Lecture 5:
Attacks against TLS

CPSC 444/544, 2026 Spring

Instructor: Fan Zhang

Yale

Computer Science

TLS Handshake v.s. AKE

- The core cryptography is the same as AKE
  - While one-sided is common, TLS supports mutual authentication.
- TLS handshake has other features
  - Resumption, renegotiation, etc.

Handshake in action

Handshake in TLS 1.2 (Full HS, one-sided version)

cial specification: RFC RFC 5246 (Published in 2008)

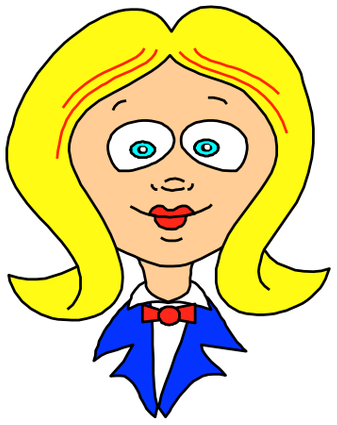
History leading to TLS 1.2

SSL 1.0	Terribly insecure; never released.
SSL 2.0	Released 1995; terribly insecure, deprecated in 2011
SSL 3.0	Released 1996; insecure, deprecated in 2015.
TLS 1.0	Released 1999; deprecated in 2020.
TLS 1.1	Released 2006; deprecated in 2020.
TLS 1.2	Released 2008. Ok.
TLS 1.3	Standardized in August 2018 and is being rolled out now; major change from TLS 1.2.

- Fair to say that 1.2 the **first proper TLS**
- Flaws have been found, and fixed.
- 99.99% of websites still support TLS 1.2 today

TLS 1.2 handshake with ECDHE (akin to AKE3)

ClientHello: r_c, list of cipher suites



Client

Server

“cipher suit”

Cipher suites: list of options like:
`TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256`

This says to use

- elliptic curve Diffie-Hellman for key exchange
- RSA digital signatures
- 128-bit AES for symmetric encryption
- GCM (Galois Counter Mode) AES mode of operation
- SHA-256 for hash function

Why supporting many ciphersuits?

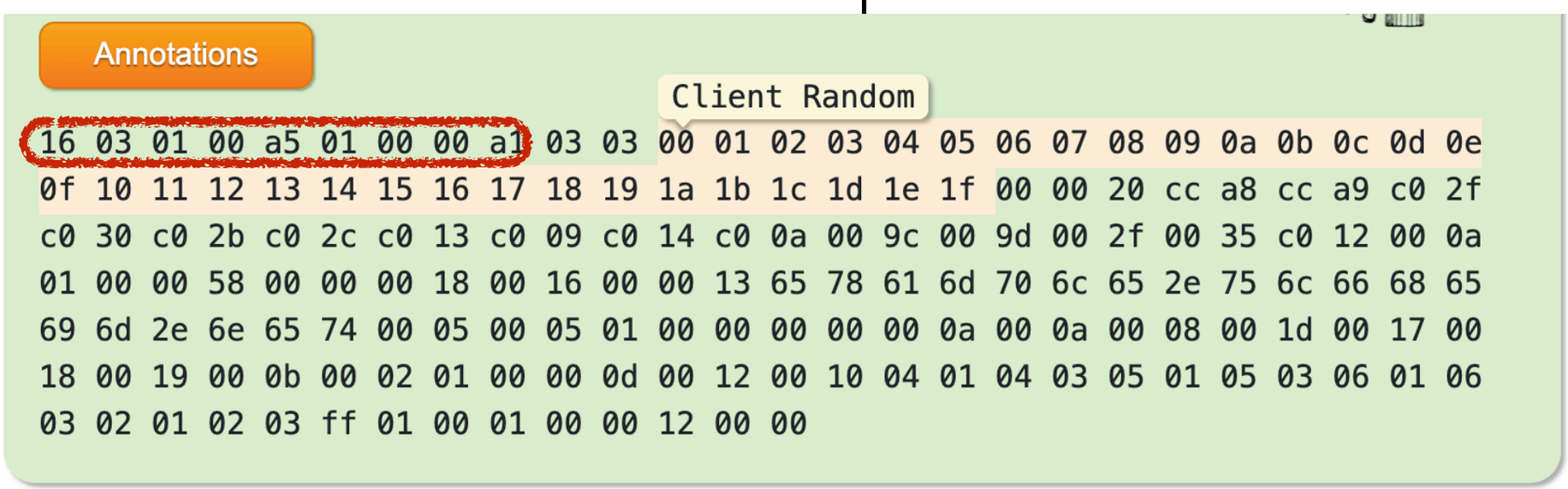
- Backward compatibility
- Performance v.s. security
  - Low-power devices (e.g., IoT devices) may only afford less secure schemes
- Regulatory requirements

What are some cons?

“Client Hello”

```
struct {
    ProtocolVersion client_version;
    Random random;
    SessionID session_id;
    CipherSuite cipher_suites<2..2^16-2>;
    CompressionMethod compression_methods<1..2^8-1>;
    select (extensions_present) {
        case false:
            struct {};
        case true:
            Extension extensions<0..2^16-1>;
    };
} ClientHello;
```

- Taken from RFC5246 (https://datatracker.ietf.org/doc/html/rfc5246#appendix-A.4.1).
- You will parse these bytes for the next lab :)
- Note: no integrity protection in this message.



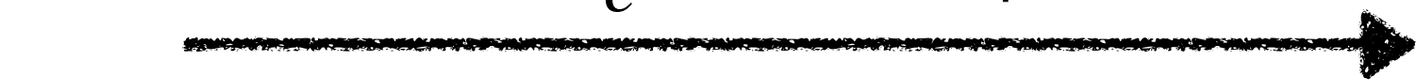
TLS Header

Payload (e.g., HS msg, app data)

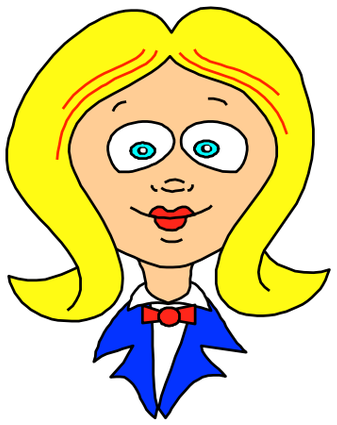
https://tls12.xargs.org/ is your friend too.

TLS 1.2 handshake with ECDHE

ClientHello: r_c, list of cipher suites



ServerHello: r_s, chosen cipher suite



Client

Server

Server Hello

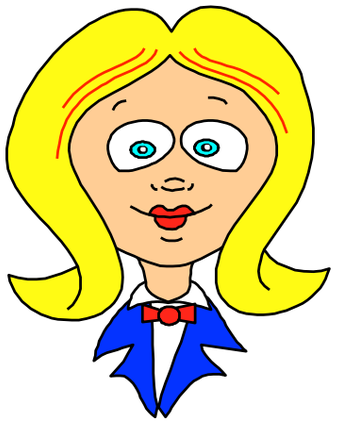
```
struct {
    ProtocolVersion server_version;
    Random random;
    SessionID session_id;
    CipherSuite cipher_suite;
    CompressionMethod compression_method;
    select (extensions_present) {
        case false:
            struct {};
        case true:
            Extension extensions<0..2^16-1>;
    };
} ServerHello;
```

TLS 1.2 handshake with ECDHE

ClientHello: r_c, list of cipher suites

ServerHello: r_s, chosen cipher suite

Certificate $\mathbf{Cert}_S$



Client

Server

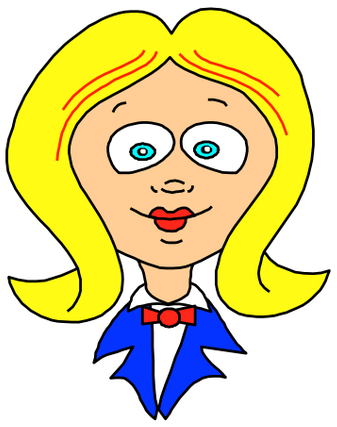
TLS 1.2 handshake with ECDHE

ClientHello: r_c, list of cipher suites

ServerHello: r_s, chosen cipher suite

Certificate $\mathbf{Cert}_S$

ServerKeyEx: $g^a, \mathbf{Sig}_S(g^a, r_c, r_s)$



Client

Server

- Client verifies the signature against the cert
- Like in AKE2, a ephemeral key g^a is used (**FS**)
- Like in AKE3, server signs random challenges each time (**HSM security**)

TLS 1.2 handshake with ECDHE

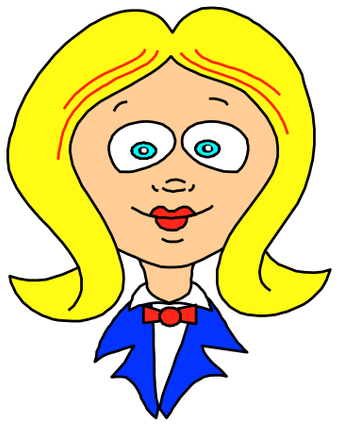
ClientHello: r_c, list of cipher suites

ServerHello: r_s, chosen cipher suite

Certificate $\mathbf{Cert}_S$

ServerKeyEx: $g^a, \mathbf{Sig}_S(g^a, r_c, r_s)$

ClientKeyEx: g^b



Client

Server

Derive sessions keys from g^{ab}

Derive sessions keys from g^{ab}

Session Keys

HMAC based

MS = g^{ab}

To generate the key material, compute

```
   key_block = PRF(SecurityParameters.master_secret,
                   "key expansion",
                   SecurityParameters.server_random +
                   SecurityParameters.client_random);
```

(r_s + r_c)

until enough output has been generated. Then, the key_block is partitioned as follows:

	GCM	CBC
client_write_MAC_key[SecurityParameters.mac_key_length]	0	32
server_write_MAC_key[SecurityParameters.mac_key_length]	0	32
client_write_key[SecurityParameters.enc_key_length]	16	16
server_write_key[SecurityParameters.enc_key_length]	16	16
client_write_IV[SecurityParameters.fixed_iv_length]	4	16
server_write_IV[SecurityParameters.fixed_iv_length]	4	16

Currently, the client_write_IV and server_write_IV are only generated for implicit nonce techniques as described in Section 3.2.1 of [AEAD].

https://datatracker.ietf.org/doc/html/rfc5246#section-6.3

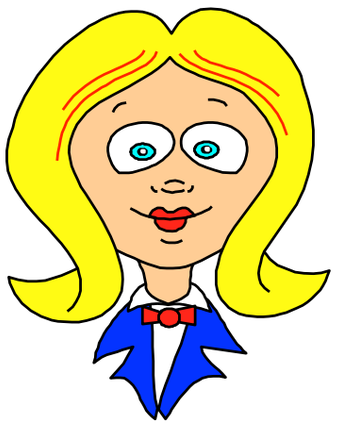
TLS 1.2 handshake with ECDHE

ClientHello: r_c, list of cipher suites

ServerHello: r_s, chosen cipher suite

Certificate $\mathbf{Cert}_S$

ServerKeyEx: $g^a, \mathbf{Sig}_S(g^a, r_c, r_s)$



Client

ClientKeyEx: g^b

ClientFinished

ServerFinished

Server

Essentially a MAC on all messages so far.

