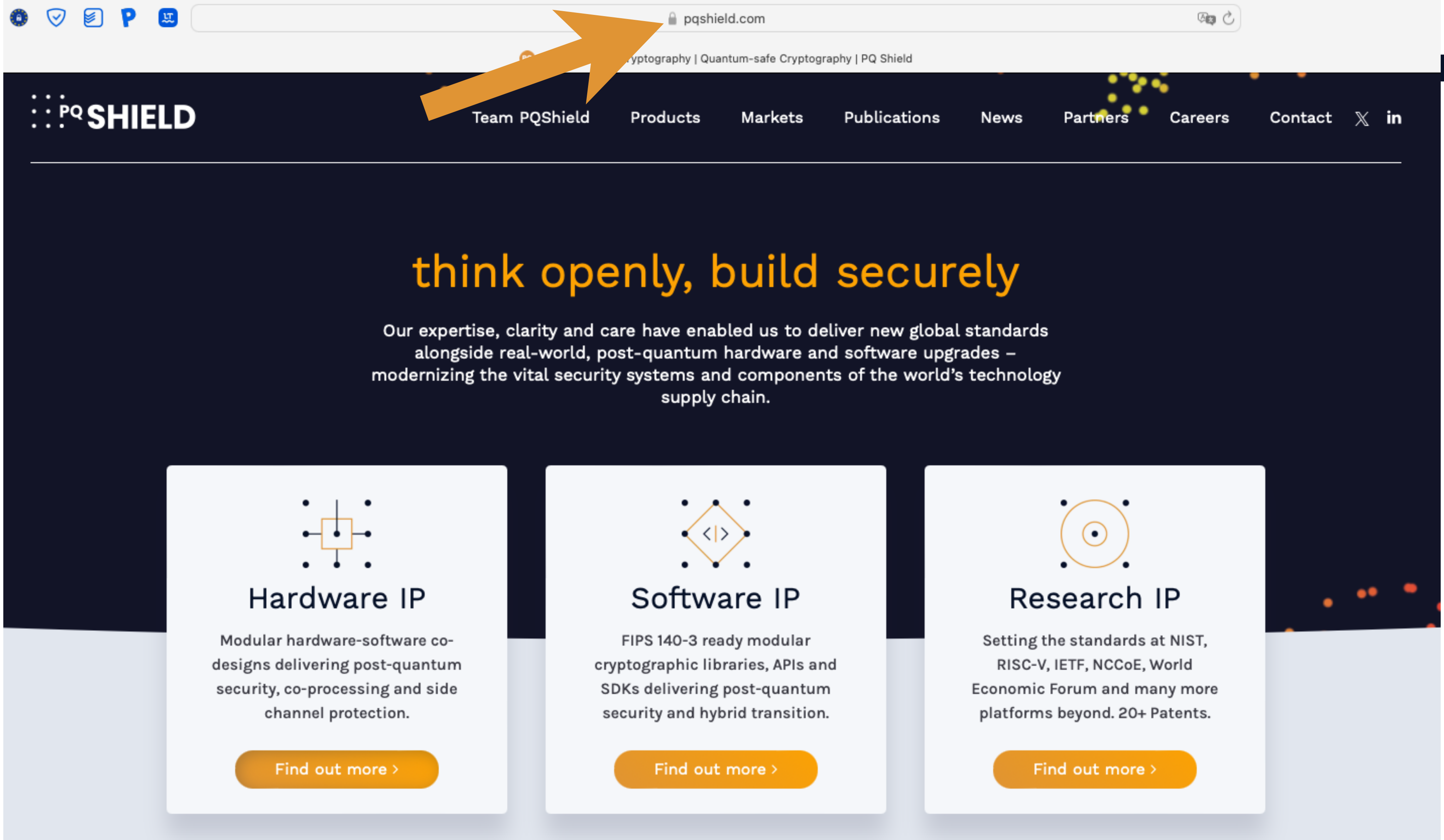
PQ TLS and WebPKI

(or: Are we PQ yet?)

Thom Wiggers

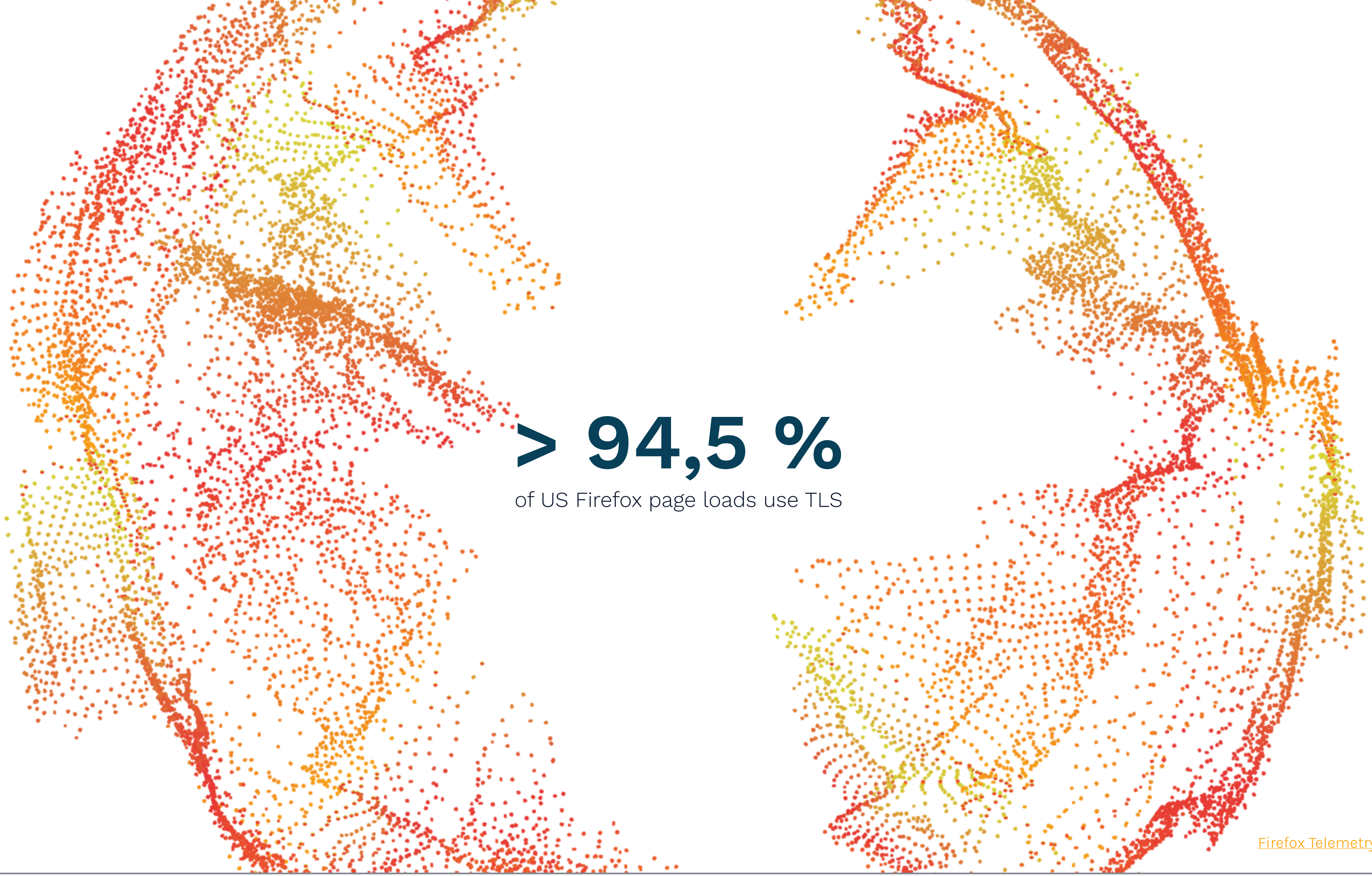
Thom Wiggers

- Cryptography researcher at PQShield
    - Oxford University spin-off
    - We develop and license PQC hardware and software IP
    - Side-channel protected hardware designs
    - FIPS 140-3 validated software
    - We also do fundamental research
- Research interest: applying PQC to real-world systems
    - Post-Quantum TLS
    - Secure messaging
- Ph.D from Radboud University (2024)
    - Dissertation: Post-Quantum TLS


pqshield.com
yptography | Quantum-safe Cryptography | PQ Shield
PQ SHIELD
Team PQShield
Products
Markets
Publications
News
Partners
Careers
Contact
in
think openly, build securely
Our expertise, clarity and care have enabled us to deliver new global standards alongside real-world, post-quantum hardware and software upgrades – modernizing the vital security systems and components of the world's technology supply chain.
Hardware IP
Modular hardware-software co-designs delivering post-quantum security, co-processing and side channel protection.
Find out more ›
Software IP
FIPS 140-3 ready modular cryptographic libraries, APIs and SDKs delivering post-quantum security and hybrid transition.
Find out more ›
Research IP
Setting the standards at NIST, RISC-V, IETF, NCCoE, World Economic Forum and many more platforms beyond. 20+ Patents.
Find out more ›

“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



> 94,5 %

of US Firefox page loads use TLS

Firefox Telemetry, 2024-04-23

TLS

Imagine it's 1997

1. You want to set up a web shop
2. You need to process credit card information
   a. If bad guys obtain a credit card number…
3. You are advertising in newspapers
   a. You can hardly distribute key material beforehand

