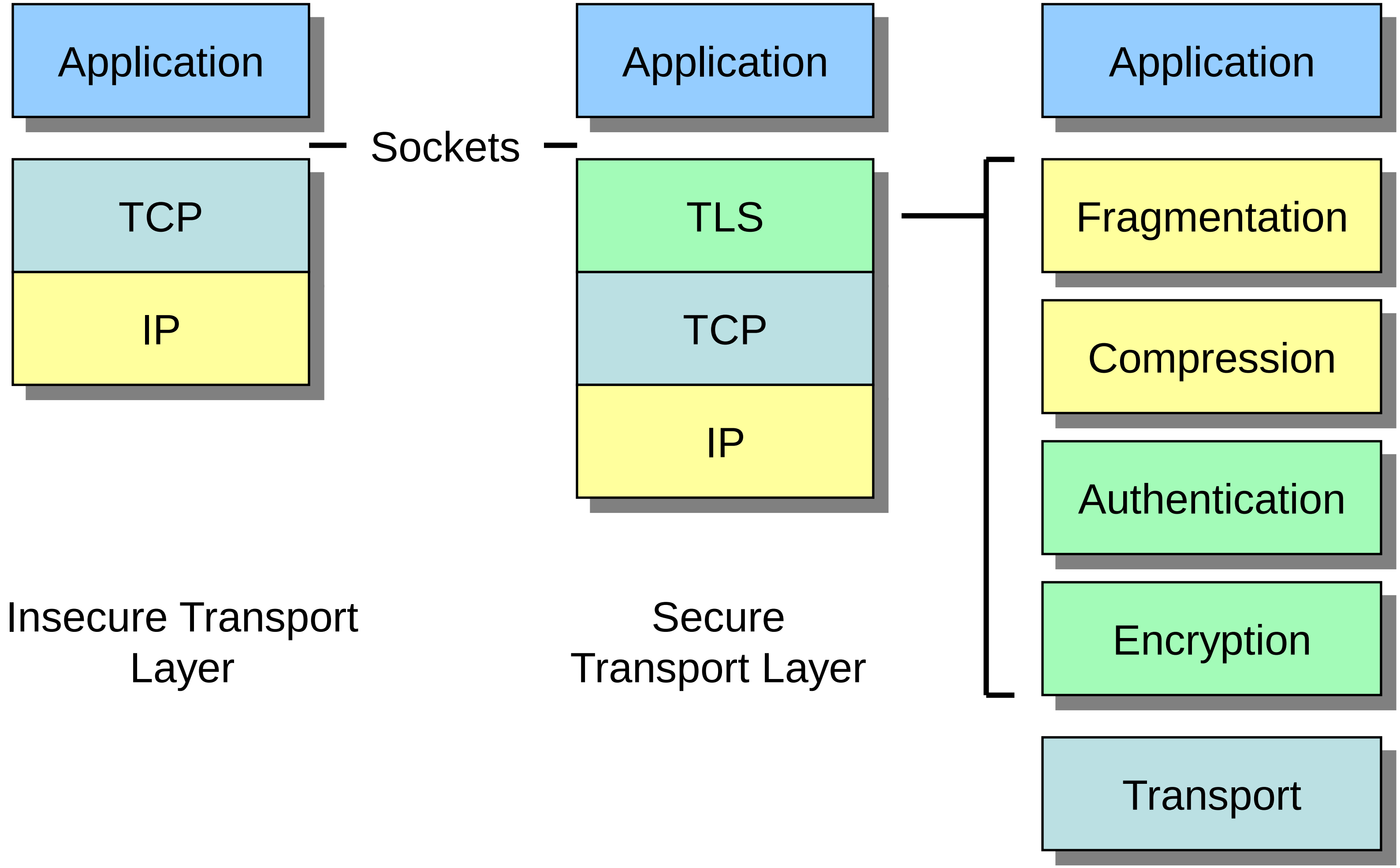
SSL / TLS

PPKE, ITK
Csapodi Márton

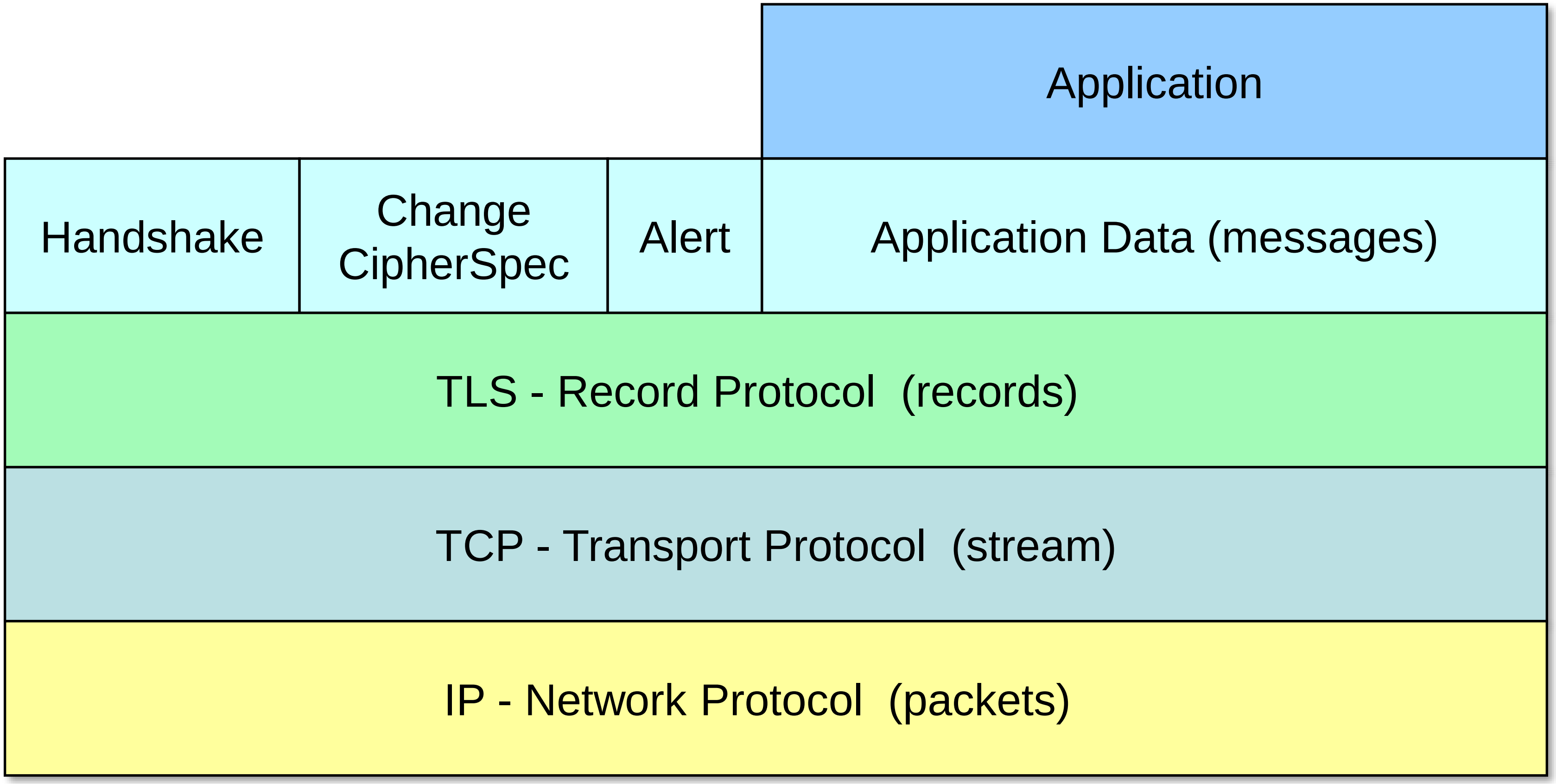
Secure Network Protocols for the OSI Stack

Communication layers	Security protocols
Application layer	ssh, S/MIME, PGP, Kerberos, WSS
Transport layer	TLS, [SSL]
Network layer	IPsec
Data Link layer	[PPTP, L2TP], IEEE 802.1X, IEEE 802.1AE, IEEE 802.11i
Physical layer	Quantum Cryptography