“Finished”

- Essentially, MAC(MS, “handshake msgs”)
- “I agree on the whole story so far”
- Prevents, e.g., an attack dropping some cipher suits.

```
Structure of this message:

   struct {
       opaque verify_data[verify_data_length];
   } Finished;

   verify_data
      PRF(master_secret, finished_label, Hash(handshake_messages))
         [0..verify_data_length-1];

   finished_label
      For Finished messages sent by the client, the string
      "client finished".  For Finished messages sent by the server,
      the string "server finished".
```

https://www.rfc-editor.org/rfc/rfc5246#section-7.4.9

“Finished”

```
Meaning of this message:

   The Finished message is the first one protected with the just
   negotiated algorithms, keys, and secrets.  Recipients of Finished
   messages MUST verify that the contents are correct.  Once a side
   has sent its Finished message and received and validated the
   Finished message from its peer, it may begin to send and receive
   application data over the connection.

Structure of this message:

   struct {
       opaque verify_data[verify_data_length];
   } Finished;

   verify_data
      PRF(master_secret, finished_label, Hash(handshake_messages))
         [0..verify_data_length-1];

   finished_label
      For Finished messages sent by the client, the string
      "client finished".  For Finished messages sent by the server,
      the string "server finished".
```

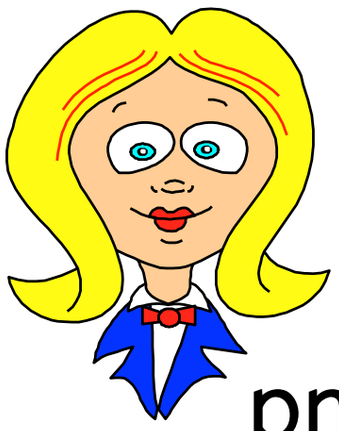
https://www.rfc-editor.org/rfc/rfc5246#section-7.4.9

Another mode: AKE1

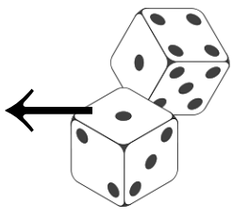
ClientHello: r_c, list of cipher suites

ServerHello: r_s, chosen cipher suite

Certificate $\mathbf{Cert}_S$



pms



Client

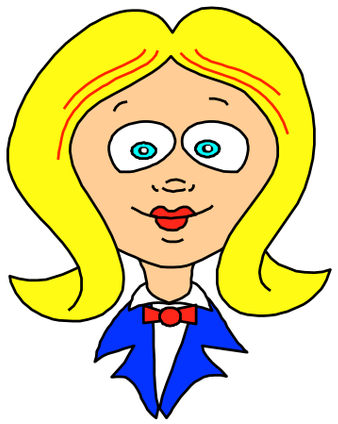
Server

ClientKeyEx: $\mathbf{Enc}_S(\mathbf{pms})$

Pms = pre-master-secret.
The rest is the similar as before

Downside?

Recap (w/ ECDHE)



Client

r_c

r_s, g^a, signature, $\mathbf{Cert}_S$

g^b, Finished

Finished

Application Data

Server

Encrypted

- **Two round trips** before app data can be sent!

It’d be nice to reduce rounds

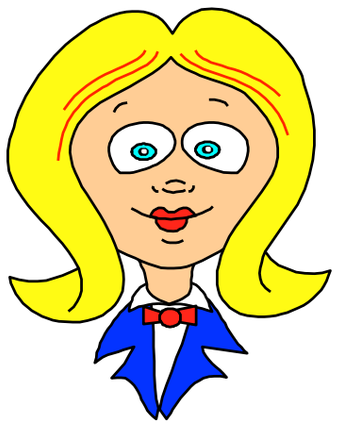
- “Every **100ms** delay cost Amazon **1% of sales**.” — Amazon 2006
- “With a **0.1s** improvement in site speed, we observed that retail consumers **spent almost 10% more**” — Deloitte 2020

High-level description of **TLS 1.3 handshake**

g^a, r_c, list of cipher suites

g^b, r_s, chosen cipher suite

Encrypted



Client

$\mathbf{Cert}_S$, $\mathbf{Sig}_S$(all msgs so far), Finished

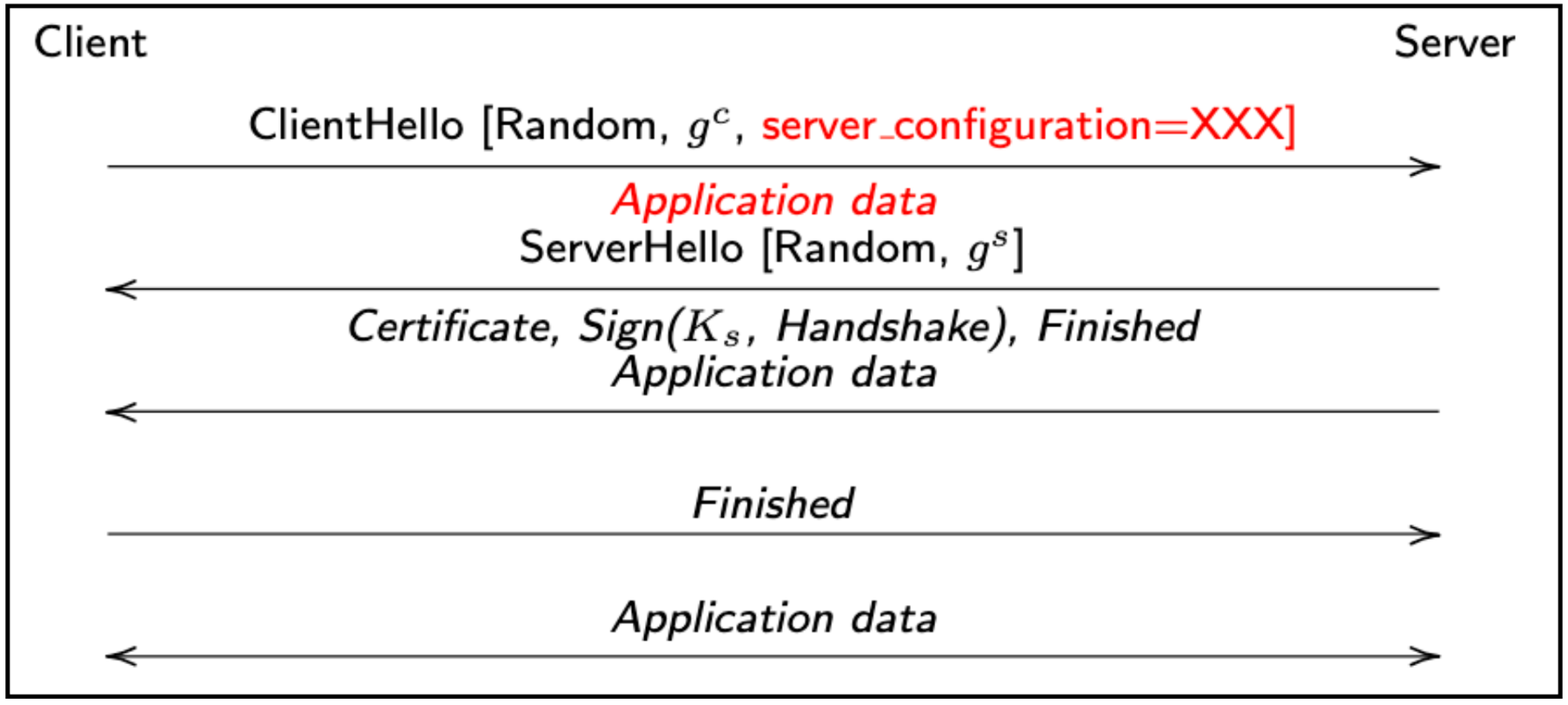
ClientKeyFinished | Application data

Server

- Similar to AKE4 (identity protection)
- **Only one round trip** before app data can be sent!

0-RTT Handshake

- New in TLS 1.3
- **Early data** is encrypted with keys from a previous sessions (marked so for applications)
- Keys renew after handshake finished
- Danger: replay of early data (left for application to handle)



Triple Handshakes and Cookie Cutters: Breaking and Fixing Authentication over TLS

Karthikeyan Bhargavan*, Antoine Delignat-Lavaud*, Cédric Fournet†, Alfredo Pironti* and Pierre-Yves Strub‡
*INRIA Paris-Rocquencourt †Microsoft Research ‡IMDEA Software Institute

***Abstract*—TLS was designed as a transparent channel abstraction to allow developers with no cryptographic expertise to protect their application against attackers that may control some clients, some servers, and may have the capability to tamper with network connections. However, the security guarantees of TLS fall short of those of a secure channel, leading to a variety of attacks.**

We show how some widespread false beliefs about these guarantees can be exploited to attack popular applications and defeat several standard authentication methods that rely too naively on TLS. We present new client impersonation attacks against TLS renegotiations, wireless networks, challenge-response protocols, and channel-bound cookies. Our attacks exploit combinations of RSA and Diffie-Hellman key exchange, session resumption, and renegotiation to bypass many recent countermeasures. We also demonstrate new ways to exploit known weaknesses of HTTP over TLS. We investigate the root causes for these attacks and propose new countermeasures. At the protocol level, we design and implement two new TLS extensions that strengthen the authentication guarantees of the handshake. At the application level, we develop an exemplary HTTPS client library that implements several mitigations, on top of a previously verified TLS implementation, and verify that their composition provides strong, simple application security.

sessions, validating certificates, etc. Meanwhile, TLS applications continue to rely on URLs, passwords, and cookies; they mix secure and insecure transports; and they often ignore lower-level signals such as handshake completion, session resumption, and truncated connections.

Many persistent problems can be blamed on a mismatch between the authentication guarantees expected by the application and those actually provided by TLS. To illustrate our point, we list below a few myths about those guarantees, which we debunk in this paper. Once a connection is established:

1) the principal at the other end cannot change;
2) the master secret is shared only between the two peers, so it can be used to derive fresh application-level keys;
3) the `tls-unique` channel binding [6] uniquely identifies the connection;
4) the connection authenticates the whole data stream, so it is safe to start processing application data as it arrives.

The first is widely believed to be ensured by the TLS renegotiation extension [49]. The second and third are used for man-in-the-middle protections in tunneled protocols like PEAP and some authentication modes in SASL and GSS-API. The fourth forms the basis of HTTPS sessions on the web.

