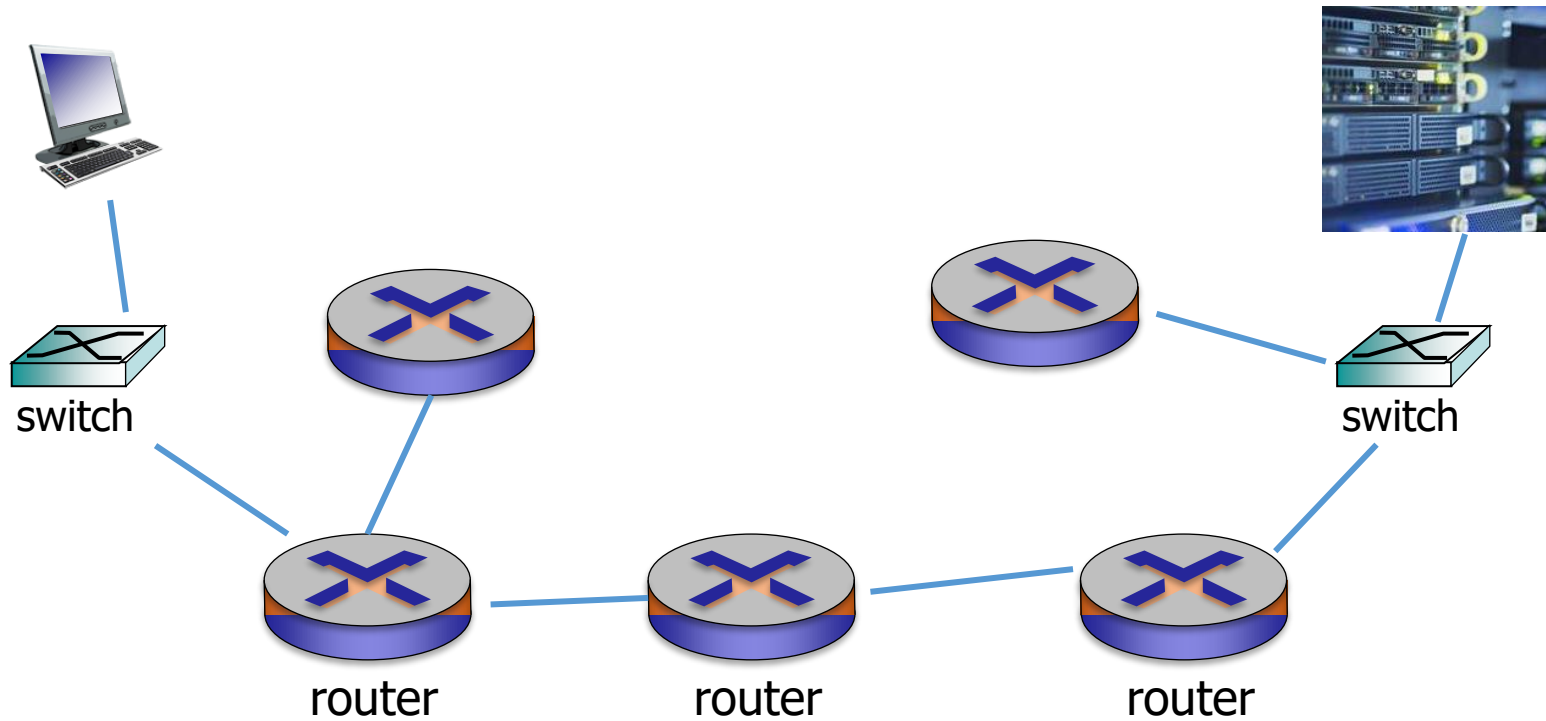

CSE 3400 - Introduction to Computer & Network Security
(aka: Introduction to Cybersecurity)

Transport Layer Security (TLS) and Secure Socket Layer (SSL)

Z. Jerry Shi

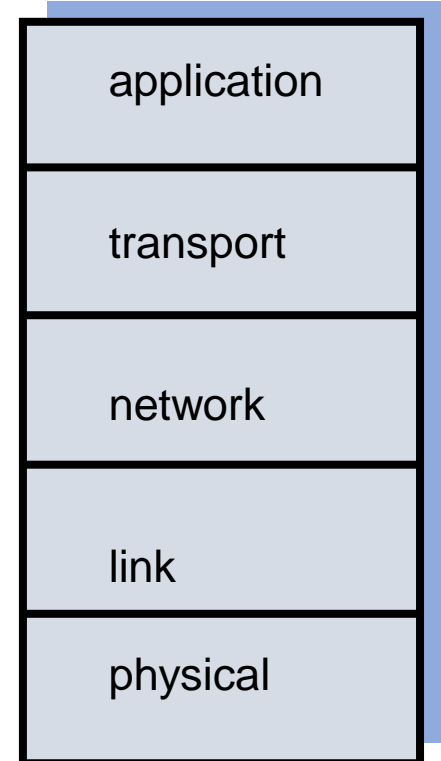
Based on Textbook Slides by Prof. Amir Herzberg

How the Internet works?



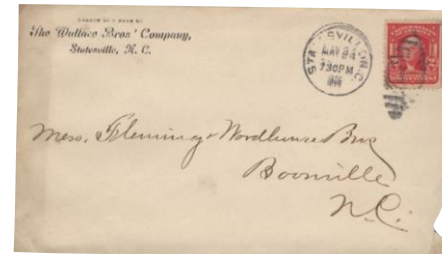
Internet protocol stack

- *application*: supporting network applications
 - FTP, SMTP, HTTP
- *transport*: process-process data transfer
 - TCP, UDP
- *network*: routing of datagrams from source to destination
 - IP, routing protocols
- *link*: data transfer between neighboring network elements
 - Ethernet, 802.111 (WiFi), PPP
- *physical*: bits “on the wire”



IP (Internet Protocol)

- A datagram service at the network layer
 - Data are divided into packets
 - Every packet has a destination “label” : **the IP address!**
 - Packets are handled independently
 - And delivered with best-effort
 - However, you can see loss, out of order, duplication,...



The Internet Transport Layer

Two protocols above IP

- TCP (Transmission Control Protocol) → TCP/IP
 - **Connection-oriented** like POTS (plain old telephone service)
 - Flow-control and bi-directional
 - Reliable
 - UDP (User Datagram Protocol) → UDP/IP
 - **Connection-less** like (unregistered) snail mail
 - No acknowledgments or retransmissions
 - Packets may be delivered out of order and have duplicates
 - Retransmission can be handled in applications
-

Domain name for dummies

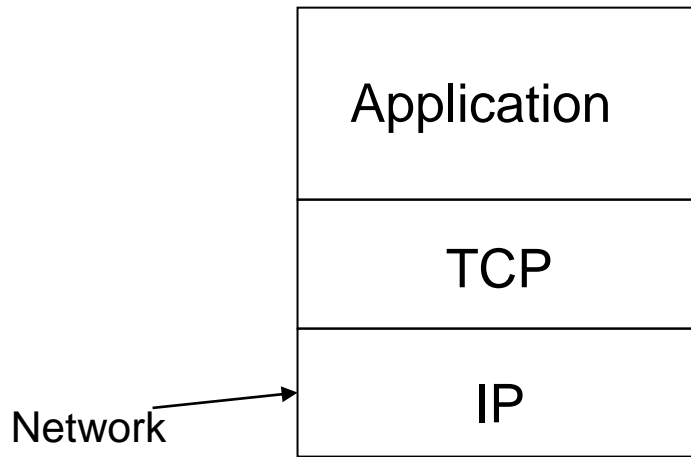
- A domain name is an identification string
 - Formed by rules and procedures in DNS (Domain name system)
 - Hierarchical
- Fully qualified domain names (FQDNs) is a complete domain name for a specific computer on the Internet
- Domain name servers (also a DNS) translate domain names to IP address
 - Phonebook for the Internet

Example:

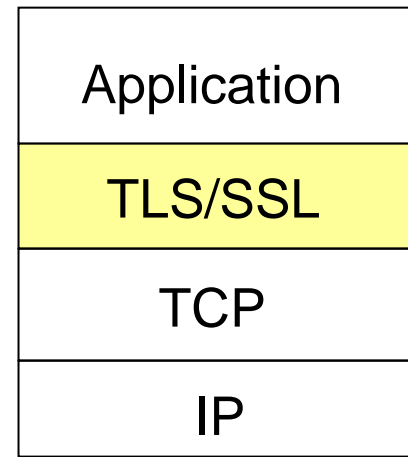
www.uconn.edu is a FQDN.

www.google.com is a FQDN. Its IP address is 172.217.7.228

SSL/TLS and TCP/IP



normal application



application with SSL

- SSL provides application programming interface (API) to applications
 - SSL libraries/classes readily available on many operating systems, for many programming languages

History of the protocol

- SSL([Secure Sockets Layers](#)) 1.0 (Netscape, 1994)
 - Mechanisms: [Woo 1994], implementation: Netscape (Internal?)
 - SSL 2.0 (Netscape, Feb1995)
 - Badly broken
 - SSL 3.0 (Netscape and Paul Kocher, 1996)
 - TLS ([Transport Layer Security](#)) 1.0 (Jan 1999)
 - RFC 2246, Standard based on SSL 3.0
 - Not interoperable with SSL 3.0
 - TLS 1.1 (April 2006)
 - More protection for CBC mode (IV and padding)
 - TLS 1.2. RFC 5246. (Aug 2008)
 - Replacing MD5 (with SHA-256 or SHA-1), Adding AES, etc.
 - TLS 1.3. RFC 8446. (Aug 2018)
 - In Use.
-