SSL/TLS Protocol Layers

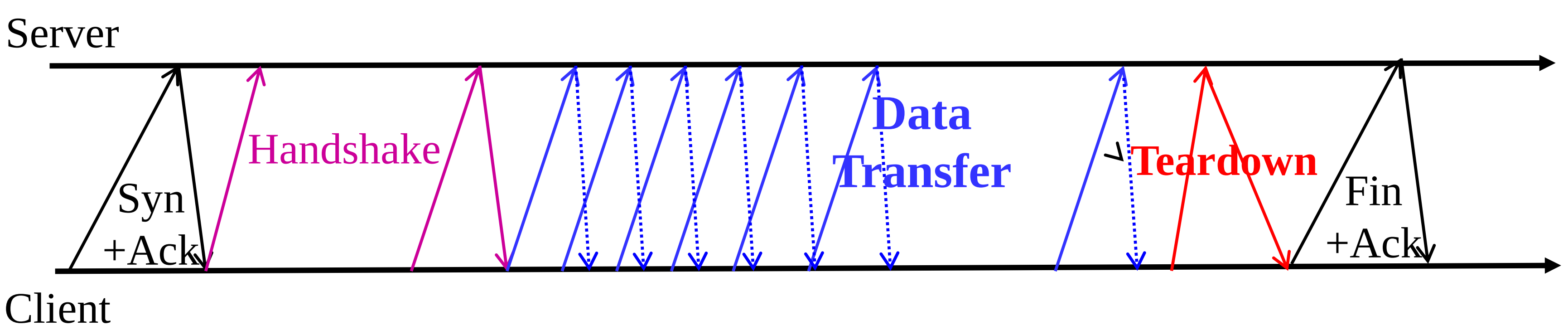


SSL/TLS Protocol Layers



SSL/TLS Operation Phases (high level)

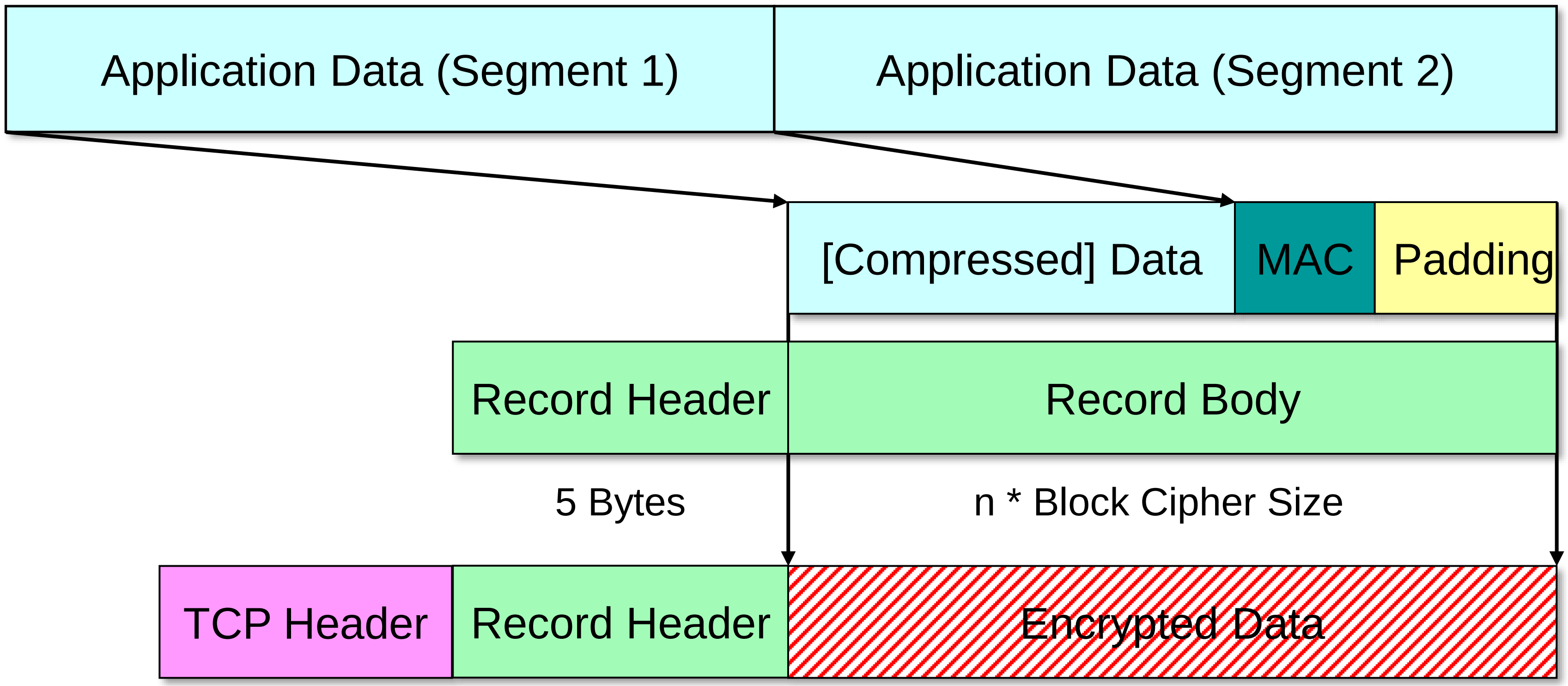
- TCP Connection setup (Syn+Ack)
- Handshake (key establishment)
  - Negotiate (agree on) algorithms, methods
  - Authenticate server and optionally client, establish keys
- Data transfer
- Secure Teardown
- TCP connection closure (Fin+Ack)



