TLS

Kenny Paterson
Applied Cryptography Group
ETH Zurich

Overview

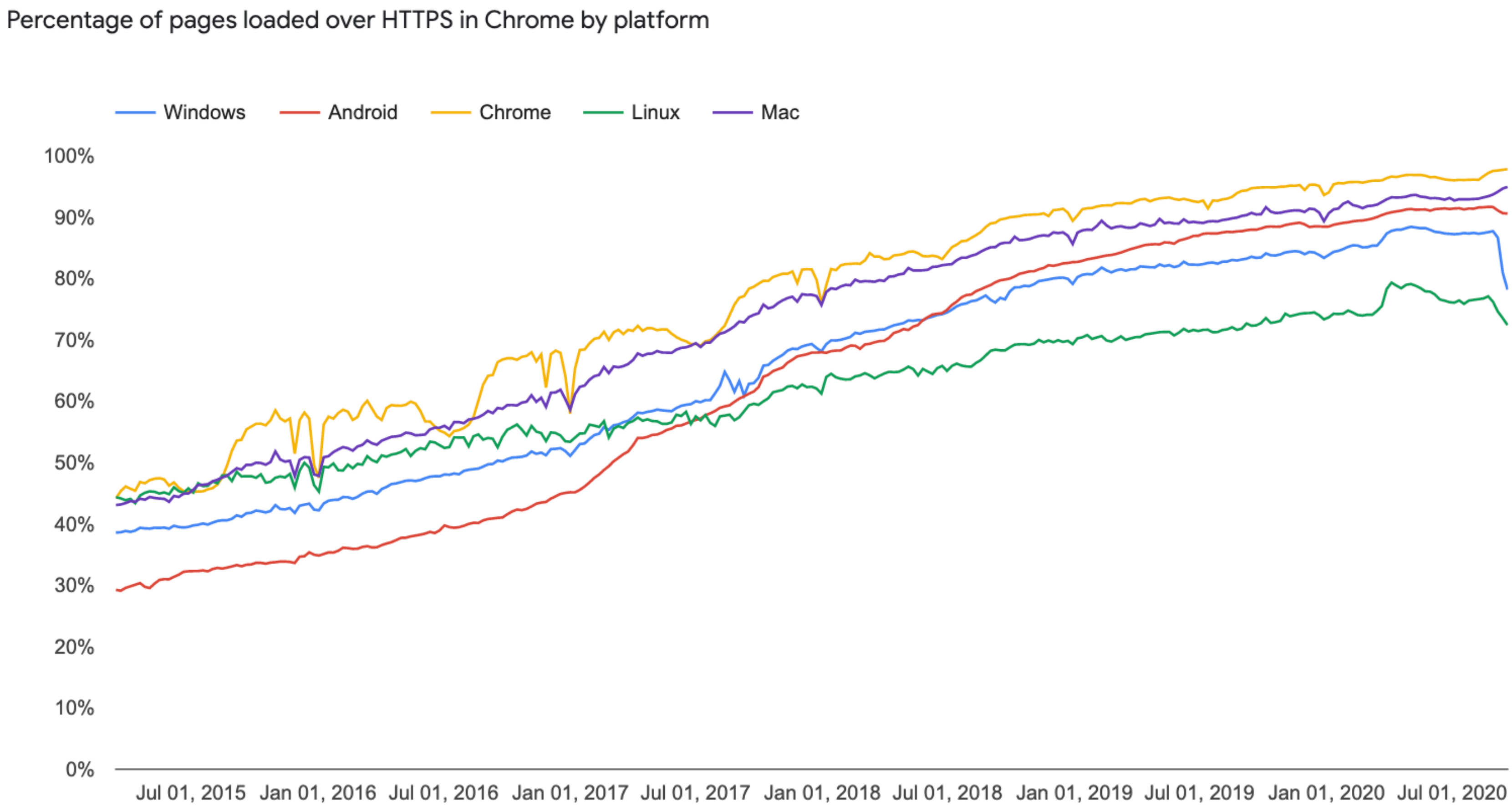
- Introducing TLS
- Motivation for TLS 1.3
- TLS 1.3 Record Protocol
- TLS 1.3 Handshake Protocol
- TLS 1.3 resumption and 0-RTT feature
- The future of TLS

Introducing TLS

Importance of TLS

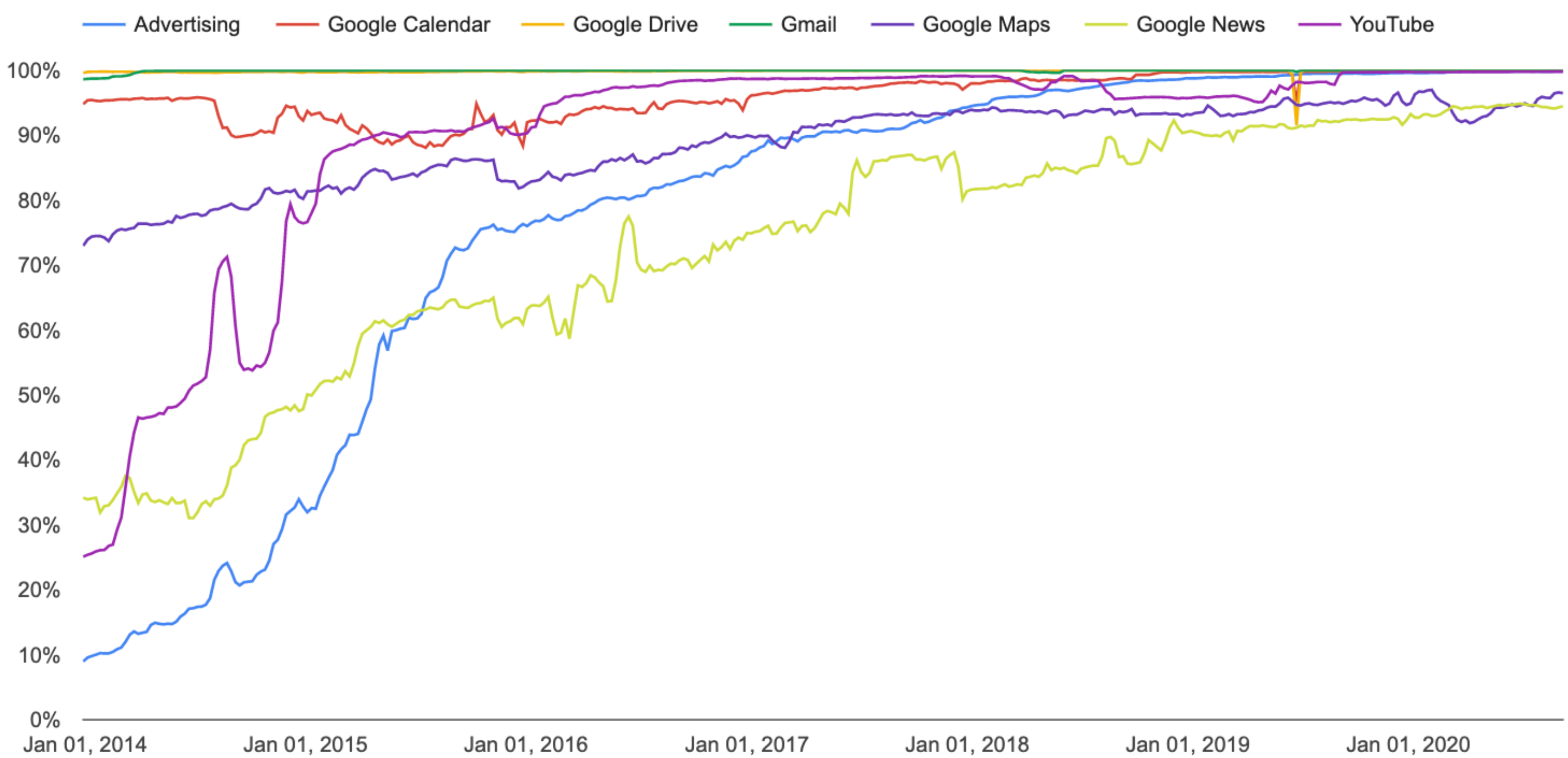
- Originally designed for secure e-commerce, now used much more widely.
  - Retail customer access to online banking facilities.
  - Access to gmail, facebook, Yahoo, etc.
  - Mobile applications, including but not only banking apps.
  - Payment infrastructures.
  - Back-end operations of large organisations, e.g. Google.
- TLS has become the *de facto* secure communications protocol of choice.
  - Used by hundreds of millions (billions?) of people and devices every day.

Importance of TLS in the web



https://transparencyreport.google.com/https/overview?hl=en, accessed 27/09/2020.

Importance of TLS more generally (at Google)



https://transparencyreport.google.com/https/overview?hl=en, accessed 27/09/2020.

SSL and TLS versions

- SSL = Secure Sockets Layer.
  - Developed by Netscape in mid 1990s.
  - SSLv1 broken at birth (1994).
  - SSLv2 (1995) seriously flawed in various ways, now IETF-deprecated (RFC 6176).
  - SSLv3 (1996) now considered broken (POODLE + RC4 attacks).
- TLS = Transport Layer Security.
  - IETF-standardised version of SSL.
  - TLS 1.0 in RFC 2246 (1999): translation of SSL 3.0.
  - TLS 1.1 in RFC 4346 (2006): security tweaks.
  - TLS 1.2 in RFC 5246 (2008): security tweaks, AEAD.
  - TLS 1.3 in RFC 8446 (2018): major re-design.

TLS – High Level Goals

"The primary goal of TLS is to provide a **secure channel between two peers**"

\- TLS 1.3 [RFC 8446]

Entity authentication:

- **Server** side of the channel is **always* authenticated.**
- **Client** side is **optionally authenticated.**
- Via **asymmetric crypto** (e.g., signatures) or a symmetric **pre-shared key.**

Confidentiality:

- **Data** sent over the channel is **only visible to the endpoints.**
- TLS does **not hide the length** of the data it transmits (but allows padding).

Integrity:

- **Data** sent over the channel **cannot be modified** without detection.
- Integrity guarantees also cover reordering, insertion, deletion of data.

TLS aims for security in the face of **attacker who has complete control of the network.**

Only requirement from underlying transport: reliable, in-order data stream.

TLS – Secondary Goals

Efficiency:

- Attempt to minimise crypto overhead.
- Minimal use of public key techniques; maximal use of symmetric key techniques.
- Minimise number of communication round trip before secure channel can be used.

Flexibilty:

- Protocol supports flexible choices of algorithms and authentication methods.
- Differences between TLS 1.3 and earlier versions.

Self-negotiation:

- The choice is done "in-band", i.e. as part of the protocol itself.
- This is done through the version negotiation and cipher suite negotiation process: client offers, server selects.

Protection of negotiation:

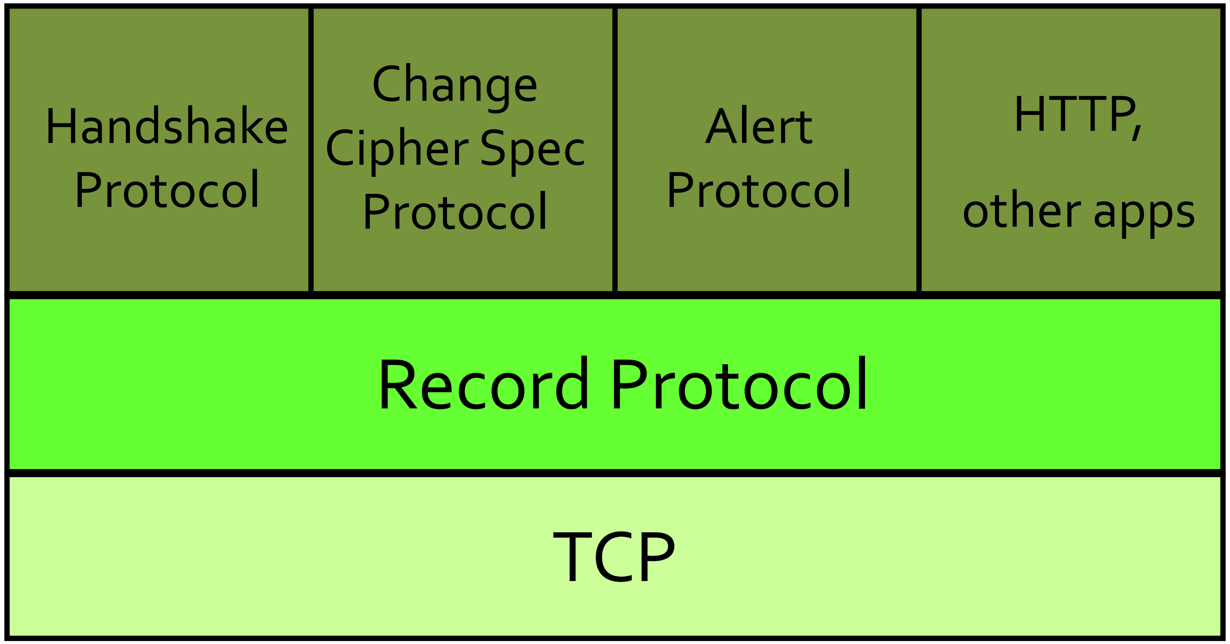
- Aim to prevent MITM attacker from performing version and cipher suite downgrade attacks.
- So the cryptography used in the protocol should also protect the *choice* of cryptography made.