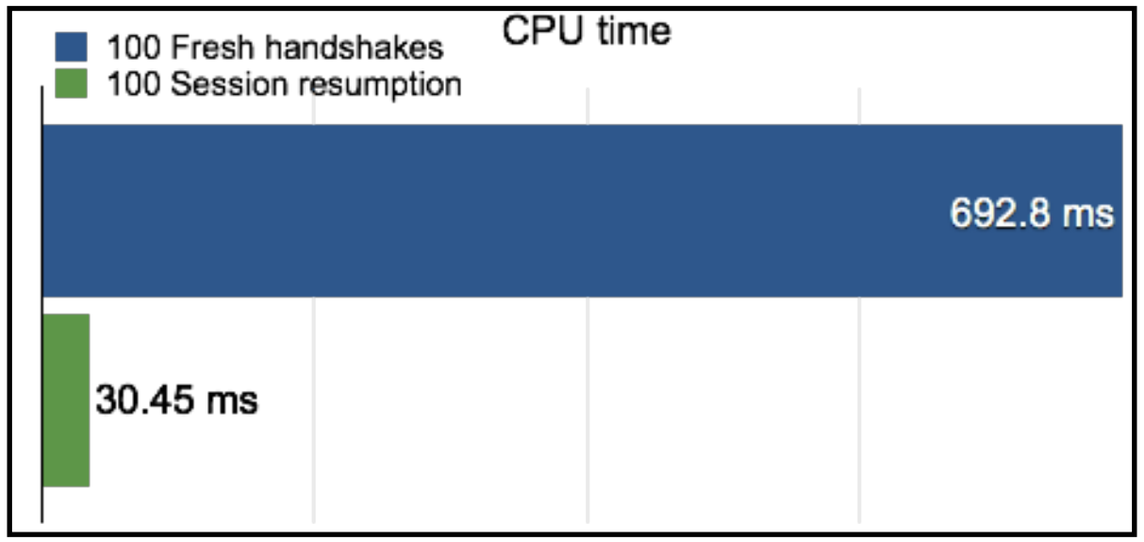
In IEEE S&P 2014

Feature/problem

- TLS supports **three** different types of handshakes
  - *full* handshake, which we have seen
  - resumption (*brief* handshake)
  - renegotiation (handshake amid an ongoing TLS session)

Session resumption (Why)

- Public-key cryptography (PKC) is comparatively expensive

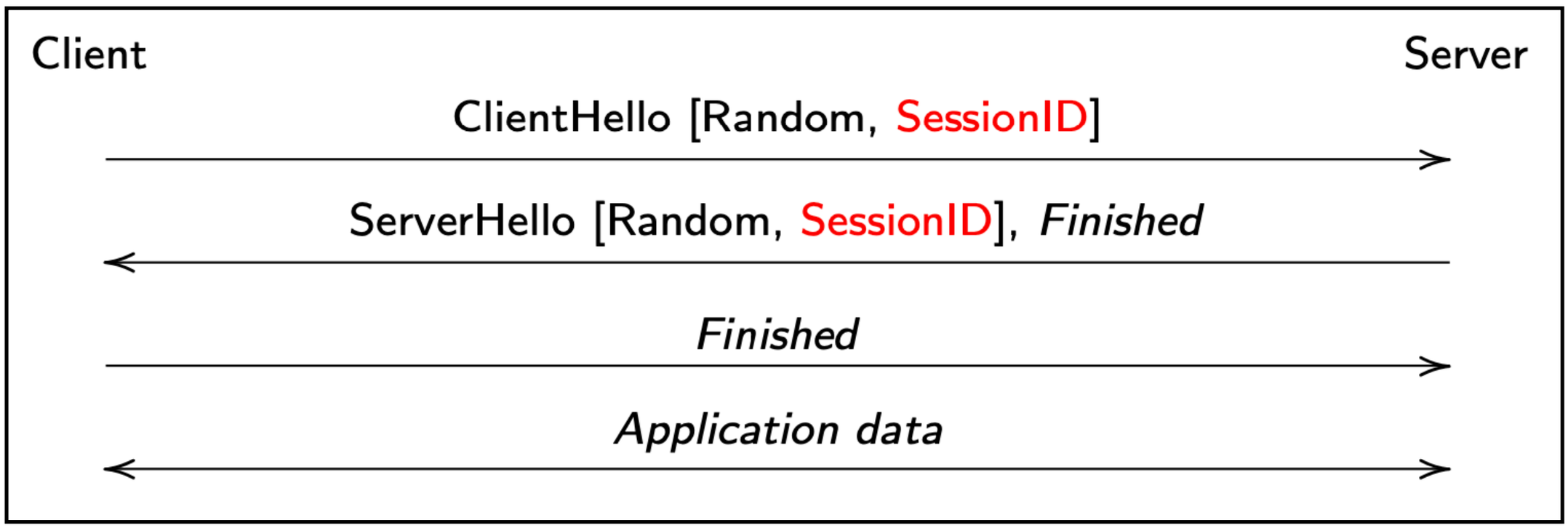


https://blog.cloudflare.com/tls-session-resumption-full-speed-and-secure/

- Resumption: re-use Master Secret in multiple sessions

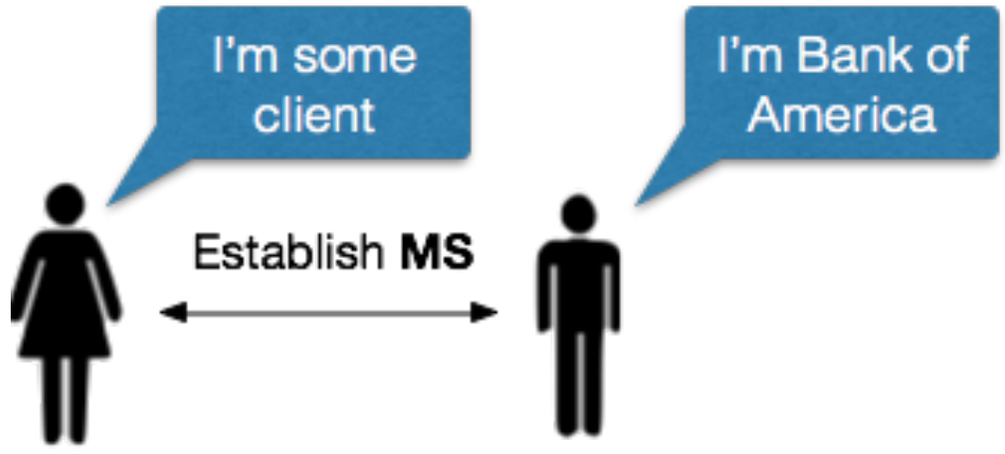
Session resumption (How)

- Server assigns a **session id (sid)** in ServerHello
- Client and server locally cache (MS, sid)
- Client can request to resume a session in the future.
  - Succeed if both party have same MS (Or Finished msgs won't match)
  - New keys are generated (b/c new random is used)
  - No PK operations



Renegotiation

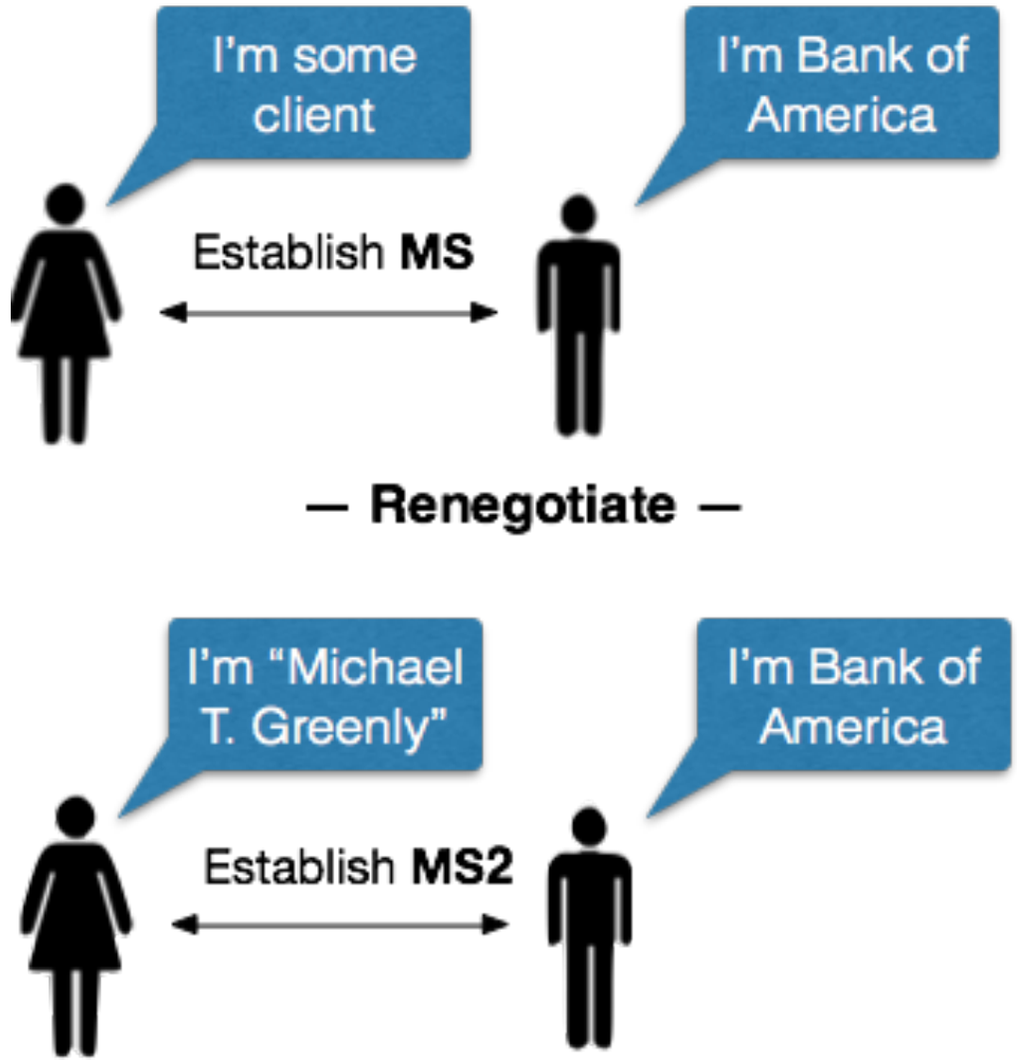
- What: handshake ***again*** in an ongoing session.