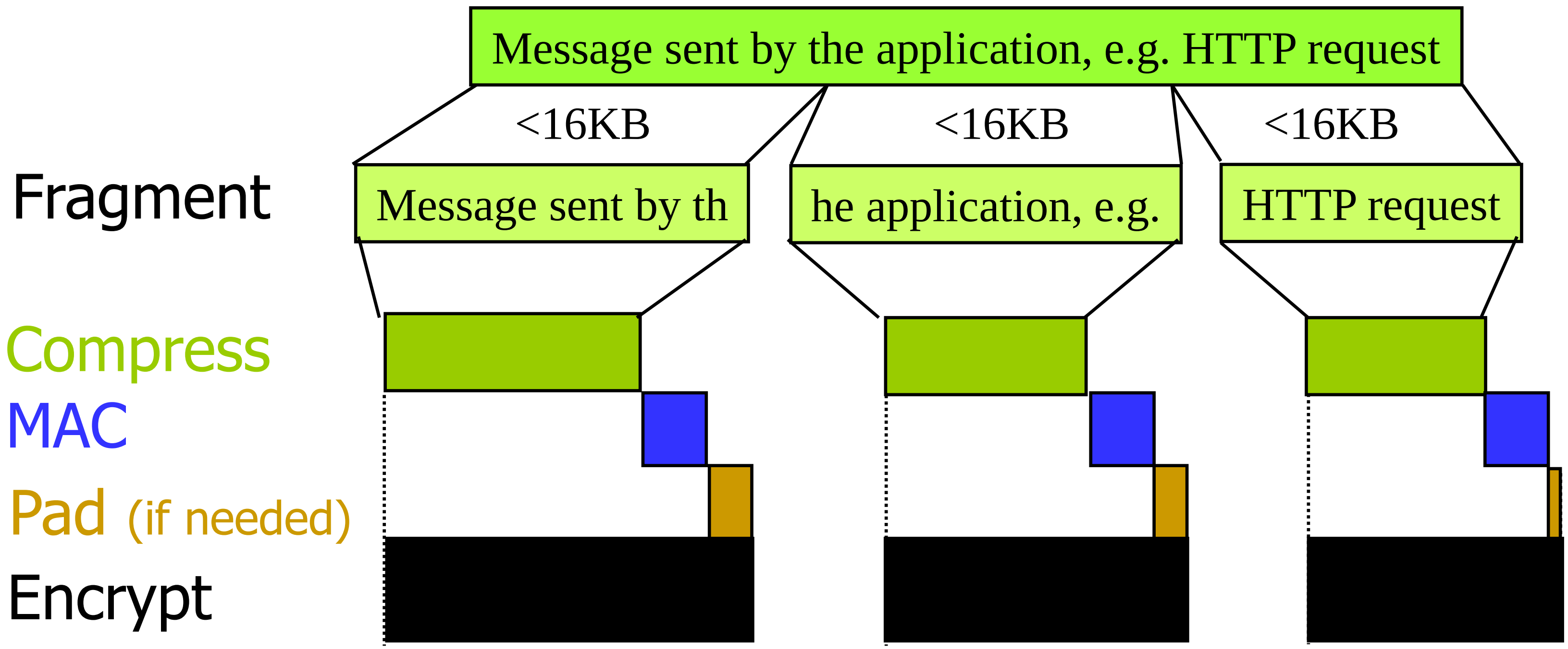
Data transfer: Record Protocol

- Assumes underlying reliable communication (TCP)
- Four services (in order):
  - Fragment: break TCP stream into fragments (<16KB)
    - Pipeline: send processed frag 1 while processing 2 and receiving 3
  - Compress (lossless) each fragment
    - Reduce processing, communication time
    - Ciphertext cannot be compressed – must compress before
    - Risk: exposure of amount of redundancy → *compression attacks*
  - Authenticate: [seq#||type||version||length||comp_fragment]
  - Encrypt
    - After padding (if necessary)
- Finally, add header: type (protocol), version & length

Fragmentation, compression, authentication, encryption



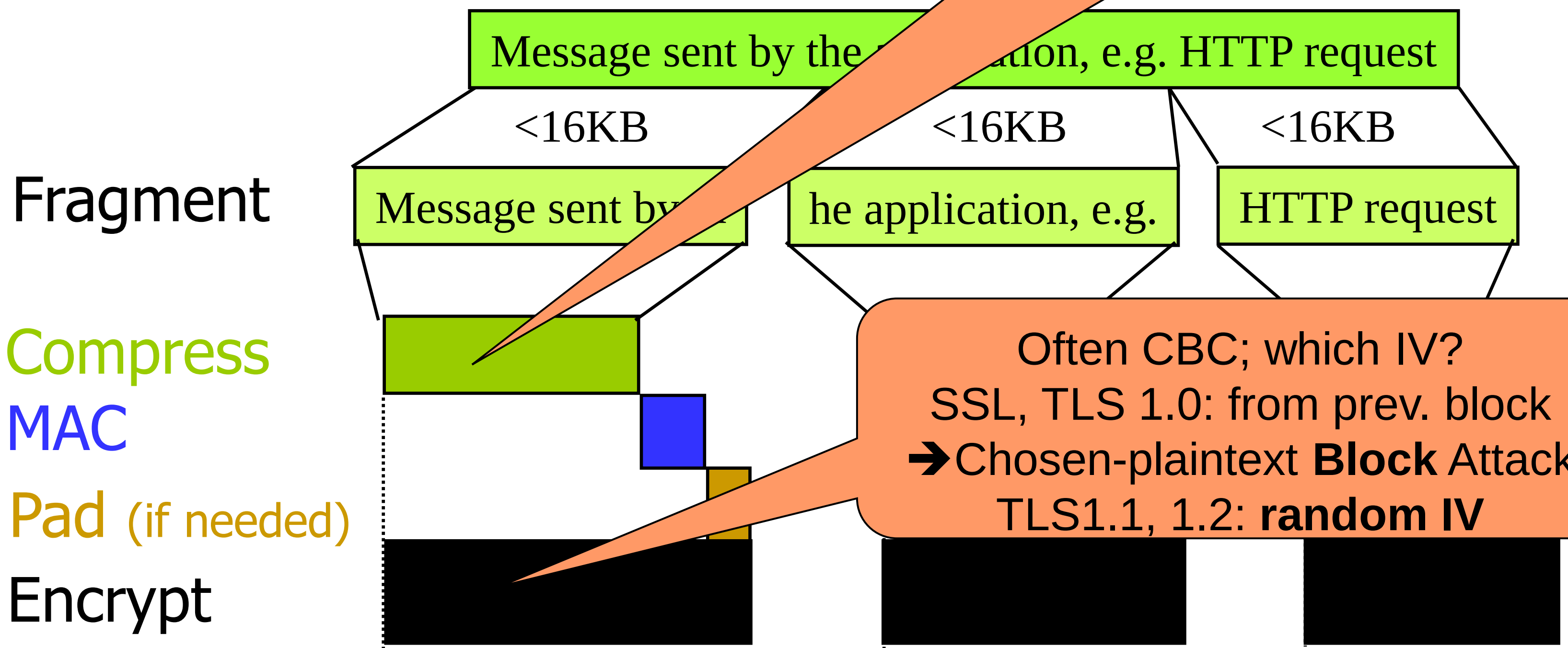
Fragmentation, compression, authentication, encryption



Send each block via TCP

Fragmentation, c… authentication, …

Fragment then Compress:
simpler - but revealing ?
TLS1.1,1.2: pad to fixed-lengths
to hide **exact** length
Exploited: CRIME, TIME attacks



Often CBC; which IV?
SSL, TLS 1.0: from prev. block
➔Chosen-plaintext **Block** Attack
TLS1.1, 1.2: **random IV**

Send each block via TCP

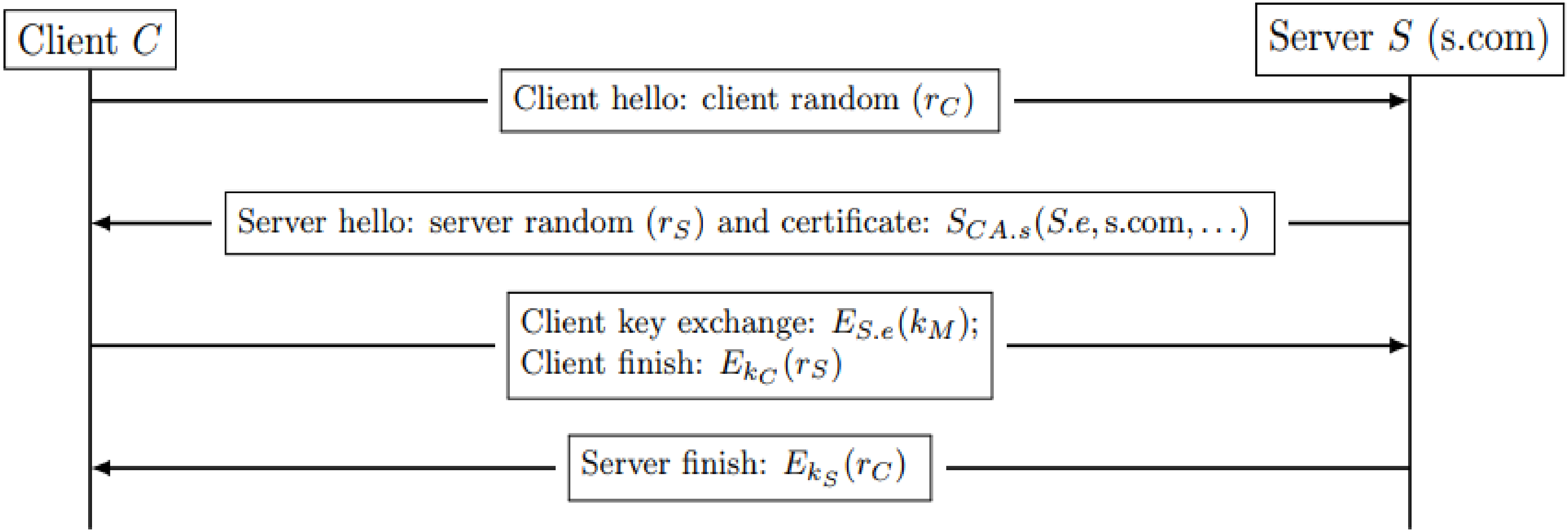
Vulnerabilities

- Surprisingly many found, exploited!
- ➔ SSL, TLS1.0: vulnerable record protocol:
  - Attacks on RC4 → to be avoided
  - CBC IV reuse in session (BEAST)
  - MAC-then-encrypt: padding attacks (Lucky13, POODLE)
  - Compress-then-encrypt: CRIME, TIME
  - downgrading to use vulnerable version
  - etc.