TLS/SSL: Security Goals

- Connection integrity and confidentiality
 - Key exchange: setup shared key
 - Server authentication
 - Client authentication (optional – and rarely used)
 - Cipher agility
 - Robust crypto
 - Perfect forward secrecy
 - MitM attacker model
-

TLS/SSL: Engineering Goals

- Efficiency
 - Session resumption
 - Minimizing round trips
 - Extensibility and versatility
 - Ease of deployment and use
-

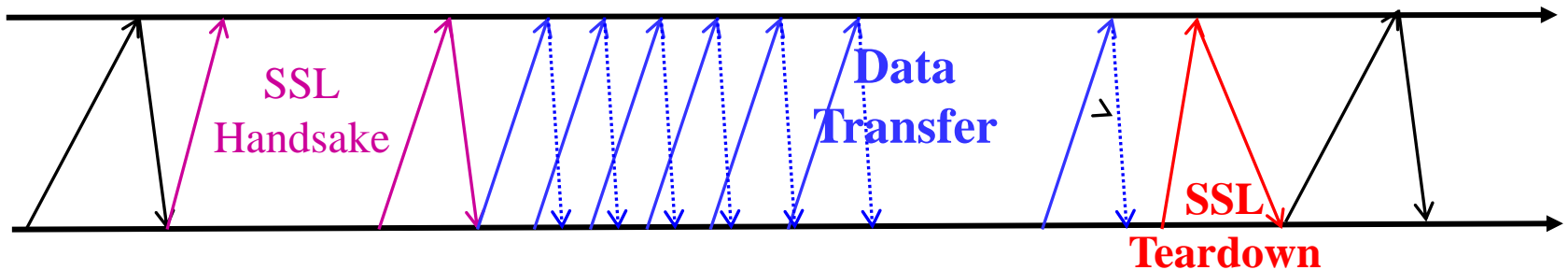
SSL/TLS Architecture

- SSL/TLS is built in two layers:
 - **Handshake Layer** – server[+client] auth, key exchange, cipher suite negotiation, extensions...
 - **Record Layer** – secure communication between client and server using exchanged session keys

TLS Handshake	HTTPS	...	HTTP	...
TLS record			HTTP	...
TCP sockets API				
TCP				
IP				

TLS/SSL 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 (why?)**
- TCP connection closure (Fin+Ack)

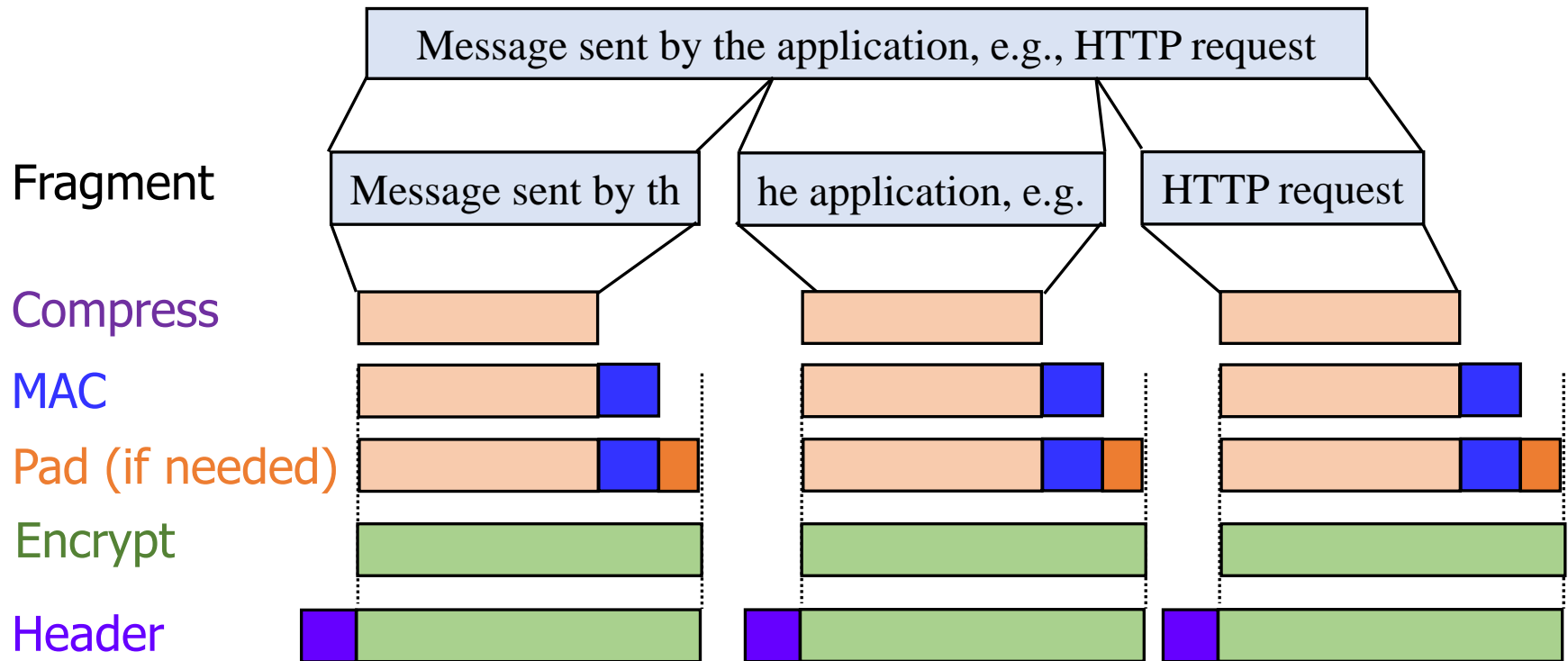


SSL Record Layer

- 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
-

TLS(till 1.3)/SSL Record Layer

- Assumes underlying reliable communication (TCP)
- Fragmentation, compression, authentication, encryption

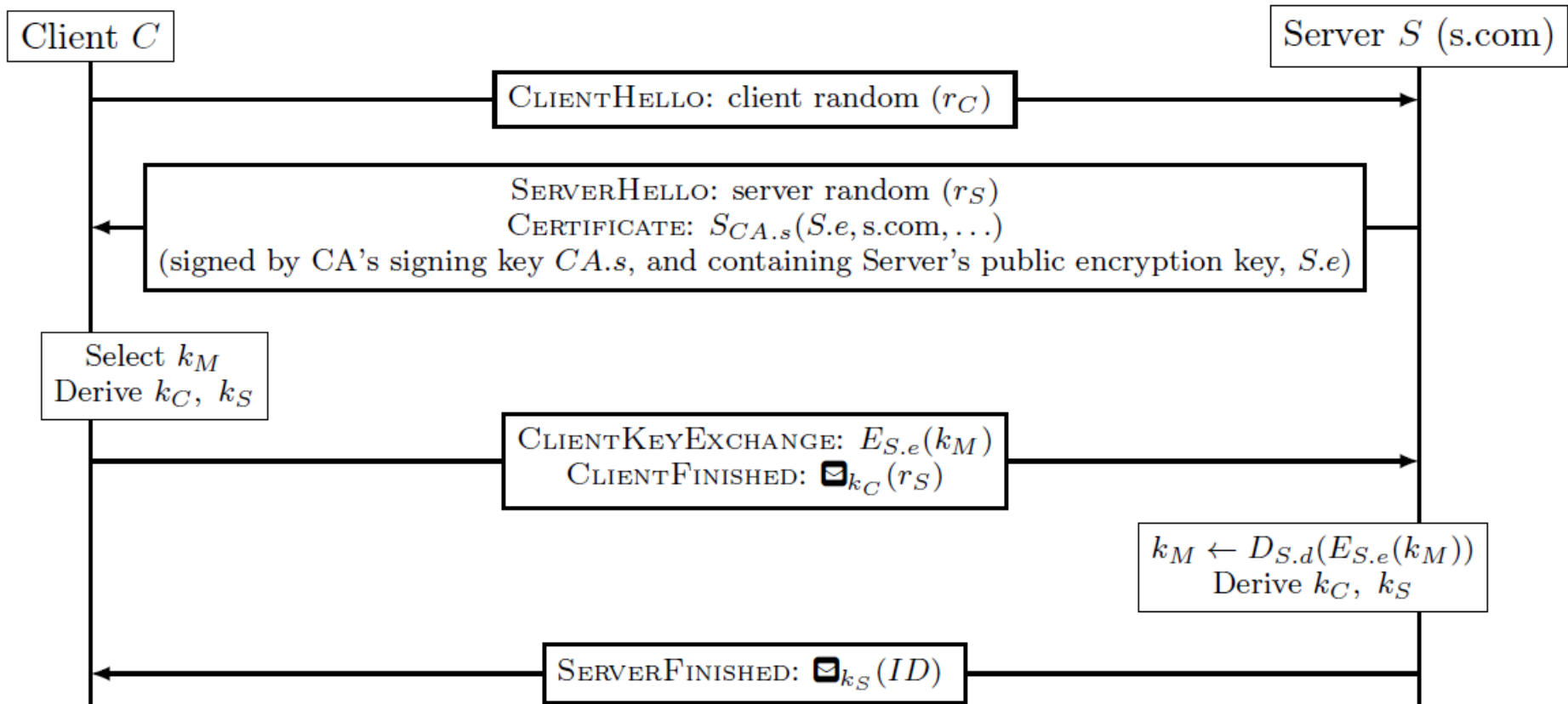


Finally, send each (processed) fragment via TCP

Record Layer Vulnerabilities

- Surprisingly many found, exploited!
 - → SSL, TLS1.0: vulnerable record protocol
 - Examples...
 - Attacks on RC4 → to be avoided
 - CBC IV reuse in session (BEAST)
 - `MAC-then-Encrypt': padding attacks [Lucky13,POODLE, ...]
 - Compress-then-encrypt: CRIME, TIME
 - Our focus is handshake
 - Includes: downgrading to use vulnerable version!
-


