Hands-on\Tools

```
>openssl enc  -aes-128-cbc -k deadbeefdeadbeef -iv deaddead -nosalt -nopad -in Example_AA_BB.bin -out Example_AA_BB_cbc.enc
>hexdump.exe Example_AA_BB.bin
>hexdump Example_AA_BB_cbc.enc

>openssl enc  -d  -aes-128-cbc -k deadbeefdeadbeef -iv deaddead -nosalt -nopad -in Example_AA_BB_cbc.enc -out Example_AA_BB_clear_cbc.bin

> hexdump Example_AA_BB_clear_cbc.bin
```

AES CFB

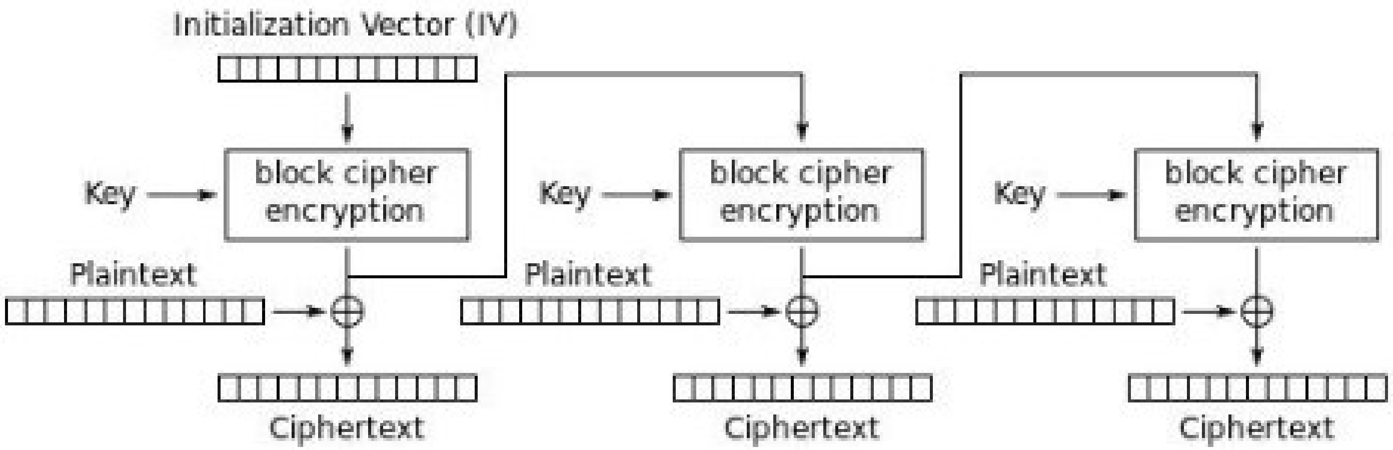


Cipher Feedback (CFB) mode encryption

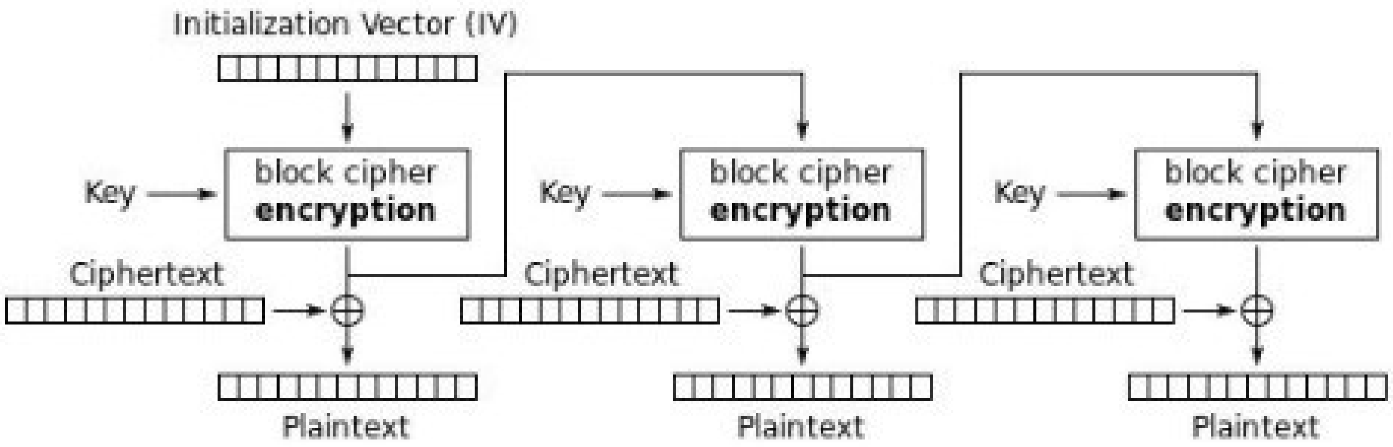


Cipher Feedback (CFB) mode decryption

AES OFB



Output Feedback (OFB) mode encryption



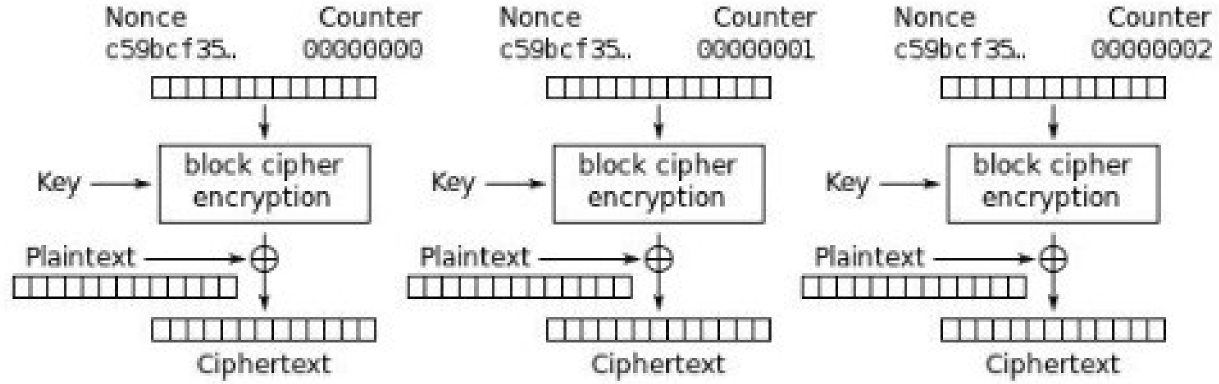
Output Feedback (OFB) mode decryption

Counter mode

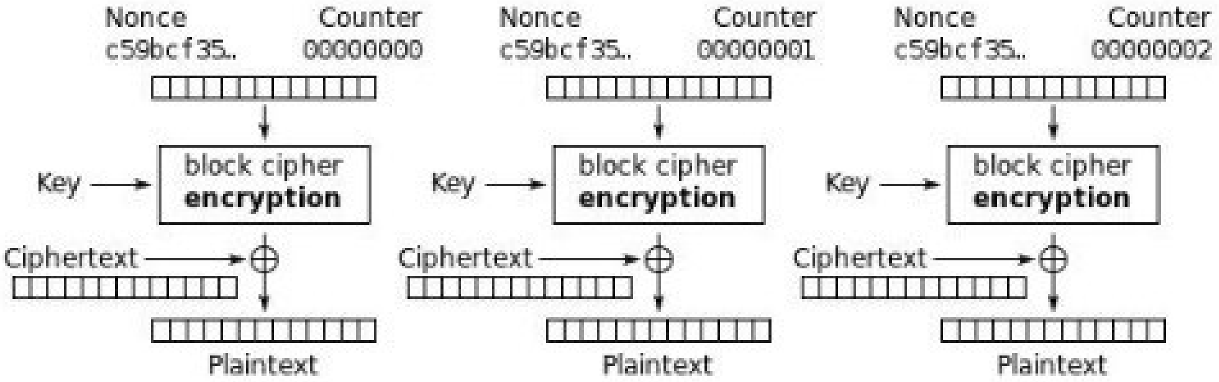
- An other solution would be to use random data and a counter to encrypt each block…This is called counter mode.

- Main counter mode : CTR, CCM, GCM

AES CTR



Counter (CTR) mode encryption



Counter (CTR) mode decryption

Hands-on

- In the folder Hands-on\Tools

```
>openssl enc  -aes-128-ctr -k deadbeefdeadbeef -iv dead0000 -nosalt -nopad -in
Example_AA_BB.bin -out Example_AA_BB_ctr.enc
>hexdump.exe Example_AA_BB.bin
>hexdump Example_AA_BB_ctr.enc
```

```
>openssl enc  -d  -aes-128-ctr -k deadbeefdeadbeef -iv dead0000 -nosalt -nopad -in
Example_AA_BB_cbc.enc -out Example_AA_BB_clear_ctr.bin
```

```
> hexdump Example_AA_BB_clear_ctr.bin
```

Comparison of symmetric algorithm

- AES could be considered as the successor of TDES
- TDES could encountered issue if you encode more than 32 giga bytes of data with the same key
- AES is faster than TDES with a lower memory footprint

Agenda

- Principle:
  - Symmetric encryption/decryption theory
  - Asymmetric encryption/decryption theory
  - Combination of Symmetric/Asymmetric
  - Shared secret generation
- Main algorithm
  - Symmetric : TDES, AES
  - Asymmetric : RSA, Elliptic curves, Diffie-Hellman

Asymmetric cryptography

- Asymmetric encryption characteristic :
  - Algorithm (RSA / ECC)
  - Key size

- Main asymmetric encryption algorithm :
  - RSA (Rivest–Shamir–Adleman) : key size commonly used 2048 to 4096 bits
  - ECC (Elliptic-curve cryptography) : key size commonly used 160 to 512 bits

- Complex operation (HW or SW), generally not used to encrypt big data.

RSA

- RSA (Rivest–Shamir–Adleman)
  - Based on the practical difficulty in factorization of the product of two large prime number

    N = P*Q (with P and Q are large and prime number)

    Knowing N, computing the individual values of P and Q is impossible in practice

  - Commonly used key size from 2048 to 4096 bits

RSA key creation

- First choose 2 prime number P and Q, compute N=Q*P

P=5, Q=11 so N=55

- Then chose E prime number which should have no prime factor common with (P-1)*(Q-1)

(P-1)*(Q-1) = (5-1) * (11-1) = 40 = 2*2*2*5 so E could be 7

- Public key will be : E = 7 and N = 55
- Now chose a number D which respect this rule : E*D mod ((P-1) * (Q-1)) = 1

7*D mod 40 = 1… D=23 is a good candidate.

- Private key will be : D = 23 and N = 55

How to encrypt a number M ?

$C = M^{E}$ modulo N

How to decrypt a number C ?

$M = C^{D}$ modulo N

RSA key format

- RSA key pair is composed :
  - E: public exponent
  - N: modulus
  - D: private exponent
  - P: prime 1
  - Q: prime 2

Public Key

Private Key

Needed for key creation but not used for encryption or decryption… Anyway should be secret.

Hands-on

- In the folder Hands-on\Tools

```
> openssl genrsa -out MyPrivKey.pem 2048
> openssl rsa -in MyPrivKey.pem -pubout -out MyPubKey.pem
> openssl rsa -in  MyPrivKey.pem -text
> openssl rsa -in  MyPubKey.pem -text -pubin
```

Hands-on

- In the folder Hands-on\Tools

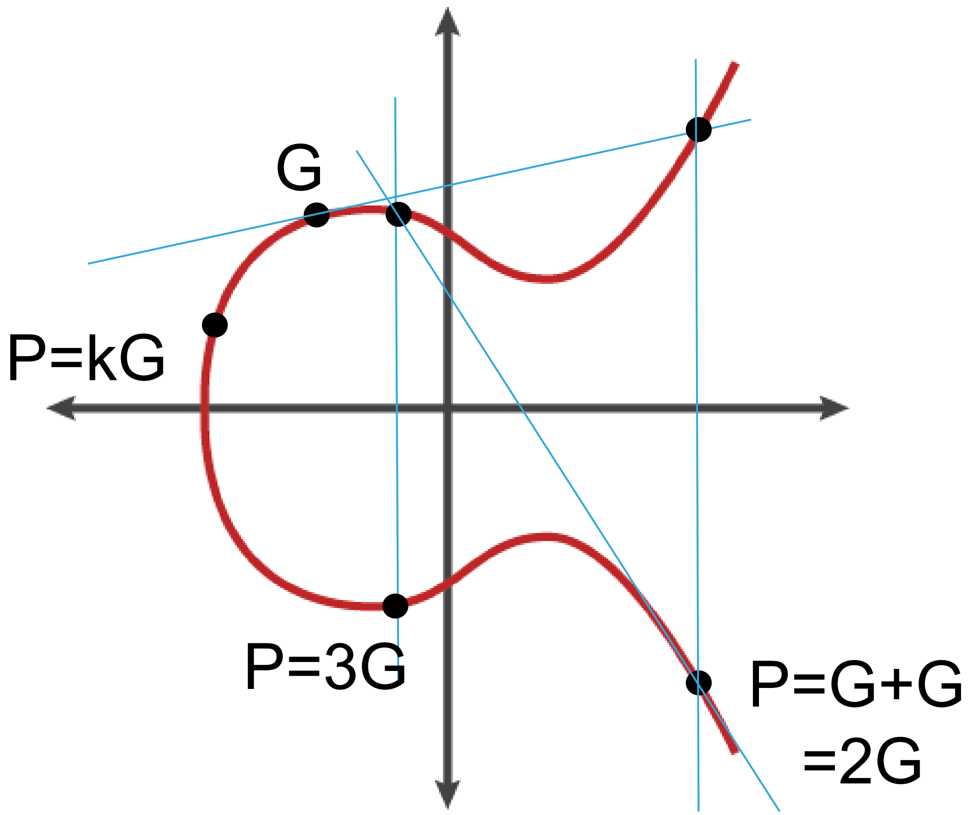
```
> openssl rsautl -encrypt -inkey MyPubKey.pem -pubin -in Example_AA_BB.bin -out Example_AA_BB_rsa.enc

> openssl rsautl -decrypt -inkey MyPrivKey.pem -in Example_AA_BB_rsa.enc -out Example_AA_BB_clear_rsa.bin

> hexdump Example_AA_BB_rsa.enc

> hexdump Example_AA_BB_clear_rsa.bin
```

ECC

- ECC (Elliptic-curve cryptography)
  - Based on elliptic curves over finite fields
  - Key size from 160 up to 512 bits

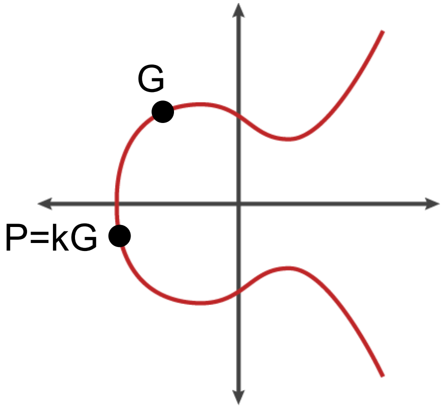
ECC

- Elliptic curve over finite field (order n): $y^2 = x^3 + ax + b \pmod p$



ECC

- Elliptic curve over finite field (order n): $y^2 = x^3 + ax + b \pmod{p}$
  - a and b are characteristic of the curve selected.
  - n is the number of different points on the curve which can be generated by multiplying a scalar with G
  - p is the modulus and is a prime number



- ECC key pair :
  - G and P : two points on the curve linked by the properties P=kG Those could be considered as the components of public key
  - k will be the private key

Diffie-Hellman

- Purpose : create a shared secret between Bob and Alice without communicate private information

