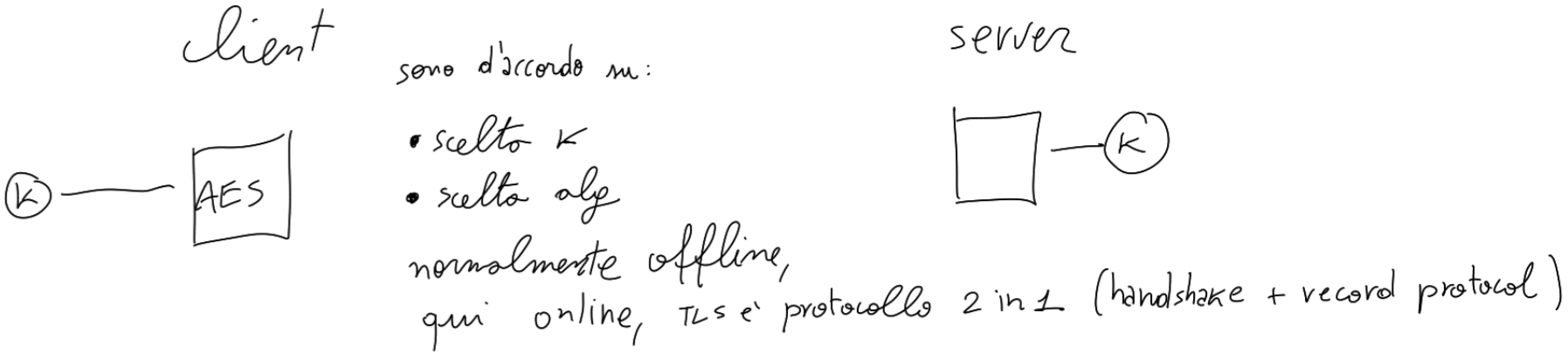


TLS Handshake Protocol

up to TLSv1.2!!!

TLSv1.3 differs, more later

Handshake: when

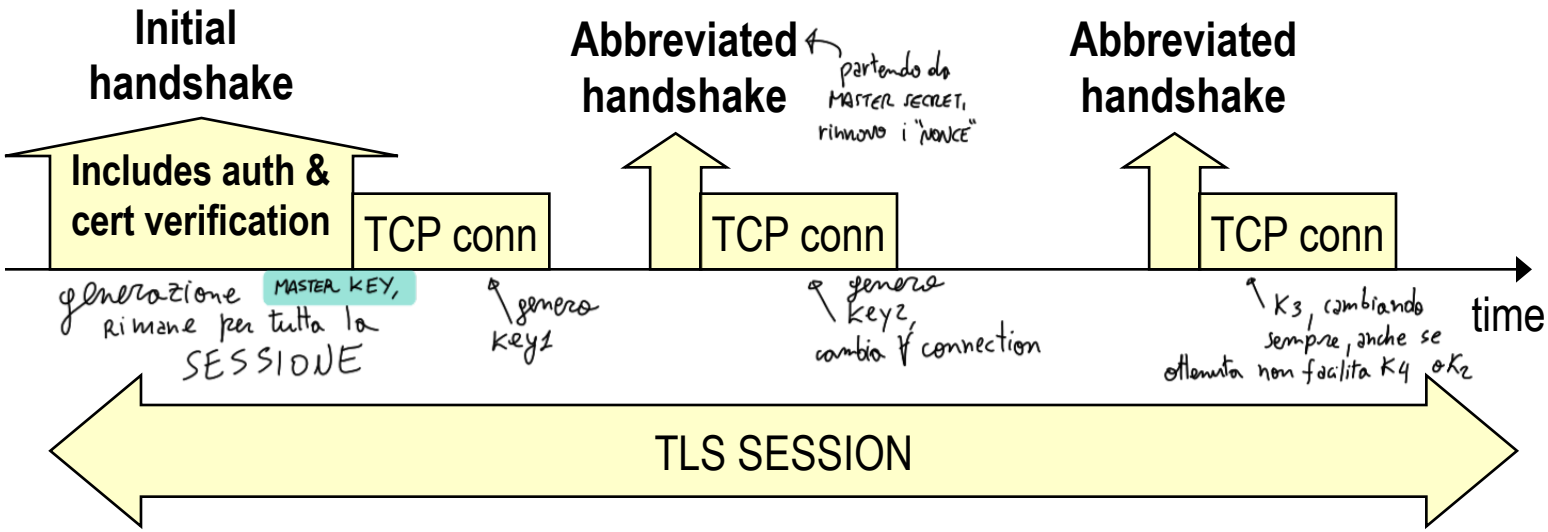
- **Initial negotiation and exchange of parameters**
  - (mutually) authenticate (o single-side, dipende)
  - agree on algorithms (attaccabile)
  - exchange random values (nonces)
  - Exchange secrets or information to compute secrets

Importante dividere algoritmo da protocollo!
non deve avere alg. hardcoded

vorrei condividere segreti per instaurare connessione segrete, cioè introduco PUBLIC KEY

NB: telefoni hanno public key solo dal 5g, con chiave nella SIM.

- **session resumption**
  - Re-keying ("refresh" secrets)



Handshake: goals

che crypto serve in TLS?
- decidere asymetric encrypt (tanti meccanismi, tante varianti)
- quale simmetric encryption? (fino a TLS 1.2 devo scegliere separati "encrypt" e "mac")
- quale HMAC integrity algorithm? (cioè quale hash in HMAC?)

esempio WireShark:

TLS_RSA_WITH_AES_256_CBC_SHA

asimmetrica (handshake) — simmetrica — hash function HMAC

➔ **Secure negotiation of shared secrets**

➔ **Via Asymmetric cryptography**

⇨ Never transmitted in clear text;

⇨ derived from crypto parameters exchanged

➔ **Optional authentication**

⇨ for both client and/or server

➔ in practice always required for server

⇨ Robust to MITM attacks

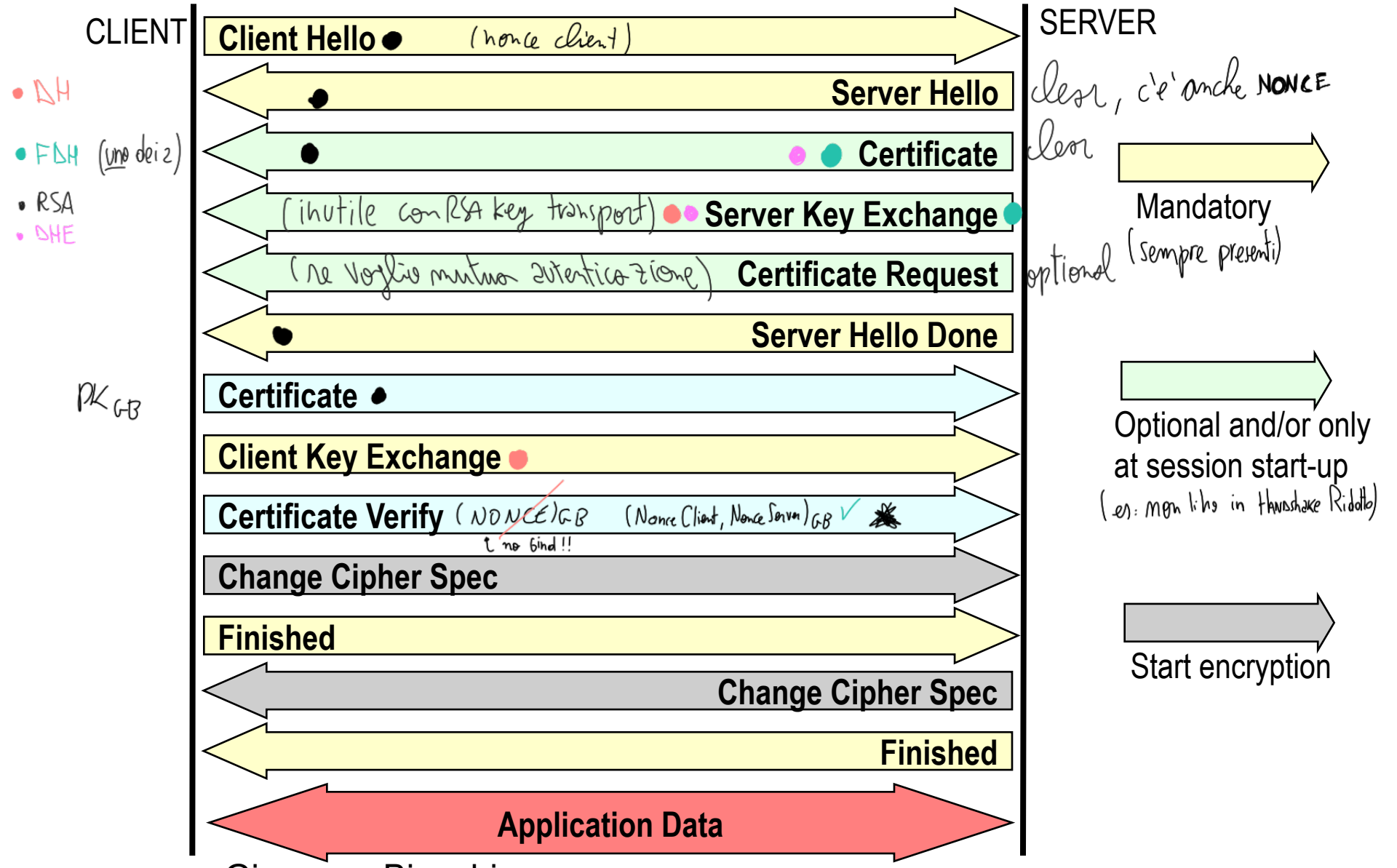
➔ **Reliable Negotiation:** (Potrei negoziare un qualcosa di broken)

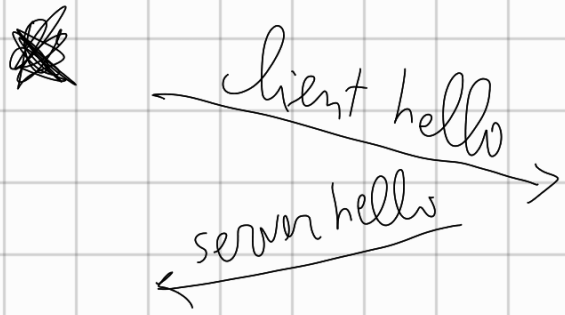
⇨ Attacker cannot tamper communication and affect/alter the negotiation outcome without being detected by the involved C&S parties

➔ Fundamental: negotiation amenable to downgrade attacks!!