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 and k_S

SSLv2: important concepts

Randomly selected
by the client



- Derive two separate keys from master key k_M
 - 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

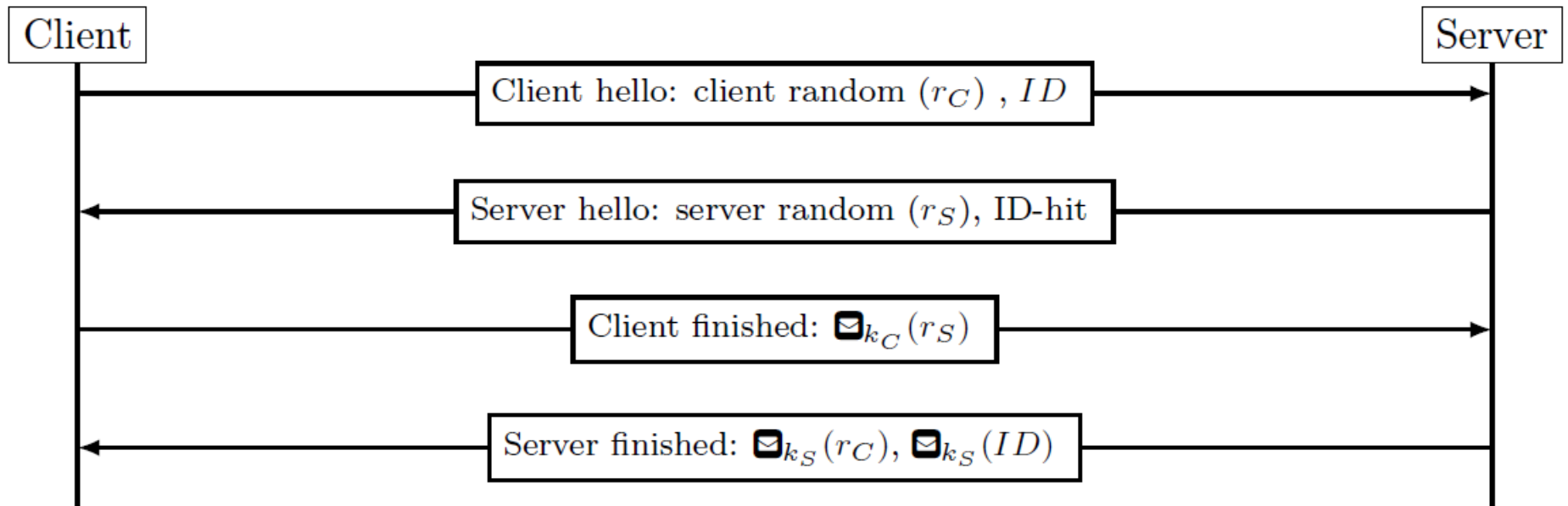
$$k_C = MD5(k_M \parallel "1" \parallel r_C \parallel r_S)$$

$$k_S = MD5(k_M \parallel "0" \parallel r_C \parallel r_S)$$

- Sessions: reusing public-key operations
- Cipher-agility
- Optional client authentication

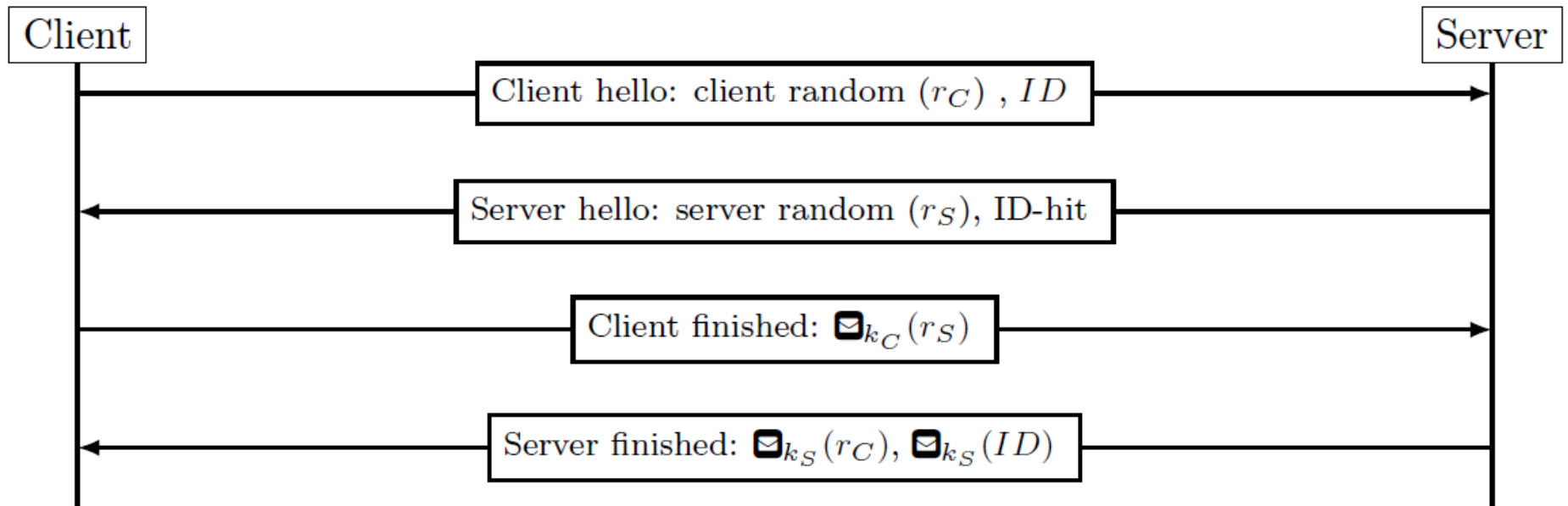
SSLv2 Session Resumption – ID based

- Goal: resume session faster,
 - Both caches (ID, k_M), per peer
 - Client includes ID in Client hello
 - If server knows ID, it sends only nonce (no cert req')
 - Server sends (new) identifier ID' at end of handshake



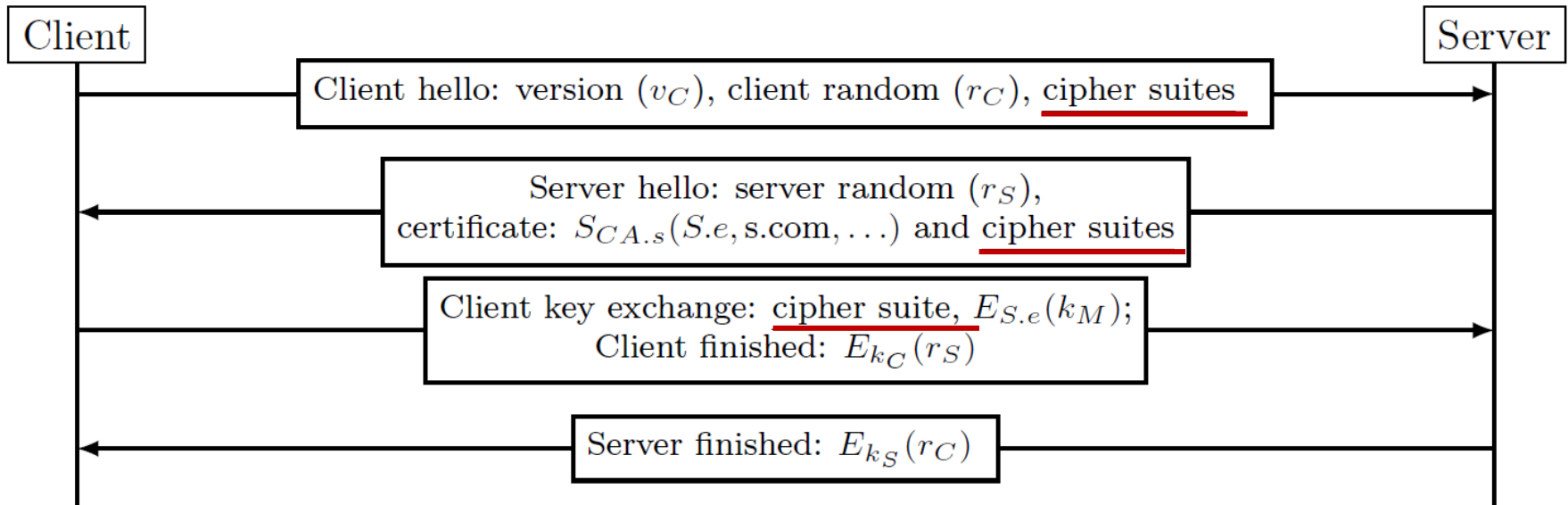
SSLv2 Session Resumption: Exercise

- Demonstrate (replay) MitM attack on SSLv2, if using a fixed value for:
 - (1) Client random,
 - (2) Server random