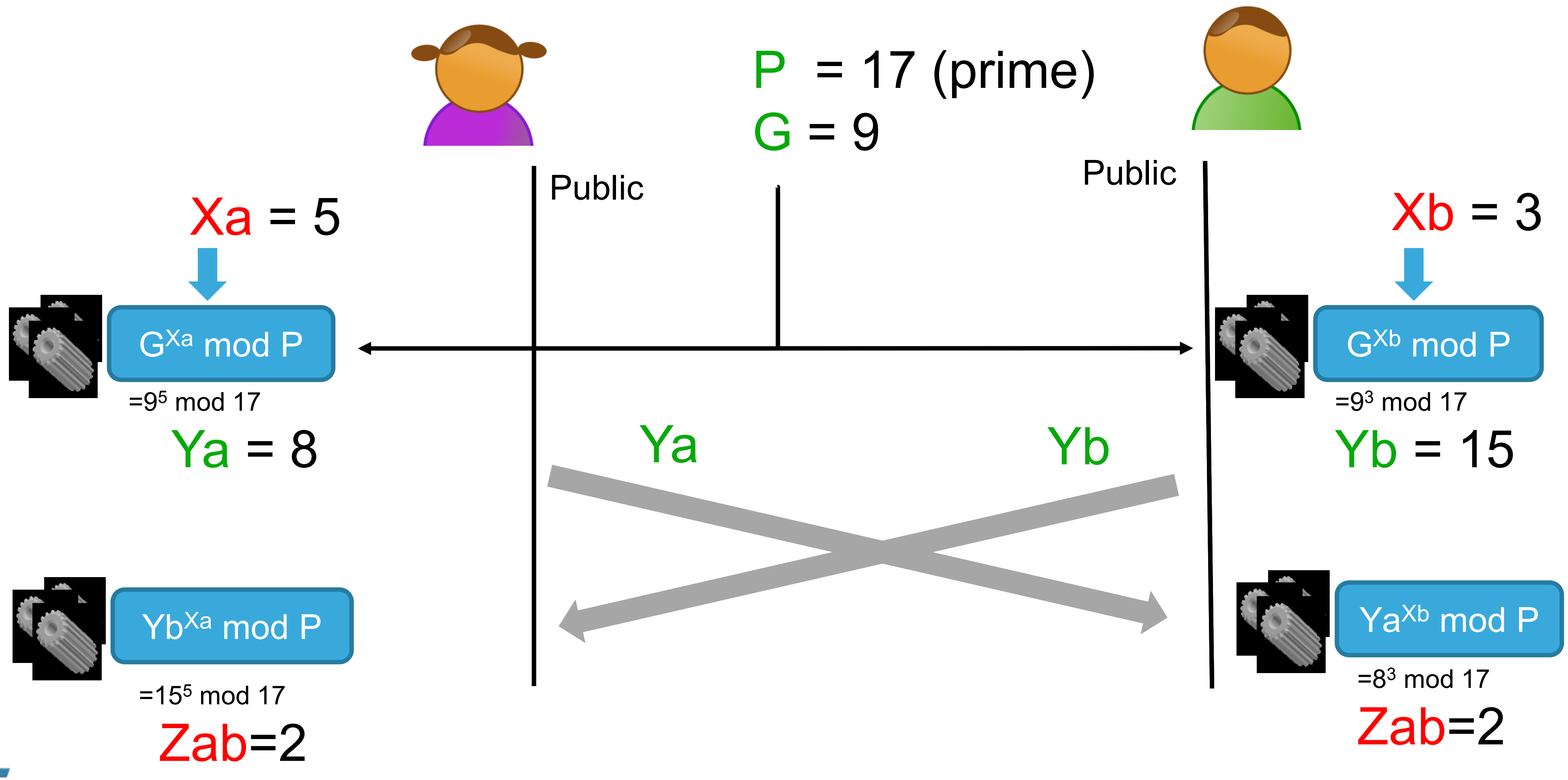
- It relies on the discrete logarithm problem:

  $Y = G^{x} \bmod P$ (with P prime number)

  - For example :
    - $8 = 9^{x} \bmod 17$

Create a shared secret

- Diffie-Hellman :



$P = 17$ (prime)

$G = 9$

Public

Public

$Xa = 5$

$G^{Xa} \bmod P$

$= 9^5 \bmod 17$

$Ya = 8$

$Xb = 3$

$G^{Xb} \bmod P$

$= 9^3 \bmod 17$

$Yb = 15$

Ya

Yb

$Yb^{Xa} \bmod P$

$= 15^5 \bmod 17$

$Zab = 2$

$Ya^{Xb} \bmod P$

$= 8^3 \bmod 17$

$Zab = 2$

Diffie-Hellman

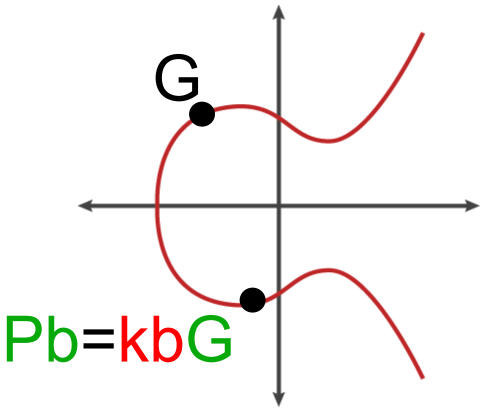
- Mathematically with discrete logarithm problem
  - Alice compute $Ya= G^{xa}$ mod P and send it to Bob
  - Bob compute Zab = Ya^{xb} mod P = $(G^{xa}$ mod $P)^{xb}$ mod P = G^{xa*xb} mod P
  - Bob compute $Yb = G^{xb}$ mod P and send it to Alice
  - Alice compute Zab = Yb^{xa} mod P= $(G^{xb}$ mod $P)^{xa}$ mod P = G^{xb*xa} mod P

Elliptic-curve Diffie–Hellman

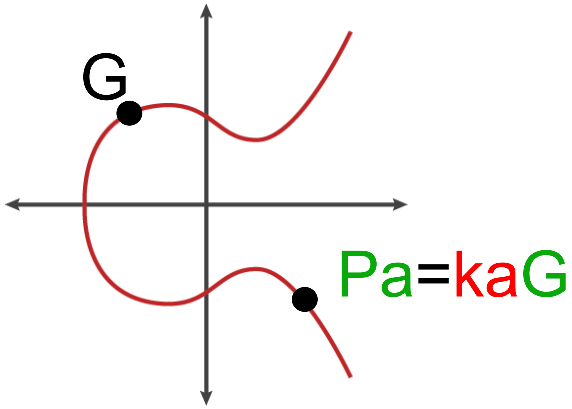
- We can also use the elliptic curve associated with diffie-Hellman

ECDH : Elliptic-curve Diffie–Hellman

- Bob compute Pb = kbG and send it to Alice

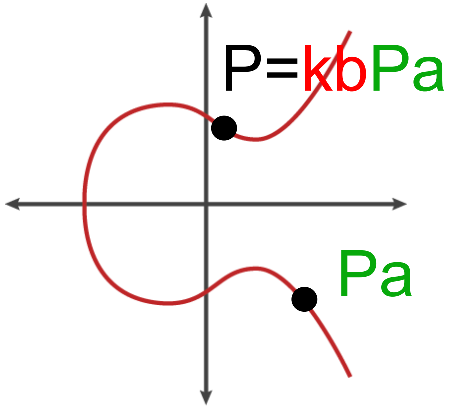


- Alice compute Pa= kaG and send it to Bob

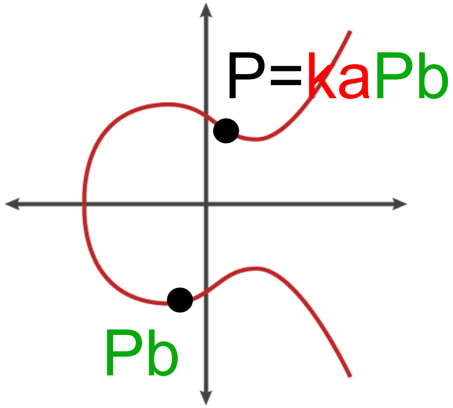


Elliptic-curve Diffie–Hellman

- Bob compute P = kbPa = kb(kaG)



- Alice compute P = kaPb = ka(kbG)



Elliptic-curve Diffie–Hellman

- ECDH : Elliptic-curve Diffie–Hellman
  - Bob compute Pb = kbG and send it to Alice
  - Alice compute Pa= kaG and send it to Bob
  - Bob compute P = kbPa = kb(kaG)
  - Alice compute P = kaPb = ka(kbG)

Hands-on

- In the folder Hands-on\Tools

```
> openssl genpkey -genparam -algorithm DH -out dhp.pem
> openssl pkeyparam -in dhp.pem -text
> openssl genpkey -paramfile dhp.pem -out dhkey_Alice.pem
> openssl pkey -in dhkey_Alice.pem -text -noout
> openssl pkey -in dhkey_Alice.pem -pubout -out dhpub_Alice.pem
> openssl pkey -pubin -in dhpub_Alice.pem -text
```

Hands-on

- In the folder Hands-on\Tools

```
> openssl genpkey -paramfile dhp.pem -out dhkey_Bob.pem
> openssl pkey -in dhkey_Bob.pem -text -noout
> openssl pkey -in dhkey_Bob.pem -pubout -out dhpub_Bob.pem
> openssl pkey -pubin -in dhpub_Bob.pem -text

> openssl pkeyutl -derive -inkey dhkey_Alice.pem -peerkey dhpub_Bob.pem -out secret1.bin
>openssl pkeyutl -derive -inkey dhkey_Bob.pem -peerkey dhpub_Alice.pem -out secret2.bin
```

Agenda

- Encryption theory:
  - Symmetric encryption/decryption theory
  - Asymmetric encryption/decryption theory
  - Combination of Symmetric/Asymmetric
  - Shared secret generation
- Main algorithm
  - Symmetric : TDES, AES
  - Asymmetric : RSA, Elliptic curves, Diffie-Hellman
  - Combination of Symmetric/Asymmetric: IES

Diffie-Hellman and encryption

Integrated encryption scheme (IES) are standardized :

- Discrete Logarithm Integrated Encryption Scheme (DLIES)
- Elliptic Curve Integrated Encryption Scheme (ECIES)

Where we are now ?

- We have tool to encrypt our message
  - We use asymmetric cryptography (RSA/ECC/Diffie-Hellman) to share or generate a common secret between Bob and Alice
  - We encrypt the data thanks to this common secret and the symmetric cryptography algorithm (TDES/AES)
- 2 remaining points:
  - Integrity check : insure nobody has modified the information
  - Authentication : insure people involve in the exchange are the real one

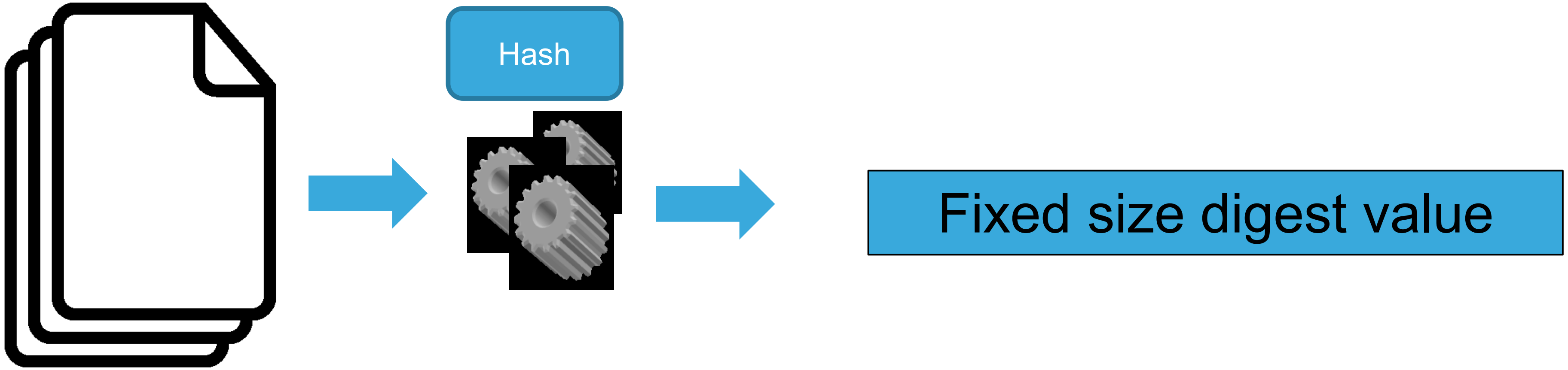
Data integrity verification

Agenda

- Hash function
- Integrity and Security
- Message authentication
  - With symmetric cryptography (HMAC / AES GCM)
  - With asymmetric cryptography (Signature RSA/ECC)

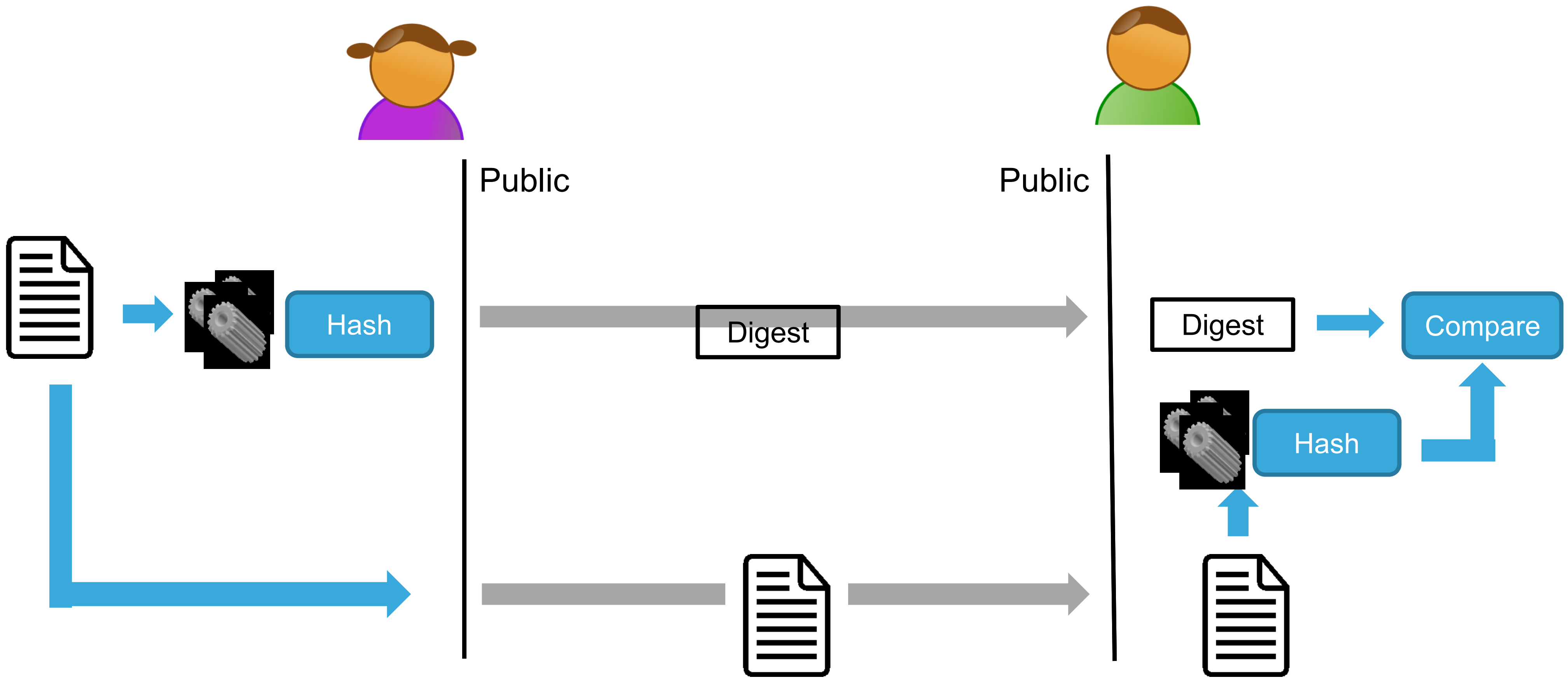
Hash function

- Purpose : generate a fixed size value based on an unknown size input data



- Ideal properties :
  - Any modification in the input data change the digest value generated
  - Impossible to find input values based on the digest value
  - Different input can’t generate the same digest value

Integrity check



Hash algorithm

- MD5 (Message Digest 5)
  - output size 128 bits, status : broken

- SHA1 (Secure Hash Algorithm 1)
  - output size 160 bits, status : broken

- SHA2 (Secure Hash Algorithm 2)
  - output size 224/256/384/512 bits, status : considered secure started from output size of 256

- SHA3 (Secure Hash Algorithm 3)
  - output size 224/256/384/512 bits, status : New US NIST standard (2015)

Hands-on

- In the folder Hands-on\Tools

```
> openssl dgst -md5  Example_AA_BB.bin
> openssl dgst -md5  Example_AA_BB_1bit_modified.bin
> openssl dgst -sha1  Example_AA_BB.bin
> openssl dgst -sha1  Example_AA_BB_1bit_modified.bin
> openssl dgst -sha256  Example_AA_BB.bin
> openssl dgst -sha256  Example_AA_BB_1bit_modified.bin
> openssl dgst –sha512  Example_AA_BB.bin
> openssl dgst –sha512  Example_AA_BB_1bit_modified.bin
```

Integrity versus security

- Hash as message integrity isn’t very useful from a security standpoint

- Attacker on communication channel can:
  - Alter the message
  - Recompute the digest on the altered message
  - Replace the original message digest with its own digest

