TLS

And some stuff about making it Post-Quantum

Thom Wiggers, 2022-06-01, Radboud University - Applied Cryptography

>93 %

Of US Firefox page loads use TLS

Firefox Telemetry, 2022-05-30

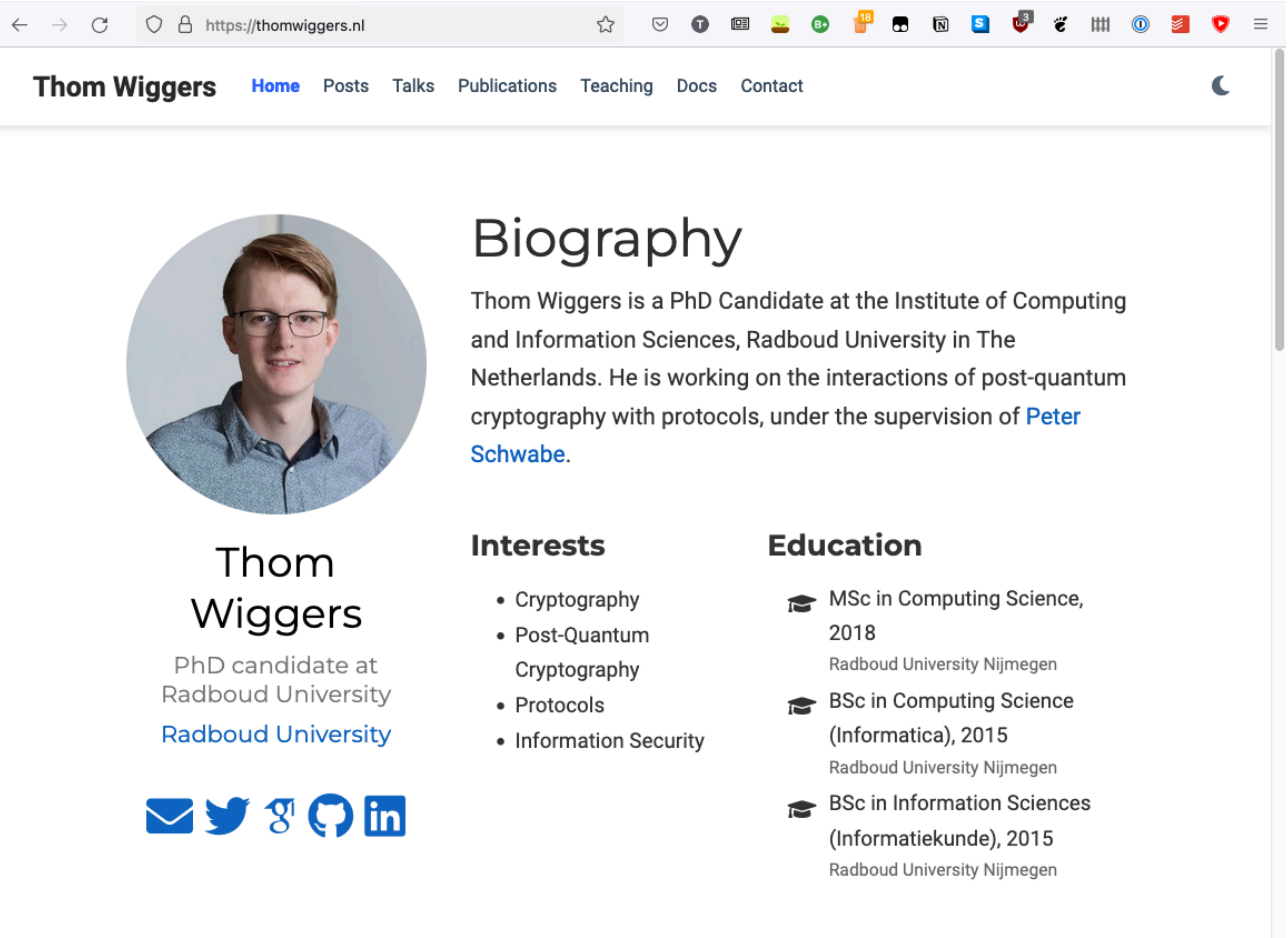
Agenda

What are we talking about today

- TLS
  - History
  - TLS 1.3
- PKI
- Making stuff PQ
  - PQTLS
  - KEMTLS

> “TLS allows client/server applications to communicate over the Internet in a way that is designed to prevent eavesdropping, tampering, and message forgery.”

RFC 8446: The Transport Layer Security (TLS) Protocol Version 1.3



Thom Wiggers

PhD candidate at Radboud University

Radboud University

Biography

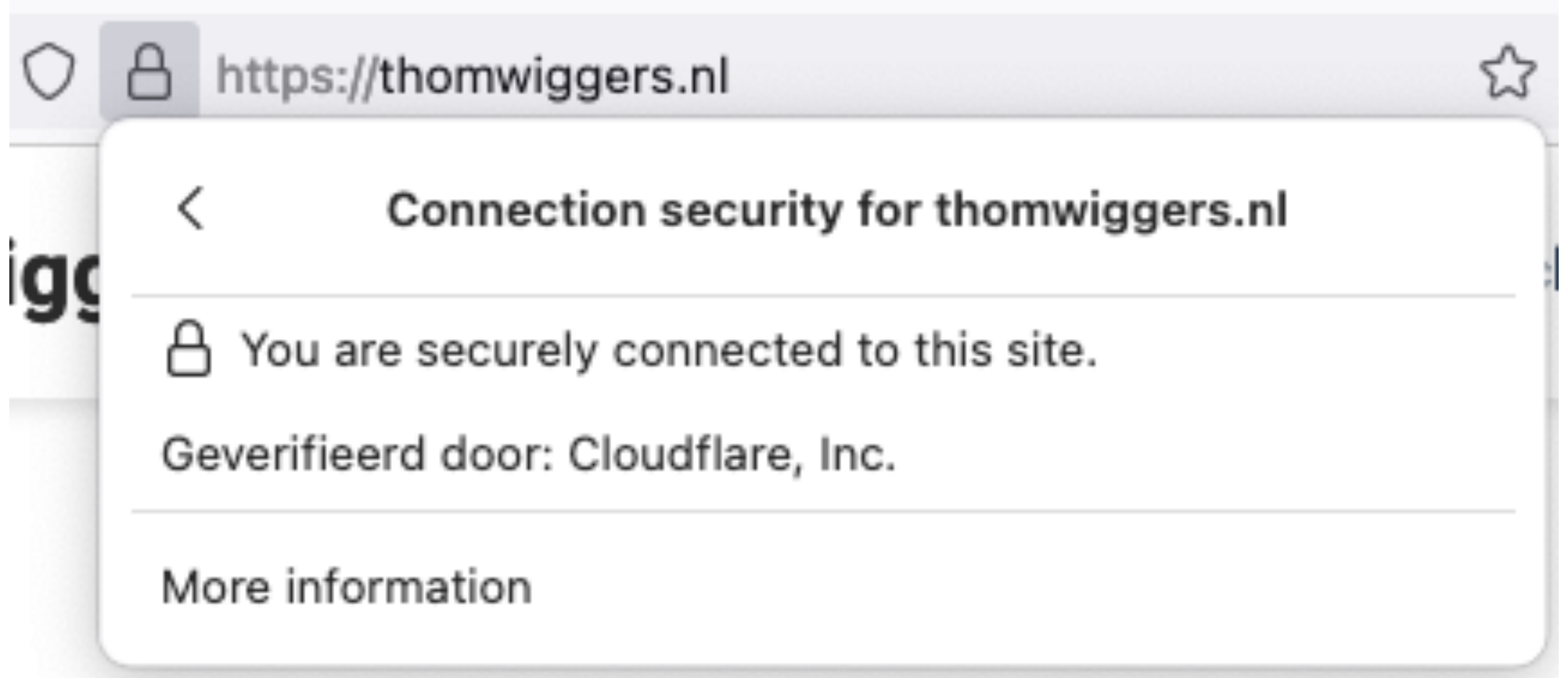
Thom Wiggers is a PhD Candidate at the Institute of Computing and Information Sciences, Radboud University in The Netherlands. He is working on the interactions of post-quantum cryptography with protocols, under the supervision of Peter Schwabe.

Interests

- Cryptography
- Post-Quantum Cryptography
- Protocols
- Information Security

Education

- MSc in Computing Science, 2018
  Radboud University Nijmegen
- BSc in Computing Science (Informatica), 2015
  Radboud University Nijmegen
- BSc in Information Sciences (Informatiekunde), 2015
  Radboud University Nijmegen



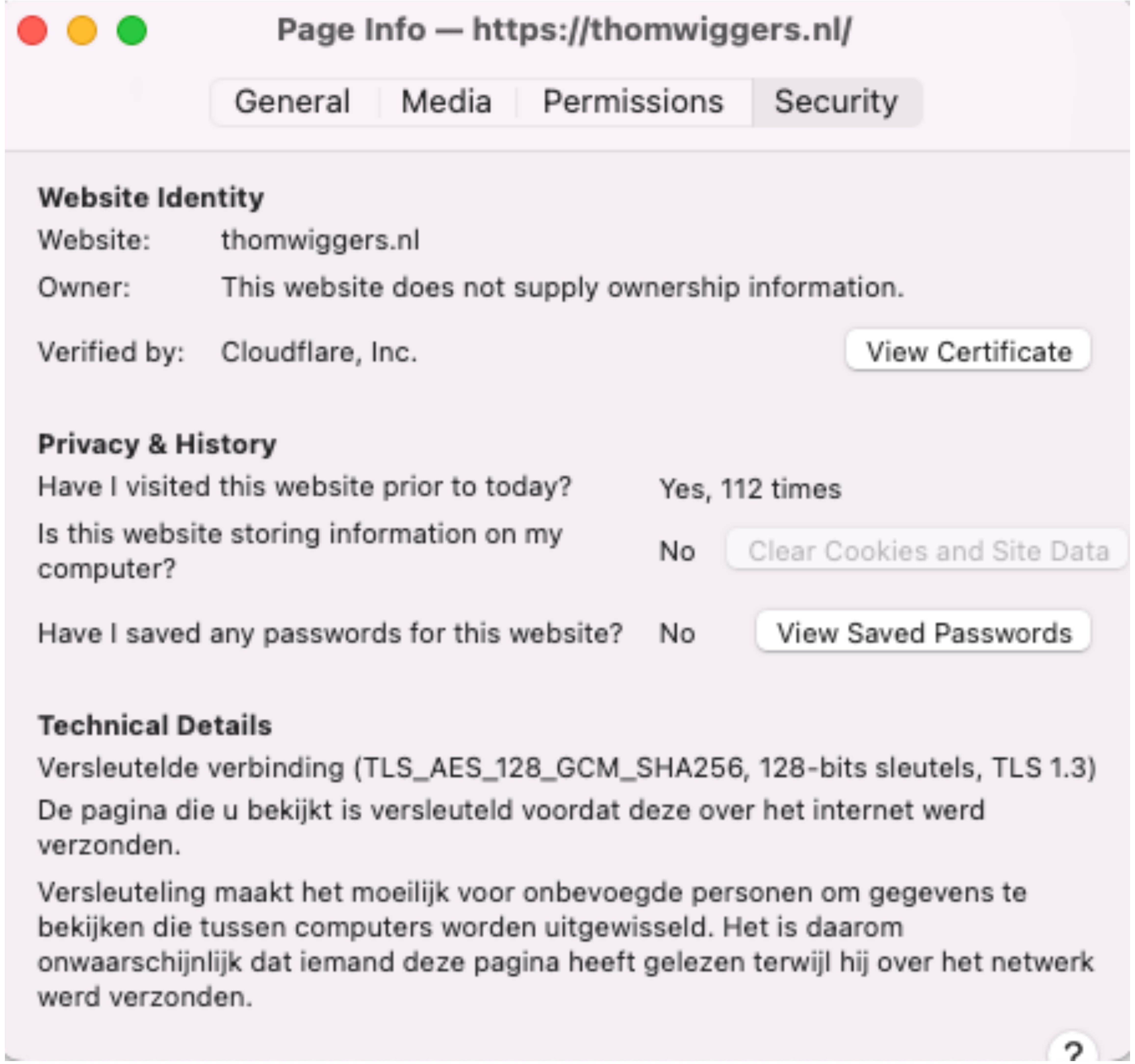
https://thomwiggers.nl

Connection security for thomwiggers.nl

You are securely connected to this site.

Geverifieerd door: Cloudflare, Inc.

More information



TLS

Checklist

Requirements for TLS

- □ Establish a shared secret key for application traffic
- □ Transmit the identity during the protocol
  - so we don’t need to know it beforehand
- □ Be Secure

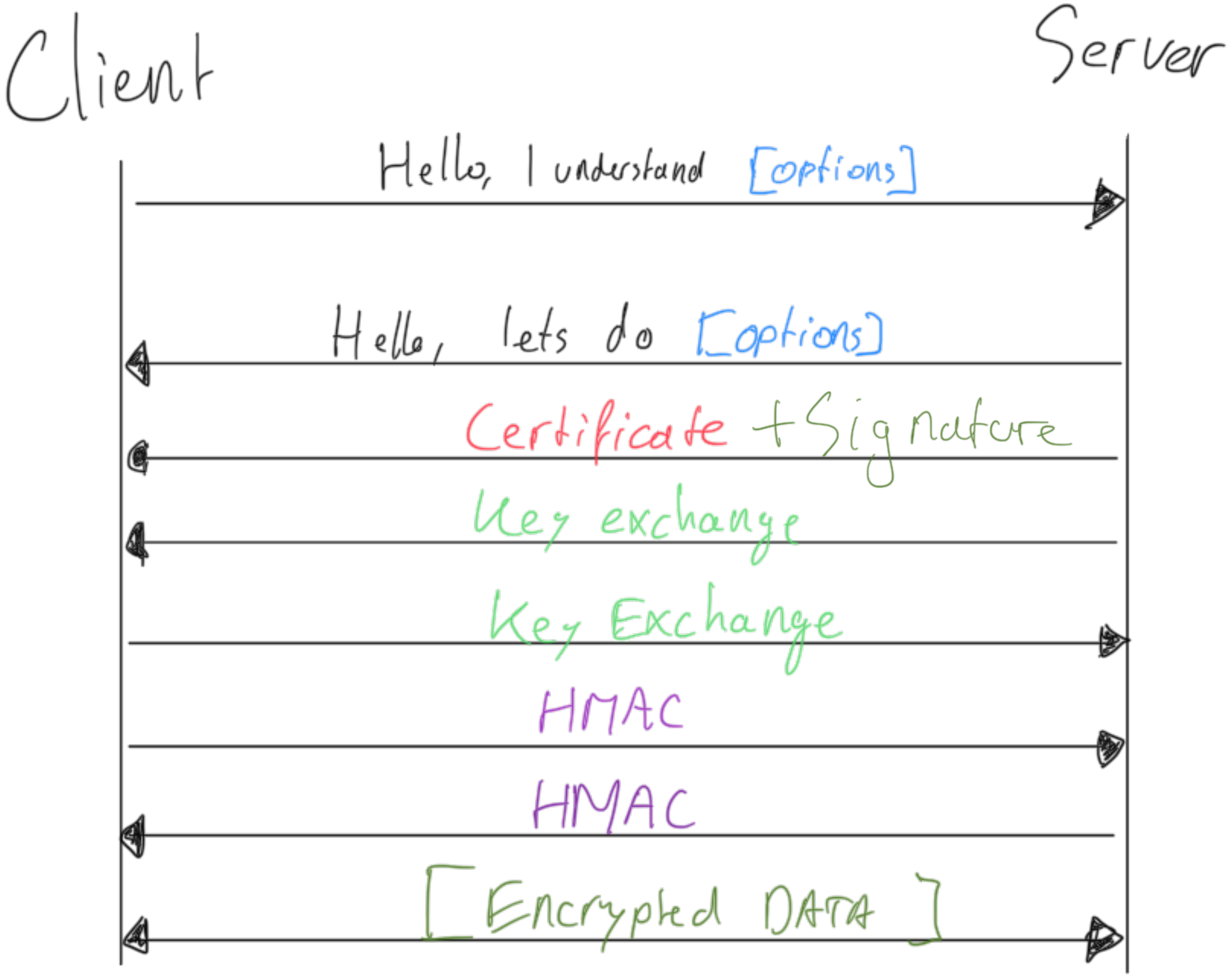
Transport Layer Security

A brief history

- 1995: SSL 2.0 (“Secure Sockets Layer”) ☠️ (insecure)
- 1996: SSL 3.0 update ☠️ (insecure)
    - Already fixes many problems in 2.0
- 1999: TLS 1.0 📛 (deprecated)
- 2006: TLS 1.1 📛 (deprecated)
- 2008: TLS 1.2 (okay with the right config)
- 2018: TLS 1.3