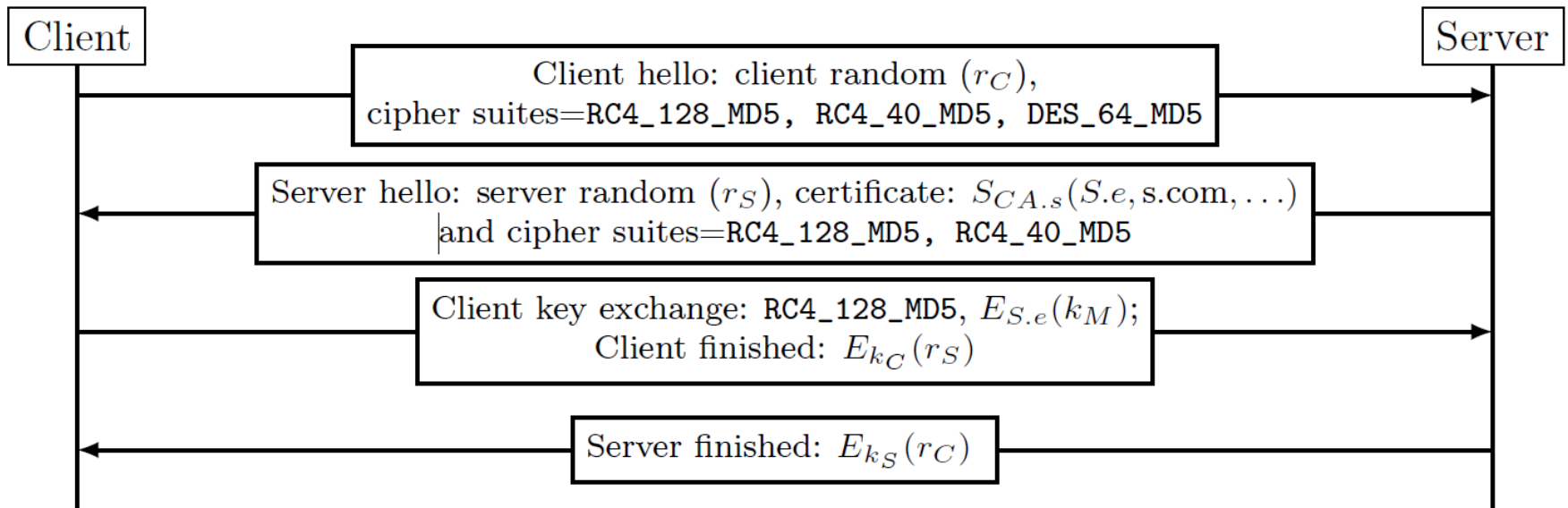
SSLv2 Ciphersuite Negotiation

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



SSLv2 Ciphersuite Negotiation Example

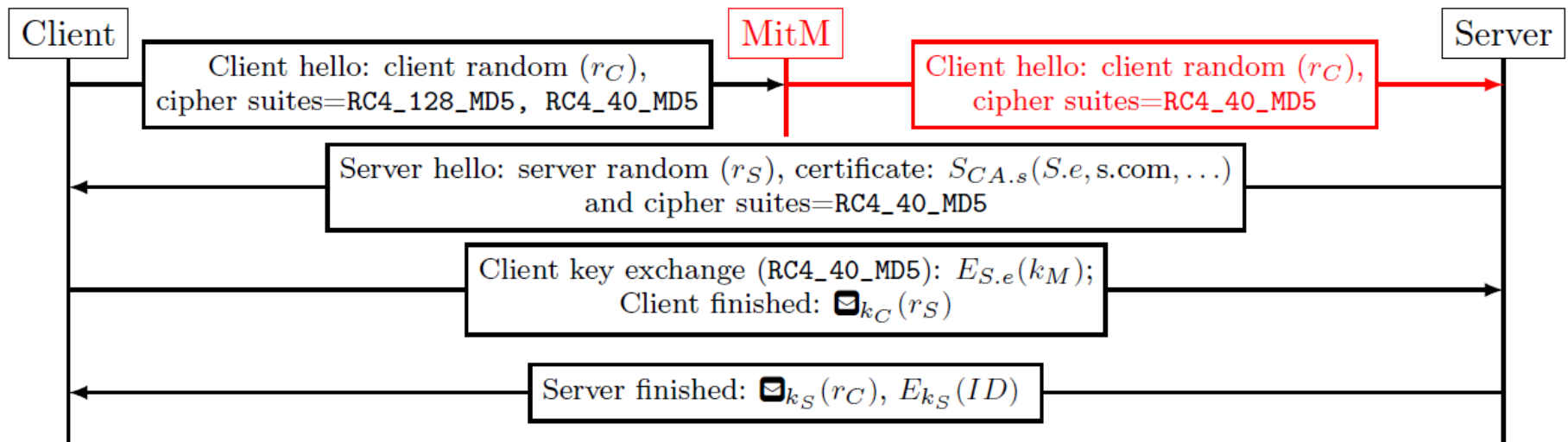
- Example: RC4_128_MD5 chosen



- 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
 - SSLv2 allowed only RSA; TLS 1.3: only PFS
 - Session-ticket resumption
-

SSLv3-TLS1.2: Key Derivation - 1

- Handshake exchanges premaster key k_{PM}
- Derive master key:

$$k_M = \text{PRF}_{k_{PM}}(\text{"master secret"} \parallel r_C \parallel r_S)$$

Why this extra step of premaster key?

Premaster key might not be (fully) random

- Weak randomness at a (weak) client
- Weak client reuses same PK-encrypted key
- DH-derived premaster key

SSLv3-TLS1.2: Key Derivation - 2

- Derive key block from master key:

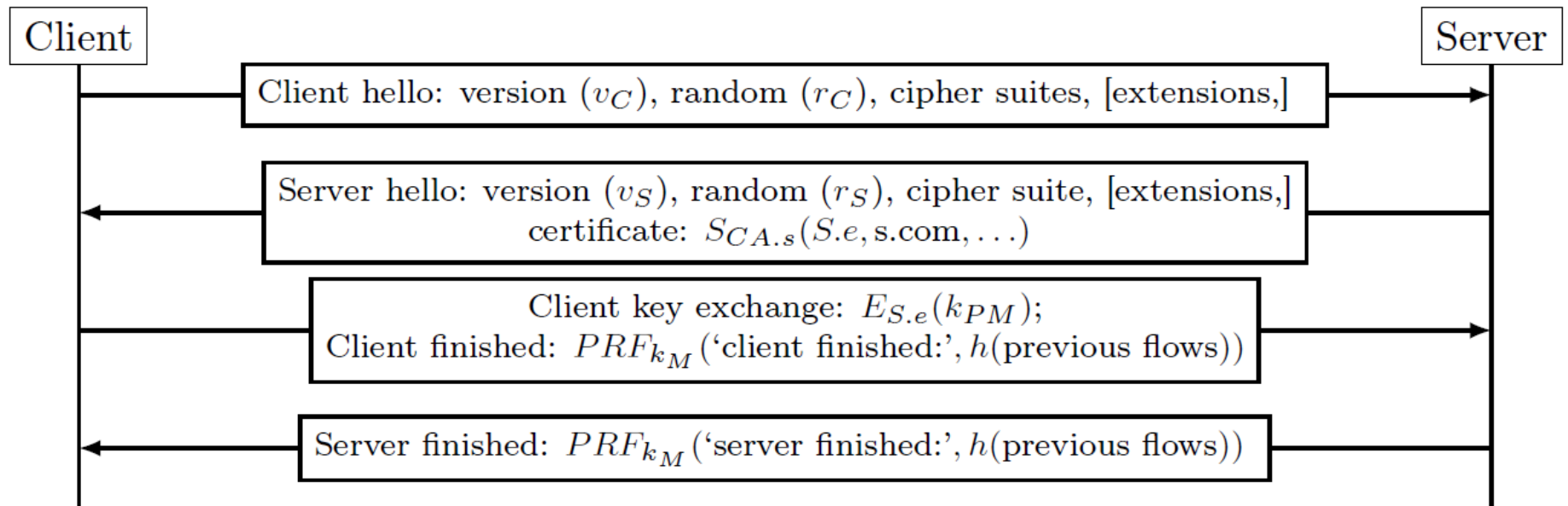
$$\text{key-block} = \text{PRF}_{k_M}(\text{"key expansion"} \parallel r_C \parallel r_S)$$

- Chop the key block into keys

$\text{key-block} = \text{PRF}_{k_M}(\text{"key expansion"} \parallel r_C \parallel r_S)$					
k_C^A	k_S^A	k_C^E	k_S^E	IV_C	IV_S