- Solution: combination with encryption. This is the message authentication

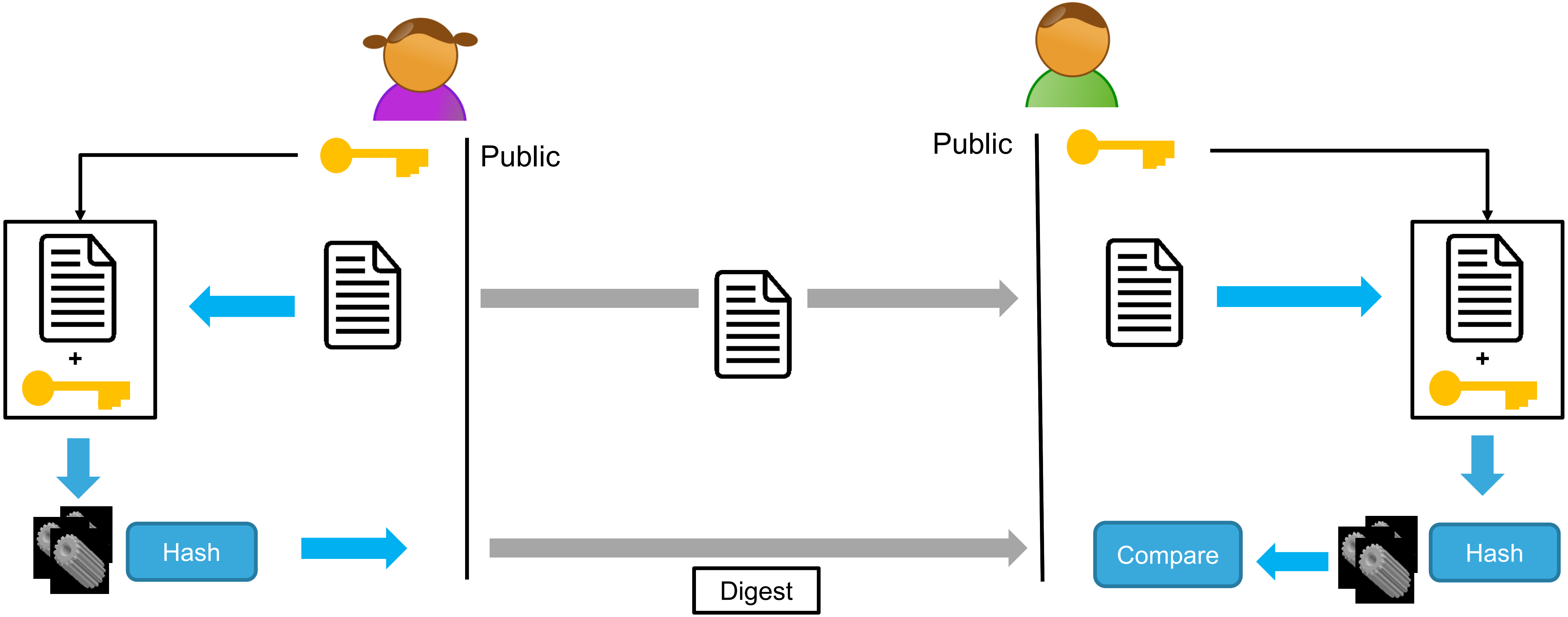
Message authentication

- Combine Hash function and :
  - Symmetric cryptography (HMAC / AES GCM)
  - Asymmetric cryptography (signature RSA/ECC)

- Purpose : Insure an attacker can’t alter the data/digest without been detected.

With symmetric cryptography

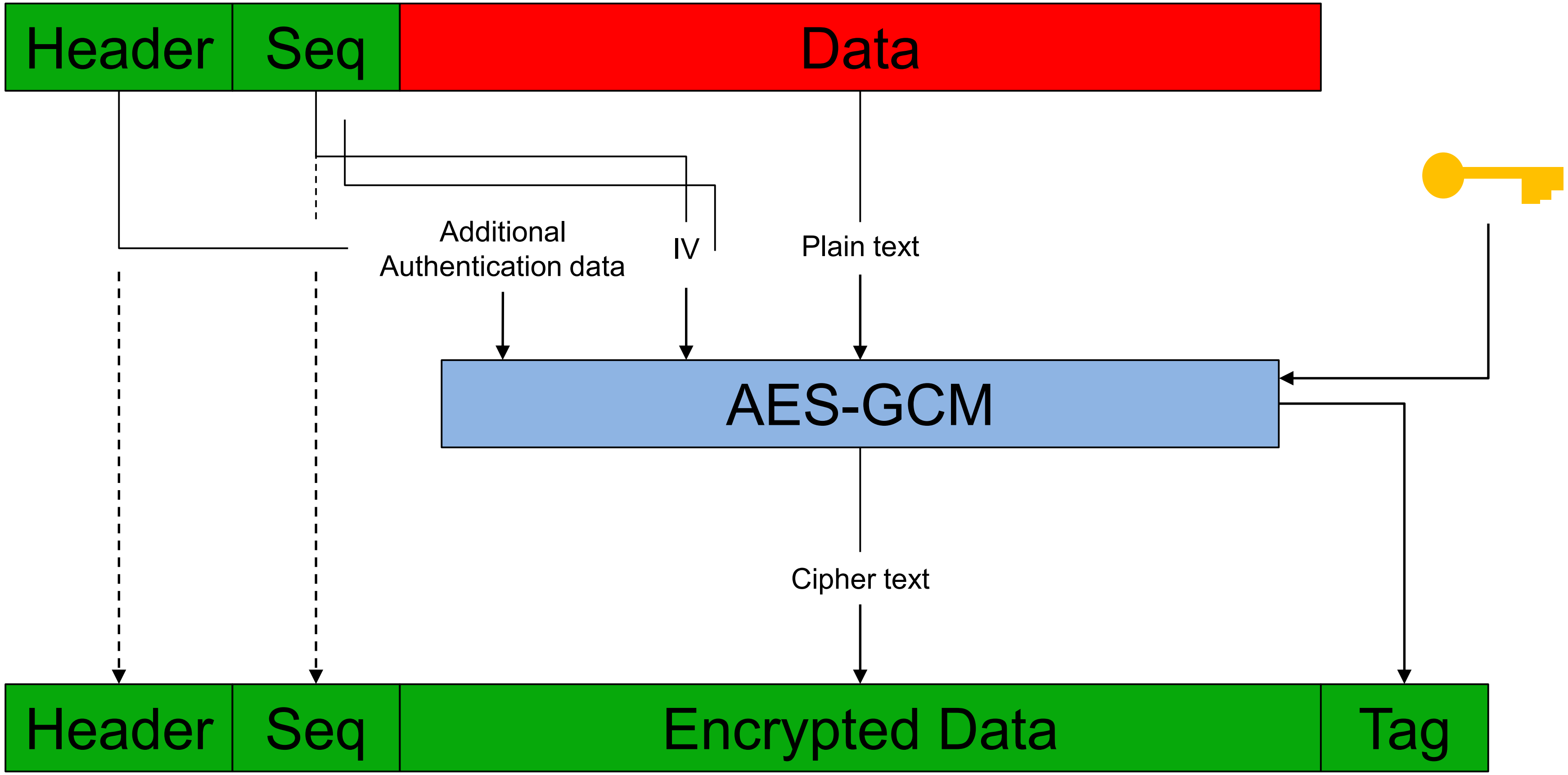
- HMAC : use common secret to insure integrity



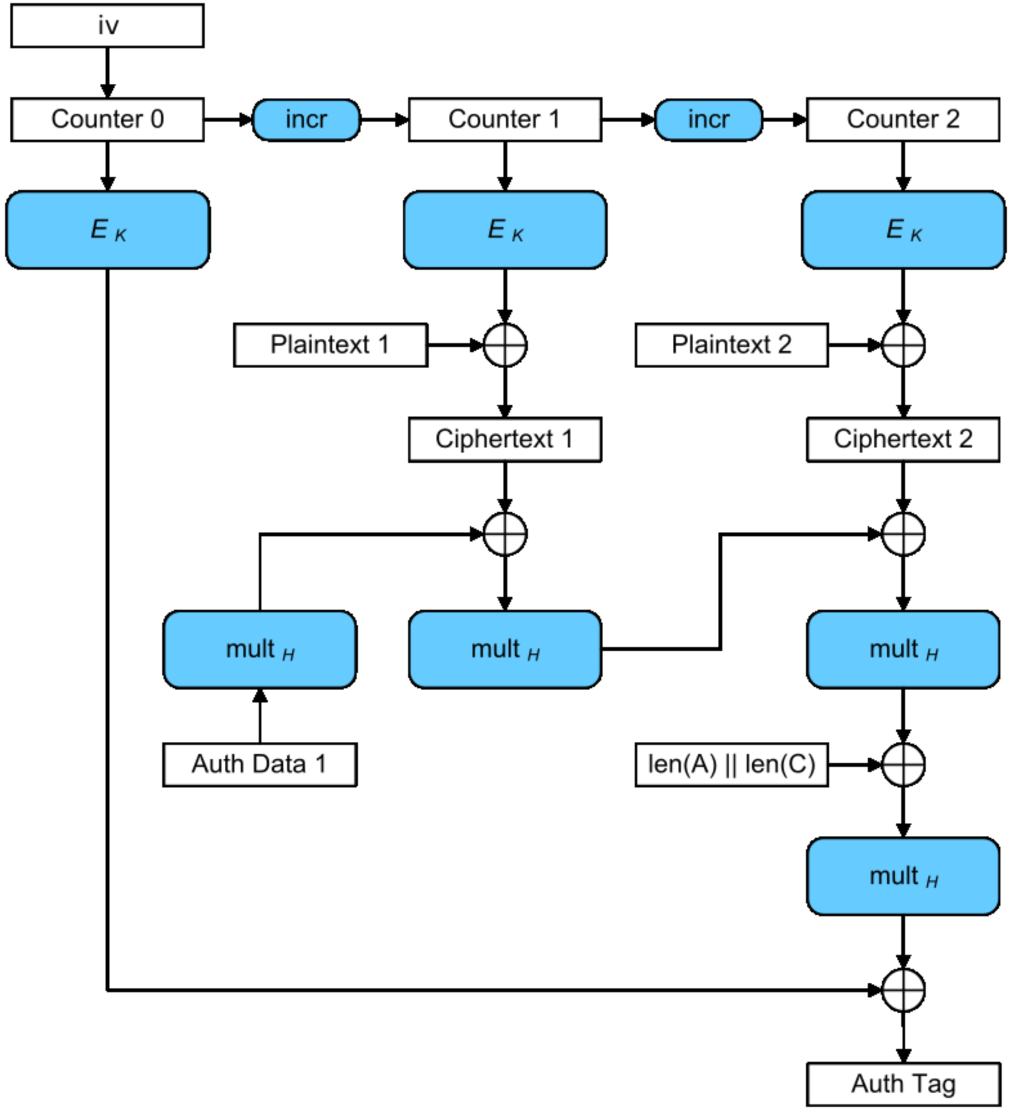
With symmetric cryptography

- AES GCM : Galois Counter Mode
  - Combination of AES counter mode and a specific hash function (which rely on galois field multiplication)
  - Encrypt the data to insure confidentiality
  - Generate a Tag which allow to insure message authentication

AES GCM


Header
Seq
Data
Additional Authentication data
IV
Plain text
AES-GCM
Cipher text
Header
Seq
Encrypted Data
Tag

AES GCM

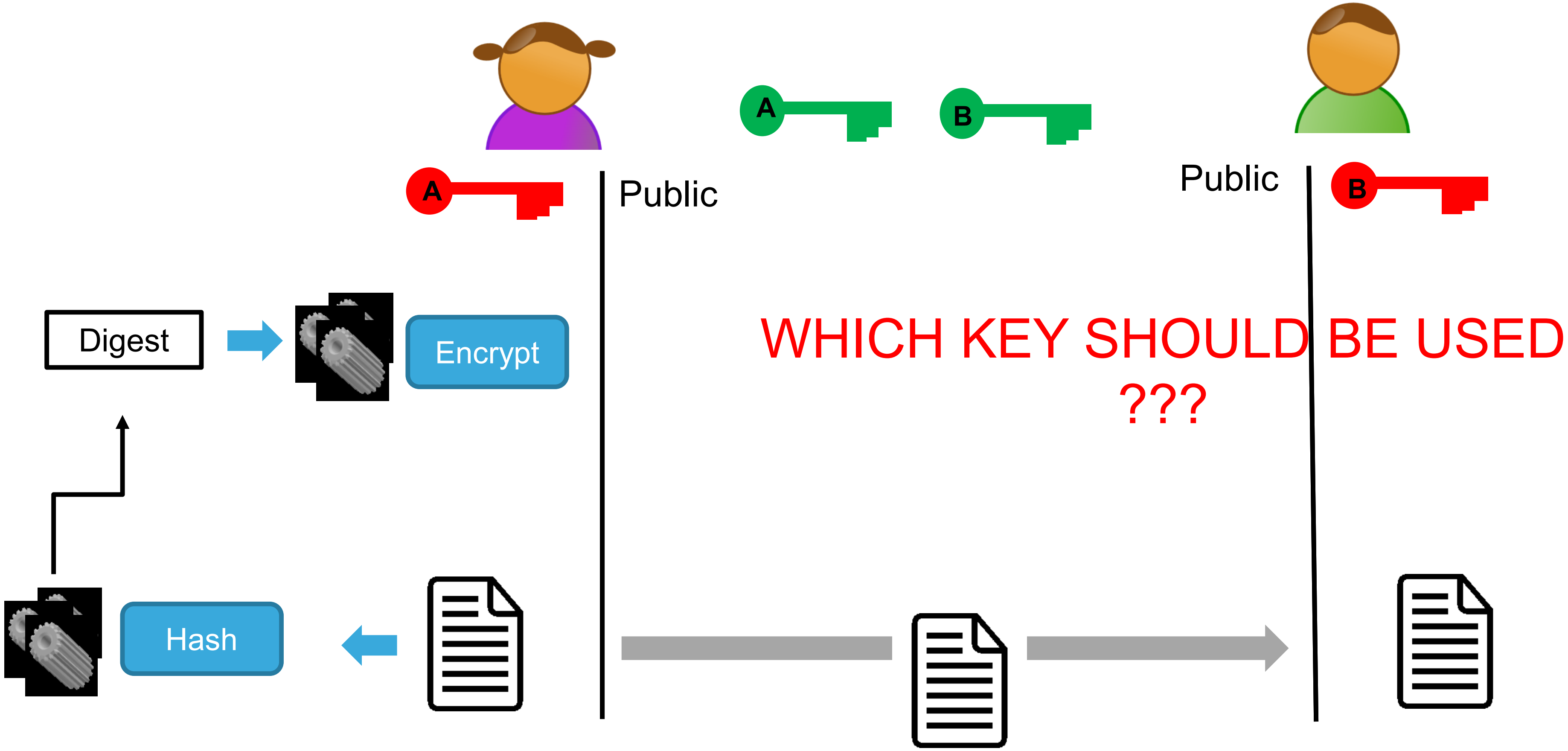

iv
Counter 0
incr
Counter 1
incr
Counter 2
E_K
E_K
E_K
Plaintext 1
Plaintext 2
Ciphertext 1
Ciphertext 2
mult_H
mult_H
mult_H
Auth Data 1
len(A) || len(C)
mult_H
Auth Tag

Agenda

- Hash function
- Integrity and Security
- Message authentication
  - With symmetric cryptography (HMAC / AES GCM)
  - With asymmetric cryptography (Signature RSA/ECC)

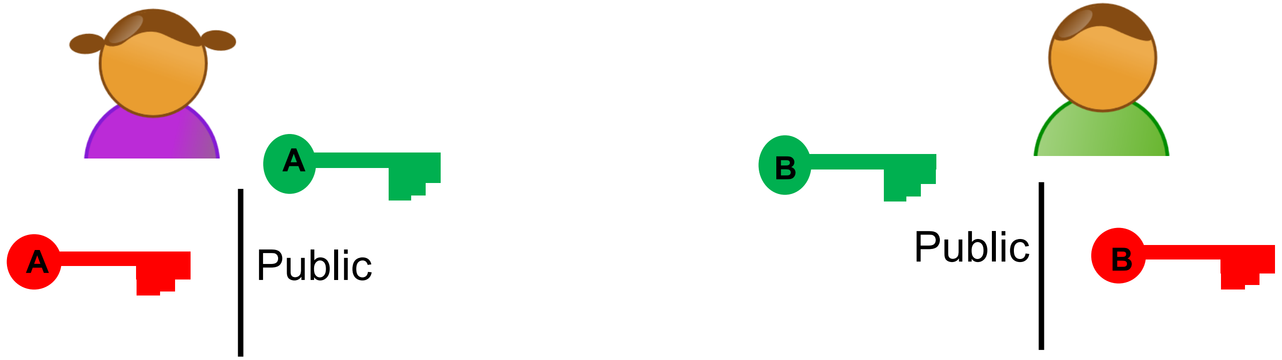
With Asymmetric cryptography

- Signature : encryption of the digest thanks asymmetric cryptography



WHICH KEY SHOULD BE USED ???