TLS 1.2 and earlier

A sketch



TLS 1.2 problems

AKA why you should use TLS 1.3

- Too many *round-trips*
- Certificates are sent in the clear
  - Everybody can see you’re connecting to wggrs.nl
  - Especially problematic for client authentication
- A lot of legacy cryptography and patches against attacks

Attacks

Attacks on TLS

A 👉🏻 very incomplete 👈🏻 history

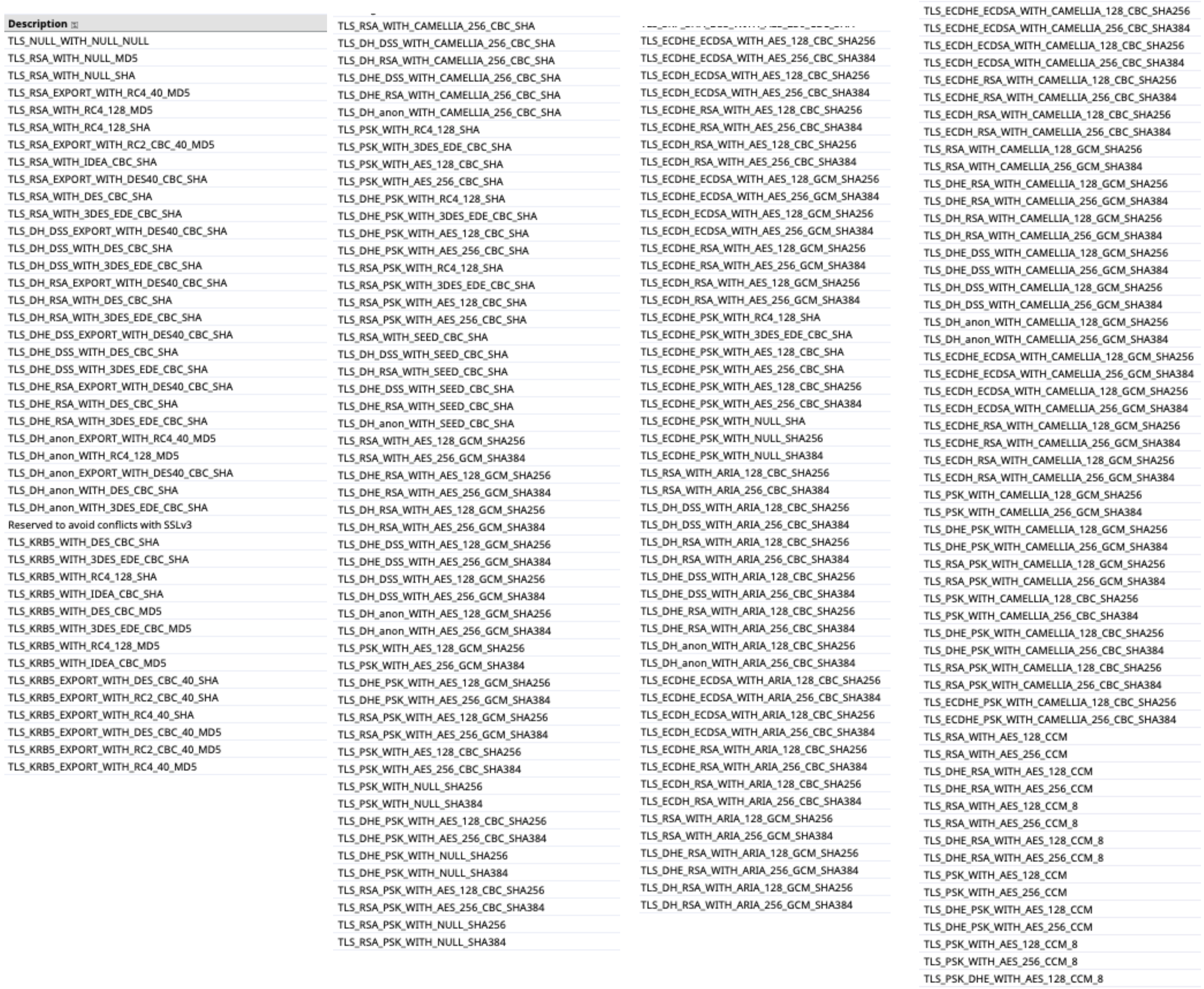
- 1998, 2006: **Bleichenbacher** breaks RSA encryption and RSA signatures using errors as side-channel
- 2011: **BEAST**: breaks SSL 3.0 and TLS 1.0 (nobody was using TLS 1.1 (2006) or 1.2 (2008)…)
  - avoid attack by using RC4 (but since 2013 RC4 is considered ☠️…)
- 2012/2013: **CRIME / BREACH**: compression in TLS is bad
- 2013: **Lucky Thirteen**: timing attack on encrypt-then-MAC
- 2014: **POODLE**: destroys SSL 3.0
- 2014: Bleichenbacher again (**BERserk**): signature forgery
- 2015/2016: **FREAK / Logjam**
  - implementation flaws downgrade to EXPORT cryptography
- 2016: **DROWN**: use the server's SSLv2 support to break SSLv3/TLS 1.{0,1,2}
- 2018: **ROBOT**: Bleichenbacher's 1998 attack is still valid on many TLS 1.2 implementations

Attacks on TLS

Some common themes

- Attacks on old versions of TLS remain valid for **decades**
    - XP, Vista, Android <5 never supported TLS 1.1, 1.2
- Many attacks are possible because legacy algorithms are never turned off by servers
    - FREAK/Logjam: 512-bit RSA/Diffie-Hellman (‘Export’ crypto)
- Setting up TLS servers is a massive headache
    - So many ciphersuites, key exchange groups, …

This isn’t even all of them!



Description
TLS_NULL_WITH_NULL_NULL
TLS_RSA_WITH_NULL_MD5
TLS_RSA_WITH_NULL_SHA
TLS_RSA_EXPORT_WITH_RC4_40_MD5
TLS_RSA_WITH_RC4_128_MD5
TLS_RSA_WITH_RC4_128_SHA
TLS_RSA_EXPORT_WITH_RC2_CBC_40_MD5
TLS_RSA_WITH_IDEA_CBC_SHA
TLS_RSA_EXPORT_WITH_DES40_CBC_SHA
TLS_RSA_WITH_DES_CBC_SHA
TLS_RSA_WITH_3DES_EDE_CBC_SHA
TLS_DH_DSS_EXPORT_WITH_DES40_CBC_SHA
TLS_DH_DSS_WITH_DES_CBC_SHA
TLS_DH_DSS_WITH_3DES_EDE_CBC_SHA
TLS_DH_RSA_EXPORT_WITH_DES40_CBC_SHA
TLS_DH_RSA_WITH_DES_CBC_SHA
TLS_DH_RSA_WITH_3DES_EDE_CBC_SHA
TLS_DHE_DSS_EXPORT_WITH_DES40_CBC_SHA
TLS_DHE_DSS_WITH_DES_CBC_SHA
TLS_DHE_DSS_WITH_3DES_EDE_CBC_SHA
TLS_DHE_RSA_EXPORT_WITH_DES40_CBC_SHA
TLS_DHE_RSA_WITH_DES_CBC_SHA
TLS_DHE_RSA_WITH_3DES_EDE_CBC_SHA
TLS_DH_anon_EXPORT_WITH_RC4_40_MD5
TLS_DH_anon_WITH_RC4_128_MD5
TLS_DH_anon_EXPORT_WITH_DES40_CBC_SHA
TLS_DH_anon_WITH_DES_CBC_SHA
TLS_DH_anon_WITH_3DES_EDE_CBC_SHA
Reserved to avoid conflicts with SSLv3
TLS_KRB5_WITH_DES_CBC_SHA
TLS_KRB5_WITH_3DES_EDE_CBC_SHA
TLS_KRB5_WITH_RC4_128_SHA
TLS_KRB5_WITH_IDEA_CBC_SHA
TLS_KRB5_WITH_DES_CBC_MD5
TLS_KRB5_WITH_3DES_EDE_CBC_MD5
TLS_KRB5_WITH_RC4_128_MD5
TLS_KRB5_WITH_IDEA_CBC_MD5
TLS_KRB5_EXPORT_WITH_DES_CBC_40_SHA
TLS_KRB5_EXPORT_WITH_RC2_CBC_40_SHA
TLS_KRB5_EXPORT_WITH_RC4_40_SHA
TLS_KRB5_EXPORT_WITH_DES_CBC_40_MD5
TLS_KRB5_EXPORT_WITH_RC2_CBC_40_MD5
TLS_KRB5_EXPORT_WITH_RC4_40_MD5
TLS_RSA_WITH_CAMELLIA_256_CBC_SHA
TLS_DH_DSS_WITH_CAMELLIA_256_CBC_SHA
TLS_DH_RSA_WITH_CAMELLIA_256_CBC_SHA
TLS_DHE_DSS_WITH_CAMELLIA_256_CBC_SHA
TLS_DHE_RSA_WITH_CAMELLIA_256_CBC_SHA
TLS_DH_anon_WITH_CAMELLIA_256_CBC_SHA
TLS_PSK_WITH_RC4_128_SHA
TLS_PSK_WITH_3DES_EDE_CBC_SHA
TLS_PSK_WITH_AES_128_CBC_SHA
TLS_PSK_WITH_AES_256_CBC_SHA
TLS_DHE_PSK_WITH_RC4_128_SHA
TLS_DHE_PSK_WITH_3DES_EDE_CBC_SHA
TLS_DHE_PSK_WITH_AES_128_CBC_SHA
TLS_DHE_PSK_WITH_AES_256_CBC_SHA
TLS_RSA_PSK_WITH_RC4_128_SHA
TLS_RSA_PSK_WITH_3DES_EDE_CBC_SHA
TLS_RSA_PSK_WITH_AES_128_CBC_SHA
TLS_RSA_PSK_WITH_AES_256_CBC_SHA
TLS_RSA_WITH_SEED_CBC_SHA
TLS_DH_DSS_WITH_SEED_CBC_SHA
TLS_DH_RSA_WITH_SEED_CBC_SHA
TLS_DHE_DSS_WITH_SEED_CBC_SHA
TLS_DHE_RSA_WITH_SEED_CBC_SHA
TLS_DH_anon_WITH_SEED_CBC_SHA
TLS_RSA_WITH_AES_128_GCM_SHA256
TLS_RSA_WITH_AES_256_GCM_SHA384
TLS_DHE_RSA_WITH_AES_128_GCM_SHA256
TLS_DHE_RSA_WITH_AES_256_GCM_SHA384
TLS_DH_RSA_WITH_AES_128_GCM_SHA256
TLS_DH_RSA_WITH_AES_256_GCM_SHA384
TLS_DHE_DSS_WITH_AES_128_GCM_SHA256
TLS_DHE_DSS_WITH_AES_256_GCM_SHA384
TLS_DH_DSS_WITH_AES_128_GCM_SHA256
TLS_DH_DSS_WITH_AES_256_GCM_SHA384
TLS_DH_anon_WITH_AES_128_GCM_SHA256
TLS_DH_anon_WITH_AES_256_GCM_SHA384
TLS_PSK_WITH_AES_128_GCM_SHA256
TLS_PSK_WITH_AES_256_GCM_SHA384
TLS_DHE_PSK_WITH_AES_128_GCM_SHA256
TLS_DHE_PSK_WITH_AES_256_GCM_SHA384
TLS_RSA_PSK_WITH_AES_128_GCM_SHA256
TLS_RSA_PSK_WITH_AES_256_GCM_SHA384
TLS_PSK_WITH_AES_128_CBC_SHA256
TLS_PSK_WITH_AES_256_CBC_SHA384
TLS_PSK_WITH_NULL_SHA256
TLS_PSK_WITH_NULL_SHA384
TLS_DHE_PSK_WITH_AES_128_CBC_SHA256
TLS_DHE_PSK_WITH_AES_256_CBC_SHA384
TLS_DHE_PSK_WITH_NULL_SHA256
TLS_DHE_PSK_WITH_NULL_SHA384
TLS_RSA_PSK_WITH_AES_128_CBC_SHA256
TLS_RSA_PSK_WITH_AES_256_CBC_SHA384
TLS_RSA_PSK_WITH_NULL_SHA256
TLS_RSA_PSK_WITH_NULL_SHA384
[illegible]
TLS_ECDHE_ECDSA_WITH_AES_128_CBC_SHA256
TLS_ECDHE_ECDSA_WITH_AES_256_CBC_SHA384
TLS_ECDH_ECDSA_WITH_AES_128_CBC_SHA256
TLS_ECDH_ECDSA_WITH_AES_256_CBC_SHA384
TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA256
TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA384
TLS_ECDH_RSA_WITH_AES_128_CBC_SHA256
TLS_ECDH_RSA_WITH_AES_256_CBC_SHA384
TLS_ECDHE_ECDSA_WITH_AES_128_GCM_SHA256
TLS_ECDHE_ECDSA_WITH_AES_256_GCM_SHA384
TLS_ECDH_ECDSA_WITH_AES_128_GCM_SHA256
TLS_ECDH_ECDSA_WITH_AES_256_GCM_SHA384
TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256
TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384
TLS_ECDH_RSA_WITH_AES_128_GCM_SHA256
TLS_ECDH_RSA_WITH_AES_256_GCM_SHA384
TLS_ECDHE_PSK_WITH_RC4_128_SHA
TLS_ECDHE_PSK_WITH_3DES_EDE_CBC_SHA
TLS_ECDHE_PSK_WITH_AES_128_CBC_SHA
TLS_ECDHE_PSK_WITH_AES_256_CBC_SHA
TLS_ECDHE_PSK_WITH_AES_128_CBC_SHA256
TLS_ECDHE_PSK_WITH_AES_256_CBC_SHA384
TLS_ECDHE_PSK_WITH_NULL_SHA
TLS_ECDHE_PSK_WITH_NULL_SHA256
TLS_ECDHE_PSK_WITH_NULL_SHA384
TLS_RSA_WITH_ARIA_128_CBC_SHA256
TLS_RSA_WITH_ARIA_256_CBC_SHA384
TLS_DH_DSS_WITH_ARIA_128_CBC_SHA256
TLS_DH_DSS_WITH_ARIA_256_CBC_SHA384
TLS_DH_RSA_WITH_ARIA_128_CBC_SHA256
TLS_DH_RSA_WITH_ARIA_256_CBC_SHA384
TLS_DHE_DSS_WITH_ARIA_128_CBC_SHA256
TLS_DHE_DSS_WITH_ARIA_256_CBC_SHA384
TLS_DHE_RSA_WITH_ARIA_128_CBC_SHA256
TLS_DHE_RSA_WITH_ARIA_256_CBC_SHA384
TLS_DH_anon_WITH_ARIA_128_CBC_SHA256
TLS_DH_anon_WITH_ARIA_256_CBC_SHA384
TLS_ECDHE_ECDSA_WITH_ARIA_128_CBC_SHA256
TLS_ECDHE_ECDSA_WITH_ARIA_256_CBC_SHA384
TLS_ECDH_ECDSA_WITH_ARIA_128_CBC_SHA256
TLS_ECDH_ECDSA_WITH_ARIA_256_CBC_SHA384
TLS_ECDHE_RSA_WITH_ARIA_128_CBC_SHA256
TLS_ECDHE_RSA_WITH_ARIA_256_CBC_SHA384
TLS_ECDH_RSA_WITH_ARIA_128_CBC_SHA256
TLS_ECDH_RSA_WITH_ARIA_256_CBC_SHA384
TLS_RSA_WITH_ARIA_128_GCM_SHA256
TLS_RSA_WITH_ARIA_256_GCM_SHA384
TLS_DHE_RSA_WITH_ARIA_128_GCM_SHA256
TLS_DHE_RSA_WITH_ARIA_256_GCM_SHA384
TLS_DH_RSA_WITH_ARIA_128_GCM_SHA256
TLS_DH_RSA_WITH_ARIA_256_GCM_SHA384
TLS_ECDHE_ECDSA_WITH_CAMELLIA_128_CBC_SHA256
TLS_ECDHE_ECDSA_WITH_CAMELLIA_256_CBC_SHA384
TLS_ECDH_ECDSA_WITH_CAMELLIA_128_CBC_SHA256
TLS_ECDH_ECDSA_WITH_CAMELLIA_256_CBC_SHA384
TLS_ECDHE_RSA_WITH_CAMELLIA_128_CBC_SHA256
TLS_ECDHE_RSA_WITH_CAMELLIA_256_CBC_SHA384
TLS_ECDH_RSA_WITH_CAMELLIA_128_CBC_SHA256
TLS_ECDH_RSA_WITH_CAMELLIA_256_CBC_SHA384
TLS_RSA_WITH_CAMELLIA_128_GCM_SHA256
TLS_RSA_WITH_CAMELLIA_256_GCM_SHA384
TLS_DHE_RSA_WITH_CAMELLIA_128_GCM_SHA256
TLS_DHE_RSA_WITH_CAMELLIA_256_GCM_SHA384
TLS_DH_RSA_WITH_CAMELLIA_128_GCM_SHA256
TLS_DH_RSA_WITH_CAMELLIA_256_GCM_SHA384
TLS_DHE_DSS_WITH_CAMELLIA_128_GCM_SHA256
TLS_DHE_DSS_WITH_CAMELLIA_256_GCM_SHA384
TLS_DH_DSS_WITH_CAMELLIA_128_GCM_SHA256
TLS_DH_DSS_WITH_CAMELLIA_256_GCM_SHA384
TLS_DH_anon_WITH_CAMELLIA_128_GCM_SHA256
TLS_DH_anon_WITH_CAMELLIA_256_GCM_SHA384
TLS_ECDHE_ECDSA_WITH_CAMELLIA_128_GCM_SHA256
TLS_ECDHE_ECDSA_WITH_CAMELLIA_256_GCM_SHA384
TLS_ECDH_ECDSA_WITH_CAMELLIA_128_GCM_SHA256
TLS_ECDH_ECDSA_WITH_CAMELLIA_256_GCM_SHA384
TLS_ECDHE_RSA_WITH_CAMELLIA_128_GCM_SHA256
TLS_ECDHE_RSA_WITH_CAMELLIA_256_GCM_SHA384
TLS_ECDH_RSA_WITH_CAMELLIA_128_GCM_SHA256
TLS_ECDH_RSA_WITH_CAMELLIA_256_GCM_SHA384
TLS_PSK_WITH_CAMELLIA_128_GCM_SHA256
TLS_PSK_WITH_CAMELLIA_256_GCM_SHA384
TLS_DHE_PSK_WITH_CAMELLIA_128_GCM_SHA256
TLS_DHE_PSK_WITH_CAMELLIA_256_GCM_SHA384
TLS_RSA_PSK_WITH_CAMELLIA_128_GCM_SHA256
TLS_RSA_PSK_WITH_CAMELLIA_256_GCM_SHA384
TLS_PSK_WITH_CAMELLIA_128_CBC_SHA256
TLS_PSK_WITH_CAMELLIA_256_CBC_SHA384
TLS_DHE_PSK_WITH_CAMELLIA_128_CBC_SHA256
TLS_DHE_PSK_WITH_CAMELLIA_256_CBC_SHA384
TLS_RSA_PSK_WITH_CAMELLIA_128_CBC_SHA256
TLS_RSA_PSK_WITH_CAMELLIA_256_CBC_SHA384
TLS_ECDHE_PSK_WITH_CAMELLIA_128_CBC_SHA256
TLS_ECDHE_PSK_WITH_CAMELLIA_256_CBC_SHA384
TLS_RSA_WITH_AES_128_CCM
TLS_RSA_WITH_AES_256_CCM
TLS_DHE_RSA_WITH_AES_128_CCM
TLS_DHE_RSA_WITH_AES_256_CCM
TLS_RSA_WITH_AES_128_CCM_8
TLS_RSA_WITH_AES_256_CCM_8
TLS_DHE_RSA_WITH_AES_128_CCM_8
TLS_DHE_RSA_WITH_AES_256_CCM_8
TLS_PSK_WITH_AES_128_CCM
TLS_PSK_WITH_AES_256_CCM
TLS_DHE_PSK_WITH_AES_128_CCM
TLS_DHE_PSK_WITH_AES_256_CCM
TLS_PSK_WITH_AES_128_CCM_8
TLS_PSK_WITH_AES_256_CCM_8
TLS_PSK_DHE_WITH_AES_128_CCM_8