Highly Simplified View of TLS

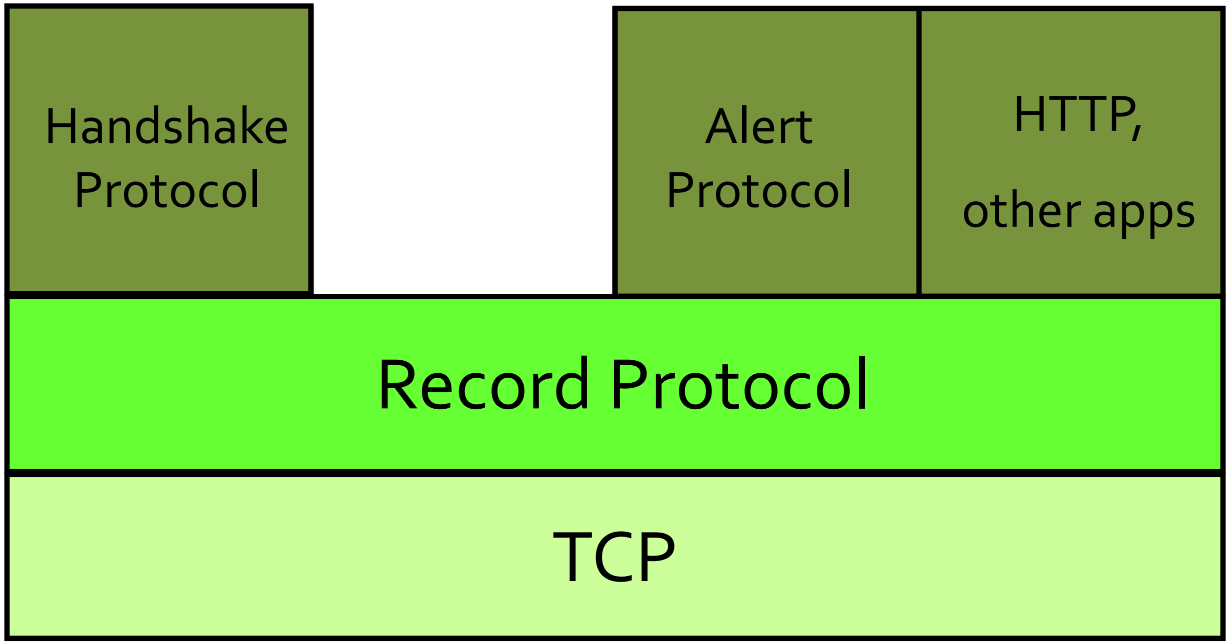
Used by client and server to:
1. Negotiate **cipher suite.**
2. Authenticate.
3. Establish keys used in the Record Protocol.

Provides confidentiality and integrity of application layer data using keys from Handshake Protocol

TLS Protocol Architecture (TLS 1.2)



TLS Protocol Architecture (TLS 1.3)



The TLS Ecosystem (1/3)

- Servers
  - Including managed service providers (Cloudflare, Akamai,…).
- Clients
  - Of all shapes and sizes.
  - Web browsers to embedded devices.
- Certification Authorities (CAs)
  - Of all shapes, sizes and levels of security.
  - Typically 300 root CA keys in browser.
- Implementations
  - From Google (BoringSSL), Facebook, AWS (s2n) down to small open-source operations.
  - OpenSSL somewhere in-between, once used by 80- 90% of web servers.
- Hardware vendors, e.g. F5.

The TLS Ecosystem (2/3)

- TLS versions:
  - SSL 3.0, TLS 1.0, TLS 1.1, TLS 1.2, TLS 1.3.
  - Some servers even still support SSL 2.0 (!)
- 337 cipher suites (see https://ciphersuite.info/cs/)
  - Some very common, e.g.
    TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256.
  - Some highly esoteric, e.g.
    TLS_KRB5_WITH_3DES_EDE_CBC_MD5.
  - 15 EXPORT, 27 ANON (2 with both).
  - And: TLS_NULL_WITH_NULL_NULL !
  - Reduced to just 5 cipher suites in TLS 1.3.
- Numerous TLS extensions
- DTLS: TLS over UDP

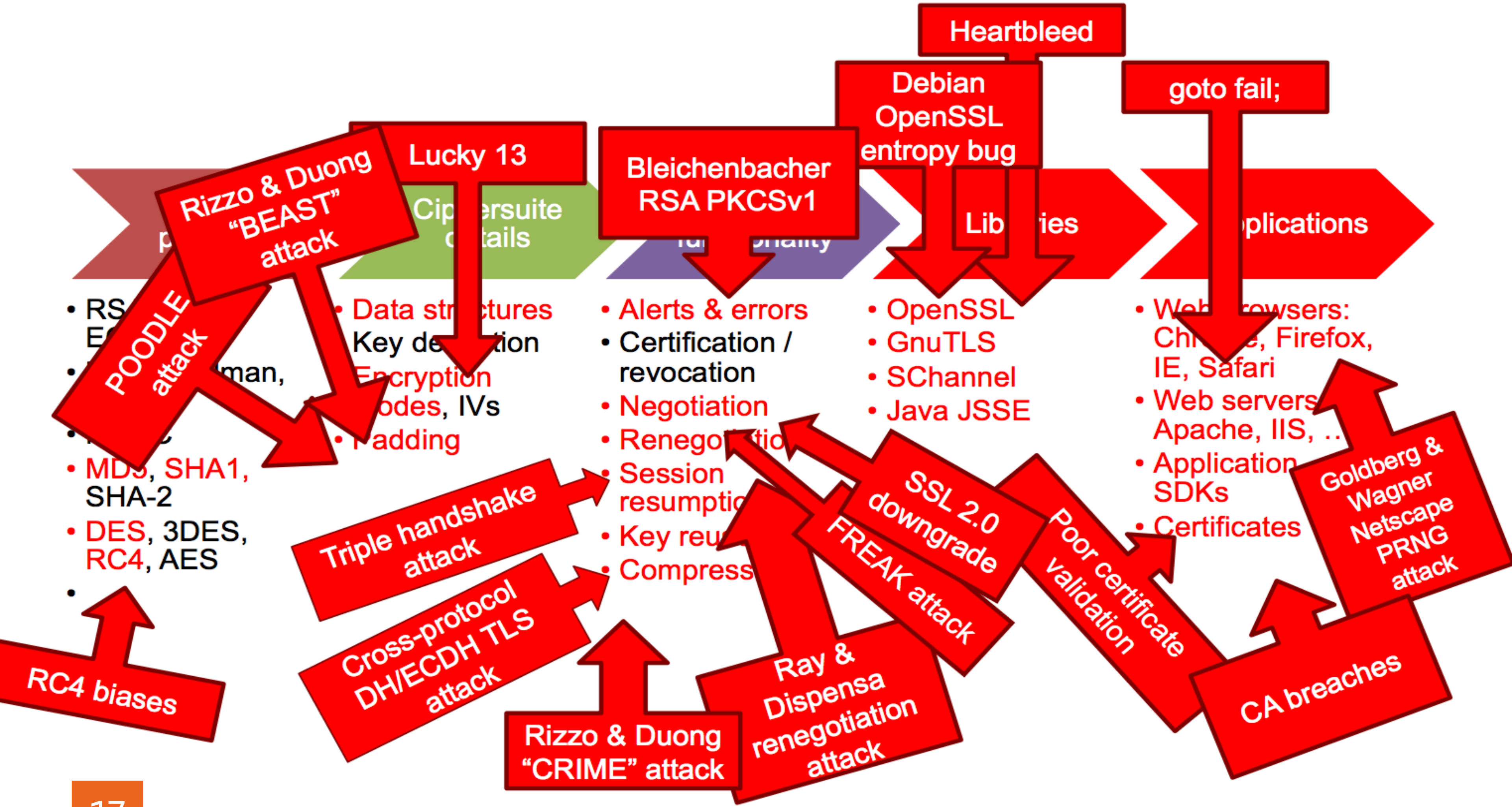
The TLS Ecosystem (3/3)

- IETF TLS Working Group
  - And CFRG (Crypto Forum Research Group)
- TLS research community
  - Finding attacks and building security proofs
  - Analysis of TLS 1.3 during its development.
- The TLS ecosystem has become very complex and vibrant.
  - With great industry-academia-IETF interaction during the development of TLS 1.3.

Motivation for TLS 1.3

Why TLS 1.3?

(slide from Douglas Stebila)



Why TLS 1.3?

Many attacks on the TLS protocol were discovered, mostly from 2012 onwards.

Reflection of:

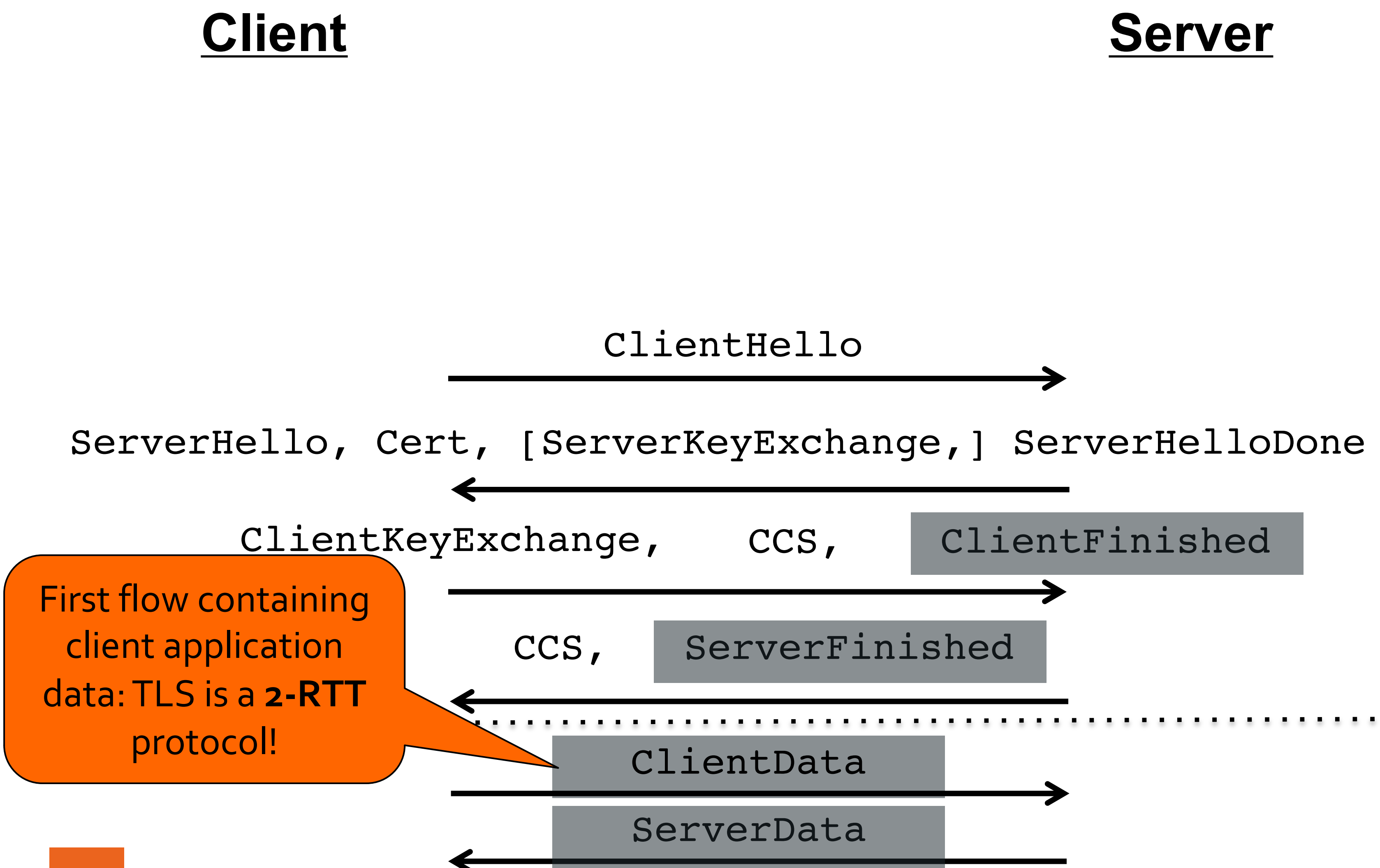
- Non-graceful ageing of protocol + poor quality of many implementations.
- Increasing importance of protocol.
- Increasing interest from research community.

Attacks broadly of two types: protocol-level and implementation-specific.