- Use cases:
  - Requiring client certificate privately
  - Refresh keys in long-lived sessions
  - Change cipher suites

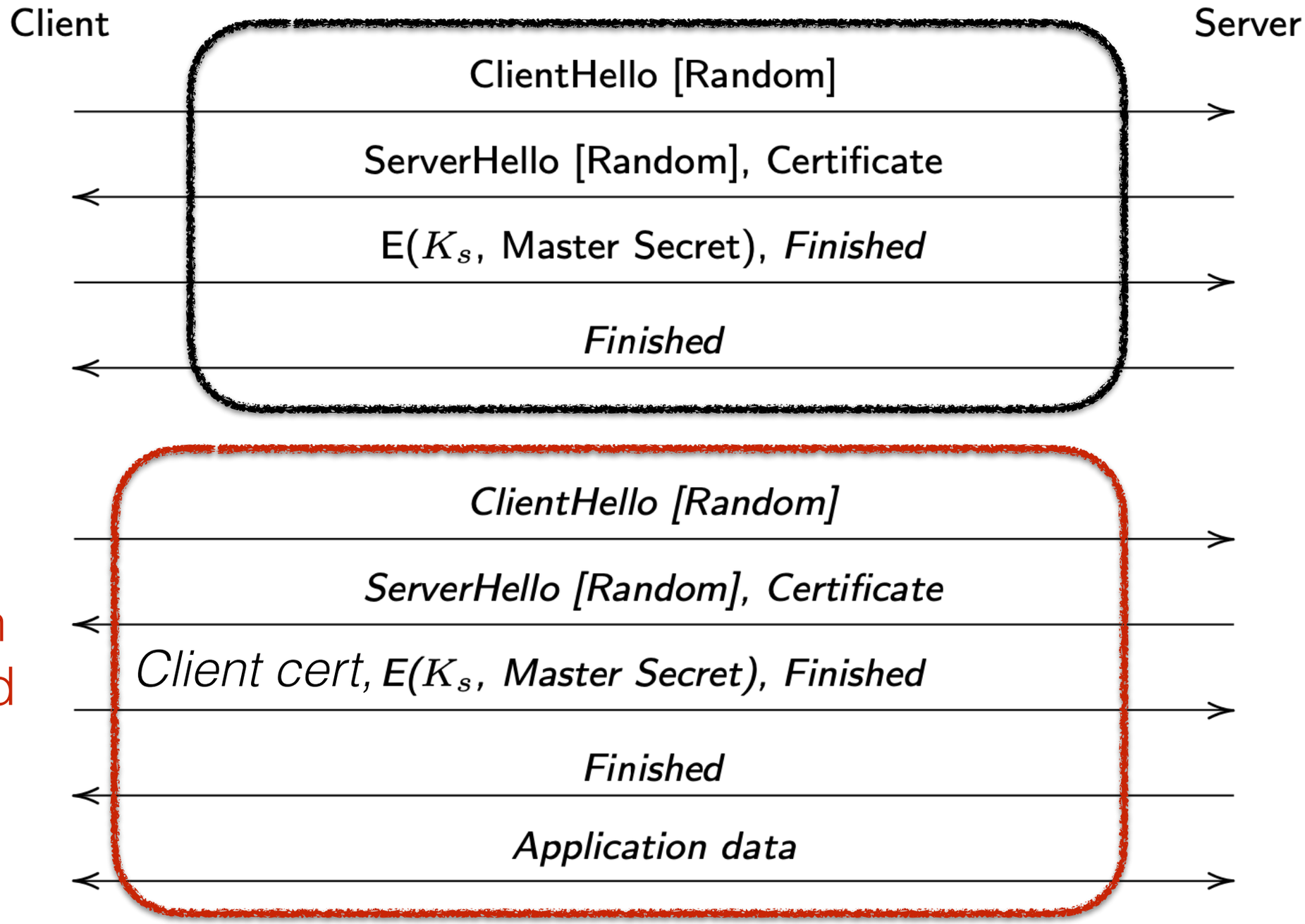
Renegotiation

- What: handshake ***again*** in an ongoing session.
- Use cases:
  - Requiring client certificate privately
  - Refresh keys in long-lived sessions
  - Change cipher suites



A server initially accepts an **anonymous** connection but later require the client to authenticate using a certificate to access **protected resources**.

Renegotiation (How)



1st HS

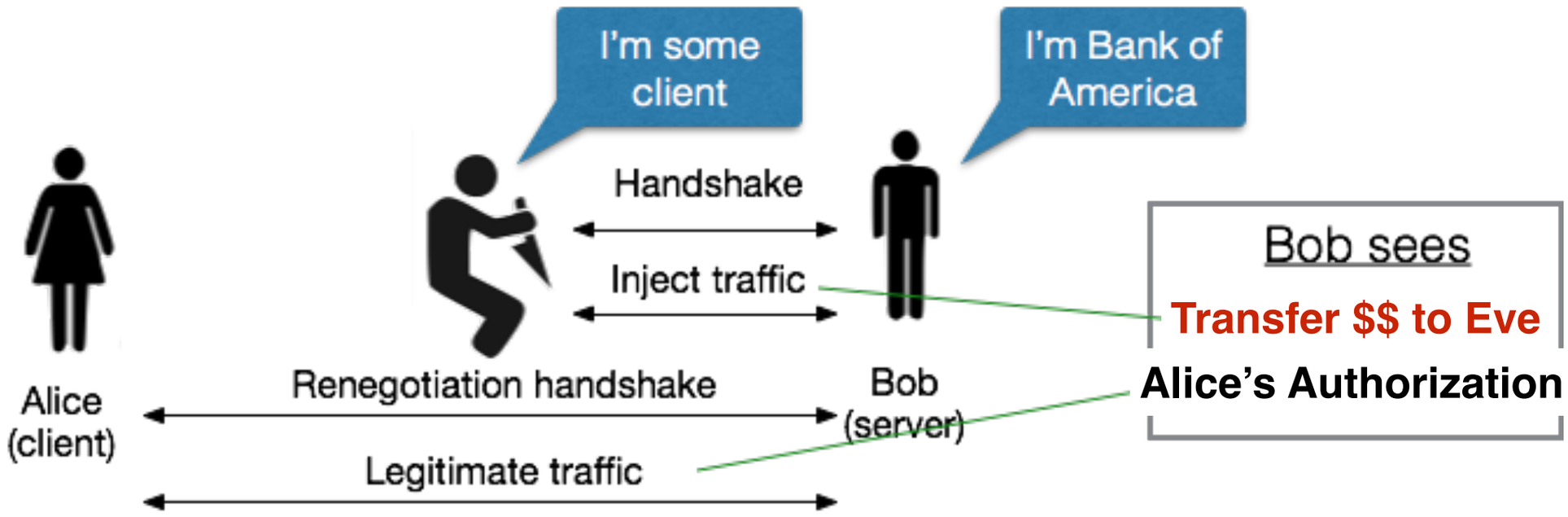
Renegotiation HS (encrypted using keys from 1st HS)

Problem: maybe two HSs are not done by the same client.

Renegotiation Attack (09)

This attack allows the attacker to **prefix the client traffic**.

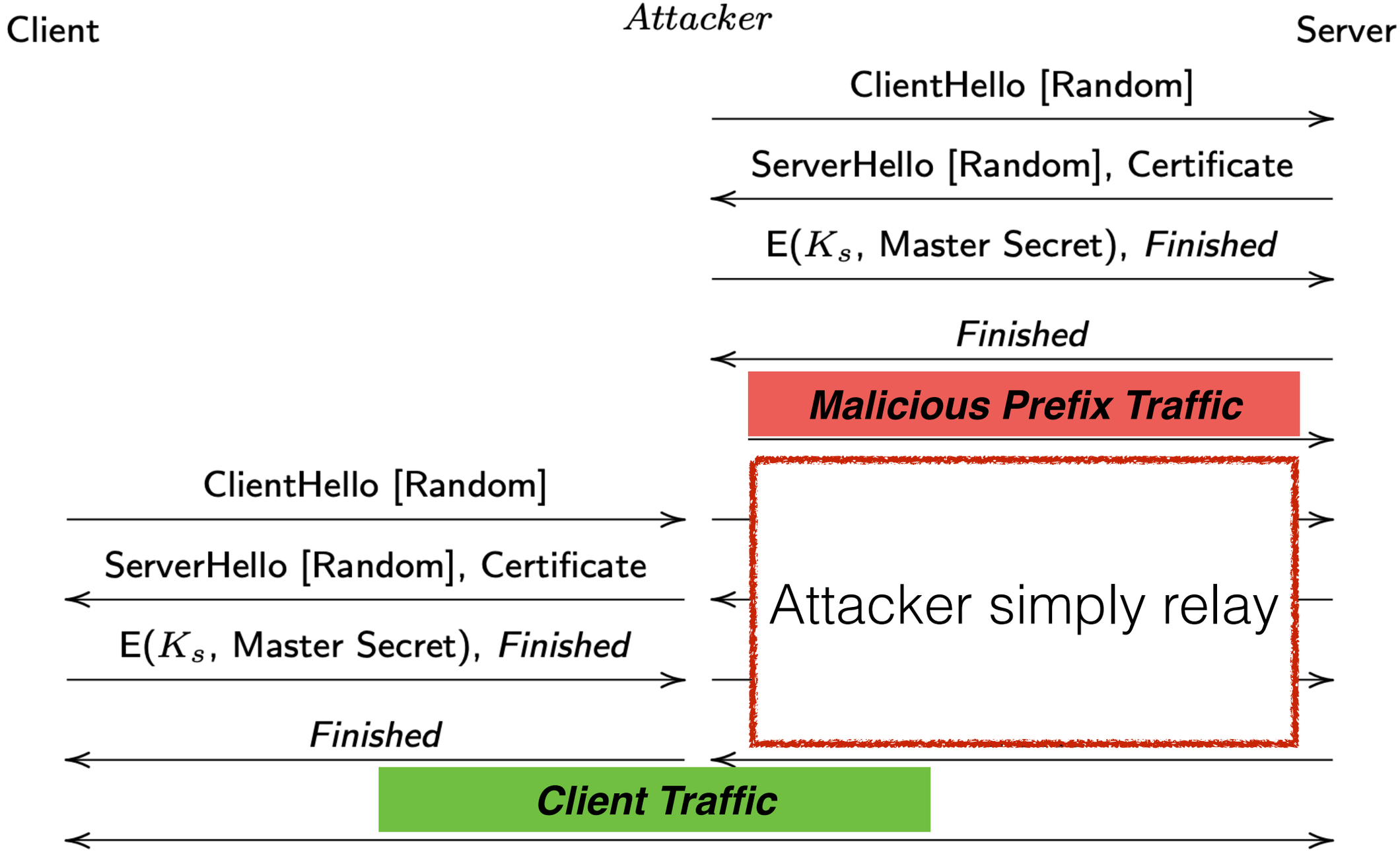
Server will see **malicious traffic + client traffic** in one session



1. Eve establishes a TLS channel (CH) with server; sends **prefix traffic**
2. Eve intercepts a unrelated handshake from Alice
3. Eve replays Alice's messages through CH
4. Server thinks that Eve is re-negotiating
5. Alice sends **client traffic**
6. Server thinks **malicious traffic + client traffic** coming from **Eve**

Figure from https://blog.cryptographyengineering.com/2014/04/

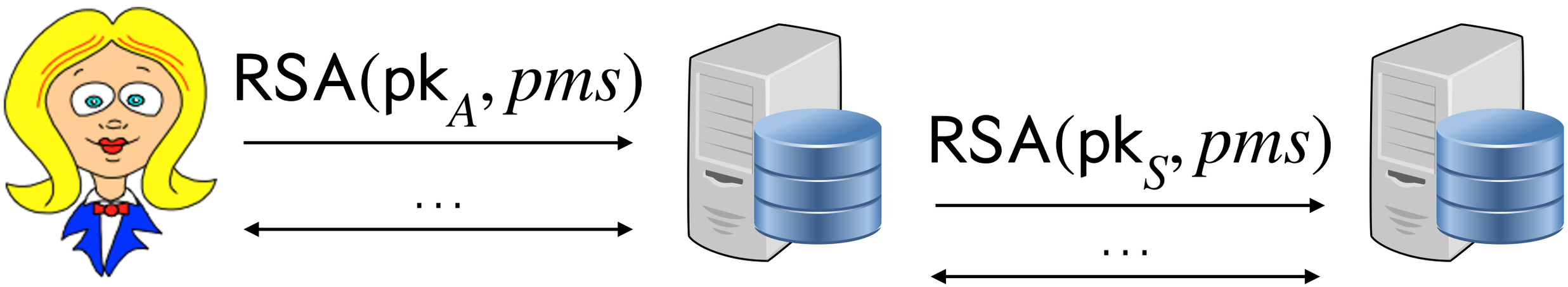
Detail


Client
Attacker
Server
ClientHello [Random]
ServerHello [Random], Certificate
E(K_s, Master Secret), Finished
Finished
Malicious Prefix Traffic
ClientHello [Random]
ServerHello [Random], Certificate
E(K_s, Master Secret), Finished
Finished
Attacker simply relay
Client Traffic