SSL/TLS Handshake Protocol

- The beginning: SSLv2
  - SSLv1 was never published, released
- The evolution: from SSLv3 to TLS 1.2
  - TLS: the IETF version of SSL
- State-of-Art: TLS 1.3
  - Significant changes
- Our focus is on the handshake protocol

Simplified SSLv2 Handshake



- Key derivation in SSLv2:
  - Client randomly selects k_M and sends to server
  - Client and server derive (directional) encryption keys:

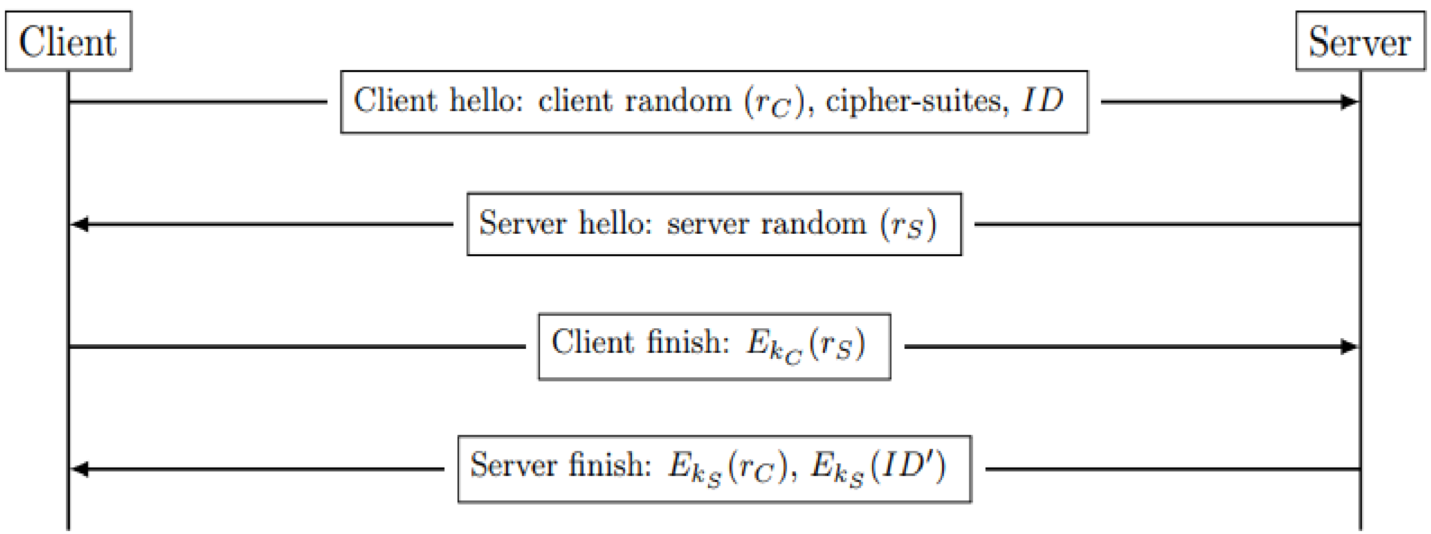
$$k_C = MD5(k_M || \text{“0”} || r_C || r_S) \qquad k_S = MD5(k_M || \text{“1”} || r_C || r_S)$$

SSLv2: important concepts

- Derive, from master key k_M, two separate keys:
  - k_C, for protecting traffic from client to server
  - k_S, for protecting traffic from server to client
  - Nonces r_C, r_S protect against replay
    - Even if client reuses same PK encryption of k_M
- Sessions: reusing public-key operations
- Cipher-agility
- Optional client authentication

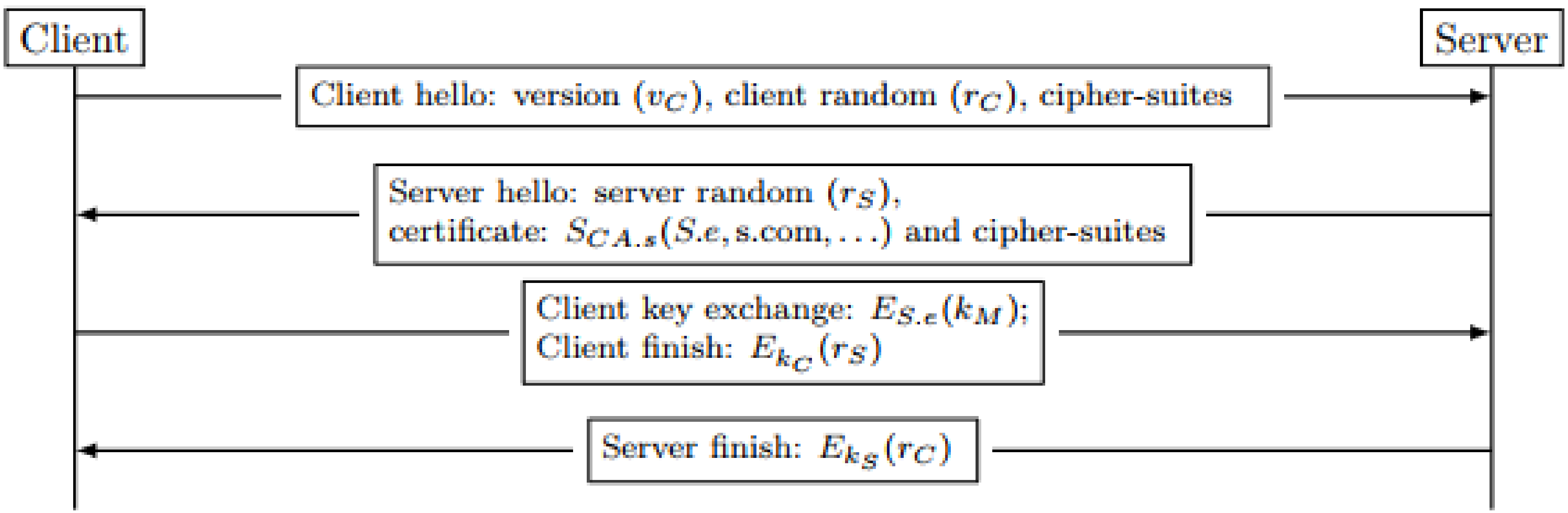
SSLv2 Session Resumption

- Goal: cache shared master key k_M (and *ID*)
  - By both client and server
  - Client identifies cached key by sending *ID* (if known)
  - If server knows *ID*, it sends only nonce (no cert)
  - Server sends (new) identifier *ID'* at end of handshake



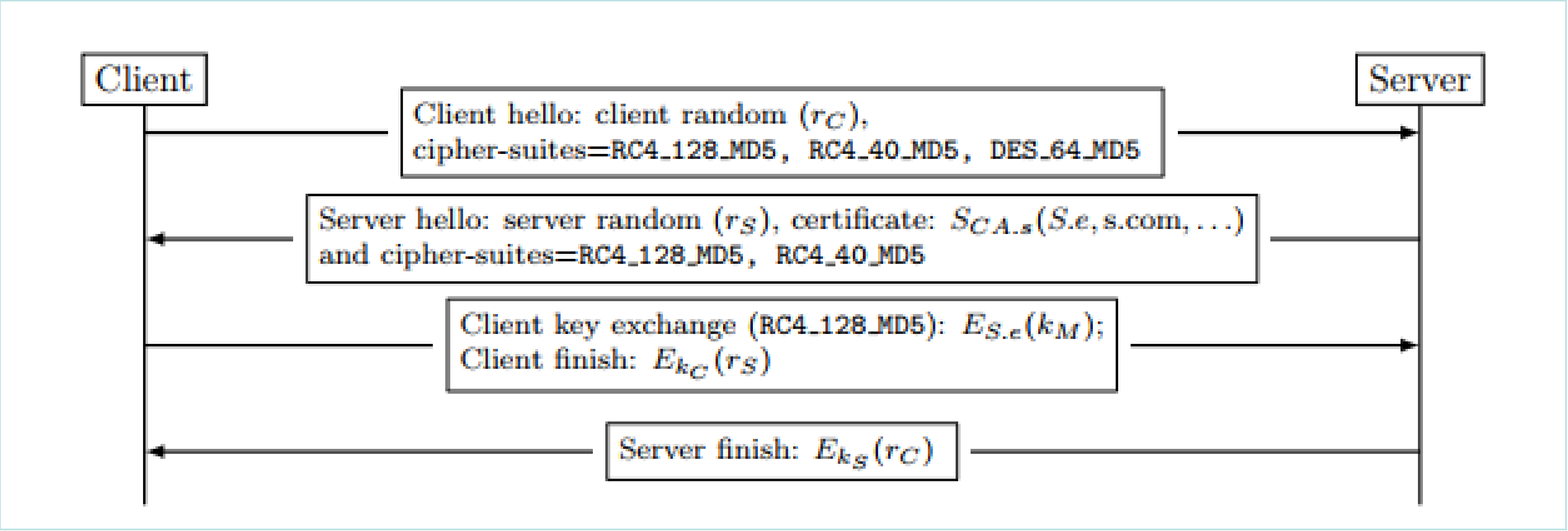
SSLv2 Ciphersuite Negotiation

- Client, server sends cipher-suites
- Client specifies choice in client-key-exchange