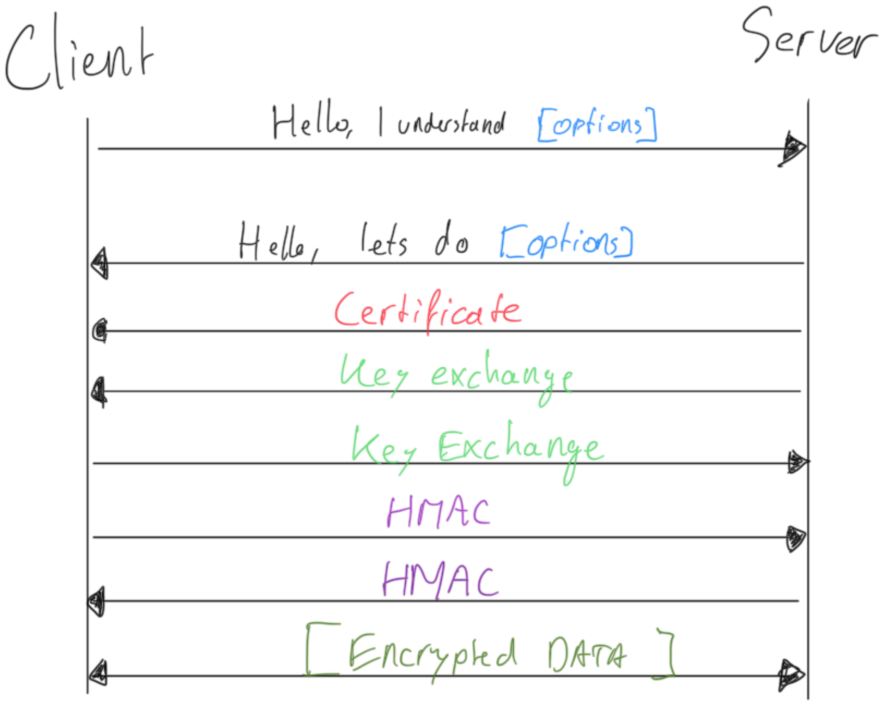
TLS Handshake Requirements

- ❑ Set up a shared secret key for encrypting application traffic
- ❑ Transmit the identity and key material during the protocol handshake
    - ❑ Don't require prior knowledge of the server
- ❑ Be secure

TLS Version 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



TLS 1.2 problems

- 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 on TLS (subset)

- 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
- 2023: **Everlasting ROBOT**: Bleichenbacher's 1998 attack is still, still valid on many TLS 1.2 implementations

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, …

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
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

PQ SHIELD

Ciphersuites in TLS

This isn't even all of them!

Room for improvement

- TLS 1.2 is not very robust against attacks
- TLS 1.2 leaks server and user identities in the handshake
- TLS 1.2 is not super efficient in 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: RFC 8446 (2018)

- Move key exchange into the first two messages
- Encrypt everything afterwards
- 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

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

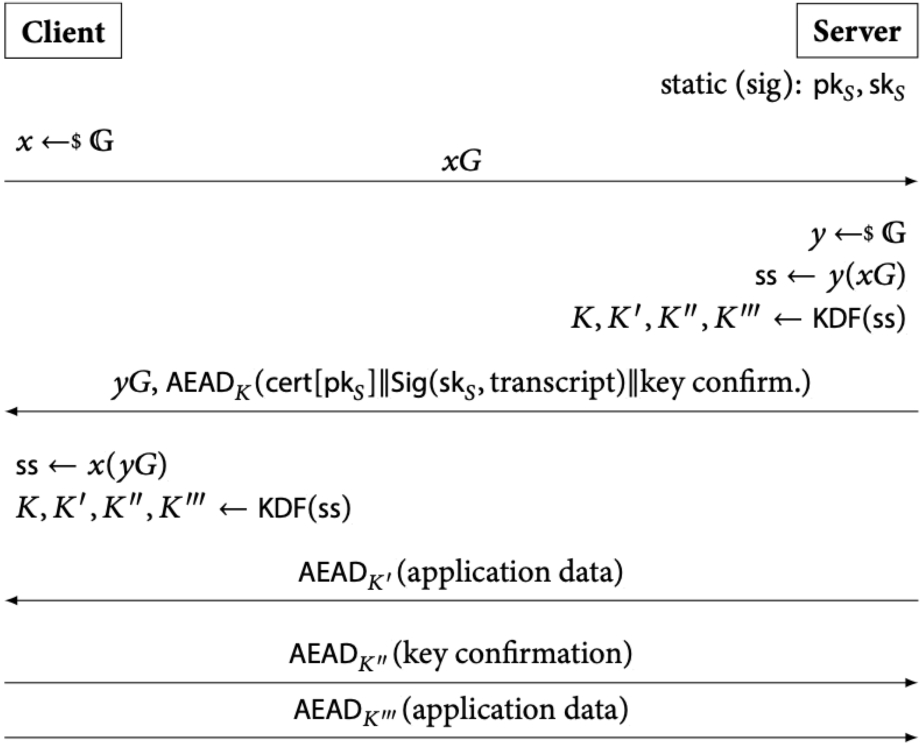


Figure 3.1: High-level overview of the TLS 1.3 handshake.

TLS 1.3 Resumption and 0-RTT

- 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
- No certificates
- 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

```
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.
```

RFC 8446 page 18

Why 0-RTT?

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

But are you sure your application is completely robust to replays?

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

TLS 1.3 standardization

- 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

Much less ad-hoc design: design-break-patch-release process instead of design-release-break-patch

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

❑ Post-quantum?

Server Name Indication: the remaining privacy problem

- TLS 1.3 encrypts the ServerCertificate and ClientCertificate messages
- But, Client includes the domain that they want to talk to in ClientHello in **plain text**!
- This allows CDNs / "virtual hosts" to serve many sites off of one IP address
- Problem: no keys established beforehand to encrypt the hostname
- Current proposed solution: put HPKE (RFC9180) keys in DNS so that server name can be encrypted (ECH: Encrypted Client Hello, WIP)
    - Post-Quantum challenges: DNS has significant size restrictions; adds additional ciphertext
- Only a real solution when many names map to the same IP (i.e. big-enough anonymity set implies CDN)
- ECH KEM keys are not useful for server (host) authentication (due to anonymity set)

Post-Quantum TLS


Peter Shor
ECC
g^x
RSA

SHIELD

An official website of the United States government Here's how you know

NIST

Search CSRC

CSRC MENU

Information Technology Laboratory

COMPUTER SECURITY RESOURCE CENTER

NIST COMPUTER SECURITY RESOURCE CENTER CSRC

PROJECTS

Post-Quantum Cryptography PQC

Overview

Public comments are available for Draft FIPS 203, Draft FIPS 204 and Draft FIPS 205, which specify algorithms derived from CRYSTALS-Dilithium, CRYSTALS-KYBER and SPHINCS+. The public comment period closed November 22, 2023.

PQC Seminars
Next Talk: April 23, 2024

4th Round KEMs

Additional Digital Signature Schemes - Round 1 Submissions

PQC License Summary & Excerpts

Background

NIST 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.**

In recent years, there has been a substantial amount of research on quantum computers – machines that exploit quantum

PROJECT LINKS

- Overview
- FAQs
- News & Updates
- Events
- Publications
- Presentations

ADDITIONAL PAGES

- Post-Quantum Cryptography Standardization
  - Call for Proposals
  - Example Files
  - Round 1 Submissions
  - Round 2 Submissions
  - Round 3 Submissions
    - *Round 3 Seminars*
- Round 4 Submissions
- Selected Algorithms 2022

TLS 1.3

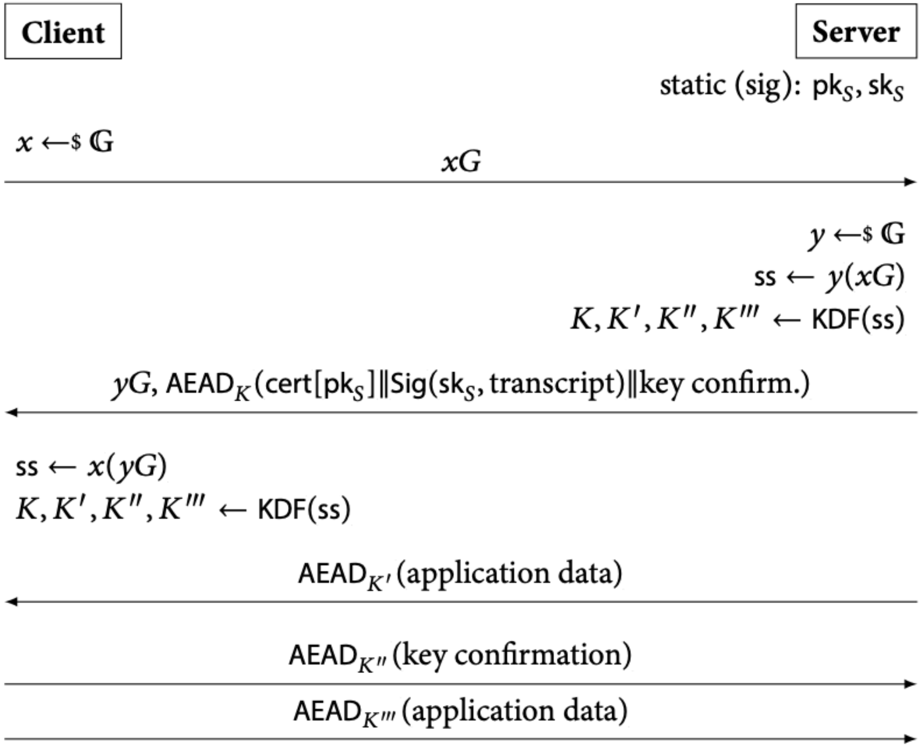


Figure 3.1: High-level overview of the TLS 1.3 handshake.