Ciphersuites in TLS

Checklist

TLS 1.2 and earlier

☑ Establish a shared secret key for application traffic

☑ Transmit the identity during the protocol

- so we don’t need to know it beforehand

❗ Be secure

  ❗ Be robust against attacks

  ❗ Protect identities during the handshake

TLS 1.3

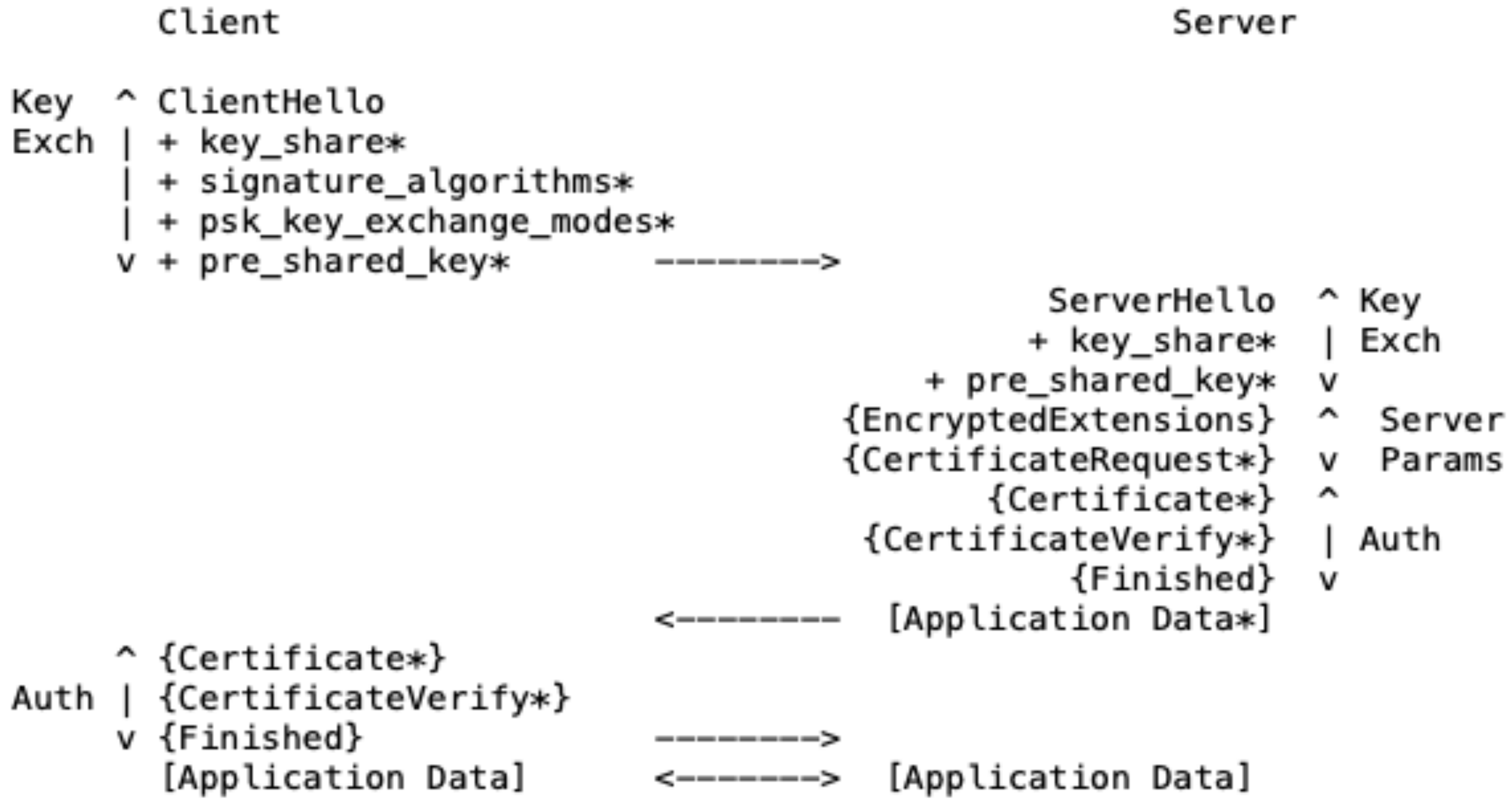
Wishlist

- Secure handshake
  - More privacy
  - Only forward secret key exchanges
  - Get rid of MD5, SHA1, 3DES, EXPORT, NULL, …
- Simplify parameters
- More robust cryptography
- Faster, 1-RTT protocol
- 0-RTT resumption

TLS 1.3

RFC8446

- Move key exchange into the first two messages
- Encrypt as much as possible
- Be done as soon as possible



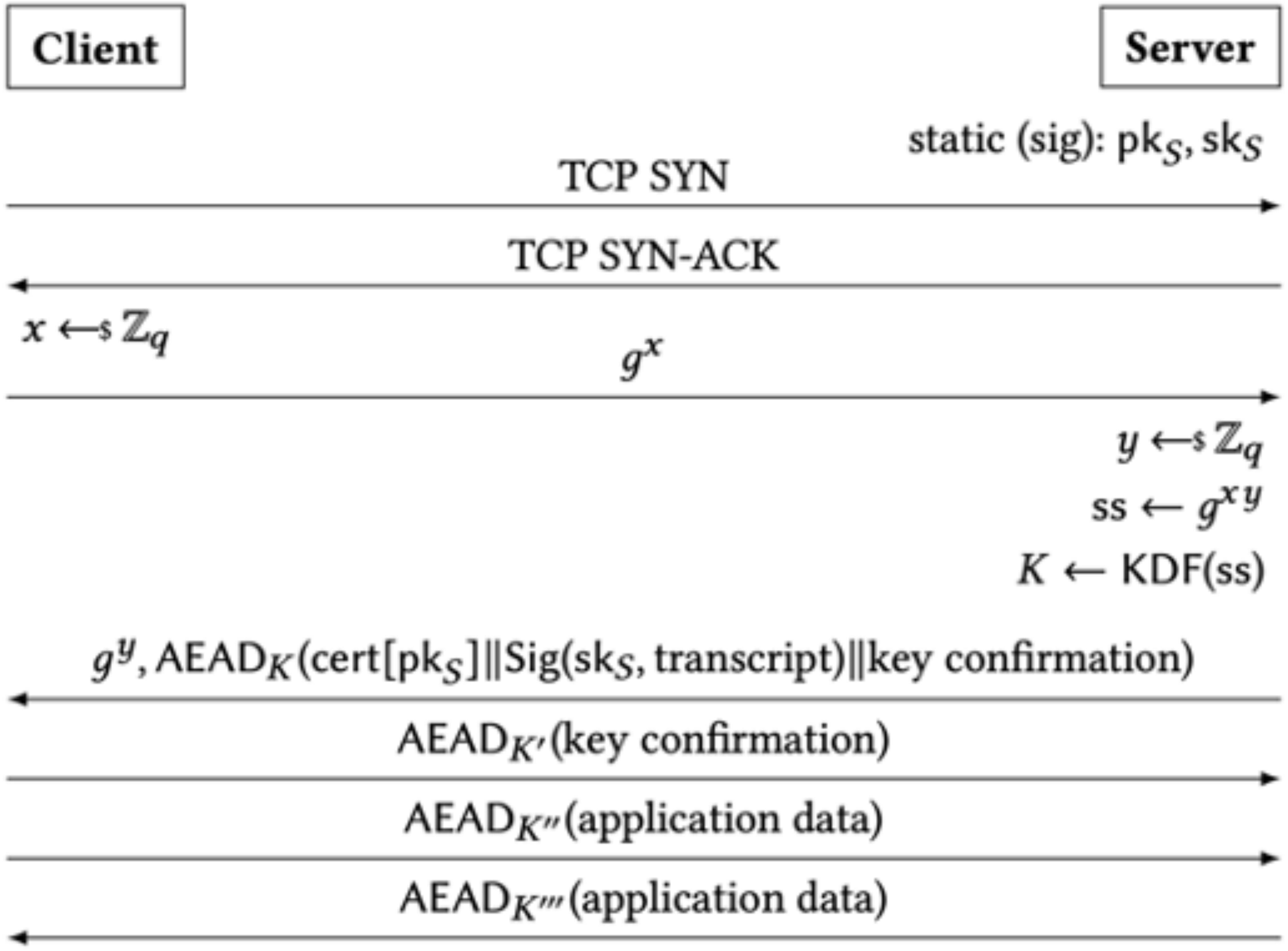
```
       Client                                           Server

Key  ^ ClientHello
Exch | + key_share*
     | + signature_algorithms*
     | + psk_key_exchange_modes*
     v + pre_shared_key*       -------->
                                                  ServerHello  ^ Key
                                                 + key_share*  | Exch
                                            + pre_shared_key*  v
                                        {EncryptedExtensions}  ^  Server
                                        {CertificateRequest*}  v  Params
                                               {Certificate*}  ^
                                         {CertificateVerify*}  | Auth
                                                   {Finished}  v
                               <--------  [Application Data*]
     ^ {Certificate*}
Auth | {CertificateVerify*}
     v {Finished}              -------->
       [Application Data]      <------->  [Application Data]
```

TLS 1.3 full handshake

The crypto

- Key exchange via ECDH
  - only ephemeral key exchange
- Server Authentication: Signature
- Handshake authentication: HMAC-SHA256
  - “key confirmation”
- AEAD: only AES-GCM or ChaCha20-Poly1305

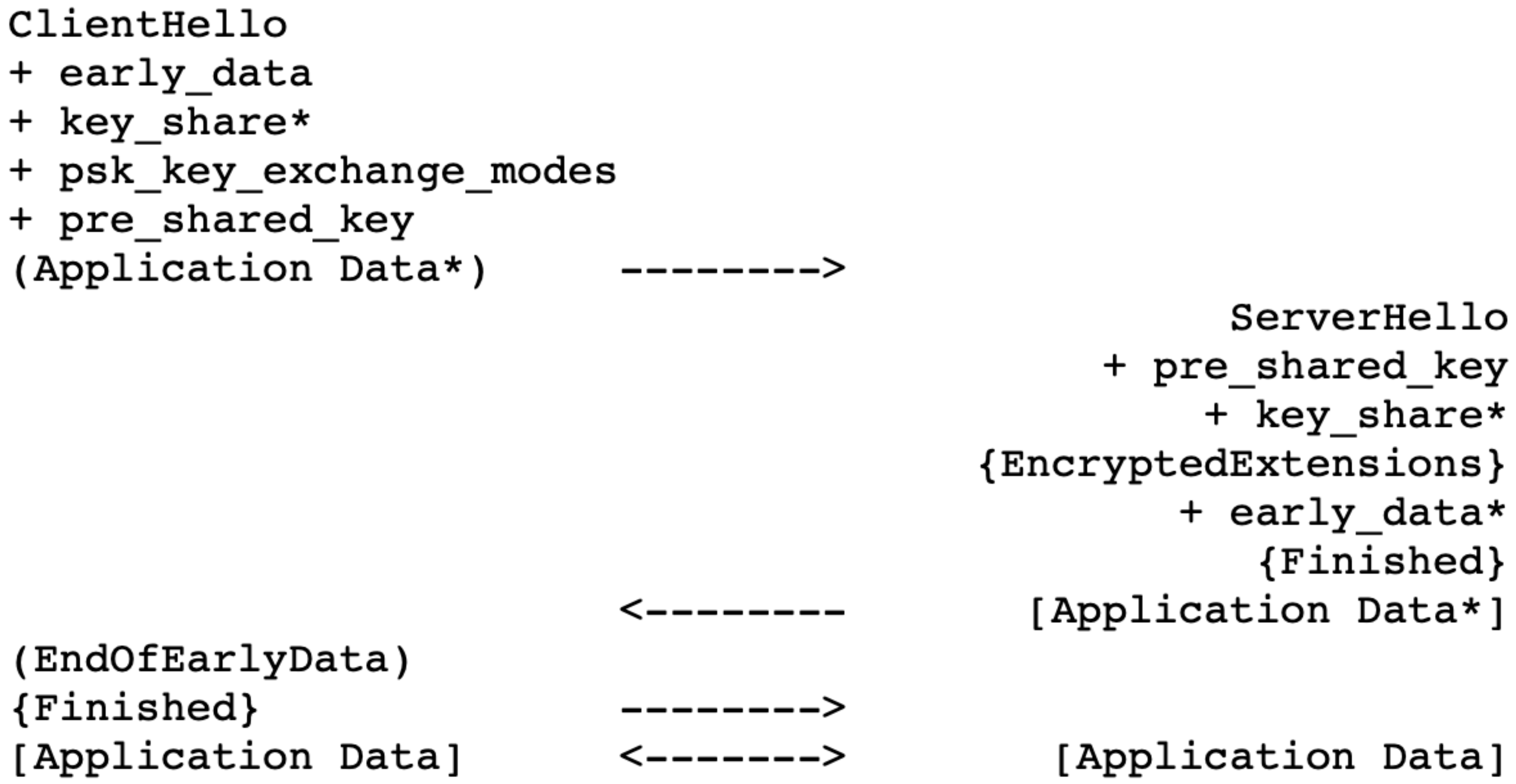

Client
Server
static (sig): $\mathsf{pk}_S, \mathsf{sk}_S$
TCP SYN
TCP SYN-ACK
$x \leftarrow_{\$} \mathbb{Z}_q$
g^x
$y \leftarrow_{\$} \mathbb{Z}_q$
$\mathsf{ss} \leftarrow g^{xy}$
$K \leftarrow \mathsf{KDF}(\mathsf{ss})$
g^y, AEAD$_K$(cert[pk$_S$]‖Sig(sk$_S$, transcript)‖key confirmation)
AEAD$_{K'}$(key confirmation)
AEAD$_{K''}$(application data)
AEAD$_{K'''}$(application data)

TLS 1.3 overview
K, K’, K”, K’”: bunch of purpose-specific keys

TLS 1.3 Resumption and 0-RTT

Got a ticket?