SSLv2 Ciphersuite Negotiation

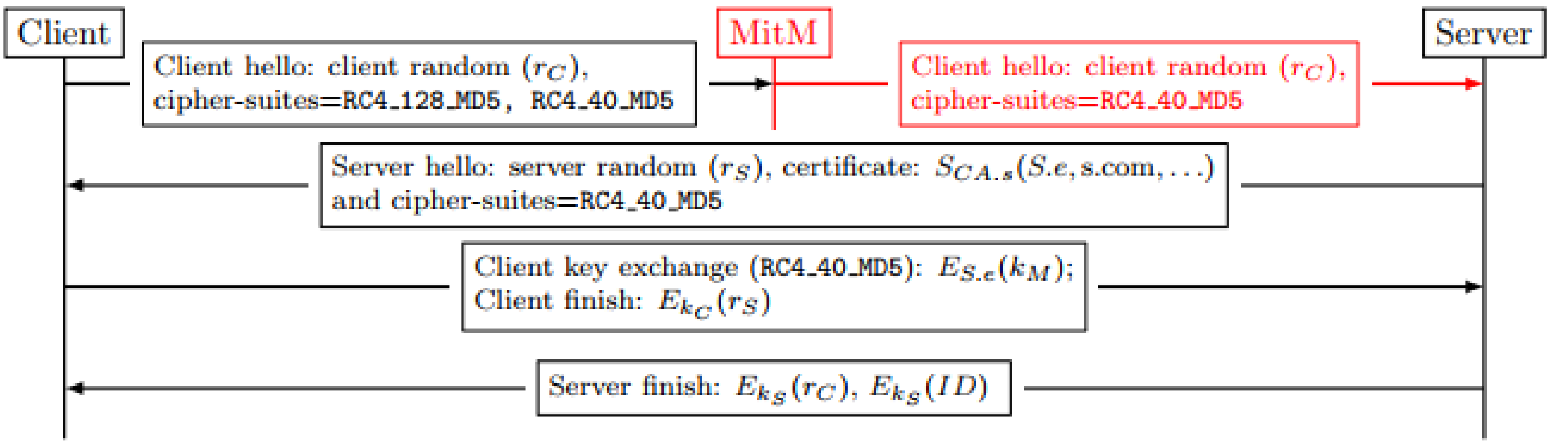
- Client, server sends cipher-suites
- Client specifies choice in client-key-exchange



- Example: RC4_128_MD5 chosen
- Vulnerable to downgrade attack!

SSLv2 Downgrade Attack

- Server and client tricked into using (insecure) 40-bit encryption (`export version')

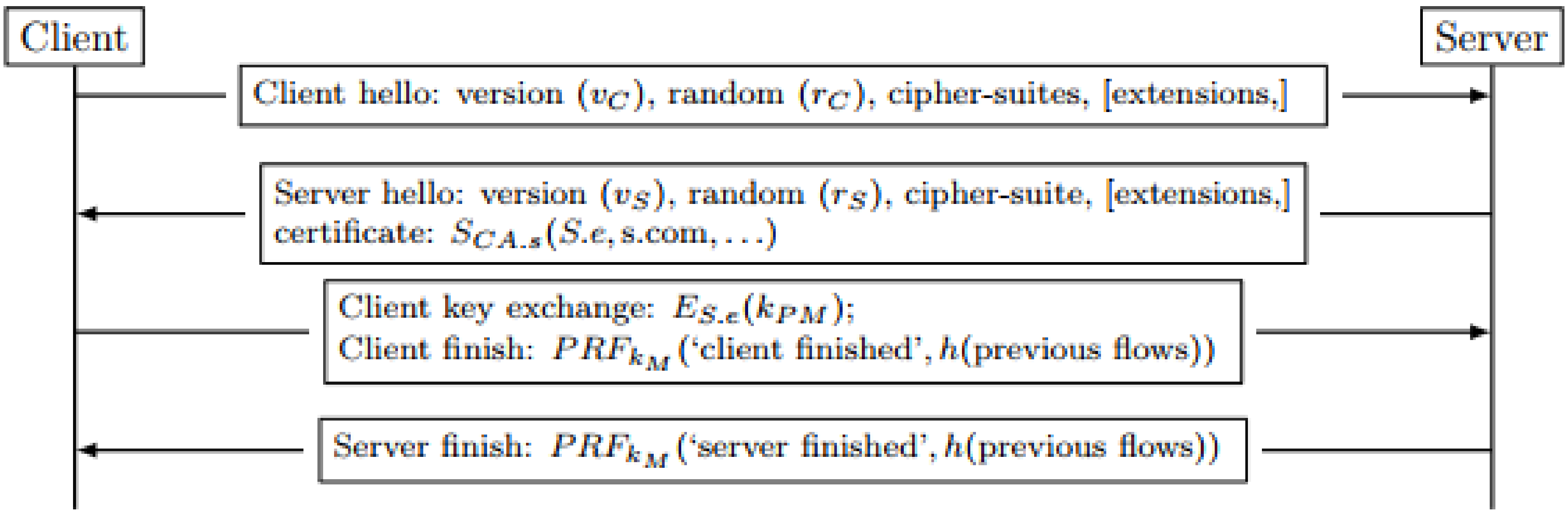


- Attacker may record connection and decrypt later – no need for real-time cryptanalysis!

The evolution: SSLv3, TLS1.0, 1.1, 1.2

- Main improvements:
  - Improved key derivation
    - Premaster key → master key → connection keys
  - Improved negotiation and handshake integrity
    - Prevents SSLv2 downgrade attack
    - Secure extensions, protocol-negotiation & more
  - DH key exchange and PFS (perfect forward secrecy)
    - SSLv2 allowed only RSA; TLS 1.3: only PFS
  - Session-ticket resumption

Basic RSA Handshake: SSL3-TLS1.2



$$k_M = PRF_{k_{PM}}(\text{“master secret”}\|r_C\|r_S)$$

$\textit{key-block} = PRF_{k_M}(\text{‘key expansion’}\|r_C\|r_S)$					
k_C^A	k_S^A	k_C^E	k_S^E	IV_C	IV_S

SSL3-TLS1.2: Key Derivation

- Handshake exchanges premaster key
- Derive master key (PRF: pseudo random function):

$$k_M = PRF_{k_{PM}}(\text{“master secret”} \| r_C \| r_S)$$

  - In case premaster key is not (fully) random
    - Weak randomness at a (weak) client
    - Weak client reuses same PK-encrypted key
    - DH-derived premaster key

SSL3-TLS1.2: Key Derivation

- Handshake exchanges premaster key
- Derive master key:

$$k_M = PRF_{k_{PM}}(\text{“master secret”} \| r_C \| r_S)$$

- Derive key block from master key:

$$key\text{-}block = PRF_{k_M}(\text{‘key expansion’} \| r_C \| r_S)$$

- Chop keys from key-block (A: authentication, E: encryption):