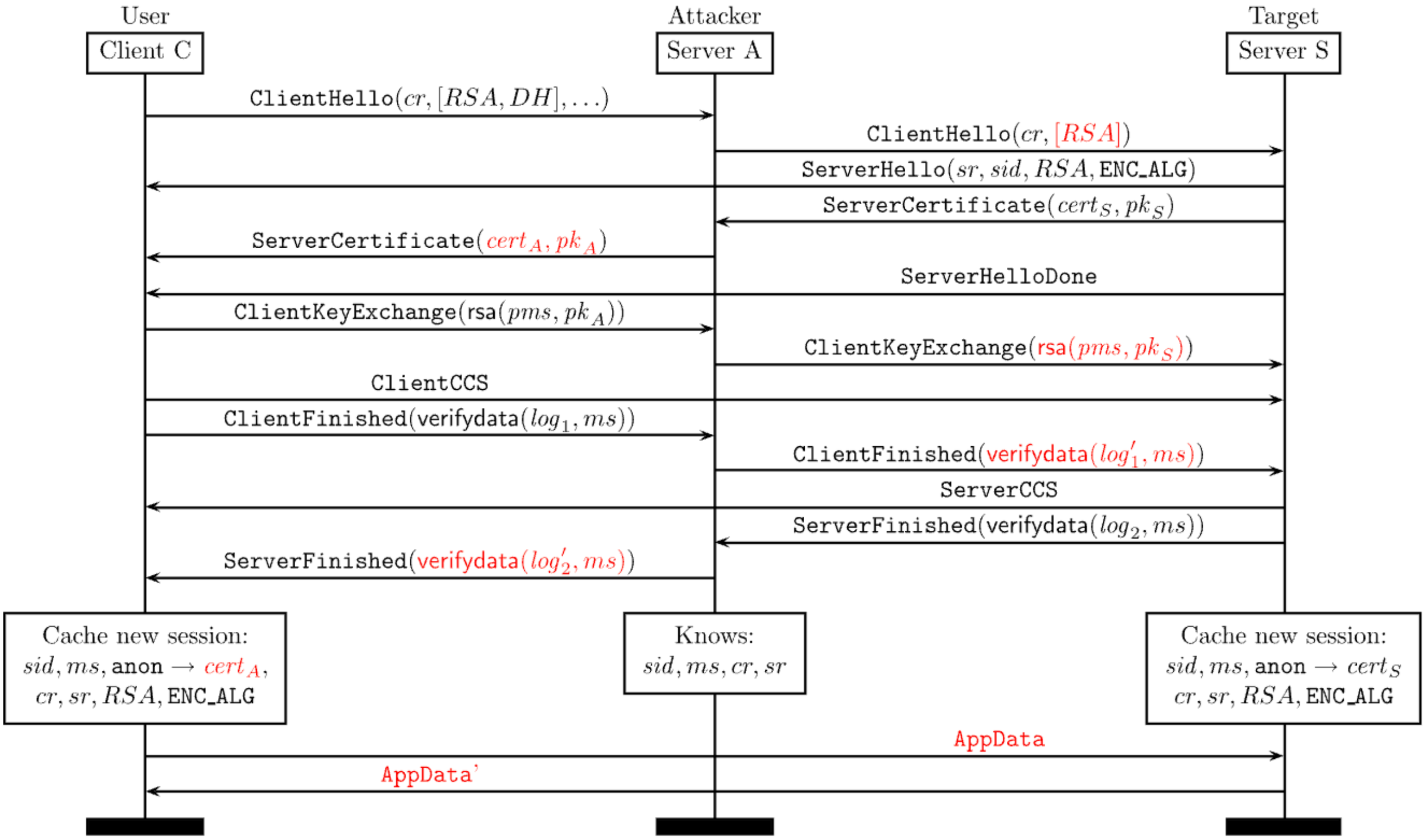
Fix (IETF RFC 5746)

- Renegotiation `ClientHello` in must include the Finished message of the enclosing channel
  - Eve replaying Alice's `ClientHello` won't work
  - 1) This `ClientHello does not point to a previous Finished`
  - `2) Even if it does, Alice's Finished is not the same as Eve's Finished`
- Assumption: Finished msgs are unique session identifiers. I.e., no two distinct HS will have the same Finished message.
- Shown to be wrong by "Triple Handshake" Attack

Step 1: Two separate sessions with the same key

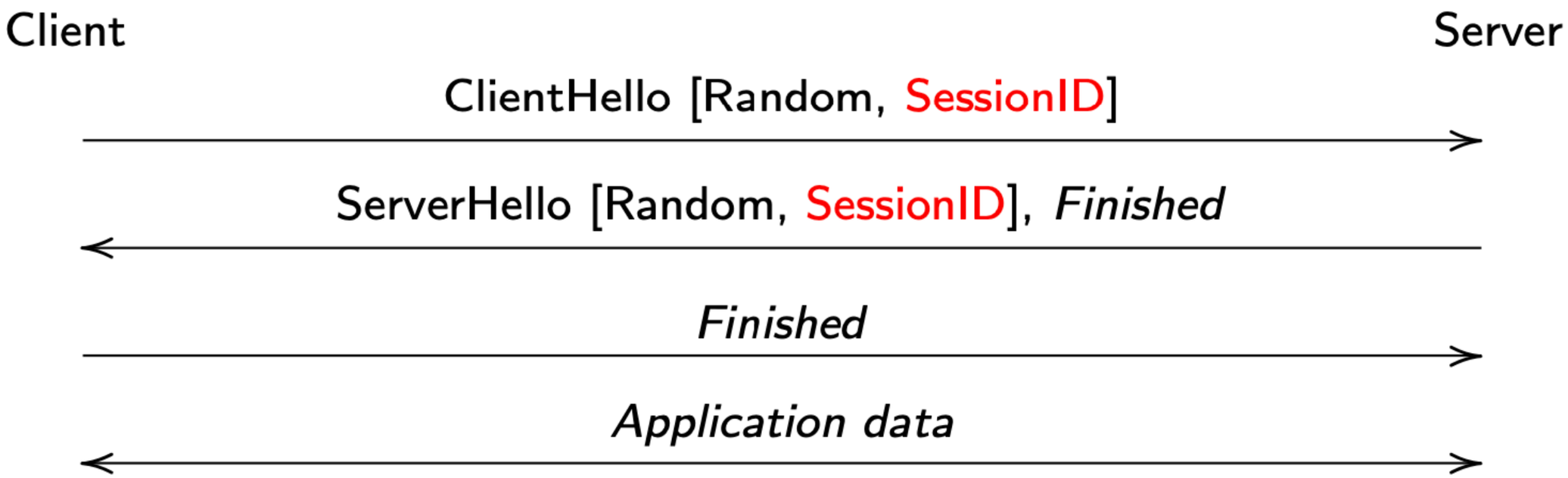


- Client connects to **malicious server A** and with pms
- A connects to another server B re-using pms
- A also re-uses randomness, so two sessions have **same keys**.
- Reminiscent of **identity misbinding**
- More details on next slide (same can be done w ECDHE)



These two sessions have **same keys**.
Finished messages are different though
(e.g., certificates are different).

But: Finished msgs in resumption HS does not depend on certs



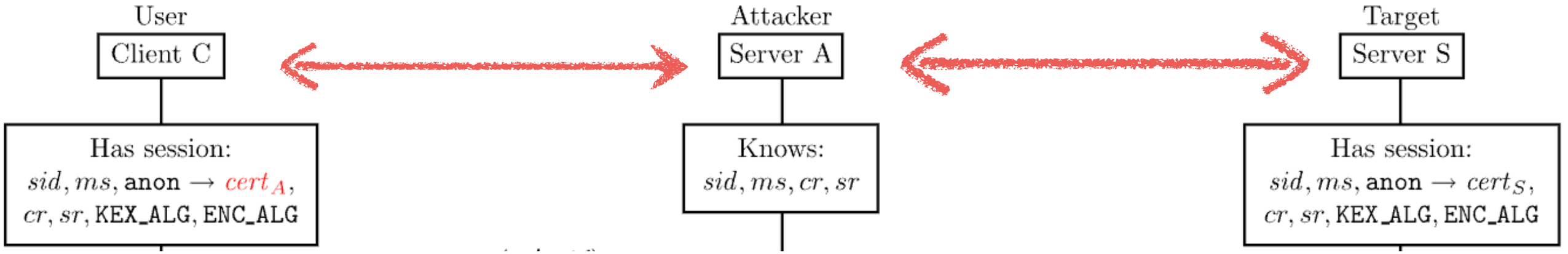
- In resumed sessions, Finished message only depends on r_c, r_s and session ID.

Attacker’s strategy

- Create two sessions (user - attacker, attacker - server) with the same MS; attacker terminates with server.
- Wait until client resume the 1st session
- *Replay* client msgs to server to resume its own session with server
- Both sessions now have **the same Finished message**
- Now, do the re-negotiation attack

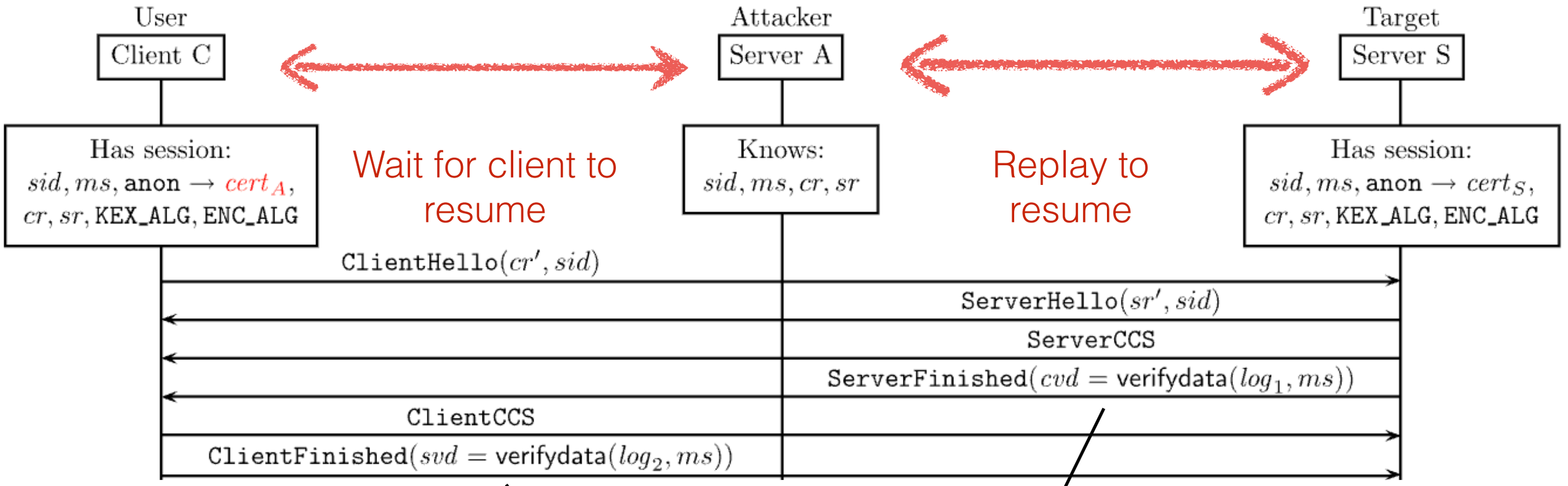
Step 2: Two sessions with same Finished msgs