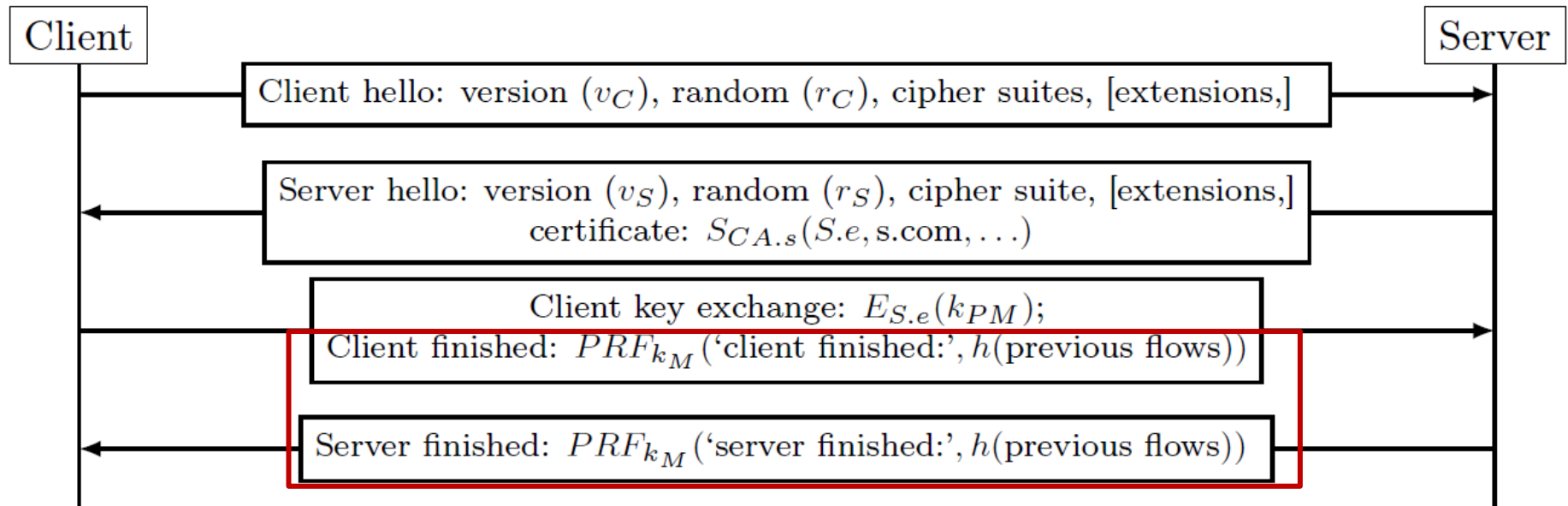
Basic RSA Handshake: SSLv3-TLS1.2

- Used in SSLv3 and TLSv1.0, 1.1 and 1.2
- The client generates k_{PM} and sends to the server



SSLv3-TLS1.2: Handshake integrity

- Extend the finish-message validation:
authenticate **entire** previous handshake flows
 - Foils the downgrade attack on SSLv2
 - Some differences between versions: simplified



SSLv3-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: Backwards compatibility

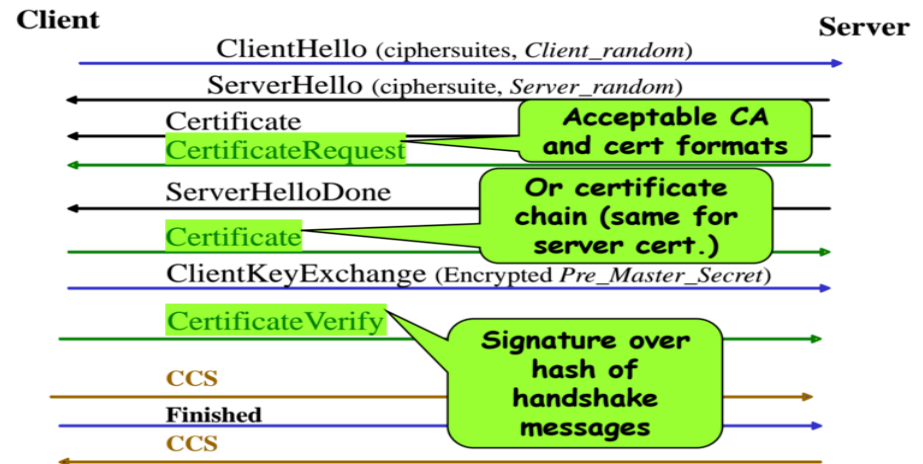
- Challenge: upgrading existing protocol
 - Unrealistic: all upgrade at same day
 - Backward compatibility: new (server, client) can still work with old (client, server)
 - Server selects version based on client's (in 'hello')
 - Prevent downgrade using 'finish' integrity check
 - Dilemmas for clients:
 - Some servers fail to respond to new handshake
 - 'Downgrade-dance' clients: try new versions, then older → vulnerable!
 - Downgrade to SSLv2 (no integrity!)
 - Disallowed – in SSL3, allow with 'trick' / vulnerable
 - Immediate discovery of key → forge MAC [LOGJAM]
-

Advanced Handshake Features

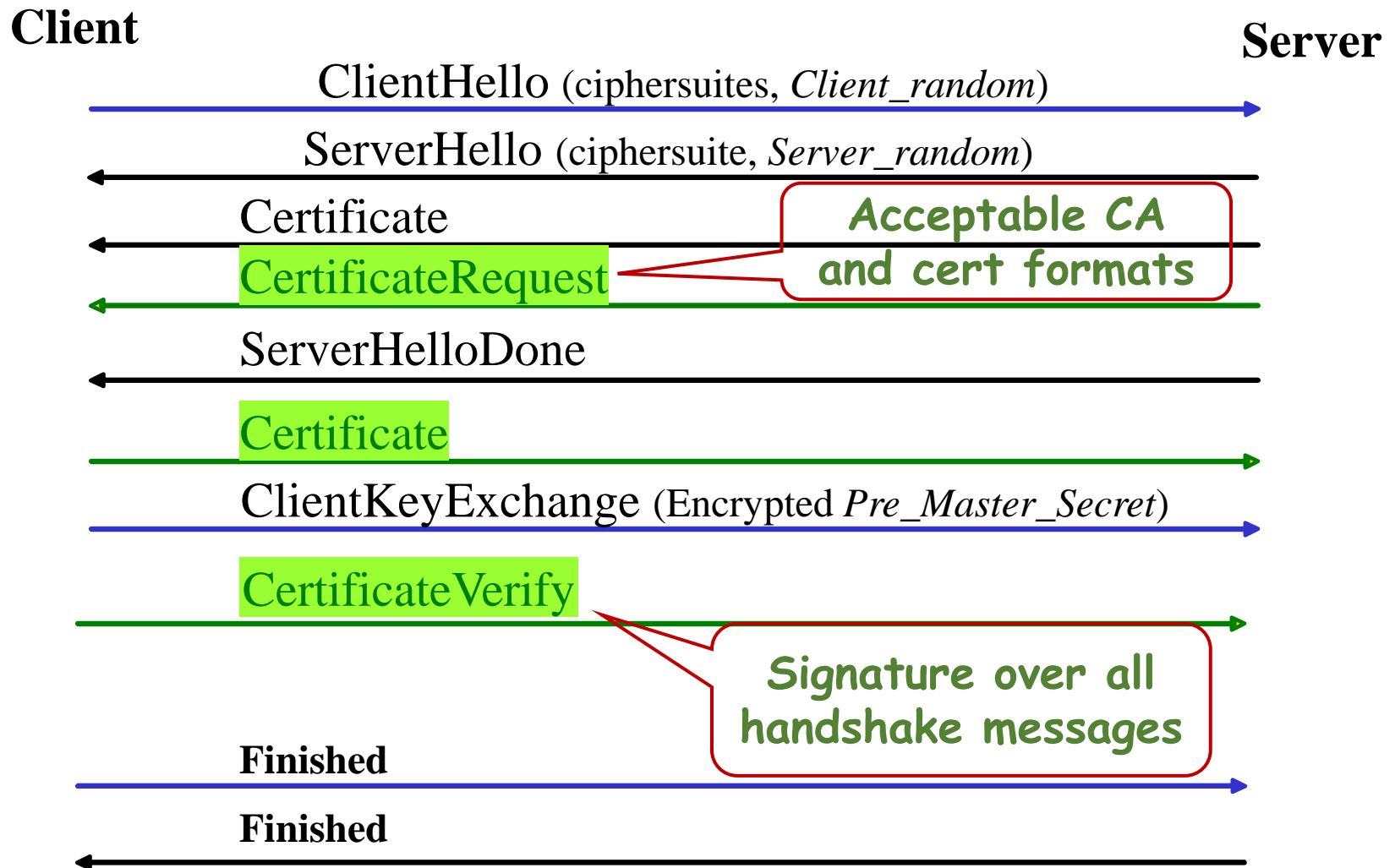
- Client authentication
 - Perfect Forward Secrecy (PFS) - ephemeral keys
 - Session resumption (ID-based, ticket)
 - TLS 1.3 handshakes
-

TLS/SSL Client Authentication

- Usually, TLS/SSL used only with server's public key
 - Only allows client to check server's keys
 - Client authentication: encrypt secret (pw, cookie)
- But TLS/SSL also allows client certificates
- How?
 - Client authenticates by signing with certified PK
- Easy – no PW!