Post-Quantum

TLS 1.3

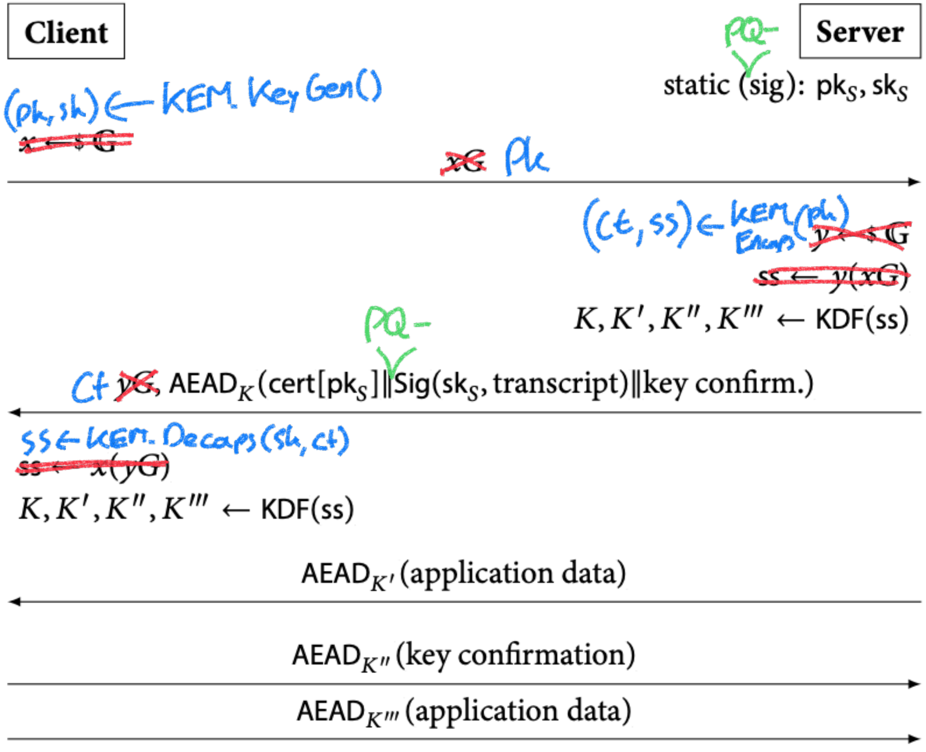


Figure 3.1: High-level overview of the TLS 1.3 handshake.

(AES-128 is fine btw)

Post-Quantum KEMs

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

	Public key	Ciphertext
ML-KEM 512	800 b	768 b
ML-KEM 768	1184 b	1088b
ML-KEM 1024	1568 b	1568 b

X25519: 64 bytes

Post-Quantum Signatures: NIST Standards

	Public key	Signature
ML-DSA 44	1312 b	2420 b
ML-DSA 65	1952 b	3309 b
ML-DSA 87	2592 b	4627 b

Formerly Dilithium

	Public key	Signature
Falcon-512	897 b	666 b
Falcon-1024	1793 b	1280 b

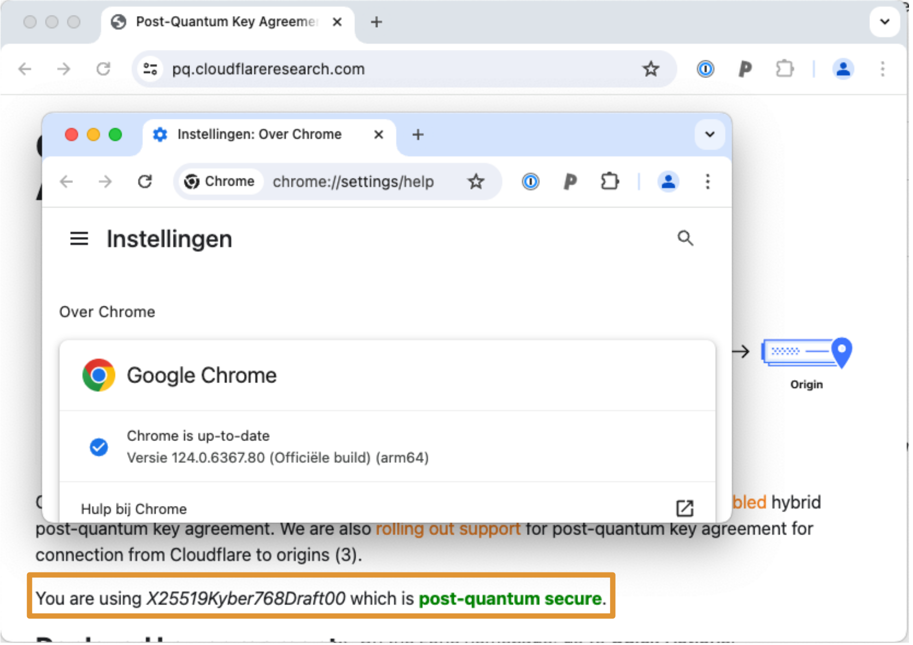
⚠️ Falcon *signing* uses 64-bit floats: side-channel issues

SLH-DSA	Public Key	Signature
128s	32 b	7856 b
128f	32 b	17088 b
192s	48 b	16224 b
192f	48 b	35664 b
256s	64 b	29792 b
256f	64 b	49856 b

Formerly known as SPHINCS+

RSA-2048: 544 bytes

By the way: Chrome 124.0



David Adrian Archief...omwiggers.nl 3 juni 2024 om 16:21

[TLS]Re: Curve-popularity data?

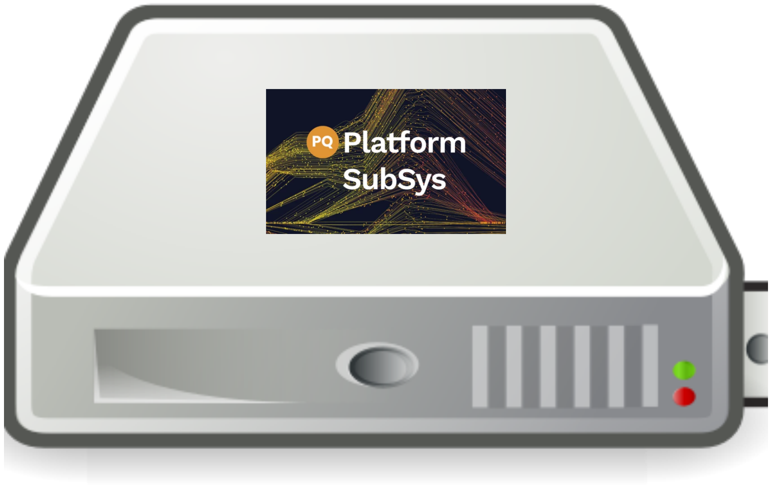
Aan: [redacted] Kopie: [redacted] <tls@ietf.org> <tls@ietf.org> Details

I don't really see why popularity of previous methods is relevant to picking what the necessarily new method will be is, but from the perspective of Chrome on Windows, across all ephemeral TCP TLS (1.2 and 1.3, excluding 1.2 RSA), the breakdown is roughly:

15% P256
3% P384
56% X25519
26% X25519+Kyber

Authentication

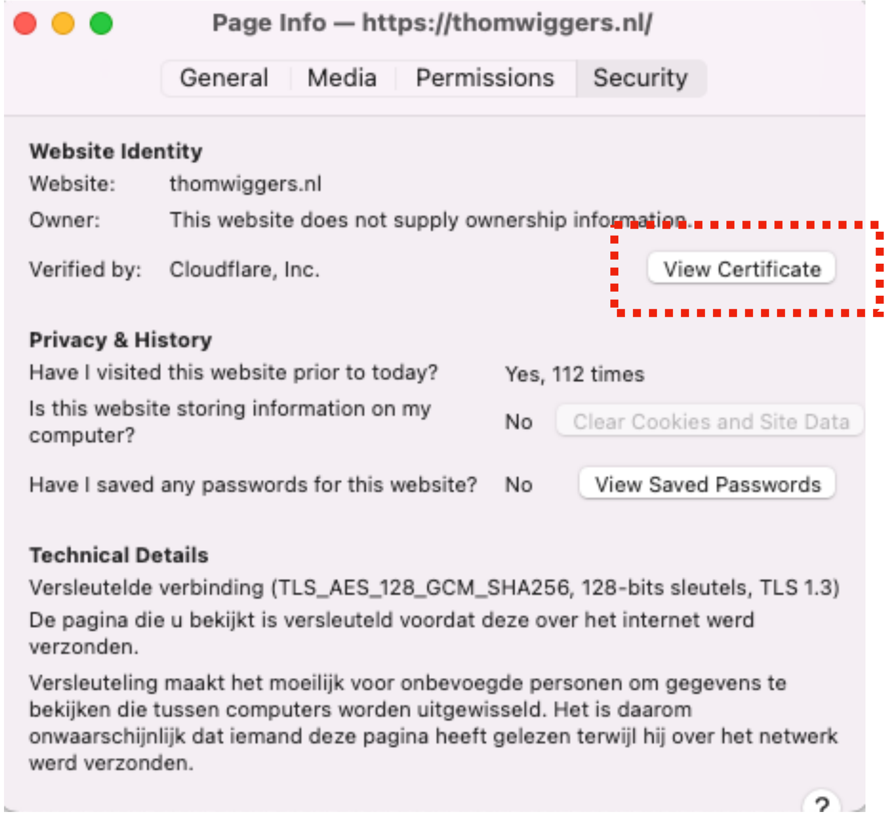
1. CAs just buy new hardware
2. We buy new certificates
3. ???
4. Profit?



https://commons.wikimedia.org/wiki/File:Server.svg

Muhammad Mahdi Karim/www.micro2macro.net
GFDL 1.2 Source

WebPKI


Page Info — https://thomwiggers.nl/
General
Media
Permissions
Security
Website Identity
Website: thomwiggers.nl
Owner: This website does not supply ownership information.
Verified by: Cloudflare, Inc.
View Certificate
Privacy & History
Have I visited this website prior to today? Yes, 112 times
Is this website storing information on my computer? No
Clear Cookies and Site Data
Have I saved any passwords for this website? No
View Saved Passwords
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.
?