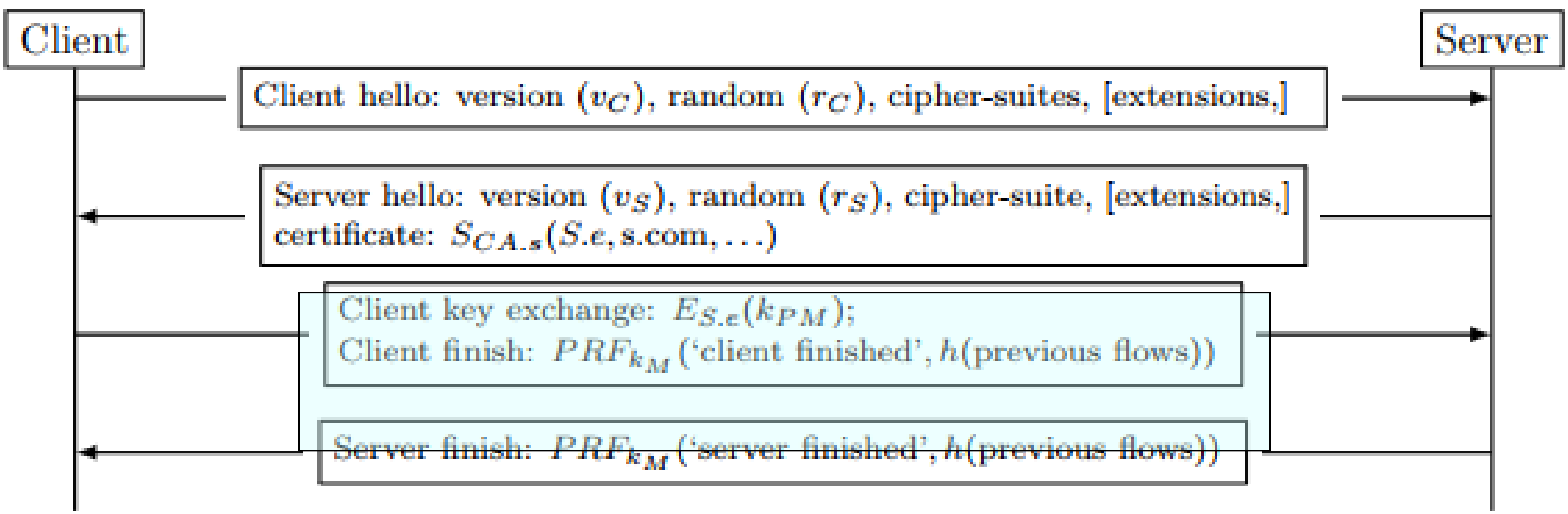
$key\text{-}block = PRF_{k_M}(\text{‘key expansion’} \| r_C \| r_S)$					
k_C^A	k_S^A	k_C^E	k_S^E	IV_C	IV_S

SSL3-TLS1.2: Agility and Integrity

- SSLv2: limited cipher-agility (ciphersuites)
  - And no integrity: vulnerable to downgrade attack
- SSLv3 to TLS1.2: integrity + improved agility:
  - Handshake integrity – foils downgrade attack!
  - Backwards compatibility
  - TLS extensions
  - Version-dependent key separation

SSL3-TLS1.2: Handshake integrity

- Foils the downgrade attack on SSLv2
- Extend the finish-message validation: authenticate entire previous handshake flows



SSL3-TLS1.2: Backward compatibility

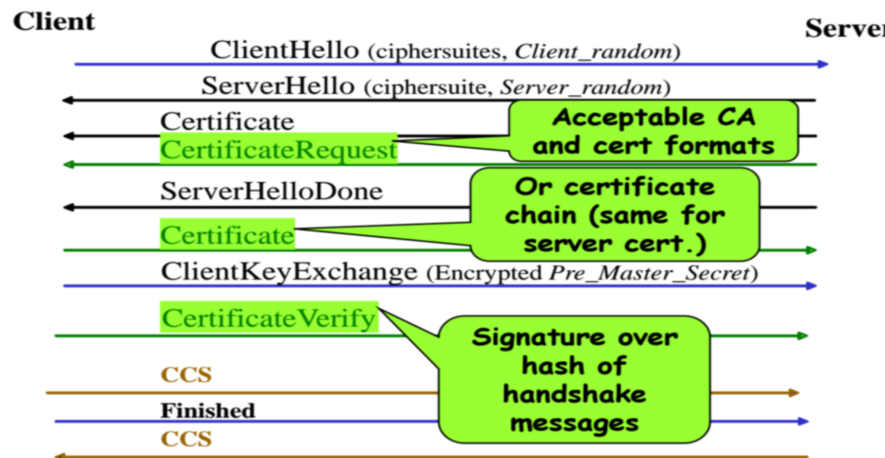
- Challenge: upgrading existing protocol
  - Unrealistic: all upgrade at same day
  - Backward compatibility: new (server, client) can still work with old (client, server)
    - Server selects version based on client's (in 'hello')
    - Downgrade prevented using 'finish' authentication
- Dilemmas for clients:
  - Some servers fail to respond to new handshake
  - 'Downgrade-dance' clients: try new versions, then older → vulnerable!

Advanced Handshake Features

- Client authentication
- Perfect Forward Secrecy (PFS)
  - ephemeral Diffie-Hellman keys
- Session resumption (ID-based, ticket)
- TLS 1.3 handshakes