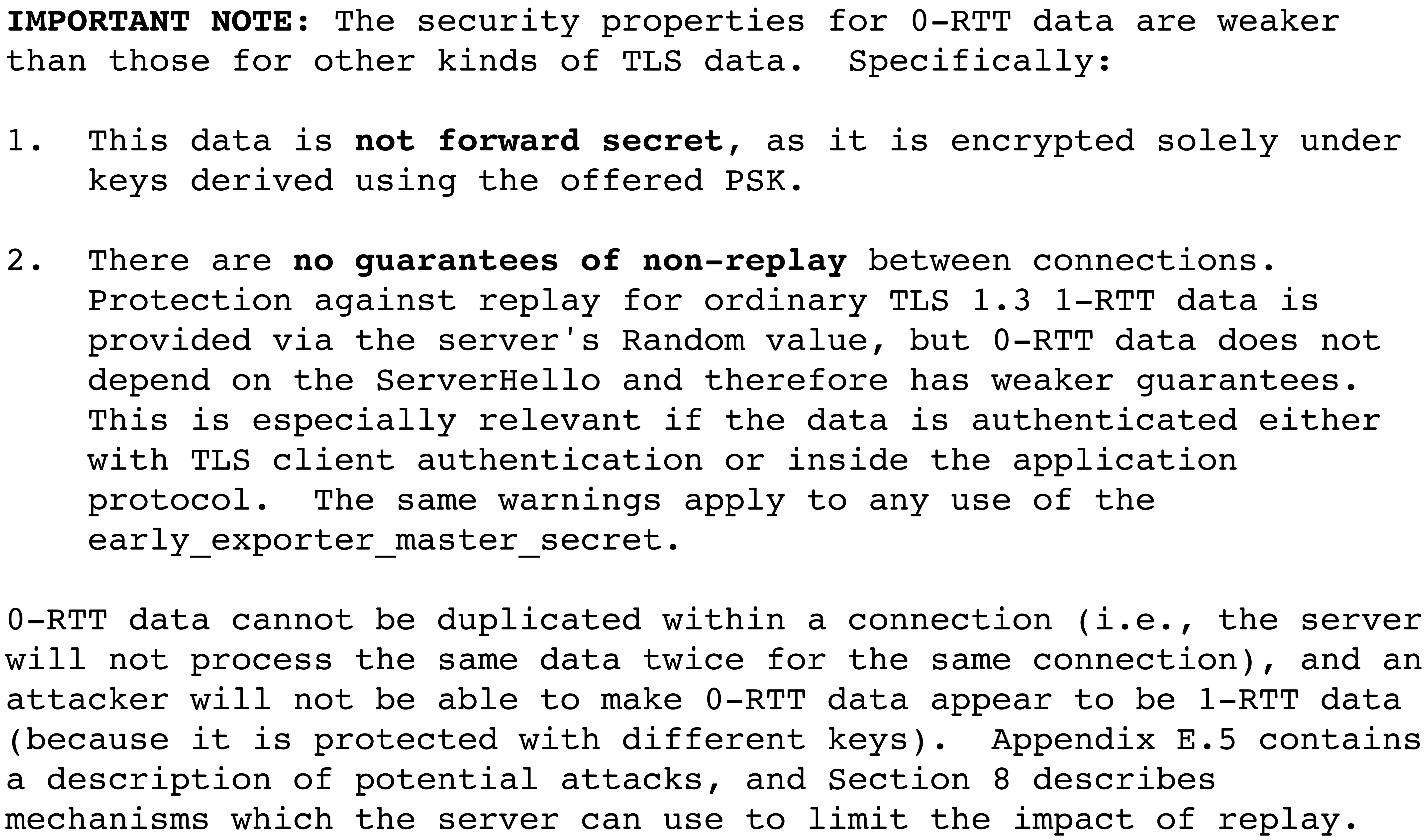
- If you have a pre-shared key, you can do a bunch of stuff faster!
- Use PSK to compute traffic secret
- Ephemeral key exchange *optional*
- Use PSK to encrypt “Early Data”



```
ClientHello
+ early_data
+ key_share*
+ psk_key_exchange_modes
+ pre_shared_key
(Application Data*)     -------->
                                                ServerHello
                                           + pre_shared_key
                                               + key_share*
                                      {EncryptedExtensions}
                                              + early_data*
                                                 {Finished}
                        <--------       [Application Data*]
(EndOfEarlyData)
{Finished}              -------->
[Application Data]      <------->        [Application Data]
```

0-RTT caveats

RFC 8446 page 18



IMPORTANT NOTE: The security properties for 0-RTT data are weaker than those for other kinds of TLS data. Specifically:

1. This data is **not forward secret**, as it is encrypted solely under keys derived using the offered PSK.

2. There are **no guarantees of non-replay** between connections. Protection against replay for ordinary TLS 1.3 1-RTT data is provided via the server's Random value, but 0-RTT data does not depend on the ServerHello and therefore has weaker guarantees. This is especially relevant if the data is authenticated either with TLS client authentication or inside the application protocol. The same warnings apply to any use of the early_exporter_master_secret.

0-RTT data cannot be duplicated within a connection (i.e., the server will not process the same data twice for the same connection), and an attacker will not be able to make 0-RTT data appear to be 1-RTT data (because it is protected with different keys). Appendix E.5 contains a description of potential attacks, and Section 8 describes mechanisms which the server can use to limit the impact of replay.

0-RTT?

For the impatient

- Siri requests
- GET requests on websites*
- Other stateless stuff

But are you sure that your application is completely robust against replays?

```
GET /?query=INSERT into payments (to, amount)
                           VALUES (“thom”, 1000);
```

TLS 1.3 standardization

A brief evaluation

- Strong collaboration with academics for protocol evaluation
  - Proofs on pen/paper, and using tools like ProVerif, Tamarin
- Academic results influenced protocol design
- But TLS working group gonna TLS working group
  - State machines are still only in the appendix

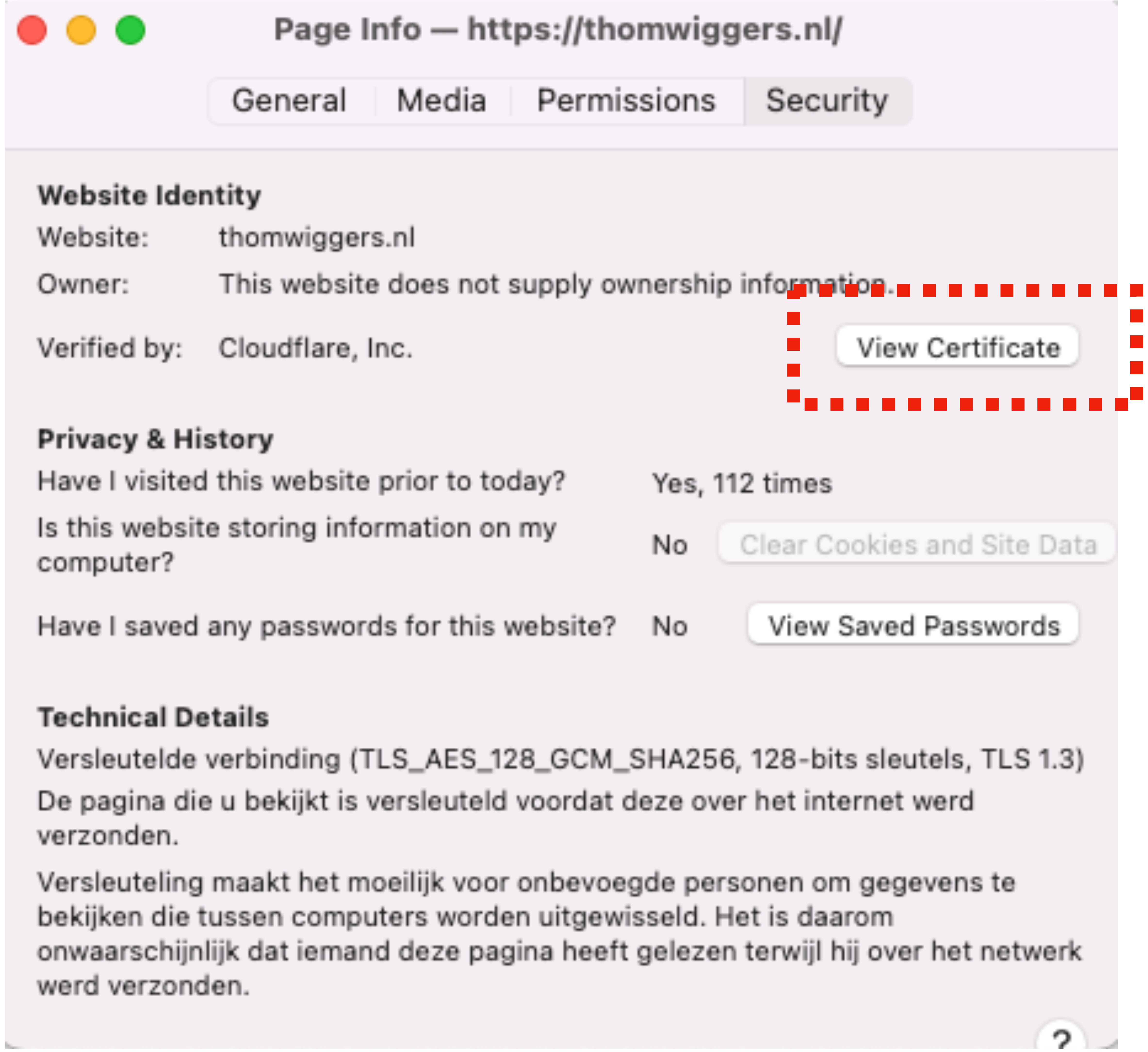
TLS 1.3

Wishlist

- ☑ Secure handshake
  - ☑ More privacy
  - ☑ Only forward secret key exchanges
  - ☑ Less MD5
- ☑ Simplify parameters
- ☑ More robust cryptography
- ☑ Faster, 1-RTT protocol
- ☑ 0-RTT resumption

□ Post-Quantum?

PKI



Website Identity

Website: thomwiggers.nl

Owner: This website does not supply ownership information.

Verified by: Cloudflare, Inc.

Privacy & History

Have I visited this website prior to today? Yes, 112 times

Is this website storing information on my computer? No

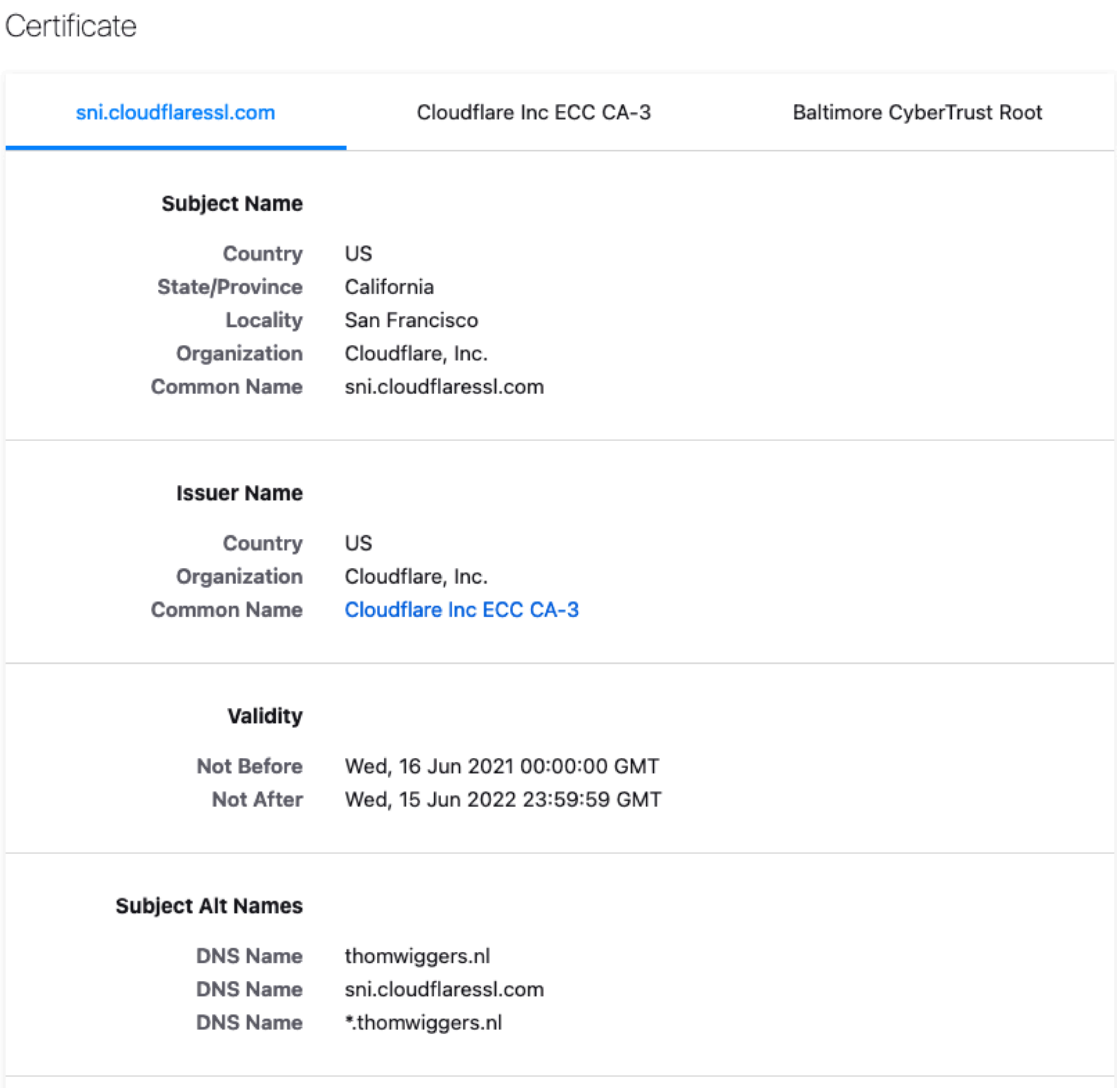
Have I saved any passwords for this website? No

Technical Details

Versleutelde verbinding (TLS_AES_128_GCM_SHA256, 128-bits sleutels, TLS 1.3)

De pagina die u bekijkt is versleuteld voordat deze over het internet werd verzonden.

Versleuteling maakt het moeilijk voor onbevoegde personen om gegevens te bekijken die tussen computers worden uitgewisseld. Het is daarom onwaarschijnlijk dat iemand deze pagina heeft gelezen terwijl hij over het netwerk werd verzonden.