So far: two sessions with the same MS



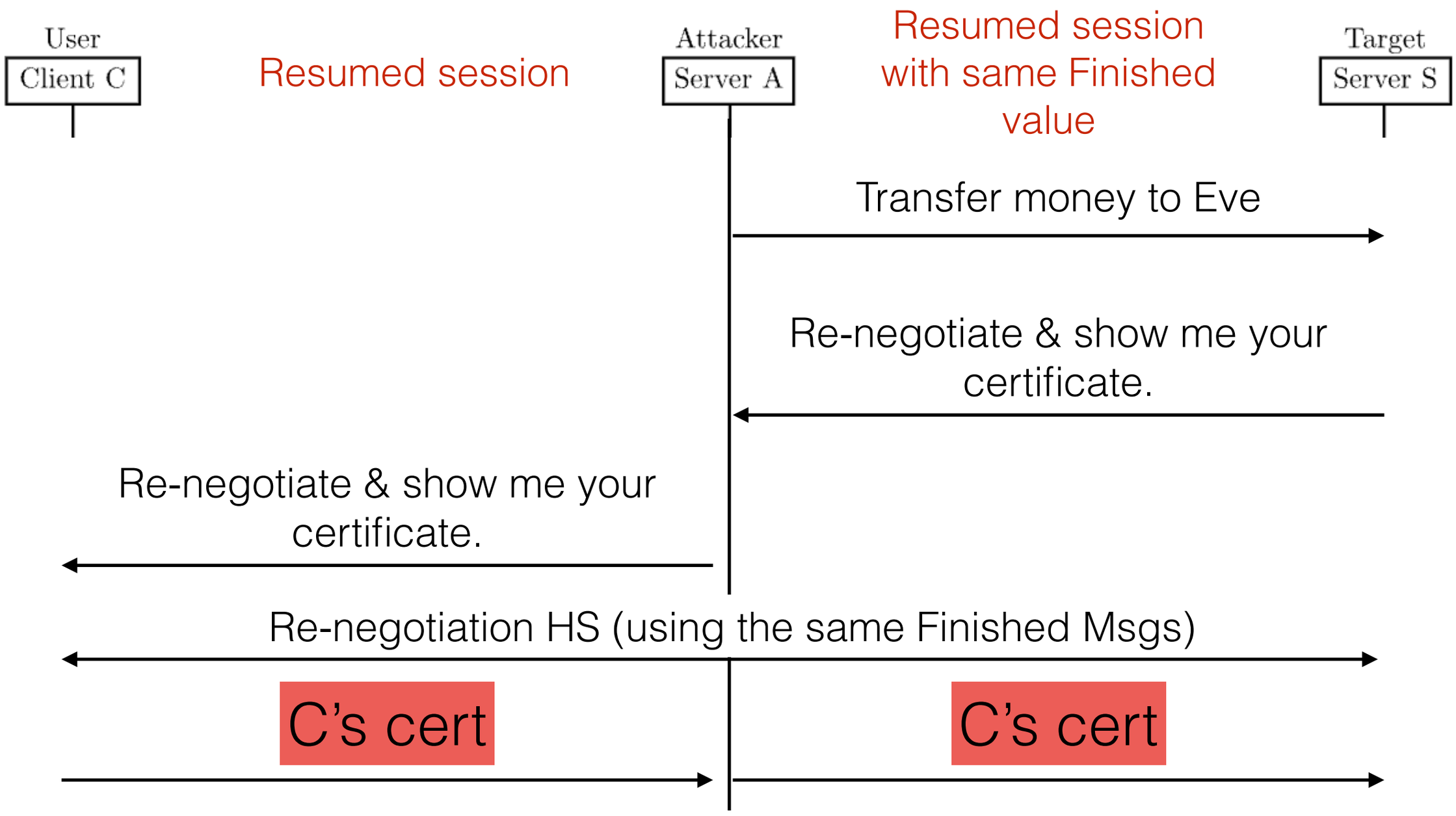
Step 2: Two sessions with same Finished msgs

So far: two sessions with the same MS



Post resumption, both channels have the same Finished messages

Step 3: Re-negotiation attack in resumed sessions (we've seen before)



What went wrong?

- The assumption that “no two distinct HS will have the same Finished message” is wrong
  - …due to subtle interaction between three kinds of handshakes (regular, re-negotiation, resumption)
- TLS 1.2 has been proven secure by **several papers**, but they ignore at least one kind of handshakes (or two).
- Formal verification usually ignore them too

TLS 1.3 objectives

- *Clean up:* Remove unused or unsafe features
- *Security:* Improve security by using modern security analysis techniques
- *Privacy:* Encrypt more of the protocol
- *Performance:* Our target is a 1-RTT handshake for naive clients; 0-RTT handshake for repeat connections
- *Continuity:* Maintain existing important use cases

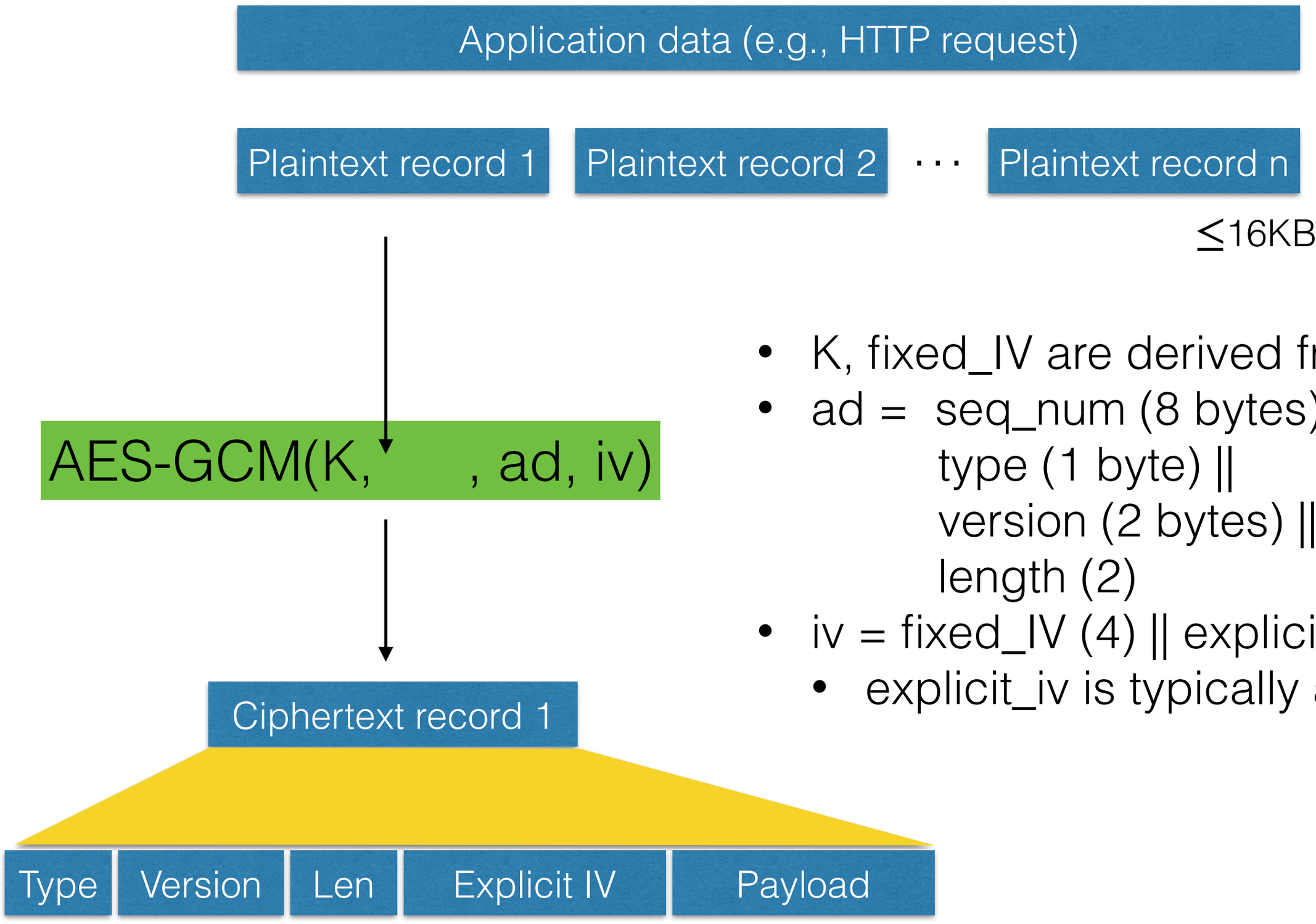
Removed features

- Static RSA
- Custom (EC)DHE groups
- Compression
- Renegotiation*
- Non-AEAD ciphers
- Simplified resumption