TLS/SSL Client Authentication



SSL Client Authentication: Issues

- PKI challenges. Which identifier?
 - No global, unique namespace
 - Result: each server use its own client names, certificates
 - Device dependency. Support for mobility of cert and key...
 - Smartcard, USB `stick`?
- ➔ **Limited use**, mainly within organization/community
-

Ephemeral public keys

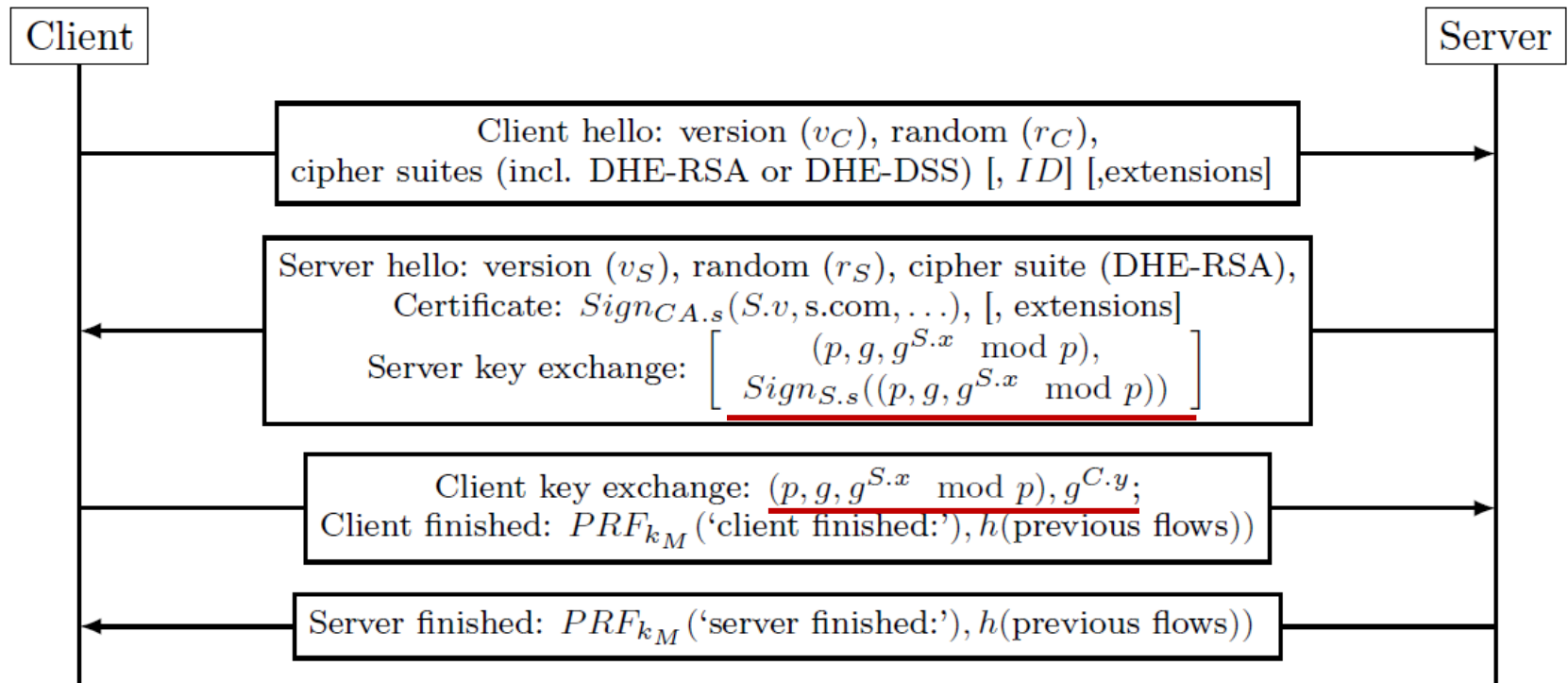
- Ephemeral keys: per-connection
 - Per-connection public keys ? Why?
- Motivations?
 - **Perfect forward security**: present traffic immune from future exposure – including of **past keys**
 - Historical: ‘export-grade’ (weak) keys [512b RSA]
- How?
 - Diffie-Hellman key exchange
 - Authenticated using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

The premaster key is

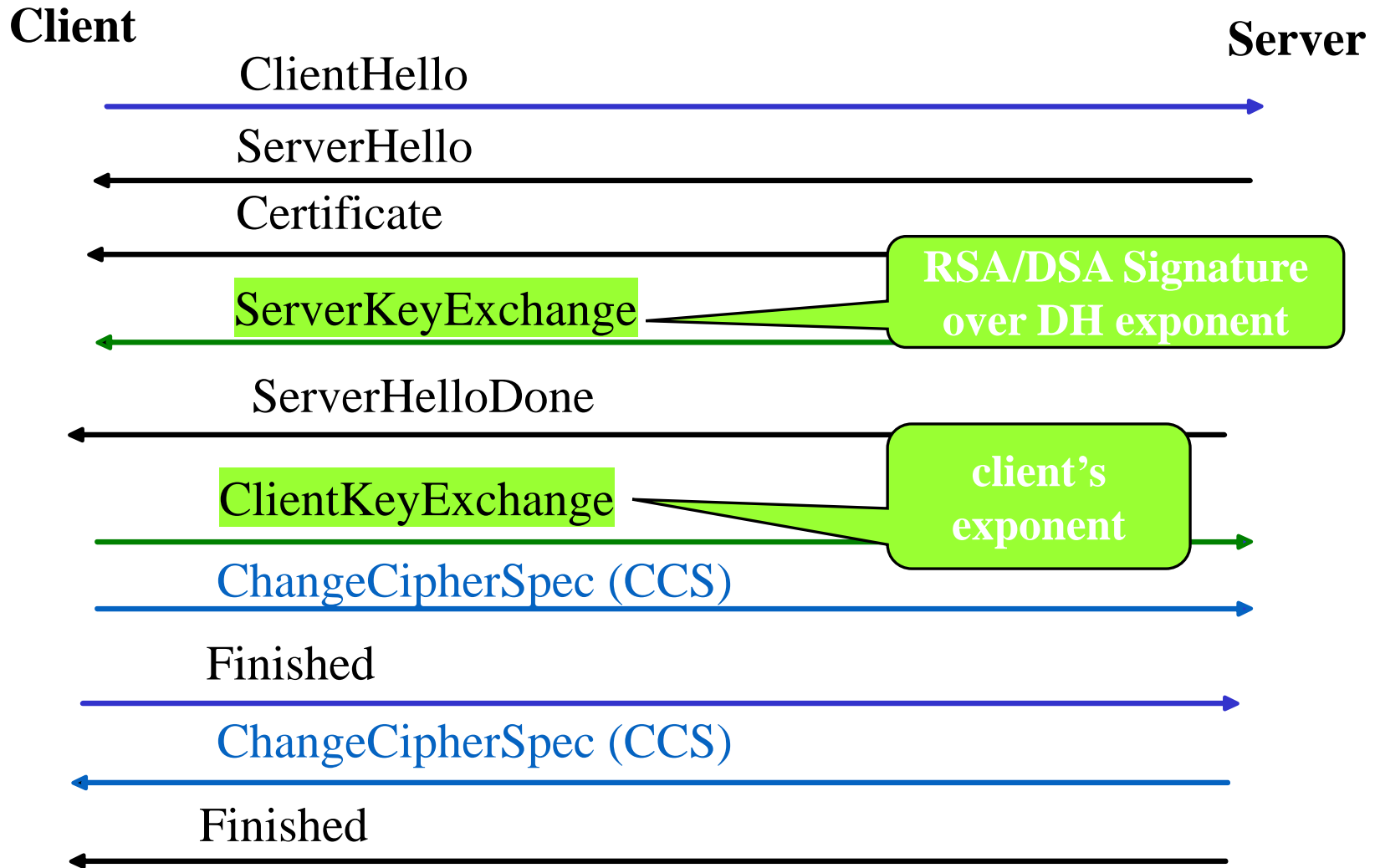
$$k_{PM} = g^{C.y.S.x} \bmod p$$

TLS/SSLv3 DH Ephemeral (DHE) Handshake:

- Server signs a DH exponent $g^{S.x}$
 - E.g., using RSA signatures
- Client just sends DH exponent $g^{C.y}$