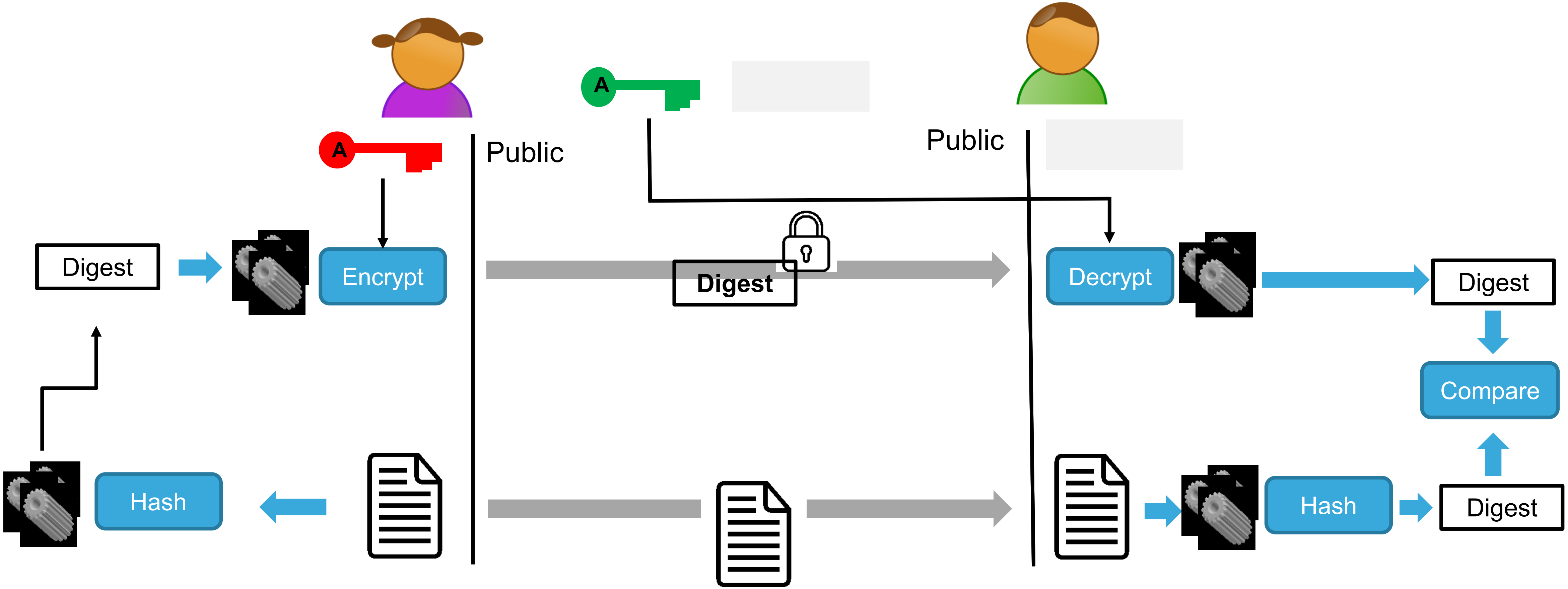
Signature process



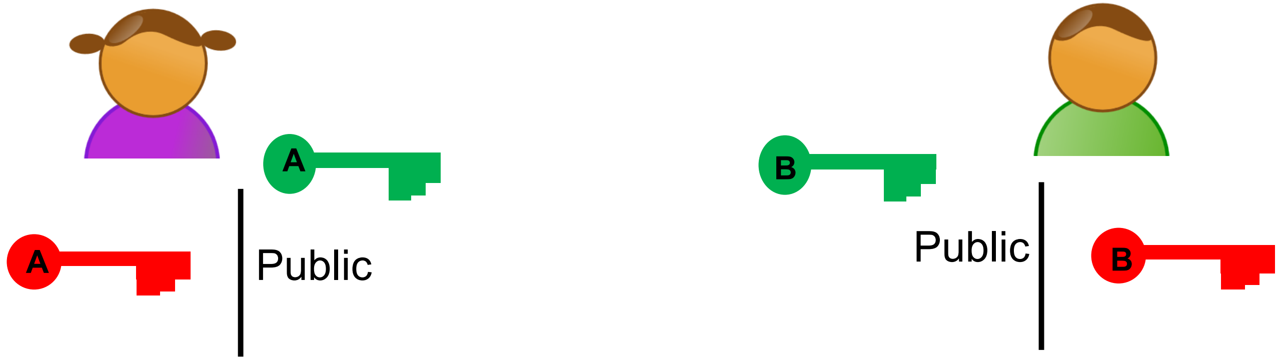
- Alice want to encrypt a hash to generate a signature…Which key should she use for encryption ?
  - Alice public key 🟢A ….Only Alice could decrypt it with 🔴A
  - Bob public key 🟢B …..Only Bob could decrypt it with 🔴B
    But as it's a public key, any body could create a new signature …
  - Alice private key 🔴A ….So everybody could decrypt this signature with 🟢A

Asymmetric cryptography signature

- Signature process :use asymmetric to insure integrity



To keep in mind…



- Only the owner of the key pair could create a signature … Signature is generated thanks private key.

- Signature could be check by any body thanks the public key of the key pair owner !

Hands-on

- In the folder Hands-on\Tools

```
> echo abcdefghijklmnopqrstuvwxyz > myfile.txt
> openssl dgst -sha256 -sign MyPrivKey.pem -out signature.bin myfile.txt
> openssl dgst -sha256 -verify MyPubKey.pem -signature signature.bin myfile.txt
> echo aacdefghijklmnopqrstuvwxyz > myfile2.txt
> openssl dgst -sha256 -verify MyPubKey.pem -signature signature.bin myfile2.txt
```

Hands-on

- In the folder Hands-on\Tools

```
> openssl ecparam -list_curves
> openssl ecparam -param_enc explicit -conv_form uncompressed -text -noout -no_seed -name secp384r1
> openssl ecparam -name secp384r1 -genkey -out MyECCKey.pem
> openssl ec -in MyECCKey.pem -pubout -out MyECCPubKey.pem
> openssl dgst -sha256 -sign MyECCKey.pem <  myfile.txt > signature.bin
> openssl dgst -sha256 -verify MyECCPubKey.pem -signature signature.bin < myfile.txt
```

Integrity

- Integrity check need to have message authentication mechanism

- Symmetric crypto (HMAC, AES GCM..)

- Asymmetric crypto : signature mechanism (RSA or ECC)
  - RSA Signature result from digest encryption
  - ECC Signature result from the ECDSA (Elliptic curve digital signature algorithm)
  - Signature generation is done thanks private key
  - Signature verification is done thanks public key

Where we are now ?

- We have tool to encrypt our message
  - We use asymmetric cryptography (RSA/ECC/Diffie-Helman) to generate a common secret between Bob and Alice
  - We encrypt the data thanks this common secret and the symmetric cryptography algorithm
- We know how to check message integrity/authentication
  - HMAC (symmetric crypto) / AES GCM
  - Signature (AES /ECC)
- What is missing ?
  - Authentication : insure people involve in the exchange are the real one

Authentication

Authentication

- Purpose : be sure Alice is talking with Bob and not someone else (and symmetrically)

Agenda

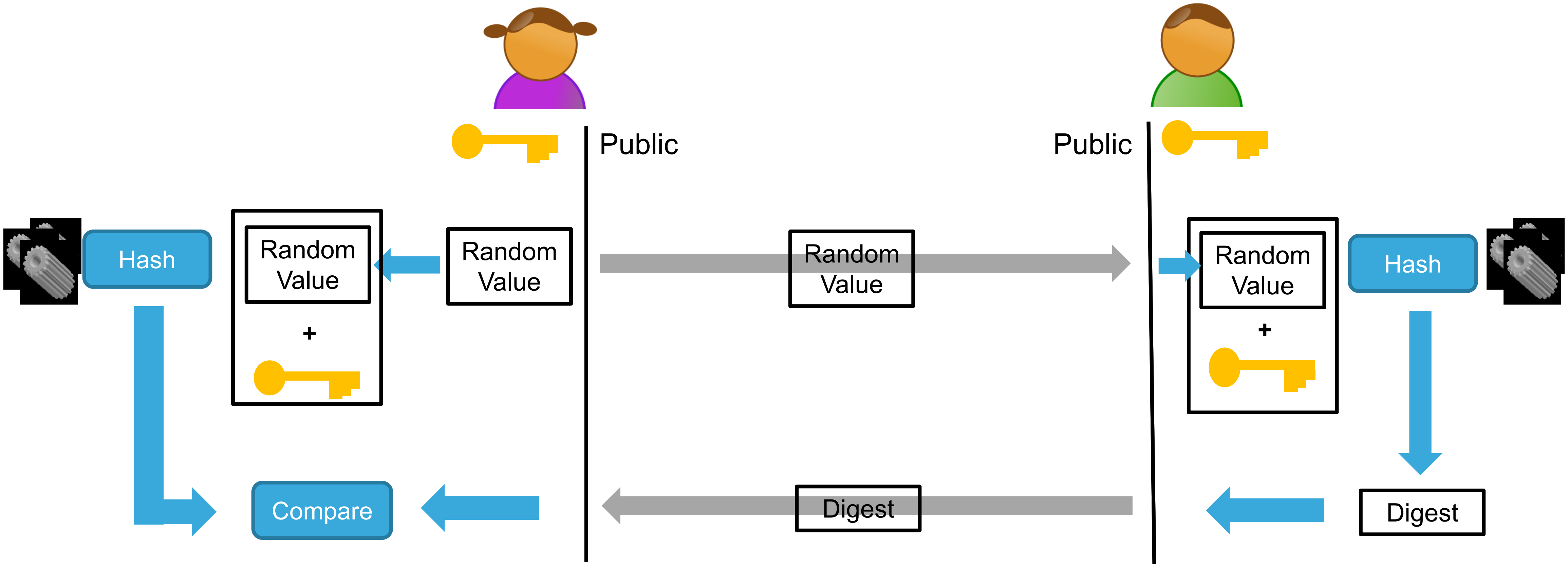
- Authentication
  - With symmetric cryptography (HMAC / AES)
  - With asymmetric cryptography (RSA/ECC)
- Risk of impersonation
- Certificate authority

Authentication

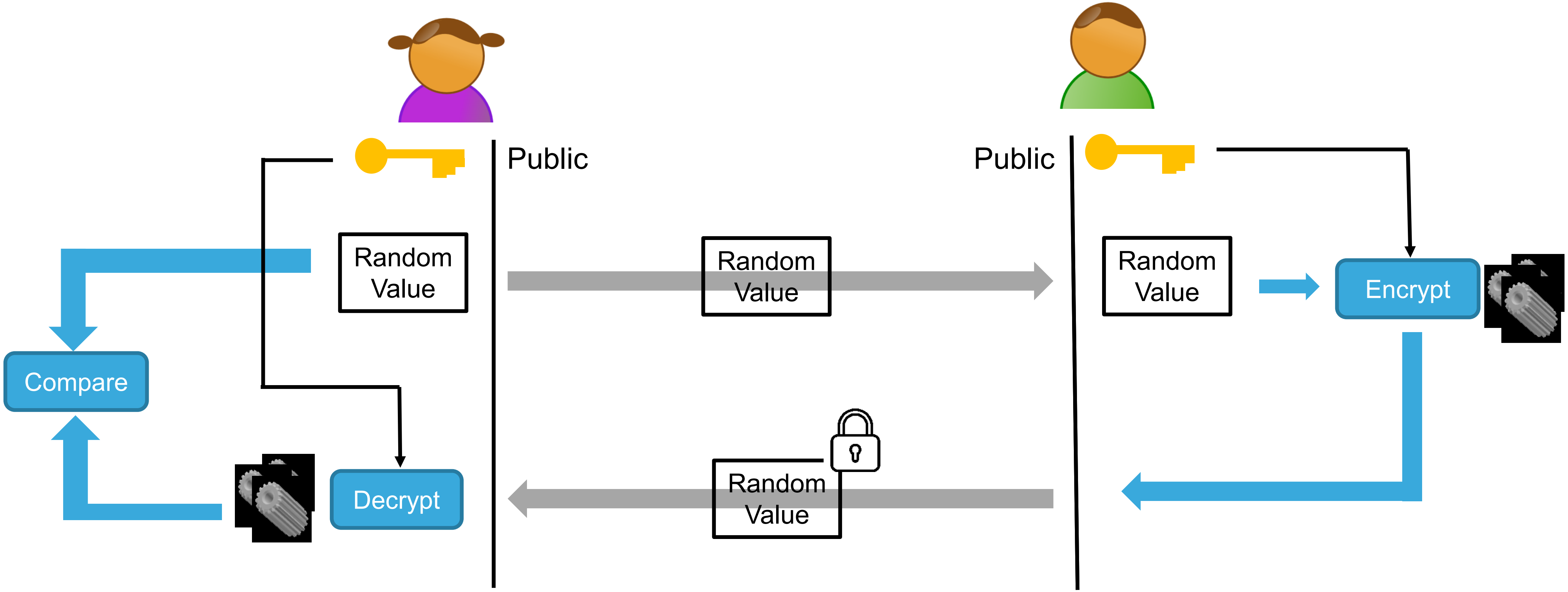
- It could rely on 2 mechanisms
  - Symmetric encryption
  - Asymmetric encryption

- Principle is always the same :
  - Alice choses a random value, also called challenge, and share it with Bob
  - Bob will return the result of the encryption this challenge.
  - Alice will check the result to insure about Bob identity

