Bringing PQC into practice

Bas Westerbaan, Cloudflare Research

QSI Spring School, Porto, March 15th, 2024

This lecture

1. ✅ Design post-quantum algorithm mathematically
2. ❔ [...] ⬅️ topic of this lecture.
3. ⚛️ Deploy PQC worldwide

But first, for your context

About Cloudflare

We run a global network spanning 310 cities in over 120 countries.

Started of as a CDN and DDoS mitigation company, we now offer many more services, including

- 1.1.1.1, public DNS resolver
- Workers, serverless compute
- SASE, to protect corporate networks

We serve nearly [20% of all websites]() and process 55 million HTTP requests per second. Approximately 30% of Fortune 1000 are paying customers.

Building a better Internet

Cloudflare cares deeply about a private, secure and fast Internet, helping design, and adopt, among others:

- Free SSL (2014), TLS 1.3 and QUIC
- DNS-over-HTTPS
- Private Relay / OHTTP
- Encrypted ClientHello

And, the topic today:

- Migrating to post-quantum cryptography.

1. ✅ Design post-quantum algorithm mathematically
2. Cryptography engineering and a lot more research
3. ⚛️ Deploy PQC worldwide

There is a lot involved in bringing cryptography into practice. In this short lecture we can only touch upon a few topics briefly.

What is involved

Fast, secure, and correct implementation.

Standardisation: fix wire format, APIs, integration into protocols, and other menial details.

Work around protocol constraints, and buggy software / hardware.

(…)

As we’ll see: these are not separate steps. All have been happening at the same time, are mostly at odds with each other, and feed back into the design.

Fast implementation

"[...] saving a single hash in TLS saves compute time worth millions of dollars/CO2 emissions/energy [...]"

— Sophie Schmieg, Google, source.

Example: polynomial product

In many lattice-based cryptosystems, we need to compute the product $s \cdot t$ of two polynomials s, t over $GF(q)$ modulo x^n+1.

(For instance, in Kyber q=3329 and n=256.)

The schoolbook method requires n^2 modular multiplications.

For instance, for n=3, we have $x^3=-1$, and so:

$$
\begin{aligned}
s \cdot t = \quad & s_0t_0 - s_1t_2 - s_2t_1 + \\
& (s_0t_1 + s_1t_0 - s_2t_2)\, x + \\
& (s_0t_2 + s_1t_1 + s_0t_2)\, x^2
\end{aligned}
$$

Schoolbook

```
def schoolbook_product(s, t):
    cs = [0]*n
    for i in range(n):
        for j in range(i):
            cs[i] = (cs[i] + s[j]*t[i-j]) % q
    for i in range(n-1):
        for j in range(n-i-1):
            cs[i] = (cs[i] - s[n-1-j]*t[i+j+1]) % q
    return cs
```

Not fast, but clearly correct.

(⚠️ Don't write cryptography in pure Python.)

Schoolbook

```
def schoolbook_product(s, t):
    cs = [0]*n
    for i in range(n):
        for j in range(i):
            cs[i] = (cs[i] + s[j]*t[i-j]) % q
    for i in range(n-1):
        for j in range(n-i-1):
            cs[i] = (cs[i] - s[n-1-j]*t[i+j+1]) % q
    return cs
```

Not fast, but ~~clearly correct~~.

Schoolbook

```
def schoolbook_product(s, t):
    cs = [0]*n
    for i in range(n):
        for j in range(i+1):
            cs[i] = (cs[i] + s[j]*t[i-j]) % q
    for i in range(n-1):
        for j in range(n-i-1):
            cs[i] = (cs[i] - s[n-1-j]*t[i+j+1]) % q
    return cs
```

Not fast, but ~~clearly~~ correct. Is the implementation safe?

(... assuming this was written in Rust or C.)

Safe?

```python
def schoolbook_product(s, t):
    cs = [0]*n
    for i in range(n):
        for j in range(i+1):
            cs[i] = (cs[i] + s[j]*t[i-j]) % q
    for i in range(n-1):
        for j in range(n-i-1):
            cs[i] = (cs[i] - s[n-1-j]*t[i+j+1]) % q
    return cs
```

On most platforms, the runtime of modulus / division depends on the arguments. “Not constant-time”. If *s* or *t* is secret, this could be problematic. (For recent similar issue, see KyberSlash.)

Barrett modular reduction

For q=3329, we can compute x mod q for $0 \leq x < 2^{16}$ as

```
def barrett(x):
    return x - ((x * 20159) >> 26) * q
```

Barrett modular reduction

For q=3329, we can compute x mod q for $0 \leq x < 2^{16}$ as

```
def barrett(x):
    return x - ((x * 20159) >> 26) * q
```

We have $x \bmod q = x - \lfloor x/q \rfloor q$ for any x and $\lfloor x/2^a \rfloor = x \gg a$.

For q=3329, we have $1/q \approx 20159 / 2^{26}$.

For $0 \leq x < 153{,}133$ the error disappears in the floor.

Barrett modular reduction

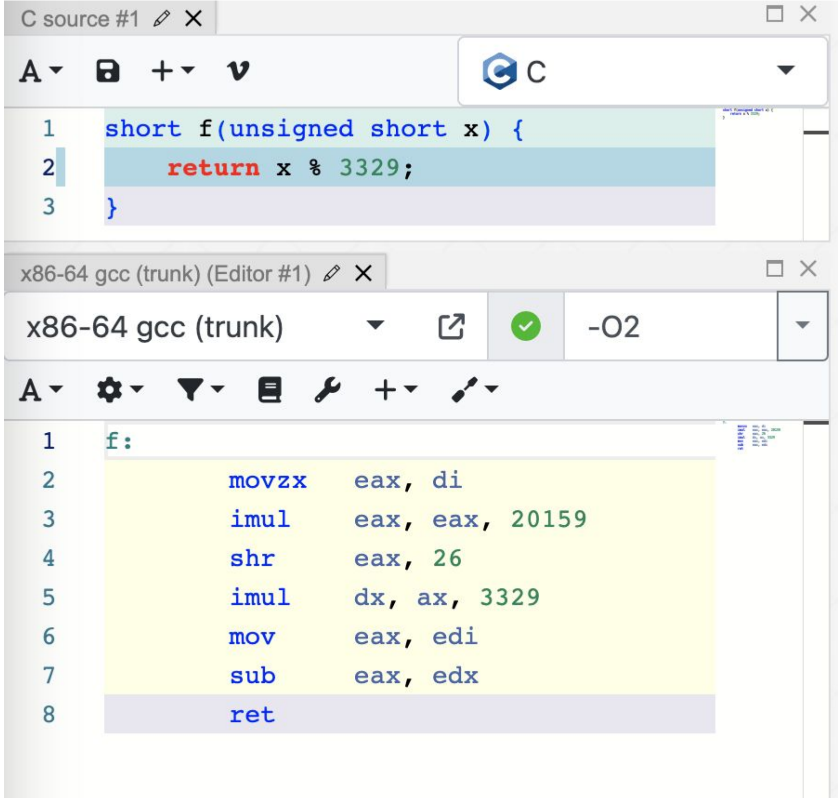
For q=3329, we can compute x mod q for $0 \leq x < 2^{16}$ as

```
def barrett(x):
    return x - ((x * 20159) >> 26) * q
```

⚠️ On many, but not every platform, multiplication runs in constant time.