↳ vedremo un trick essenziale per non avere problemi

Handshake Messages





invece di estrarre solo il nonce, prendo tutto e lo ripeto, stavolta integrity protected msg, così ho authentication e downgrade protection!

Ancora meglio log (ClientHello, ServerHello, ..., Client Key Exchange) e mando come Certificate Verify, proteggo la versione (ordine msg ok!) e mando solo il tag.

Il server, per via TCP, vede tutta la sessione e ricompila tutto il LOG, confrontandolo.

Certificate Verify prova chi è chi, ma protegge contro downgrade ed altro! (NEGOZIAZIONE PROTETTA)
PROBLEMA: molti msg opzionali!!

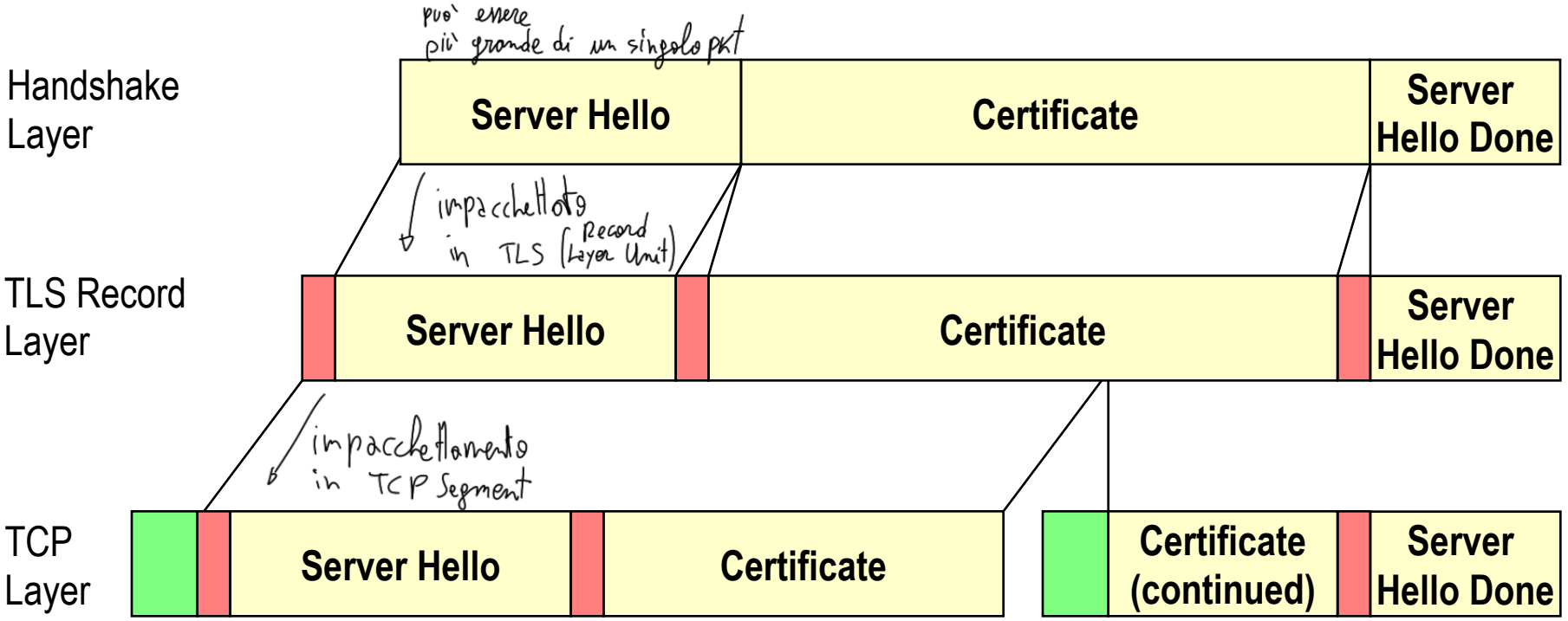
RSA non obbligato protegge fino a Client Key Exchange

in Finished-client faccio H(Log fino Change Cipher Spec lato client), proteggo ciò che ho negoziato
in Finished-server faccio H_{MAC} o AES-ECB (fino "Finished" lato server)

se ragazzi pono a scambio dati

DIMOSTRA che TLS with encryption only IS broken! (no encryption)
Posso creare msg che rompe handshake?

TCP segmentation (example)



➔ **Simplified example: server responds to client hello with certificate only**

➔ **Arbitrary correspondence between TLS Records and TCP segments (TLS records fragmented into TCP segments)**

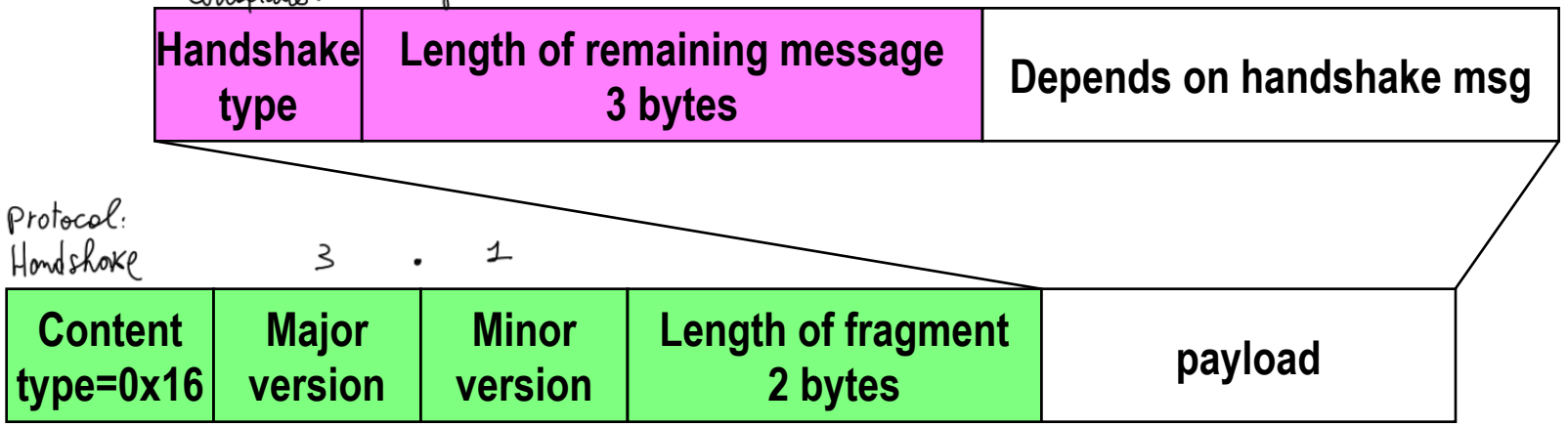
⇨ Obvious! Do you remember TCP? ☺

Handshake message format

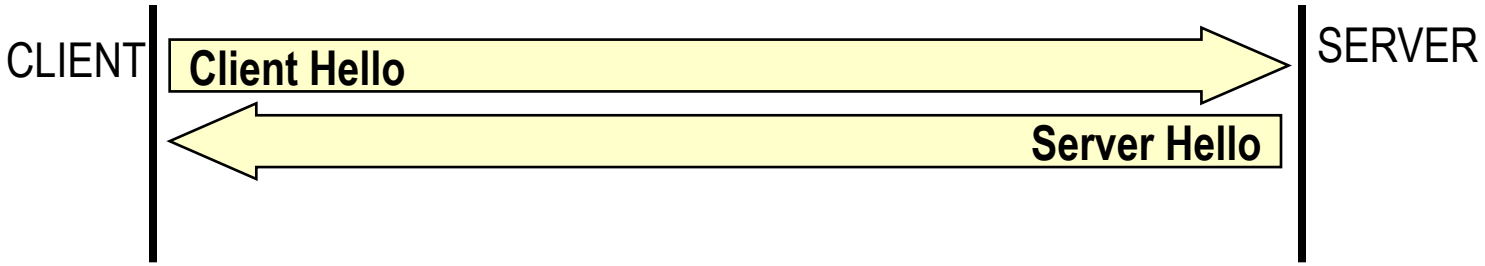
- **Incapsulated in TLS Record**
- **Handshake types:**
  - hello_request(0), client_hello(1), server_hello(2),
  - certificate(11), server_key_exchange (12),
  - certificate_request(13), server_hello_done(14),
  - certificate_verify(15), client_key_exchange(16),
  - finished(20)

- Non avendo ancora la chiave, non ho ancora protezione.

Server/Client Hello? Certificate? ognuno lo calcola come vuole



Handshake phase 1



➔ **General goal:**

⇨ Create TLS/SSL connection between client and server

➔ **Detailed goals**

⇨ Agree on TLS/SSL version

⇨ Define session ID

⇨ Exchange nonces

→ timestamp + random values used to, e.g., generate keys

⇨ Agree on cipher algorithms (ho una lista di cipher, dal minimo al max, se volessi uno non presente ⇒ non mi connetto)

⇨ Agree on compression algorithm

Client Hello

non si tratta di una semplice "introduzione", bensì di una negoziazione di parametri.

- **First message, sent in plain text**
- **Incapsulated in 5 bytes TLS Record**
- **Content:**
  - Handshake Type (1 byte), Length (3 bytes), Version (2 bytes)
    - Type 01 for Client Hello
    - Version: 0300 for SSLv3, 0301 for TLS
  - 32 bytes Random (4 bytes Timestamp + 28 bytes random)
    - First 4 bytes in standard UNIX 32-bit format
      - » Seconds since the midnight starting Jan 1, 1970, GMT
  - (Session ID length, session ID)
    - 1+32 bytes – either empty or previous session ID
      - » Previous session ID = resumed session state
  - (Cipher Suites length, Cipher suites)
    - 2+Nx2 bytes, in order of client preference
  - (Compression length, compression algorithm)
    - 1+1 byte (in all cases, today: compression algo = 00 = null)

Cipher suites

PUBLIC-KEY ALGORITHM | SYMMETRIC ALGORITHM | HASH ALGORITHM

TLS_AAAAAAA_WITH_BBBBBBBB_CCCCCCCC

TLS_NULL_WITH_NULL_NULL ← INITIAL (NULL) CIPHER SUITE (solo per compressione) (ora disabilitata)

- NON posso negoziare encryption senza integrity (cioè MAC) (principalmente per sicurezza...)
- posso fare il contrario! (solo integrità)

TLS_RSA_WITH_NULL_MD5
TLS_RSA_WITH_NULL_SHA

TLS_RSA_WITH_RC4_128_MD5
TLS_RSA_EXPORT_WITH_RC2_CBC_40_MD5
TLS_RSA_WITH_3DES_EDE_CBC_SHA

TLS_DH_DSS_WITH_3DES_EDE_CBC_SHA
TLS_DH_RSA_WITH_DES_CBC_SHA

TLS_DHE_DSS_WITH_DES_CBC_SHA
TLS_DHE_RSA_WITH_3DES_EDE_CBC_SHA

TLS_DH_anon_WITH_RC4_128_MD5

JUST A FEW EXAMPLES
(FULL LIST IN RFC)

Note: arbitrary combination not possible: must choose from a (small) list of combinations

Server Hello

- **In reply to Client Hello, sent in plain text**
- **Content:**
  - Handshake Type (1 byte), Length (3 bytes)
    - Type = 02 for Server Hello)
  - Version
    - Highest supported by both Client and Server
  - 32 bytes Random (4 bytes Timestamp + 28 bytes random)
    - Different random value than Client Hello
  - (Session ID length, session ID)
    - If Client session ID=0, then generate session ID
    - Otherwise, if resumed session ID OK also for Server, use it;
    - Otherwise generate new one
  - Cipher Suite, 2 bytes
    - Selected from client list (usually best one, but no obligation)
  - Compression algo, 1 byte
    - Selected from Client list

Interlude: how public key cryptography is used in TLS

(basic idea, details later on)

Come fare setup shared secret?

