MAX PLANCK INSTITUTE FOR SECURITY AND PRIVACY

UNIVERSITY OF WATERLOO

Post-Quantum TLS without handshake signatures

Sofía Celi[1], Armando Faz Hernández[1], Peter Schwabe[2,4], Douglas Stebila[3], Thom Wiggers[4]

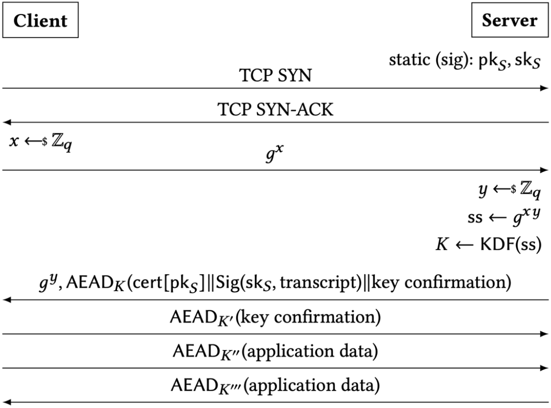
[1] Cloudflare [2] MPI Security and Privacy
[3] University of Waterloo [4] Radboud University

CLOUDFLARE

Radboud University

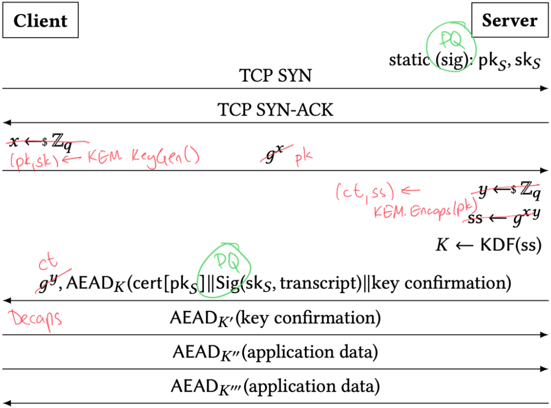
TLS 1.3

TLS 1.3 Handshake



- Key exchange: Diffie–Hellman
- Authentication: Signatures

Post-Quantum TLS 1.3 Handshake



- Key exchange: Post-Quantum Key-Encapsulation Mechanisms
- Authentication: Post-Quantum Signatures

Problem

Post-Quantum signatures are…

Problem

Post-Quantum signatures are…

- quite a bit bigger than KEMs
- quite a bit slower than KEMs
- quite a bit of extra code

Use PQ KEMs for authentication instead

KEM

Definition (Key Encapsulation Mechanism (KEM))

- $(\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{KEM.Keygen}()$
- $(\mathsf{ss}, \mathsf{ct}) \leftarrow \mathsf{KEM.Encapsulate}(\mathsf{pk})$
- $\mathsf{ss} \leftarrow \mathsf{KEM.Decapsulate}(\mathsf{ct}, \mathsf{sk})$

Example

To authenticate Douglas to Peter

Peter		**Douglas**
$(\mathsf{ss}, \mathsf{ct}) \leftarrow \mathsf{KEM.Encapsulate}(\mathsf{pk}_{\mathsf{Douglas}})$		
	$\xrightarrow{\mathsf{ct}}$	
		$\mathsf{ss} \leftarrow \mathsf{KEM.Decapsulate}(\mathsf{ct}, \mathsf{sk}_{\mathsf{Douglas}})$
$K \leftarrow \mathsf{KDF}(\mathsf{ss})$		$K \leftarrow \mathsf{KDF}(\mathsf{ss})$
	$\xleftarrow{\mathsf{MAC}_K(\cdots)}$	

KEM

Definition (Key Encapsulation Mechanism (KEM))

- $(\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{KEM.Keygen}()$
- $(\mathsf{ss}, \mathsf{ct}) \leftarrow \mathsf{KEM.Encapsulate}(\mathsf{pk})$
- $\mathsf{ss} \leftarrow \mathsf{KEM.Decapsulate}(\mathsf{ct}, \mathsf{sk})$