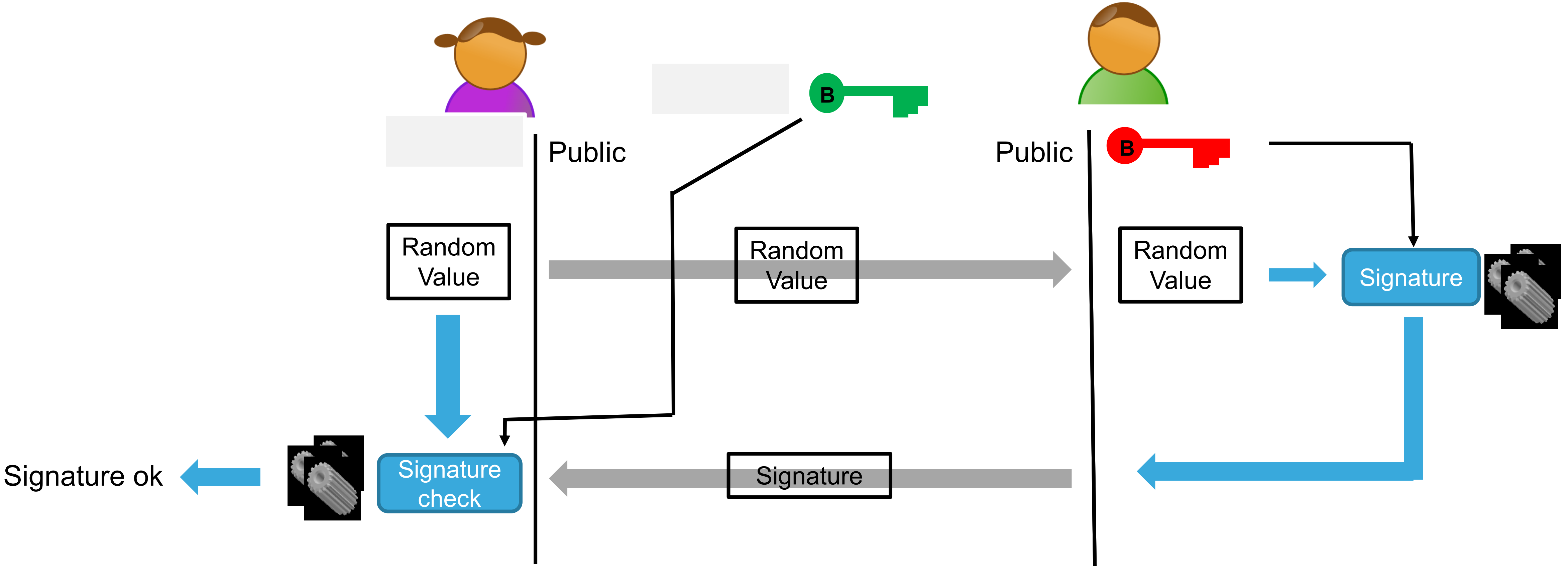
Authentication hash : HMAC



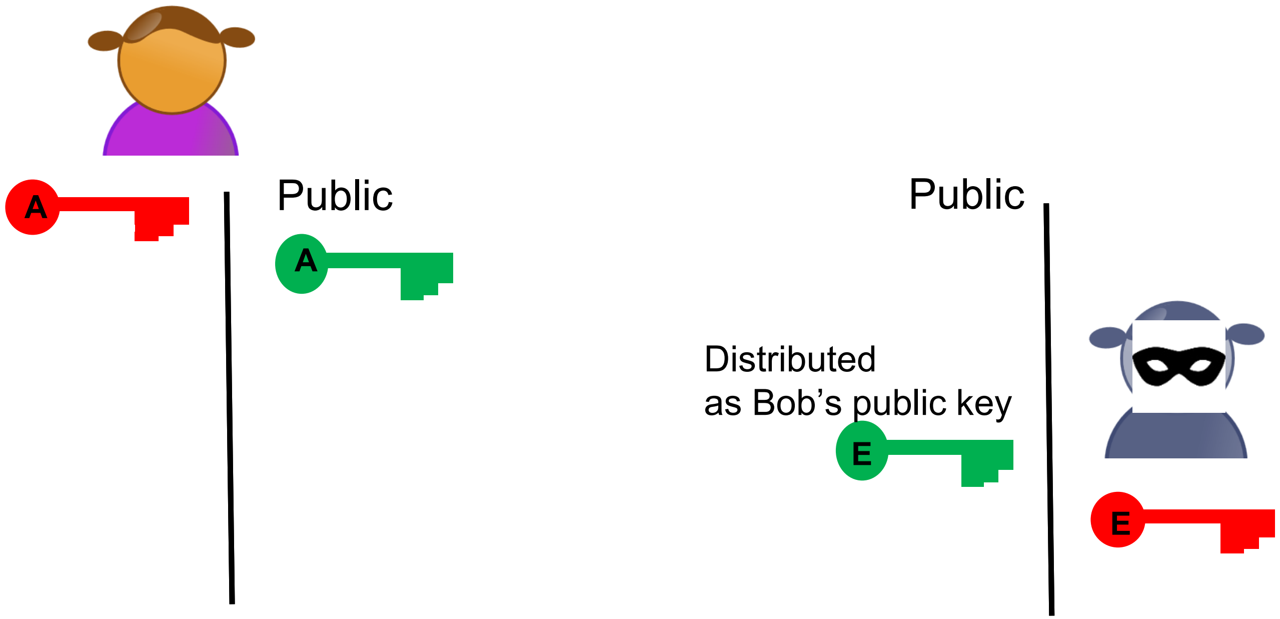
Authentication symmetric



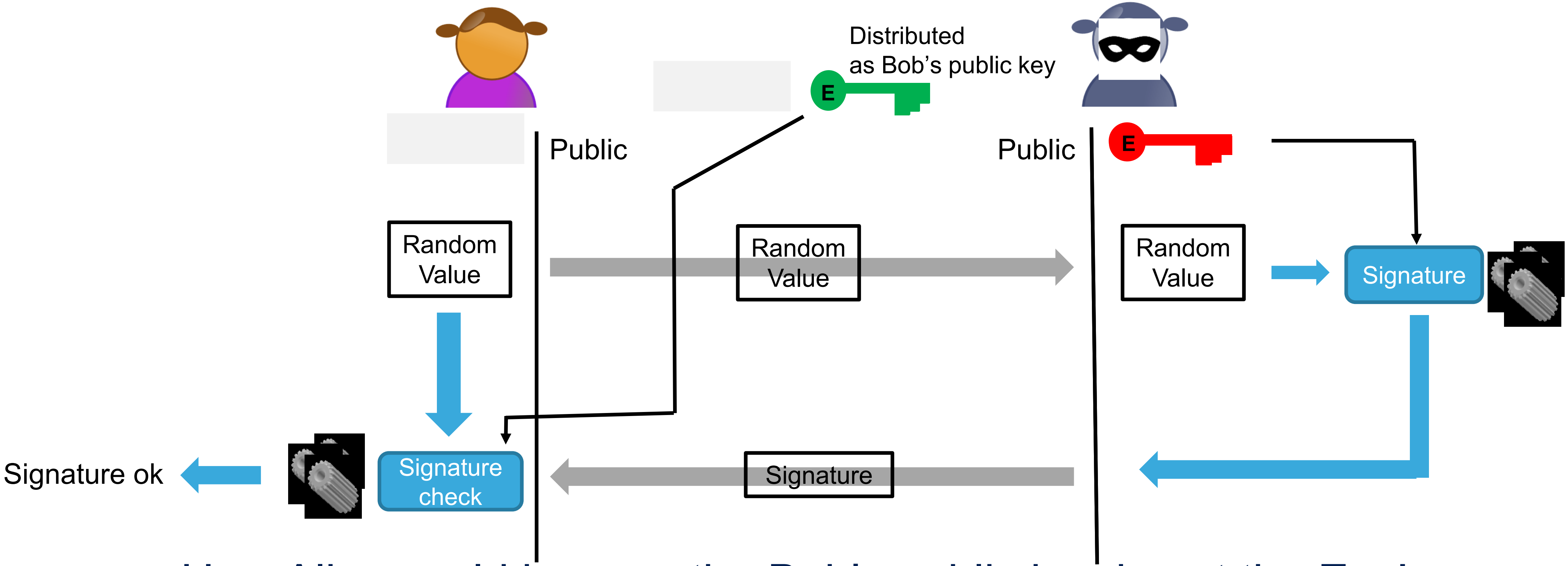
Authentication asymmetric



Risk of impersonation



Risk of impersonation



- How Alice could be sure the Bob’s public key is not the Eve's public key ?

Risk of impersonation

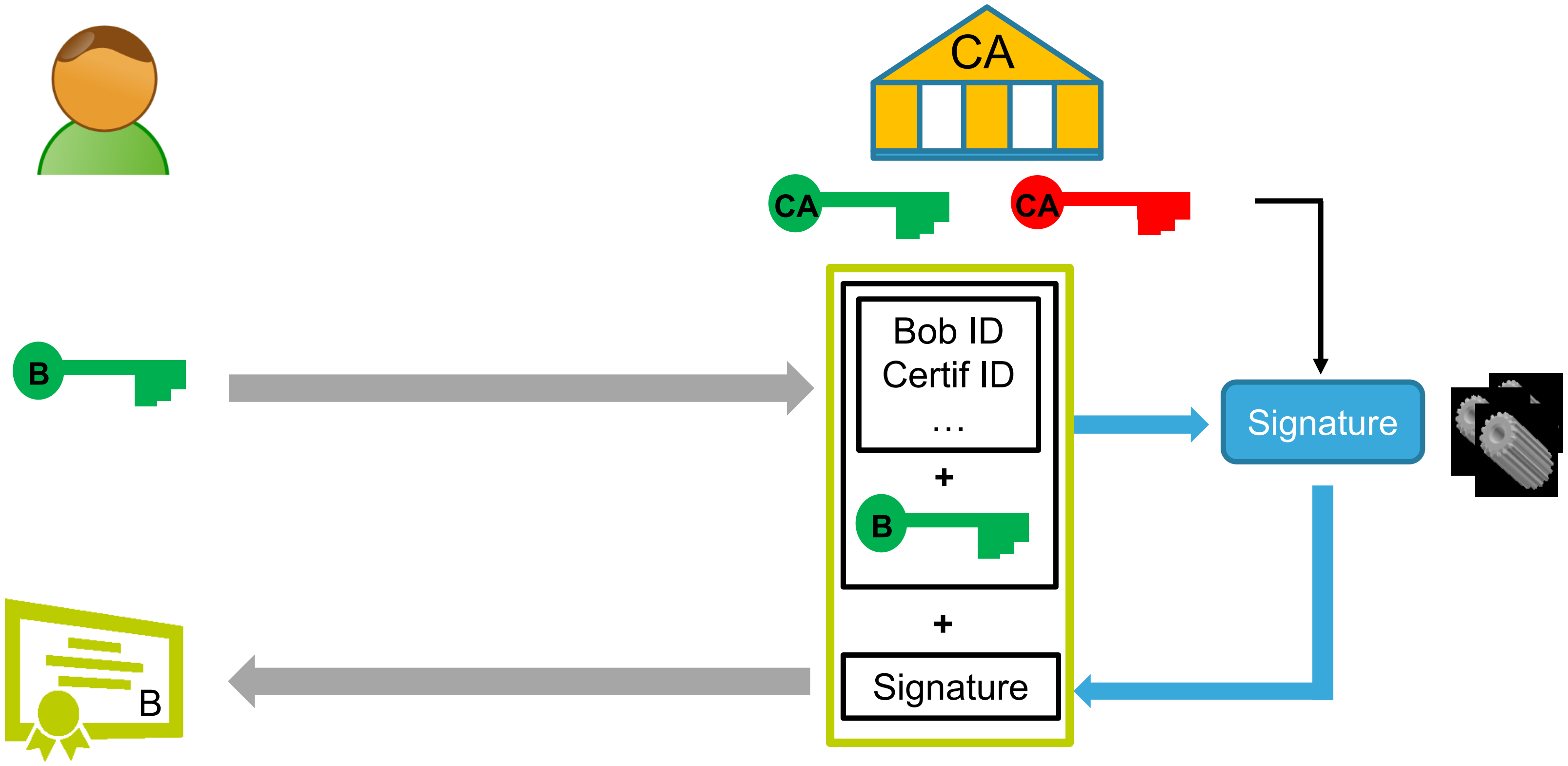
- The issue is how to trust Bob’s public key ?

Thank a certificate and certificate authority !

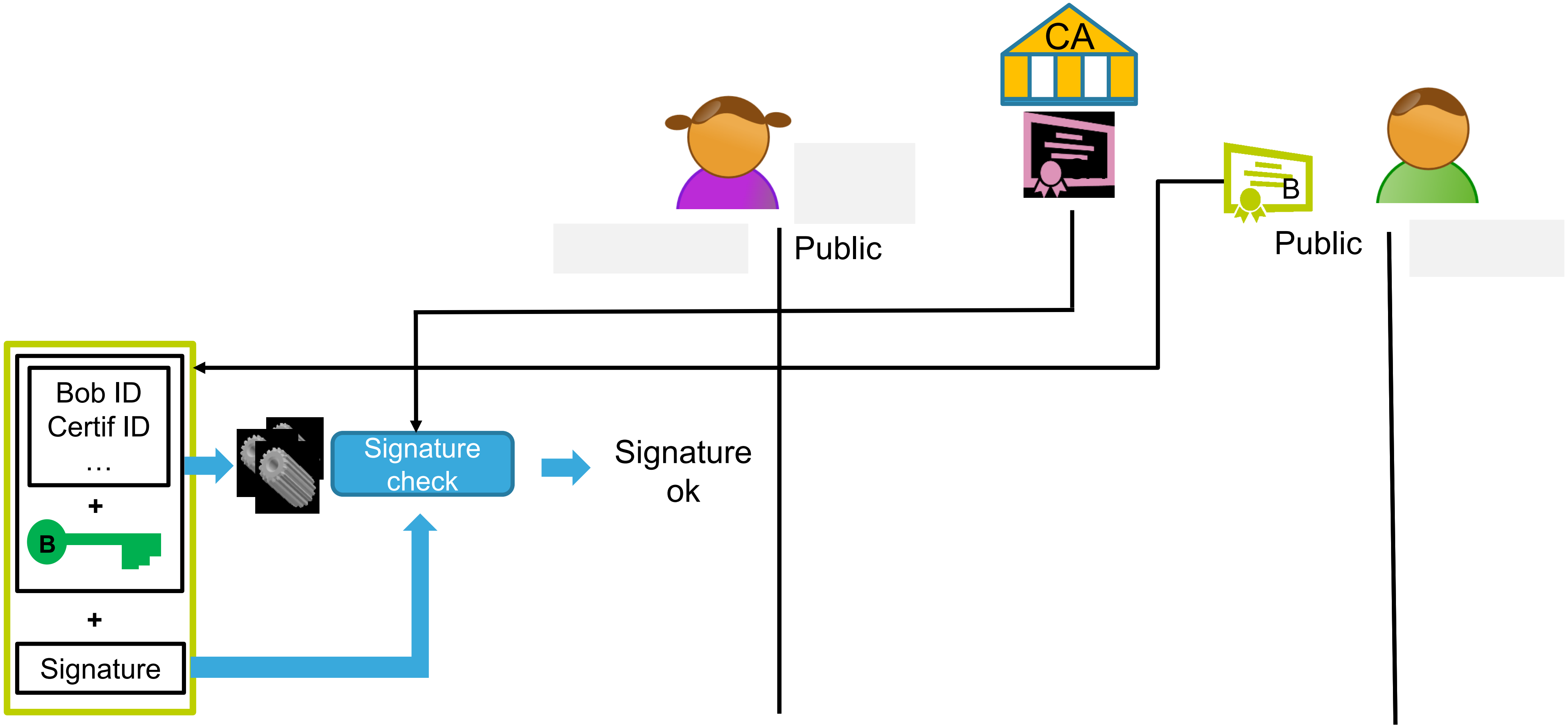
Certificate and CA

- Certificate is a digital data that includes at least :
  - Issuer identification
  - Certificate identification
  - Public Key
  - Signature of the previous data
- CA is a Certificate Authority
  - The CA is in charge of generating Certificate
- Most used Certificate format is X509 DER encoded structure
- The certificate is a way to exchange a public key in a trusted way

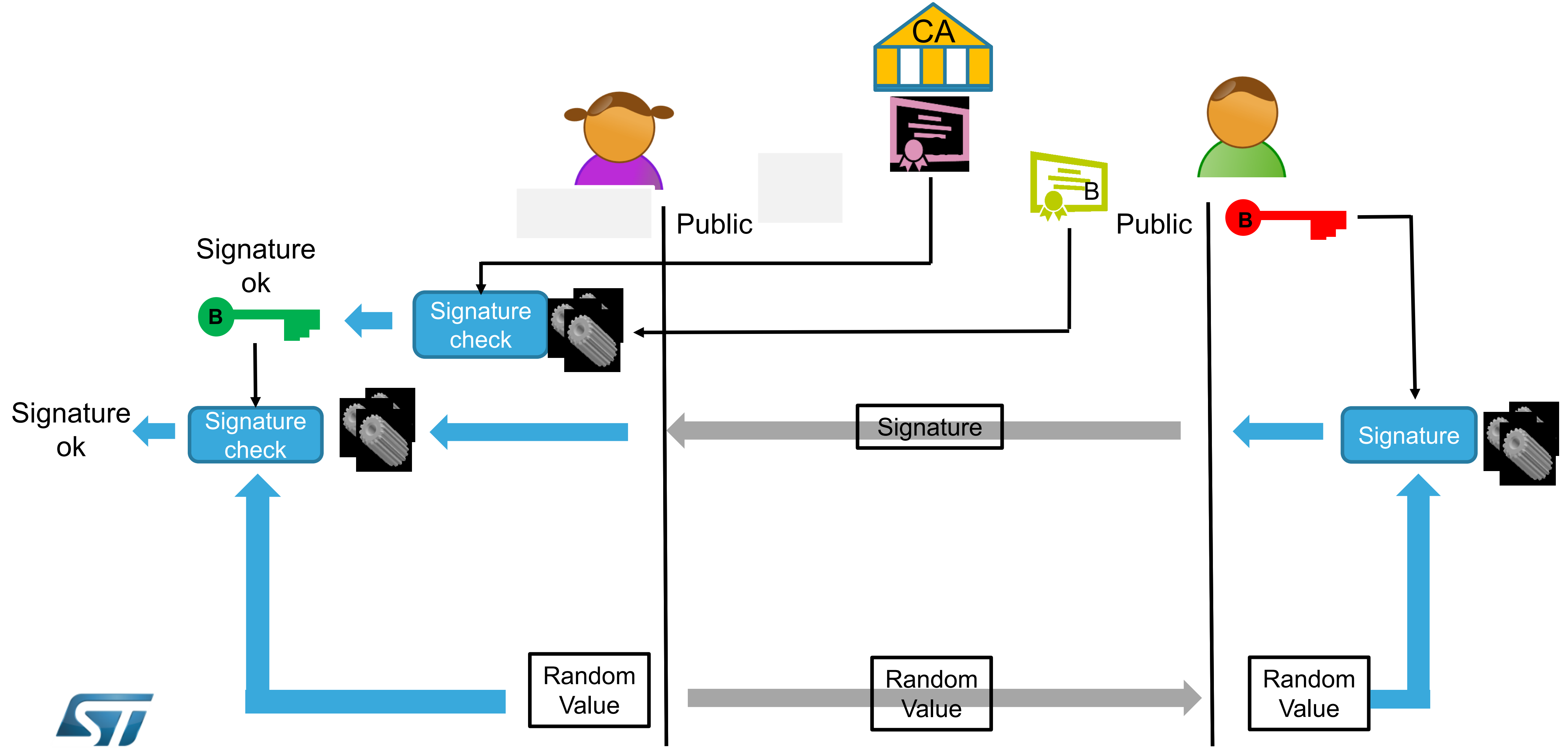
Bob will require his certificate



Check Bob certificate



Basic Authentication with CA



Authentication

- Authentication is done thanks a challenge and can use
  - Symmetric or asymmetric cryptography

- Using the asymmetric, there is a risk of impersonation

- Solution : using a certificate authority which deliver certificate to insure a proper authentication.
  When the both side authenticate each other, we are talking of mutual authentication

Everything is in place

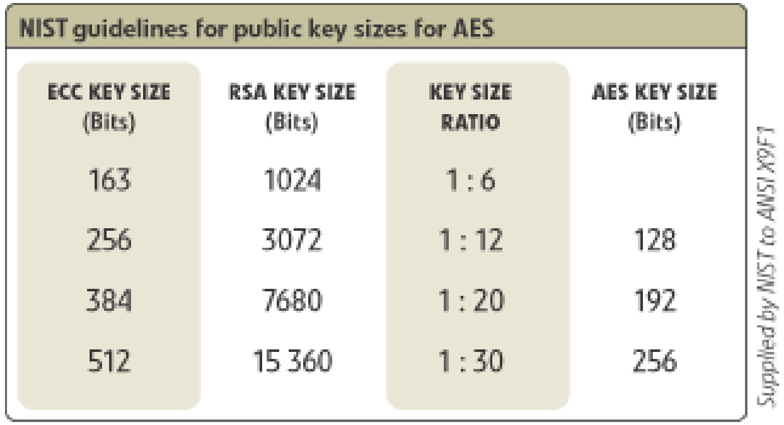
- Exchange confidential messages
- Ensure message integrity
- Authentication: Bob to be sure he exchanges with Alice and not with somebody else