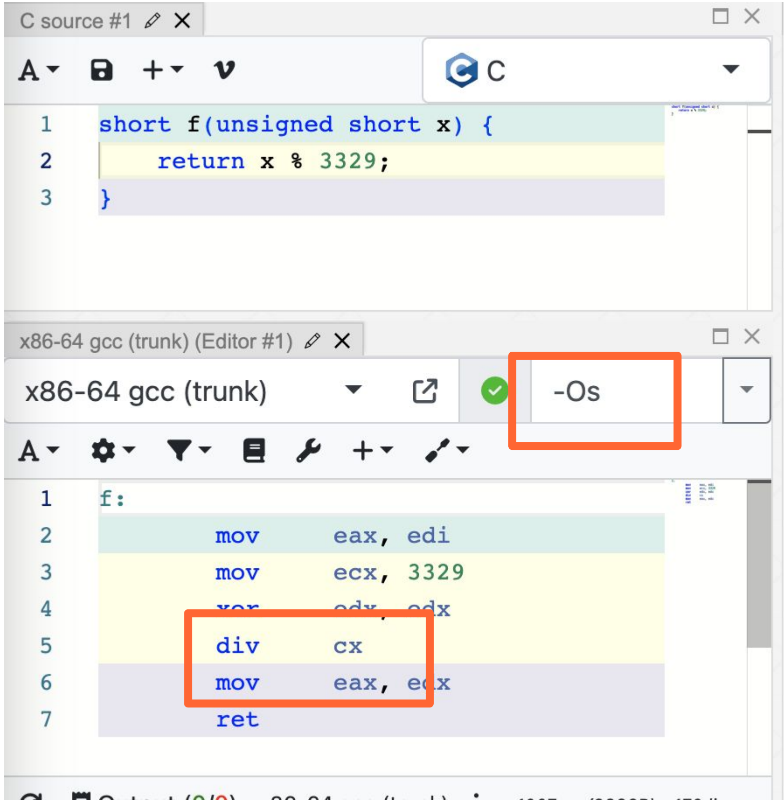
Typically Barrett reduction is faster than native division, thus...

Many compilers know about Barrett



This is about the speed improvement — compilers typically don't care about constant-time code.

Don’t rely on the compiler!



Luckily compilers don’t replace Barrett reduction by divisions, yet…

To be 100% sure, you need to write crypto by hand in assembly, for a particular platform.

An aside: other timing side-channels

Because of processor memory and instruction caches, we can't index (eg. `array[secret]`) or branch (eg. `1 if secret < 0 else 0`) on secret values.

The latter can be done in constant-time, on a platform with two's complement integers, for `int32_t` in C as:

```
(uint32_t)secret >> 31
```

(Fun exercise: how would you do the former `uint32_t[8]` ?)

Schoolbook

```
def schoolbook_product(s, t):
    cs = [0]*n
    for i in range(n):
        for j in range(i+1):
            cs[i] = barrett(cs[i] + s[j]*t[i-j])
    for i in range(n-1):
        for j in range(n-i-1):
            cs[i] = barrett(cs[i] - s[n-1-j]*t[i+j+1])
    return cs
```

Still slow: $2n^2$ multiplications.

Lazy reduction

```
def schoolbook_product(s, t):
    cs = [0]*n
    for i in range(n):
        for j in range(i+1):
            cs[i] = cs[i] + s[j]*t[i-j]
    for i in range(n-1):
        for j in range(n-i-1):
            cs[i] = cs[i] - s[n-1-j]*t[i+j+1]
    return [barrett(cs[i]) for i in range(n)]
```

Move modular reduction to the end. Saves n(n-1) multiplications. ⚠️ We need to be mindful of integer overflow.

We got to go faster

There are many techniques for faster polynomial multiplication. One particularly popular method is the number theoretic transform (NTT), which works best for specific q and rings.

Kyber, Dilithium, and Falcon choose their polynomial rings so that they can use NTT-style speed ups.

NTT (1)

q is chosen such that $256 \mid q - 1$, which ensures there is a 256th primitive root of unity ζ. That is: $\zeta^{256} = 1$, and $\zeta^{128} \neq 1$. So $\zeta^{128} = -1$. That allows us to split x^n+1 completely:

$$\begin{aligned}
x^N + 1 &= x^N - \zeta^N \\
&= (x^{\frac{N}{2}} - \zeta^{\frac{N}{2}})(x^{\frac{N}{2}} + \zeta^{\frac{N}{2}}) \\
&= (x^{\frac{N}{2}} - \zeta^{\frac{N}{2}})(x^{\frac{N}{2}} - \zeta^{\frac{3N}{2}}) \\
&= (x^{\frac{N}{4}} - \zeta^{\frac{N}{4}})(x^{\frac{N}{4}} + \zeta^{\frac{N}{4}})(x^{\frac{N}{4}} - \zeta^{\frac{3N}{4}})(x^{\frac{N}{4}} + \zeta^{\frac{3N}{4}}) \\
&\vdots \\
&= \prod_{i=0}^{N-1} x - \zeta^{2i+1}.
\end{aligned}$$

NTT (2)

This allows us to factor our ring by the Chinese remainder thm:

$$\mathbb{F}_q[x]/_{\langle x^N+1\rangle} \cong \prod_{i=0}^{N-1} \mathbb{F}_q[x]/_{\langle x-\zeta^{2i+1}\rangle} \cong \mathbb{F}_q^N$$

Multiplication on the right is fast: just componentwise.

The isomorphism is given by evaluating on odds powers of ζ:

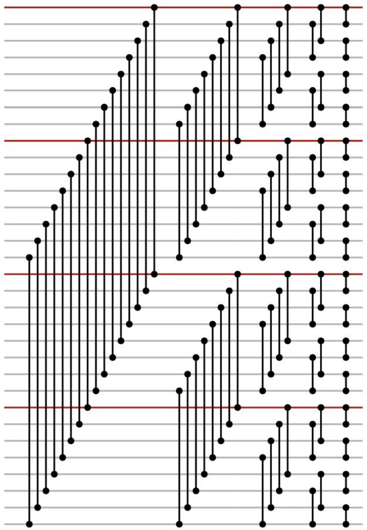
$$p \mapsto (p(\zeta), p(\zeta^3), \ldots, p(\zeta^{2N+1}))$$

Computing it this way is slow, but...

NTT (3)

We can evaluate the isomorphism step-by-step:

$$\mathbb{F}_q[x]/_{\langle x^N+1\rangle} \cong \mathbb{F}_q[x]/_{\langle x^{\frac{N}{2}}-\zeta^{\frac{N}{2}}\rangle} \times \mathbb{F}_q[x]/_{\langle x^{\frac{N}{2}}-\zeta^{\frac{3N}{2}}\rangle} \cong \cdots \cong \mathbb{F}_q^N$$



The map is given by the picture on the left (for n=32), where each vertical line represents the map

$$(a, b) \rightarrow (a+\zeta^r b, a-\zeta^r b)$$

for the appropriate *r*.

These are called Cooley–Tukey butterflies.

NTT (4)

```
def brv(x): # bit reversal of x
    return int(''.join(reversed(bin(x)[2:].zfill(nBits))), 2)

def ntt(p):
    cs = list(p)
    layer = n // 2; zi = 0
    while layer >= 1:
        for offset in range(0, n-layer, 2*layer):
            zi += 1
            z = pow(zeta, brv(zi), q)
            for j in range(offset, offset+layer):
                t = barrett(z * cs[j + layer])
                cs[j + layer] = barrett(cs[j] - t)
                cs[j] = barrett(cs[j] + t)
        layer //= 2
    return cs
```

$\log_2 n$ layers.