Example

To authenticate Douglas to Peter

Peter		**Douglas**
$(\mathsf{ss}, \mathsf{ct}) \leftarrow \mathsf{KEM.Encapsulate}(\mathsf{pk}_{\mathsf{Douglas}})$		
	$\xrightarrow{\mathsf{ct}}$	
		$\mathsf{ss} \leftarrow \mathsf{KEM.Decapsulate}(\mathsf{ct}, \mathsf{sk}_{\mathsf{Douglas}})$
$K \leftarrow \mathsf{KDF}(\mathsf{ss})$		$K \leftarrow \mathsf{KDF}(\mathsf{ss})$
	$\xleftarrow{\mathsf{MAC}_K(\cdots)}$	

KEM

Definition (Key Encapsulation Mechanism (KEM))

- $(\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{KEM.Keygen}()$
- $(\mathsf{ss}, \mathsf{ct}) \leftarrow \mathsf{KEM.Encapsulate}(\mathsf{pk})$
- $\mathsf{ss} \leftarrow \mathsf{KEM.Decapsulate}(\mathsf{ct}, \mathsf{sk})$

Example

To authenticate Douglas to Peter

Peter		**Douglas**
$(\mathsf{ss}, \mathsf{ct}) \leftarrow \mathsf{KEM.Encapsulate}(\mathsf{pk}_{\mathsf{Douglas}})$		
	$\xrightarrow{\mathsf{ct}}$	
		$\mathsf{ss} \leftarrow \mathsf{KEM.Decapsulate}(\mathsf{ct}, \mathsf{sk}_{\mathsf{Douglas}})$
$K \leftarrow \mathsf{KDF}(\mathsf{ss})$		$K \leftarrow \mathsf{KDF}(\mathsf{ss})$
	$\xleftarrow{\mathsf{MAC}_K(\cdots)}$	

KEM

Definition (Key Encapsulation Mechanism (KEM))

- $(\mathrm{pk}, \mathrm{sk}) \leftarrow \mathrm{KEM.Keygen}()$
- $(\mathrm{ss}, \mathrm{ct}) \leftarrow \mathrm{KEM.Encapsulate}(\mathrm{pk})$
- $\mathrm{ss} \leftarrow \mathrm{KEM.Decapsulate}(\mathrm{ct}, \mathrm{sk})$

Example

To authenticate Douglas to Peter

Peter		**Douglas**
$(\mathrm{ss}, \mathrm{ct}) \leftarrow \mathrm{KEM.Encapsulate}(\mathrm{pk}_{\mathrm{Douglas}})$		
	$\xrightarrow{\mathrm{ct}}$	
		$\mathrm{ss} \leftarrow \mathrm{KEM.Decapsulate}(\mathrm{ct}, \mathrm{sk}_{\mathrm{Douglas}})$
$K \leftarrow \mathrm{KDF}(\mathrm{ss})$		$K \leftarrow \mathrm{KDF}(\mathrm{ss})$
	$\xleftarrow{\mathrm{MAC}_K(\cdots)}$	

KEM authentication in TLS

Problem

- In TLS, the client doesn't already have the public key of the server!
- To put this in TLS 1.3, we need an extra roundtrip!
- TLS 1.3 tried very hard to finish the handshake a single roundtrip.

KEM authentication in TLS

Problem

- In TLS, the client doesn't already have the public key of the server!
- To put this in TLS 1.3, we need an extra roundtrip!
- TLS 1.3 tried very hard to finish the handshake a single roundtrip.

Solution

Implicitly authenticated key exchange: the client encapsulates to the server's long-term public key *but does not wait until they get the* MAC before sending data!