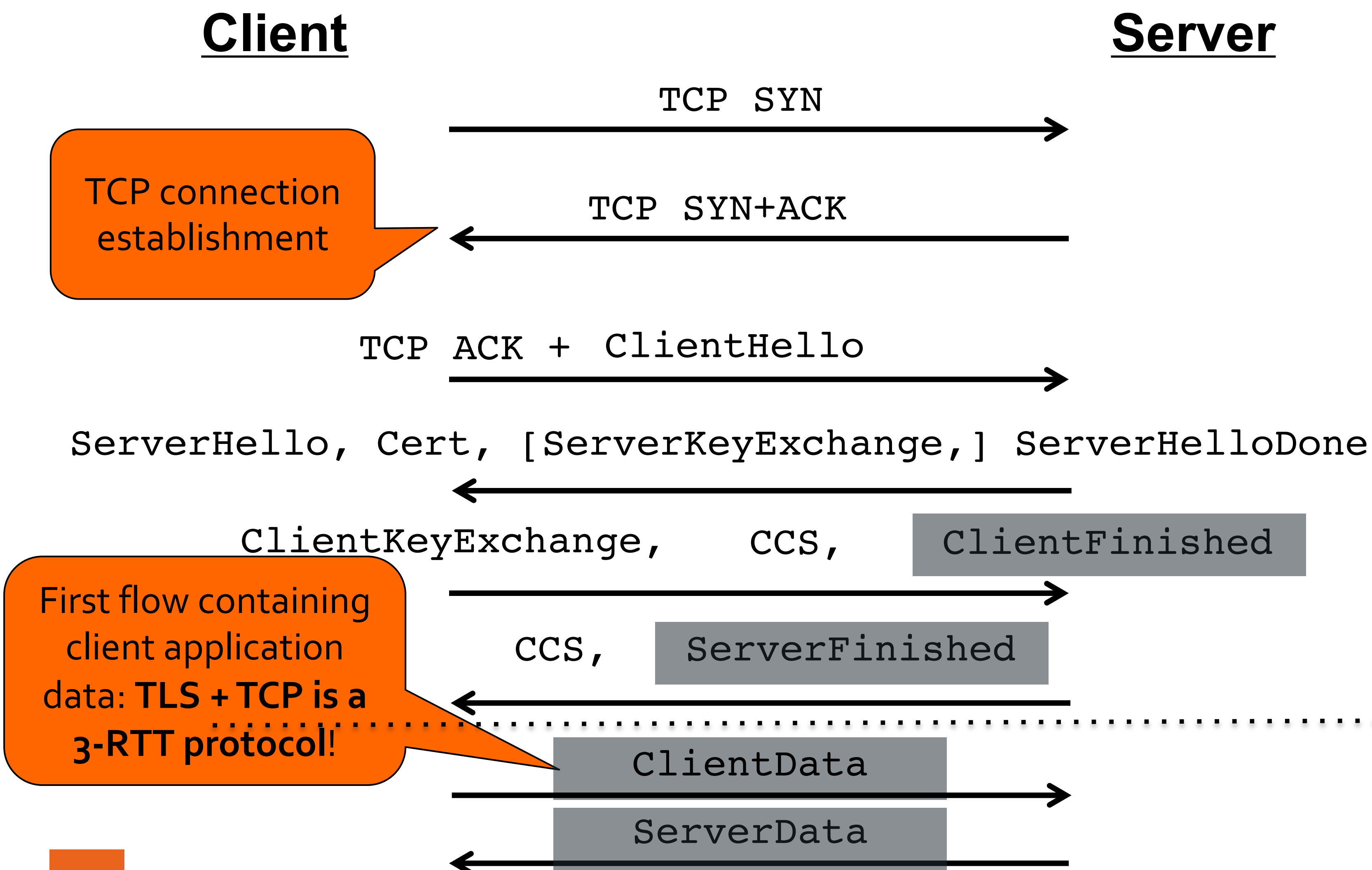
Why TLS 1.3?

- TLS came under pressure from newer protocol designs, especially Google QUIC.
- Key **physics** issue: the speed of light is finite (299,792,458 m/s *in vacuo*).
- Processing at end-points and intermediate routers also adds delay.
- Key **networking** issue: the Internet is a distributed system, with clients and servers that are geographically separated.
- Ameliorated to some extent through use of CDNs and caching, but round-trip-time (RTT) is typically 50-200 milliseconds.
- Key **protocol-level** issue: TLS requires multiple round trips before first (client) encrypted data can be sent.

The “Full” TLS Handshake Protocol, TLS 1.2



The “Full” TLS Handshake Protocol, TLS 1.2 + TCP



Why TLS 1.3?

- TLS 1.2 (full handshake, no resumption): TCP connection overhead (1 RTT: SYN, SYN+ACK, ACK+data) + TLS Handshake Protocol overhead (2 RTTs): total of 3 RTTs to first secured client data!
- Improvements via TLS resumption, TLS Snap Start,…
- QUIC runs over UDP and so avoids TCP connection establishment overhead.
- QUIC also offered a native 1-RTT Handshake, as well as a 0-RTT mode: often 1-RTT and potentially 0-RTT to first secured client data.
- TLS 1.3 essentially mimics QUIC in achieving same RTT profile (excluding TCP overhead which is unavoidable for TLS).
- TLS 1.3 Handshake later adopted in QUIC via IETF standardisation process.

The TLS 1.3 Design Process – Goals

- **Clean up:** get rid of flawed and unused crypto & features.
- **Improve latency:** for main handshake and repeated connections (while maintaining security).
- **Improve privacy:** hide as much of the handshake as possible.
- **Continuity:** maintain interoperability with previous versions and support existing important use cases.
- **Security Assurance (added later):** have supporting analyses for changes.

The TLS RFC – https://tools.ietf.org/html/rfc8446

PROPOSED STANDARD
Errata Exist

Internet Engineering Task Force (IETF)
Request for Comments: 8446
Obsoletes: 5077, 5246, 6961
Updates: 5705, 6066
Category: Standards Track
ISSN: 2070-1721

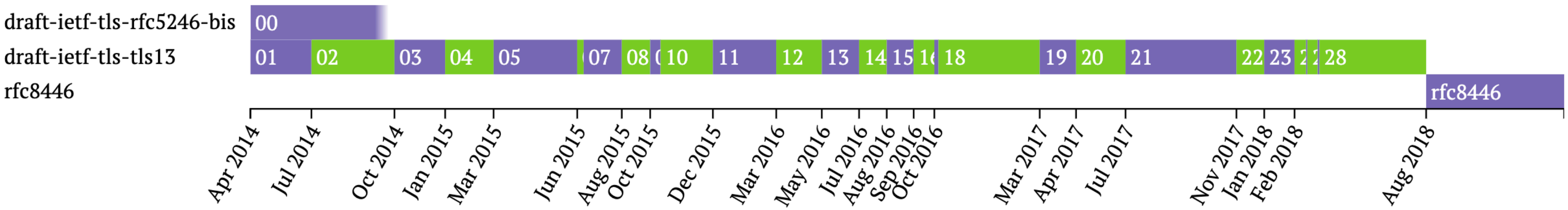
E. Rescorla
Mozilla
August 2018

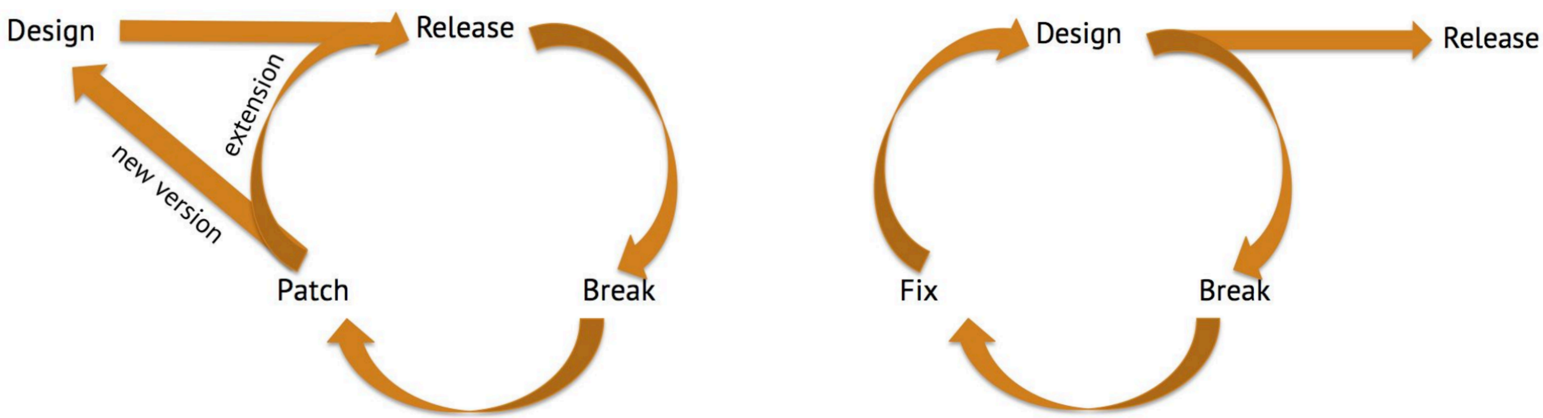
The Transport Layer Security (TLS) Protocol Version 1.3

Abstract

This document specifies version 1.3 of the Transport Layer Security (TLS) protocol. TLS allows client/server applications to communicate over the Internet in a way that is designed to prevent eavesdropping, tampering, and message forgery.

The TLS 1.3 Process



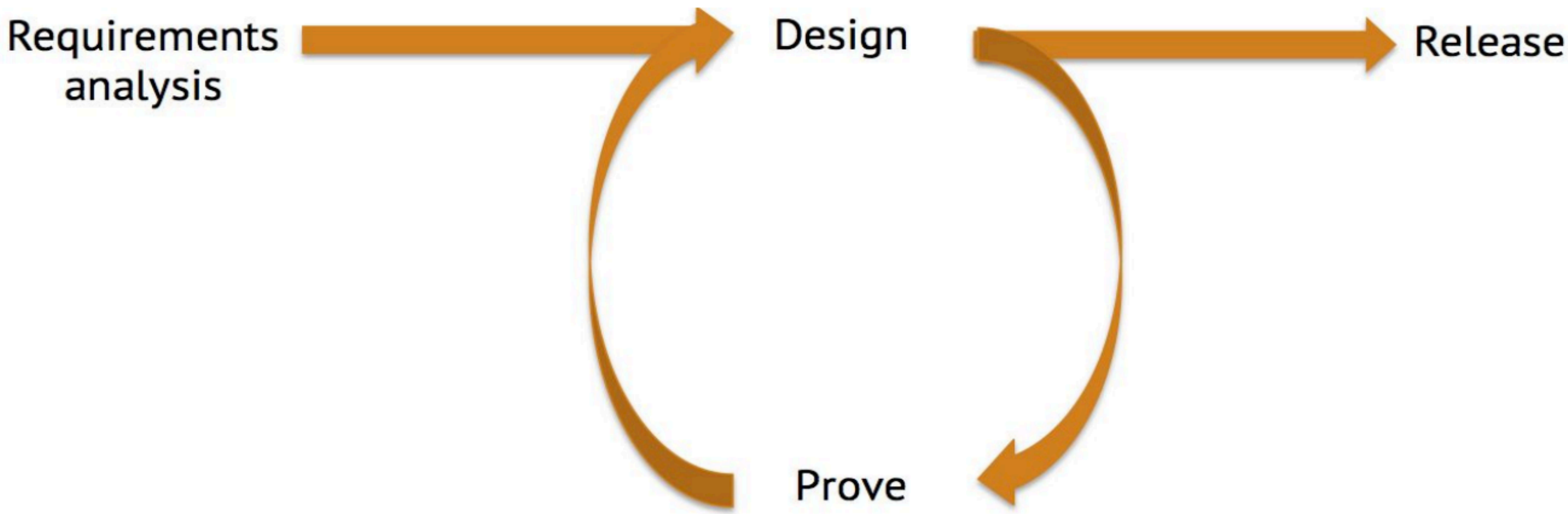


TLS 1.2 and before: Reactive

TLS 1.3: Proactive

K.G. Paterson and T. van der Merwe, Reactive and Proactive Standardisation of TLS, SSR 2016.

The TLS 1.3 Process – Room for Improvement?