Thus $3n \log_2 n$ multiplications.

We can reduce to essentially $n \log_2 n$ by lazy reductions.

NTT (5), putting it together

```
def ntt_product(s, t):
    hat_s = ntt(s)
    hat_t = ntt(t)
    hat_st = [barrett(a*b)
              for a,b in zip(hat_s, hat_t)]
    return intt(hat_st)
```

Much faster: approximately 2 $n \log_2 n$ multiplications.

How do we know it's correct?

Got to go even faster

Most modern CPU have *single-instruction/multiple-data* (SIMD) registers, such as AVX2 on x64, and NEON on ARM.

On AVX2, there are sixteen 256-bit registers, that can be used in different ways. For instance: 4 times u64, or 16 times u16.

The VPMULLW instruction pairwise multiplies the sixteen u16 in two given SIMD registers.

Intrinsics

Some languages (eg. Rust, C, Zig) make it easy to use SIMD.

```
#![feature(portable_simd)]

use std::simd::u16x16;
use std::simd::Simd;

fn main() {
    let x : u16x16 = Simd::from_array(
        [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]);
    let y : u16x16 =  Simd::from_array(
        [15, 14, 13, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0]);
    let z : u16x16 = x * y;
    println!("{:?}", z.to_array());
}
```

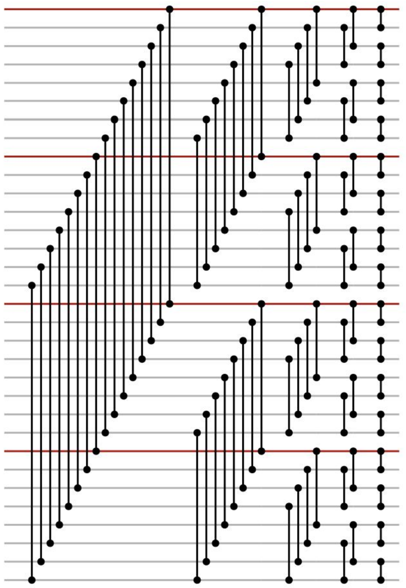
Standard Output

```
[0, 14, 26, 36, 44, 50, 54, 56, 56, 54, 50, 44, 36, 26, 14, 0]
```

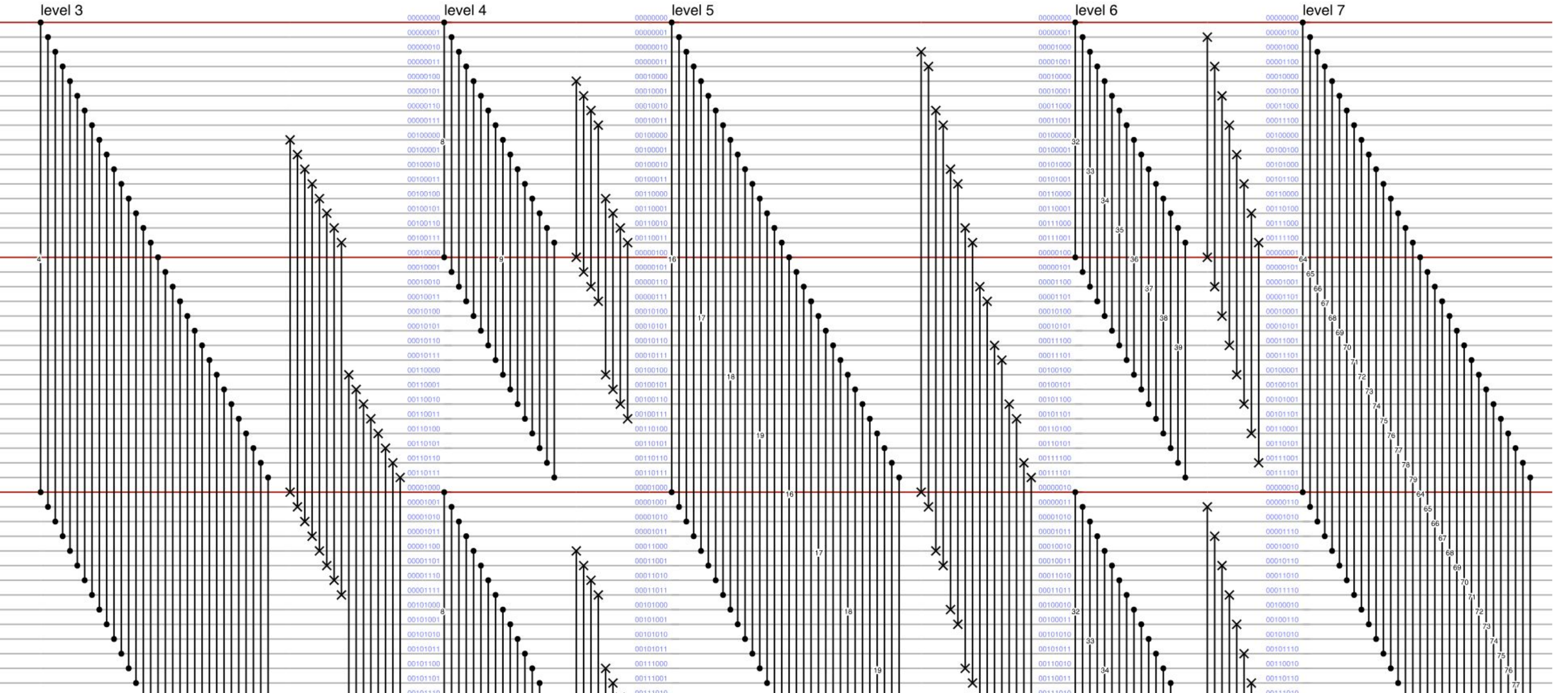
Accelerating NTT with SIMD

With AVX2 we can compute 16 butterflies for *q*=3329 at the same time. The hard part: the right coefficients have to be in the right SIMD registers! With approach on the right we can only use full potential for first layer.

The trick is to do some clever shuffling in between.



(Part of the) AVX2 NTT for Kyber (source)



```
TEXT ·nttAVX2(SB), NOSPLIT, $0-8
        MOVQ         p+0(FP), AX
        LEAQ         ·ZetasAVX2+0(SB), CX
        MOVL         $0x00000d01, DX
        VMOVD        DX, X0
        VPBROADCASTW X0, Y15
        VPBROADCASTW (CX), Y0
        VPBROADCASTW 2(CX), Y1
        VMOVDQU      (AX), Y7
        VMOVDQU      32(AX), Y8
        VMOVDQU      64(AX), Y9
        VMOVDQU      96(AX), Y10
        VMOVDQU      256(AX), Y11
        VMOVDQU      288(AX), Y12
        VMOVDQU      320(AX), Y13
        VMOVDQU      352(AX), Y14
        VPMULLW      Y11, Y0, Y2
        VPMULLW      Y12, Y0, Y3
        VPMULLW      Y13, Y0, Y4
        VPMULLW      Y14, Y0, Y5
        VPMULHW      Y11, Y1, Y11
        VPMULHW      Y12, Y1, Y12
        VPMULHW      Y13, Y1, Y13
        VPMULHW      Y14, Y1, Y14
        VPMULHW      Y2, Y15, Y2
```

(... 556 more instructions)

With handcrafted assembly we can make the best use of the AVX2 registers.