*Special accommodation for inline client authentication


INTERMISSION

Attacks against TLS encryption



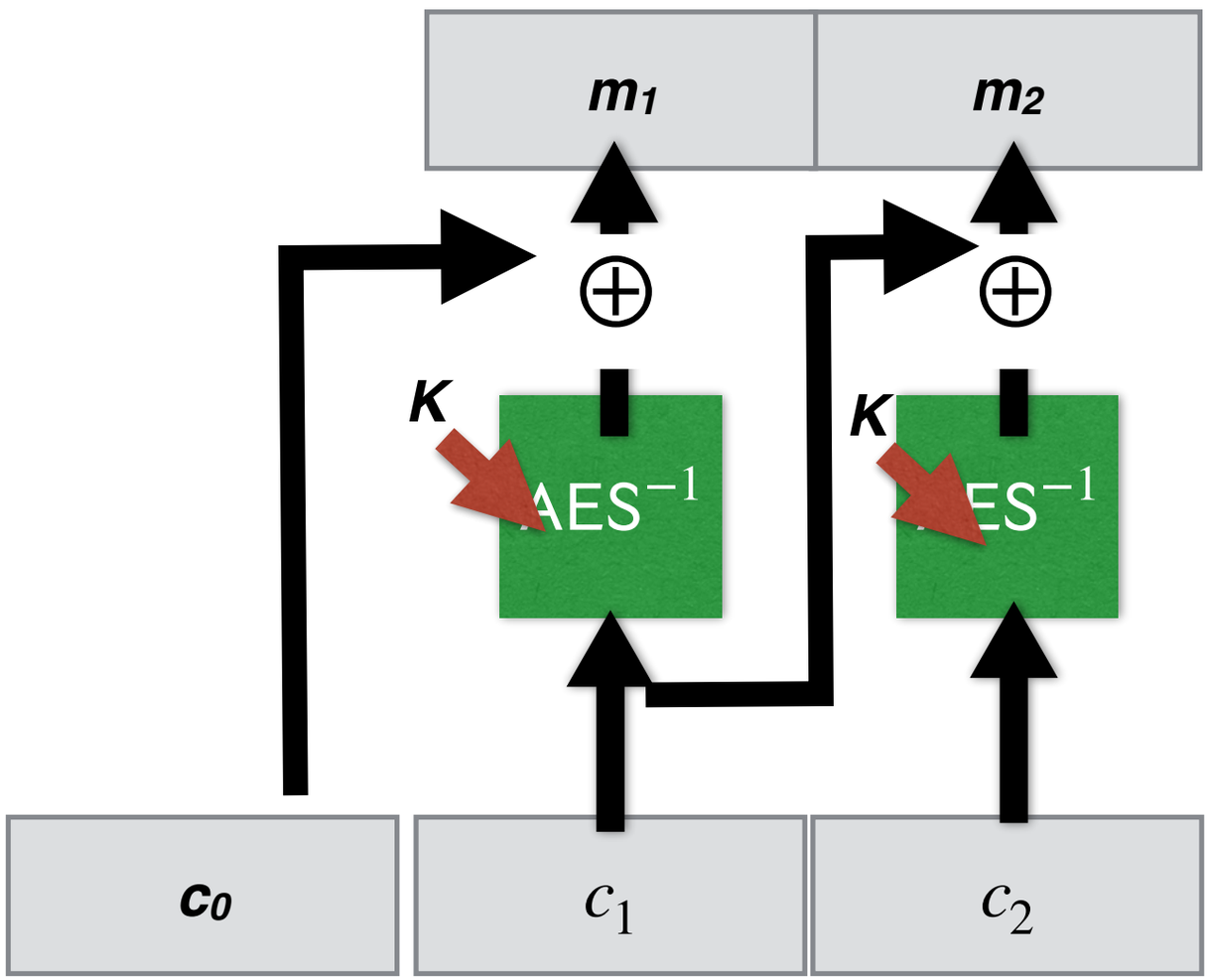
- K, fixed_IV are derived from MS
- ad = seq_num (8 bytes) ||
  type (1 byte) ||
  version (2 bytes) ||
  length (2)
- iv = fixed_IV (4) || explicit_iv (8)
  - explicit_iv is typically a counter

Ciphertext records are send over TCP to the destination …

Warm up: Encrypted Email

- Consider an encrypted mail service.
- Emails $m = (m_{\text{addr}}, m_{\text{body}})$ are encrypted under a key known only to the mail server.
- Upon receiving a ciphertext, the mail server decrypts it to get $(m_{\text{addr}}, m_{\text{body}})$, and forward m_{body} to the address in m_{addr}.
  - Let's say encryption is AES-CBC
  - for simply, assume m_{addr} is in a separate AES block.
- **Now: Eve intercepted an encrypted email addressed to Bob. How can she read it?**

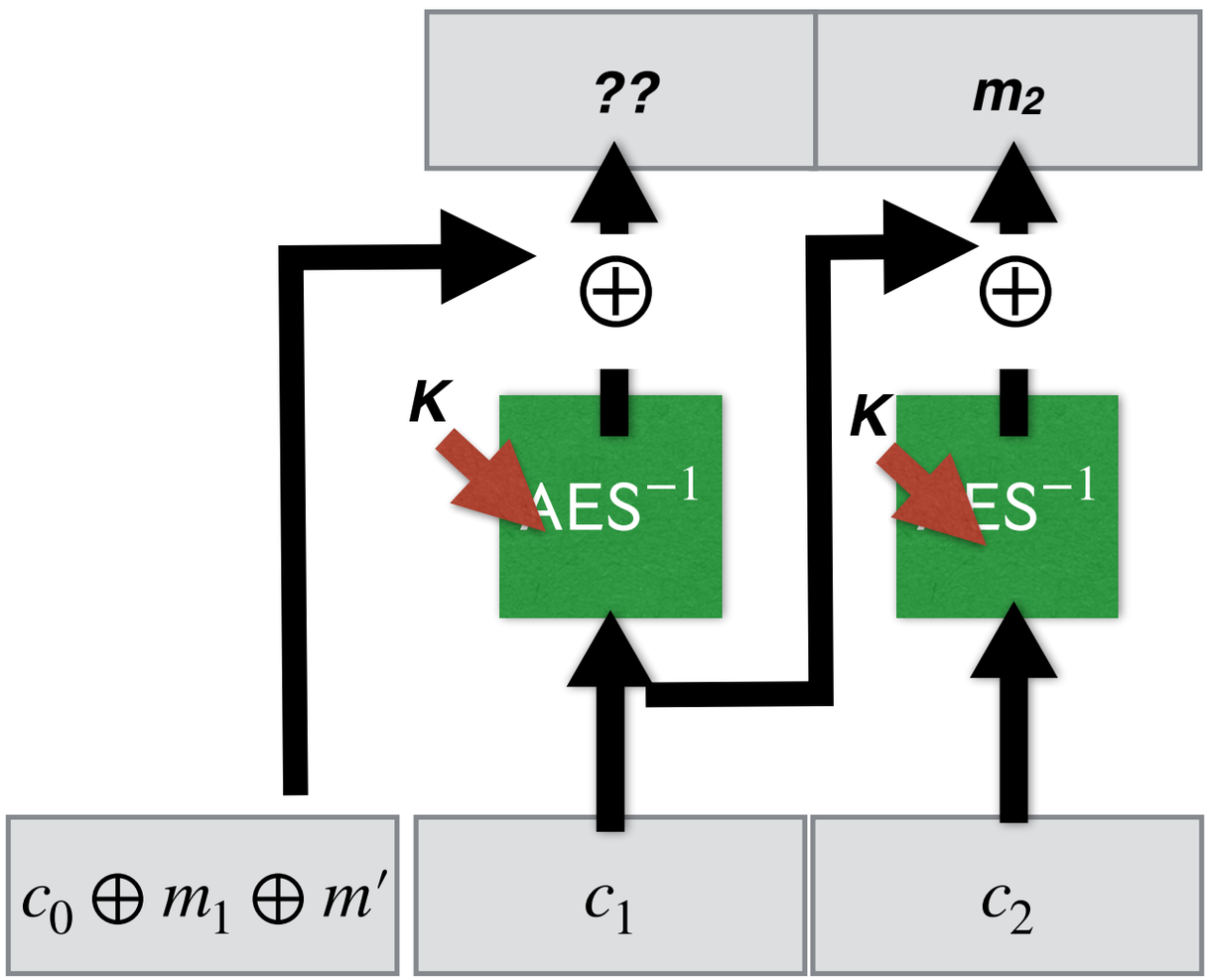
Warm up: Encrypted Email



- Eve has an a ciphertext (c_0, c_1, c_2) for (m_1, m_2) where
  - m_1 = **bob@mail.com**
  - m_2 is the body.
- To trick the mail server to decrypt for Eve, Eve needs a ciphertext that decrypts to (m_1', m_2) where
  - m_1' =eve@mail.com
- Exploit the decryption logic.

$$m_1 = \mathsf{AES}^{-1}(K, c_1) \oplus c_0$$

Warm up: Encrypted Email



- Answer: Eve just needs to send $(c_0 \oplus m_1 \oplus m', c_1, c_2)$ to the mail server.

- First block decrypts to

$$\mathrm{AES}^{-1}(K, c_1) \oplus c_0 \oplus m_1 \oplus m'$$

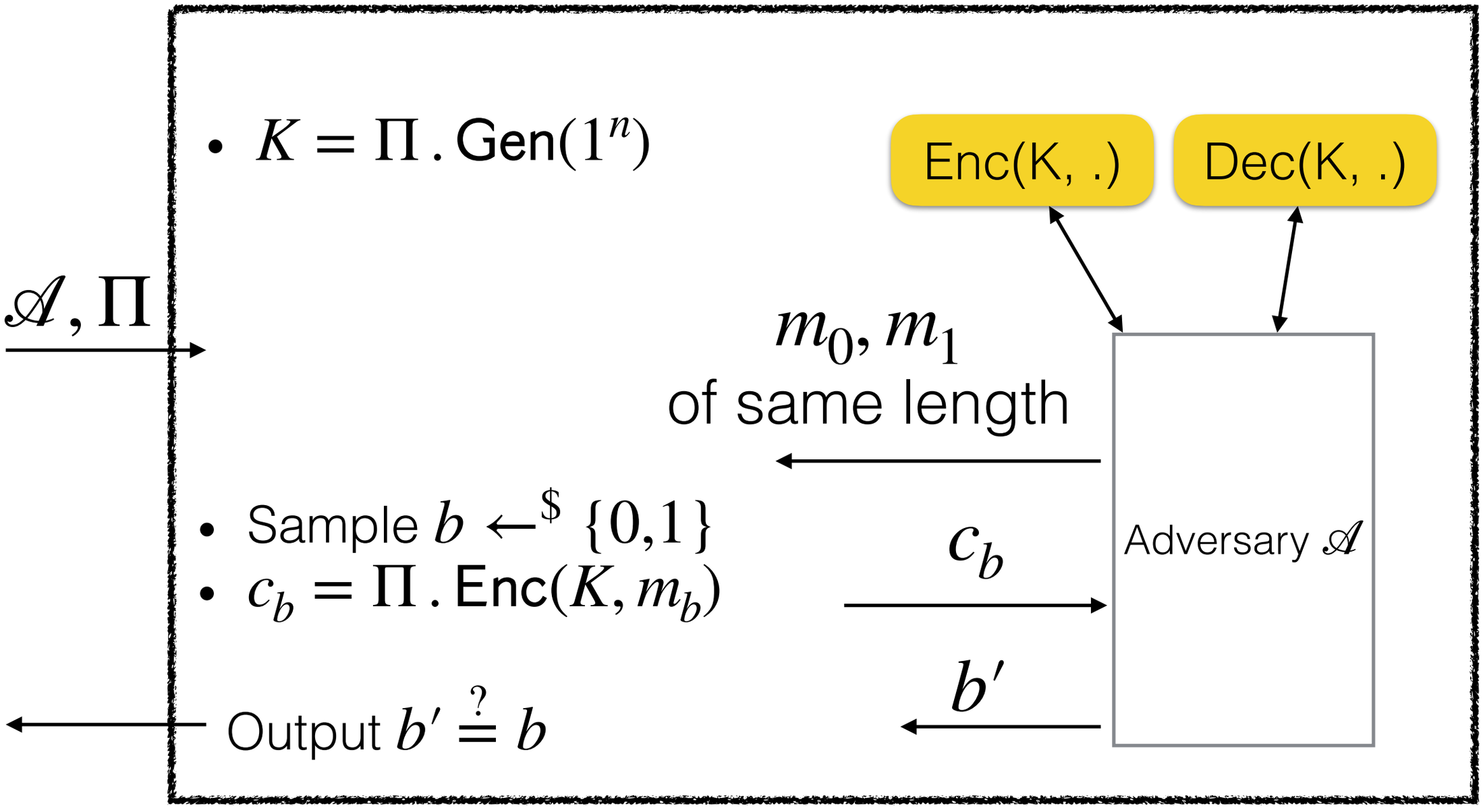
$$= m_1 \oplus m_1 \oplus m'$$

$$= m'$$

Warm up: Encrypted Email

- What just happened? Why can we break a secure cipher?
- Eve is given the ability to decrypt *related ciphertext*
- Chosen-ciphertext Attack (CCA)
- Lack of **ciphertext integrity** *can* completely compromise secrecy.
  - Decrypting related msgs is not always problematic (in which case CPA may suffice), but when it is, you need CCA security.