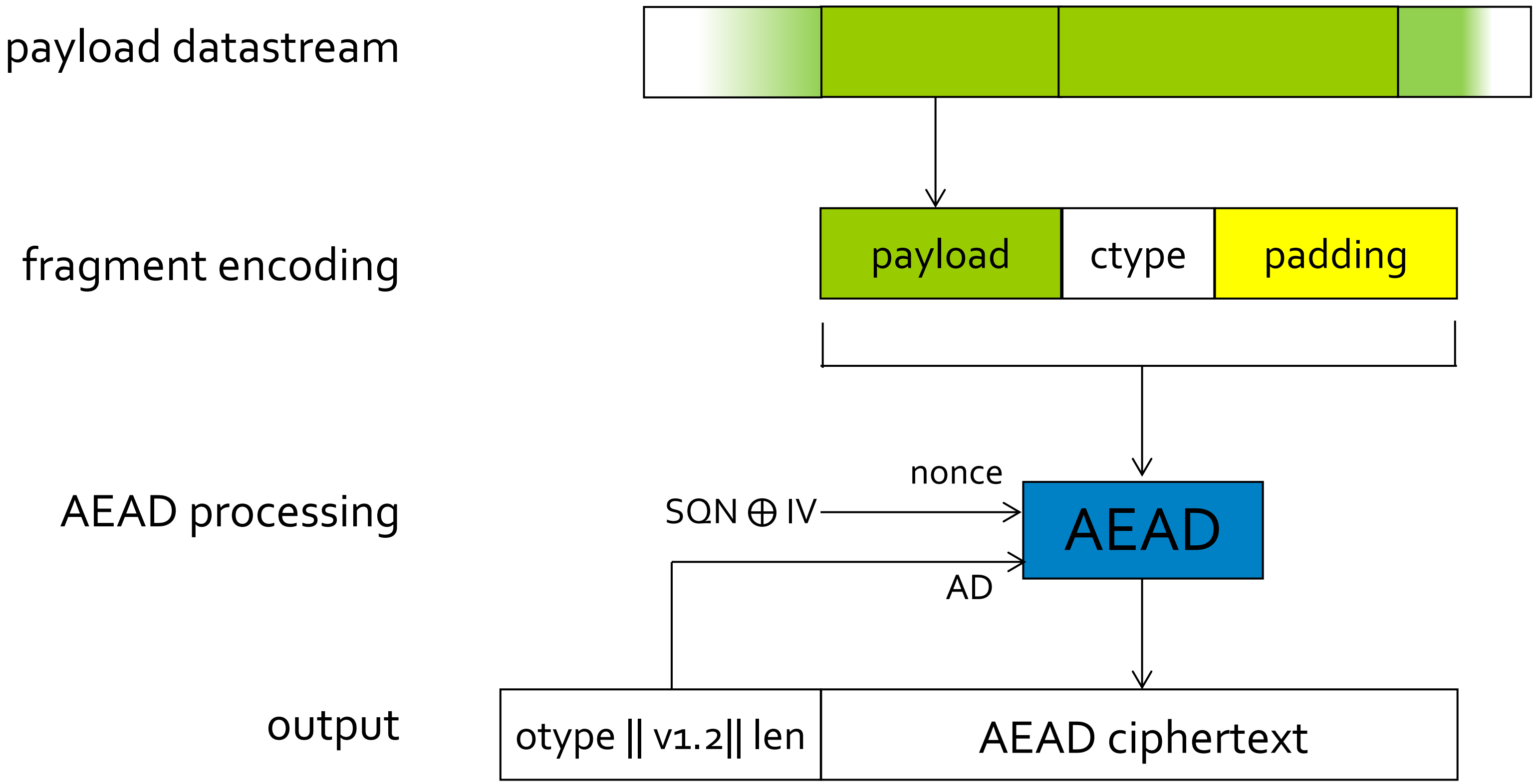
K.G. Paterson and T. van der Merwe, Reactive and Proactive Standardisation of TLS, SSR 2016.

TLS 1.3 Record Protocol

TLS 1.3 Record Protocol

- The TLS Record Protocol provides a **stream-oriented** API for applications making use of it.
  - Hence TLS may fragment into smaller units or coalesce into larger units any data supplied by the calling application.
  - Protocol data units in TLS are called **records**.
  - So each record is a fragment from a data stream.
- Cryptographic protections in the TLS Record Protocol:
  - Data origin authentication, integrity for records using a MAC.
  - Confidentiality for records using a symmetric encryption algorithm.
  - Prevention of replay, reordering, deletion of records using per record sequence number protected by the MAC.
  - Encryption and MAC provided simultaneously by use of AEAD in TLS 1.3.
  - Prevention of reflection attacks by key separation (different symmetric keys in different directions, but see Selfie attack).

TLS 1.3 Record Protocol: Record Processing



TLS 1.3 Record Protocol

- ctype field:
  - Single byte representing content type – indicates whether content is handshake message, alert message or application data.
  - AEAD-encrypted inside record; header contains dummy value to limit traffic analysis.
- Padding:
  - Optional feature that can be used to hide true lengths of fragments.
  - Not needed for encryption (cf. earlier versions of TLS using CBC mode).
  - Sequence of 0x00 bytes after non-0x00 content type field.
  - Removed after integrity check, so no padding oracle issues arise.

TLS 1.3 Record Protocol

- AEAD nonce:
  - Constructed from 64-bit sequence number (SQN).
  - SQN is incremented for each record sent on a connection.
  - SQN is masked by XOR with IV field.
  - IV is a fixed (per TLS connection) pseudorandom value derived from secrets in TLS Handshake Protocol.
  - IV masking ensures nonce sequence is "unique" per connection, good for analysing security in multi-connection setting.
- Record header:
  - Contains dummy type field ("application data", 1 byte), legacy version field (2 bytes), length of AEAD ciphertext (2 bytes).
  - Version field is anyway securely negotiated during handshake.
  - SQN is not included in header, but is maintained as a counter at each end of the connection (send and receive).

TLS 1.3 Record Protocol

- AEAD options: AES_128_GCM; AES_256_GCM; ChaCha20Poly1305; AES_128_CCM; AES_256_CCM.
- Additional feature: **rekeying** of TLS connection based on theoretical data limits for algorithm selected, e.g. rekey every $2^{24.5}$ records for AES_128/256_GCM; also improves forward security of Record Protocol.
- Any AEAD-decryption failures are **fatal**: connection is torn down, key material thrown away.
  - How does this help to prevent attacks on the TLS Record Protocol?
- Attacks not prevented by TLS 1.3 Record Protocol:
  - Truncation attacks on the stream of records.
  - Application-layer confusion: record boundaries ≠ APDU boundaries.
  - Timing attacks on the padding scheme (recognised in RFC).

TLS 1.3 Record Protocol vs Earlier Versions

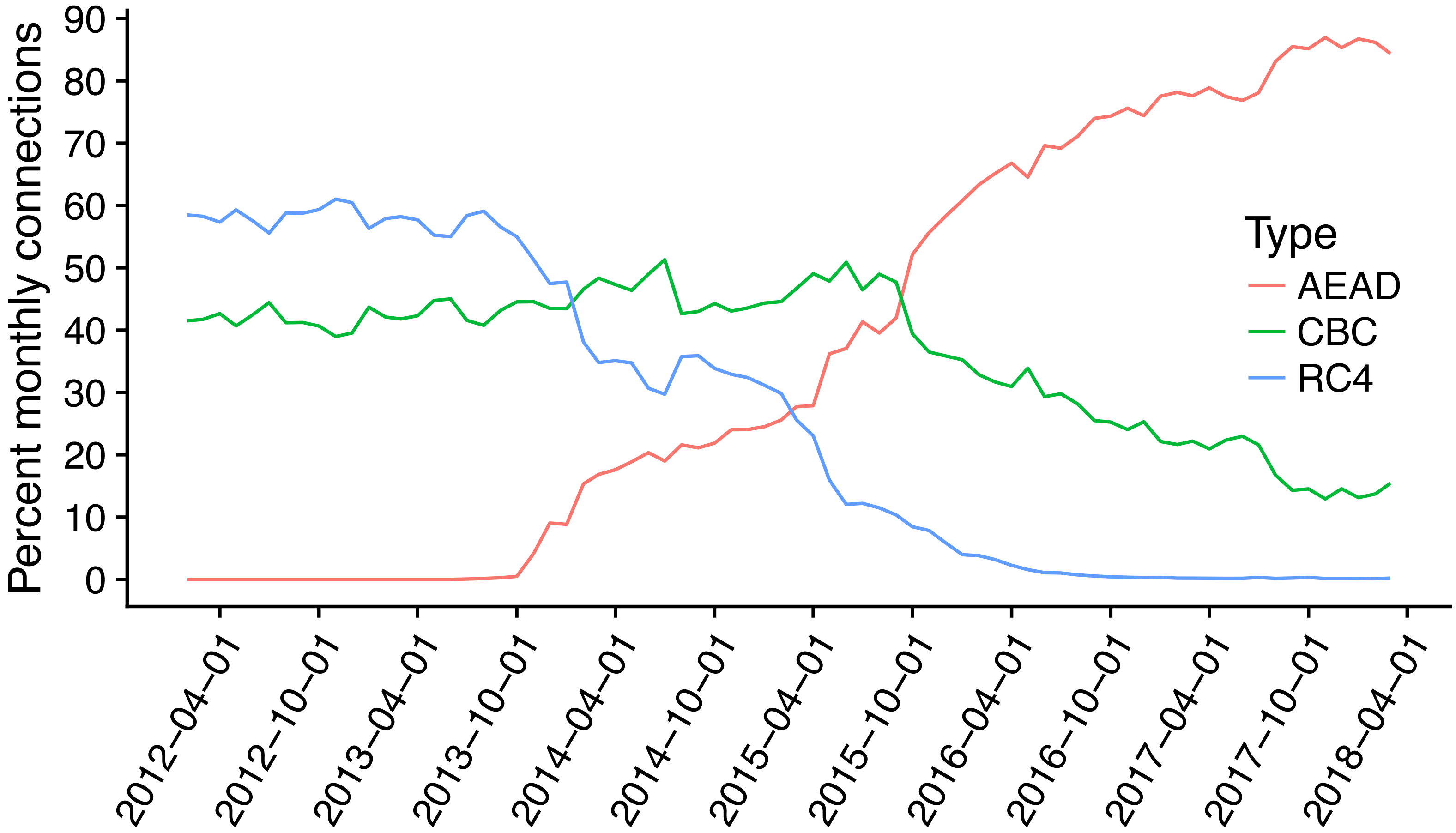
- AEAD-only in TLS 1.3, while earlier versions of TLS supported "MAC-encode-encrypt" (MEE) as well as AEAD (TLS 1.2 only).
- MEE with RC4 encryption was **weak** because of statistical weaknesses in RC4.
- MEE in CBC-mode with DES/triple-DES was **vulnerable** to Sweet32 attack, based on small (64-bit) block-size.
- MEE in CBC-mode had particular issues with **padding oracle attacks**, e.g. Lucky 13 attack.
- MEE in CBC-mode with predictable IVs (TLS 1.0 and earlier) was vulnerable to **BEAST attack**.
- MEE in CBC-mode with relaxed padding (SSLv3, bad TLS implementations) was vulnerable to **POODLE attack**.
- TLS 1.2 and earlier had optional compression feature, enabling **CRIME** attack; TLS 1.3 removes compression (but CRIME-like attacks are still possible at the application layer).

Historical Note: AEAD and TLS 1.2 Record Protocol

AEAD was already added to TLS in TLS 1.2.

- Escapes from the MEE template that was only option up to TLS 1.1.
- AES-GCM specified in RFC 5288; AES-CCM specified in RFC 6655.
- But was not supported by any mainstream browsers or by OpenSSL (dominant on server-side) until 2013.
- Now widely supported in TLS 1.2 implementations and used in TLS, obligatory in TLS 1.3.
- Uptake in TS 1.2 was driven by the aforementioned attacks.

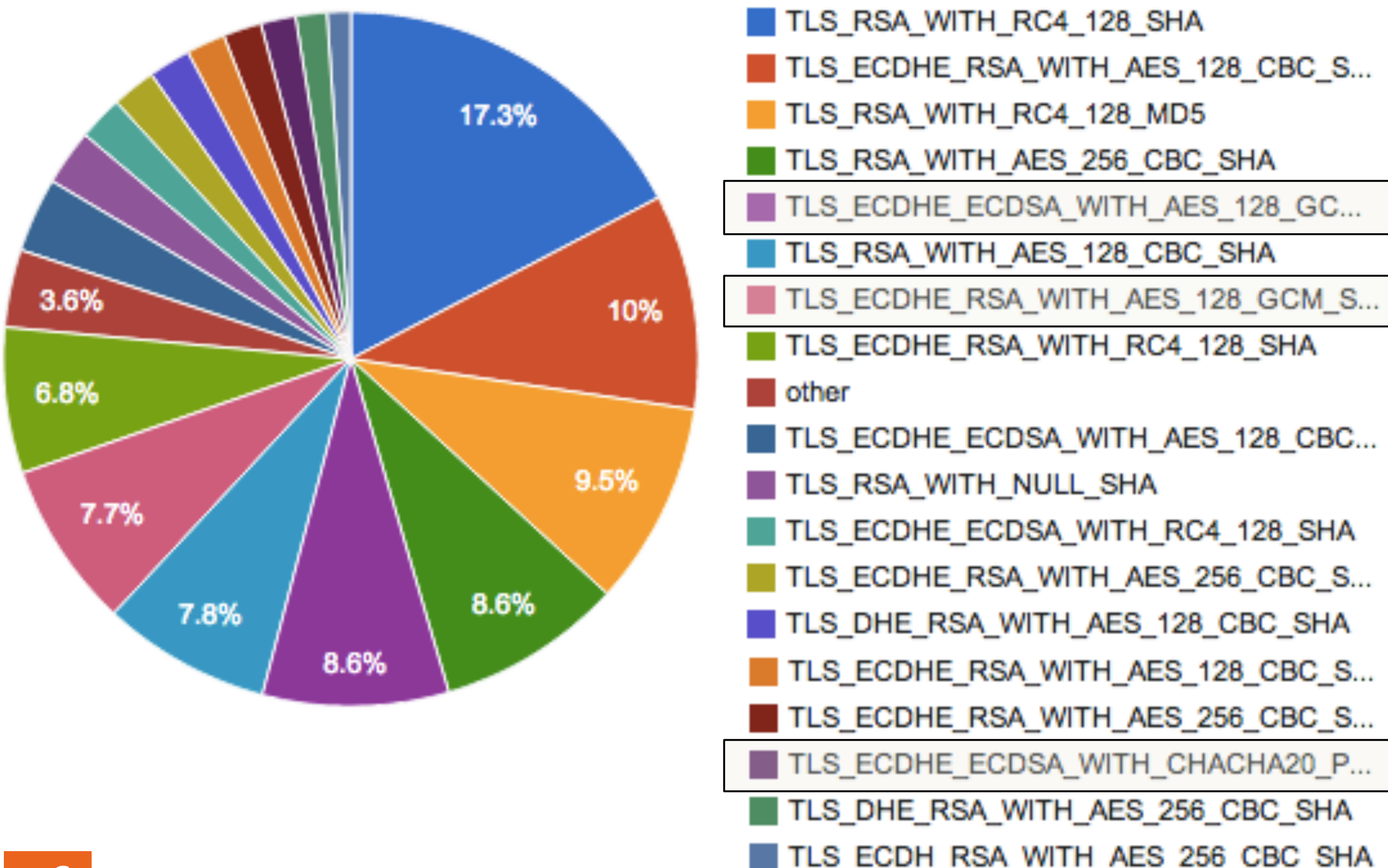
TLS Record Protocol Algorithms in Use, 2012-2018