KEM authentication in TLS

Problem

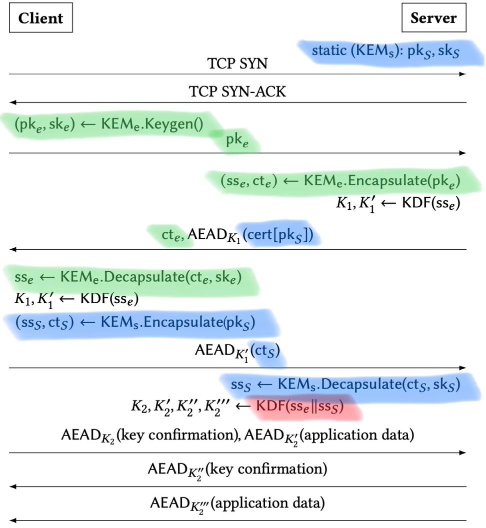
- In TLS, the client doesn't already have the public key of the server!
- To put this in TLS 1.3, we need an extra roundtrip!
- TLS 1.3 tried very hard to finish the handshake a single roundtrip.

Solution

Implicitly authenticated key exchange: the client encapsulates to the server's long-term public key *but does not wait until they get the* MAC before sending data!
Seen in HMQV (DH), BCGP09 & FSXY12 (KEMs), . . . , Signal, Noise, Wireguard, . . .

KEMTLS

- Ephemeral key exchange
- Static-KEM authentication
- Combine shared secrets
- Allow client to send application data before receiving server's key confirmation



Client | **Server**

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

TCP SYN →

← TCP SYN-ACK

$(\mathrm{pk}_e, \mathrm{sk}_e) \leftarrow \mathrm{KEM_e.Keygen}()$

pk_e →

$(\mathrm{ss}_e, \mathrm{ct}_e) \leftarrow \mathrm{KEM_e.Encapsulate}(\mathrm{pk}_e)$

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

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

$\mathrm{ss}_e \leftarrow \mathrm{KEM_e.Decapsulate}(\mathrm{ct}_e, \mathrm{sk}_e)$

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

$(\mathrm{ss}_S, \mathrm{ct}_S) \leftarrow \mathrm{KEM_s.Encapsulate}(\mathrm{pk}_S)$

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

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

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

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

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

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

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

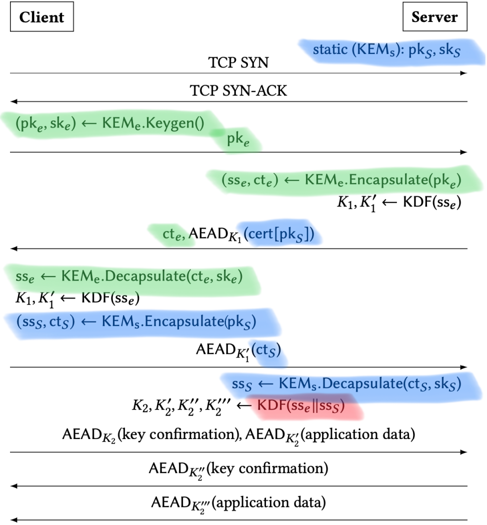
$\mathsf{AEAD}_{K_2}(\text{key confirmation}), \mathsf{AEAD}_{K_2'}(\text{application data})$ →

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

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

KEMTLS

- Ephemeral key exchange
- Static-KEM authentication
- Combine shared secrets
- Allow client to send application data before receiving server's key confirmation



Choosing algorithms

Ephemeral Key Exchange

- $\sim$ IND-CCA KEM
- Ideally fast with small pk $+$ ct

KEM for server authentication

- IND-CCA KEM
- Ideally fast with small pk $+$ ct

Choosing algorithms

Ephemeral Key Exchange

- $\sim$ IND-CCA KEM
- Ideally fast with small pk $+$ ct

KEM for server authentication

- IND-CCA KEM
- Ideally fast with small pk $+$ ct

Root CA certificate

- Already present on client
- Only care about signature size

Choosing algorithms

Ephemeral Key Exchange

- $\sim$ IND-CCA KEM
- Ideally fast with small pk $+$ ct

KEM for server authentication

- IND-CCA KEM
- Ideally fast with small pk $+$ ct

Intermediate CA certificate

- Small public key + signature size

Root CA certificate

- Already present on client
- Only care about signature size

Comparison[1]

Labels ABCD:

A = ephemeral KEM

B = leaf certificate

C = intermediate CA

D = root CA

Dilithium

Falcon

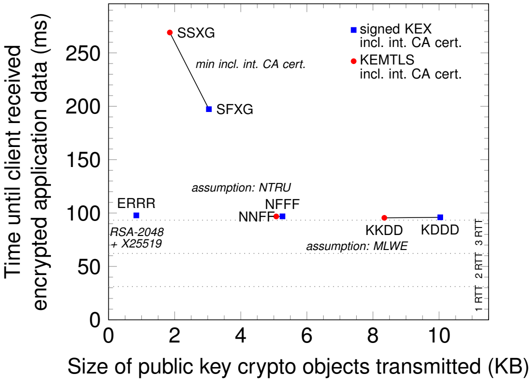
GeMSS

Kyber

NTRU

SIKE

$XMSS_s^{MT}$



[1] Rustls with AVX2 implementations. Emulated network: latency 31.1 ms, 1000 Mbps, no packet loss. Average of 100000 iterations.

Observations on emulated experiments

- 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)

Real world measurements ft. Cloudflare

Experimenting in the real world

- Experimental implementation in Go standard library TLS
- Branch https://github.com/cloudflare/go/tree/cf-pq-kemtls
- Based on Delegated Credentials (`draft-ietf-tls-subcerts`)
- Also implements client authentication

Experimenting in the real world

- Experimental implementation in Go standard library TLS
- Branch https://github.com/cloudflare/go/tree/cf-pq-kemtls
- Based on Delegated Credentials (`draft-ietf-tls-subcerts`)
- Also implements client authentication

We intend to do measurements on real networks

- between Cloudflare DCs
- Measure more aspects than just handshake time
- ???

Hope to report more soon

Post-Quantum TLS without Handshake signatures

- Implicit authentication via KEMs
- Preserve client ability to do request after 1RTT
- Saves bytes on the wire and server CPU cycles

- ACM CCS 2020 doi: 10.1145/3372297.3423350
- Full version with proofs: ia.cr/2020/534

Cloudflare is helping us investigate KEMTLS in the **real world**.
Experimental implementation in branch `cf-pq-kemtls` at github.com/cloudflare/go.
Hopefully results soon™ — keep an eye on the Cloudflare Research Blog.

Appendix

FAQ

- Client authentication?
  - We provide a sketch in Appendix D, but mostly leave it for future work
  - Naive way does require a full additional round-trip
- What about TLS 1.3 0-RTT?
  - 0-RTT is for resumption. You can do the same thing in KEMTLS .
  - We see opportunities for more efficient handshakes when resuming or in scenarios with pre-distributed KEM public keys.
- Server can't send application data in its first TLS flow. Will that break HTTP/3 where the server sends a `SETTINGS` frame?
  - Could be included in an extension as a server-side variant of ALPN
- How do you do certificate lifecycle management (issuance, revocation) with KEM public keys?
  - At first glance many of these issues seem non-trivial: currently these assume the public key can be used for signatures in some way
  - Another good direction for future work