Certificate

Subject Name

Country	US
State/Province	California
Locality	San Francisco
Organization	Cloudflare, Inc.
Common Name	sni.cloudflaressl.com

Issuer Name

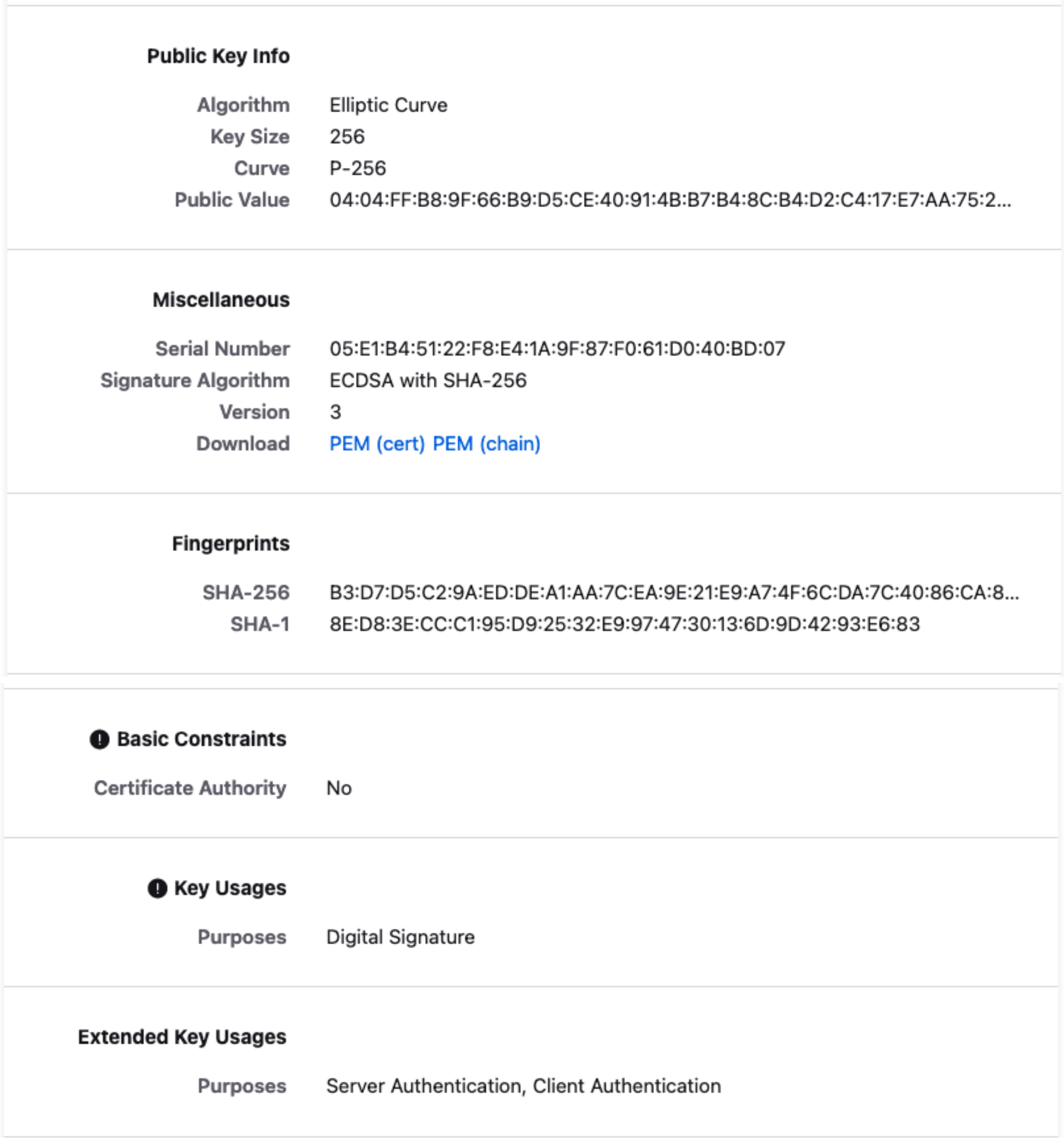
Country	US
Organization	Cloudflare, Inc.
Common Name	Cloudflare Inc ECC CA-3

Validity

Not Before	Wed, 16 Jun 2021 00:00:00 GMT
Not After	Wed, 15 Jun 2022 23:59:59 GMT

Subject Alt Names

DNS Name	thomwiggers.nl
DNS Name	sni.cloudflaressl.com
DNS Name	*.thomwiggers.nl



Public Key Info

Algorithm	Elliptic Curve
Key Size	256
Curve	P-256
Public Value	04:04:FF:B8:9F:66:B9:D5:CE:40:91:4B:B7:B4:8C:B4:D2:C4:17:E7:AA:75:2...

Miscellaneous

Serial Number	05:E1:B4:51:22:F8:E4:1A:9F:87:F0:61:D0:40:BD:07
Signature Algorithm	ECDSA with SHA-256
Version	3
Download	PEM (cert) PEM (chain)

Fingerprints

SHA-256	B3:D7:D5:C2:9A:ED:DE:A1:AA:7C:EA:9E:21:E9:A7:4F:6C:DA:7C:40:86:CA:8...
SHA-1	8E:D8:3E:CC:C1:95:D9:25:32:E9:97:47:30:13:6D:9D:42:93:E6:83

! Basic Constraints

Certificate Authority	No

! Key Usages

Purposes	Digital Signature

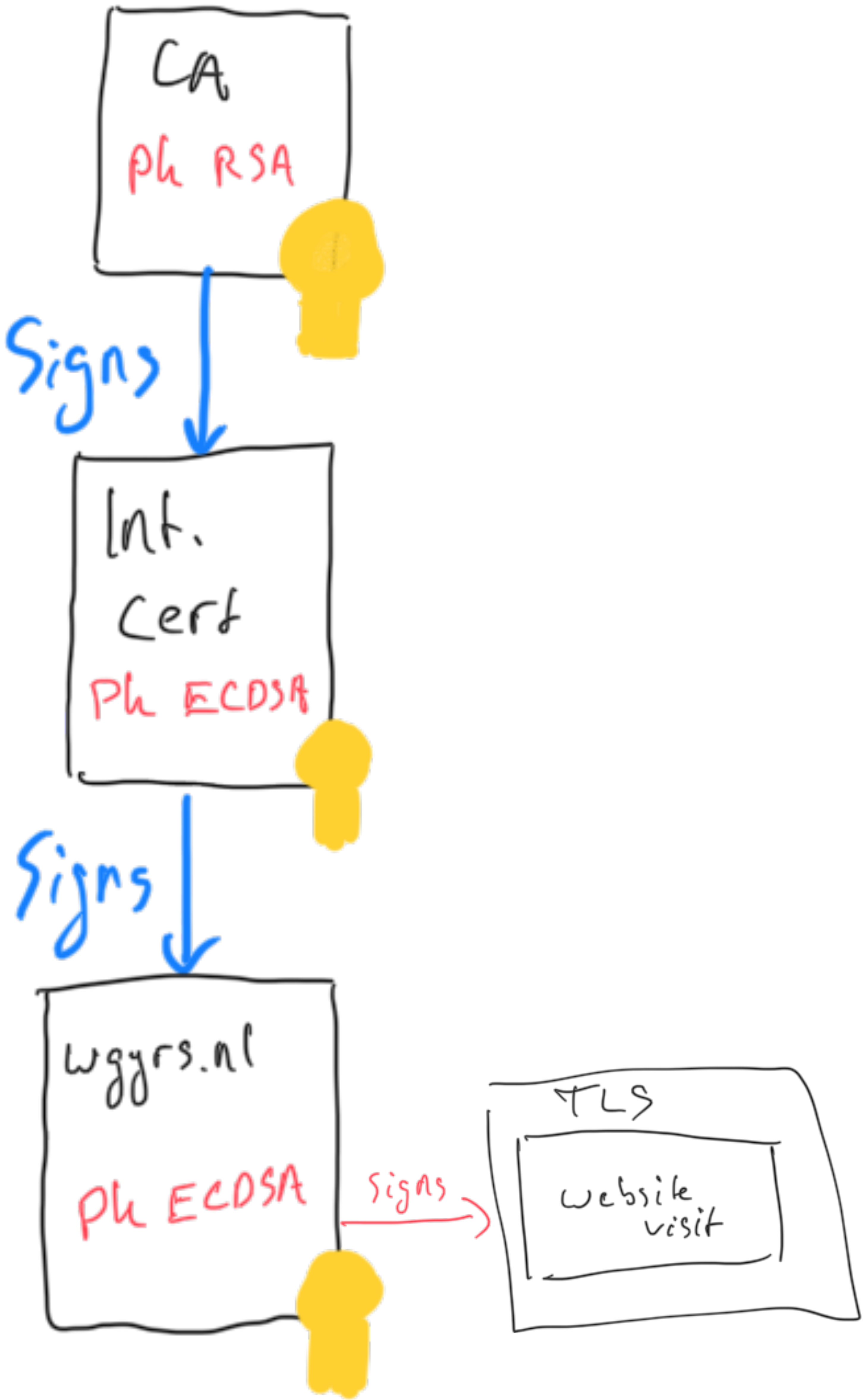
Extended Key Usages

Purposes	Server Authentication, Client Authentication

Public Key Infrastructure

Oversimplified

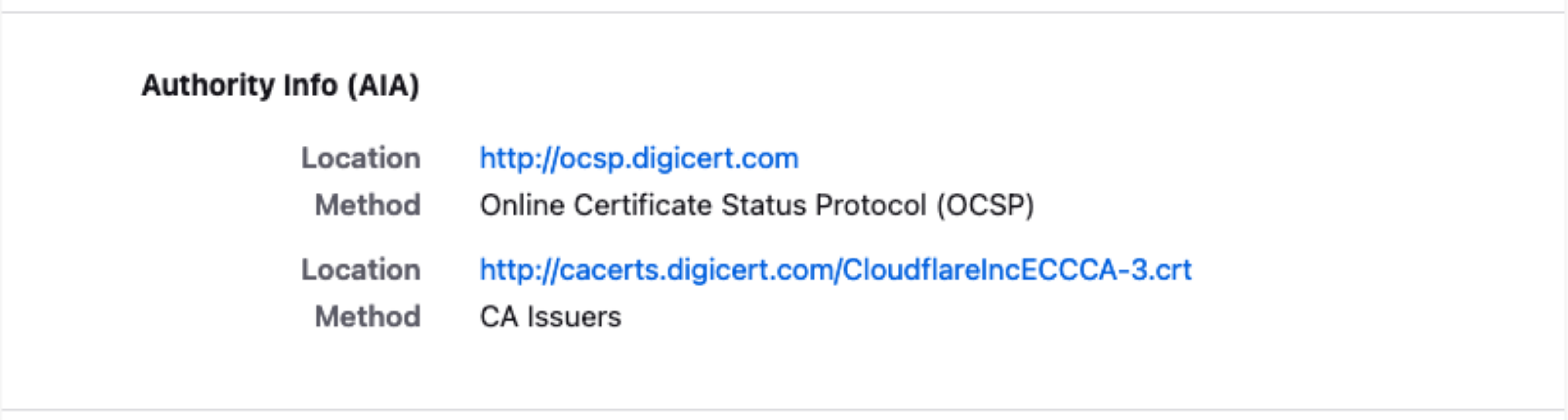
- Certificate Authorities (CA)
- Become a trusted CA by:
  - spending 💰💰 on audits
  - convince vendors to install your certificate
- Vendors trust CAs to check if I own wggrs.nl
- Intermediate CA certs make key management easier
  - (offline master signing key, etc)



Aside: PKI open problems

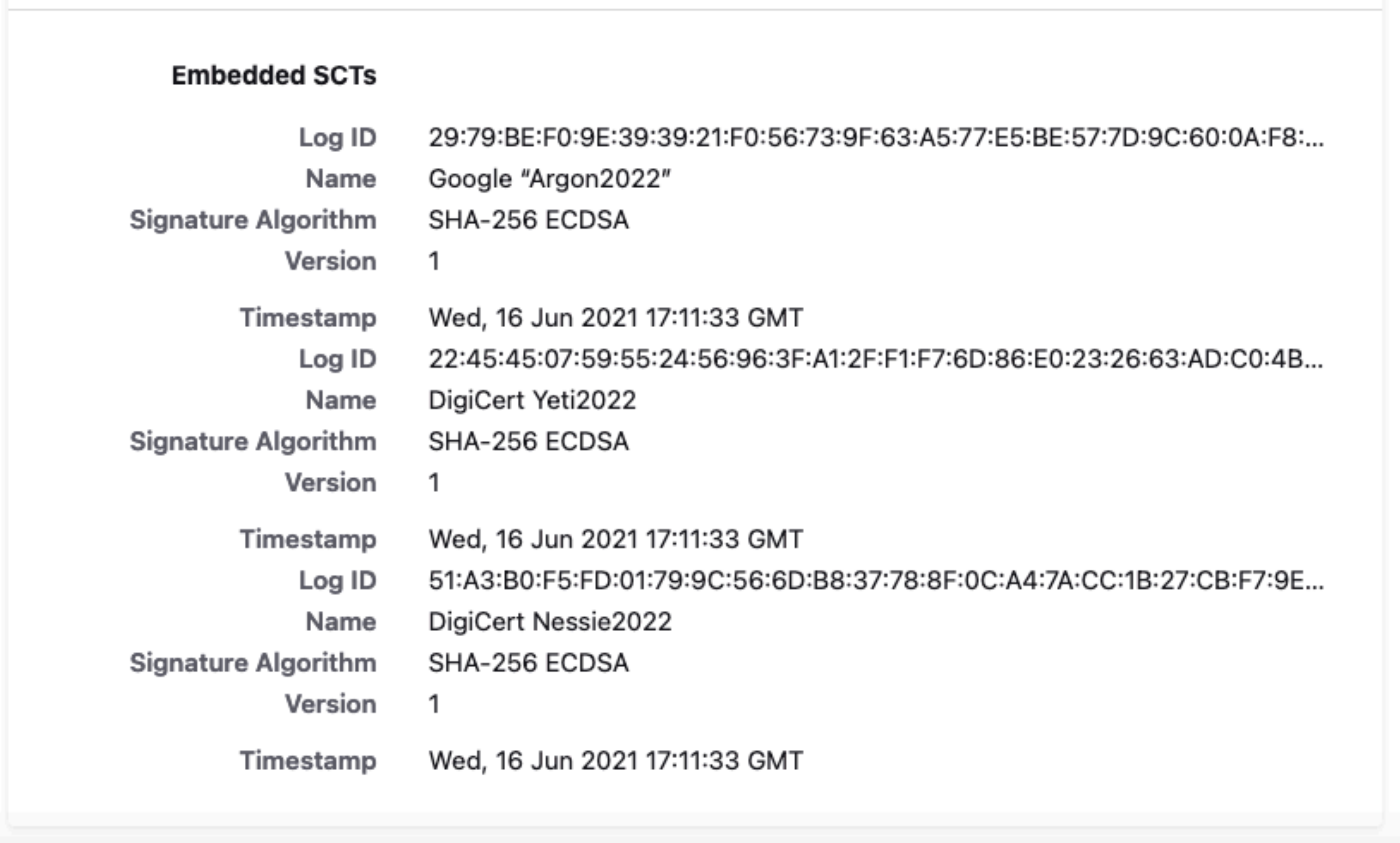
What we've oversimplified

- Certificate issuance
- Certificate Revocation
  - Certificate Revocation Lists (CRL)
  - Online Certificate Status Protocol (OCSP)
- Any trusted CA can issue a certificate for anyone
  - Famously abused by Iran(?) to attack Gmail in DigiNotar.nl hack
  - "Certificate Transparancy" (CT)



Authority Info (AIA)

Location	http://ocsp.digicert.com
Method	Online Certificate Status Protocol (OCSP)
Location	http://cacerts.digicert.com/CloudflareIncECCCA-3.crt
Method	CA Issuers

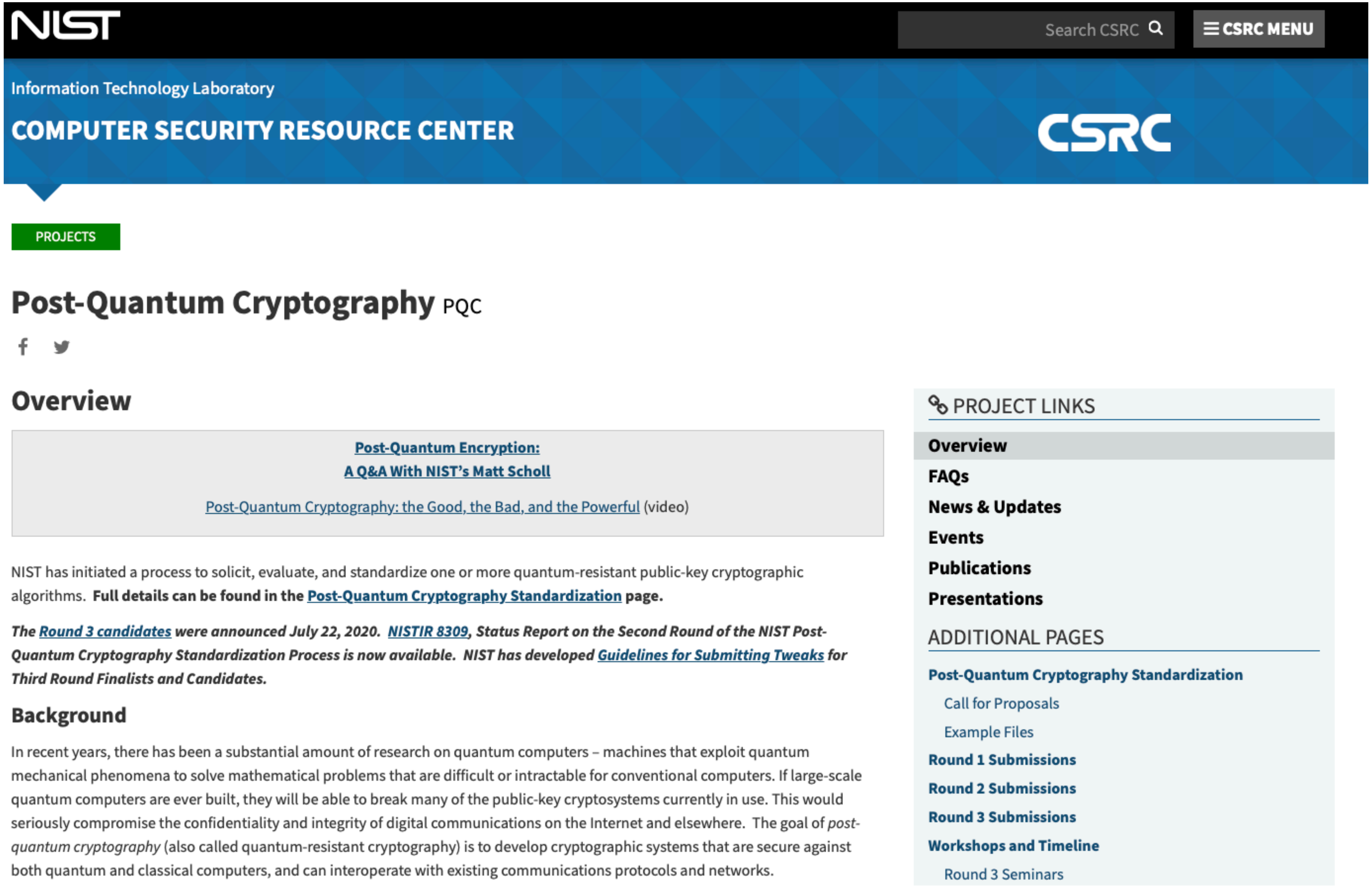


Embedded SCTs

Log ID	29:79:BE:F0:9E:39:39:21:F0:56:73:9F:63:A5:77:E5:BE:57:7D:9C:60:0A:F8:...
Name	Google “Argon2022”
Signature Algorithm	SHA-256 ECDSA
Version	1
Timestamp	Wed, 16 Jun 2021 17:11:33 GMT
Log ID	22:45:45:07:59:55:24:56:96:3F:A1:2F:F1:F7:6D:86:E0:23:26:63:AD:C0:4B...
Name	DigiCert Yeti2022
Signature Algorithm	SHA-256 ECDSA
Version	1
Timestamp	Wed, 16 Jun 2021 17:11:33 GMT
Log ID	51:A3:B0:F5:FD:01:79:9C:56:6D:B8:37:78:8F:0C:A4:7A:CC:1B:27:CB:F7:9E...
Name	DigiCert Nessie2022
Signature Algorithm	SHA-256 ECDSA
Version	1
Timestamp	Wed, 16 Jun 2021 17:11:33 GMT