Source: Kotzias *et al.*, *Coming of Age: A Longitudinal Study of TLS deployment*. IMC 2018.

AEAD Usage in TLS: September 2014

Snapshot from ICSI Certificate Notary Project

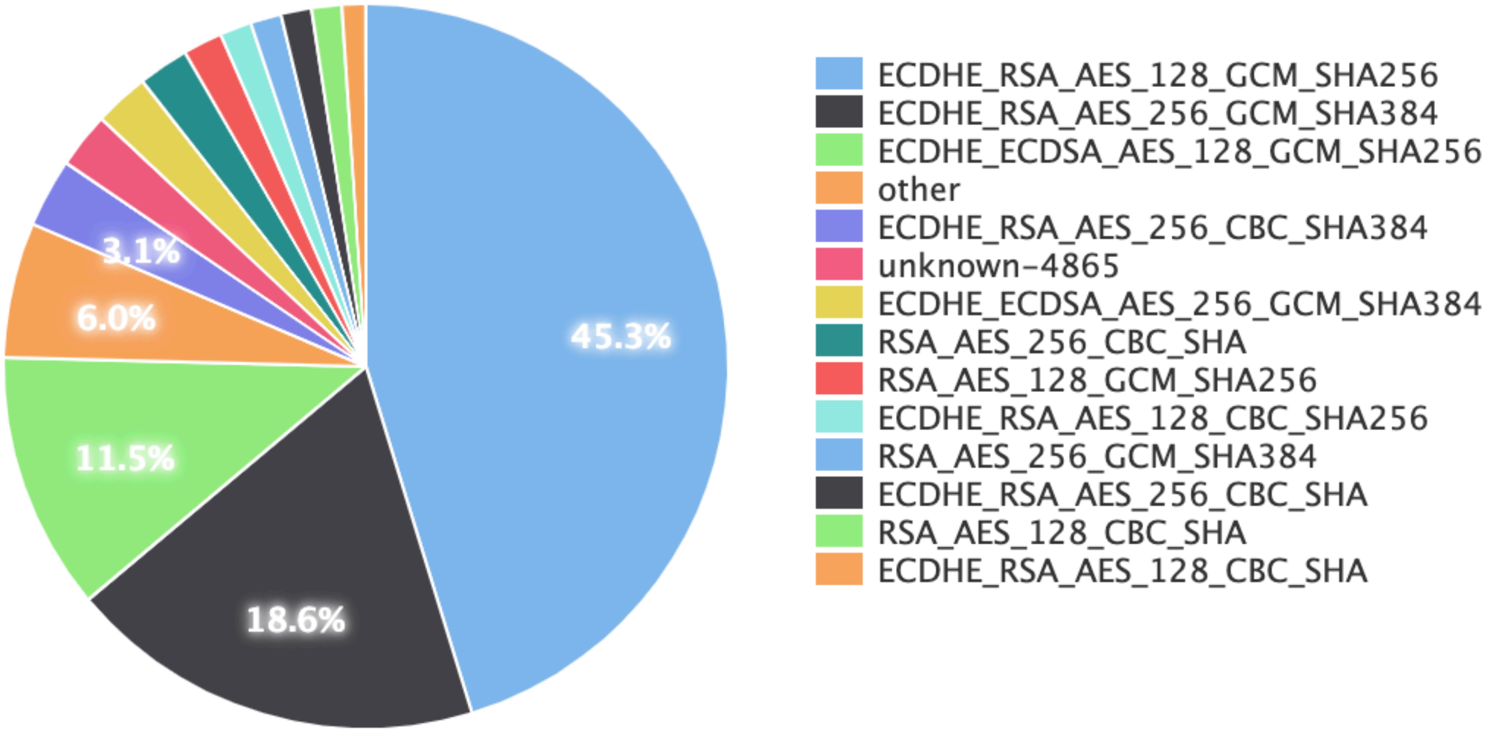


16.3%

1.6%

AEAD Usage in TLS: September 2019

Snapshot from ICSI Certificate Notary Project



TLS 1.3 Handshake Protocol

TLS 1.3 Handshake

- TLS 1.2 and earlier are slow: 2 RTTs before client can securely send data (3 if we include TCP connection establishment).
- TLS 1.3: **full handshake in 1 RTT.**
  - Achieved via feature reduction: we always do (EC)DHE in one of a shortlist of groups.
  - Client speculatively sends several DH shares in supported groups.
  - Server picks one, replies with its share, and can already derive Record Protocol keys.
- **0-RTT handshake** when resuming a previously established connection.
  - Client+server keep shared state enabling them to derive a PSK (pre-shared key).
  - Client derives an "early data" encryption key from the PSK and can use it to include encrypted application data along with its first handshake message.
  - But: 0-RTT **sacrifices** certain security properties (more later).

TLS 1.3 Handshake

Improving privacy

- TLS 1.2 and earlier: complete handshake in the clear (incl. certificates, extensions).
- TLS 1.3: **encrypts almost all handshake messages.**
- TLS 1.3 derives separate key to protect handshake messages.
- This provides security against passive/active attackers (for server/client).

Continuity

- e.g. remove complex renegotiation protocol, but keep some features (key update + client authentication option).
- Interoperability/ease of deployment: make TLS 1.3 ClientHello look like TLS 1.2, so middleboxes do not block the protocol.

Basis of TLS 1.3 Handshake: Signed Diffie-Hellman

Client | **Server**

`ClientNonce,` g^x $\longrightarrow$

`K = KDF(`g^{xy}`, context)`

g^y`, ServerSignature(ClientNonce,` g^x`,` g^y`), AppData` $\longleftarrow$

1-RTT key agreement

Server can already send encrypted data here.

`K = KDF(`g^{xy}`, context)`

`AppData` $\longrightarrow$

- Pre-supposes client and server know which group they will use for DH.
- Server signature on ClientNonce (random value) authenticates server to client.
- Ignores PKI/certification aspects.
- Does not give explicit key confirmation: client/server know the same key.

TLS 1.3 Handshake – 1-RTT (simplified)

Client		Server
ClientHello(ClientNonce) ClientKeyShare	→	
	←	ServerHello(ServerNonce) ServerKeyShare
	←	EncryptedExtensions* CertificateRequest* ServerCertificate ServerCertificateVerify ServerFinished
ClientCerificate* ClientCertificateVerify* ClientFinished	→	ApplicationData
ApplicationData	↔	ApplicationData

Client guesses which DH group(s) server will support.

Server sends its DH value in one of groups indicated by client

TLS 1.3 Handshake – 1-RTT

- Client includes DH share(s) in its first message, along with `ClientHello`, anticipating group(s) that server will accept.
- Server responds with single DH share in its `ServerKeyShare` response.
- If this works, a forward-secure key is established after 1 round trip (1-RTT).
- If server does not like DH group(s) offered by client, it sends a `HelloRetryRequest` and a group description back to client.
- In this case, the handshake will be 2-RTT.

TLS 1.3 Handshake – DH and ECDH groups

- Limited set of DH and ECDH groups are supported in TLS 1.3.
- Reduces likelihood of fall-back to 2-RTT.
- Removes problem of client not being able to validate DH parameters that was inherent in TLS 1.2 and earlier.
  - In TLS 1.2 and earlier, server sent (p, g, g^x) for finite-field DH, but no information about the order of g; no guarantee that order is prime, hard to check that g^x is in right subgroup.
- Removes complexity from implementations.
- We refer to DHE and ECDHE cipher suites; no other options in TLS 1.3.
- E = Ephemeral.

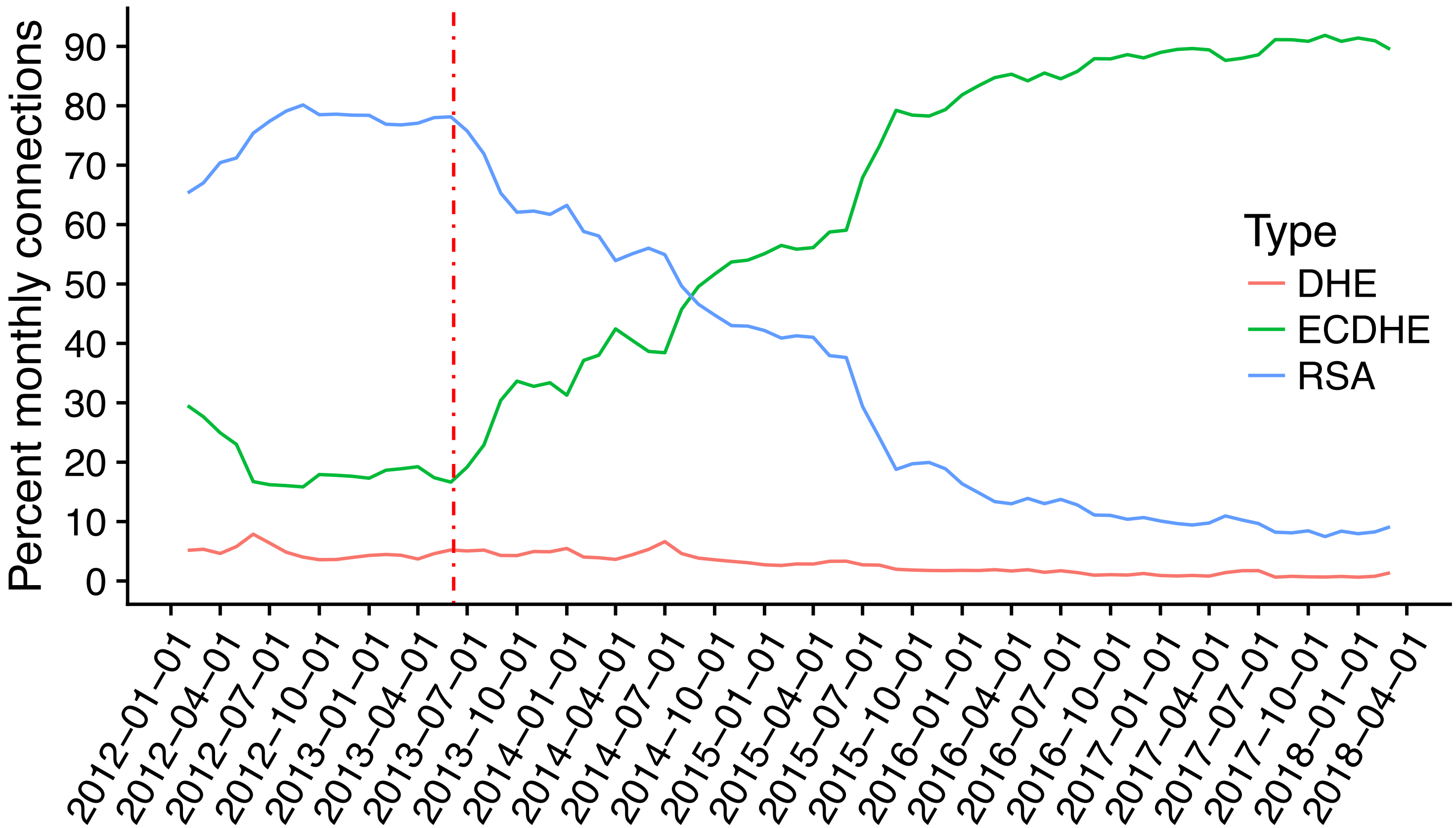
TLS 1.3 Handshake – DH and ECDH groups

- DH groups:
  - Specified in RFC 7919
  - |p| = 2048, 3072, 4096, 6144, 8192.
  - All p are such that q = (p-1)/2 is prime.
  - Removes several avenues of attack: backdoored primes, small subgroup attacks, etc.
- ECDH groups:
  - Some existing curves from RFC 4492 and 2 new curves in RFC 7748.
  - NIST P256, P384, P521; Curve25519, Curve448.