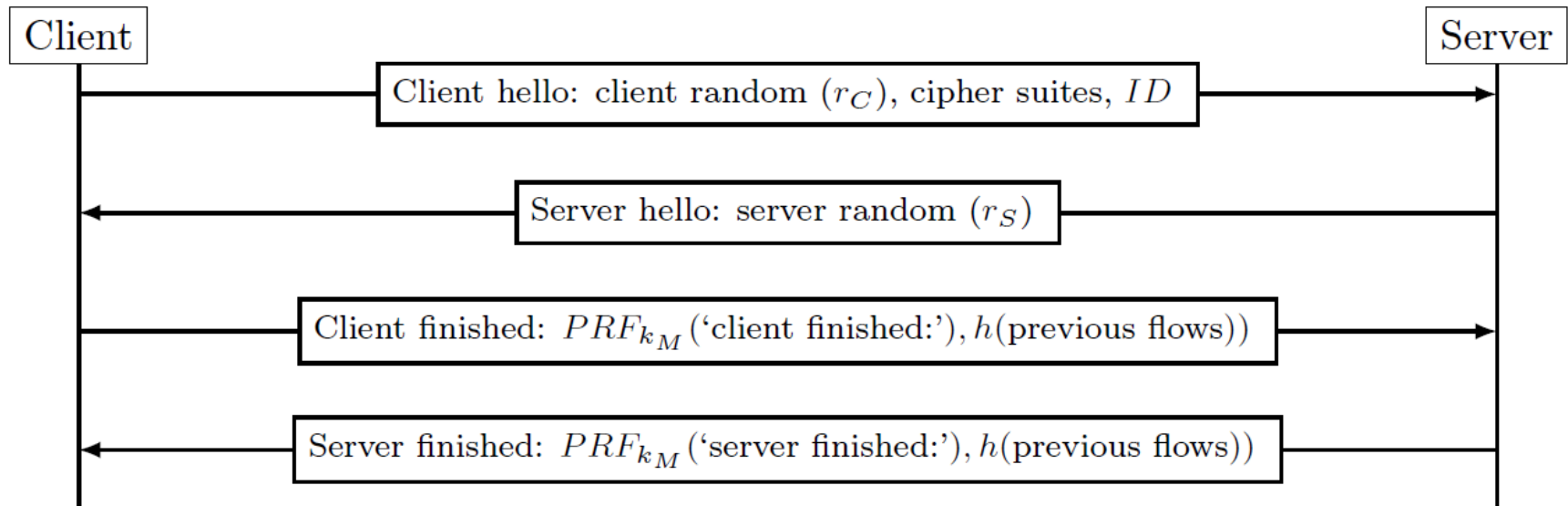


TLS/SSL Ephemeral PK Handshake



ID-based Session Resumption

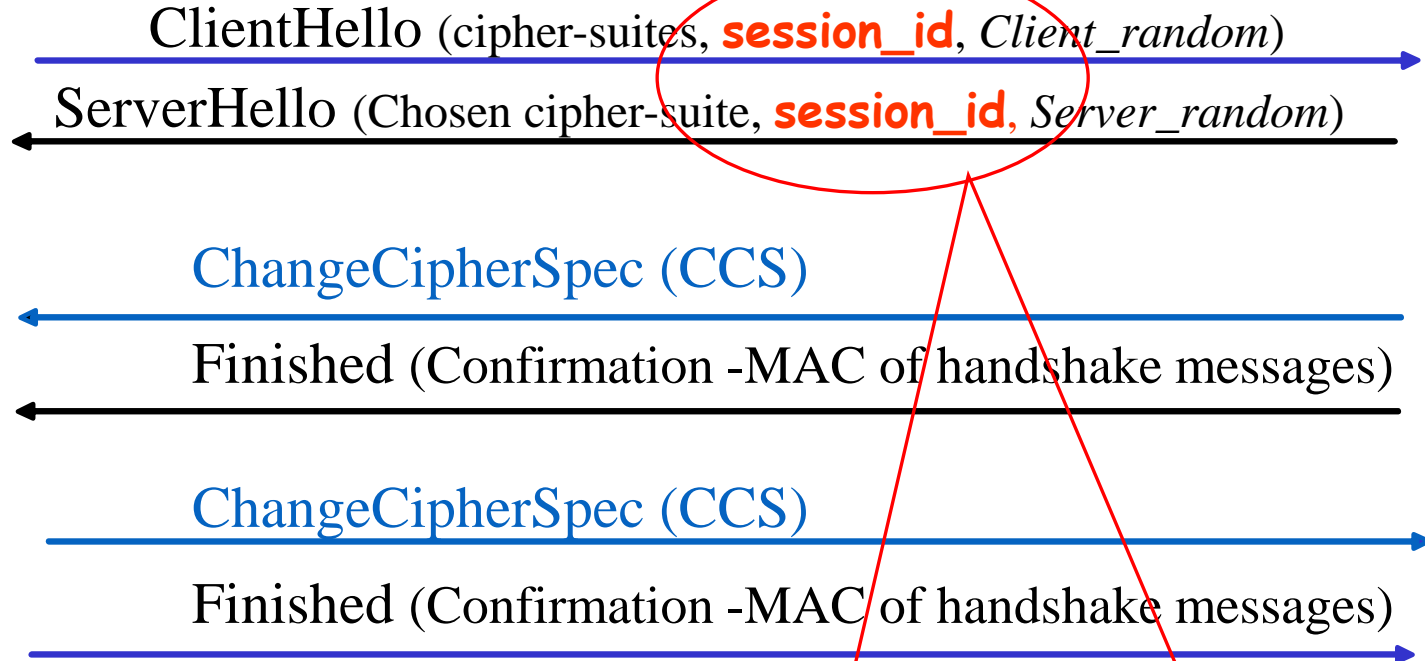
- Idea: server and client store (ID, key) per peer
- Reuse in new connections between the same pair
- Save public key operations (CPU and bandwidth)



Session-ID Resumption Handshake

Client

Server



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'

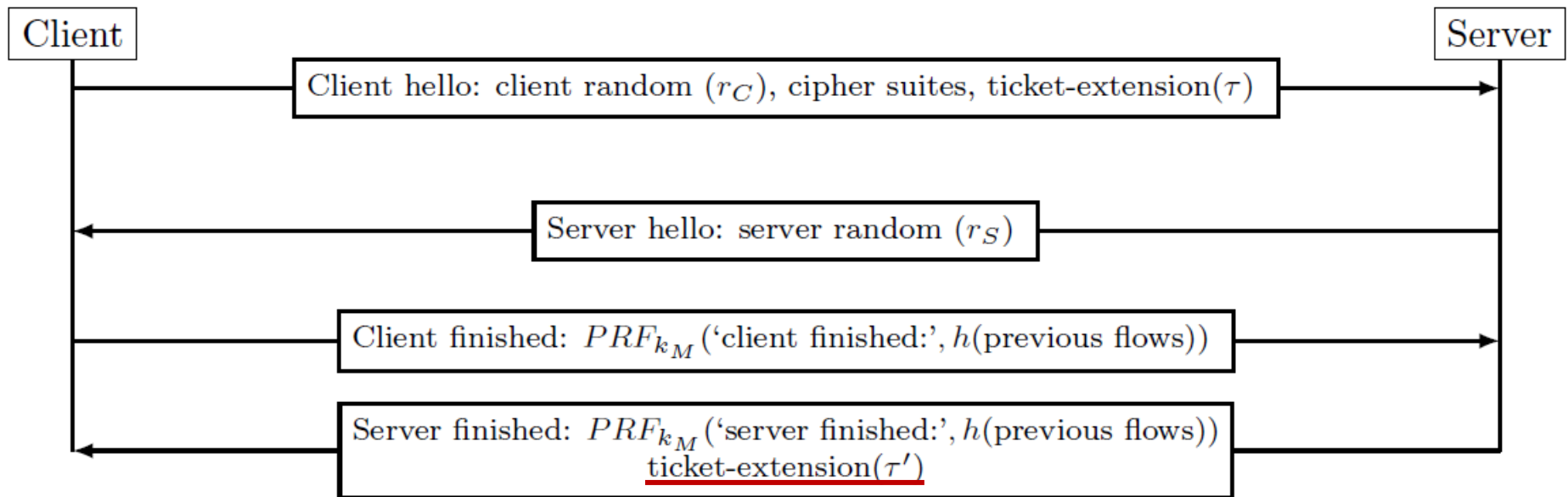
Solution: Client-side caching

(Session-Ticket Hello Extension, as TLS extension)

- **Ticket** contains master key (and other information), encrypted by a secret **session ticket encryption key**
 - The ticket keys are only known to servers, and can be shared with other servers of this site
 - Change the ticket keys periodically to enforce PFS
-

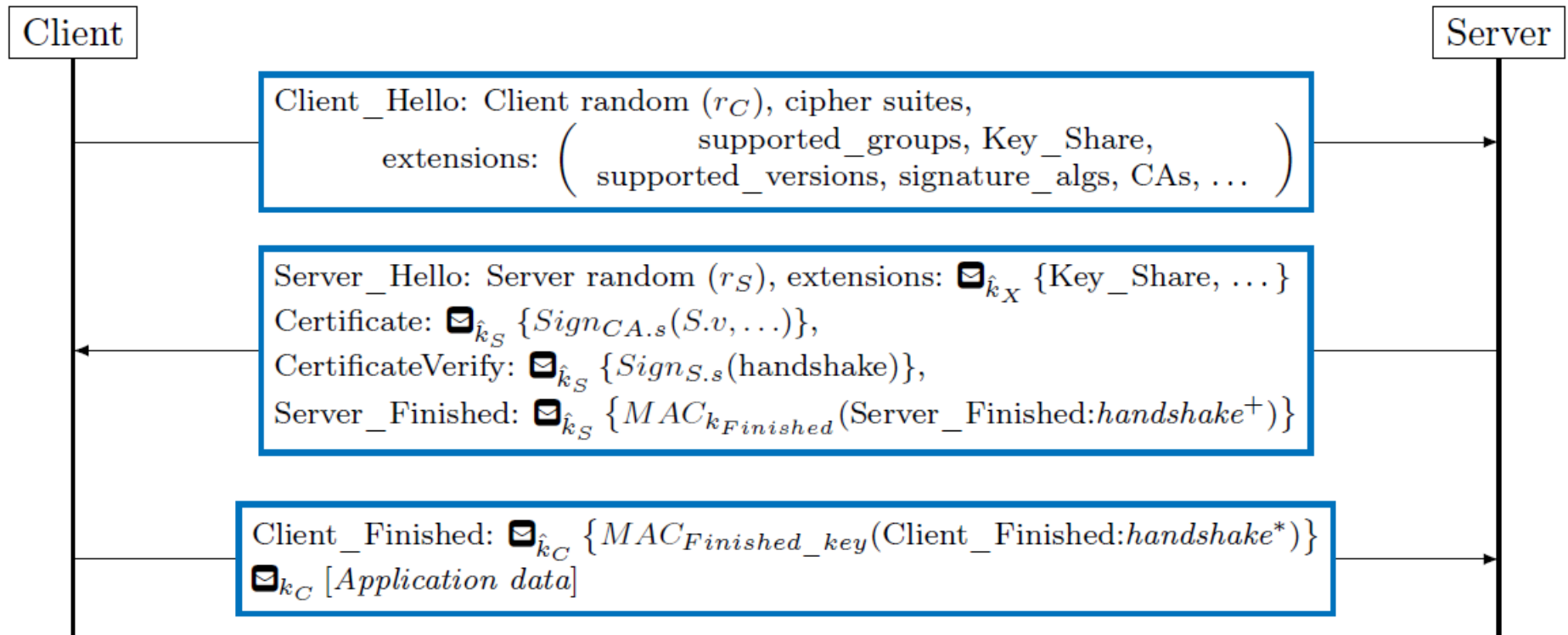
Ticket-based Session Resumption

- 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

- TLS 1.3 Full handshake always uses DH protocol
 - 1-RTT
 - Client sends key-share, one per cipher option, in Hello !