3 Certificates

Certificate

sni.cloudflaressl.com | Cloudflare Inc ECC CA-3 | Baltimore CyberTrust Root

Pre-installed

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

handshake signature
+ leaf certificate public key + intermediate certificate signature
+ root signature on intermediate
= 3 signatures and 2 public keys

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

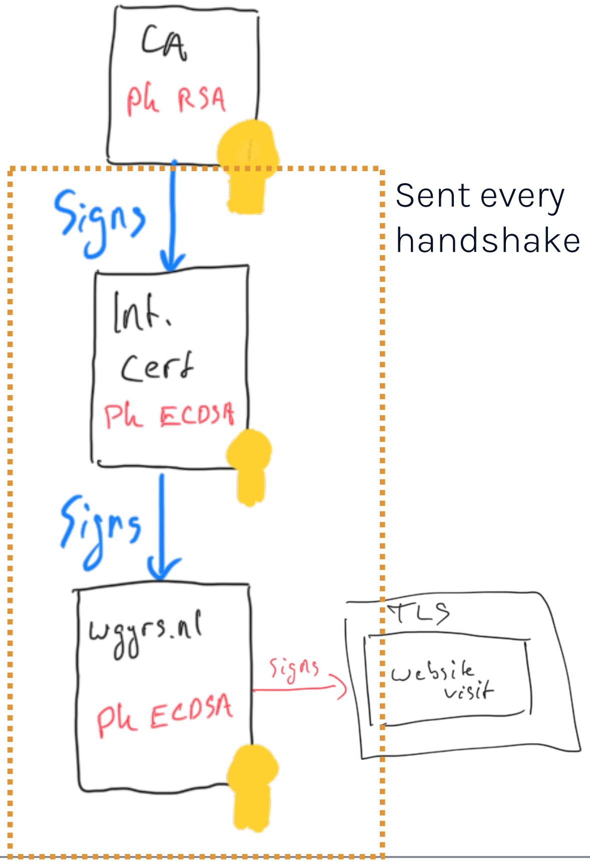
Purposes	Digital Signature

Extended Key Usages

Purposes	Server Authentication, Client Authentication

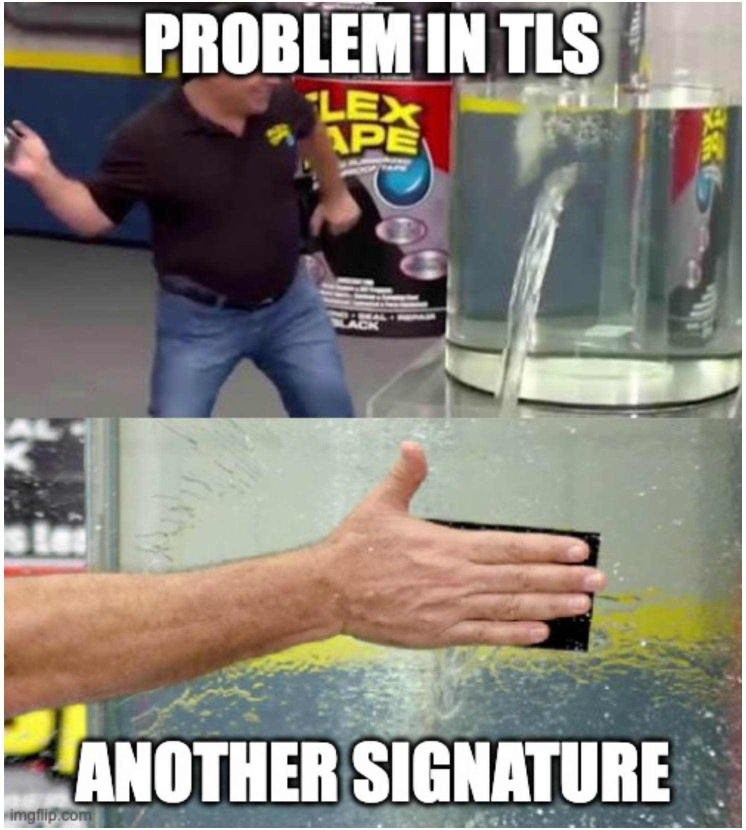
Public Key Infrastructure

- 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

- 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 Transparency” (CT)


PROBLEM IN TLS
ANOTHER SIGNATURE
imgflip.com

Slap another signature on it

PQ SHIELD

Online Certificate Status Protocol

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

+= 1 signature

Certificate Transparency

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

+= 3 signatures

Certificate Transparency

- Chrome, Safari require all certificates to be submitted to at least 2 certificate transparency logs
- Log is a Merkle tree of hostnames and hashes of included certificates
  - No privacy! You can search this using https://crt.sh
- Auditing, etc, are part of the design

- SCT proofs in certificates are promises of inclusion within 24 hours for deployment reasons
- CT logs typically only accept certificates from trusted issuers

Summarising

- Typical **web** TLS handshake:
  - ephemeral key exchange
  - handshake signature
  - leaf certificate:
    pk
    + signature by intermediate CA crt
    + OCSP staple
    + 3x SCT
  - intermediate CA certificate:
    pk + signature by root CA
  - root certificate (preinstalled)

1 online keygen+key exchange

1 online signing operation

6 offline signatures

PQ Performance

Impact of PQ

- ~~Kyber~~ML-KEM key exchange: ~1.5kB
- ML-DSA-44: 18 kB of certificates!!
- Falcon-512: ~5 kB

Note: TCP congestion control

On connection establishment, TCP will allow you to send some amount of data before acknowledgement from the other side.

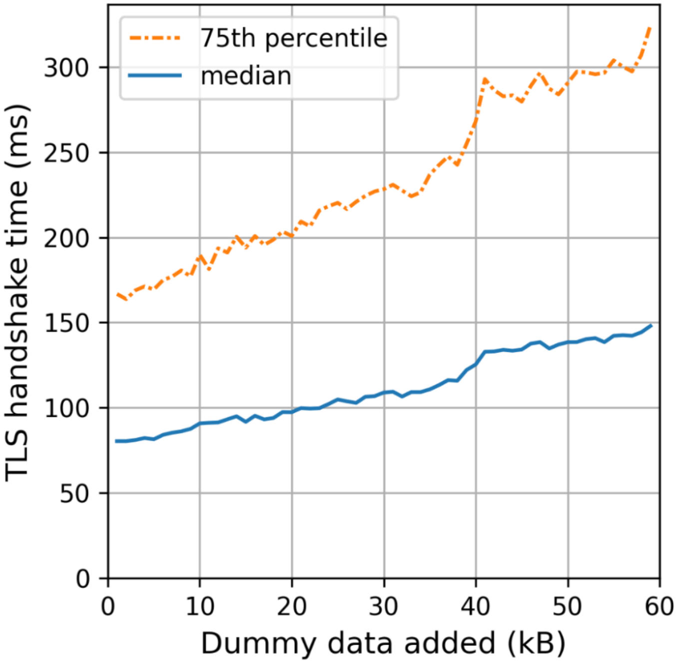
This window (and thus available connection bandwidth) scales as the connection is proven reliable when receiving TCP ACKs.

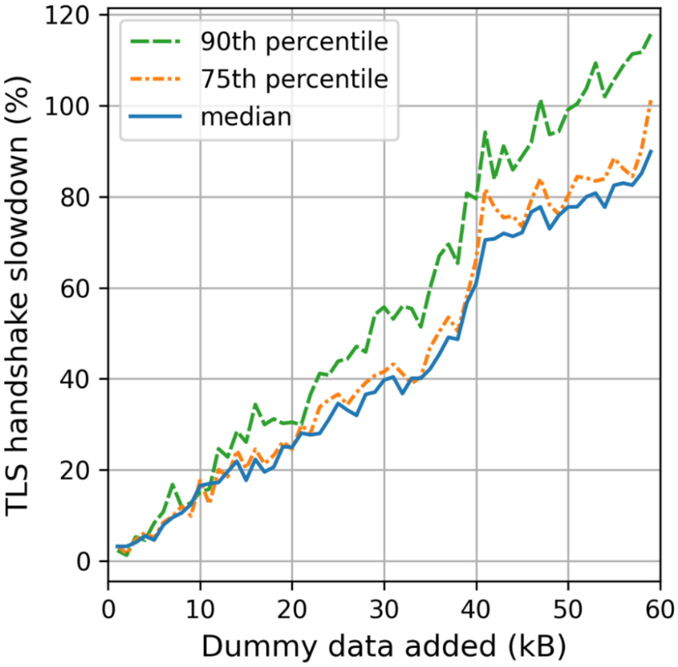
The default initial window on Linux is 10 packets, so **if you send more than ~15 kB of data, you're stuck waiting for an extra round-trip!**

Even without congestion control, more bytes = more slowlier

sizes per https://blog.cloudflare.com/sizing-up-post-quantum-signatures/

Cloudflare live internet experiment: More data results in slowdown





Bas Westerbaan, https://blog.cloudflare.com/sizing-up-post-quantum-signatures/. **Cloudflare has a 30 MSS = ~40kb congestion window**

Table 11.1: Instantiations at NIST level I of unilaterally authenticated post-quantum TLS handshakes and the sizes of the public-key cryptography elements in bytes.