Communications sizes

		Abbrv.	KEX (pk+ct)	Excluding intermediate CA certificate HS auth (ct/sig)	Leaf crt. subject (pk)	Leaf crt. (signature)	Sum excl. int. CA cert.	Including intermediate CA certificate 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 (Signed KEX)	**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	2711
	Min. incl. int. CA cert.	SFXG	SIKE 405	Falcon 690	Falcon 897	XMSS_s^{MT} 979	**2971**	XMSS_s^{MT} 32	GeMSS 32	**3035**	GeMSS 352180	4056
	Min. excl. int. CA cert.	SFGG	SIKE 405	Falcon 690	Falcon 897	GeMSS 32	**2024**	GeMSS 352180	GeMSS 32	**354236**	GeMSS 352180	355737
	Assumption: MLWE+MSIS	KDDD	Kyber 1536	Dilithium 2044	Dilithium 1184	Dilithium 2044	**6808**	Dilithium 1184	Dilithium 2044	**10036**	Dilithium 1184	11094
	Assumption: NTRU	NFFF	NTRU 1398	Falcon 690	Falcon 897	Falcon 690	**3675**	Falcon 897	Falcon 690	**5262**	Falcon 897	6227
KEMTLS	**Min. incl. int. CA cert.**	SSXG	SIKE 405	SIKE 209	SIKE 196	XMSS_s^{MT} 979	**1789**	XMSS_s^{MT} 32	GeMSS 32	**1853**	GeMSS 352180	2898
	Min. excl. int. CA cert.	SSGG	SIKE 405	SIKE 209	SIKE 196	GeMSS 32	**842**	GeMSS 352180	GeMSS 32	**353054**	GeMSS 352180	354578
	Assumption: MLWE+MSIS	KKDD	Kyber 1536	Kyber 736	Kyber 800	Dilithium 2044	**5116**	Dilithium 1184	Dilithium 2044	**8344**	Dilithium 1184	9398
	Assumption: NTRU	NNFF	NTRU 1398	NTRU 699	NTRU 699	Falcon 690	**3486**	Falcon 897	Falcon 690	**5073**	Falcon 897	6066

Time measurements

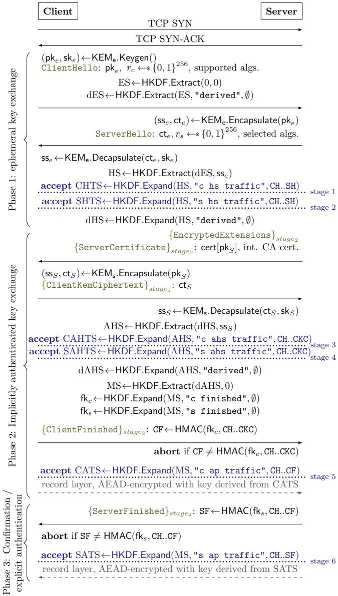
		Computation time for asymmetric crypto				Handshake time (31.1 ms latency, 1000 Mbps bandwidth)						Handshake time (195.6 ms latency, 10 Mbps bandwidth)					
		Excl. int. CA cert.		Incl. int. CA cert.		Excl. int. CA cert.			Incl. int. CA cert.			Excl. int. CA cert.			Incl. int. CA cert.		
		Client	Server	Client	Server	Client sent req.	Client recv. resp.	Server HS done	Client sent req.	Client recv. resp.	Server HS done	Client sent req.	Client recv. resp.	Server HS done	Client sent req.	Client recv. resp.	Server HS done
TLS 1.3	**ERRR**	0.134	0.629	0.150	0.629	66.4	**97.6**	35.4	66.6	**97.8**	35.6	397.1	**593.3**	201.3	398.2	**594.3**	202.3
	SFXG	40.058	21.676	40.094	21.676	165.8	**196.9**	134.0	166.2	**197.3**	134.4	482.1	**678.4**	285.8	482.5	**678.8**	286.2
	SFGG	34.104	21.676	34.141	21.676	154.9	**186.0**	123.1	259.0	**290.2**	227.1	473.7	**669.8**	277.5	10936.3	**11902.5**	10384.1
	KDDD	0.080	0.087	0.111	0.087	64.3	**95.5**	33.3	64.8	**96.0**	33.8	411.6	**852.4**	446.1	415.9	**854.7**	448.0
	NFFF	0.141	0.254	0.181	0.254	65.1	**96.3**	34.1	65.6	**96.9**	34.7	398.1	**662.2**	269.2	406.7	**842.8**	443.5
KEMTLS	**SSXG**	61.456	41.712	61.493	41.712	202.1	**268.8**	205.6	202.3	**269.1**	205.9	505.8	**732.0**	339.7	506.1	**732.4**	340.1
	SSGG	55.503	41.712	55.540	41.712	190.4	**256.6**	193.4	293.3	**359.5**	296.3	496.8	**723.0**	330.8	10859.5	**11861.0**	10331.7
	KKDD	0.060	0.021	0.091	0.021	63.4	**95.0**	32.7	63.9	**95.5**	33.2	399.2	**835.1**	439.9	418.9	**864.2**	447.6
	NNFF	0.118	0.027	0.158	0.027	63.6	**95.2**	32.9	64.2	**95.8**	33.5	396.2	**593.4**	200.6	400.0	**835.6**	440.2