TLS 1.3 Handshake – Forward Security

- Because of reliance on Ephemeral DH key exchange, TLS 1.3 Handshake (in this 1-RTT mode) is **forward secure**.
- This (informally) means: compromise of all session keys, DH values and signing keys has no impact on the security of earlier sessions.
- This also means: if NSA subpoenas a server's long-term (signing) keys, then they still can carry out active attacks on future sessions involving that server, but they cannot passively decrypt them, and they cannot decrypt earlier sessions.
- Compare to RSA key transport option in TLS 1.2 and earlier: past and future passive interception using RSA private key.

SSL/TLS: Key Exchange Methods in Use, 2012-2018



Source: Kotzias *et al.*, *Coming of Age: A Longitudinal Study of TLS deployment*. IMC 2018.

TLS 1.3 Handshake – 1-RTT: Server Authentication

Client | **Server**

```
ClientHello(ClientNonce)
ClientKeyShare                    →
```

```
                                  ServerHello(ServerNonce)
                                  ServerKeyShare
                                  EncryptedExtensions*
                                  CertificateRequest*
                                  ServerCertificate
                          ←       ServerCertificateVerify
                                  ServerFinished
                                  ApplicationData
```

This pair contains server's public key + signature on Handshake transcript

```
ClientCerificate*
ClientCertificateVeri
ClientFinished
```

Computed as HMAC on Handshake transcript; for key conf and server auth in PSK modes.

```
ApplicationData                   ApplicationData
```

TLS 1.3 Handshake – 1-RTT: Client Authentication

Client | **Server**

`ClientHello(ClientNonce)`
`ClientKeyShare`

`ServerHello(ServerNonce)`
`ServerKeyShare`
`EncryptedExtensions*`
`CertificateRequest*`
`ServerCertificate`
`erCertificateVerify`
`ServerFinished`
`ApplicationData`

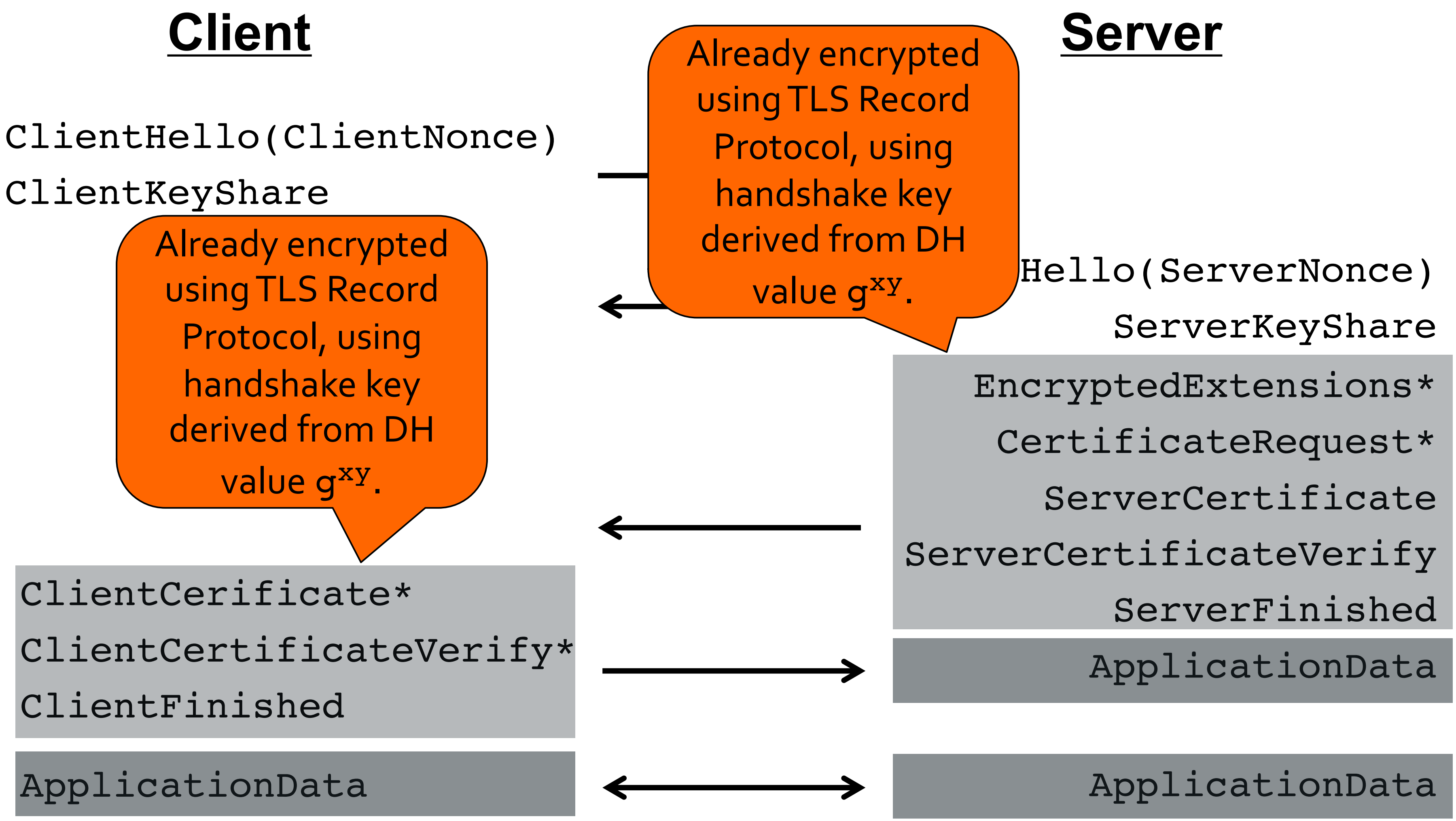
This pair contains client's public key + signature on Handshake transcript, if requested

`ClientCerificate*`
`ClientCertificateVerify*`
`ClientFinished`

Computed as HMAC on Handshake transcript; for key conf and client auth in PSK modes.

`ApplicationData` | `ApplicationData`

TLS 1.3 Handshake – 1-RTT: Handshake Encryption



TLS 1.3 Handshake – Cipher Suite and Version Negotiation

- Cipher suites in TLS 1.3 are of the form: `TLS_AEAD_HASH`
- `AEAD:` AEAD scheme used in Record Protocol.
- `HASH:` Hash algorithm used in HKDF/HMAC for key derivation and computation of `Finished` messages.
- There are 5 cipher suites (currently) for TLS 1.3:

  TLS_AES_128_GCM_SHA256

  TLS_AES_256_GCM_SHA384

  TLS_CHACHA20_POLY1305_SHA256

  TLS_AES_128_CCM_SHA256

  TLS_AES_128_CCM_8_SHA256

TLS 1.3 Handshake – Cipher Suite and Version Negotiation

- Client proposes list of cipher suites in ClientHello message.
- Each cipher suite is encoded as a 2-byte value.
- Server selects one and returns corresponding 2-byte value in ServerHello.
- Values selected are incorporated into signatures and Finished messages.
- Similarly, list of (EC)DHE groups proposed and accepted are included into signatures and Finished messages.
- Assuming those messages themselves cannot be cryptographically tampered with, then client and server get assurance that both sides have same view of what was proposed and what was accepted.
- Similar mechanism to protect TLS version negotiation (but a bit more complicated because of issues in earlier protocol versions).

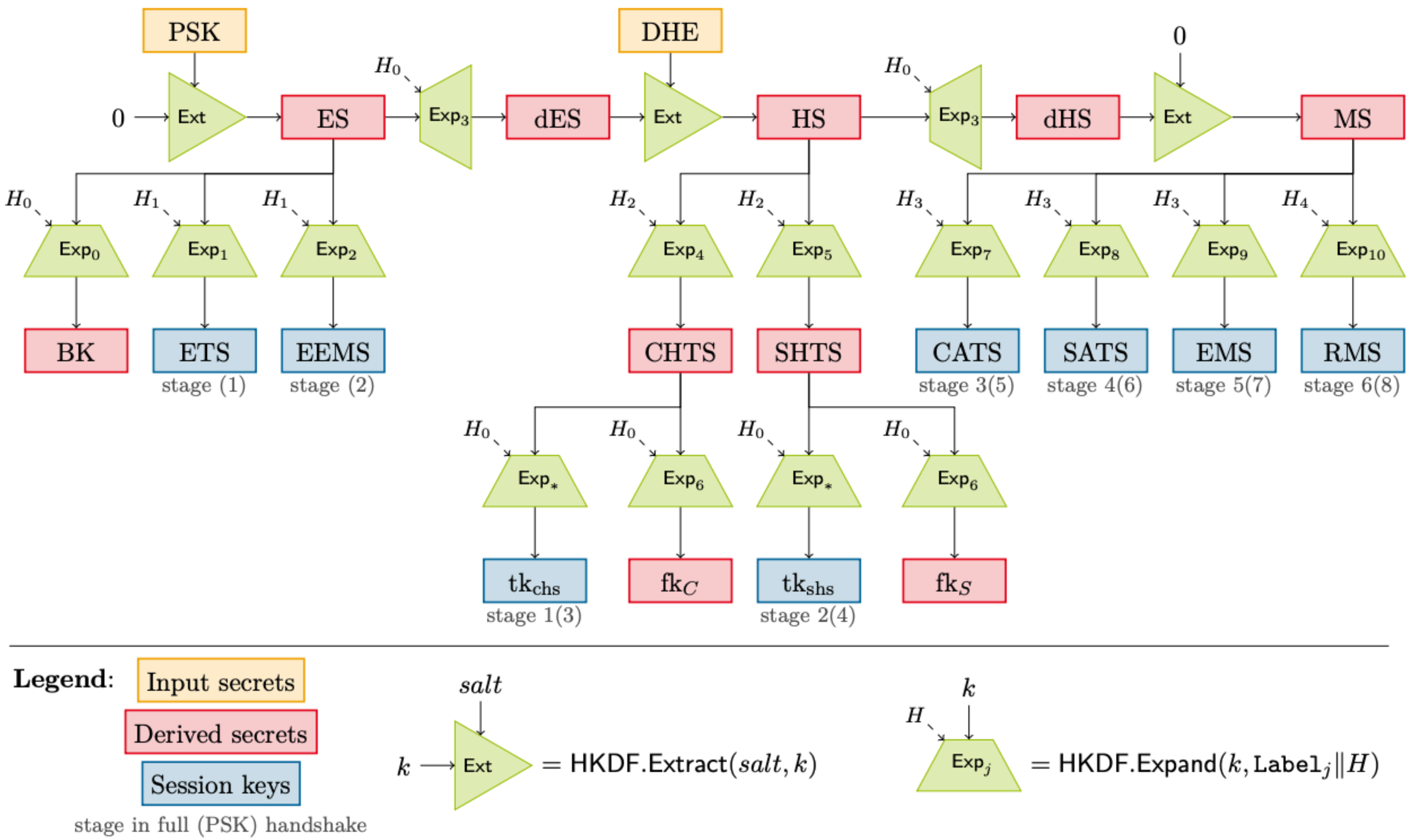
TLS 1.3 Handshake – Keys and Key Derivation

- Prior versions of TLS had a simple key schedule:

  `Pre_Master_Secret` → `Master_Secret` → `session keys`

- The key derivation process was also quite simple.
- This led to some problems, e.g. triple Handshake attack.
- TLS 1.3 adopts a more complex approach, attempting to provide much better key separation and stronger binding of keys to cryptographic context.
- TLS 1.3 relies heavily on HKDF, a hash-based key derivation function (RFC 5869).
- TLS 1.3 keys include:
  - `early_traffic_secret`
  - `client_handshake_traffic_secret; server_handshake_traffic_secret`
  - `Client_finished_key; server_finished_key`
  - `client_application_traffic_secret; server_application_traffic_secret`
  - `early_exporter_master_secret; exporter_master_secret`
  - `resumption_master_secret`

TLS 1.3 Handshake – Keys and Key Derivation



Taken from: Dowling, Fischlin, Günther, Stebila, https://eprint.iacr.org/2020/1044.pdf

TLS Handshake Protocol – Reliance on Randomness

- An attacker who can predict a client's choice of client/server DH private value can passively eavesdrop on all sessions!
  - And nonces in `Hello` messages may already leak information about state of client or server PRNG.
  - Hence backdoored PRNGs present a serious risk to TLS security: they may allow recovery of future PRNG output from observed output(s).
  - See Checkoway et al. (USENIX Security 2014) for extended analysis of exploitability of Dual EC-PRNG in the context of TLS 1.2 and earlier.