		Leaf certificate			Int. CA certificate			Offline
Experiment handle	**Key Exchange** pk+ct	**Handshake auth.** pk+sig	**Int. CA signature** sig	**Sum**	**Int. CA public key** pk	**Root CA signature** sig	**Sum**	**Root CA public key** pk
Pre-quantum errr	X25519 64	RSA-2048 528	RSA-2048 256	848	RSA-2048 272	RSA-2048 256	1 376	RSA-2048 272
Primary KDDD	Kyber-512 1568	Dilithium2 3732	Dilithium2 2420	7 720	Dilithium2 1312	Dilithium2 2420	11 452	Dilithium2 1312
Falcon KFFF	Kyber-512 1568	Falcon-512 1563	Falcon-512 666	3 797	Falcon-512 897	Falcon-512 666	5 360	Falcon-512 897
Falcon offline KDFF	Kyber-512 1568	Dilithium2 3732	Falcon-512 666	5 966	Falcon-512 897	Falcon-512 666	7 529	Falcon-512 897

Src: Post-Quantum TLS, Thom Wiggers. PhD thesis. https://wggrs.nl/p/thesis/

Excludes OCSP and SCTs!

Severe performance impact

- Kyber-768 “only” adds 2.3 kB to the handshake
- Google notes this already slows down handshakes by 4%
- Google observes a significant impact on lower-quality internet connections
    - This is why they’re only enabling this on Chrome Desktop right now

- To stay under 10% slowdown, we seem to have a budget of at most 10kB including KEX
    - We need something better than just replacing signatures

https://dadrian.io/blog/posts/pqc-signatures-2024/

https://blog.chromium.org/2024/05/advancing-our-amazing-bet-on-asymmetric.html

https://securitycryptographywhatever.com/2024/05/25/ekr/

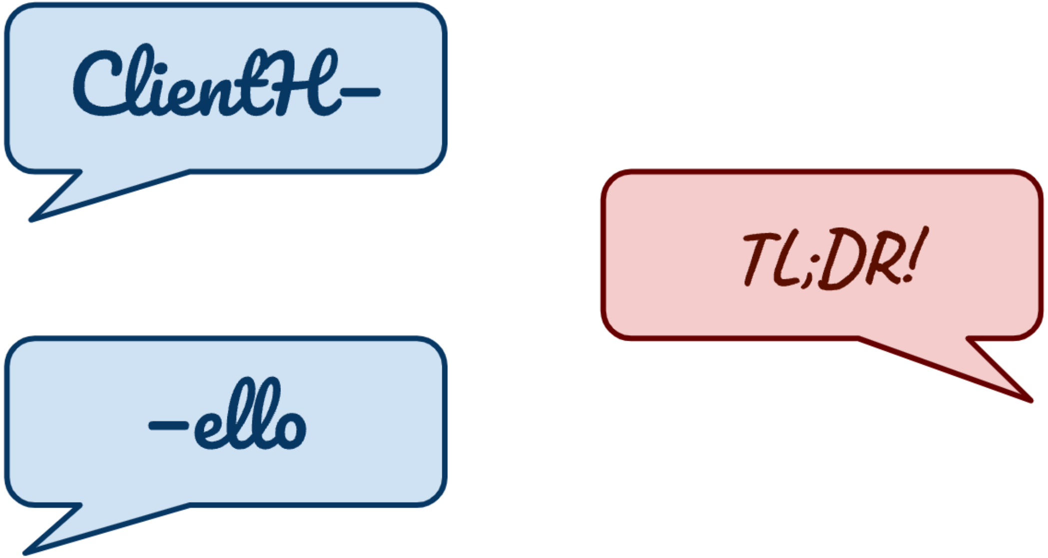
Not just speed

- Larger Hello messages can lead to fragmentation
- Not all implementations are prepared to deal with fragmented packets
- Especially middle boxes affected

Product	Status	Discovered	Via	Patched	Links
Vercel	✅	2023-08-15	Chrome Beta	2023-08-23	Twitter
ZScalar	✅	2023-08-17	Chrome Beta	2023-09-28	
Cisco		2024-04-23	Chrome 124	Unknown	Cisco Bug
Envoy	✅	2024-04-29	Chrome 124	n/a (config-only)	Github

Table last updated 2024-05-13

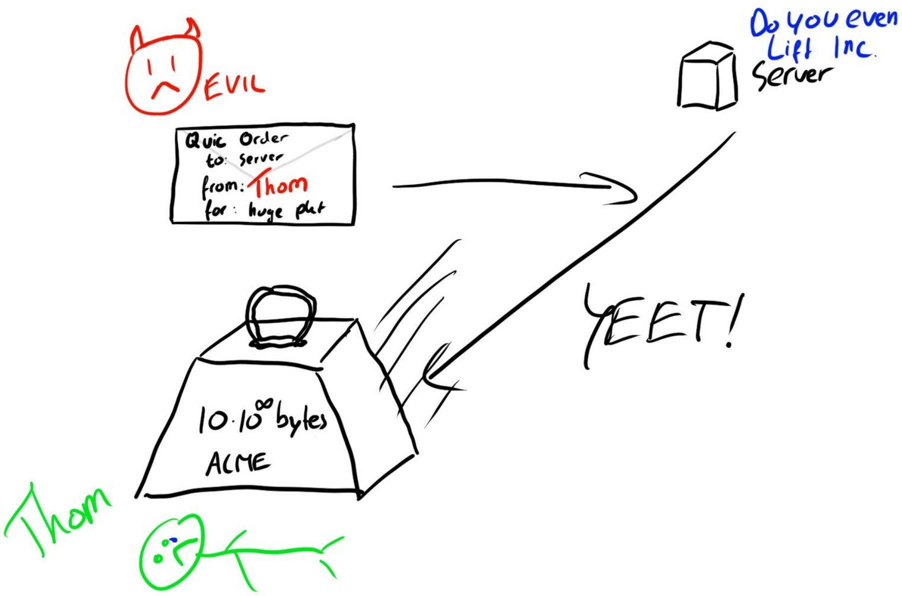
(List not exhaustive)



https://tldr.fail

More problems with sizes

- Variant protocols DTLS and QUIC are based on UDP: no TCP SYN/ACK sequence
- ClientHello message received by server could be spoofed, so QUIC allows sending back at most 3x the ClientHello size (avoids DoS amplification)
- Sending back 18kB of ML-DSA requires the client to pad its ClientHello message with ~5kB



Avoiding the costs of certificates

- Certificates are already very large, PQ makes this much worse
- We have multiple signatures that prove validity in each certificate:
  - Signature on certificate itself
  - OCSP staple that proves that certificate is currently valid
  - Certificate Transparency log inclusion proves that certificate was from a trusted issuer

Can we do things in a smarter way?

New WebPKI?

Combining different algorithms

- handshake signature
- leaf certificate:
  pk
  + signature by intermediate CA crt
  + OCSP staple
  + 3x SCT
- intermediate CA certificate:
  pk
  + signature by root CA
- root certificate (preinstalled)

Robust against side-channels, pk+sig small, fast signing

ML-DSA

Signature-verification only, pk+sig small

Falcon

Signature-verification only, signature small

UOV? (NIST additional call for signatures)

Note: using multiple algorithms also has cost!

Avoiding the costs of certificates

- Certificates are very large, PQ makes this much worse
- We have multiple signatures that prove validity in each certificate:
    - Signature on certificate itself
    - OCSP staple that proves that certificate is currently valid
    - Certificate Transparency log inclusion proves that certificate was from a trusted issuer

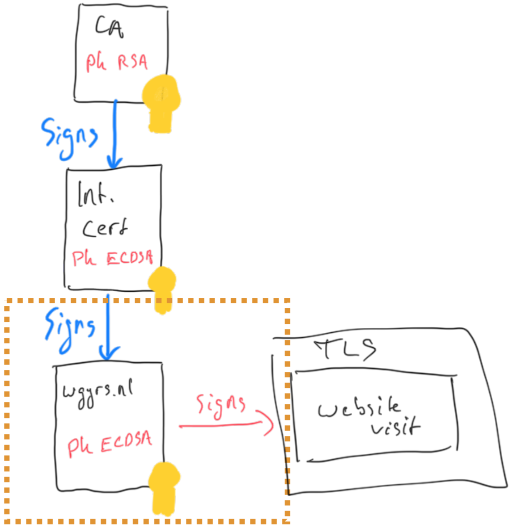
Can we do things in a smarter way?