		Computation time for asymmetric crypto				Handshake tim	
		Excl. int. CA cert.		Incl. int. CA cert.		Excl. int. CA	
		Client	Server	Client	Server	Client sent req.	Client recv. res
TLS 1.3	**ERRR**	0.134	0.629	0.150	0.629	66.4	**97**
	SFXG	40.058	21.676	40.094	21.676	165.8	**196**
	SFGG	34.104	21.676	34.141	21.676	154.9	**186**
	KDDD	0.080	0.087	0.111	0.087	64.3	**95**
	NFFF	0.141	0.254	0.181	0.254	65.1	**96**
KEMTLS	**SSXG**	61.456	41.712	61.493	41.712	202.1	**268**
	SSGG	55.503	41.712	55.540	41.712	190.4	**256**
	KKDD	0.060	0.021	0.091	0.021	63.4	**95**
	NNFF	0.118	0.027	0.158	0.027	63.6	**95**

pto rt. r	Handshake time (31.1 ms latency, 1000 Mbps bandwidth) Excl. int. CA cert. Client sent req.	Client recv. resp.	Server HS done	Incl. int. CA cert. Client sent req.	Client recv. resp.	Server HS done	H Clie sent r
629	66.4	**97.6**	35.4	66.6	**97.8**	35.6	39
676	165.8	**196.9**	134.0	166.2	**197.3**	134.4	48
676	154.9	**186.0**	123.1	259.0	**290.2**	227.1	47
087	64.3	**95.5**	33.3	64.8	**96.0**	33.8	41
254	65.1	**96.3**	34.1	65.6	**96.9**	34.7	39
712	202.1	**268.8**	205.6	202.3	**269.1**	205.9	50
712	190.4	**256.6**	193.4	293.3	**359.5**	296.3	49
021	63.4	**95.0**	32.7	63.9	**95.5**	33.2	39
027	63.6	**95.2**	32.9	64.2	**95.8**	33.5	39

idth) rt. Server HS done	Handshake time (195.6 ms latency, 10 Mbps bandwidth) Excl. int. CA cert. Client sent req.	Client recv. resp.	Server HS done	Incl. int. CA cert. Client sent req.	Client recv. resp.	Server HS done
35.6	397.1	**593.3**	201.3	398.2	**594.3**	202.3
134.4	482.1	**678.4**	285.8	482.5	**678.8**	286.2
227.1	473.7	**669.8**	277.5	10936.3	**11902.5**	10384.1
33.8	411.6	**852.4**	446.1	415.9	**854.7**	448.0
34.7	398.1	**662.2**	269.2	406.7	**842.8**	443.5
205.9	505.8	**732.0**	339.7	506.1	**732.4**	340.1
296.3	496.8	**723.0**	330.8	10859.5	**11861.0**	10331.7
33.2	399.2	**835.1**	439.9	418.9	**864.2**	447.6
33.5	396.2	**593.4**	200.6	400.0	**835.6**	440.2

KEMTLS in more detail



Sending application data before FIN

The client sends data before receiving `ServerFinished`.

Does this mean you can downgrade to weak crypto to aid future (quantum) decryption?

Sending application data before FIN

The client sends data before receiving `ServerFinished`.

Does this mean you can downgrade to weak crypto to aid future (quantum) decryption?

- Can't downgrade to signed TLS 1.3
  - TLS 1.3 handshake signature stops the attack

Sending application data before FIN

The client sends data before receiving `ServerFinished`.

Does this mean you can downgrade to weak crypto to aid future (quantum) decryption?