Post-Quantum TLS

Post-Quantum Crypto


NIST
Search CSRC
CSRC MENU
Information Technology Laboratory
COMPUTER SECURITY RESOURCE CENTER
CSRC
PROJECTS

Post-Quantum Cryptography PQC

Overview

Post-Quantum Encryption:
A Q&A With NIST's Matt Scholl

Post-Quantum Cryptography: the Good, the Bad, and the Powerful (video)

NIST has initiated a process to solicit, evaluate, and standardize one or more quantum-resistant public-key cryptographic algorithms. **Full details can be found in the Post-Quantum Cryptography Standardization page.**

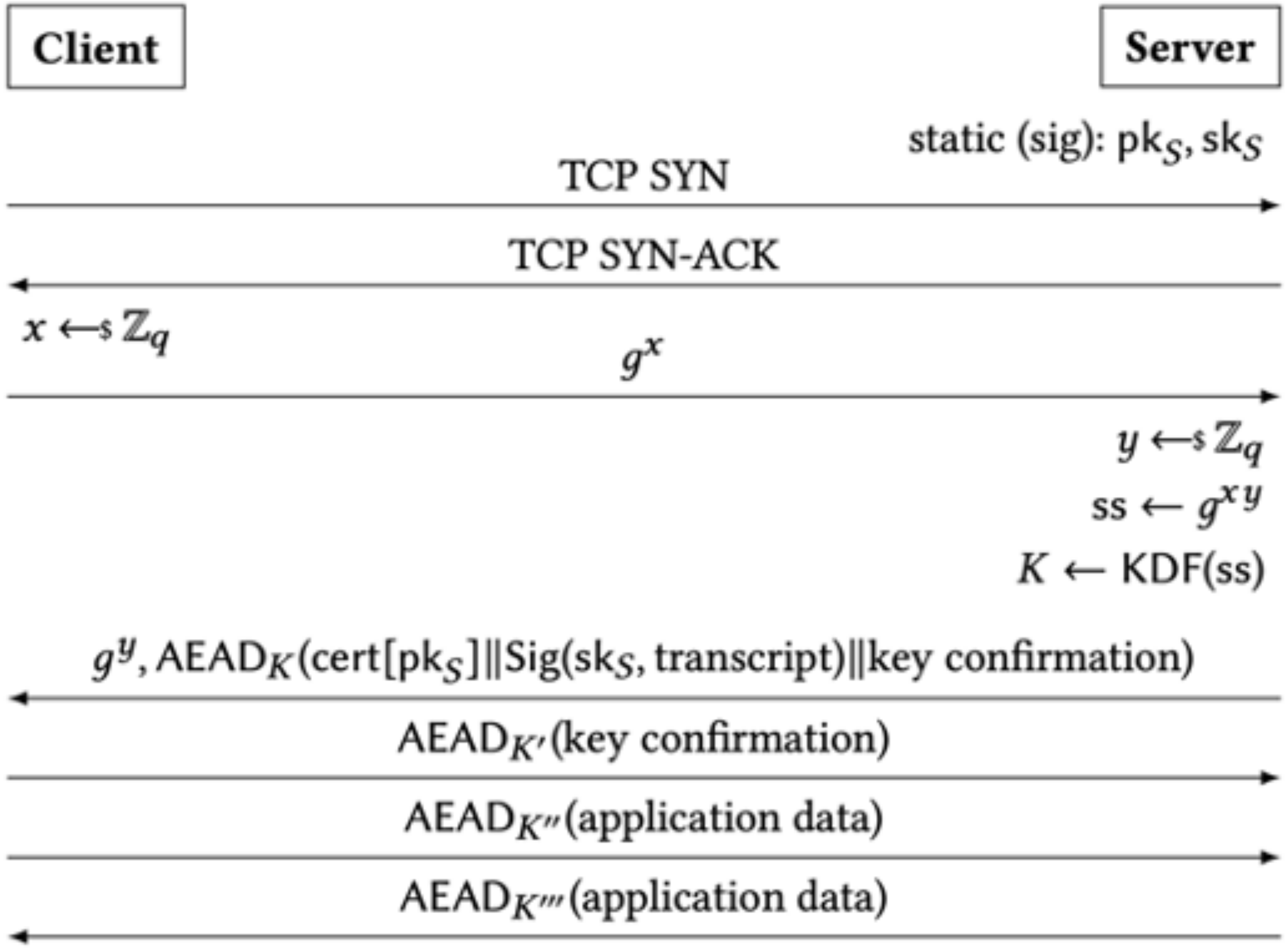
The Round 3 candidates were announced July 22, 2020. NISTIR 8309, Status Report on the Second Round of the NIST Post-Quantum Cryptography Standardization Process is now available. NIST has developed Guidelines for Submitting Tweaks for Third Round Finalists and Candidates.

Background

In recent years, there has been a substantial amount of research on quantum computers – machines that exploit quantum mechanical phenomena to solve mathematical problems that are difficult or intractable for conventional computers. If large-scale quantum computers are ever built, they will be able to break many of the public-key cryptosystems currently in use. This would seriously compromise the confidentiality and integrity of digital communications on the Internet and elsewhere. The goal of *post-quantum cryptography* (also called quantum-resistant cryptography) is to develop cryptographic systems that are secure against both quantum and classical computers, and can interoperate with existing communications protocols and networks.

TLS 1.3

Pre-Quantum



Client

Server

static (sig): $\mathrm{pk}_S, \mathrm{sk}_S$

TCP SYN

TCP SYN-ACK

$x \leftarrow_{\$} \mathbb{Z}_q$

g^x

$y \leftarrow_{\$} \mathbb{Z}_q$

$\mathrm{ss} \leftarrow g^{xy}$

$K \leftarrow \mathrm{KDF}(\mathrm{ss})$

$g^y, \mathrm{AEAD}_K(\mathrm{cert}[\mathrm{pk}_S] \| \mathrm{Sig}(\mathrm{sk}_S, \mathrm{transcript}) \| \mathrm{key\ confirmation})$

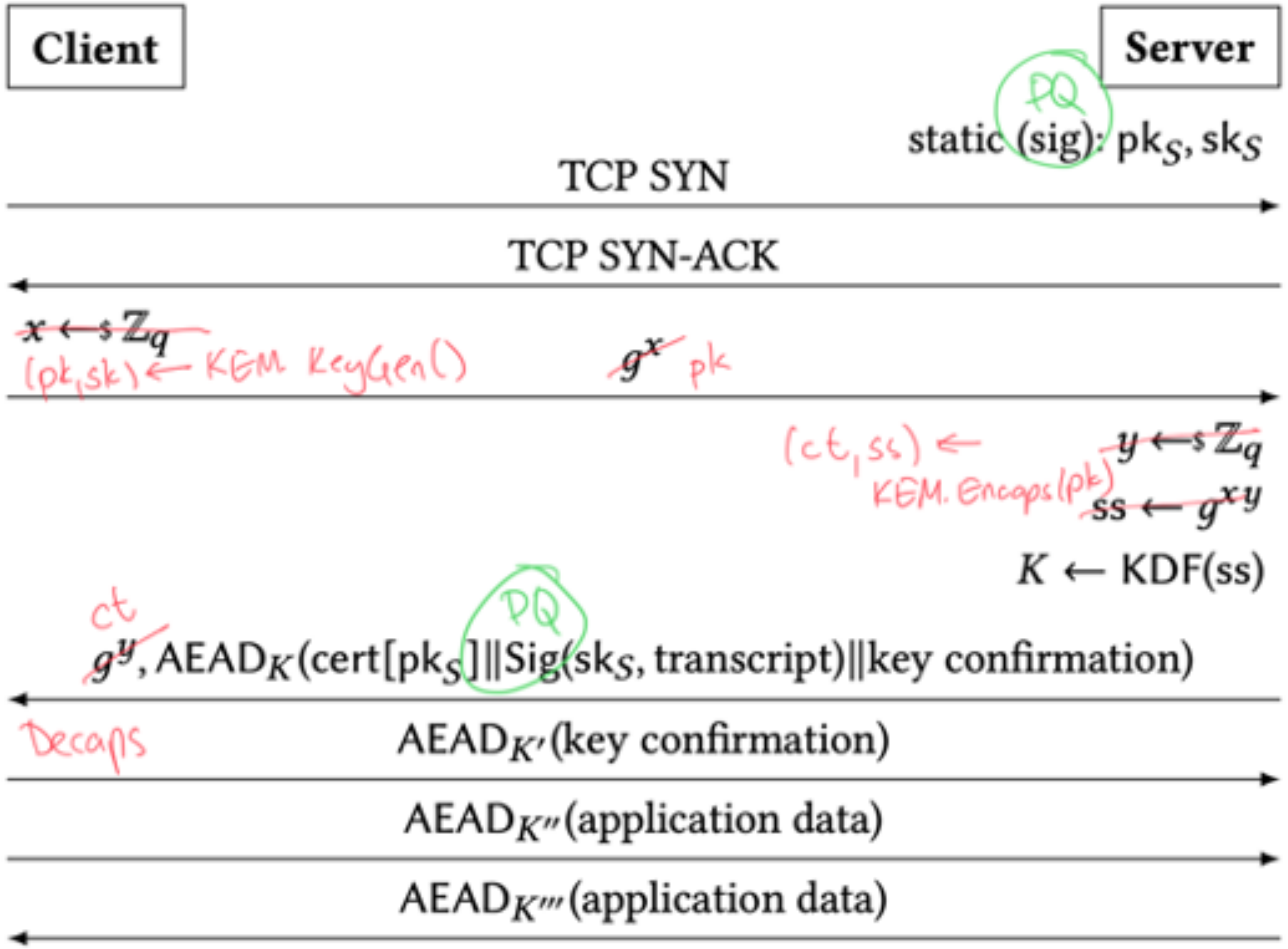
$\mathrm{AEAD}_{K'}(\mathrm{key\ confirmation})$

$\mathrm{AEAD}_{K''}(\mathrm{application\ data})$

$\mathrm{AEAD}_{K'''}(\mathrm{application\ data})$

TLS 1.3

Post-Quantum!!!1!



Client

Server

PQ

static (sig): $\mathrm{pk}_S, \mathrm{sk}_S$

TCP SYN

TCP SYN-ACK

~~$x \leftarrow_{\$} \mathbb{Z}_q$~~

(pk, sk) ← KEM KeyGen()

~~g^x~~ pk

(ct, ss) ← KEM. Encaps(pk)

~~$y \leftarrow_{\$} \mathbb{Z}_q$~~

~~$\mathrm{ss} \leftarrow g^{xy}$~~

$K \leftarrow \mathrm{KDF}(\mathrm{ss})$

ct

PQ

~~g^y~~, $\mathrm{AEAD}_K(\mathrm{cert}[\mathrm{pk}_S] \| \mathrm{Sig}(\mathrm{sk}_S, \mathrm{transcript}) \| \mathrm{key\ confirmation})$

Decaps

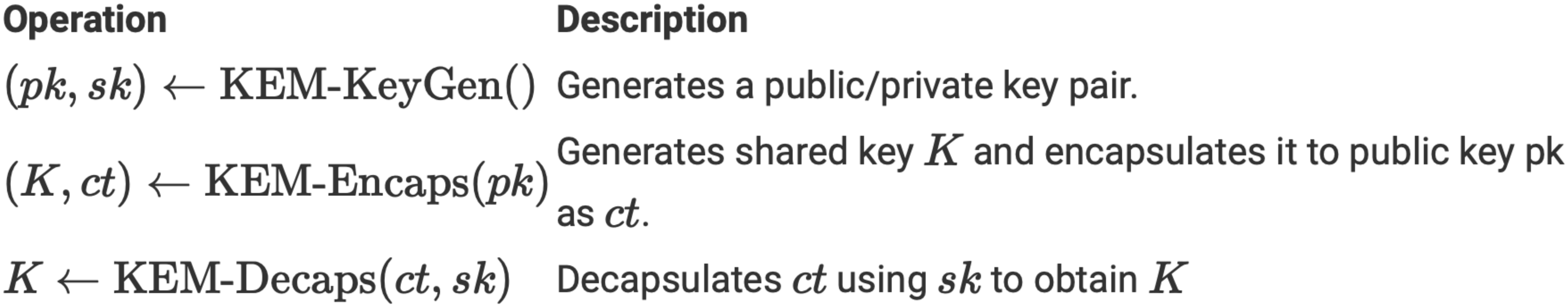
$\mathrm{AEAD}_{K'}(\mathrm{key\ confirmation})$

$\mathrm{AEAD}_{K''}(\mathrm{application\ data})$

$\mathrm{AEAD}_{K'''}(\mathrm{application\ data})$

Post-Quantum KEMs

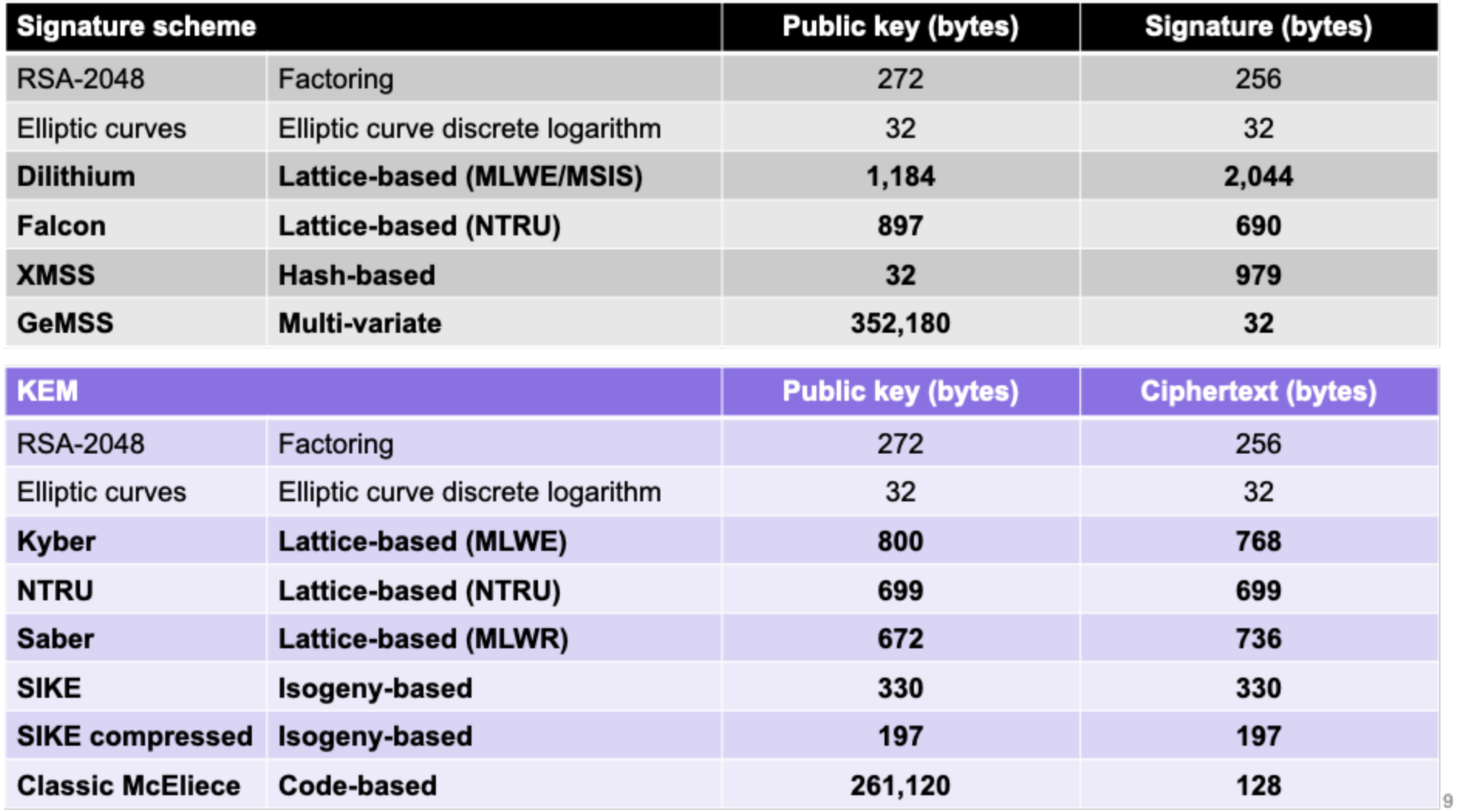
Key Encapsulation Mechanisms



Operation	Description
$(pk, sk) \leftarrow \text{KEM-KeyGen}()$	Generates a public/private key pair.
$(K, ct) \leftarrow \text{KEM-Encaps}(pk)$	Generates shared key K and encapsulates it to public key pk as ct.
$K \leftarrow \text{KEM-Decaps}(ct, sk)$	Decapsulates ct using sk to obtain K