Comparison of symmetric algorithm

- AES could be considered as the successor of TDES
- AES is faster than TDES with a lower memory footprint

Comparison of asymmetric algorithm

- ECC (same level of security)
  - ECC is faster in key generation and signature generation
  - ECC has smaller key size
  - ECC curve selection should be done carefully



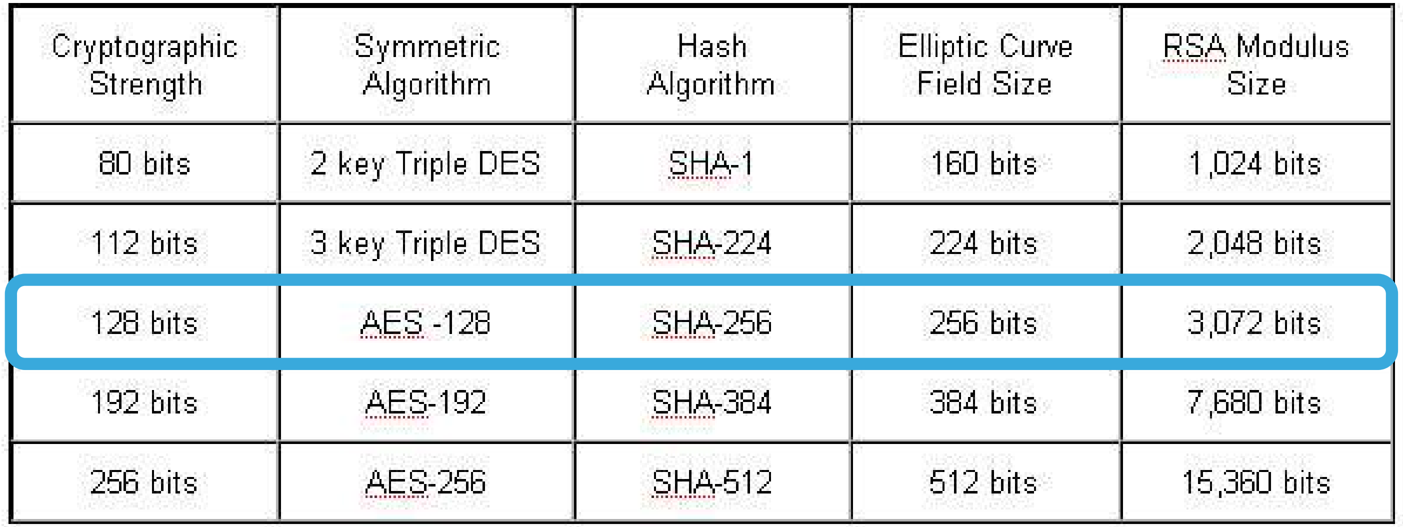
NIST guidelines for public key sizes for AES

ECC KEY SIZE (Bits)	RSA KEY SIZE (Bits)	KEY SIZE RATIO	AES KEY SIZE (Bits)
163	1024	1 : 6	
256	3072	1 : 12	128
384	7680	1 : 20	192
512	15 360	1 : 30	256

Supplied by NIST to ANSI X9F1

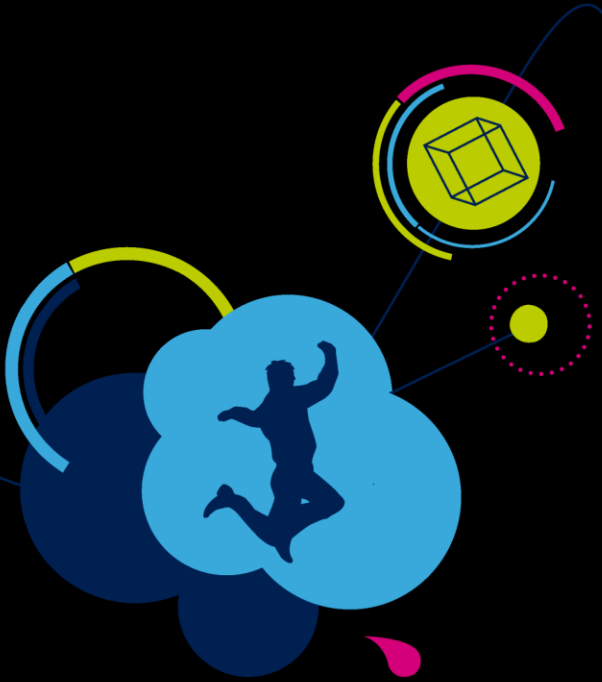
- RSA (same level of security)
  - RSA is faster signature verification
  - RSA has bigger key

Cryptographic levels and key lengths



Cryptographic Strength	Symmetric Algorithm	Hash Algorithm	Elliptic Curve Field Size	RSA Modulus Size
80 bits	2 key Triple DES	SHA-1	160 bits	1,024 bits
112 bits	3 key Triple DES	SHA-224	224 bits	2,048 bits
128 bits	AES -128	SHA-256	256 bits	3,072 bits
192 bits	AES-192	SHA-384	384 bits	7,680 bits
256 bits	AES-256	SHA-512	512 bits	15,360 bits

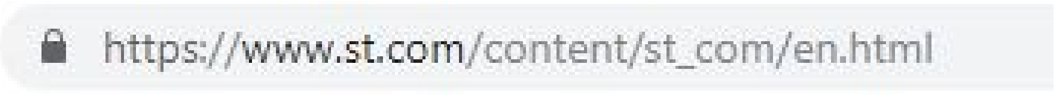
Recommended Key length for most applications



Example of crypto usage: TLS Management

What is TLS ?

- TLS : Transport Layer Security
  - It’s a cryptographic protocol
  - It allows authentication and encryption of data between servers, devices or web application
  - For example:



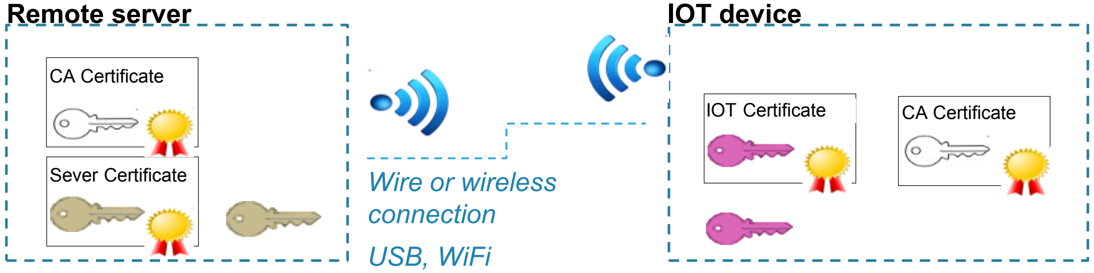
https://www.st.com/content/st_com/en.html

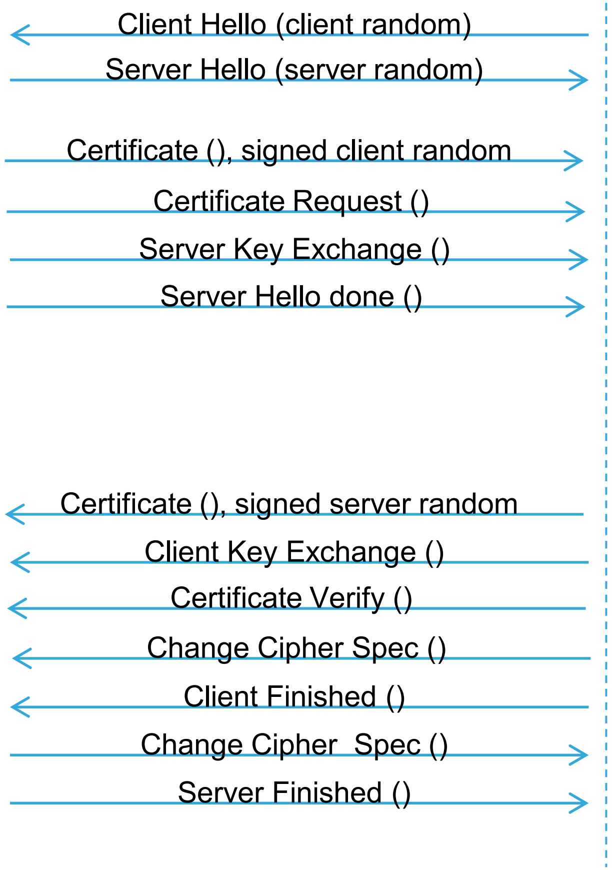
TLS Management

- TLS Handshake
  - Mutual authentication so on both side you need
    - CA certificate to authenticate remote entity
    - Device certificate
    - Device private key to generate challenge signature

  -> this is optional
  - Session secret negotiation
    - Can be based on
      - Pre Shared Keys (PSK)
      - Diffie-Hellman negotiation (with or without ephemeral keys)
- Preferred solution is to have the Diffie-Hellman negotiation algorithm

TLS Handshake V1.2 (RFC 5246)





Client Hello (client random) ←

Client provides supported TLS version, algorithms and a random

Replies algorithms choices

Server Hello (server random) →

Provides X509 certificate and signed random

Certificate (), signed client random →

Request IOT device X509 certificate

Certificate Request () →

Provides EC public key and the curve to use

Server Key Exchange () →

Server Hello done () →

Processing

- Verify Server certificates with CA certificate
- Authenticate server verifying signature
- Generate ephemeral EC key pairs
- Computes shared secret using Remote server public key

Verifies IOT Device certificate and authenticate IOT device

Certificate (), signed server random ←

Provides IOT device X509 certificate and signed random

Computes Diffie-Hellman shared secret

Client Key Exchange () ←

Provides ephemeral public key

Certificate Verify () ←

Change Cipher Spec () ←

Starts exchange ciphering

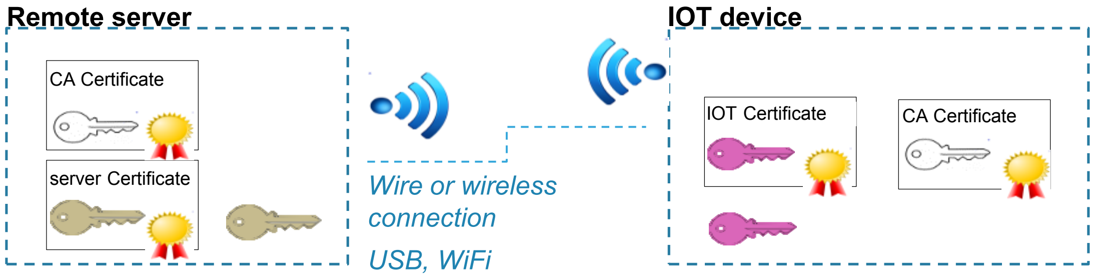
Client Finished () ←

Change Cipher Spec () →

Server Finished () →

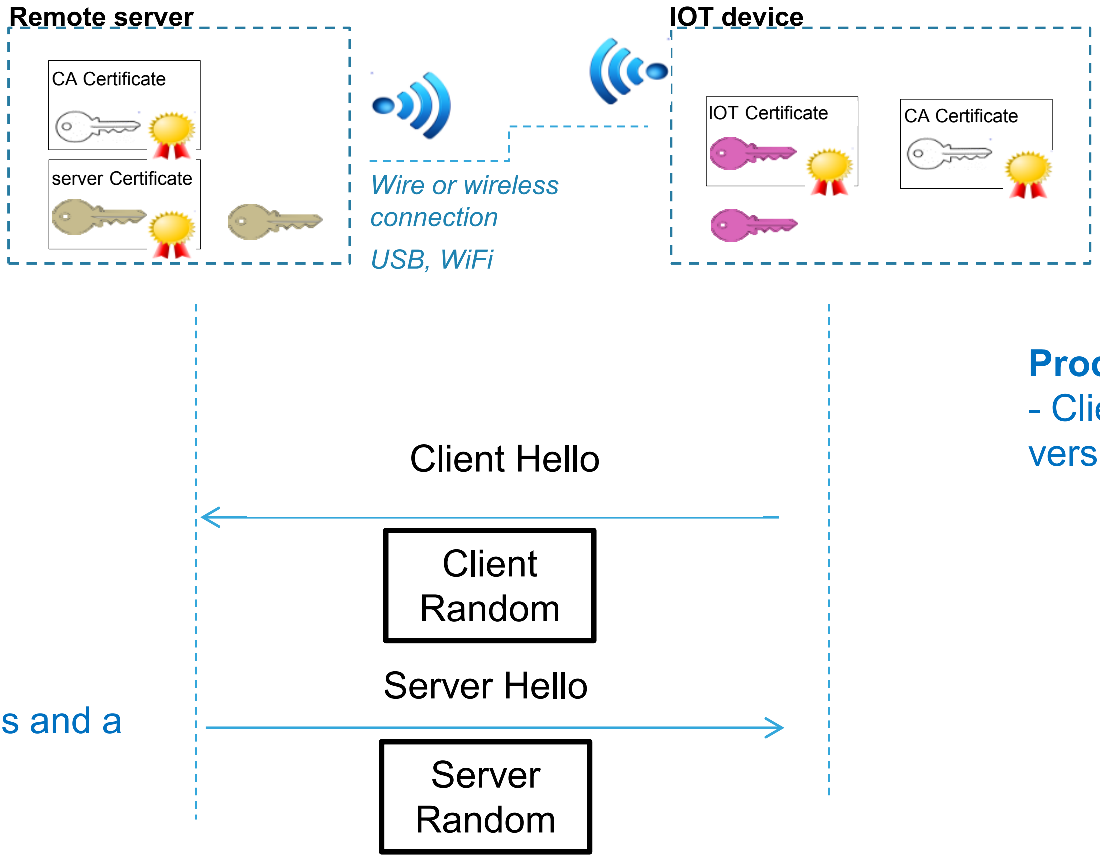
Starts exchange ciphering

TLS Handshake V1.2 (RFC 5246)



- CA certificate : public key of CA, which allow to check signature of an other public key
- Server/IOT certificate : public key of Server/IOT which allow to
  - encrypt something that can only be decrypted by the private key
  - check a signature done by the private key owner
- Server/IOT private key which allow to generate signature

TLS Handshake V1.2 (RFC 5246)



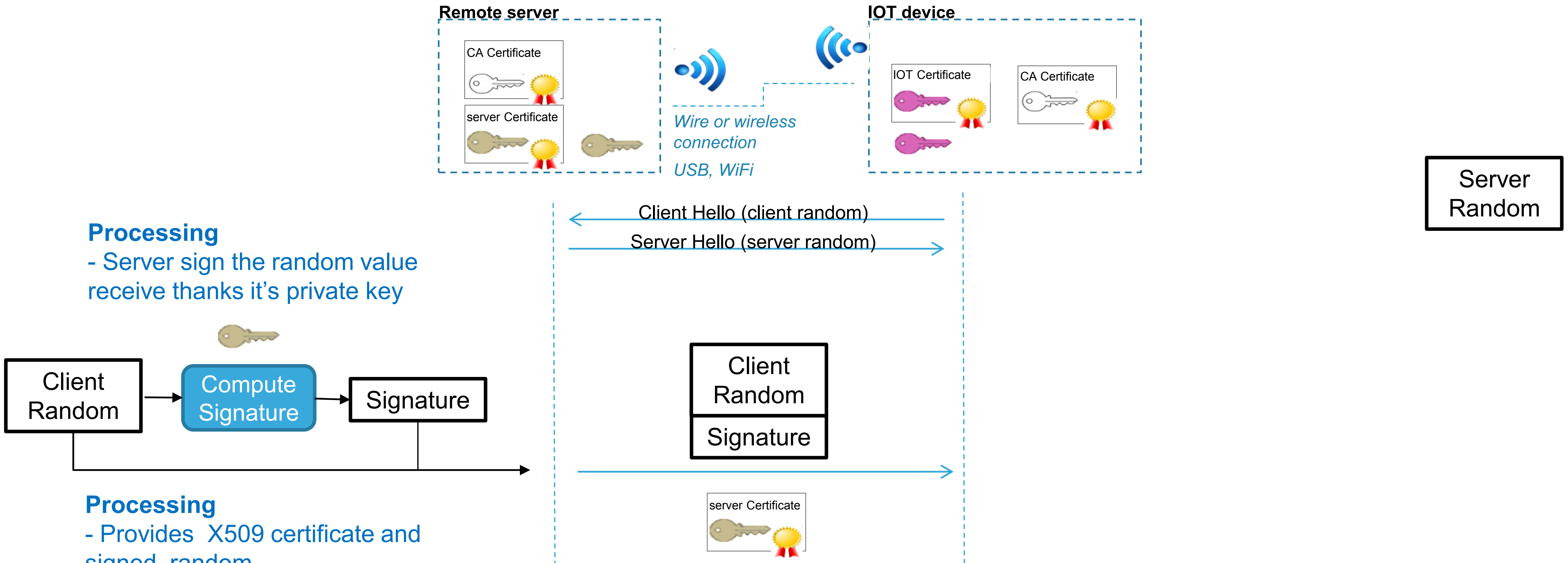
Processing
- Client provides supported TLS version, algorithms and a random