Now is the time for redesigning the PKI

Abridged Compression for WebPKI Certificates

- Browser vendors control the root certificates that are included
- Step 1: Just ship the intermediate certificates as well
  - Client indicates to the server it has version N of the intermediate certificates list
  - Server omits intermediate certificate if present in list version N
  - Immediate savings: 1 certificate including 1 public key + 1 signature

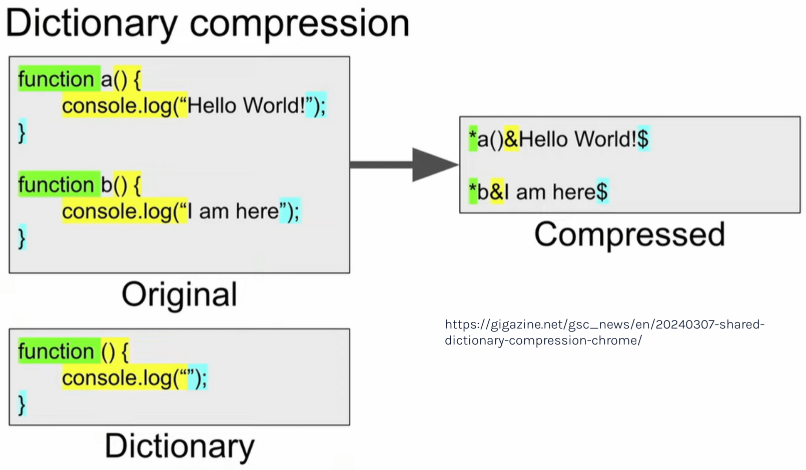
Sent every handshake



Dennis Jackson, Mozilla
https://datatracker.ietf.org/doc/draft-ietf-tls-cert-abridge/

Abridged Compression for WebPKI Certificates

- Certificates contain many common strings
  - policy urls, CA names, CT urls, extensions …
  - RFC 8879 already specifies certificate compression using zlib, brotli, zstd
- Step 2: Instead of applying compression algorithm directly, pre-train a compression dictionary based on sample certificates from all issuers
- Ship compression dictionary in browser



https://gigazine.net/gsc_news/en/20240307-shared-dictionary-compression-chrome/

https://datatracker.ietf.org/doc/draft-ietf-tls-cert-abridge/

Abridged Certificate Compression for TLS

- Step 3: compress certificates before sending using the pre-trained dictionary (if client up-to-date)

- Shipping compression dictionary out-of-band massively improves compression results
- Gain ~3000 bytes, i.e. space for 1 ML-DSA

- Remember that public keys and signatures themselves don't compress at all
- Security analysis very easy: just uncompress and you have the same TLS handshake

Scheme	Storage Footprint	p5	p50	p95
Original	0	2308	4032	5609
TLS Cert Compression	0	1619	3243	3821
Intermediate Suppression and TLS Cert Compression	0	1020	1445	3303
This Draft	65336	661	1060	1437
This Draft with opaque trained dictionary	3000	562	931	1454
Hypothetical Optimal Compression	0	377	742	1075

https://datatracker.ietf.org/doc/draft-ietf-tls-cert-abridge/

Merkle Tree Certificates

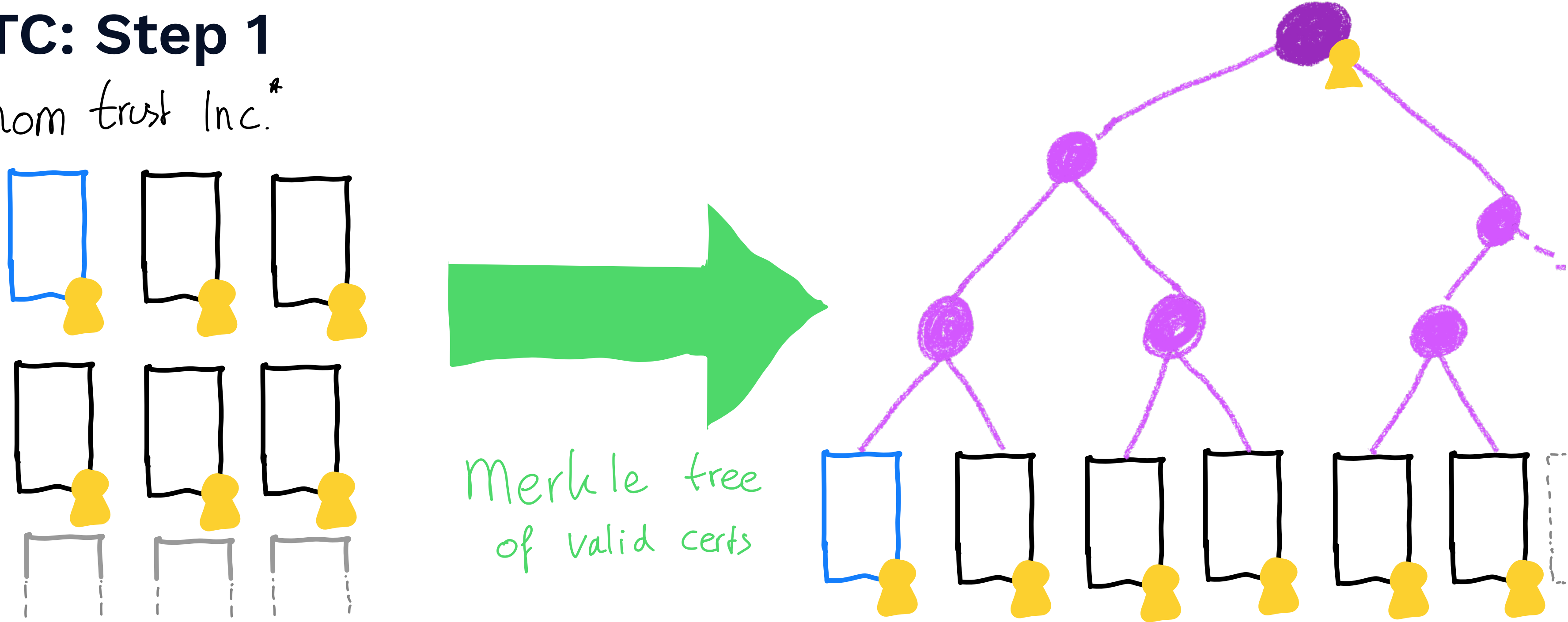
What if we build the PKI on Certificate Transparency's ideas, combined with OCSP?

```
Transport Layer Security                                   D. Benjamin
Internet-Draft                                               D. O'Brien
Intended status: Experimental                                Google LLC
Expires: 5 September 2024                               B. E. Westerbaan
                                                             Cloudflare
                                                           4 March 2024

                    Merkle Tree Certificates for TLS
                draft-davidben-tls-merkle-tree-certs-02
```