Post-Quantum key sizes

New tradeoffs in cryptography



Signature scheme		Public key (bytes)	Signature (bytes)
RSA-2048	Factoring	272	256
Elliptic curves	Elliptic curve discrete logarithm	32	32
Dilithium	**Lattice-based (MLWE/MSIS)**	**1,184**	**2,044**
Falcon	**Lattice-based (NTRU)**	**897**	**690**
XMSS	**Hash-based**	**32**	**979**
GeMSS	**Multi-variate**	**352,180**	**32**

KEM		Public key (bytes)	Ciphertext (bytes)
RSA-2048	Factoring	272	256
Elliptic curves	Elliptic curve discrete logarithm	32	32
Kyber	**Lattice-based (MLWE)**	**800**	**768**
NTRU	**Lattice-based (NTRU)**	**699**	**699**
Saber	**Lattice-based (MLWR)**	**672**	**736**
SIKE	**Isogeny-based**	**330**	**330**
SIKE compressed	**Isogeny-based**	**197**	**197**
Classic McEliece	**Code-based**	**261,120**	**128**

PQ signatures are
big and/or
slow and/or
need hw support

💡

Use key exchange for authentication

Authentication

Explicit authentication:

Alice receives assurance that she really is talking to Bob

- Signed Diffie-Hellman
- SIGMA
- TLS 1.3

Implicit authentication:

Alice is assured that only Bob would be able to compute the shared secret

- Signal
- Wireguard
- Noise Framework

Can always use MAC to confirm key

TLS handshake authentication

Recap

- Signatures allow us to authenticate immediately!

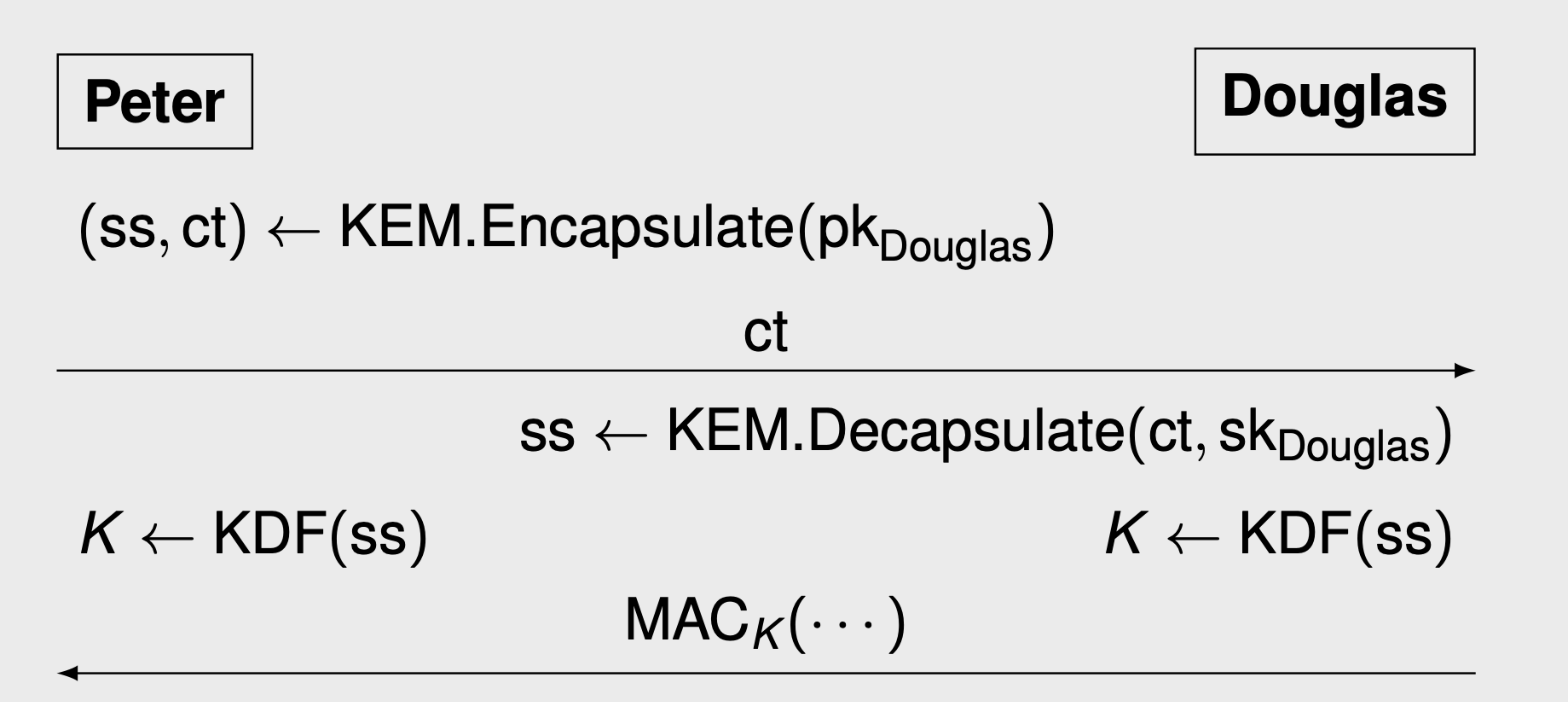
```
  Client                                        Server
 ClientHello         -------->
                     <--------         ServerHello
                                             <...>
                                <CertificateRequest>
                                       <Certificate>
                                 <CertificateVerify>
                     <--------            <Finished>
 <Certificate>
 <CertificateVerify>
 <Finished>          -------->

 [Application Data]  <------->  [Application Data]

<msg>: enc. w/ keys derived from ephemeral KEX (HS)
[msg]: enc. w/ keys derived from HS (MS)
```

Authenticated Key Exchange via KEM

An oversimplified protocol



Peter

Douglas

$(\mathsf{ss}, \mathsf{ct}) \leftarrow \mathsf{KEM.Encapsulate}(\mathsf{pk}_{\mathsf{Douglas}})$

ct

$\mathsf{ss} \leftarrow \mathsf{KEM.Decapsulate}(\mathsf{ct}, \mathsf{sk}_{\mathsf{Douglas}})$

$K \leftarrow \mathsf{KDF}(\mathsf{ss})$

$K \leftarrow \mathsf{KDF}(\mathsf{ss})$

$\mathsf{MAC}_K(\cdots)$

Note that this protocol assumes that we have already exchanged the public keys!

TLS authentication via KEM

Naively

- Signatures allow us to authenticate immediately!

- KEMs require interactivity

- Exercise for the reader: see how Diffie—Hellman's *non-interactive key exchange* property would have allowed us to do this more efficiently
  (See OPTLS by Krawczyk and Wee)

```
   Client                                  Server
 ClientHello         -------->
                     <--------         ServerHello
                                             <...>
                              <CertificateRequest>
                     <--------       <Certificate>
 <KemEncapsulation>  -------->
                     <--------          <Finished>
 <Certificate>       -------->
                     <--------  <KemEncapsulation>
 <Finished>          -------->

 [Application Data]  <------->  [Application Data]

<msg>: enc. w/ keys derived from ephemeral KEX (HS)
[msg]: enc. w/ keys derived from HS (MS)
```

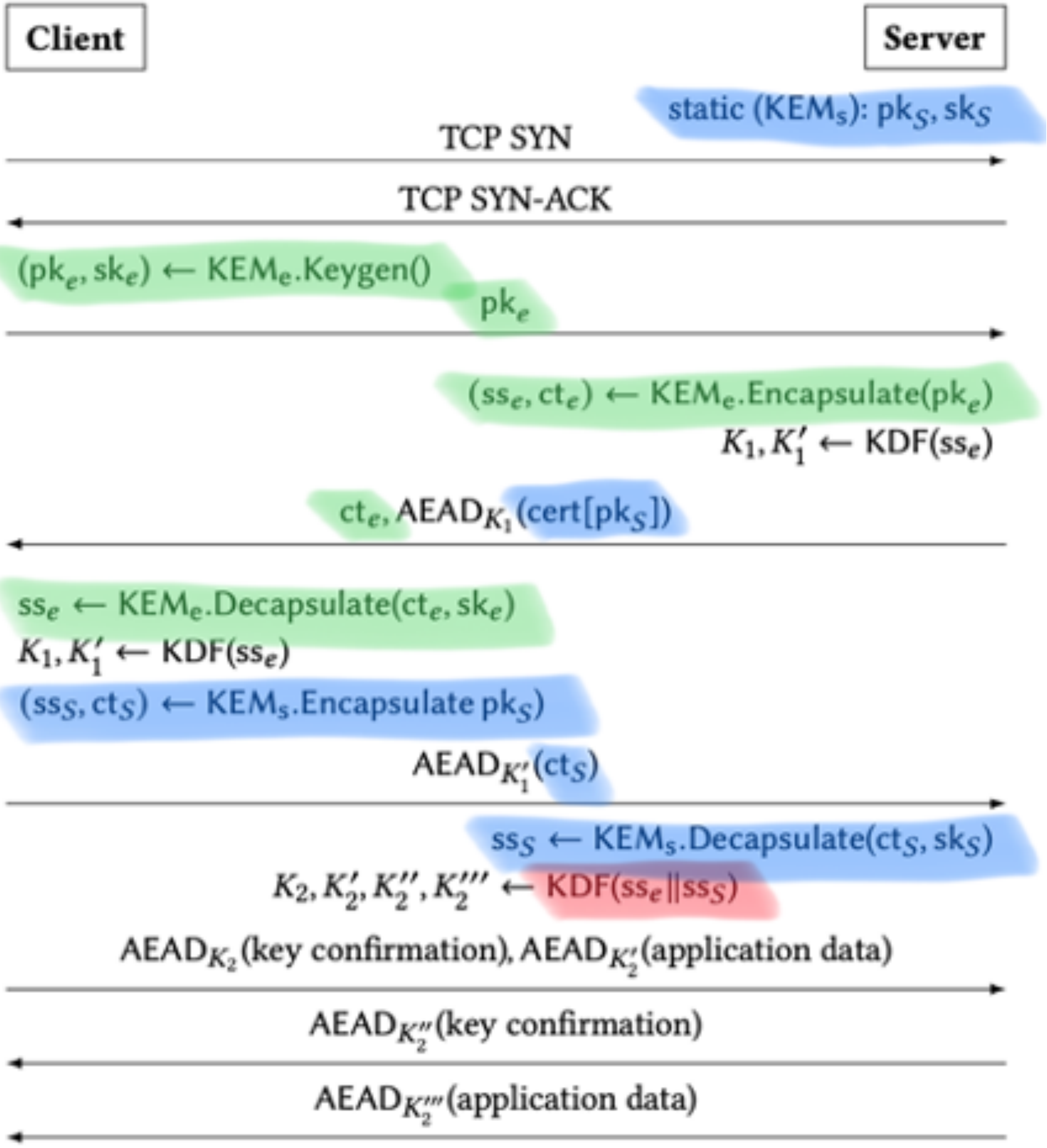
KEMTLS

ACM CCS 2020

KEM for
ephemeral key exchange

KEM for
server-to-client
authenticated key exchange

Combine shared secrets



Client

Server

static (KEM_s): $\mathsf{pk}_S, \mathsf{sk}_S$

TCP SYN

TCP SYN-ACK

$(\mathsf{pk}_e, \mathsf{sk}_e) \leftarrow \mathsf{KEM}_e.\mathsf{Keygen}()$

pk_e

$(\mathsf{ss}_e, \mathsf{ct}_e) \leftarrow \mathsf{KEM}_e.\mathsf{Encapsulate}(\mathsf{pk}_e)$

$K_1, K_1' \leftarrow \mathsf{KDF}(\mathsf{ss}_e)$

$\mathsf{ct}_e, \mathsf{AEAD}_{K_1}(\mathsf{cert}[\mathsf{pk}_S])$

$\mathsf{ss}_e \leftarrow \mathsf{KEM}_e.\mathsf{Decapsulate}(\mathsf{ct}_e, \mathsf{sk}_e)$

$K_1, K_1' \leftarrow \mathsf{KDF}(\mathsf{ss}_e)$

$(\mathsf{ss}_S, \mathsf{ct}_S) \leftarrow \mathsf{KEM}_s.\mathsf{Encapsulate}\ \mathsf{pk}_S)$

$\mathsf{AEAD}_{K_1'}(\mathsf{ct}_S)$

$\mathsf{ss}_S \leftarrow \mathsf{KEM}_s.\mathsf{Decapsulate}(\mathsf{ct}_S, \mathsf{sk}_S)$

$K_2, K_2', K_2'', K_2''' \leftarrow \mathsf{KDF}(\mathsf{ss}_e \| \mathsf{ss}_S)$

$\mathsf{AEAD}_{K_2}(\text{key confirmation}), \mathsf{AEAD}_{K_2'}(\text{application data})$

$\mathsf{AEAD}_{K_2''}(\text{key confirmation})$

$\mathsf{AEAD}_{K_2'''}(\text{application data})$

KEMTLS

The clever bit

- What can a server send to a client, before the client has said what they wanted?
- Use *implicitly authenticated* key to encrypt application message (request) to server *before* receiving Server's Finished message
- Avoid 2-RTT protocol
- Client can send HTTP request in same place as in TLS 1.3