Problems solved by asymmetric Encryption: a typical example

• come fare setup segreto senza operazioni preliminari?

Download app

Transfer encrypted data

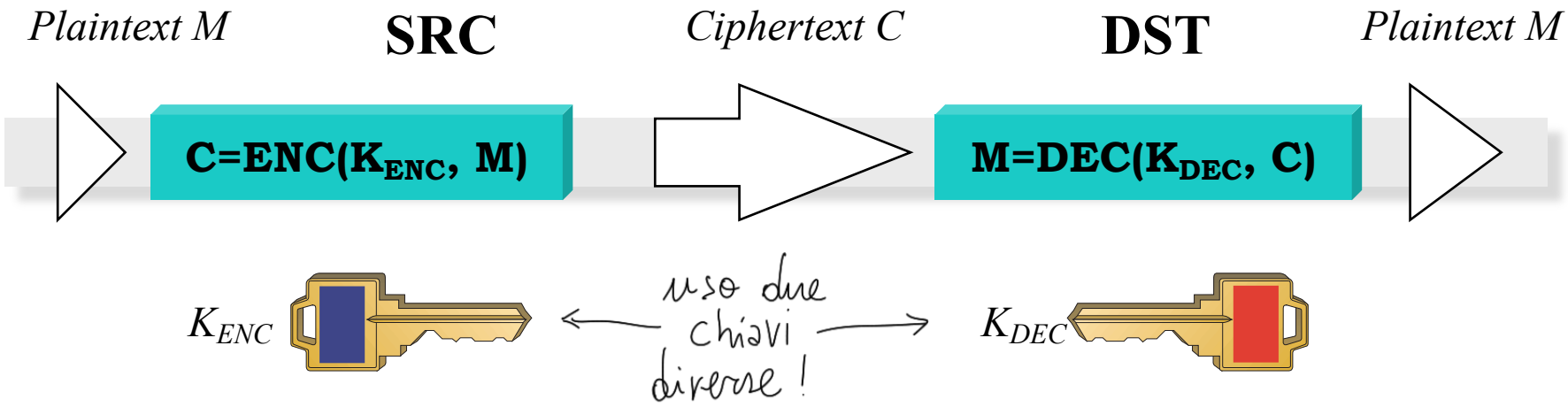
How to transfer and store symmetric key?

- **In the code? Not nearly a good idea!!!**
- **But then, how to?** Uso public key encryption per trasmettere la chiave che userò per symmetric encryption

Note the difference with cellular → key in the SIM

Asymmetric cryptography

M	plaintext
C	ciphertext



K_{ENC}	encryption key	$K_{DEC} \neq K_{ENC}$	decryption key
ENC(.)	encryption algorithm	***DEC(.)***	decryption algorithm

Encryption and decryption keys are obviously linked…
But CANNOT be determined from each other
(unless you know 'more')

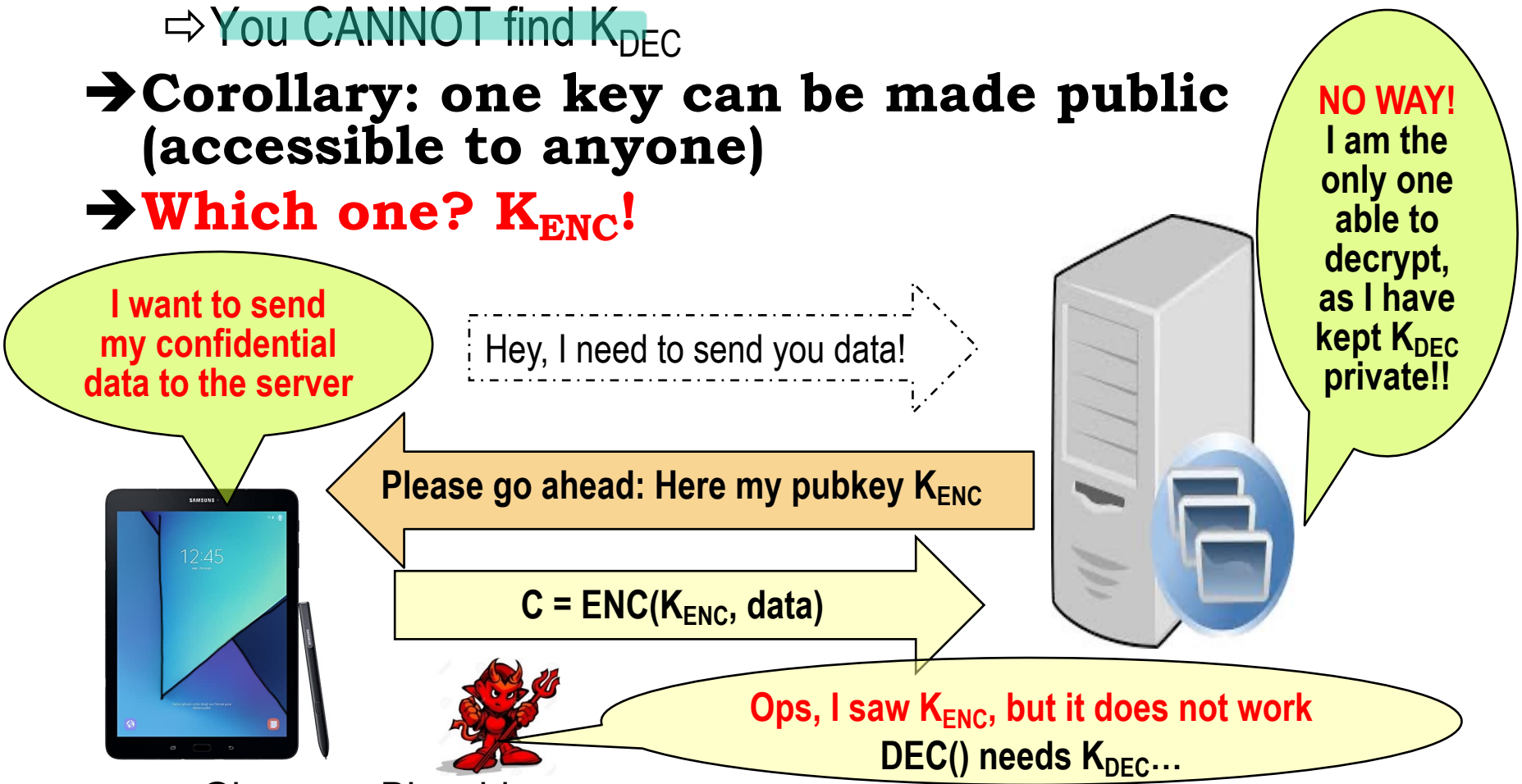
How to encrypt data? Public and private key!

➔ **Asymmetric key property:**

- ⇨ If you know K_{ENC} …
- ⇨ You CANNOT find K_{DEC}

➔ **Corollary: one key can be made public (accessible to anyone)**

➔ **Which one? K_{ENC}!**



Asymmetric vs Symmetric?

➔ **They do address DIFFERENT problems**

⇨ Would be wrong to say that one is superior than the other!

➔ **BOTH can be (in)secure**

⇨ Depends on algorithms (RSA, El Gamal, etc)

⇨ Depends on key sizes

➔ **Asymmetric is certainly more flexible…**

⇨ No need to preshare keys – more later

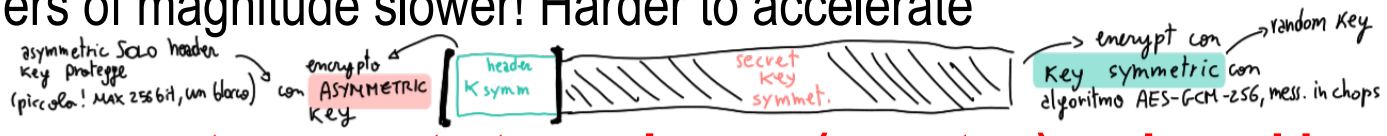
⇨ But you need also more complex protocols!

(pesante!)