Client Random

Processing
- Replies algorithms choices and a random

Server Random

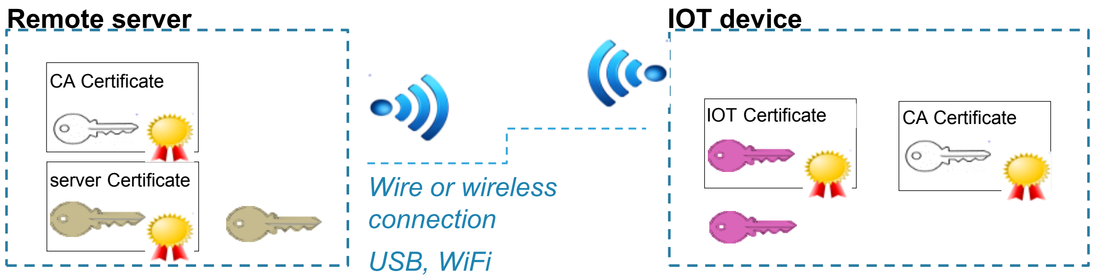
TLS Handshake V1.2 (RFC 5246)

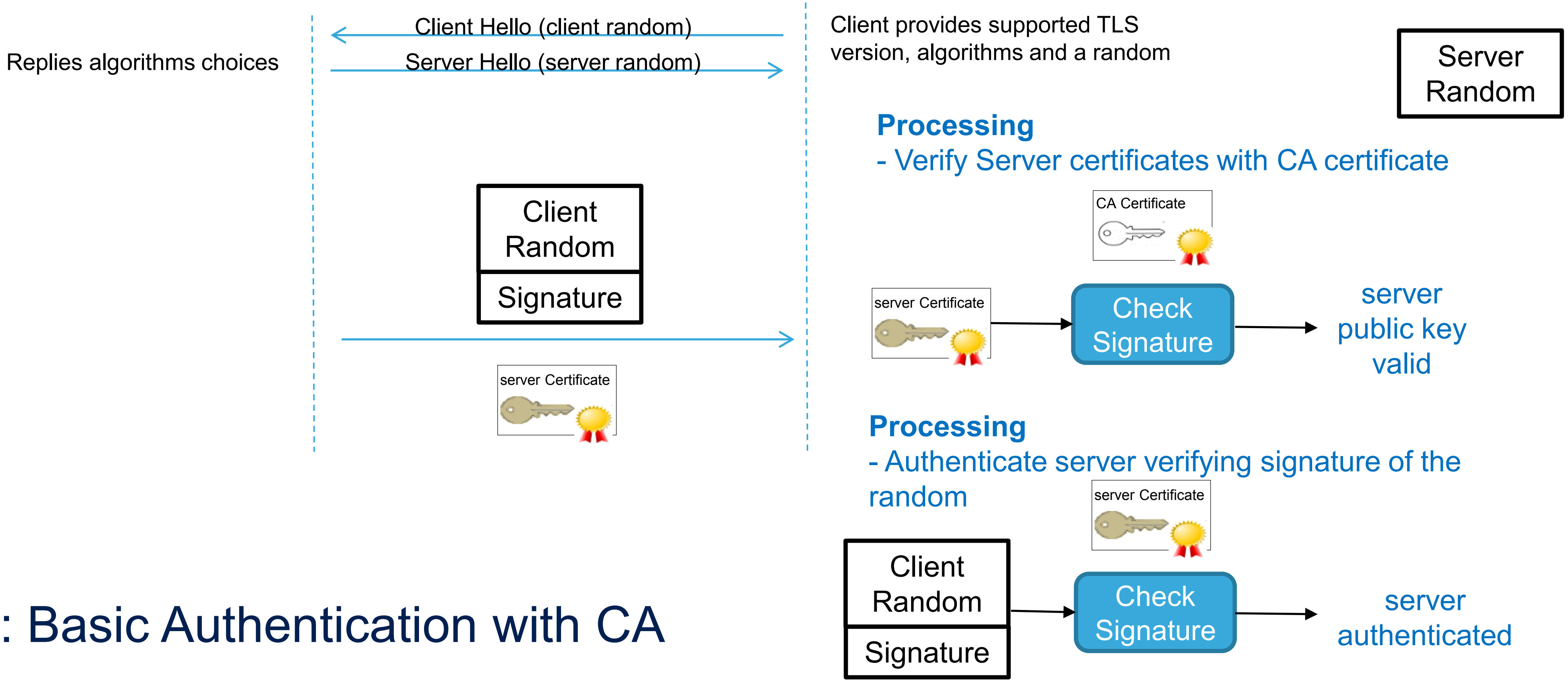


Processing
- Server sign the random value receive thanks it’s private key

Processing
- Provides X509 certificate and signed random

TLS Handshake V1.2 (RFC 5246)





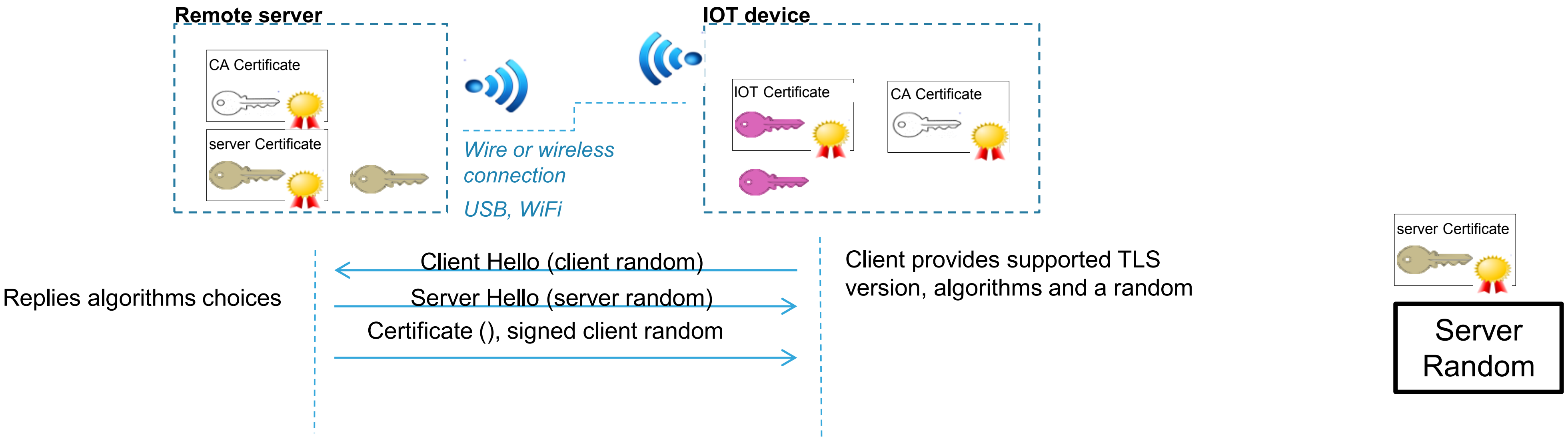
Processing
- Verify Server certificates with CA certificate

Processing
- Authenticate server verifying signature of the random

Slide : Basic Authentication with CA

life.augmented

TLS Handshake V1.2 (RFC 5246)



Replies algorithms choices

Client Hello (client random)

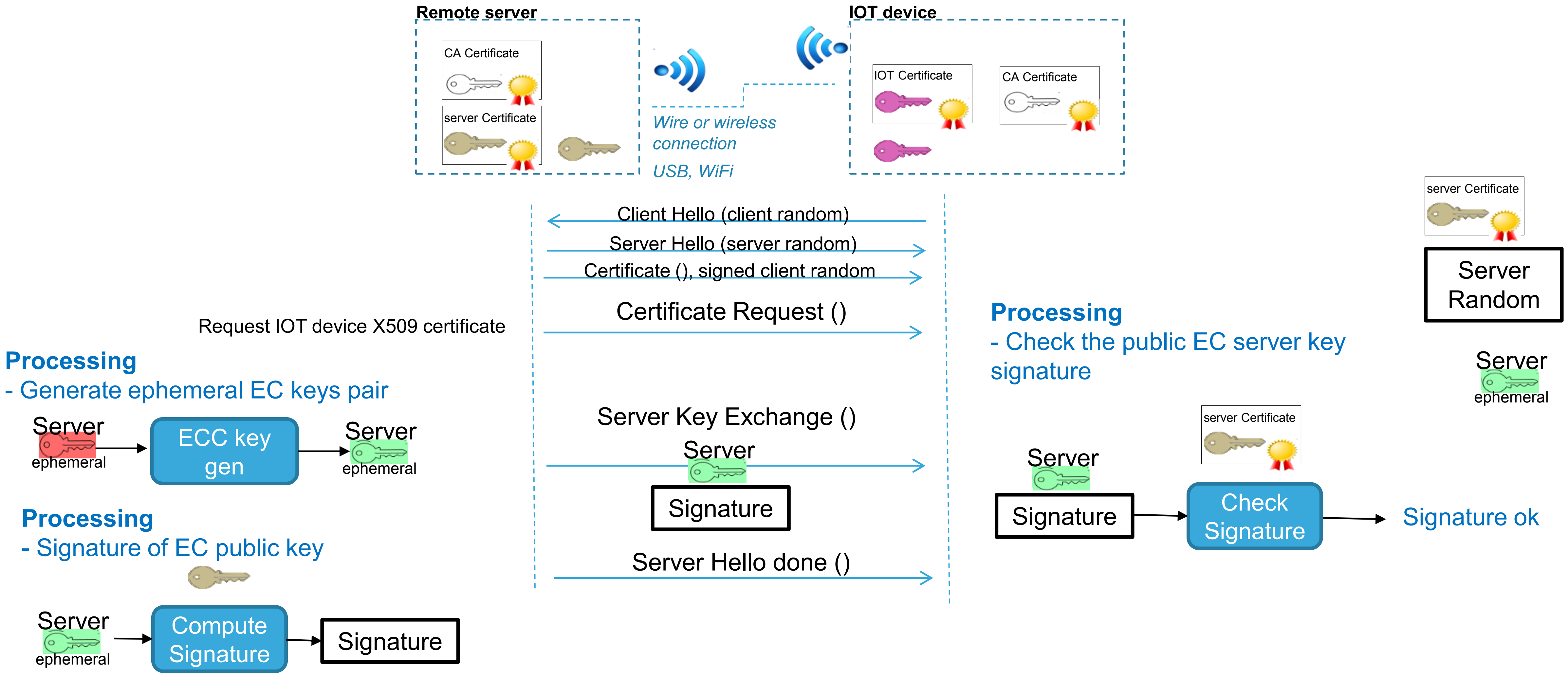
Server Hello (server random)

Certificate (), signed client random

Client provides supported TLS version, algorithms and a random

- IOT device is now sure to exchange with genuine Server
  - The Server certificate has been check
  - Remote server has the private key associated

TLS Handshake V1.2 (RFC 5246)



Remote server

CA Certificate

server Certificate

Wire or wireless connection

USB, WiFi

IOT device

IOT Certificate

CA Certificate

Client Hello (client random)

Server Hello (server random)

Certificate (), signed client random

Certificate Request ()

Request IOT device X509 certificate

Processing

- Generate ephemeral EC keys pair

Server Key Exchange ()

Server Hello done ()

Processing

- Signature of EC public key

server Certificate

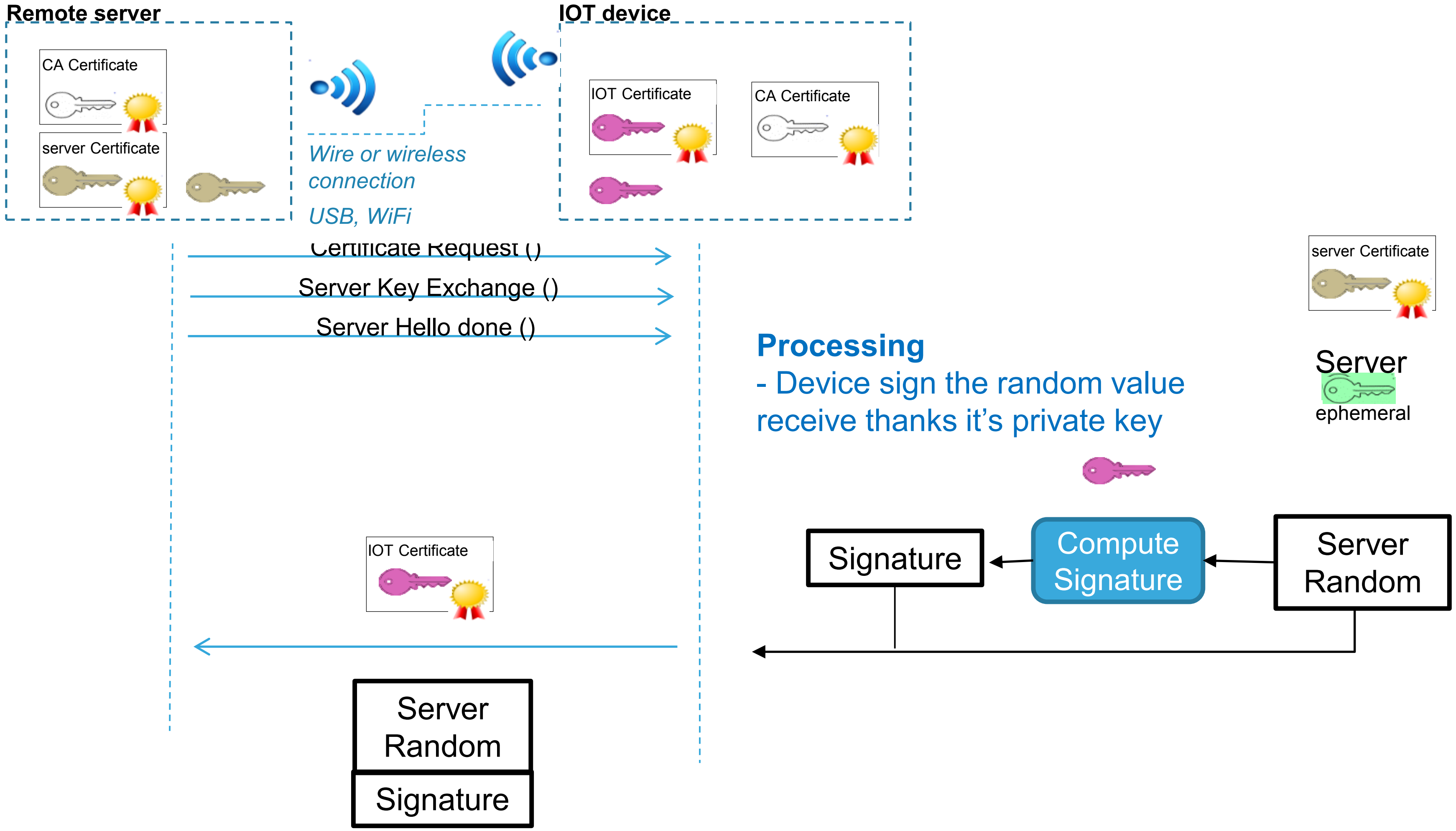
Processing

- Check the public EC server key signature

Signature ok

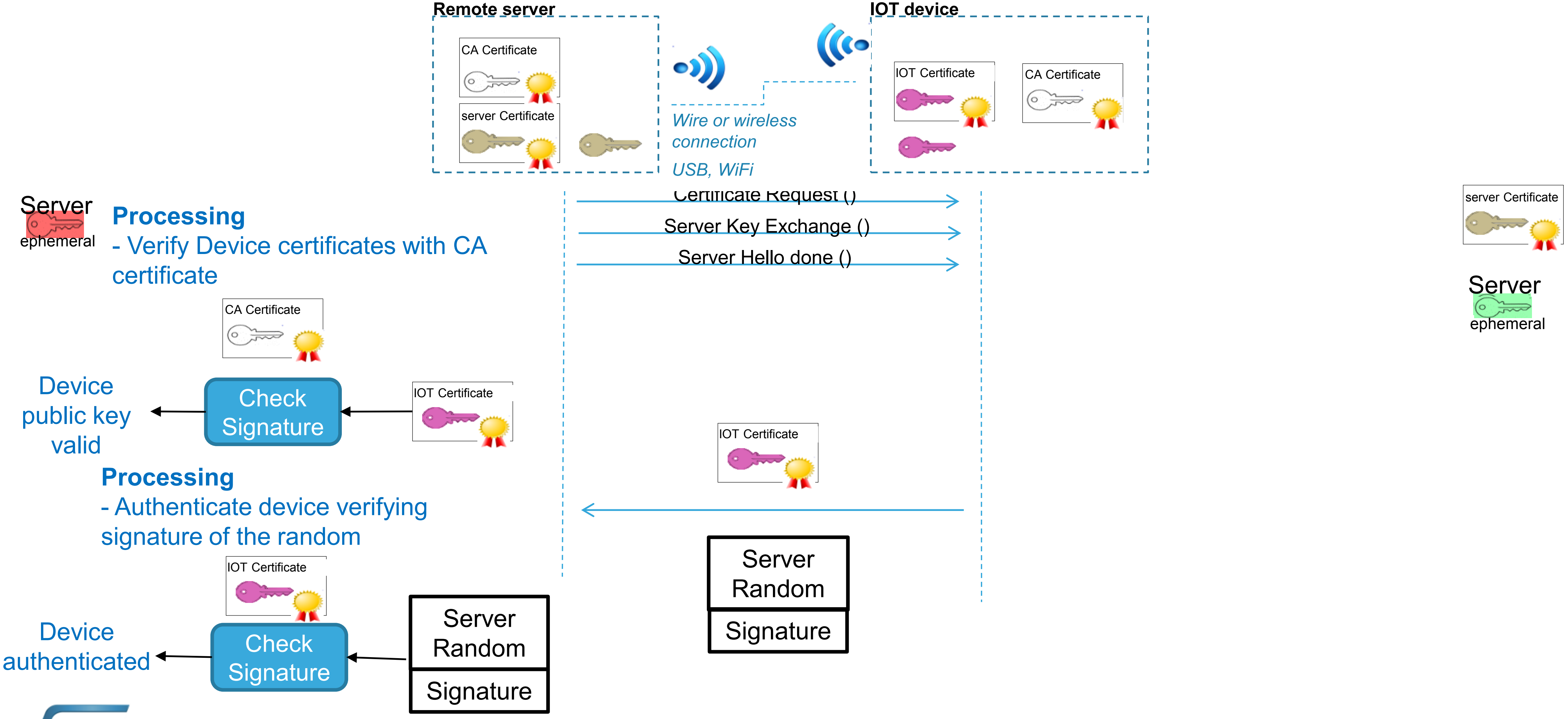
TLS Handshake V1.2 (RFC 5246)