KEMTLS

Sizes of instantiations

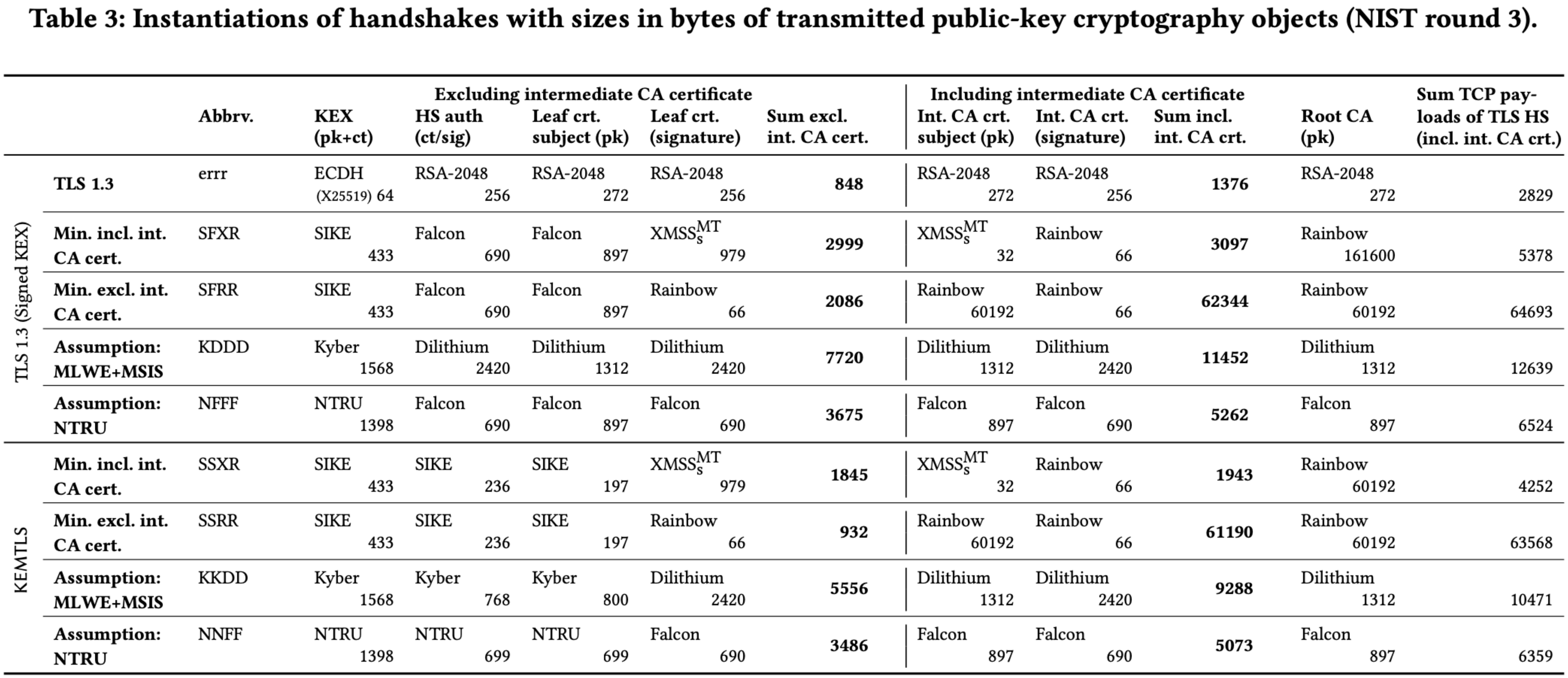


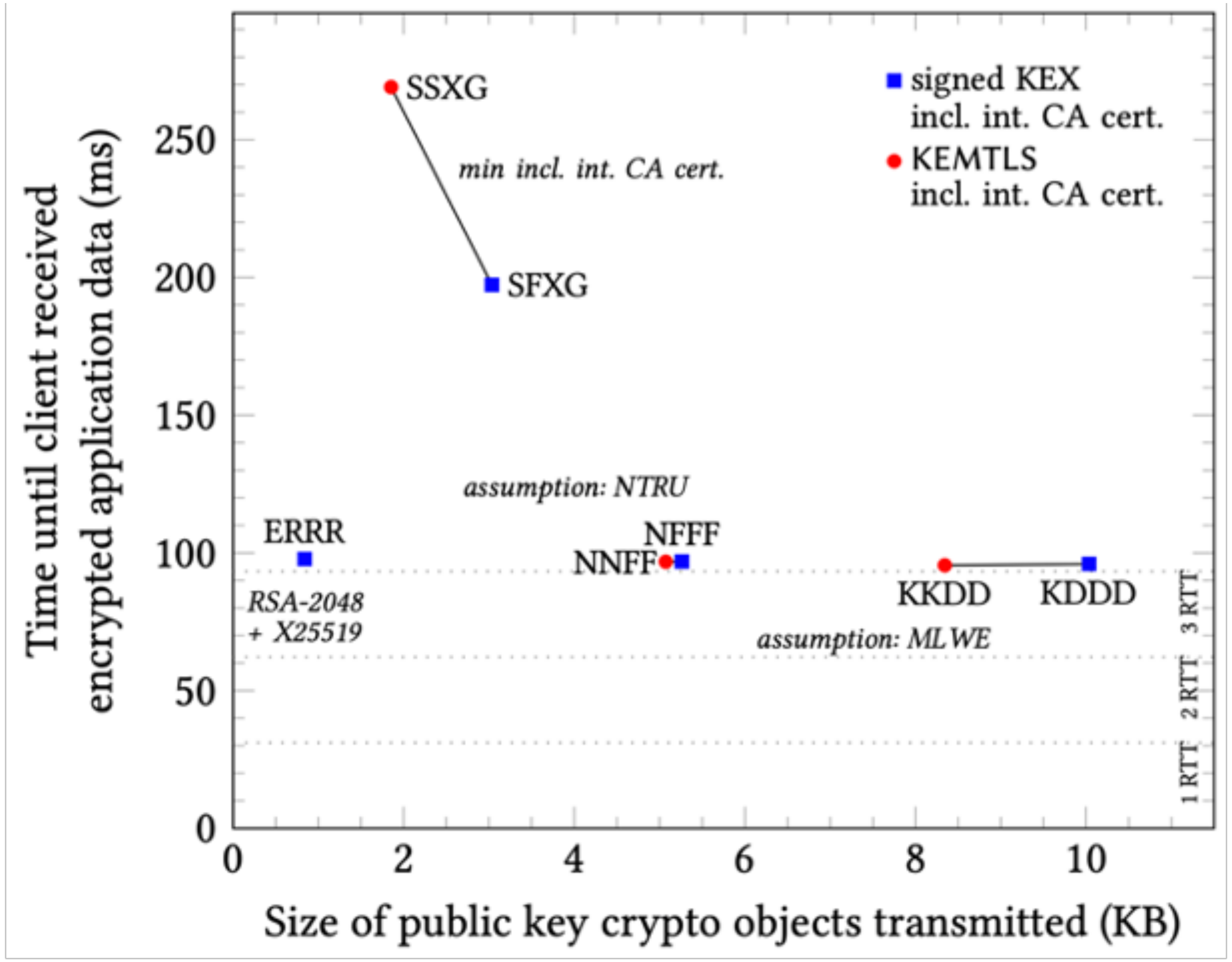
Table 3: Instantiations of handshakes with sizes in bytes of transmitted public-key cryptography objects (NIST round 3).

			Excluding intermediate CA certificate					Including intermediate CA certificate				
		Abbrv.	KEX (pk+ct)	HS auth (ct/sig)	Leaf crt. subject (pk)	Leaf crt. (signature)	Sum excl. int. CA cert.	Int. CA crt. subject (pk)	Int. CA crt. (signature)	Sum incl. int. CA crt.	Root CA (pk)	Sum TCP payloads of TLS HS (incl. int. CA crt.)
	TLS 1.3	errr	ECDH (X25519) 64	RSA-2048 256	RSA-2048 272	RSA-2048 256	**848**	RSA-2048 272	RSA-2048 256	**1376**	RSA-2048 272	2829
TLS 1.3 (Signed KEX)	**Min. incl. int. CA cert.**	SFXR	SIKE 433	Falcon 690	Falcon 897	$\text{XMSS}_s^{\text{MT}}$ 979	**2999**	$\text{XMSS}_s^{\text{MT}}$ 32	Rainbow 66	**3097**	Rainbow 161600	5378
	Min. excl. int. CA cert.	SFRR	SIKE 433	Falcon 690	Falcon 897	Rainbow 66	**2086**	Rainbow 60192	Rainbow 66	**62344**	Rainbow 60192	64693
	Assumption: MLWE+MSIS	KDDD	Kyber 1568	Dilithium 2420	Dilithium 1312	Dilithium 2420	**7720**	Dilithium 1312	Dilithium 2420	**11452**	Dilithium 1312	12639
	Assumption: NTRU	NFFF	NTRU 1398	Falcon 690	Falcon 897	Falcon 690	**3675**	Falcon 897	Falcon 690	**5262**	Falcon 897	6524
KEMTLS	**Min. incl. int. CA cert.**	SSXR	SIKE 433	SIKE 236	SIKE 197	$\text{XMSS}_s^{\text{MT}}$ 979	**1845**	$\text{XMSS}_s^{\text{MT}}$ 32	Rainbow 66	**1943**	Rainbow 60192	4252
	Min. excl. int. CA cert.	SSRR	SIKE 433	SIKE 236	SIKE 197	Rainbow 66	**932**	Rainbow 60192	Rainbow 66	**61190**	Rainbow 60192	63568
	Assumption: MLWE+MSIS	KKDD	Kyber 1568	Kyber 768	Kyber 800	Dilithium 2420	**5556**	Dilithium 1312	Dilithium 2420	**9288**	Dilithium 1312	10471
	Assumption: NTRU	NNFF	NTRU 1398	NTRU 699	NTRU 699	Falcon 690	**3486**	Falcon 897	Falcon 690	**5073**	Falcon 897	6359

Signed KEX versus KEMTLS

Labels ABCD:
A = ephemeral KEM
B = leaf certificate
C = intermediate CA
D = root CA

Algorithms: (all level 1)
Dilithium,
ECDH X25519,
Falcon,
GeMSS,
Kyber,
NTRU,
RSA-2048,
SIKE,
XMSS'



Rustls client/server with some AVX2 implementations. Emulated network: latency 31.1 ms, bandwidth 1000 Mbps, 0% packet loss. Average of 100000 iterations.

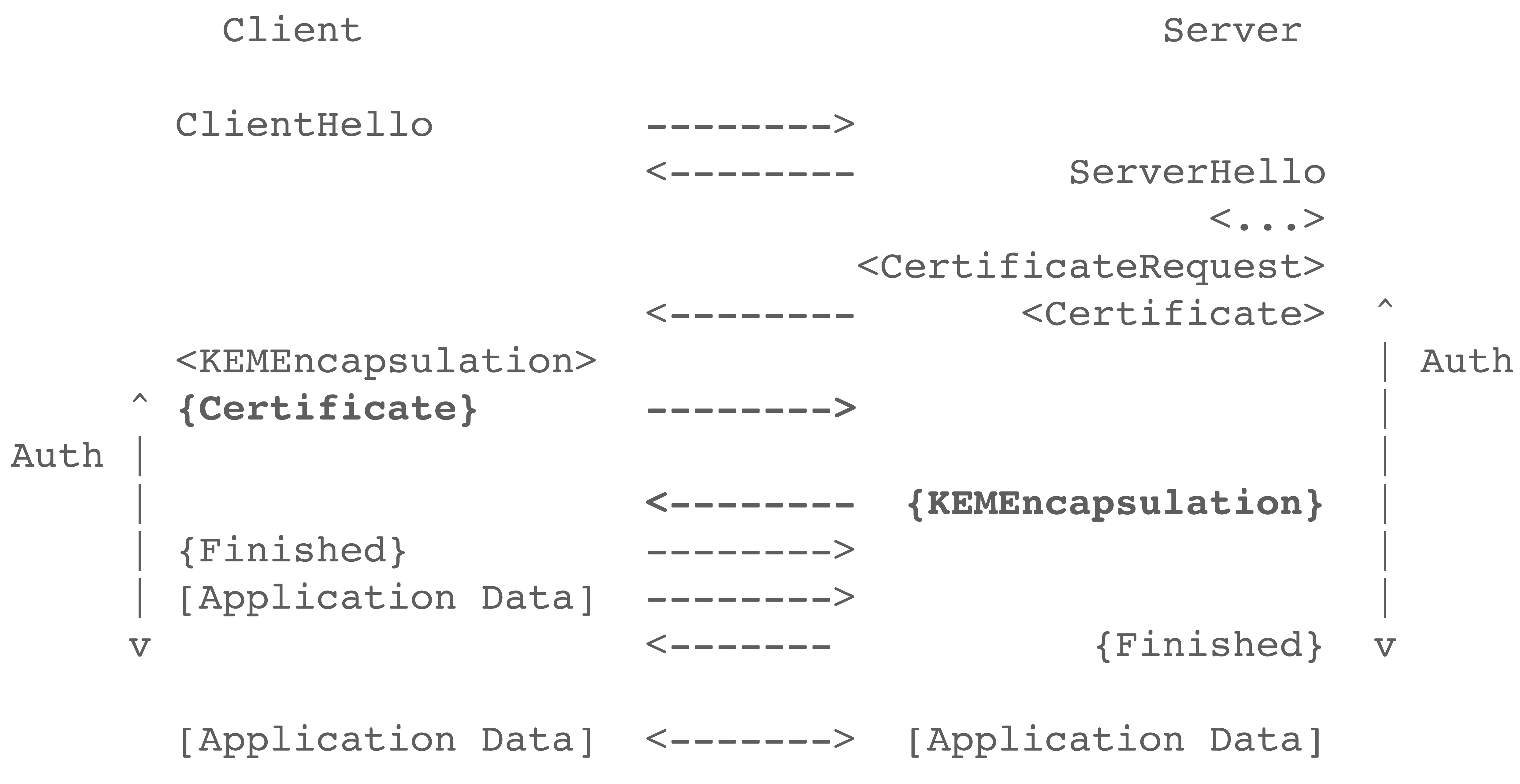
KEMTLS

Client Authentication

- Unfortunately, no nice tricks exist for the client certificate …

- **Full extra round-trip** in KEMTLS

- Also: we need an extra "authenticated" handshake traffic secret to protect the client certificate



```
           Client                                        Server

          ClientHello          --------->
                               <---------           ServerHello
                                                          <...>
                                          <CertificateRequest>
                               <---------         <Certificate>  ^
          <KEMEncapsulation>                                     | Auth
        ^ {Certificate}        --------->                        |
   Auth |                                                        |
        |                      <---------   {KEMEncapsulation}   |
        | {Finished}           --------->                        |
        | [Application Data]   --------->                        |
        v                      <--------             {Finished}  v

          [Application Data]   <-------->   [Application Data]
```

<msg>: enc. w/ keys derived from ephemeral KEX (HS)
{msg}: enc. w/ keys derived from HS+srv. KEM Auth (AHS)
[msg]: enc. w/ keys derived from AHS+cl. KEM Auth (MS)

KEMTLS-PDK

Pre-Distributed Keys

- The client often knows the server:
  - It’s the 10th time you refreshed the front page of Reddit in the past 5 minutes
  - You’ve been doom-scrolling /r/wallstreetbets 📉 for two hours already
  - Or the client is a too-cheap IoT security camera ~~spying on you for China~~ checking firmware updates from the same server every day

➡️ The client reasonably might know the server’s long-term key

KEMTLS-PDK

Pre-Distributed Keys

- Use server's long-term (certificate) public key to encaps *before ClientHello*
- Send the ciphertext *with ClientHello*
- Don't transmit certificates anymore
- Save even more bytes

```
    Client                                     Server

ct <- KEM.Encaps(skS)
 ClientHello
  + …
  + KemEncapsulation --------->
                     <---------           ServerHello
                                                <...>
                     <---------            <Finished>
                     <---------    [Application Data]
 <Finished>          --------->
 [Application Data]  <-------->    [Application Data]

<msg>: enc. w/ keys derived from KEX+srv. KEM auth (HS)
[msg]: enc. w/ traffic keys derived from HS (MS)
```

KEMTLS-PDK

Client Authentication

- We now have an implicitly authenticated key already *before we sent the ClientHello message!*

- Use this to also encrypt and send over the client’s certificate

- Or 0-RTT?

- ❗ No replay protection

- ❗ No forward secrecy

```
    Client                                      Server

  ClientHello
   + KemEncapsulation
   {Certificate}        --------->
                        <---------          ServerHello
                                                  <...>
                                    <KEMEncapsulation>
                        <---------           <Finished>
                        <---------   [Application Data]
  <Finished>            --------->
  [Application Data]    <-------->   [Application Data]

 {msg}: enc. w/ keys derived from srv. KEM auth (ES)
 <msg>: enc. w/ keys derived from KEX+srv. KEM auth (HS)
 [msg]: enc. w/ keys derived from HS+cl. KEM auth (MS)
```

KEMTLS

Benefits

- Size-optimized KEMTLS requires < ½ communication of size-optimized PQ signed-KEM
- Speed-optimized KEMTLS uses 90% fewer server CPU cycles and still reduces communication
- NTRU KEX (27 μs) 10x faster than Falcon signing (254 μs)
- No extra round trips required until client starts sending application data
- Smaller trusted code base (no signature generation on client/server)

TLS ecosystem challenges

So much going on…

- Datagram TLS
- Use of TLS handshake in other protocols
  - e.g. QUIC
- Application-specific behaviour
  - e.g. HTTP3 SETTINGS frame not server authenticated
- PKI involving KEM public keys
- Long tail of implementations
- …

KEMTLS

Standardizing?

- Authentication bits from KEMTLS have been submitted to the TLS working group at the Internet Engineering Task Force (IETF) (aka the RFC people)
  - https://datatracker.ietf.org/doc/draft-celi-wiggers-tls-authkem/
  - https://wggrs.nl/docs/authkem-abridged/

Transitioning to PQ

Wrap-up

- The transition to post-quantum means:
  - KEMs are less flexible than Diffie—Hellman
    - No non-interactive key exchange
  - PQ is bigger than ECC we got used to
  - Post-Quantum Signatures are big
- KEMTLS really explores **new tradeoffs**
  - Signing and key exchange are not the same operations anymore
  - Transitioning to PQ is an opportunity to reconsider some established protocols!

Thanks for your attention