Defining CCA game



- A is given both encryption and decryption oracles
- A is not allowed to decrypt challenge cipher text

Pop quiz (2 min)

- Recall that $\mathsf{Enc}(K, m) := (r \leftarrow^{\$}, \mathsf{AES}(K, r) \oplus m)$ is CPA secure.
- Given an adversary that wins the CCA game with probability 1.

- Adv chooses $m_0 = 0^n, m_1 = 1^n$
- When receives a challenge $c_b = (r_b, s_b)$, Adv flip first bit of s_b to get s'_b, and call decryption oracle with (r_b, s'_b)
- Result is either 100.. or 011…
- Output 0 in the former case, and 1 otherwise.

MAC-then-encrypt

In SSL 3.0, TLS 1.0, and 802.11i WiFi encryption protocol.

- mac-then-enc(k, m):
  - $k_e, k_m \leftarrow \mathsf{KDF}(k)$
  - $t := \mathsf{Mac}(K_m, m)$
  - $c := \mathsf{Enc}(K_e, m \| t)$
  - Output c

This approach is not secure in general.

Padding oracles attack in SSL 3.0

- Padding in SSL 3.0:
  - Add paddings so msg || padding is multiple of 16 bytes
    - if a pad of p bytes is needed, add p-1 random bytes, then (p-1)
      - E.g., `de ad be ef 04` (a 5-byte pad)
    - If no pad is needed, add a dummy block of 15 random bytes, and 15.

Padding oracles attack in SSL 3.0

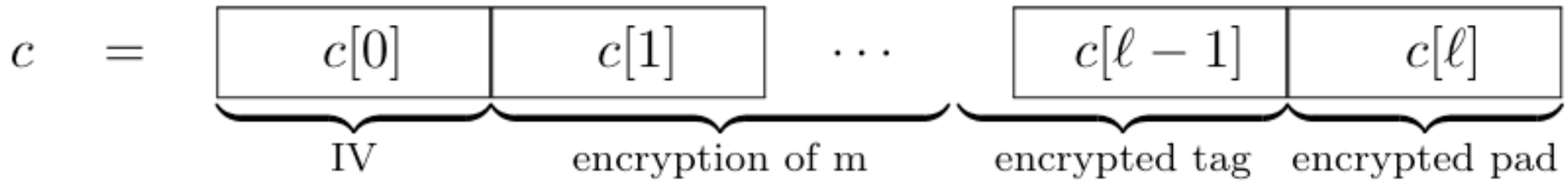
- Encryption with AES-CBC
  - Compute tag and add padding

  - Encrypt the above using CBC
- Decryption by server
  - CBC decryption. Pop last byte (=b), skip b bytes. If tag verifies, pass msg for processing. Otherwise, alert sender with `reject`.

CCA attacker will try to make use of this information.

Padding oracles attack in SSL 3.0

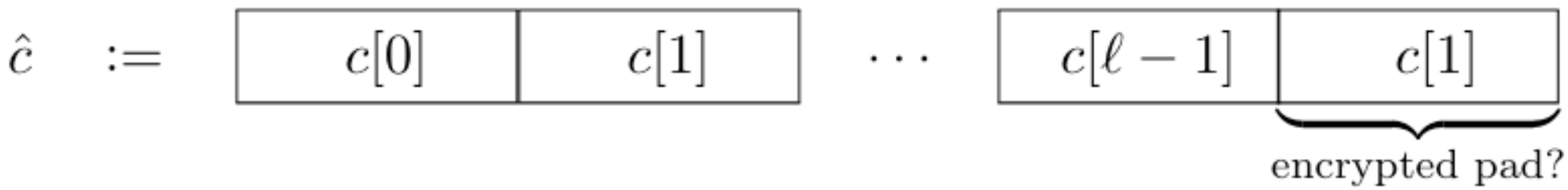
- **Assuming** len(m||t) is a multiple of 16, so a full dummy pad is added.



- Basic idea: sends any ciphertext c' to the server. If no error: last byte of decryption = 15.

A simple example

Replacing the last block with $c[1]$



After CBC decryption of $\hat{c}$, the last block is

$AES^{-1}(c[1]) \oplus c[\ell\text{-}1] = m[1] \oplus IV \oplus c[\ell\text{-}1]$

- If no error, $LB(m[1] \oplus IV \oplus c[\ell\text{-}1]) = 15$, so LB(m[1]) can be recovered.
- This is the starting point of a complete CCA attack that can decrypt the whole thing.
- SSL 3.0 and TLS 1.0 completely broken.

Padding Oracle Attacks

- General flow:
    1. Attacker intercepts ciphertext c
    2. Craft new ciphertext c’ and ask the system to decrypt. System leaks if the decryption was correctly padded**,** through explicit errors, or timing side channels (Luck13).
    3. Recover information about plaintext based on whether or not c’ triggers an error
- Led to full plaintext recover against SSL 3.0, TLS 1.0.

The Lessons

- Moxie's Doom Principle
- "When it comes to designing secure protocols, I have a principle that goes like this: **if you have to perform *any* cryptographic operation before verifying the MAC on a message you've received, it will somehow inevitably lead to doom.**"

Co-author of the Signal Protocol
Ex head of security @ Twitter

https://moxie.org/2011/12/13/the-cryptographic-doom-principle.html

Encrypt-then-MAC (EtM)

Supported by TLS 1.2 with AES-CBC-HMAC.

- Enc-then-mac(k, m):
  - $k_e, k_m \leftarrow \mathsf{KDF}(k)$
  - $c := \mathsf{Enc}(K_e, m)$
  - $t := \mathsf{Mac}(K_m, c)$
  - Output (c, t)
- Satisfies the Doom Principle

Authenticated Encryption

- Even better is Authenticated Encryption with Associated Data (AEAD)
  - Better because it's a one-stop solution
- You should now be familiar with AES-GCM
  - Another example: ChaCha20-Poly1305

- **TLS 1.3 only supports AEAD cipherts**

Security definition for AE

- A cipher is **Authenticate Encryption (AE)** if it 1) is IND-CPA secure; ii) providers cipher text integrity.
- Extends to AEAD naturally
- New concept **cipher text integrity**: A cipher provides cipher text integrity if no PPT adversary can produce a new valid cipher text , even after obtaining many cipher text.
  - valid = decryption doesn’t return err
  - Formal game is akin to that of MAC

AE implies CCA

- If cipher $\mathscr{E}$ is AE, then $\mathscr{E}$ is also CCA [Thm 9.1 in BV]
- Proof idea
  - For any adversary $\mathscr{A}$, decryption oracle must responds with “err” (or $\mathscr{A}$ wins CI game)
  - Then decryption oracle is useless to $\mathscr{A}$
  - $\mathscr{A}$ is in fact playing the CPA game
  - $\mathscr{A}$ can’t win, because $\mathscr{E}$ is CPA secure (by def)

Adoption of TLS 1.3

- TLS 1.3 is gaining fast adoption though there will probably be a long tail.

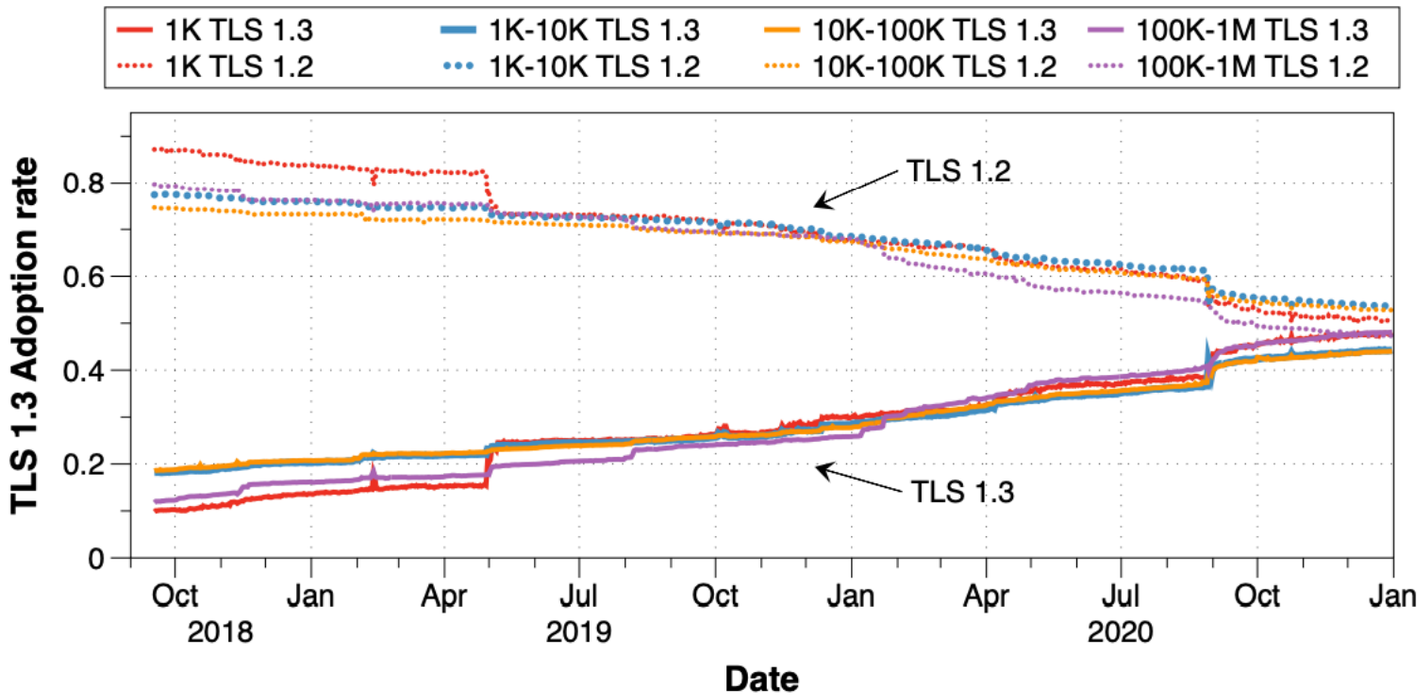


Figure 2: TLS 1.3 adoption ratio by Alexa rank. There is no significant difference in the trend of TLS 1.3 adoption between Alexa top 1K, 10K, 100K, and 1M sites.

https://userlab.utk.edu/files/papers/kim/2021/lee2021tls.pdf

Summary of TLS

- TLS allows two parties to establish *secure channels* over untrusted network for interactive communication
- Handshake protocol + record protocol
  - AKE: Use public key crypto (RSA, El Gamal/ECDHE) to share on a symmetric key. Reply on certificates to authenticate the server.
  - Authenticated Encryption**:** Use symmetric key and AEAD to encrypt and authenticate app data.
- Real-world features made TLS popular, but also complex.
- There are other secure channel protocols too, some simpler. E.g., The Noise Framework.