This makes Kyber NTT (n=256) on x64 with AVX2 very fast: ~200 cycles!

But is the assembly correct?

Computer-verified proof

At the moment, we've deployed a relatively slow (non SIMD) implementation of Kyber, as we're worried about mistakes.

We're looking into deploying an AVX2-optimised implementation by the Formosa team (who are meeting on the 9th floor now!) that comes with a computer-verified proof of correctness.

Ok, assume we got perfect implementations.

Now just ship it? 🚢

Changing the Internet / WebPKI is hard

- Very diverse. Many different users / stakeholders with varying (performance) constraints and update cycles.

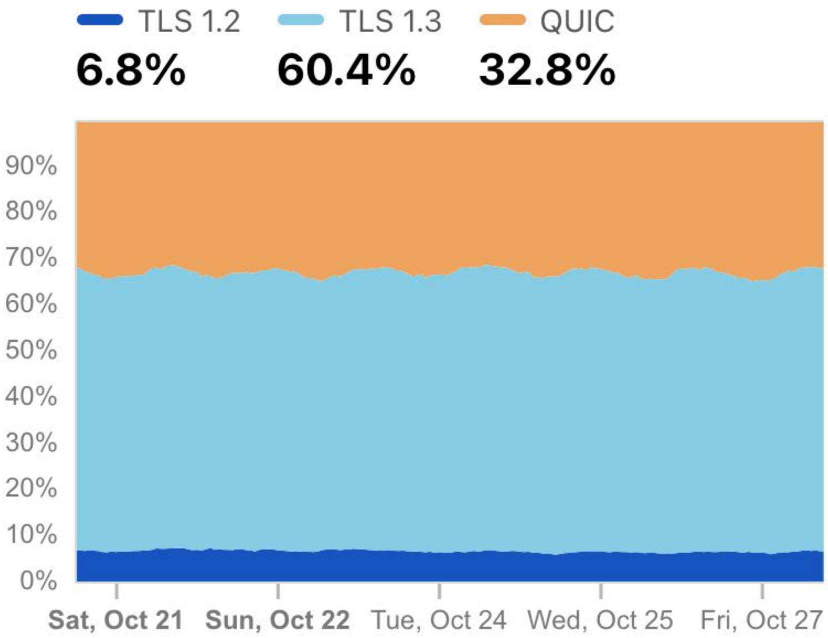
  We can't assume everyone is on fiber, or uses modern CPU, can store state, uses AVX2, or can update at all.

- Protocol ossification. Despite being designed to be upgradeable, any flexibility that isn't used in practice, is probably broken, because of faulty implementations.

TLS 1.3 migration

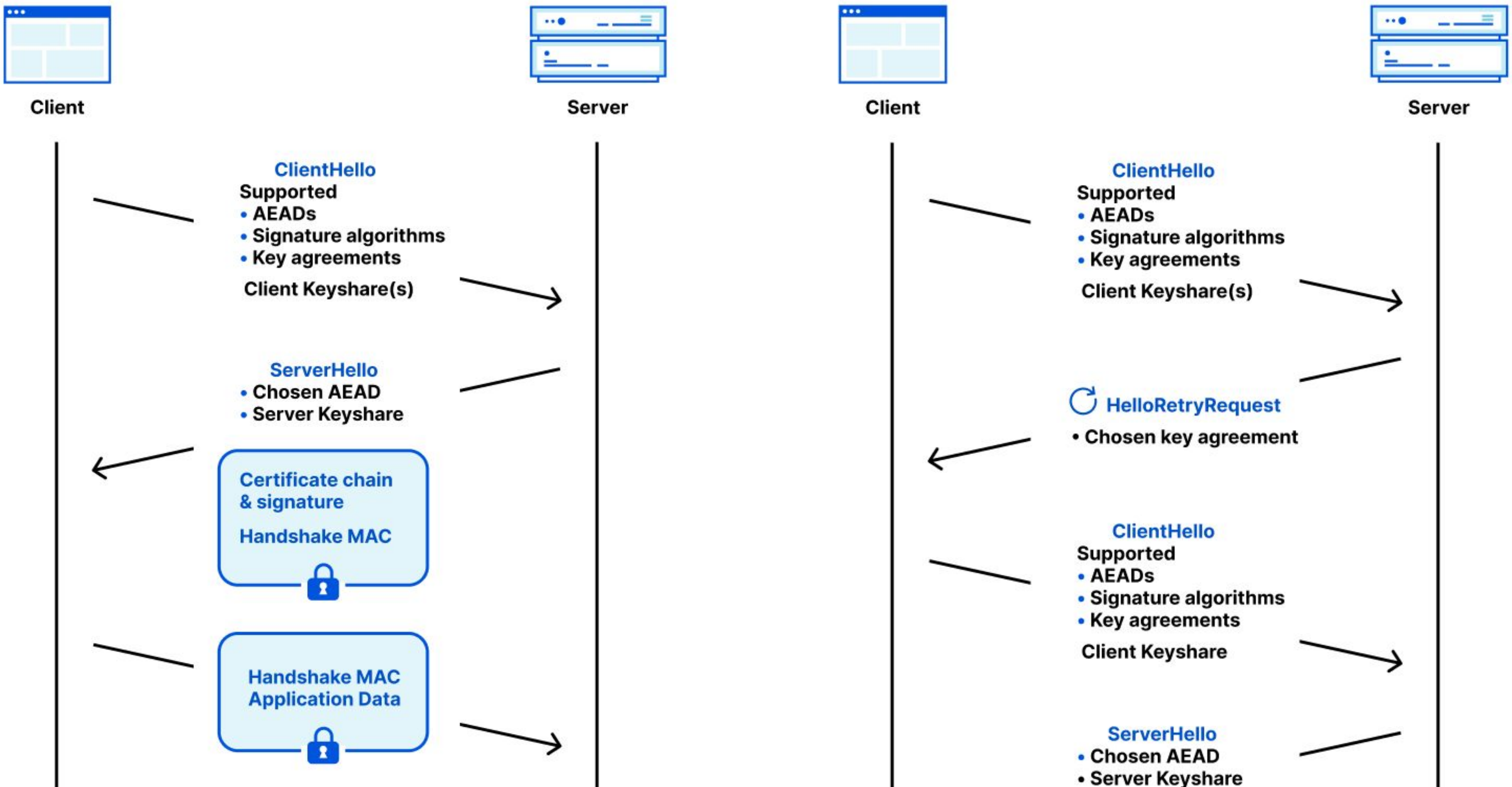
Early versions of TLS 1.3 were completely undeployable because of protocol ossification.

After six more years of testing and adding workarounds, the final version of TLS 1.3 is a success, used by over 90% of our visitors.



Cloudflare Radar

TLS 1.3 handshake



There will be *two* post-quantum migrations.

1. **Key agreement** 🤝

   Communication can be recorded today and decrypted in the future. We need to upgrade as soon as possible.

2. **Signatures** 🖋️

   Less urgent: need to be replaced before the arrival of cryptographically-relevant quantum computers.