➔ **But asymmetric is computationally heavier than symmetric**

⇨ 4-5 orders of magnitude slower! Harder to accelerate

(Hybrid Encryption)

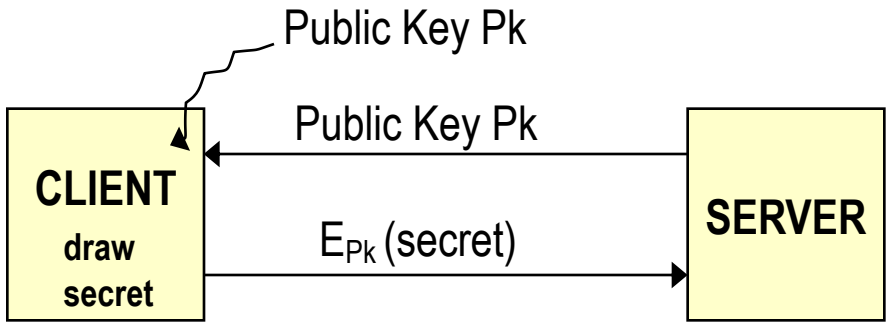


Take-home: use asymmetryc crypto to exchange (or setup) a shared key and THEN transfer massive data using symmetric encryption

Actually, TWO basic approaches to key management (both possible until TLS1.2)

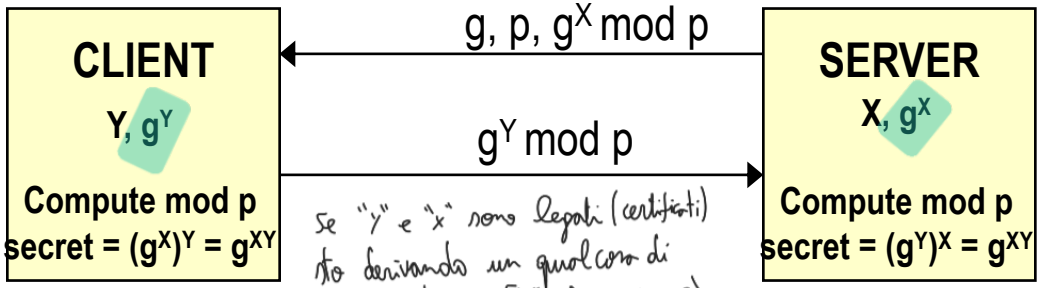
➔ **1. Key transport (e.g., RSA)**

- ⇨ Server sends Public Key
- ⇨ Client draws random secret (caso segreto)
- ⇨ Which is encrypted using server Pk
- ⇨ And transported to the server
- ⇨ **Both peers now have the same secret**



➔ **2. Key agreement (e.g., DH)**

- ⇨ Parties independently generate a private and a public quantiity (PK, SK)
- ⇨ Then exchange ONLY the public quantity
- ⇨ This, combined with the other party's private part, permits both of them to compute a SAME SECRET
- ⇨ **Both peers now have the same secret**



Se "y" e "x" sono legati (certificati) sto derivando un qualcosa di dipendente, in FIXED DH key è sempre la stessa, male!

Per negoziazione sicura comunico prima in chiaro, e poi ripeto in canale sicuro.

BID DOWN ATTACKS: tra  e  posso comunicare "capabilities" in chiaro, anche l'uso della batteria.

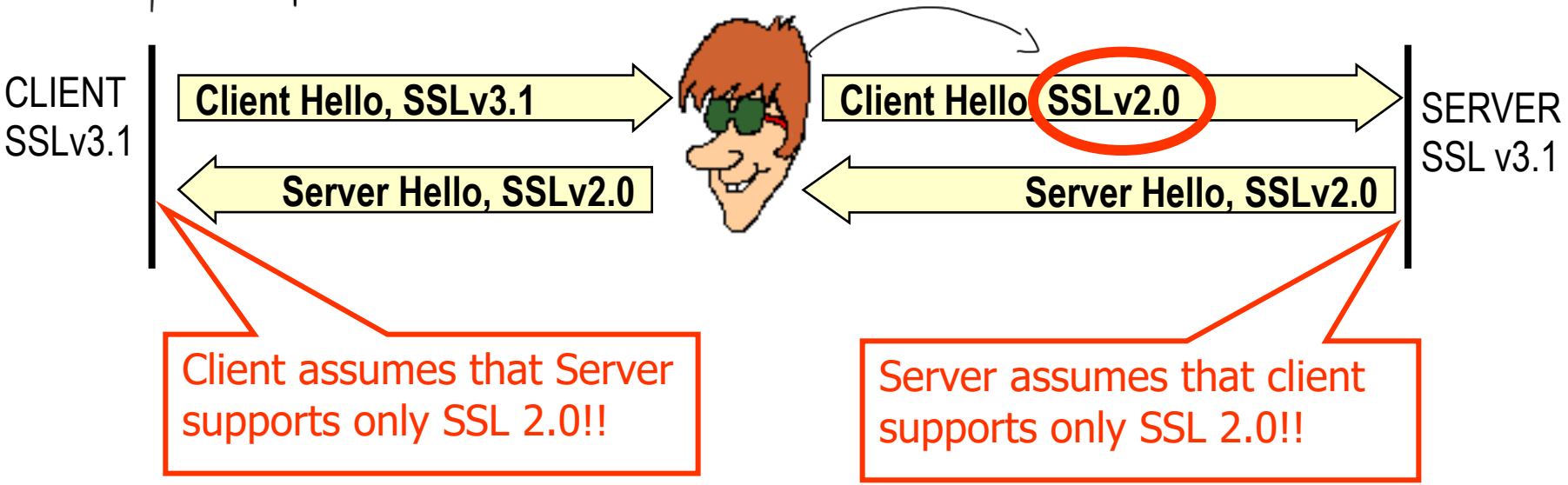
Essendo in chiaro e pre-autenticazione, un MITM può togliere il "risparmio energetico" (battery drain attack) e far scaricare il telefono.

Neanche nel 5G è stato risolto. Se rimando le capabilities, posso bloccare tale attacco!

Back to TLS handshake...

Version Downgrade attack

MITM – doesn't break SSL3.x, but knows very well how to break SSL2.0 (40 bit encryption only)

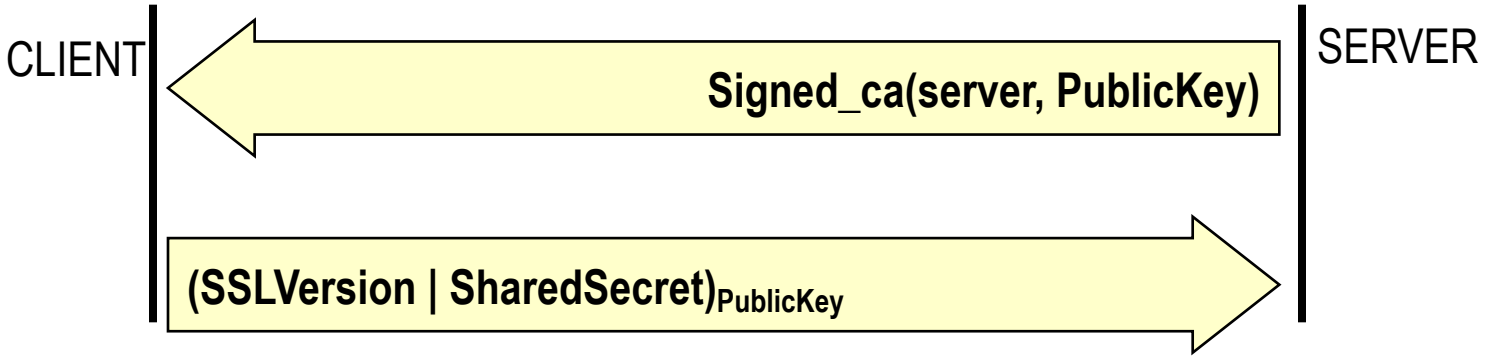


➔ Downgrade attacks on cipher suites, too

- ⇨ MITM: remove "strong" cipher suites, and leave in Client Hello only the ones he knows he can break!

Since hello message authentication not viable at the moment (not yet a shared secret available), verification must be necessarily delayed to a subsequent phase…

Handshake phase 2 & 3 (schematic, RSA case)



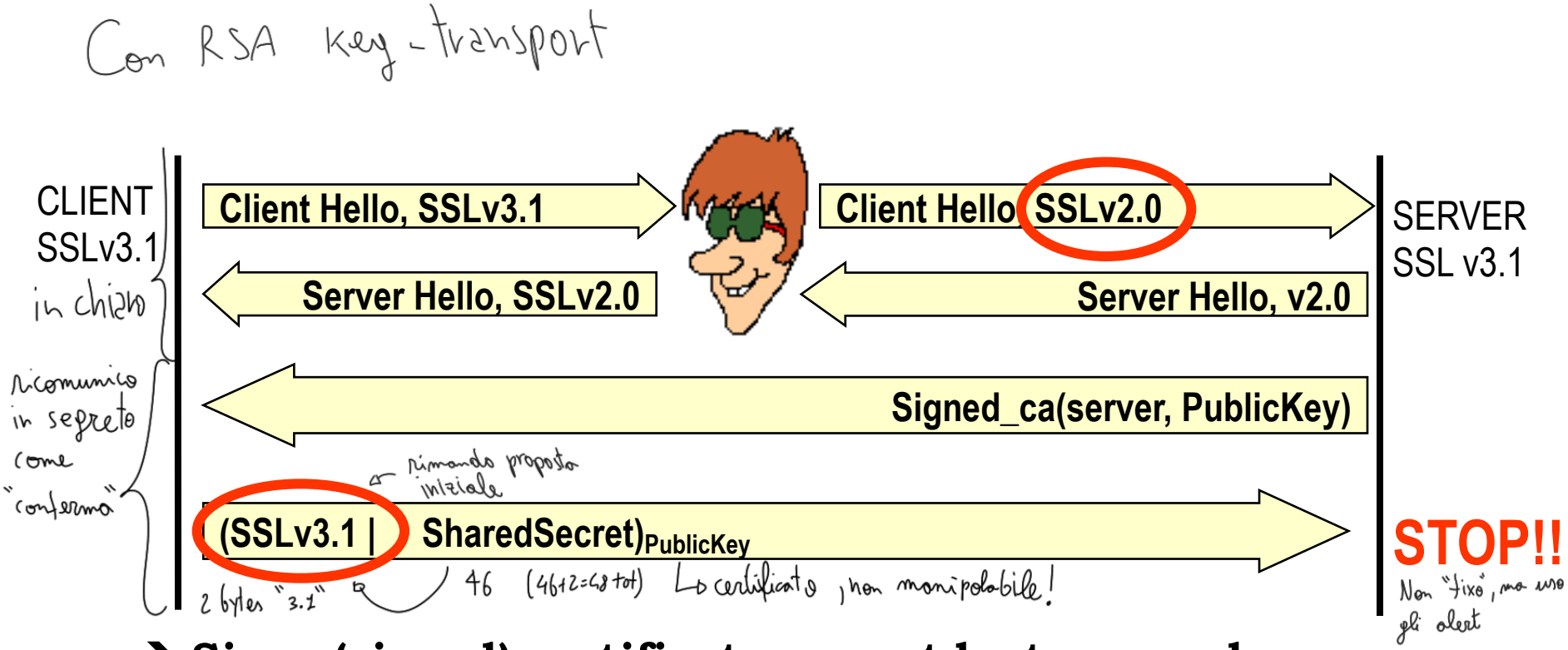
➔ **Phase 2:**

- ⇨ Server sends authentication information (certificate)
- ⇨ Along with Public Key (in certificate or in extra msg)

➔ **Phase 3:**

- ⇨ Client generates a Shared Secret
  - → Length depends on agreed cipher suite (RSA = 46 bytes)
- ⇨ And transmits it to Server, encrypted with Public Key
  - → Native SSL version included to early combat downgrade attacks

