Timings are in!

Finally we are at a stage where we can consider empirically the impact of “going post-quantum” on TLS, particularly in the IoT context. Recall the current assumption that that the protocol itself does not change, but that plausible Post Quantum candidates are simply substituted for their pre-quantum equivalents.

To make the comparison interesting and concrete we will consider a situation where the lattice-based Dilithium and Kyber primitives are substituted for standard 256-bit elliptic curves. In our case Kyber768 substitutes for the X25519 curve used for the key exchange, and Dilithium3 substitutes for the secp256r1 elliptic curve commonly used for digital signature. In the IoT setting small elliptic curves were particularly attractive, as key sizes were minimal. With PQ crypto we have to surrender that advantage as explained earlier.

Another consideration is that of client authentication. Now this is rare for standard Web traffic, but much more commonly used in the IoT setting, so we will consider timing with and without client side authentication included in the protocol. (Note that Dilithium signatures take a wildly varying amount of time, as it makes multiple attempts to generate a secure signature).

Hardware support may be available for elliptic curve cryptography, and highly optimized assembly language implementations have been written over the years for many processors. So we will include such a scenario as well. This is a little unfair on the PQ setting, as over time hardware support and assembly language code will appear for popular processors. Note that for our lattice based primitives hardware support for SHA3 would make a huge difference, as they spend most of their time hashing.

Since certificates still play a large part, we will assume a standard certificate chain of length 3 for the server, and a single self-signed cert for the client (where the associated private key may be inside of protected hardware). This implies that 3 verifications will be needed by a client to verify a full server certificate chain. In fact more may be required for longer chains, and for support for Certificate Transparency and OCSP stapling – but for now we just need 3.

First thing to note is that the dynamic memory requirement will inevitably increase. For a small ECC based implementation we find that 17K of RAM suffices. But for the PQ implementation this rises to around 58K. That is bad, but not a show stopper, as the memory resourcing of IoT nodes has improved a lot over the years. Certainly its not a problem for our two chosen devices.

But as regards timings just how bad is it going to be? One problem we have it that it is not at all clear exactly when TLS finishes and application traffic commences, as application data may be intermingled with the protocol handshake. Again to be concrete we assume that a full handshake terminates when the server has issued its resumption tickets, received a short application message from the client, and returned a short message of its own. For a resumption handshake the client sends the same short message as “early data” and is finished after it receives the same short message reply from the server.

The full results are below, and I am still digesting the implications myself. Consider only the RP2040 device. Here the starkest comparison is between what we had gotten used to – elliptic curves with hardware and assembly language support - against portable software lattice-based crypto.

With client authentication (and recall Dilithium signatures are slow)
An ECC full handshake takes 646 ms, and a resumption takes 207 ms.
A PQ full handshake takes an average of 1170 ms, and a resumption takes 351 ms.

Without client authentication
An ECC full handshake takes 584 ms, the resumption is unchanged.
A PQ full handshake takes an average of 806 ms, the resumption is unchanged.

You can of course draw your own conclusions. My conclusion is that once H/W and assembly language support for lattice based crypto starts appearing there will not really be much difference in the timings, and hence on user experience.

TinyPICO – ESP32 - 240MHz

Post Quantum Portable software only, Maximum Stack w/o client auth 54K, w client auth 58K

	Time(ms)	Time(ms)	Time(ms)	Time(ms)	Time(ms)
Full H/S	398	422	375	436	389
Full H/S + Client Auth	764	790	540	782	623
Resume+Early data	333	343	468	343	343

Kyber768 Key generation + Secret Share – 27ms
Dilithium3 Verification – 40ms (x3)
Dilithium3 Signature - 126/472/208/133/639ms

ECC Portable software only, Maximum Stack w/o client auth 17K, w client auth 17K

	Time(ms)	Time(ms)	Time(ms)	Time(ms)	Time(ms)
Full H/S	890	985	972	865	951
Full H/S + Client Auth	984	1017	1004	1004	940
Resume+Early data	202	165	183	182	170

X25519 Key generation + Secret Share – 59ms
ECDSA Verification – 109ms (x3)
ECDSA Signature – 97ms

RP2040 Connect – ARM M0+ - 133MHz

Post Quantum Portable software only

	Time(ms)	Time(ms)	Time(ms)	Time(ms)	Time(ms)
Full H/S	795	855	776	836	769
Full H/S + Client Auth	1028	1144	1523	1051	1102
Resume+Early data	326	421	368	322	318

Kyber768 Key generation + Secret Share – 51ms
Dilithium3 Verification – 110ms (x3)
Dilithium3 Signature – 1467/275/528/1748/325ms

ECC - Portable Software Only

	Time(ms)	Time(ms)	Time(ms)	Time(ms)	Time(ms)
Full H/S	1699	1615	1756	1637	1741
Full H/S + Client Auth	2051	1933	1961	1888	2057
Resume+Early data	325	391	303	303	287

X25519 Key generation + Secret Share – 199ms
ECDSA Verification – 402ms (x3)
ECDSA Signature – 333ms

ECC - Assembly (X25519) + Hardware (ECDSA)

	Time(ms)	Time(ms)	Time(ms)	Time(ms)	Time(ms)
Full H/S	494	543	630	627	625
Full H/S + Client Auth	612	649	652	662	655
Resume+Early data	144	146	215	310	221

X25519 Key generation + Secret Share – 50ms
ECDSA Verification – 68ms (x3)
ECDSA Signature – 110ms