https://datatracker.ietf.org/doc/draft-davidben-tls-merkle-tree-certs/

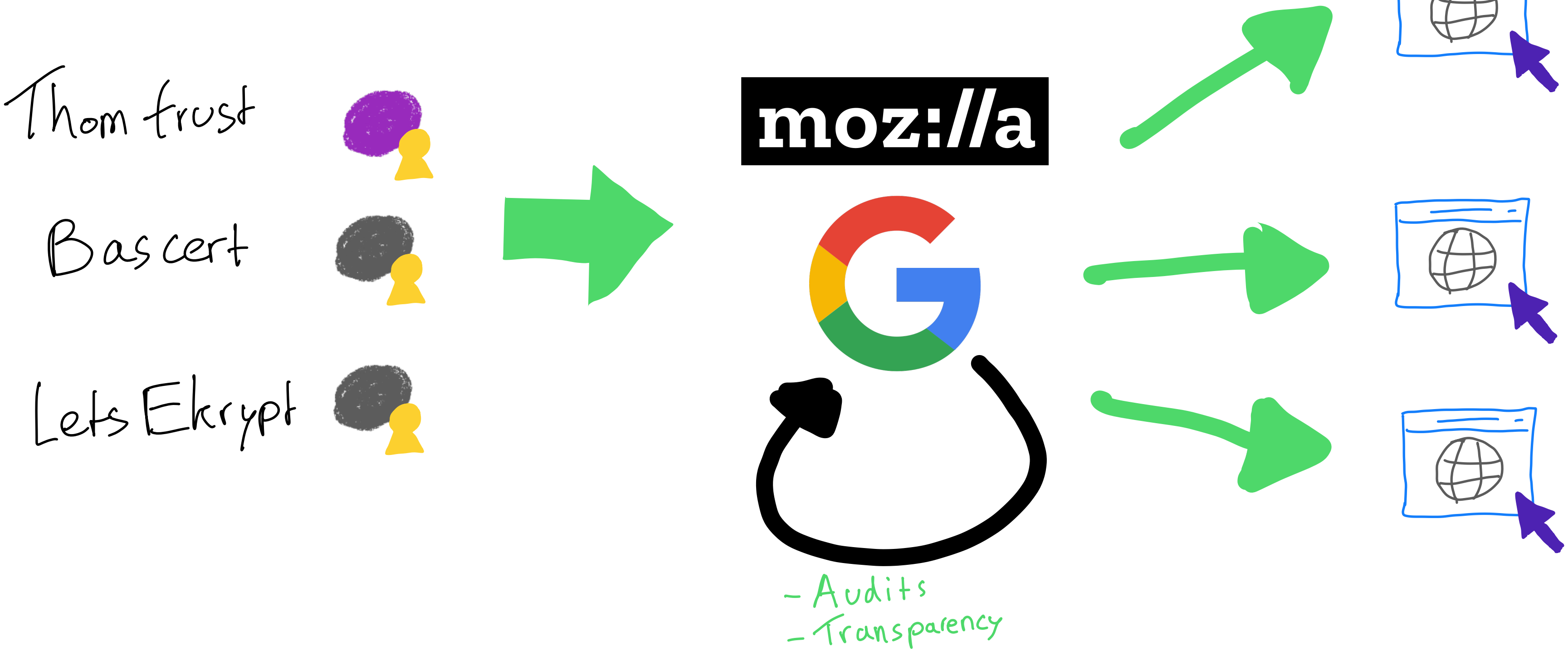
MTC: Step 1

Thom trust Inc.*

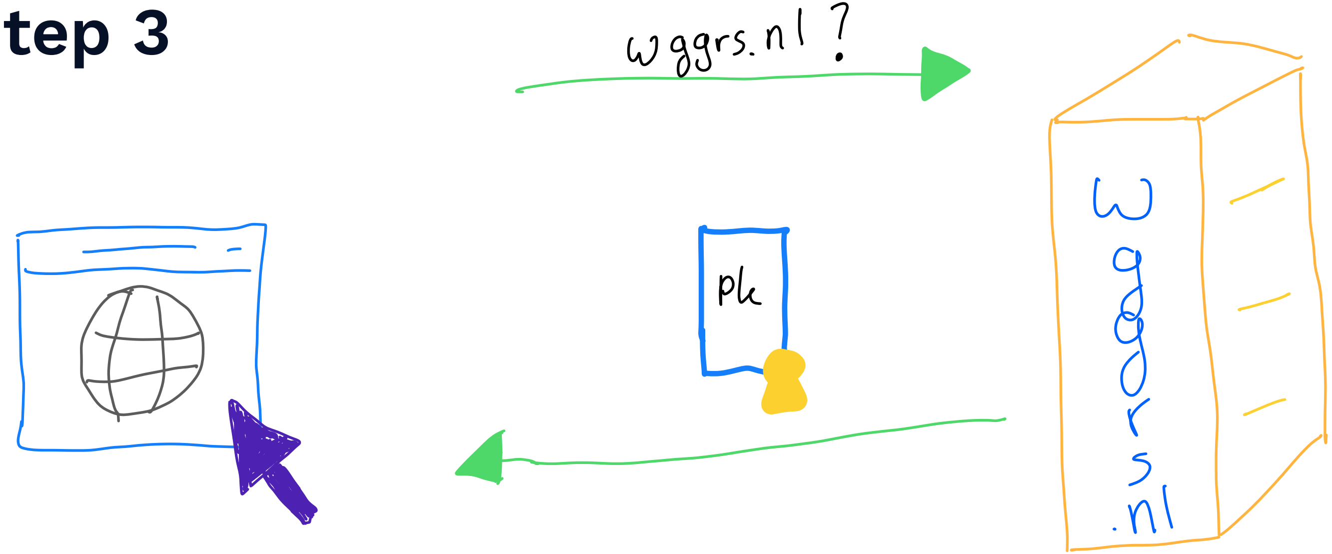


* Fund my Startup

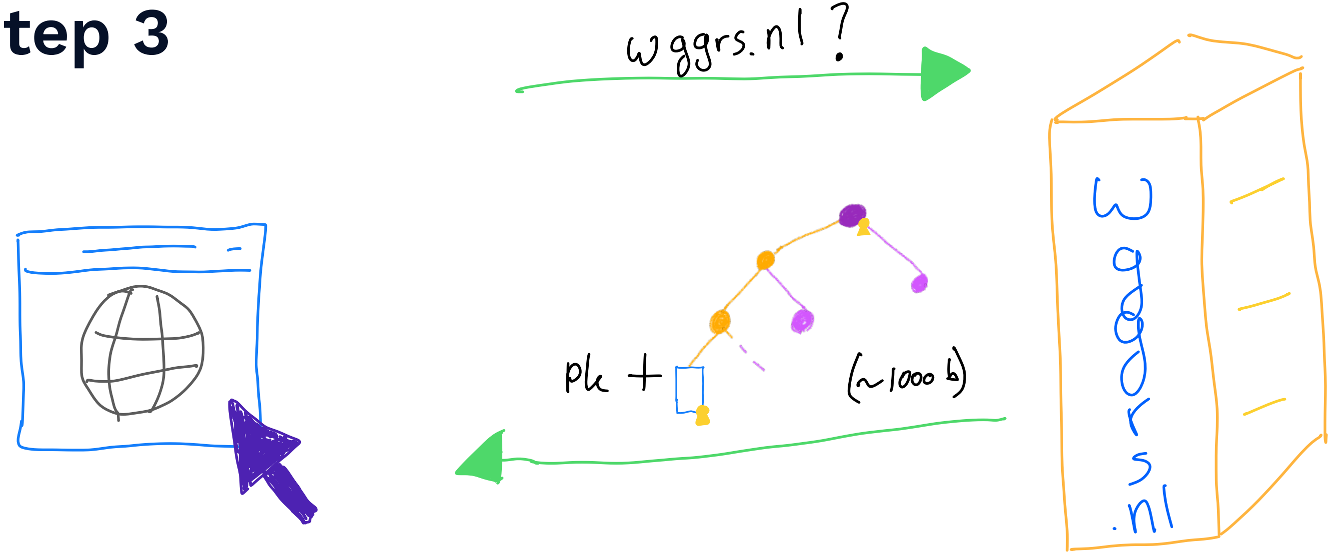
MTC: Step 2



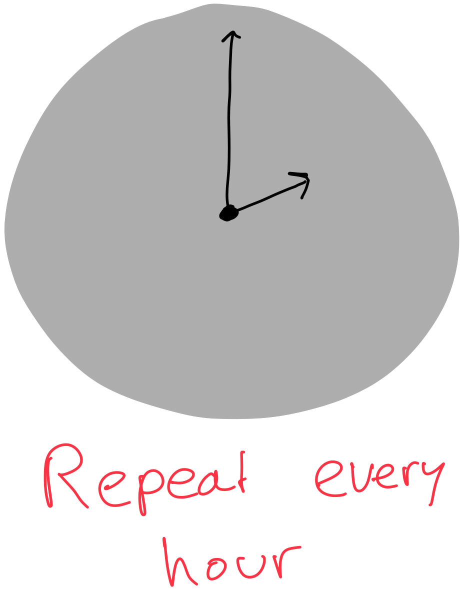
MTC: Step 3



MTC: Step 3



Merkle Tree Certificates



Merkle Tree Certificates

- Big changes necessary to every part of the ecosystem
  - Short-lived certificates
  - Webserver must continuously fetch the latest authentication paths
  - Clients must keep downloading currently valid tree heads
  - Automated certificate provisioning such as ACME [RFC8555] should help with this
- New trust model makes security analysis more complicated

- Both MTC and Abridged Compression designed for big deployments and publicly trusted CAs
  - What about IoT? What about a bank's internal stuff?

Save even more data?

- Handshake authentication still uses signatures, so ~3.5 kB (pk + sig) for Dilithium2
- KEMTLS: (implicitly) authenticate handshake by using key exchange instead
  - Put KEM public key in certificate / Merkle Tree Cert
  - Authentication in ~2 kB (ML-KEM 768)
  - Redesigns TLS handshake
  - IETF: draft-celi-wiggers-tls-authkem

- https://kemtls.org

```
Client                                            Server

ClientHello
+ key_share
+ signature_algorithms
                   -------->
                                             ServerHello
                                             + key_share
                                   <EncryptedExtensions>
                                       <KEM Certificate>  ^
                                    *no hs signature*     |
<KEMEncapsulation>  -------->                             |
{Finished}          -------->                             | Auth
[Application Data]  -------->                             |
                    <--------                  {Finished}  v

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


Figure 1: Message Flow for KEM-Authentication (KEM-Auth)
          Handshake without client authentication.
```

KEMTLS

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

- 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

Peter | **Douglas**

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

$\xrightarrow{\mathsf{ct}}$

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

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

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

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

PQ SHIELD

TLS authentication via KEM

- 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)