TLS Handshake Protocol – Reliance on Randomness

- Relatedly, some server implementations default to using a “repeated ephemeral” value for performance reasons.
- cf. CVE-2016-0701:

```
OpenSSL provides the option SSL_OP_SINGLE_DH_USE
for ephemeral DH (DHE) in TLS. It is not on by
default.
```

- Hence one-time server compromise would undermine the security of many client sessions.
- Such ephemeral reuse also makes certain side-channel attacks easier, e.g. see recent Raccoon attack (https://raccoon-attack.com/).

Gratuitous Raccoon Picture

TLS 1.3 Resumption and 0-RTT Feature

TLS 1.3 Handshake – Resumption and PSKs

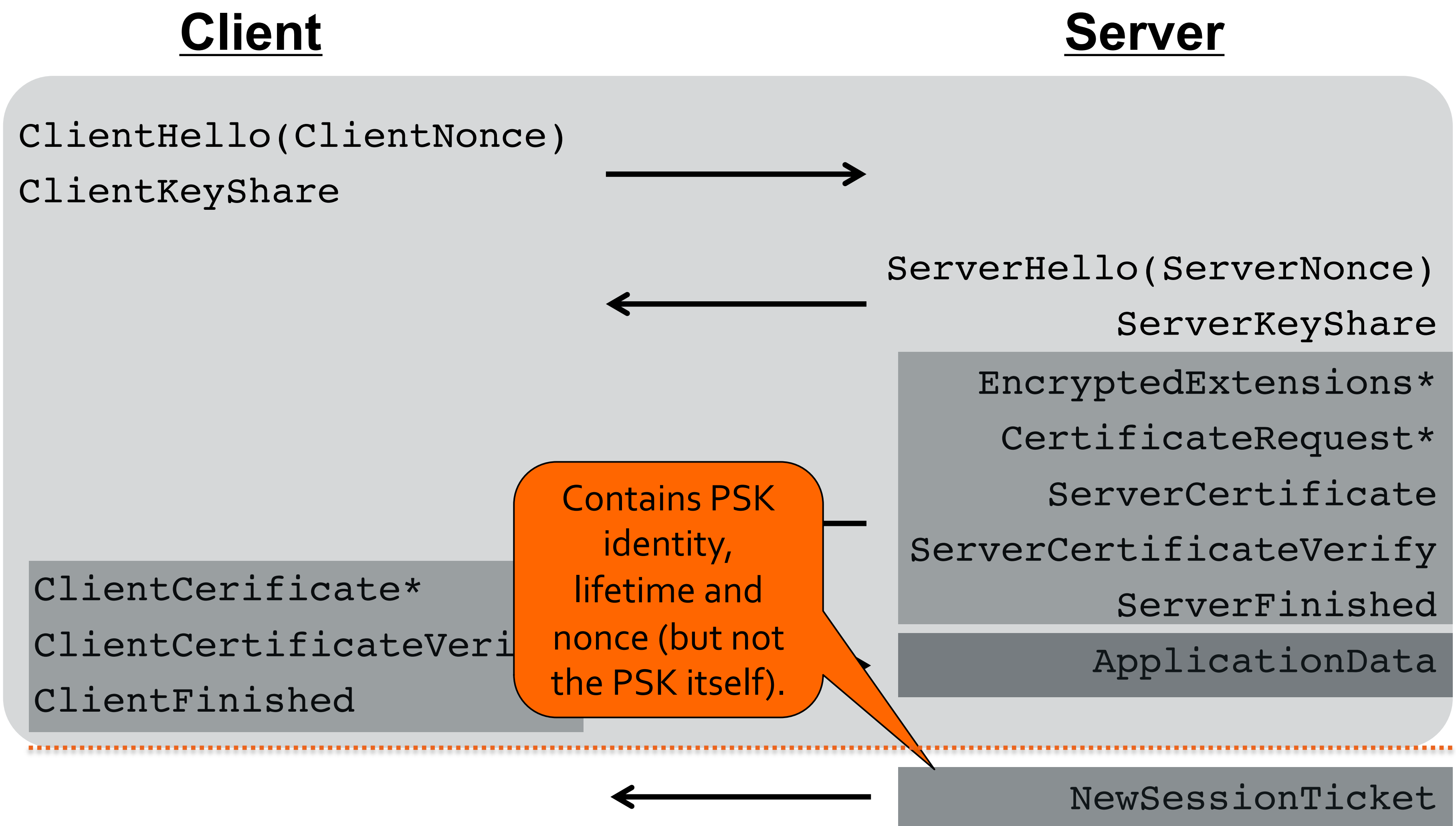
Prior versions of TLS had a session resumption feature.

- Lightweight handshake protocol, exchange of nonces and new key derivation based on existing `mastersecret`.
- Achieves 1-RTT, no public-key crypto.
- Reduces latency and server load, since clients return frequently to same servers.
- Not forward secure (since all new keys derived from existing secrets).
- Excellent for making web pages load faster – perform multiple session resumptions in parallel, each in its own TCP+TLS connection.

TLS 1.3 Handshake – Resumption and PSKs

- This feature is replicated in TLS 1.3, using the Resumption Handshake.
- Unified with PSK (Pre-Shared Key) mode for TLS 1.3.
- Client and server are assumed to have already established PSKs using `NewSessionTicket` handshake messages (or via out-of-band method in pure PSK mode).
- These are sent under the protection of existing Record Protocol.
- Each PSK has an identity – a unique string identifying it at client and server.
- `NewSessionTicket` handshake message allows server to deliver a new PSK identity (and other info about the new PSK, including its lifetime and a PSK nonce) to the client.
- Actual PSK values are derived from the current session's `resumption_master_secret` along with PSK nonce.

TLS 1.3 Handshake – NewSessionTicket

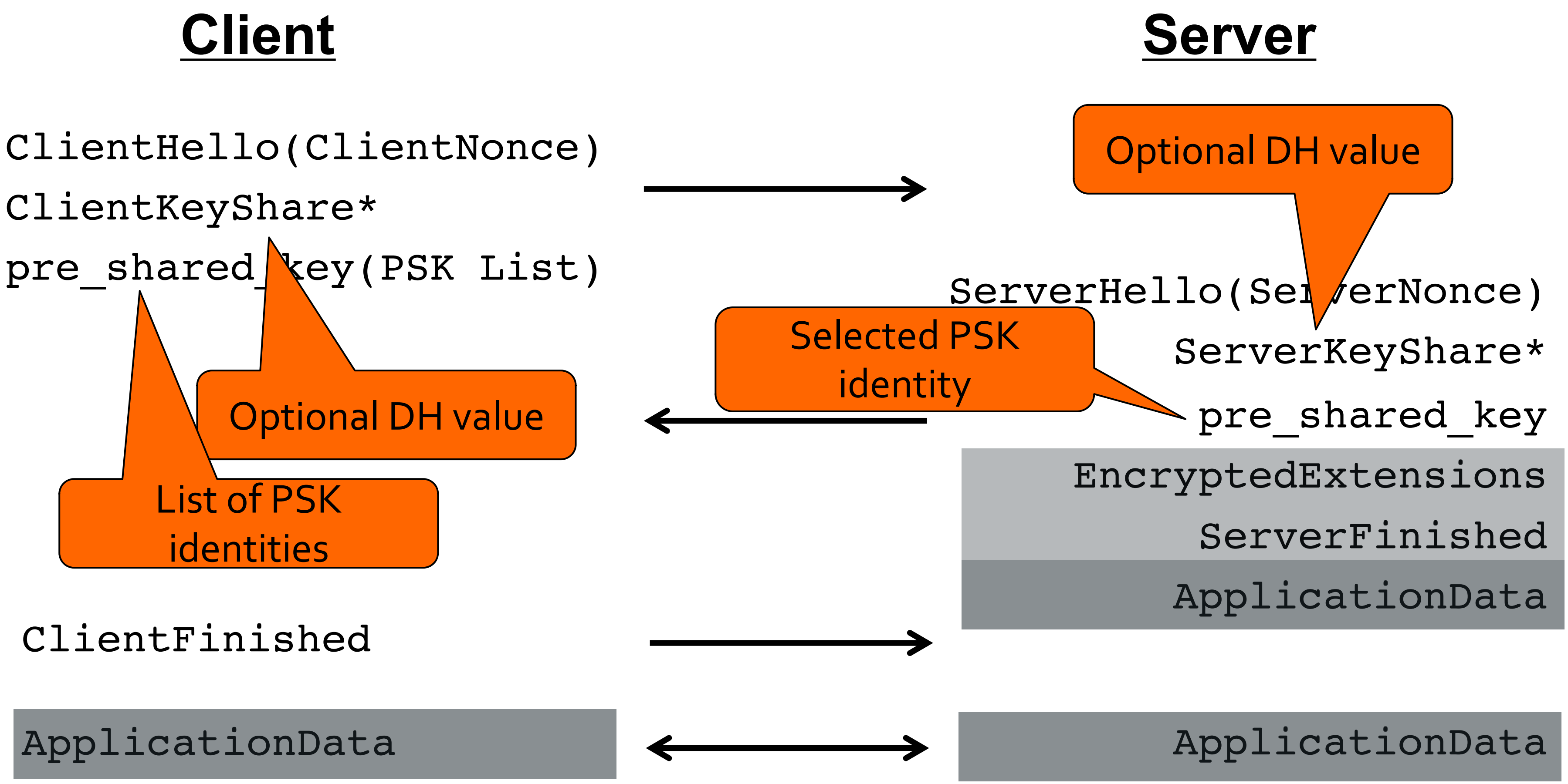


TLS 1.3 Handshake – Resumption and PSKs

The Resumption Handshake:

- Client sends list of PSK identities in a TLS extension in its first flow, plus optional (EC)DHE value.
- Server selects and sends single PSK identity, plus optional (EC)DHE value.
- Server also uses the PSK identity to look-up the session's `resumption_master_secret` in a server-side database and then uses it to compute the actual PSK.
- No server signature; authentication of both parties now based on PSK and `Finished` messages.
- If EC(DHE) values are sent during resumption, then the new session has forward security with respect to the PSK.
- That is, later compromise of the PSK key (or the relevant `resumption_master_secret`) does not affect security of the newly established session.

TLS 1.3 Handshake – Resumption Handshake (Simplified)



TLS 1.3 Handshake – Resumption and PSKs

Server-side PSK management:

- A server may issues thousands or even millions of concurrent session tickets to clients, since typical ticket/PSK lifetime is 1 week.
- Storing a database containing all the corresponding PSK info and `resumption_master_secret` values would be onerous for such a server.
- In typical deployments, the server sets the PSK identity to be an encryption of everything that is needed to later reconstruct the PSK.
- This encryption is done using a server-side "session ticket master key".
- This PSK identity is sent to the client in an earlier `NewSessionTicket` handshake message.
- The PSK identity is returned to the server in the `pre_shared_key` resumption handshake message.
- In this way, the server actually outsources storage of the database to the clients.
- None of this is specified in the TLS 1.3 RFC, but similar procedures were used earlier.
- Security consequences?

TLS 1.3 Handshake – 0-RTT

Under pressure from Google's QUIC protocol, the TLS WG in IETF also decided to add a **0-RTT** option to TLS 1.3.

- Enables client to send secure application data in its first flow in a Resumption Handshake.
- Uses `early_traffic_secret` key that is derived from the PSK whose identity was quoted in Resumption Handshake.
- The 0-RTT data does not enjoy forward security, since its protection is based on PSKs (and server-side session ticket master key if used).
- 0-RTT data cannot be replayed within a connection, and cannot be confused with 1-RTT data (by key separation).
- However, 0-RTT data can be vulnerable to replay attacks across connections, especially in distributed server environments.
- See RFC 8446 Section 8 and Appendix E.5 for extensive discussion.