Key agreement 🤝

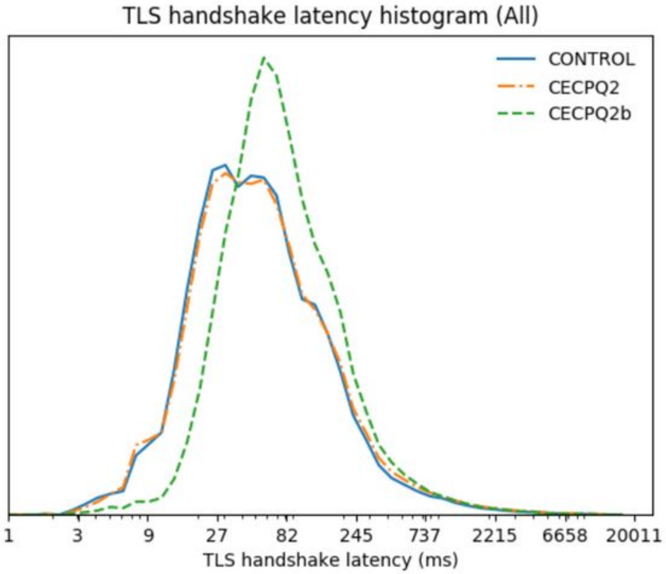
Urgent, and the *easier* one.

Feasibility study with Chrome

In 2019 we performed large-scale test of PQ kex with Chrome. Takeaways:

- Performance of lattice-based KEMs is acceptable.
- Significant amount of broken clients because of protocol ossification (*split ClientHello*.)

Google has been working with vendors to fix issues.



X25519. CECPQ2 is X25519+NTRU-HRSS (lattice) and CECPQ2b is X25519+SIKE (isogenies, broken)

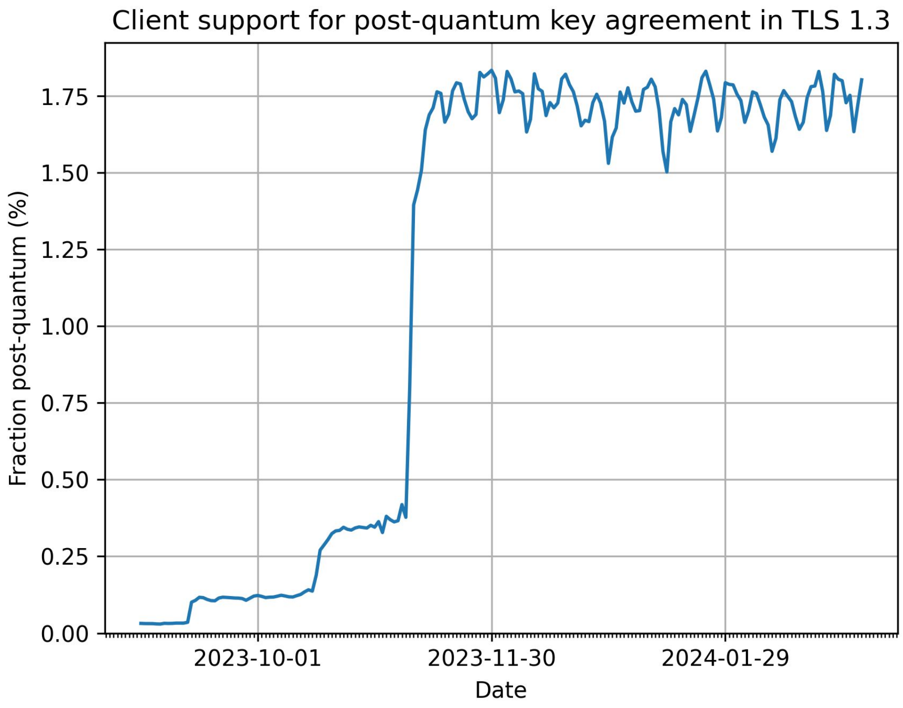
Early deployments

2022 coordinating at IETF, we enabled post-quantum key agreement (~20% Internet.)

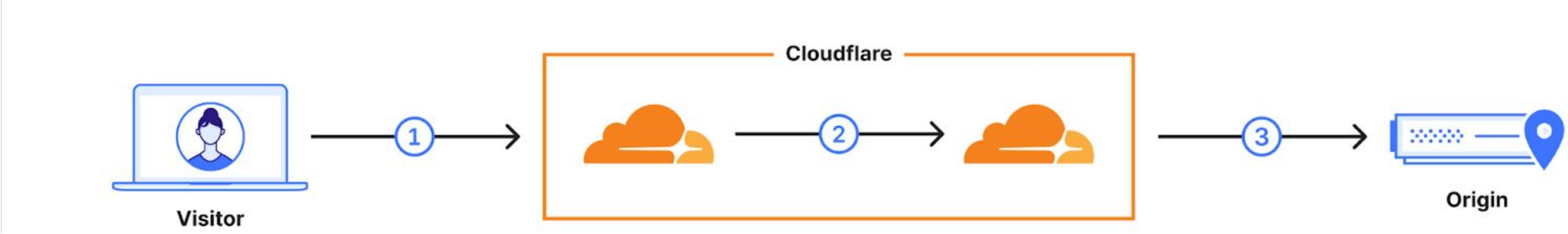
In 2023 Google enabled server-side as well.

Browsers:

- Chrome. Enabled for 10% of all traffic.
- Firefox. Opt-in in nightly.



Post-quantum to origins



We enabled support for PQ key agreement to origins (3).

0.5% of origins support PQ at time of writing.

0.34% incompatible when sending keyshare immediately. Interestingly, mostly broken *HelloRetryRequest* flow. We've reached out to customers to help remediate.

Promising early results

As of writing, no hard failures preventing further roll-out identified by Chrome 🤞.

It is likely that we will see double-digit percentage post-quantum key-agreement later this year.

Not just a technical challenge

In 2023 we've also commenced migrating our internal connections to post-quantum key agreement.

Huge effort: every engineering team created inventory of cryptography used, risks, and planned/executed migration.

Majority of our internal connections are secured (prioritizing sensitive connections), but a long fat tail remains.

On the upside: we did not encounter any performance or compatibility issues.

Key agreement 🤝

Urgent and the easier of the two to deploy. We're on track for ~30% client-side deployment in 2024.

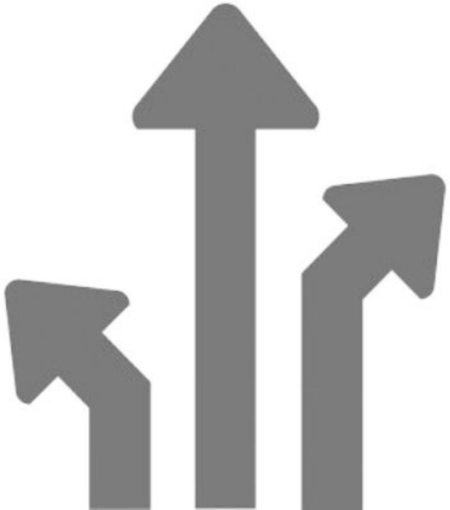
That took 5 years.

Signatures 🖋️

Less urgent, but much more challenging.

#1, many more parties involved:

Cryptography library developers, browsers, certificate authorities, HSM manufacturers, CT logs, and every server admin that cobbled together a PKI script.