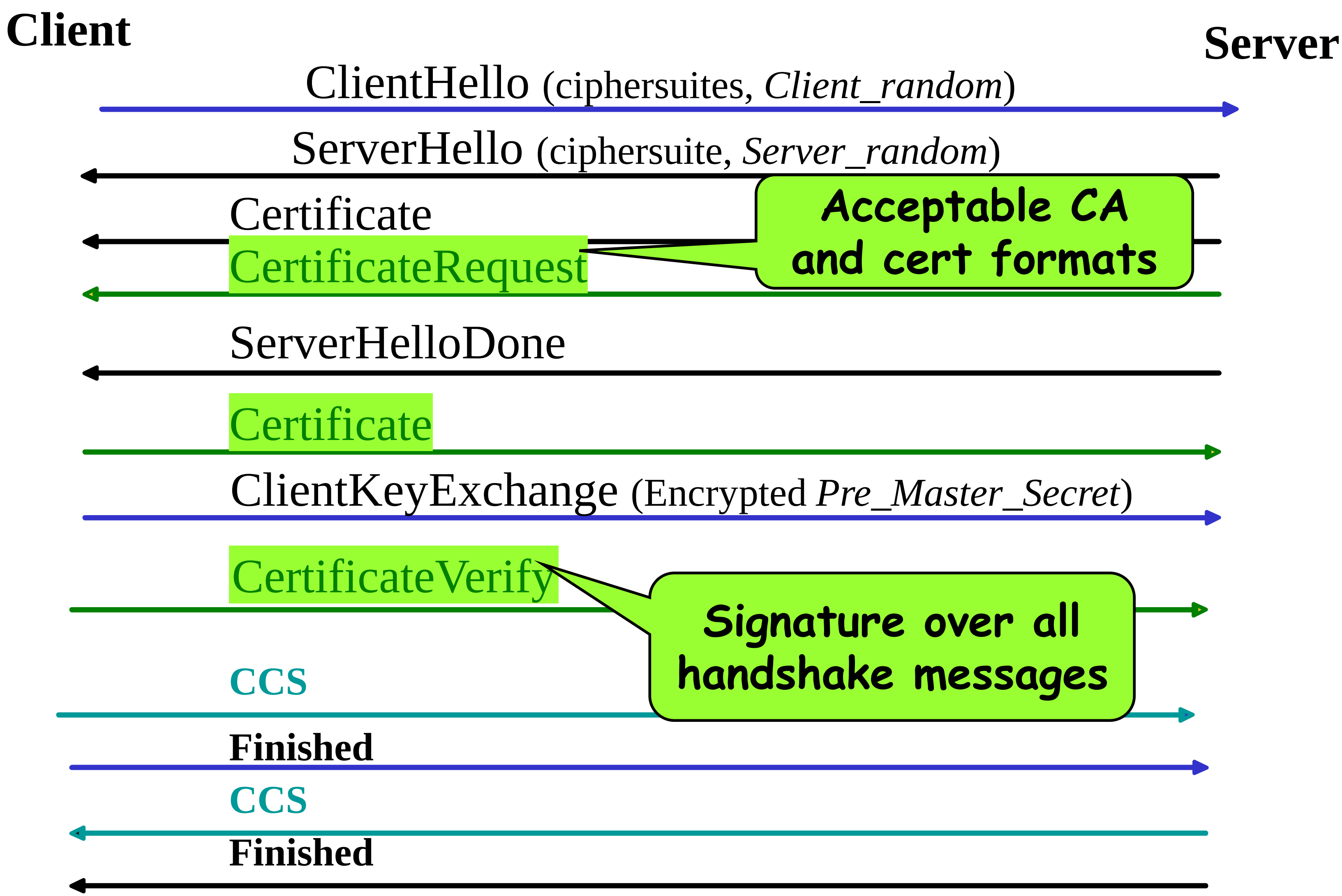
TLS/SSL Client Authentication

- Usually, TLS/SSL used only with server PK
  - Only allows client to authenticate server
  - Client authentication: encrypt secret (pw, cookie)
- But TLS/SSL also allows client certificates
- How?
  - Client authenticates by signing with certified PK



- Easy – no PW!
- But: PKI challenges, device dependency
- ➔ Limited use, mainly within organization/community

TLS/SSL Client Authentication



SSL Client Authentication: Issues

Which identifier?

- No global, unique namespace
- Result: each server use its own client names, certificates

Support for mobility of cert and key…

- Smartcard, USB `stick`?

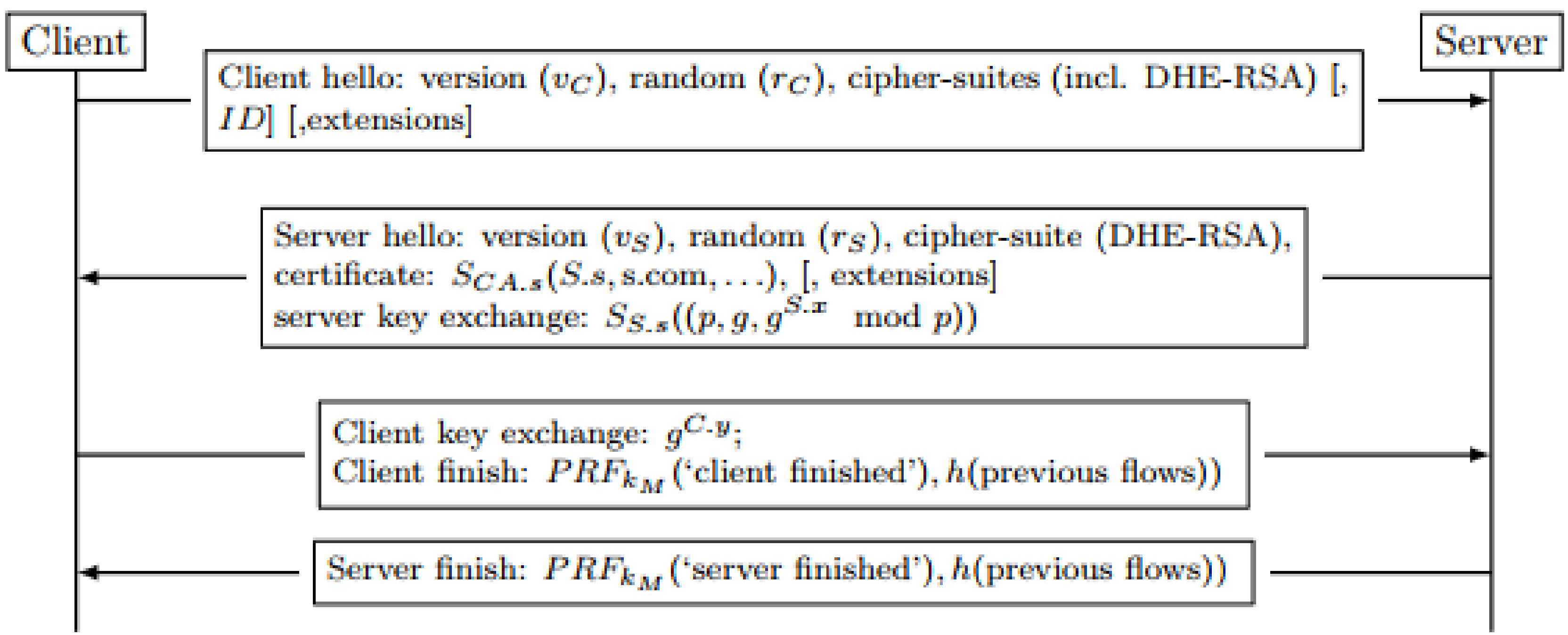
➔ Rarely used

Ephemeral Diffie-Hellman keys

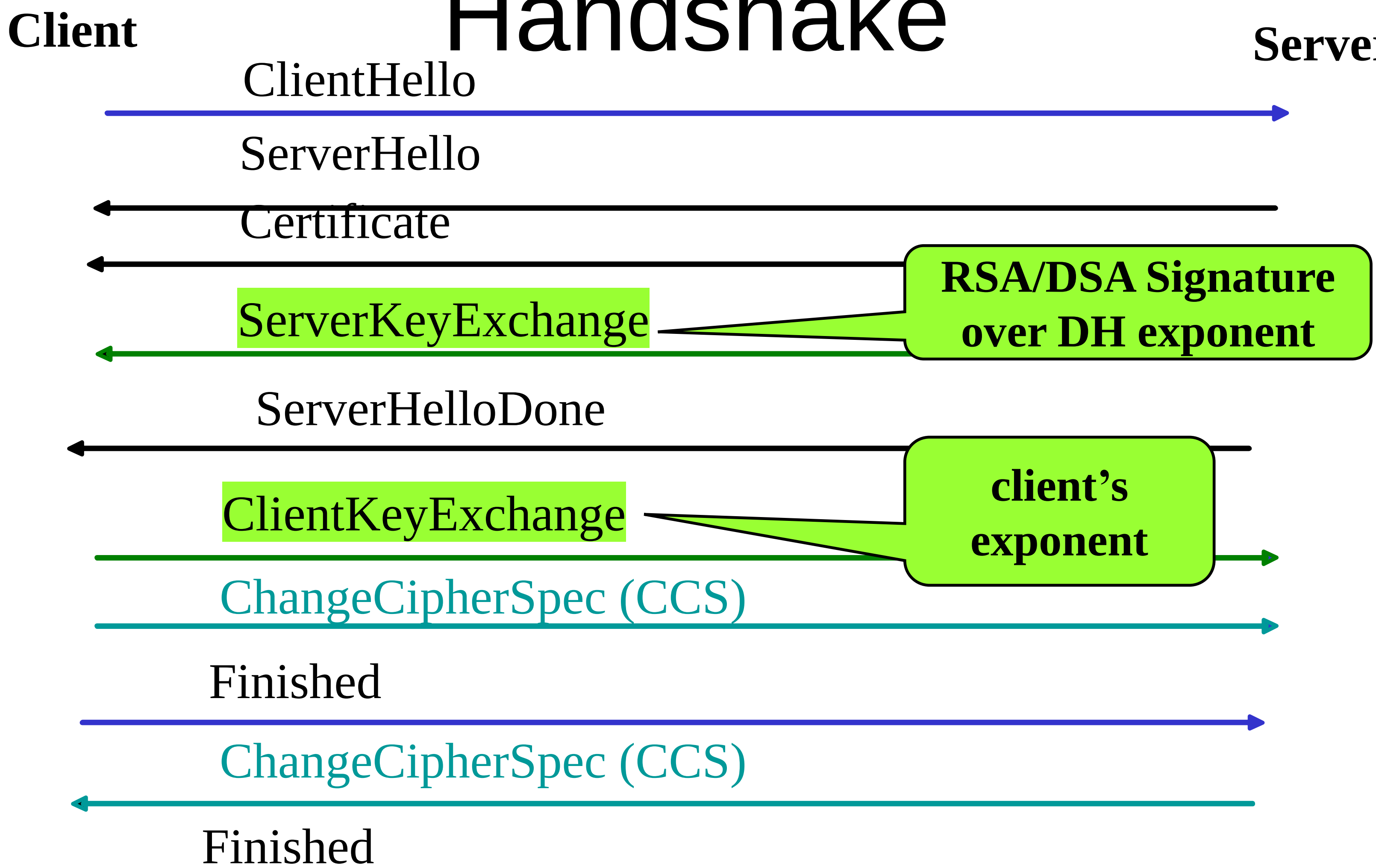
- Ephemeral keys: per-connection
  - Per-connection public keys ? Why?
- Motivations?
  - Perfect forward security: present traffic immune from future exposure – incl. of past keys
  - Historical: ‘export-grade’ (weak) keys (512 bit RSA)
- How?
  - Diffie-Hellman key exchange
  - Authenticated using long-term keys