Server
ephemeral



Processing
- Device sign the random value receive thanks it's private key

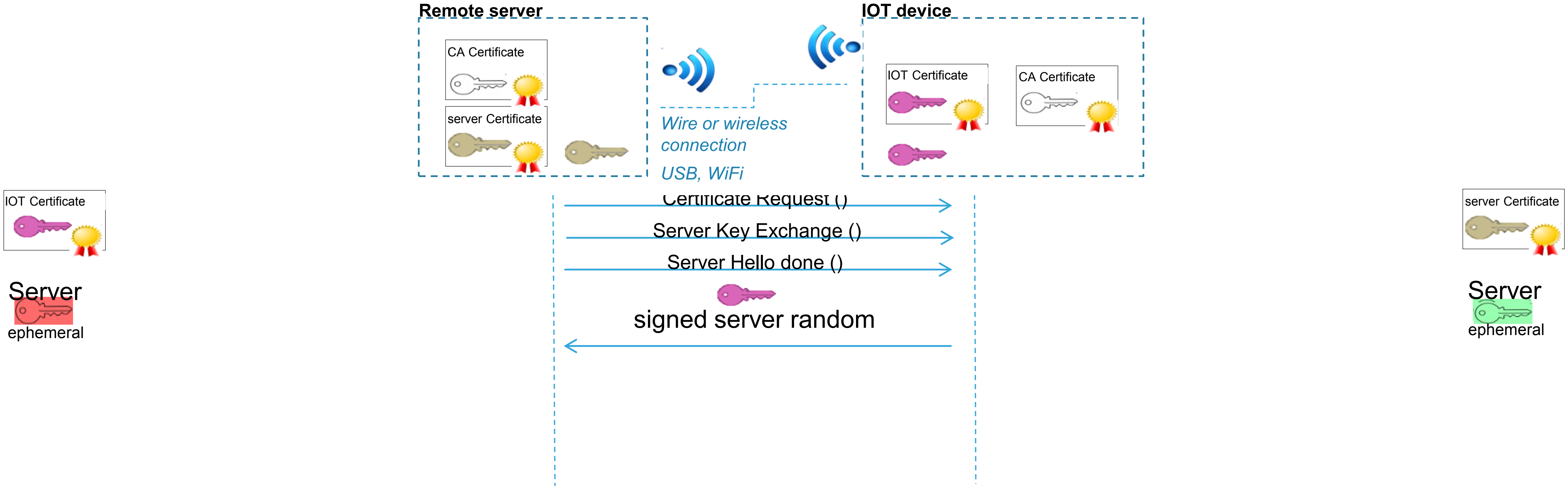
TLS Handshake V1.2 (RFC 5246)



Processing
- Verify Device certificates with CA certificate

Processing
- Authenticate device verifying signature of the random

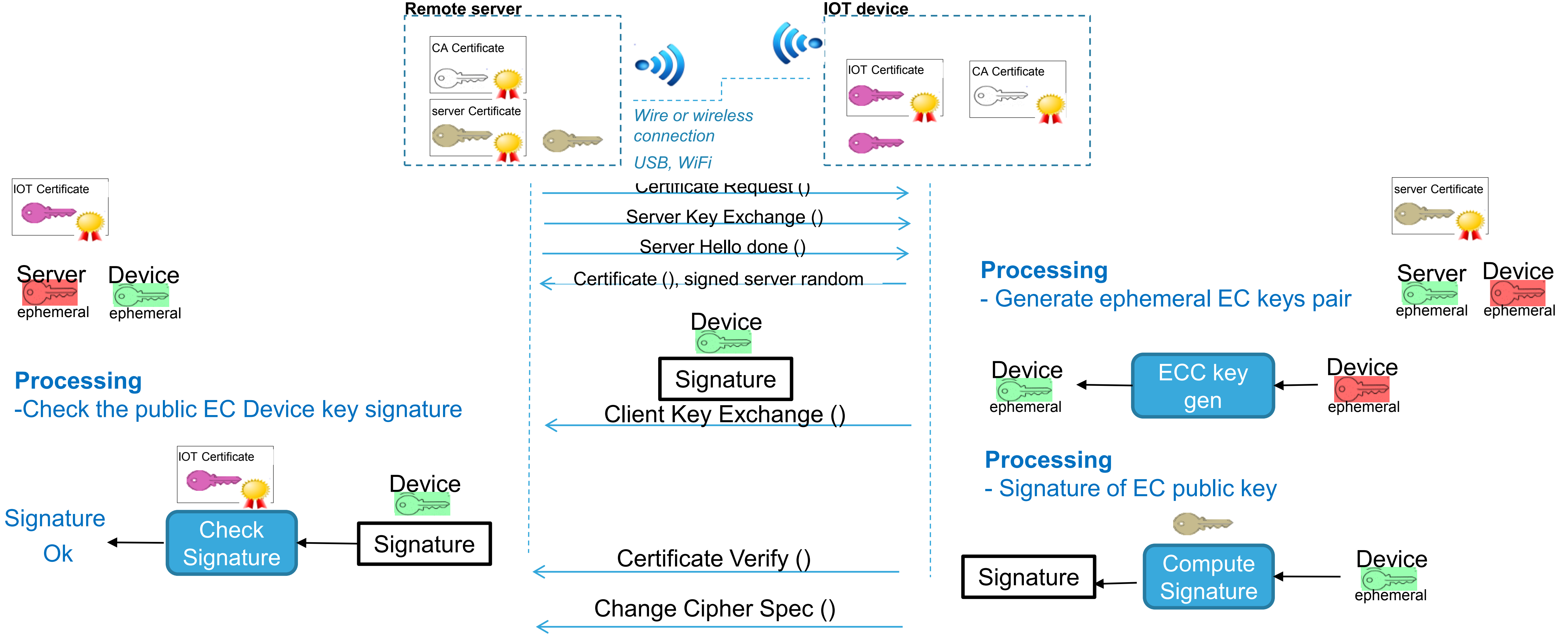
TLS Handshake V1.2 (RFC 5246)



- Server is now sure to exchange with genuine IOT Device

Associated Slide : Basic Authentication with CA

TLS Handshake V1.2 (RFC 5246)



Remote server

CA Certificate

server Certificate

Wire or wireless connection

USB, WiFi

IOT device

IOT Certificate

CA Certificate

Certificate Request ()

Server Key Exchange ()

Server Hello done ()

Certificate (), signed server random

Device

Signature

Client Key Exchange ()

Certificate Verify ()

Change Cipher Spec ()

IOT Certificate

Server ephemeral

Device ephemeral

Processing

-Check the public EC Device key signature

IOT Certificate

Device

Signature

Check Signature

Signature Ok

server Certificate

Processing

- Generate ephemeral EC keys pair

Server ephemeral

Device ephemeral

Device ephemeral

ECC key gen

Device ephemeral

Processing

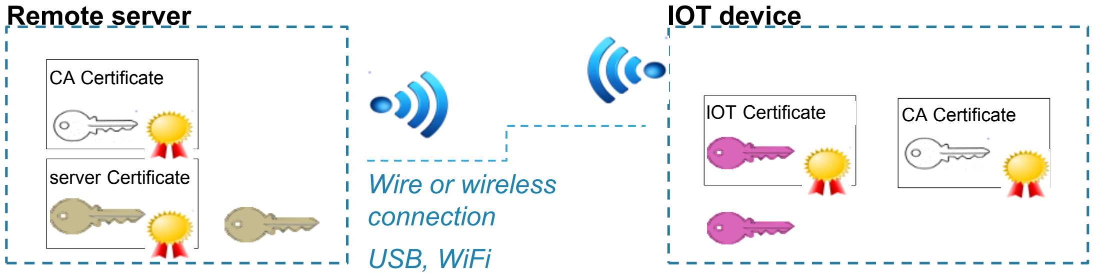
- Signature of EC public key

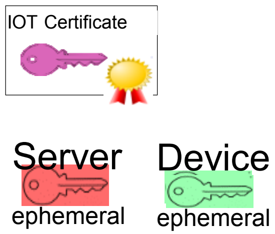
Signature

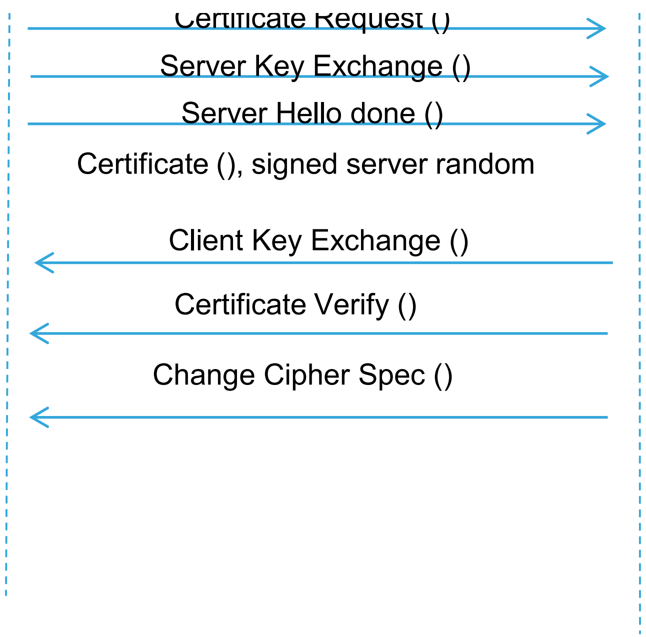
Compute Signature

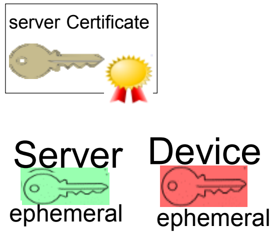
Device ephemeral

TLS Handshake V1.2 (RFC 5246)







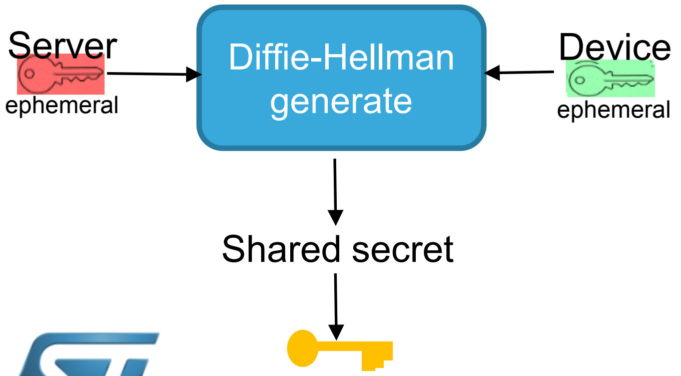


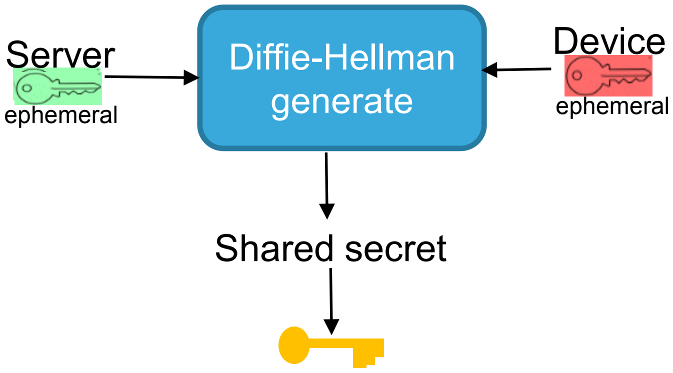
Processing

- Computes shared secret using Remote device public key

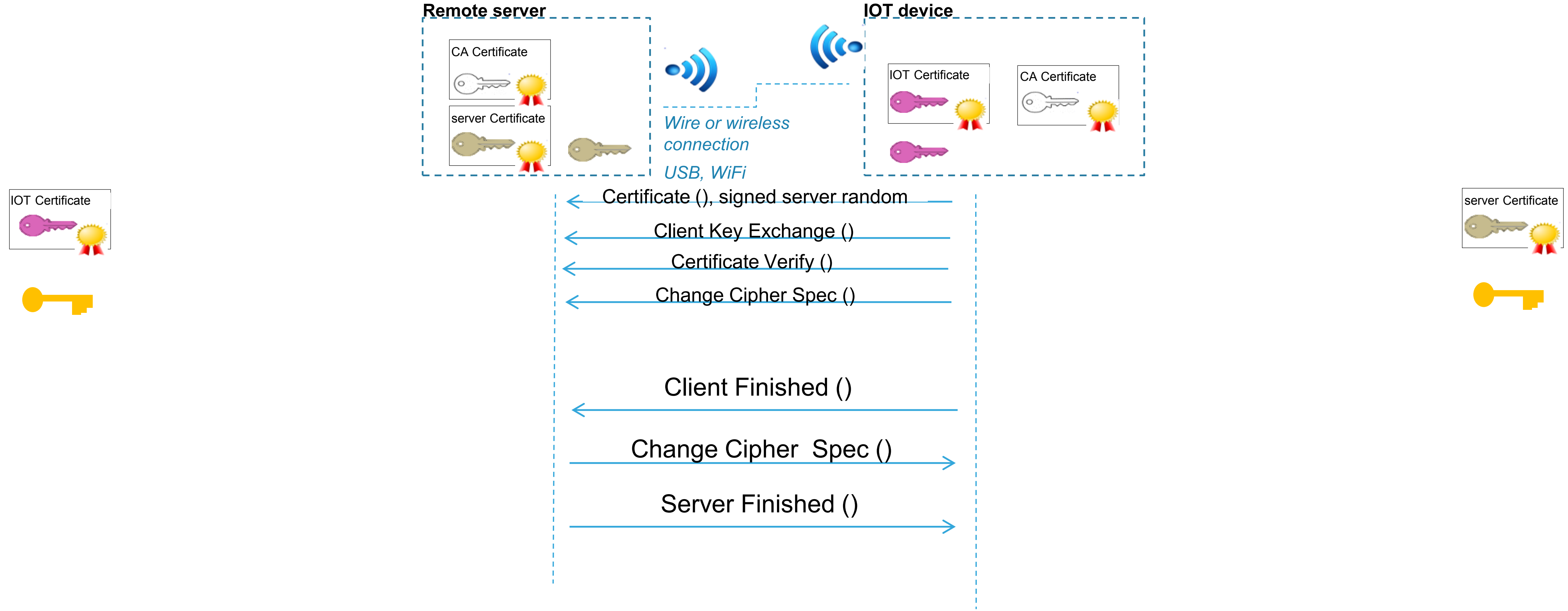
Processing

- Computes shared secret using Server public ECC key / Device private ECC key





TLS Handshake V1.2 (RFC 5246)



- Handshake is finished, mutual authentication done and a shared secret is available to encrypt the communication

Hands-on