#2, there is no all-round great PQ signature

		Size (bytes)		CPU time (lower is better)	
	PQ	**Public key**	**Signature**	**Signing**	**Verification**
Ed25519	❌	32	64	1 (baseline)	1 (baseline)
RSA-2048	❌	256	256	70	0.3
Dilithium2	✅	1,312	2,420	4.8	0.5
Falcon512	✅	897	666	8*	0.5
SPHINCS⁺128s	✅	32	7,856	8,000	2.8
SPHINCS⁺128f	✅	32	17,088	550	7

Online signing — Falcon's Achilles' heel

- For fast signing, Falcon requires a floating-point unit (FPU).
- We do not have enough experience running cryptography securely (constant-time) on the FPU.
- On commodity hardware, Falcon should not be used when signature creation can be timed, eg. TLS handshake.
- Not a problem for signature verification.

```c
static inline int64_t
fpr_rint(fpr x)
{
	/*
	 * We do not want to use llrint() since it might be not
	 * constant-time.
	 *
	 * Suppose that x >= 0. If x >= 2^52, then it is already an
	 * integer. Otherwise, if x < 2^52, then computing x+2^52 will
	 * yield a value that will be rounded to the nearest integer
	 * with exactly the right rules (round-to-nearest-even).
	 *
	 * In order to have constant-time processing, we must do the
	 * computation for both x >= 0 and x < 0 cases, and use a
	 * cast to an integer to access the sign and select the proper
	 * value. Such casts also allow us to find out if |x| < 2^52.
	 */
	int64_t sx, tx, rp, rn, m;
	uint32_t ub;

	sx = (int64_t)(x.v - 1.0);
	tx = (int64_t)x.v;
	rp = (int64_t)(x.v + 4503599627370496.0) - 4503599627370496;
	rn = (int64_t)(x.v - 4503599627370496.0) + 4503599627370496;

	/*
	 * If tx >= 2^52 or tx < -2^52, then result is tx.
	 * Otherwise, if sx >= 0, then result is rp.
	 * Otherwise, result is rn. We use the fact that when x is
	 * close to 0 (|x| <= 0.25) then both rp and rn are correct;
	 * and if x is not close to 0, then trunc(x-1.0) yields the
	 * appropriate sign.
	 */

	/*
	 * Clamp rp to zero if tx < 0.
	 * Clamp rn to zero if tx >= 0.
	 */
	m = sx >> 63;
	rn &= m;
	rp &= ~m;

	/*
	 * Get the 12 upper bits of tx; if they are not all zeros or
	 * all ones, then tx >= 2^52 or tx < -2^52, and we clamp both
	 * rp and rn to zero. Otherwise, we clamp tx to zero.
	 */
	ub = (uint32_t)((uint64_t)tx >> 52);
	m = -(int64_t)((((ub + 1) & 0xFFF) - 2) >> 31);
	rp &= m;
	rn &= m;
	tx &= ~m;

	/*
	 * Only one of tx, rn or rp (at most) can be non-zero at this
	 * point.
	 */
	return tx | rn | rp;
}
```

This function from Falcon as submitted to round 3 is not constant-time on ARMv7 as claimed.

Can you spot the error?

#3, there are many signatures on the Web

- Root on intermediate
- Intermediate on leaf
- Leaf on handshake
- Two SCTs for Certificate Transparency
- An OCSP staple

Typically 6 signatures
and 2 public keys
when visiting a website.

(And we're not even counting DNSSEC.)