Against downgrade attack



➔ **Since (signed) certificate cannot be tampered…**

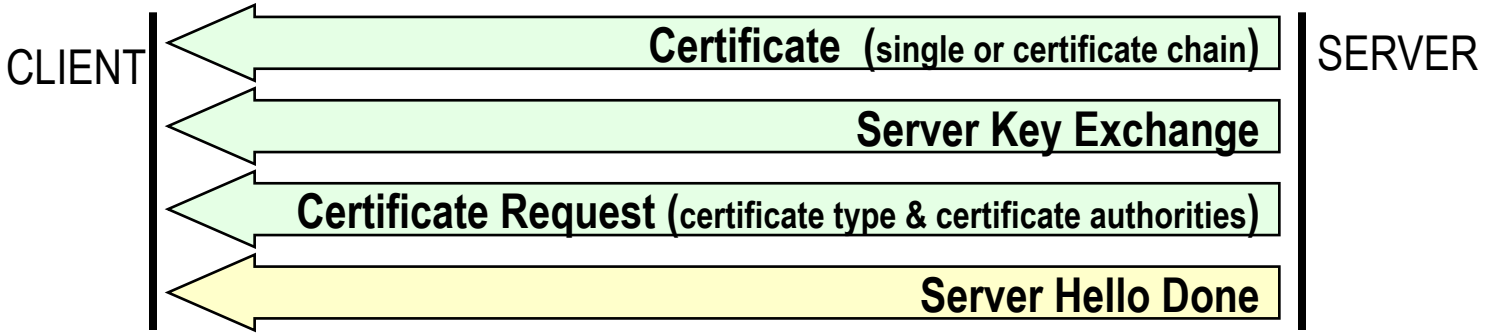
➔ **… and since MITM cannot decrypt SharedSecret**

⇨ and must act as pass-through

➔ **Downgrade attack on SSL version easily discovered!**

⇨ Encrypted SSL version is the one initially proposed, NOT the one negotiated in server hello, of course

Handshake Phase 2 details



- **Certificate**
  - typically a certificate chain
- **Server Key exchange**
  - When certificate not issued (no server authentication required – UNSAFE!!)
  - When certificate key is for signature only (not for encryption)
  - When certificate key cannot be used for legal reasons
    - E.g. RSA key larger than 512 bits may not be used for encryption outside US, but only for signature
    - Larger RSA key encoded in certificates may be used to sign shorter, temporary, RSA key
  - When Ephemeral Diffie-Hellman used
- **Certificate request**
  - Requires client authentication (asks for an acceptable certificate)
- **Server Hello Done**
  - empty message

Example: Diffie-Hellman / 1

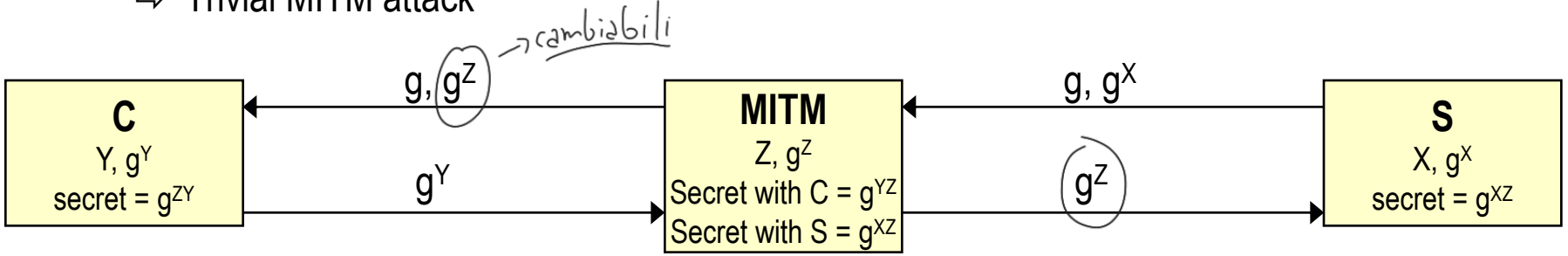
- **Review of DH**
  - Server transmits public value g^X mod p
  - Client replies with public value g^Y mod p
  - Both compute shared key as
    - $K = (g^Y)^X \bmod p = (g^X)^Y \bmod p$
- **Three basic approaches (see next slide):**
    - Anonymous
    - Fixed
    - Ephemeral
  - Depends on whether X and Y values are pre-assigned or dynamically generated and/or signed
  - Anon & Ephemeral DH transmitted in ServerKeyExchange

Example: Diffie-Hellman /2

DH variants

OK passive attack
NO active attack!

- **"Anonymous" (basic) Diffie-Hellman** (Vanilla)
  - X, Y generated on the fly and NOT authenticated (end-points)
  - Trivial MITM attack



- **(Fixed) Diffie-Hellman**
  - DH "public" parameters g^X and g^Y are static and signed by a certification authority
  - No MITM anymore
  - But they are long-lived → brute-force attacks
    (key sempre uguale se mi collego a stesso server)
- **Ephemeral Diffie-Hellman**
  - DH parameters can vary
  - But unlike anonymous DH, they are SIGNED by the party
  - Hence parties mush have an RSA or DSS secret key for signature purposes
    - → Hence, why don't just use RSA key transport?
      (we are here neglecting IPR/legal issues, of course ☺)

Posso avere i pieg di entrambi? Senza CA_autority?
Ephimeral DH exchange

DHE - ephimeral

$CA(GB, PK_{GB})$

GB
PK_{GB}, SK_{GB}

$(GB, PK_{GB})_{CA}$ non cambia nel tempo!

uso per signature

$(Giuseppe, g^y)_{SK_{GB}}$

g^y → DH coeff.

customer

GB —certificate→ $(GB, PK_{GB})_{CA}$ (una volta e basta)

—key exchange→ $(GB, g^x)_{GB}$ disabilito MITM (GB autorizzato da CA)

posso cambiare x con $x_1, x_2, x_3 \ldots$ quando voglio, perché sono autorizzato da CA, e questo lo autorizzo io.

ECDHE_ECDSA è standard-de facto

(prima delle curve ellittiche)

Example: Diffie-Hellman /3

ServerKeyExchange message format:

Content type=0x14	Major version=3	Minor version=1	Length of fragment 2 bytes = nn

TLS Record Header (5 bytes)
0x14 = Handshake protocol

Handshake type=0x0c	Length 3 bytes = nn-4

Handshake Header (4 bytes); 0x0c=ServerKeyEx

DH p length 2 bytes	p = parameter, cambiale — DH p value (prime number) ***	p
DH g length 2 bytes	DH g value ***	g
DH Ys length 2 bytes	DH Ys value ***	g^X

DH parameters (variable length)

Signature RSA → 16 MD5 + 20 SHA-1 DSA → 20 SHA-1 only

Signature size and hash used depends on signature type (RSA or DSA).
Note: RSA uses BOTH MD5 and SHA-1

Client replies with DH public value Ys only (p and g are already known by the Server ☺)

Curveball attack: signature protegge `p` e `g`?

dati g, p e (GIUSEPPE, 314218) certificate, cambio g,p con g', p' : g'^x mod p' = 314218 allora x ≡ 1,

g^x mod p (io vedo solo il valore) NON TUTTI i browser vedono binding g, p, g^x mod p

quindi non si controlla che la curva usato ≡ curva segnata con TLS.

Interlude: Entity authentication with asymmetric crypto

informal sketch, tricky details in Menezes, chapter 10.2

SKIP

Entity Authentication in a pubkey setting

➔**WRONG:**

⇨Show your certificate

⇨Why? ☺

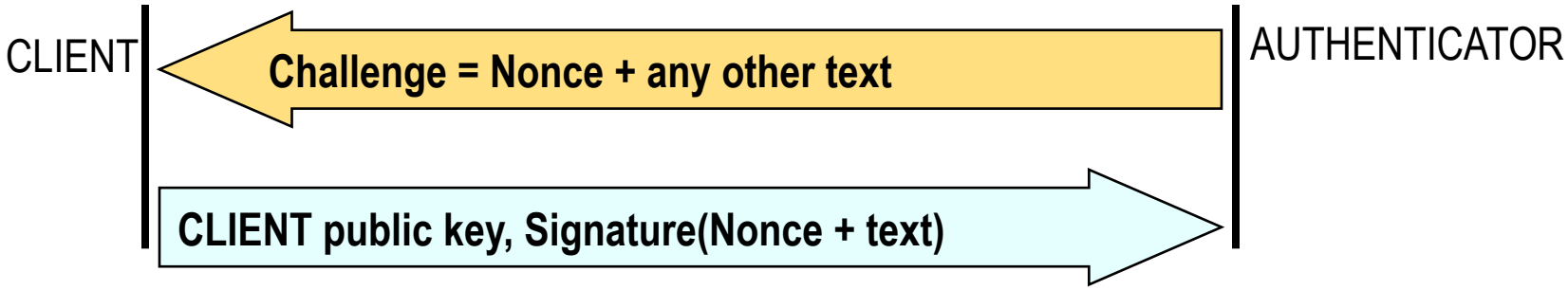
➔**RIGHT:**

⇨Prove you know the private key associated to your public key

➔**How to prove such knowledge?**

1) Use digital signature

skip (viste)

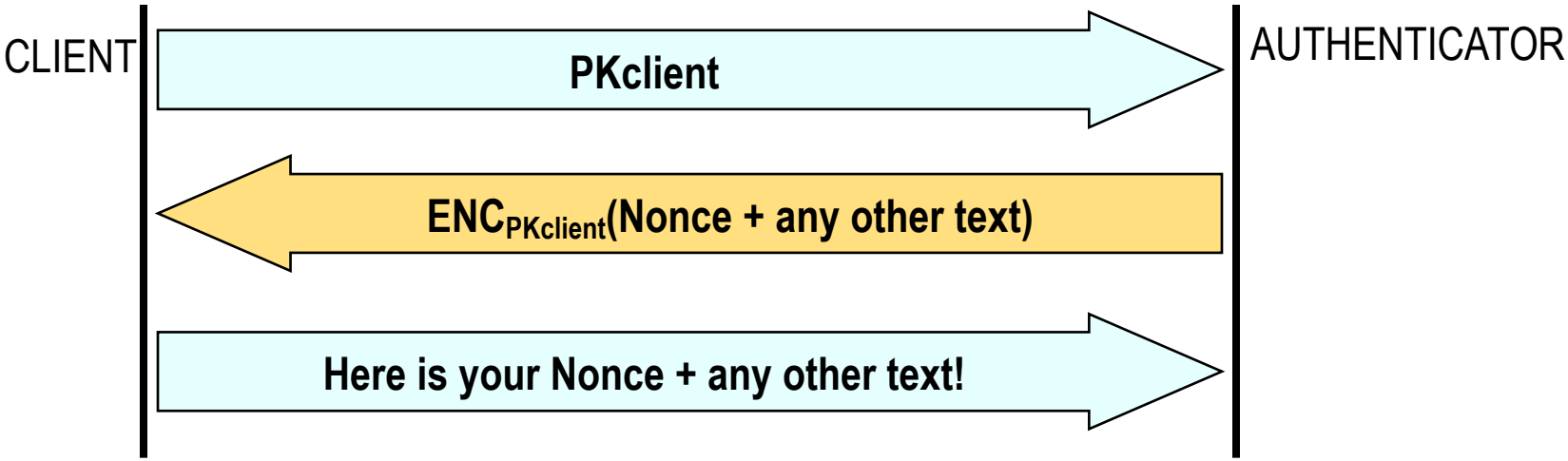


➔Rationale:

- ⇨I can sign only if I own the private key corresponding to a given public key
- ⇨of course, pubkey must be bound to client identity → certificate

2) Use encryption
(prove you can decrypt!)

viste

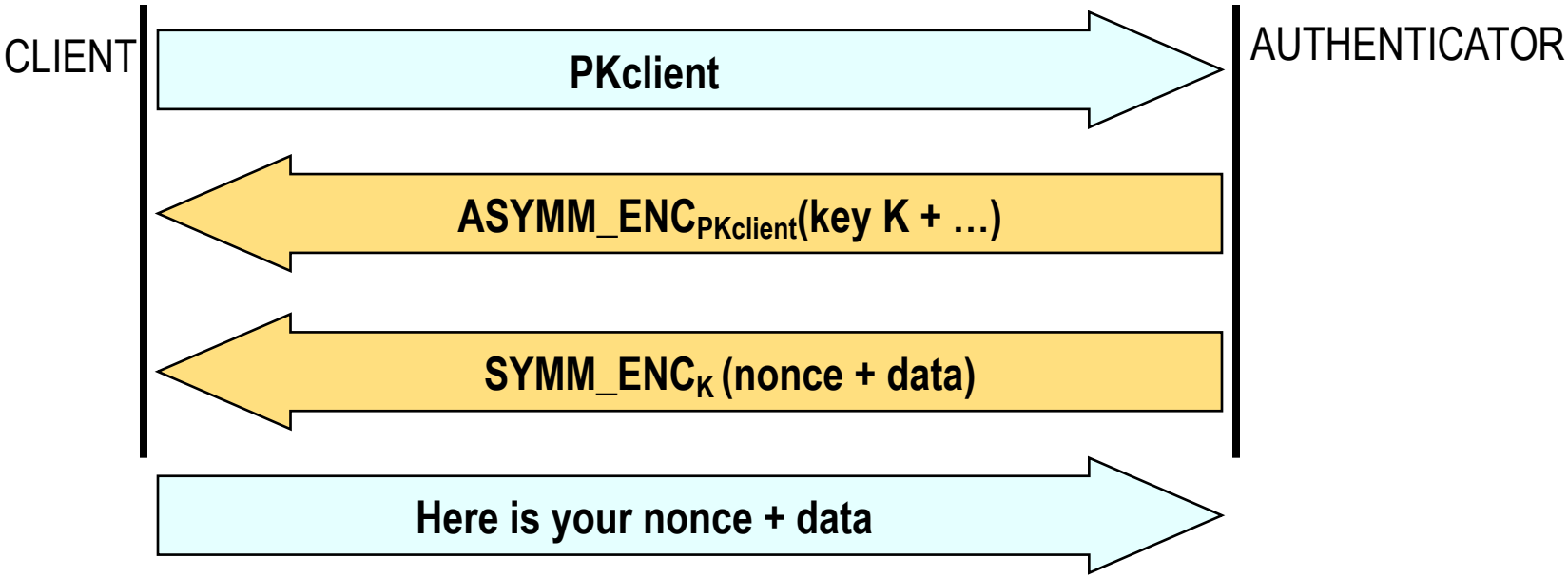


➔**Rationale:**

- ⇨I can decrypt only if I own the private key corresponding to **the public key you used to encrypt!**
- ⇨Same as before: pubkey = certificate

2bis) prove knowledge of transferred key

viste

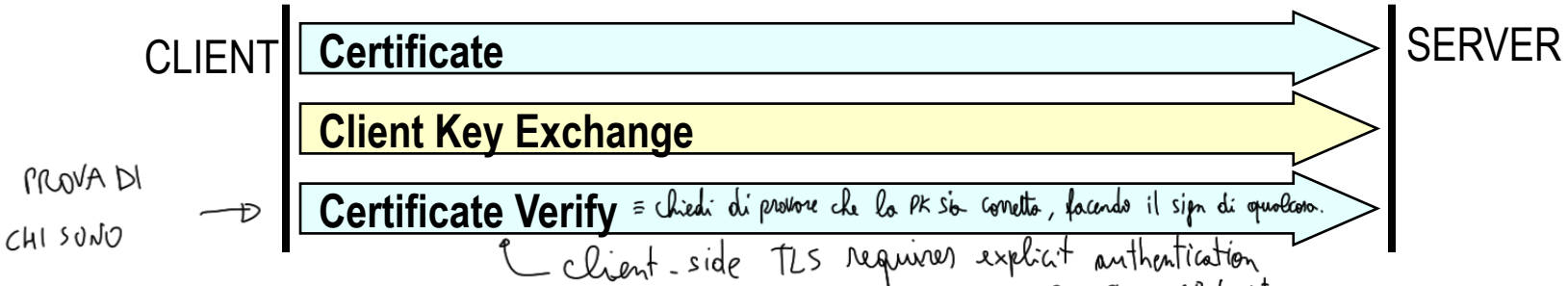


Back to TLS!
Entity authentication with asymmetric crypto

TLS, client side, richiede esplicita authenticate (diverso rispetto server side)

Handshake Phase 3 details

come fa client a dimostrare che è lui? Se mostra certificato non va bene (reply attack)



Dato (GB, PK_{GB})
- 1) invio msg a GB con PK_{GB} e GB decripta
- 2) GB sign qualcosa.

➔ **Certificate:**

⇨ Client certificate if requested

➔ **Client Key exchange**

⇨ Transmit encrypted symmetric (premaster) key or information to generate secret key at server side (e.g. Diffie-Hellman Ys)

➔ **Certificate Verify**

⇨ Signature of "something" known at both client and server

➔ ALL the messages exchanged up to now

» Which is not only known, but also useful! Allows to detect at an early stage (more later on this) tampering attacks (e.g. cipher suite downgrade)

⇨ To prove Client KNOWS the private key behind the certificate

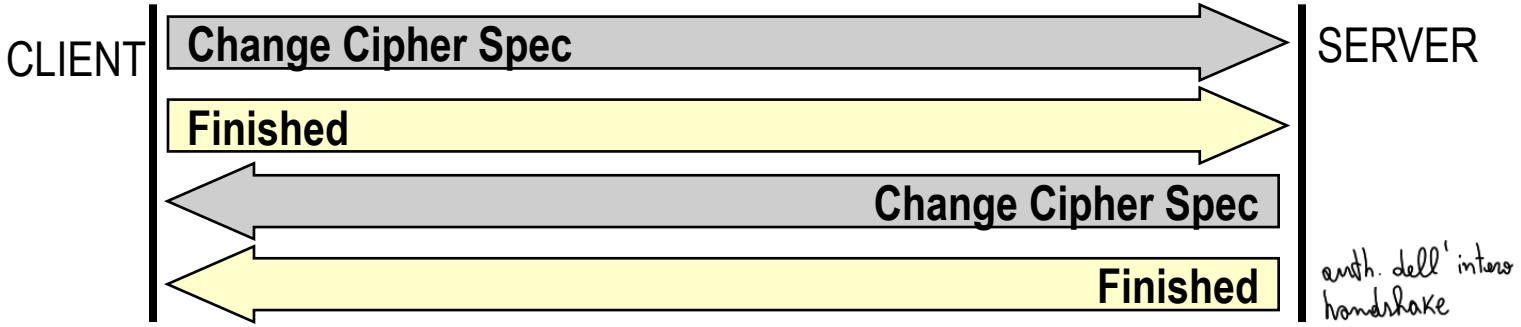
➔ Otherwise I could authenticate by simply copying a certificate ☺

A (smart) detail on certificate verify

- **Q: Why Certificate Verify does not immediately follows certificate?**
- **A: to include connection specific crypto parameters into signature!**
  - client & server random values
  - Explicit inclusion of master secret in SSLv3.0
    - » Since master secret can be computed only after the ClientKeyExchange message, Certificate Verify follows ClientKeyExchange
    - » Although this was abandoned in TLS1.x, but certificate verify still left as late as possible and hence after ClientKeyExchange

Phase 4

obbligatoria



➔ **Phase 4 has two fundamental goals**

- ⇨ Switch to the new security connection state
  - → Hence authentication and encryption based on keys computed with the exchanged security parameters
  - → Immediately applied to finished message
    - » First check that everything went OK! If Client and(orserver cannot decrypt finished message, something has gone wrong!

- ⇨ Authenticate all the previous handshake messages
  - → Finished = ~~digital signature~~ hash of all the previous handshake messages as transmitted and received by the peer up to now
  - → To avoid MITM tampering

➔ **Server authentication? Here as well!!!**

What if negotiation = encrypt but no auth?

➔**GAME OVER!**

my list

- AES_GCM (encryption only)
- STREAM_NULL

➔**Why? ☺**

da HandShake msg, modifico SOLO clientHello e negozio stream cipher.

Finished Client = H(LOG) ⊕ KeyStream, server fa H(LOG')

Finished Server: H(LOG') ⊕ KeyStream. Senza integrity ROMPO TUTTO (forbidden), senza poter fare downgrade attack.

Negoziazione attaccabile.