TLS/SSL Handshake: Ephemeral DH

- Server signs a DH exponent $g^{S.x}$
  - E.g., using RSA signatures

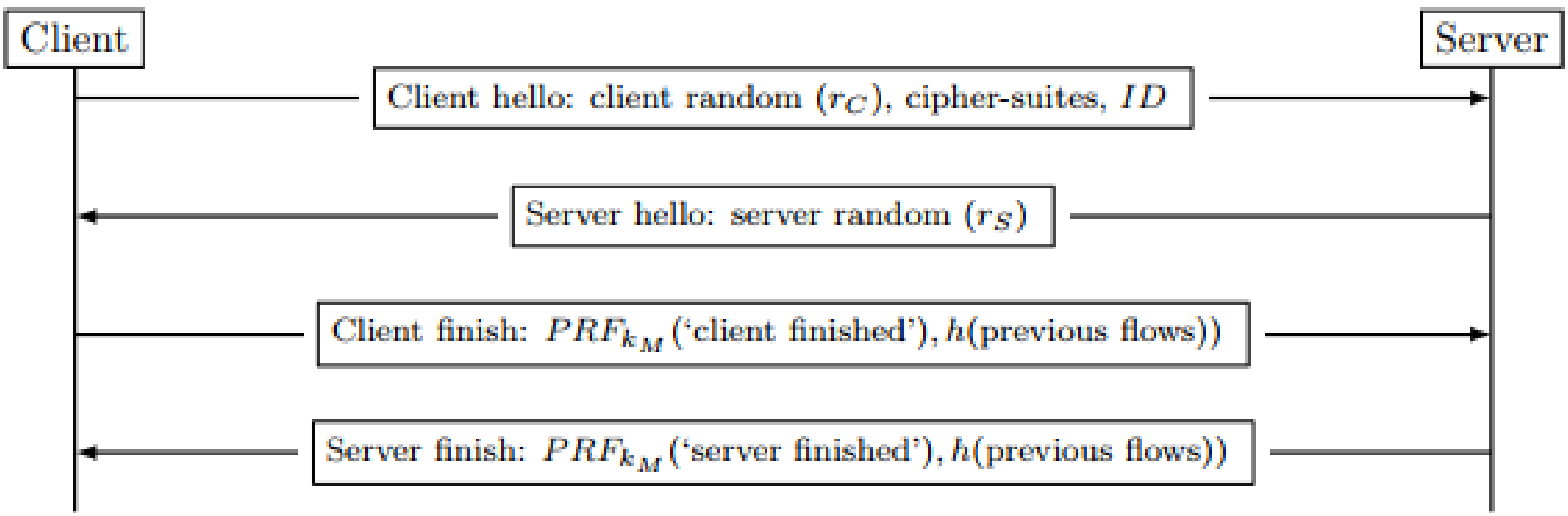


TLS/SSL Ephemeral PK Handshake



ID-based Session Resumption

- Idea: server, client store (ID, key) per peer
- Reuse in new connections btw same pair
- Saves PK operations (CPU, BW)



Session-ID Resumption Handshake

Client **Server**

ClientHello (cipher-suites, resume(**session_id**), *Client_random*) →

← ServerHello (Chosen cipher-suite, **session_id**, *Server_random*)

← ChangeCipherSpec (CCS)

← Finished (Confirmation -MAC of handshake messages)

ChangeCipherSpec (CCS) →

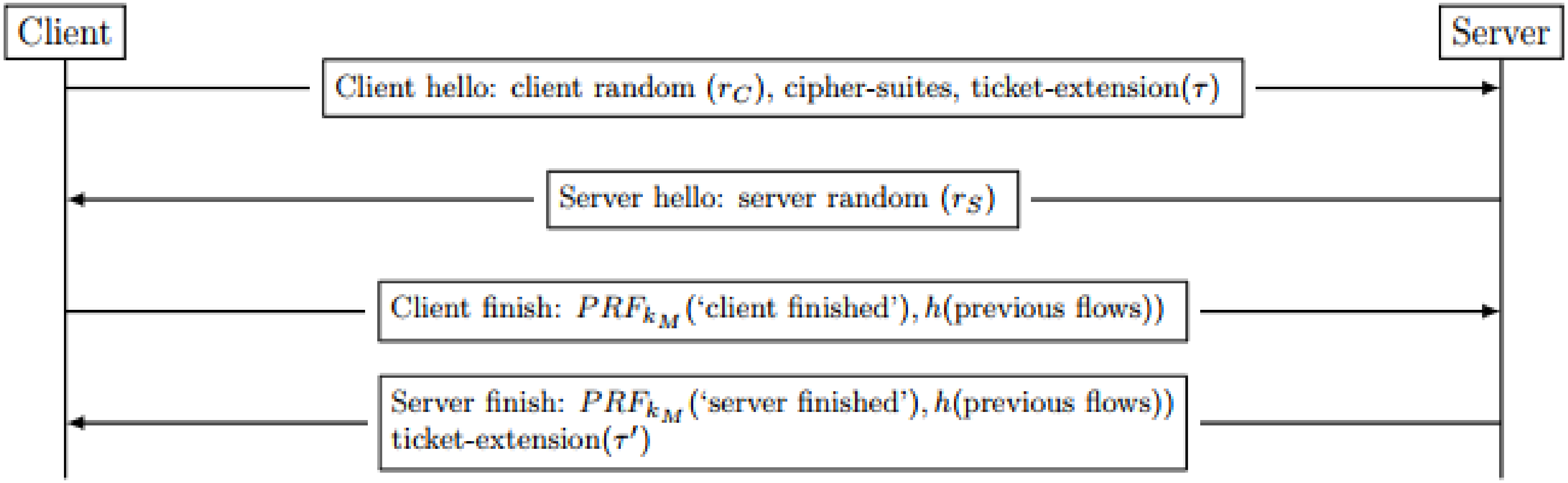
Finished (Confirmation -MAC of handshake messages) →

In first session of connection (not resumed), client does not send session_id, and only server sends it with ServerHello to allow resumption

Session Resumption Issues

- Need to keep state, lookup ID…
  - Overhead (➔small cache: less effective)
  - Need to share among (many!) replicates of server
  - For PFS: ensure keys disappear after ‘period’
- Solution: Client-side caching (Session-Ticket Hello Extension)
  - Ticket contains master key, encrypted by a secret session ticket key, known (only) to server
    - Share with other servers of this site
    - Change periodically to enforce PFS
  - Uses TLS extension (not in SSL)

Session-Ticket Resumption



- To preserve PFS:
  - Tickets 'expire' after 'time period' (e.g., 24 hours)
  - Ticket-key changed rapidly (e.g., every hour or few)
  - Ticket-key erased after `time period' ends (e.g., daily)
- Problem: many servers do not limit ticket-key lifetime

TLS 1.3 ‘Full handshake’: 1-RTT

- No RSA: only DH + signature by server
- 1-RTT: one round trip time

Client **Server**

ClientHello (cipher-suites, $\{g_1^{a1}, g_2^{a2} \ldots\}$, *Client_random*)

ServerHello: *Server_random*, $g_i^b, E(extensions), cert, Sign(Hello)$)

Finished (Confirmation -MAC of handshake messages)

Finished (Confirmation -MAC of handshake messages)

Application data (protected)