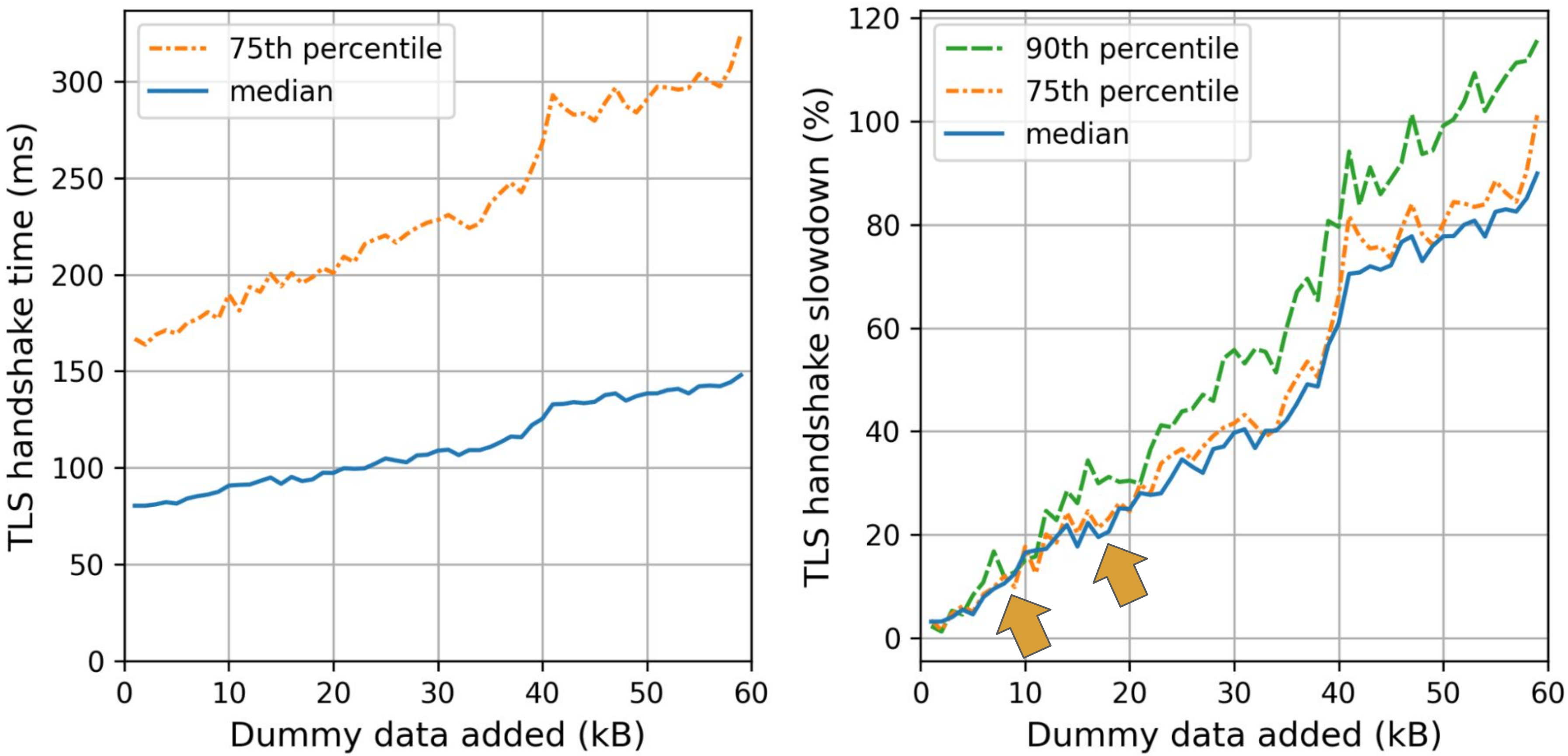
Using only Dilithium2

+17,144 bytes

Using Dilithium2 for the TLS handshake and Falcon for the rest

+7,959 bytes

Is that too much? We had a look…



blog.cloudflare.com/sizing-up-post-quantum-signatures, 2021

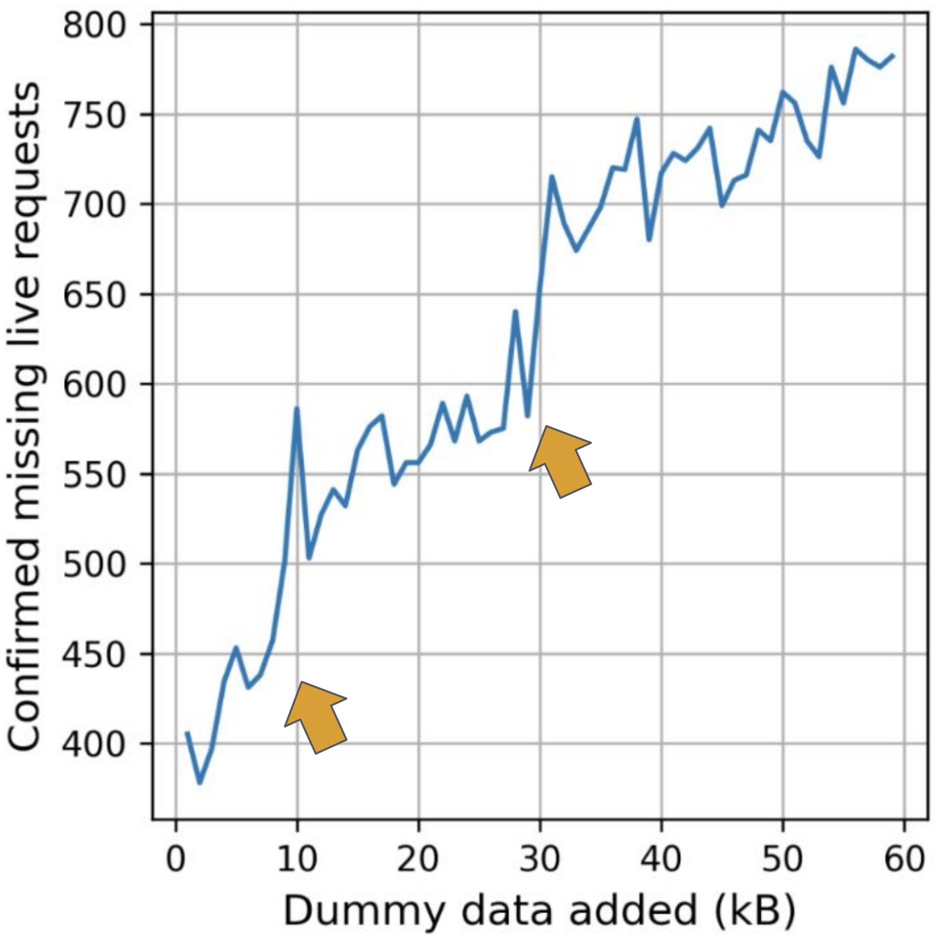
And, of course…

Protocol ossification

Bump in missing requests suggests some clients or middleboxes do not like certificate chains longer than 10kB and 30kB.

This is problematic for composite certificates.

Instead configure servers for multiple separate certificates and let TLS negotiate the one to send.



Not great, not terrible

It probably won’t break the Web, but the performance impact will delay adoption.

NIST signature on-ramp

NIST took notice and has called for new signature schemes to be submitted.

I will cover these later on.

The short of it: there are some very promising submissions, but their security is as of yet unclear.

Thus, we cannot assume that a new post-quantum signature will solve our issues.

In the meantime

There are small and larger changes possible to the protocols to reduce the number of signatures.

- Leave out intermediate certificates.
- Use key agreement for authentication.
- Overhaul WebPKI, eg. Merkle Tree Certificates.

I will discuss these in more depth later on.

Signatures 🖋️

Less urgent, but path is unclear. Real risk we will start migrating too late.

That's not all: the Internet isn't just TLS

There is much more cryptography out there with their own unique challenges.

- DNSSEC with its harder size constraints
- Research into post-quantum privacy enhancing techniques, eg. anonymous credentials, is in the early stages.

Questions so far?

Coping with post-quantum signatures

Recall: there are many signatures on the Web

- Root on intermediate
- Intermediate on leaf
- Leaf on handshake
- Two SCTs for Certificate Transparency
- An OCSP staple

Typically 6 signatures
and 2 public keys
when visiting a website.

Not all signatures are equal

The TLS handshake signature is created on-the-fly (online) and is transmitted together with its public key.

The handshake signature benefits from balanced signing/verification time, and balanced public key/signature size.

The other signatures are offline, and can trade signing time for better verification time. The intermediate's signatures are sent with their corresponding public key, and the rest (SCT/OCSP staple) without public key.

The former benefits from balanced signature/public key size. For the latter it's beneficial to trade public key and signature sizes.

			Sizes (bytes)		CPU time (lower is better)	
		PQ	Public key	Signature	Signing	Verification
Standardised	Ed25519	❌	32	64	1 (baseline)	1 (baseline)
	RSA-2048	❌	256	256	70	0.3
Hash-based	XMSS* w=256 h=20 n=16	✅	32	608	6 ⚠️	2
NIST drafts	Dilithium2	✅	1,312	2,420	4.8	0.5
	Falcon512	✅	897	666	8 ⚠️	0.5
	$SPHINCS^{+}$128s	✅	32	7,856	8,000	2.8
	$SPHINCS^{+}$128f	✅	32	17,088	550	7
Sample from signatures onramp 	$MAYO_{one}$	✅	1,168	321	4.7	0.3
	$MAYO_{two}$	✅	5,488	180	5	0.2
	SQISign I	✅	64	177	60,000	500
	UOV Is-pkc	✅	66,576	96	2.5	2
	HAWK512	✅	1,024	555	2	1

Concrete instances with NIST drafts

Using Dilithium2 for everything adds 17kB.

Using Dilithium2 for handshake and Falcon512 for the rest, adds 8kB.
⚠️ Fast and secure Falcon512 signing is hard to implement.

Using SPHINCS$^+$-128 for everything adds 50kB. Order of magnitude worse signing time than RSA. Most conservative choice.

Stateful hash-based signatures

Using XMSS$^{(MT)}$ with w=256, n=128, two subtrees for SCTs and intermediates, and single tree for the rest, and Dilithium2 for handshake signature, adds 8kB.

⚠️ n=128 and w=256 instances are not standardised.

⚠️ We lose non-repudiation.

⚠️ Large precomputations/storage required for efficient signing.

⚠️ Challenging to keep state.