NON STUDIARE

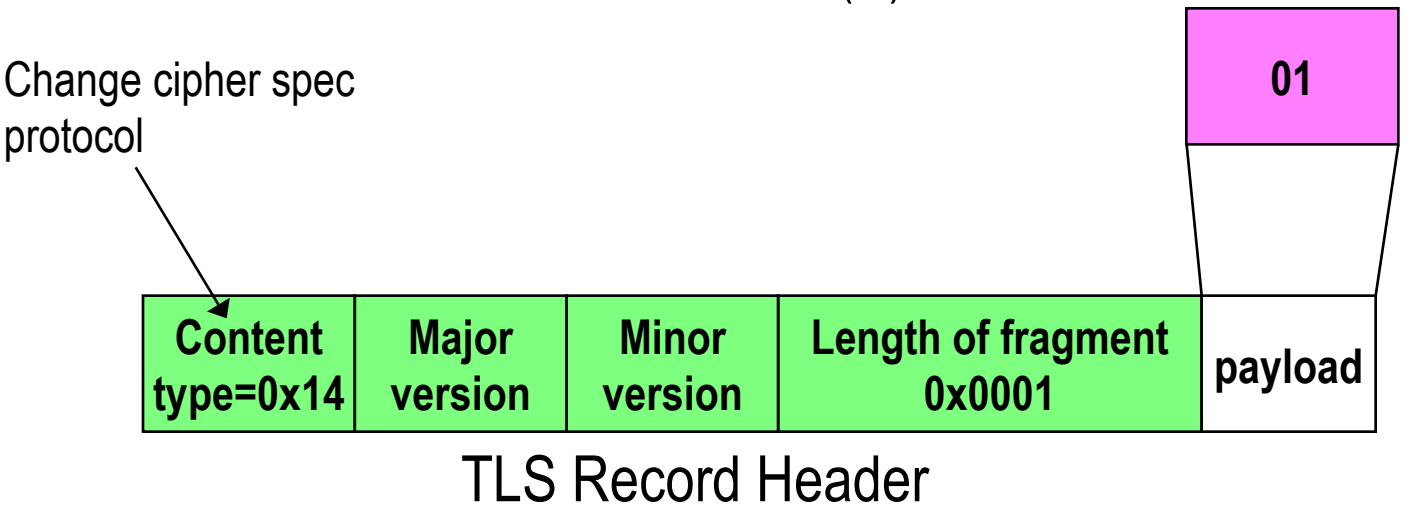
Change Cipher Spec

➔ **Defined as a separate protocol (!)**

➔ **Only one message:**

⇨ 1 single byte

⇨ Fixed content: constant value = 01 (!!!)



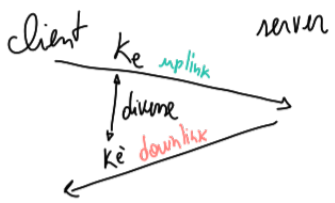
TLS Record Header

Why a separate protocol, and not part of the Handshake?

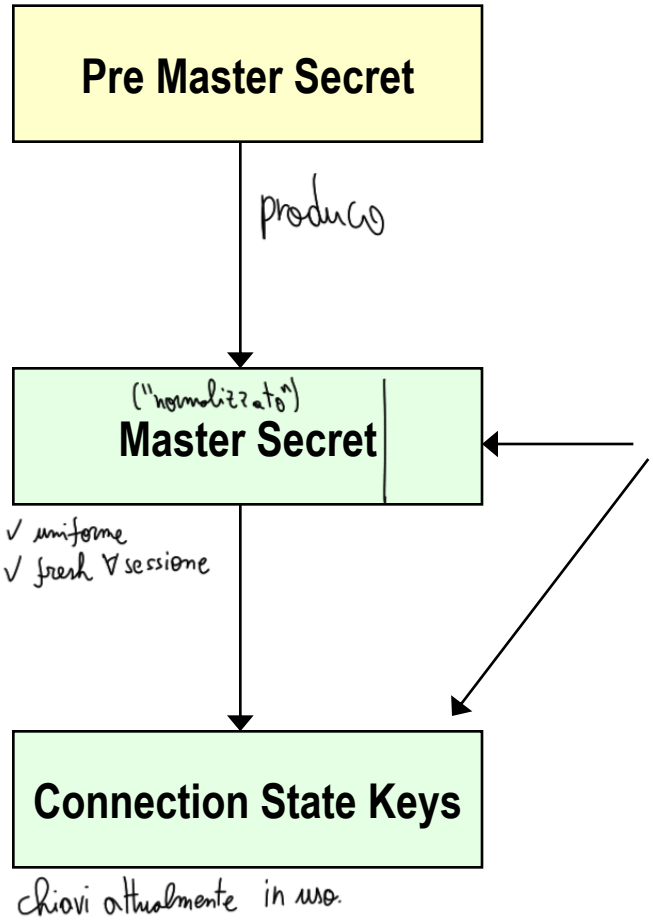
Wise choice! TLS specification allows aggregation of multiple messages of a SAME upper layer protocol into single TLS Record. How this possible with a Change Cipher Spec? (TLS Record is either ALL encrypted, or NOT encrypted)

TLS Key Computation

- Con RSA key transport, key selezionata da user;
- Con DH g^{xy} non è equamente distribuita (mod 8y → non avrò mai 90, 91...)
- operazioni extra: key computation



Secret hierarchy



Exchanged (RSA) or computed (DH) during handshake

- could be always the same value for same C-S pair (e.g., in the case fo fixed DH)

randomize, nel caso `DH` fosse "statico"

Master Secret computed from:

- Pre master
- random values from C & S
- timestamps from C & S

→ **Recomputed at session resumption**

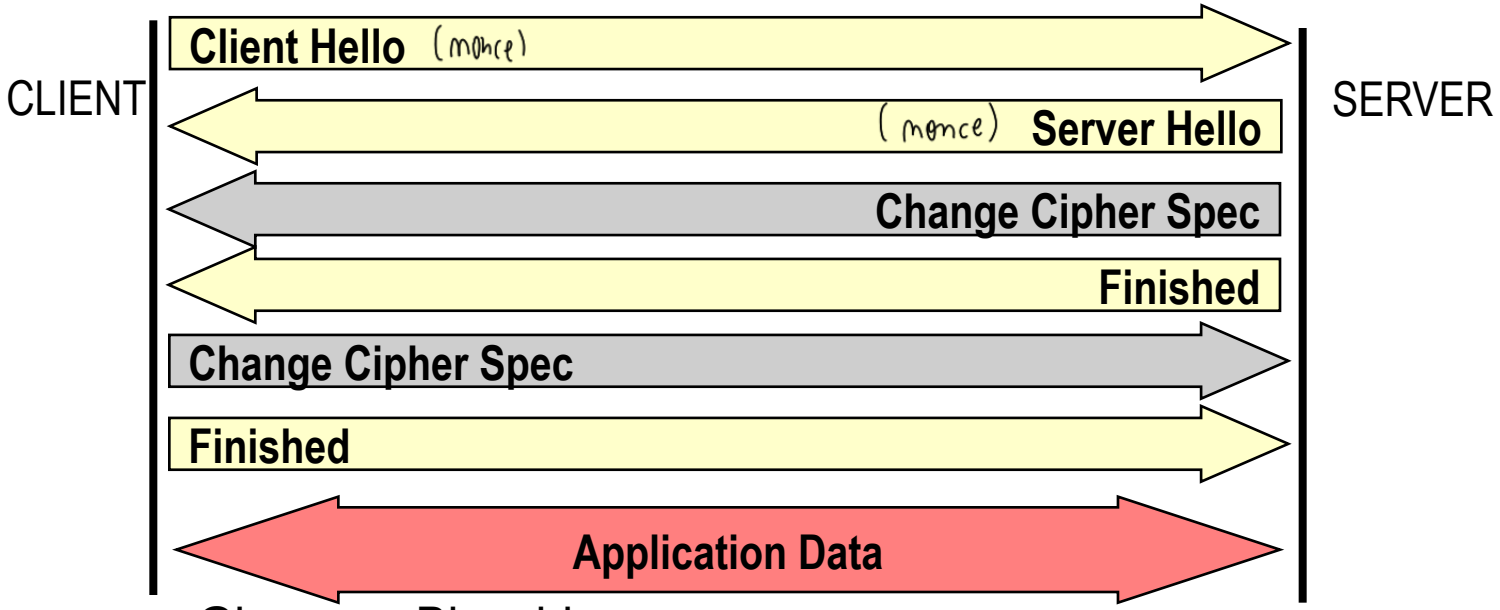
Up to 6 keys:

- encryption keys (write/read = C/S)
- authentication keys (write/read = C/S)
- initialization vector if needed (write/read)

→ **Recomputed at session resumption**

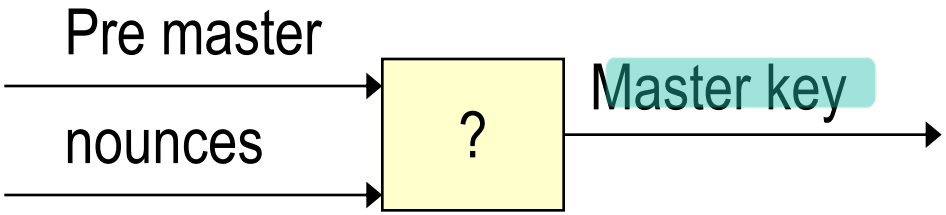
Abbreviated handshake (3-way) (session resumption)

- **Used to re-generate key material**
  - ⇨ For new connection
- **Avoids to reauthenticate peers**
  - ⇨ Done at start of session only
- **Avoid to re-exchange pre-master-secret**
- **Exchange new TS+random values**



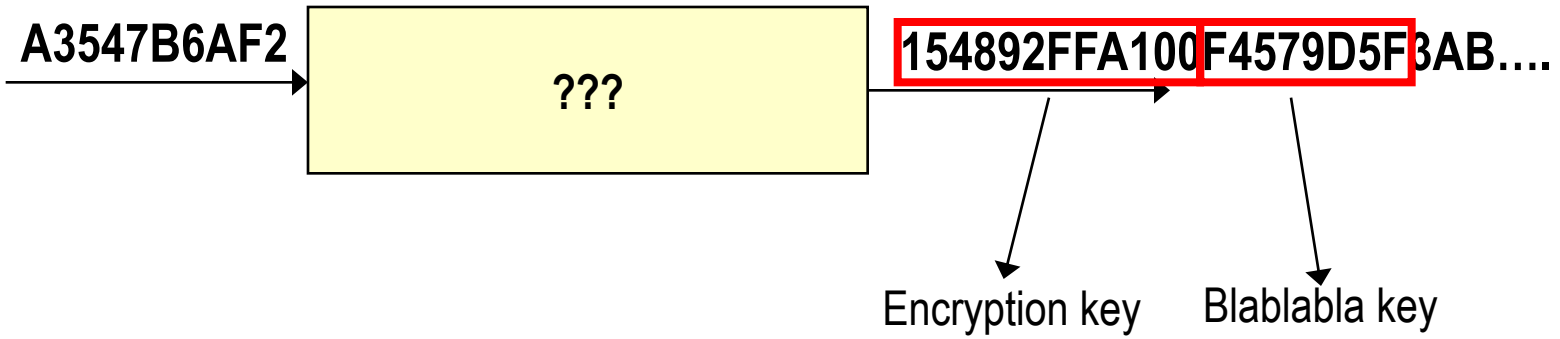
Initial keying, re-keying: Extract-then-expand