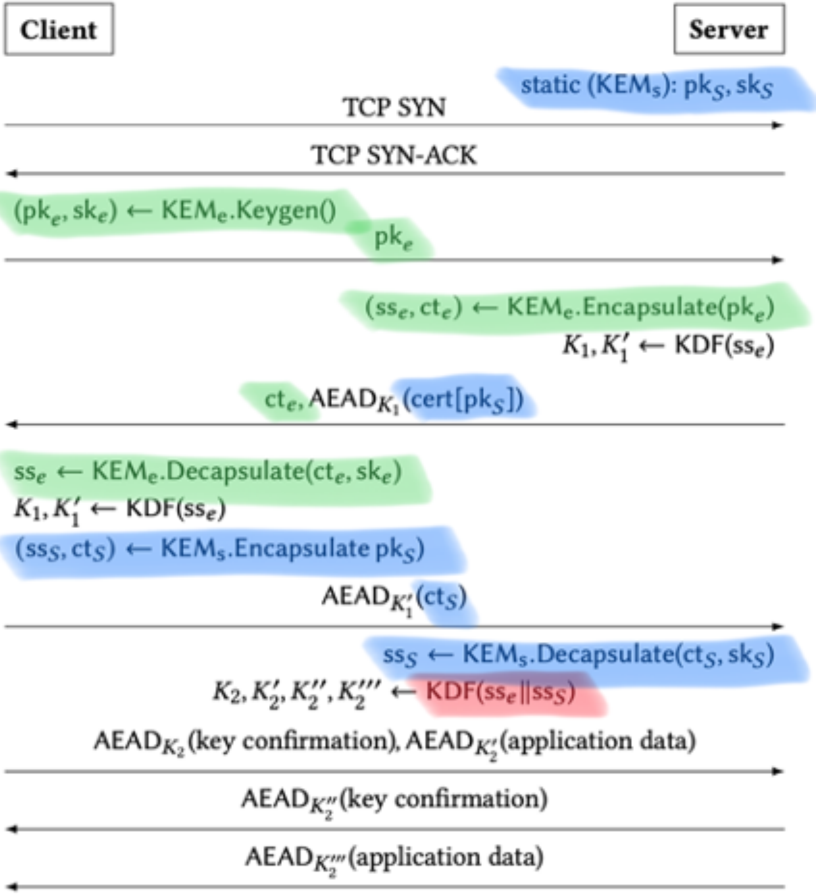
```
 Client                                           Server
ClientHello              --------->
                         <---------           ServerHello
                                                    <...>
                                    <CertificateRequest>
                         <---------         <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

KEM for
ephemeral key exchange

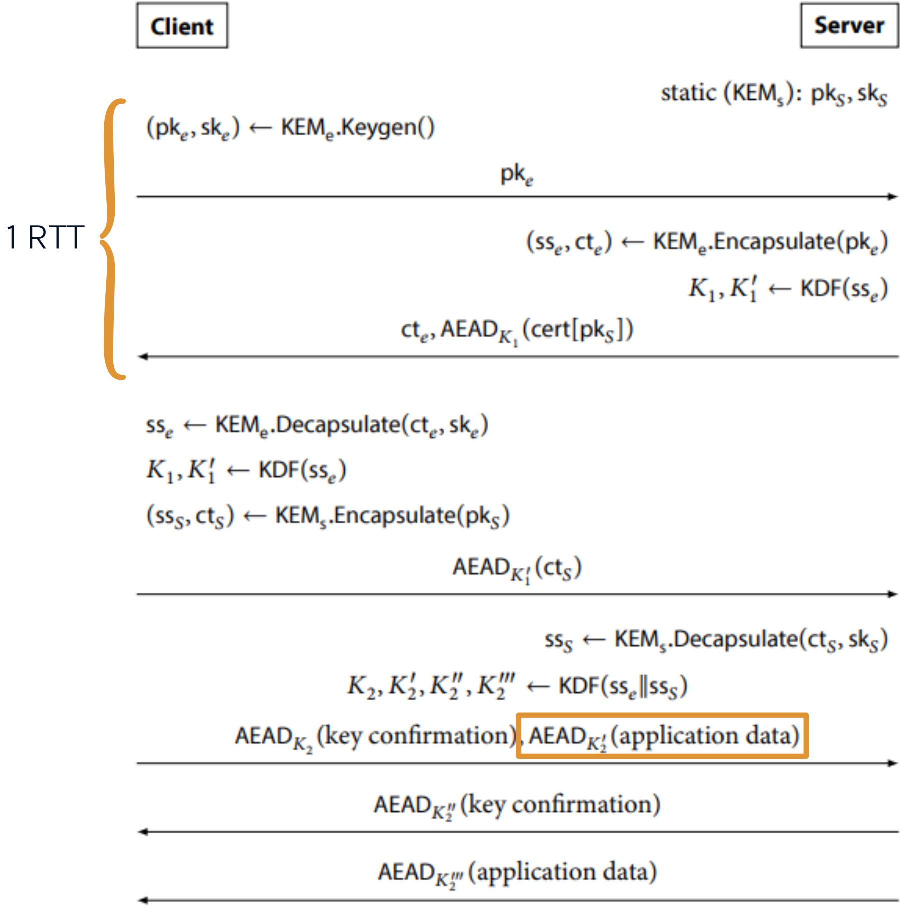
KEM for
server-to-client
authenticated key exchange

Combine shared secrets



KEMTLS

- 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



Sizes of KEMTLS

Table 13.1: Instantiations at NIST level I of unilaterally authenticated KEMTLS handshakes and the sizes of the public-key cryptography elements in bytes.

Experiment	Key Exchange	Leaf certificate: Handshake auth.	Leaf certificate: Int. CA signature	Sum	Int. CA certificate: Int. CA public key	Int. CA certificate: Root CA signature	Sum	Offline: Root CA public key
handle	pk+ct	pk+ct	sig		pk	sig		pk
Primary KKDD	Kyber-512 1568	Kyber-512 1568	Dilithium2 2420	5 556	Dilithium2 1312	Dilithium2 2420	9 288	Dilithium2 1312
Falcon KKFF	Kyber-512 1568	Kyber-512 1568	Falcon-512 666	3 802	Falcon-512 897	Falcon-512 666	5 365	Falcon-512 897
SPHINCS$^+$-f KKSfSf	Kyber-512 1568	Kyber-512 1568	SPHINCS$^+$-128f 17 088	20 224	SPHINCS$^+$-128f 32	SPHINCS$^+$-128f 17 088	37 344	SPHINCS$^+$-128f 32
SPHINCS$^+$-s KKSsSs	Kyber-512 1568	Kyber-512 1568	SPHINCS$^+$-128s 7856	10 992	SPHINCS$^+$-128s 32	SPHINCS$^+$-128s 7856	18 880	SPHINCS$^+$-128s 32
Hash-based CA KKXX	Kyber-512 1568	Kyber-512 1568	$XMSS_s^{MT}$-I 979	4 115	$XMSS_s^{MT}$-I 32	$XMSS_s^{MT}$-I 979	5 126	$XMSS_s^{MT}$-I 32

Table 13.5: Comparison of handshake size and time until the client receives a response from the server (30.9 ms, 1000 Mbps), between unilaterally authenticated post-quantum TLS 1.3 and KEMTLS instances at NIST level I.

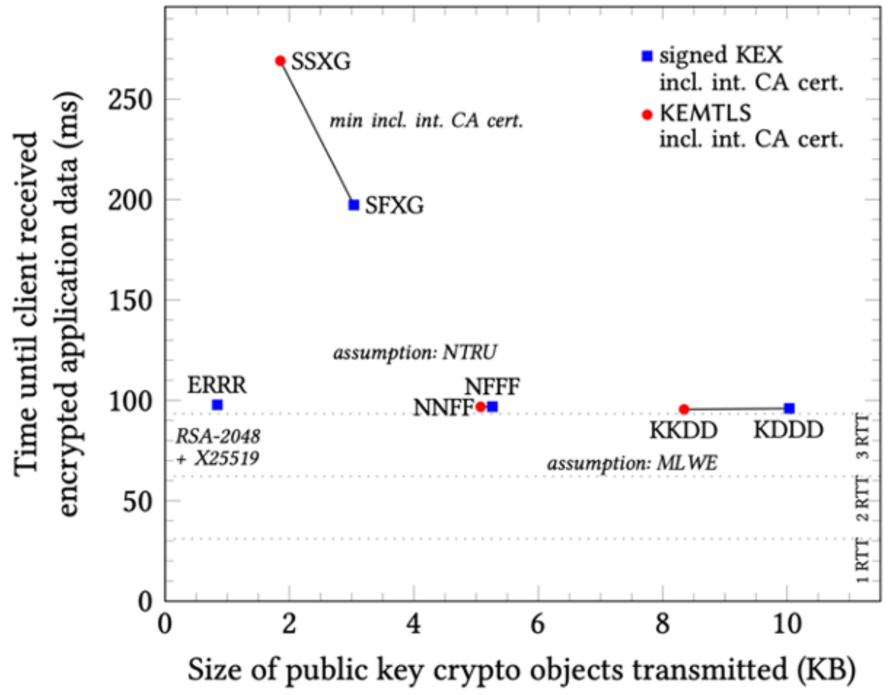
Experiment		Handshake size (bytes): No int.	Δ%	With int.	Δ%	Time until response (ms): No int.	Δ%	With int.	Δ%
TLS	KDDD	7720	−28.0 %	11 452	−18.9 %	94.8	−0.4 %	95.0	−0.3 %
KEMTLS	KKDD	5556		9288		94.4		94.8	
TLS	KFFF	3797	+0.1 %	5360	+0.1 %	95.8	−1.3 %	96.1	−1.2 %
KEMTLS	KKFF	3802		5365		94.5		94.9	
TLS	KDFF	5966	−36.3 %	7529	−28.7 %	94.8	−0.3 %	95.2	−0.3 %
KEMTLS	KKFF	3802		5365		94.5		94.9	
TLS	KSsSsSs	17 312	−36.5 %	25 200	−25.1 %	197.7	−52.0 %	198.0	−36.2 %
KEMTLS	KKSsSs	10 992		18 880		94.9		126.4	

Table excludes OCSP, SCT

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.

Slightly dated slide borrowed from Douglas Stebila

KEMTLS client auth

- 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
RTT                                                      <...>
                                            <CertificateRequest>
                              <--------           <Certificate>   ^
       <KEMEncapsulation>                                         | Auth
     ^ {Certificate}          -------->                           |
RTT  |                                                            |
     |                        <--------   {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

- 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

- 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(pkS)
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

- 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)
```

TLS ecosystem challenges

- 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
- ...

Standardizing KEMTLS

- 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://datatracker.ietf.org/doc/draft-wiggers-tls-authkem-psk/
    - https://wggrs.nl/docs/authkem-abridged/

Transitioning to PQ

- 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
- Big changes to surrounding ecosystems might be necessary
  - "Slapping another signature on it" is no longer a cheap solution
  - The WebPKI may see a big redesign
  - Even with the big redesign, we may still need **KEMTLS** (AuthKEM @ IETF) to mitigate the cost of the handshake signature to keep the slowdown under 10%