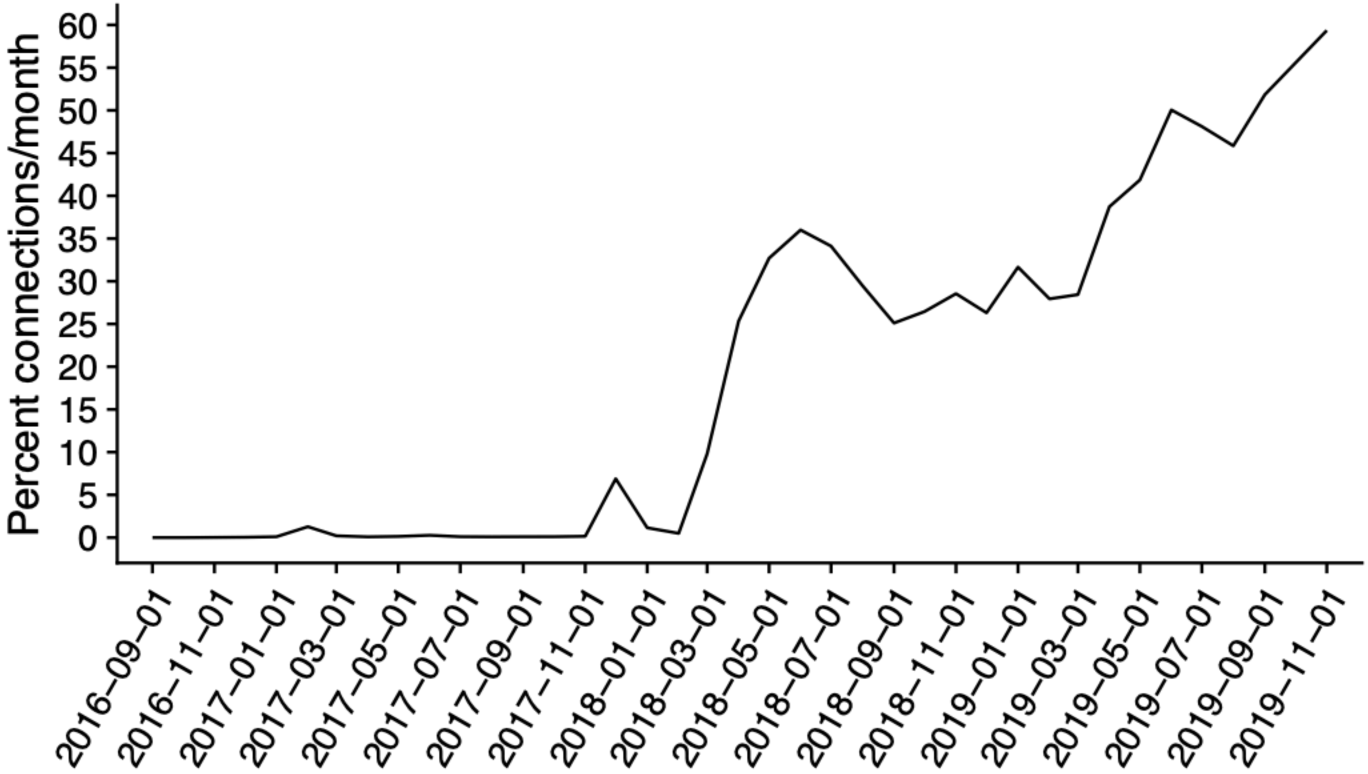
TLS 1.3 Handshake – Resumption Handshake with 0-RTT Data (Simplified)

Client		**Server**
ClientHello(ClientNonce) ClientKeyShare* pre_shared_key(PSK List) ApplicationData	→	
	←	ServerHello(ServerNonce) ServerKeyShare* pre_shared_key EncryptedExtensions ServerFinished ApplicationData
ClientFinished	→	
ApplicationData	↔	ApplicationData

0-RTT data protected using key derived from PSK

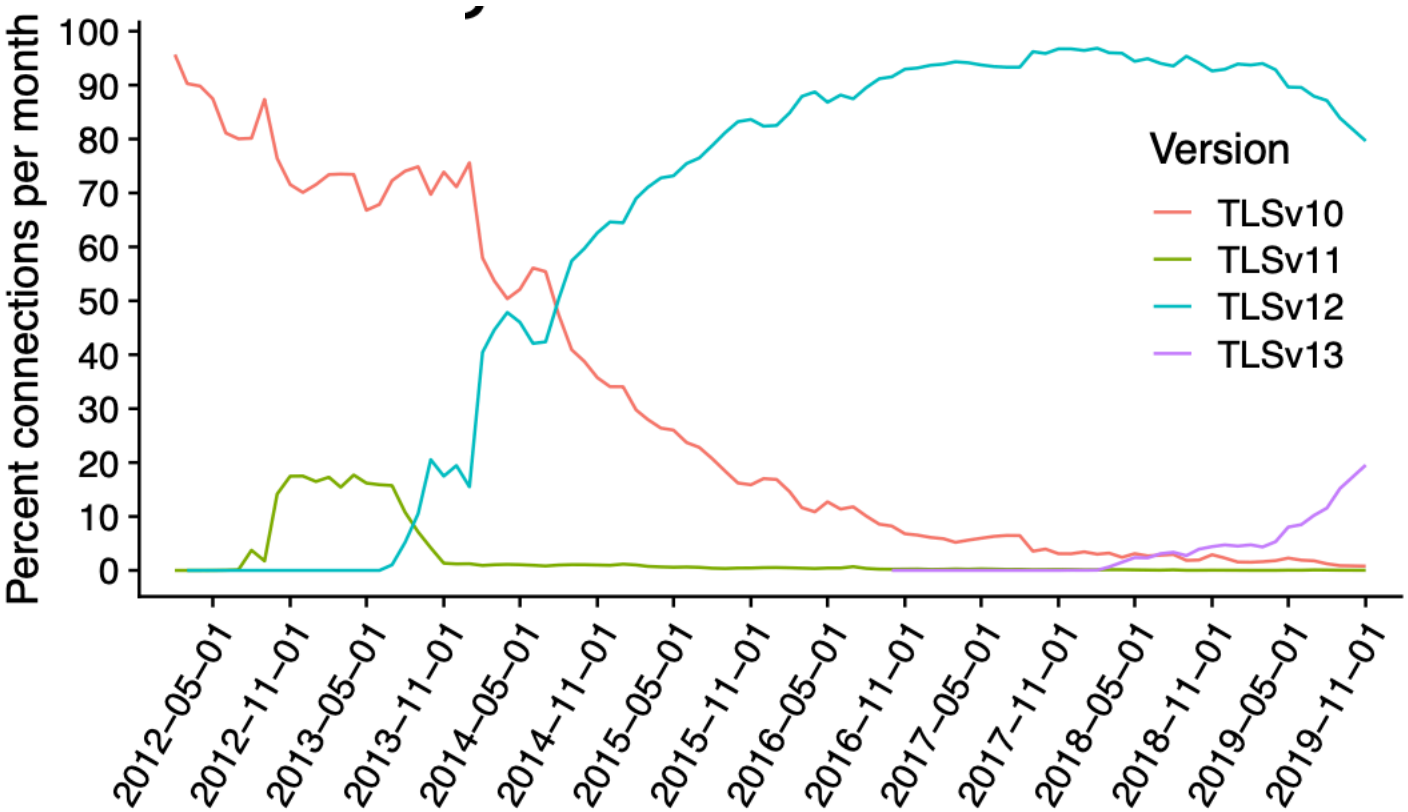
The Future of TLS

TLS 1.3 – Are We There Yet?



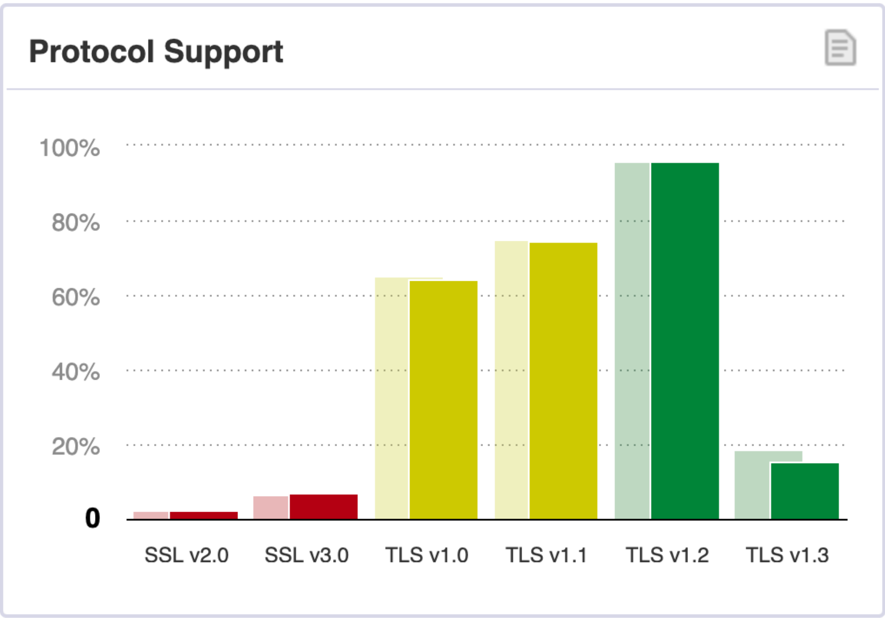
- TLS 1.3 **offered by client**, figures from Joanna Amann, Real World Crypto 2020, based on passive observation of TLS connections, 2012-2019: https://rwc.iacr.org/2020/slides/Amann.pdf

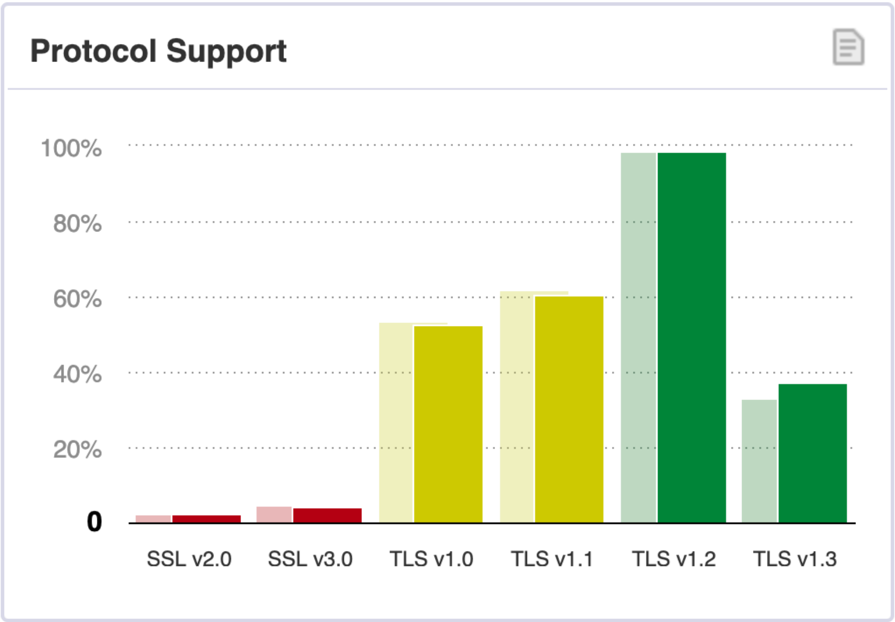
TLS 1.3 – Are We There Yet?



- TLS version **negotiated**, figures from Joanna Amann, Real World Crypto 2020, based on passive observation of TLS connections, 2012-2019: https://rwc.iacr.org/2020/slides/Amann.pdf

TLS 1.3 – Are We There Yet?





- **Server-side** TLS version support, figures from SSL Pulse, November 2019 and September 2020, based on a survey of approx. 150k popular TLS servers: https://www.ssllabs.com/ssl-pulse

The Future of TLS

- Looooong tail effects: some servers in Alexa top 150k still support SSLv2, SSLv3.
- Disabling TLS 1.0, 1.1 in Firefox, Chrome and other browsers is underway.
  - Delicate balance between improving security stance and ending up with insecure connections.
- ESNI/Encrypted ClientHello: encrypting more of client's first handshake message to further improve privacy.
- Post quantum cryptography:
  - New public key algorithms that resist attacks by quantum computers.
  - NIST process for standardising algorithms is now well-advanced.
  - Some experimentation (Google, CloudFlare) on deploying new algorithms in TLS via **hybrid** key exchanges.
- It will be interesting to observe the battle between TLS 1.3, QUIC and IP-layer alternatives (IPsec, WireGuard).

Concluding Remarks

Concluding Remarks

- TLS uses mostly “boring” cryptography yet is a very complex protocol suite.
- Some protocol design errors were made, but not too many.
- Legacy support for `EXPORT` cipher suites and long tail of old versions opened up serious vulnerabilities.
- Lack of formal state-machine description, lack of API specification, and sheer complexity of specifications have led to many serious implementation errors.
- Poor algorithm choices in the Record Protocol should have been retired more aggressively.
- Most of this has been fixed in TLS 1.3.

Concluding Remarks

- TLS 1.3 was developed hand-in-hand with formal security analysis.
- The design changed many times, often changes driven by security concerns identified through the analysis.
- Main tools:
  - Hand-generated proofs in the computational setting based on pseudo-code models, e.g. Dowling-Fischlin-Günther-Stebila.
  - (Semi-)Automated proofs in the symbolic setting based on protocol descriptions extracted from RFC, e.g. Cremers-Horvat-Hoyland-Scott-van der Merwe.
  - Automated proofs based on implementations in high-level languages that compile to efficient run-time code in a (hopefully) sound way, e.g. Bhargavan *et al.*

Concluding Remarks

- Cryptography has evolved significantly in TLS.
- The largest shift was from RSA key transport to elliptic curve Diffie-Hellman, and from CBC/RC4 to AES-GCM.
- A second shift now underway is to move to using newer elliptic curves like Curve25519, allowing greater speed and better implementation security.
- A third shift is the move away from SHA-1 in certs (mostly complete).
- A future shift may be needed to incorporate post-quantum algorithms.
- But implementation vulnerabilities are bound to continue to be discovered.

Fin