1) "Extract": Add randomness in key generation



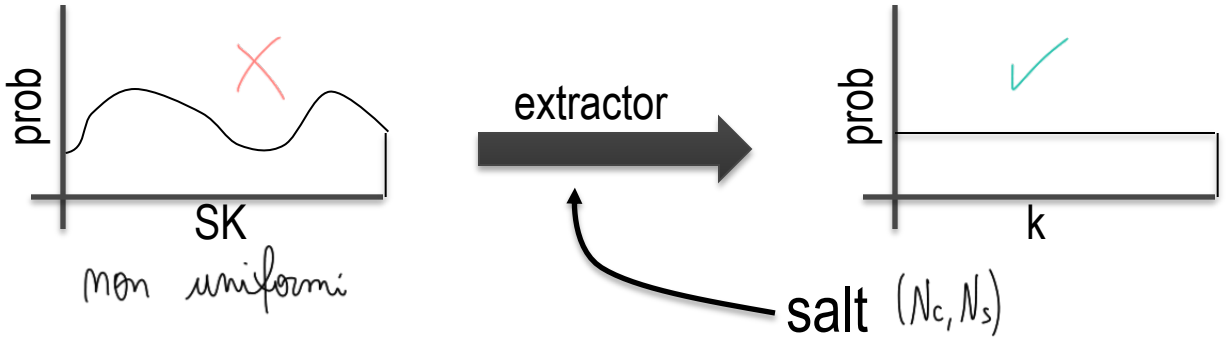
→ formisce in output: session keys

2) "Expand" initially limited secret to the needed amount of crypto material



Extract-then-Expand paradigm

Step 1: extract pseudo-random key k from source key SK (Key iniziale, g^{xy} o Secret Key, generate by client) non totally random



salt: a fixed non-secret string chosen at random

step 2: expand k by using it as a PRF key (Pseudo Random Function)

seed, K → PRF

Why extract?

➔**Constructions rely on the assumption that key k is random**

⇨Random = uniform distribution!

➔**What if this is not true?**

⇨Biases from HW random generator

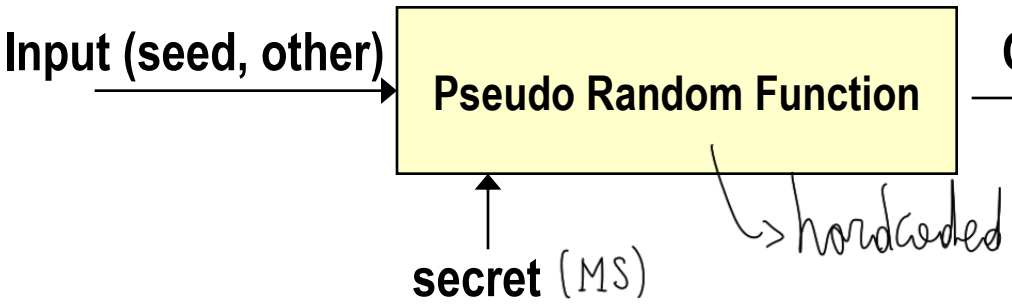
⇨Fixed DH exchange (!)

➔**Extract:**

⇨guarantees random master key

⇨“salts” it with nonces

Basic building block: PRF



TLS 1.2 usa
- PRF standard hash-based
  - hash negoziabile (default H = SHA-256)

⇒ allora "cipher suits" vlide devo parlare anche di negoziabile PRF

"TLS_DHE_RSA_WITH_AES_256_GCM_SHA384
PRF

➔ **Fundamental "brick":**
- ⇨ GOOD (and possibly fast!) PRF

➔ **PRFs in TLS**
- ⇨ SSL 3.0: not fully satisfactory PRF (broken)
- ⇨ TLS 1.0 and 1.1: good PRF but… hard-coded!
  - ➔ Construction used MD5 and SHA-1 (now weak)
  - ➔ *(unlike IPsec) TLS forgot that "good" in crypto is not forever….*
- ⇨ **TLS 1.2: negotiable PRF (at last ☺)**
  - ➔ **default one based on SHA-256**

(non studiare)

PRF from a 1-way hash: Expansion function P_{hash}

(≤ TLS 1.2)

- → **A0 = seed** = salt
- → **A1 = $HMAC_{hash}$(A0)**
- → **A2 = $HMAC_{hash}$(A1)**
- → **A3 = $HMAC_{hash}$(A2)**
- → **…**

msg

K

Key è key HMAC.

creo catena HMAC

Same secret used in all HMACs
Hash = chosen hash function

P_{hash} =

altro HMAC

$HMAC_{hash}(A_1 \mid seed)$	$HMAC_{hash}(A_2 \mid seed)$	$HMAC_{hash}(A_3 \mid seed)$	.....

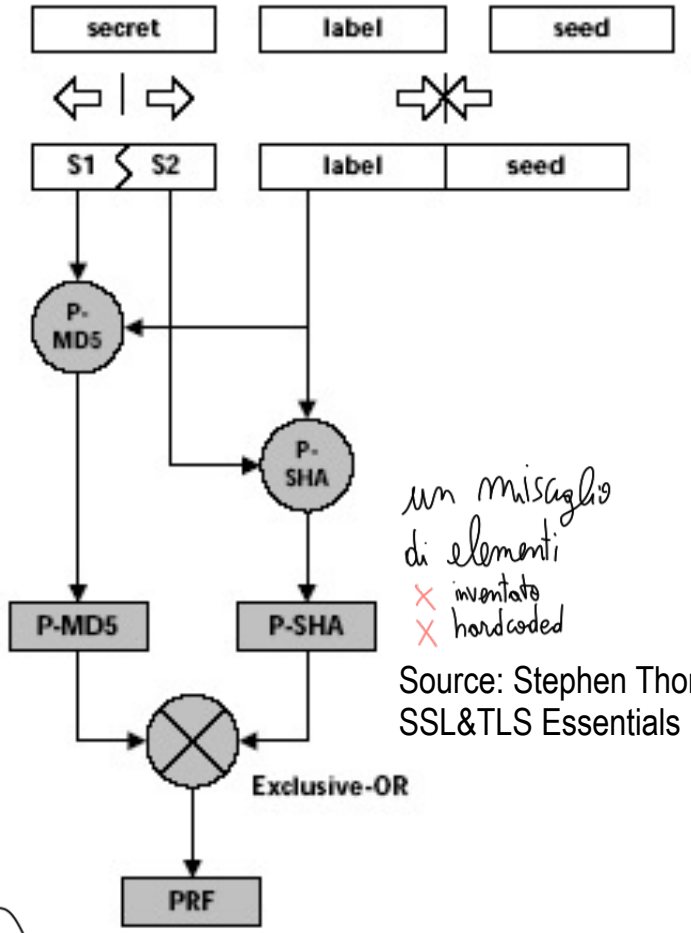
- → **P_{hash} function of:**
  - ⇨ Chosen hash function
  - ⇨ Secret
  - ⇨ seed
- → **P_{hash} size: any (unlike HMAC size)**

Posso avere ∞ len, per via della catena.
Ai crittografi non piacciono le catene, non ho security properties.
~> si para a HKDF. Questo non è stato dimostrato che sia rotto, ma il creatore di HMAC suppone di sì!

PRF until TLS 1.1: hard coded!

➔ **Employs two hash algorithms**
- ⇨ Idea: combine both MD5 and SHA-1
- ⇨ Greater robustness! Must break both…
  - → Indeed ☺

➔ **Input:**
- ⇨ Desired length
- ⇨ Secret
  - → Assume even size, split it in two parts (S_1,S_2)
- ⇨ Seed
- ⇨ Label (an ascii string)



Source: Stephen Thomas, SSL&TLS Essentials

➔ **PRF(scrt, lbl, sd) = P_{MD5}(lbl | sd) XOR P_{SHA-1}(lbl | sd)**
- ⇨ MD5 uses S1 as secret
- ⇨ SHA-1 uses S2 as secret

TLS 1.2: PRF from P_{hash}

PRF(secret, label, seed) = P_{hash}(label|seed)

➔ **TLS1.2:**

- ⇨ Default: this construction with SHA-256
  - → 32 bytes hash
- ⇨ But also negotiable PRF
  - → Hence possibly very different and not based on HMAC

➔ **Example: Master secret generation**

- ⇨ Inputs:
  - → Secret = Pre-master-secret
  - → Seed = Nonces (client-random & server-random)
- ⇨ Further input:
  - → Label = string "master secret"
- ⇨ Computation: first 48 bytes of:
  - → **PRF(pre-master-secret, "master secret", Client-Random | Server-Random)**

Key generation

➔ **Master secret [48 bytes]:**

⇨ PRF(pre-master-secret, “master secret”, Client-Random | Server-Random)

➔ **Key Block [size depends on cipher suites]:**

⇨ PRF(master-secret, “key expansion”, Server-Random | Client-Random)

➔ **Individual keys:**

⇨ Partition key block into up to 6 fields in the following order:

⇨ Client MAC, Server MAC, Client Key, Server Key, Client IV, Server IV

PRF used also in computation of finished message instead of “normal” MD5 or SHA-1 Hash. E.g. For client finished message:

PRF(master-secret, “client finished”, MD5(all handshake msg) | SHA-1(all handshake msg)) [12 bytes]

PRF in TLSv1.3: HKDF

Key Derivation Function

↳ correct construction

➔ **HMAC-based Key Derivation Function** (e' counter mode)

- ⇨ H. Krawczyk 2010, provably secure
- ⇨ New name: KDF → precisely states what this tool is used for
- ⇨ Standardized: RFC 5869
- ⇨ Included in newer TLS version (1.3 ≥)

- Start from master key K (extracted from HMAC)
- use context string (e.g. TLS' label | nonces) to distinguish (eventual) multiple usage of K

HKDF= $HMAC_{hash, K}$(context string|0) $HMAC_{hash, K}$(context string|1) …

counter = seed

secret

no chain

cambiando input → cambio output

cambio "il seed"

ma non e' numero, ma stringa, mi da possibilita' di cambiare la funzione