- Can't downgrade to signed TLS 1.3
  - TLS 1.3 handshake signature stops the attack
- Active adversary might try to downgrade first client-to-server flow
- Only to whatever algorithms the client advertised in `ClientHello`
  - Don't support pre-quantum in KEMTLS

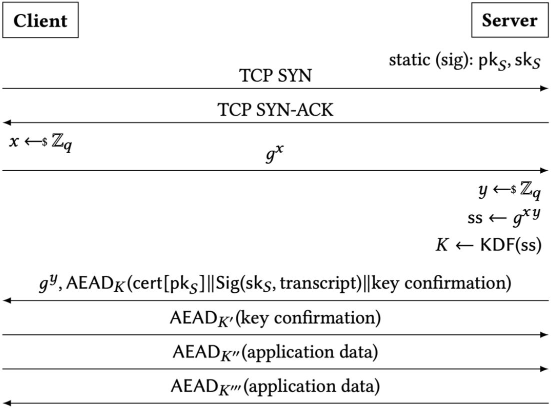
Sending application data before FIN

The client sends data before receiving `ServerFinished`.

Does this mean you can downgrade to weak crypto to aid future (quantum) decryption?

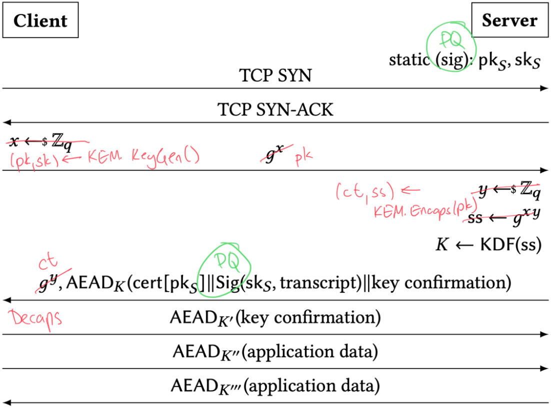
- Can't downgrade to signed TLS 1.3
  - TLS 1.3 handshake signature stops the attack
- Active adversary might try to downgrade first client-to-server flow
- Only to whatever algorithms the client advertised in `ClientHello`
  - Don't support pre-quantum in KEMTLS
- The handshake will no longer sucessfully complete
  - `ServerFinished` reveals the downgrade unless MAC, KEM, KDF or hash are broken *at time of attack*
  - Once `SF` is received: retroactive full downgrade resilience
  - You also get upgraded from weak to full forward secrecy at this stage

TLS 1.3 Handshake



- Key exchange: Diffie–Hellman
- Authentication: Signatures

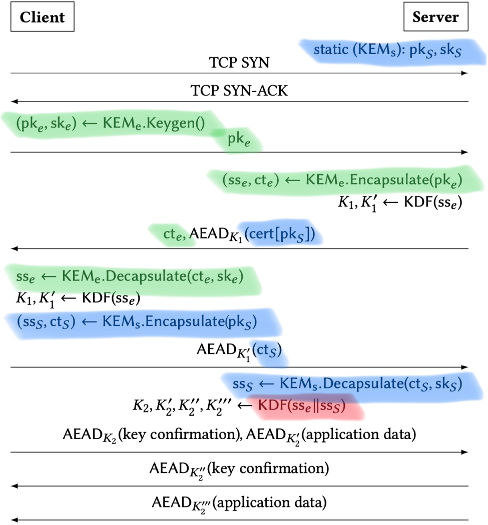
Post-Quantum TLS 1.3 Handshake



- Key exchange: Post-Quantum Key-Encapsulation Mechanisms
- Authentication: Post-Quantum Signatures

KEMTLS

- Ephemeral key exchange
- Static-KEM authentication
- Combine shared secrets
- Allow client to send application data before receiving server's key confirmation



Post-Quantum TLS without Handshake signatures

- Implicit authentication via KEMs
- Preserve client ability to do request after 1RTT
- Saves bytes on the wire and server CPU cycles

- ACM CCS 2020 doi: 10.1145/3372297.3423350
- Full version with proofs: ia.cr/2020/534

Cloudflare is helping us investigate KEMTLS in the **real world**.
Experimental implementation in branch `cf-pq-kemtls` at
github.com/cloudflare/go.
Hopefully results soon™ — keep an eye on the Cloudflare Research Blog.