TLS 1.3 Session Resumption: PSK

- Resume only using Pre-Shared Key (PSK)
 - Use PSK to authenticate key-shares and derive key
 - Optional use of DHE for PFS (ephemeral key)
 - Essentially, build-in ticket mechanism

Client

Server

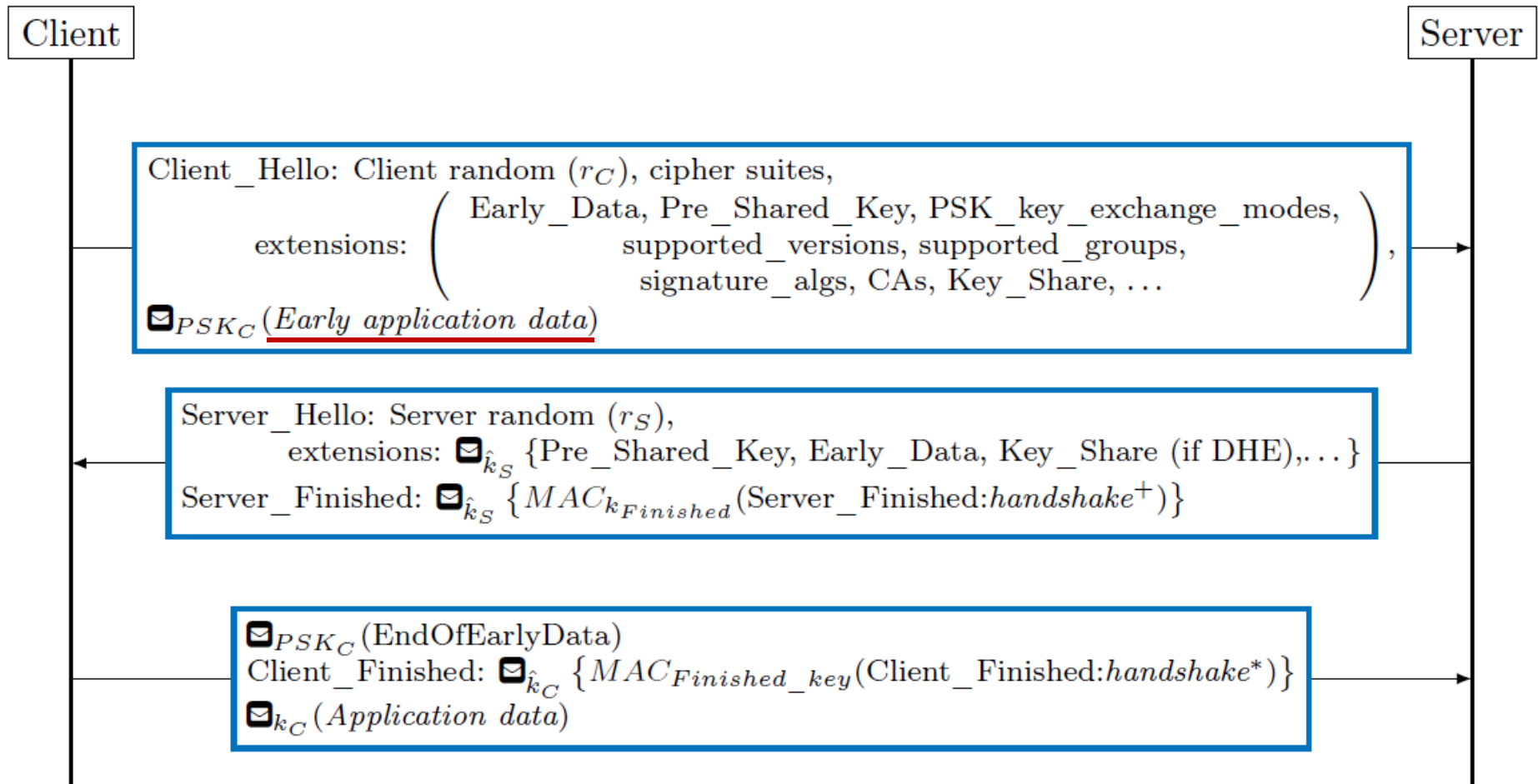
Client_Hello: Client random (r_C), cipher suites,
extensions: $\left(\begin{array}{l} \text{Pre_Shared_Key, PSK_key_exchange_modes,} \\ \text{supported_versions, supported_groups,} \\ \text{signature_algs, CAs, Key_Share, ...} \end{array} \right)$

Server_Hello: Server random (r_S),
extensions: $\boxtimes_{\hat{k}_S} \{ \text{Pre_Shared_Key, Key_Share (if DHE), ...} \}$
Server_Finished: $\boxtimes_{\hat{k}_S} \{ MAC_{k_{Finished}}(\text{Server_Finished:handshake}^+) \}$

Client_Finished: $\boxtimes_{\hat{k}_C} \{ MAC_{Finished_key}(\text{Client_Finished:handshake}^*) \}$
 $\boxtimes_{k_C}(\text{Application data})$

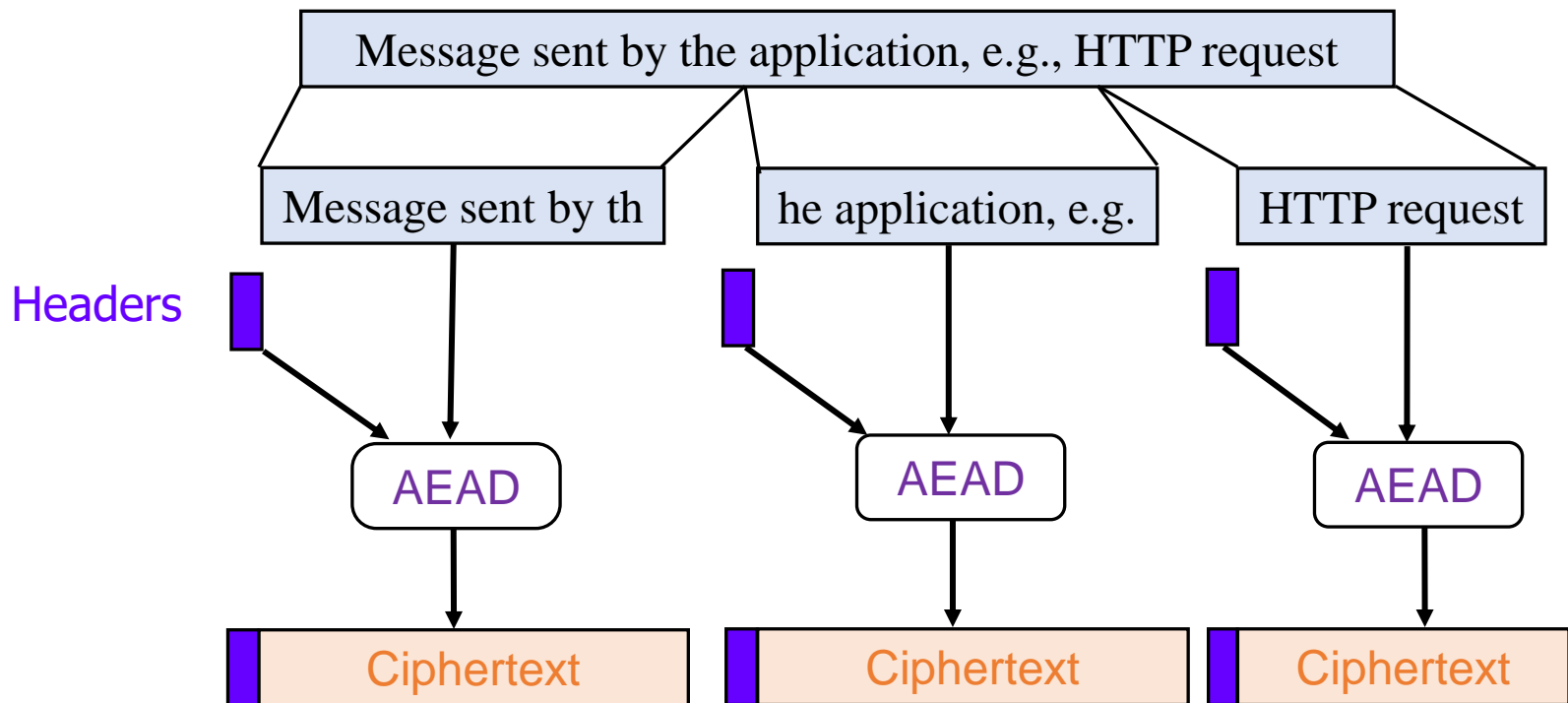
TLS 1.3 Session Resumption: 0-RTT

- TLS 1.3 even supports 0-RTT!
 - Include “Early application data” in the first message



TLS 1.3 Record Layer

- AEAD: Authenticated Encryption with Additional Data
 - After fragmentation – but no compression



TLS/SSL: Conclusion

- TLS/SSL: a mature, widely used crypto protocol
 - Many features, vulnerabilities, fixes, versions
 - Many downgrade attacks
 - More foresight, scrutiny would have saved a lot!
 - Extensibility by design principle: build into design mechanisms for secure extensions, downward-compatible versions, and negotiation
 - Improved key-separation: use independent keys for each different crypto scheme or version, and different types/sources of plaintext.
-