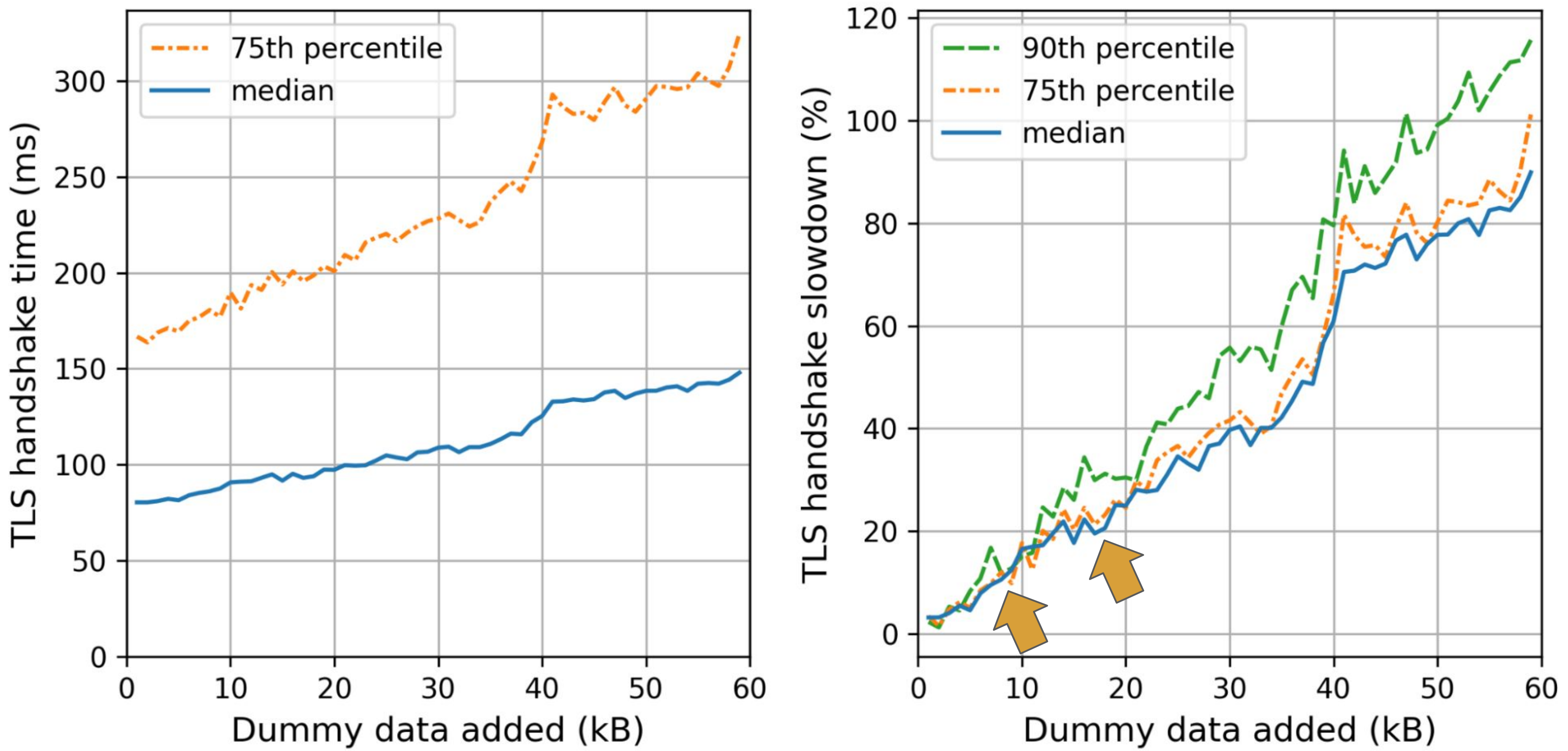
Concrete instances with on-ramp candidates

Using MAYO *one* for leaf/intermediate, and *two* for the rest, adds 3.3kB. Signing time between ECC/RSA. ⚠️ Needs more cryptanalysis.

Using UOV ls-pkc for root and SCTs, and HAWK512 for the rest, adds 3.2kB. 66kB for stored UOV public keys. HAWK relies on Falcon assumptions and then some more.

Using UOV ls-pkc again, but combined with Dilithium2. Adds 7.4kB. Relatively conservative choice.

SQIsign only. Adds 0.5kB. Signing time >1s (not constant-time), and verification time >35ms. 🐢



blog.cloudflare.com/sizing-up-post-quantum-signatures, 2021

Leaving out intermediates

Most browsers ship intermediate certificates, so why bother sending them?

Leaving out intermediates

Three proposals:

- 2019, draft-kampanakis-tls-scas, send flag to indicate server should only return leaf. Simple but error prone.
- 2022, draft-ietf-tls-cert-abridge, replaces intermediates with identifiers from yearly updated central list from CCADB. Client sends version of latest list. Also proposes tailored compression.
- 2023, draft-davidben-tls-trust-expr. Simplified: client sends which trust store it uses, and the version it has. CA adds as metadata to a certificate, in which trust store (version) it's included. Trust stores can then add intermediates as roots.

Gains leaving out intermediates: median 3kB

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

From Dennis Jackson's draft-ietf-tls-cert-abridge-00

KEMTLS (aka. Authkem)

Use KEM instead of signature for handshake authentication.

KEMTLS

Replacing Dilithium2 handshake signature with Kyber512 saves 2.9kB server → client, but adds 768B in the second flight client → server.

At the moment gains are modest. Interesting for embedded, to reduce code size by eliminating primitive. Client authentication with KEM requires extra roundtrip.

Large change to TLS. Subtle changes in security guarantees. We have a formal analysis.

Proof-of-possession unclear. Could be done with lattice-based zero-knowledge proofs or challenge-response.

Merkle Tree Certificates

Pain-points of current WebPKI

OCSP is expensive to run, whereas majority of users don’t use it, but rely on CRL instead (via eg. CRLite).

Too many signatures.

Certificate Transparency is difficult to run.

Many sharp edges: path building, punycode, constraint validation, etc.

(Domain control validation is imperfect — not addressed.)

Changing the WebPKI

With the post-quantum migration, the marginal cost of changing the WebPKI is lower than ever.

There is a huge design space, with many trade offs.

Merkle Tree Certificates (MTC) is a concrete, ambitious, but early draft. We're looking for feedback on the design and general direction.

Not a complete replacement for current WebPKI: it's an optimisation of the common case and falls back to X.509+CT.

Merkle Tree Certificates in short (1)

On a set time, eg. every hour, the CA publishes:

- The batch of assertions they certify. All assertions in a batch are implicitly valid for the same window, eg. 14 days. For each batch, the CA builds a Merkle tree on top.
- A signature on the roots of all currently valid batches.

Trust Services (eg. browser vendors) regularly pull the latest batches and window signatures from CAs, verify them for consistently, and only send the Merkle tree roots to the browsers.

Merkle Tree Certificates in short (2)

A Merke tree certificate is an assertion together with a Merkle authentication path to the root of the batch.

A server would install three certificates: two Merkle tree certificates 7 days apart, and a fall back X509 certificate.

When connecting to a server, the client sends the sequence number of the latest batches it knows of each MTC CA.

If the client is sufficiently up-to-date, the server can return one of the Merkle tree certs, and otherwise will fall back to X.509.

Merkle Tree Certificates sizes

There are currently 1 billion unexpired certificates in CT.

If reissued every 7 days by one MTC CA, we'd have batches of 6 million assertions.

That amounts to authentication paths of 736 bytes, and with a Dilithium2 public key a typical Merkle tree certificate will be well below 2.5kB, smaller than only the median compressed classical intermediate certificate of 3.2kB.

Try MTC for yourself: PoC MTC CA.

Wrapping up

We saw several different approaches to cope with large post-quantum signatures, from simple to ambitious.

There are still many unknowns: among others, compliance requirements; cryptanalytic breakthroughs; ecosystem ossification; stakeholder constraints; etc.

Which approach to take? I'd say it's good to have multiple pots on the stove.

Thank you, questions?

References

- pq.cloudflareresearch.com
- Follow along at the IETF
- Check out our recent blogpost *the state of the post-quantum Internet*, and Google's take.
- Reach out: ask-research@cloudflare.com