On TLS for the Internet of Things, in a Post Quantum world

Michael Scott

Cryptography Research Centre
Technical Innovation Institute
michael.scott@tii.ae

Abstract. The TLS (Transport Layer Security) protocol is the most important, most attacked, most analysed and most used cryptographic protocol in the world today. TLS is critical to the integrity of the Internet, and if it were to be broken e-commerce would become impossible, with very serious implications for the global economy. Furthermore TLS is likely to assume even greater significance in the near future with the rapid growth of an Internet of Things (IoT) – a multiplicity of internet connected devices all engaged in secure inter-communication. However the impending invention of a Cryptographically Relevant Quantum Computer (CRQC) would represent an existential threat to TLS in its current form. As it stands the latest version TLS1.3, benefiting as it does from years of research and study, provides effective security, but it must soon be updated to resist this new threat. In this research we first undertake a new clean-room implementation of a small-footprint open source TLS1.3, written in C++ and Rust, and suitable for IoT applications. Our implementation is designed to be cryptographically agile, so that it can easily accomodate new post-quantum cryptographic primitives. Next we use this new implementation as a vehicle to study the impact of going post-quantum, with a particular emphasis on the impact on the Internet of Things. Finally we showcase the flexibility of our implementation by proposing an implementation of TLS that uses identity-based encryption to mitigate this impact.

1 Introduction

The TLS protocol is executed by a web browser every time a website is accessed on the internet, unless its one of those rapidly diminishing number of websites (HTTP and not HTTPS) that implement no security. After the protocol is run, an encrypted tunnel is established between client and server, and the server's identity is authenticated to the client. Commonly client authentication, if required, happens at the application layer, where typically the client enters a username and password. However TLS is also relevant in non-browser contexts, and for many of those TLS can also take responsibility for client-side authentication. TLS authentication uses public key cryptography and exploits the Public Key Infrastructure (PKI).

The full official TLS 1.3 specification is provided in RFC8446. A more readable version is provided by Wong [29].

Let us briefly recap how PKI works in the context of TLS. Typically a website or TLS server is equipped with a three link certificate chain, consisting of a leaf certificate, an intermediate certificate, and a root certificate. The root certificate is self-signed, and is issued by a recognised Certificate Authority (CA). Each certificate has embedded within it a public key that can be used for signature verification, and the entity that owns the certificate has in its possession the associated private signing key. The leaf certificate is created by the website owner, and is digitally signed by the owner of the intermediate certificate. The intermediate certificate in turn is signed by the Certificate Authority (CA). Browsers maintain a store of trusted CA root certificates – there are a limited number of them. To be precise the certificate store provided with a recent Ubuntu distribution had exactly 138 root certificates. The purpose of the intermediate certificate is to limit the exposure of the CA's signing key. Longer certificate chains are possible, but rare. A website's identity, embedded in the leaf certificate, is verified when it provides a digital signature on some known data, which can then be verified using the embedded public key. The validity of the leaf certificate is verified by checking the certificate chain, that is verifying the signatures on each certificate using the relevant public key extracted from the chain. Note that the signature on the self-signed root certificate is commonly not checked – its presence in the root certificate store is sufficient to validate it.

1.1 A very brief history of TLS

It was certainly serendipitous that the requirement for transport security in the context of the internet's evolution, dovetailed nicely with the invention of public key cryptography. By 1995 the SSL standard was established, and after a few iterations it evolved to a state where it could support the deployment of e-commerce on the Internet. As it evolved improved encryption primitives were rotated in, and older, often broken, primitives rotated out. However over the years a lot of cryptographic "clutter" accumulated, which often proved difficult to eliminate, largely due to the need to maintain continuity, and to support backward compatibility. However the blame here lies largely with PKI rather than SSL/TLS. Clearly any kind of identifying credential like a certificate chain needs to have a useful lifetime, and PKI certificate chains often have validity periods lasting for decades.

By 1999 SSL had progressed to version 3.0, after which it underwent a name change and was succeeded by TLS 1.0. Further development culminated in version 1.3 launched in 2017. Fortunately SSL/TLS development mostly managed to stay ahead of many determined efforts to break its security (most notably the HeartBleed hack [15]), and since the arrival of TLS 1.3 cryptanalytic attacks have become rarer and of less significance. Furthermore the development of proofs of security has helped to cement confidence in the protocol in its most recent manifestation [8].

1.2 TLS 1.3

In TLS version 1.3 a client typically starts the protocol by sending an “Hello” message to a previously unvisited website. This will include a random key share for a Diffie-Hellman key exchange in the clients favoured elliptic curve or finite field group. The server will respond with its own “Hello” which, if it approves of and supports the chosen group, will include its key share in the same group. Note that if the server is not happy with the client choice of group, it may invoke a Handshake Retry Request (HRR), indicating its unhappiness with the client’s choice, and urging it to try again.

Assuming agreement is reached, the Diffie-Hellman key exchange completes, and both client and server now share unauthenticated keying material, from which suitable encryption keys for bidirectional communication can be derived, via a suitable Key Derivation Function (KDF). Immediately such handshake keys are available the server uses its key to encrypt the remainder of its “Hello” message and all subsequent protocol messages – the idea being to hide protocol messages as soon as a mutual key becomes available.

Next the server transmits its certificate chain to the client, followed by a digital signature applied to a hash of the entire communications transcript up to this point. When the client verifies this signature, using the public key extracted from the leaf certificate in the chain, then this has the effect of authenticating the server’s previously negotiated encryption key. Finally the server transmits to the client a Hash-based Message Authentication Code (HMAC), calculated on the communications transcript to date using the agreed key.

Now it is the client’s turn. If it is authenticating it may at this stage choose to send its own certificate chain plus signature. Otherwise it just transmits its HMAC on the transcript to the server, and we are done.

Note that this is a 3-pass protocol. It is a variant of a so-called SIGMA-I protocol of a type originally proposed by Krawczyk [18], [19]. Observe that the server bundles its “Hello”, certificate chain, signature and final HMAC into a single transmission. At the end of the process both parties share traffic encryption keys which give effect to the encrypted tunnel that now exists between them. The encrypted tunnel uses a cipher in a modern AEAD mode of operation to safely encrypt all subsequent application traffic in both directions.

The full TLS 1.3 protocol (assuming a Handshake Retry Request is not required) is illustrated in figure 1. Unencrypted communication is shown in black, encrypted communication in red. Optional components are shown dashed.

1.3 Going Post-Quantum

In a Post-Quantum world (meaning a world where CRQCs are widely deployed) the current methods for both key exchange and signature become obsolete, and must be abandoned. The simplest approach to going post-quantum is to simply substitute in newly standardised post quantum cryptographic primitives. But this will have consequences, which have been the subject of much recent research activity, see for example [12], [11].

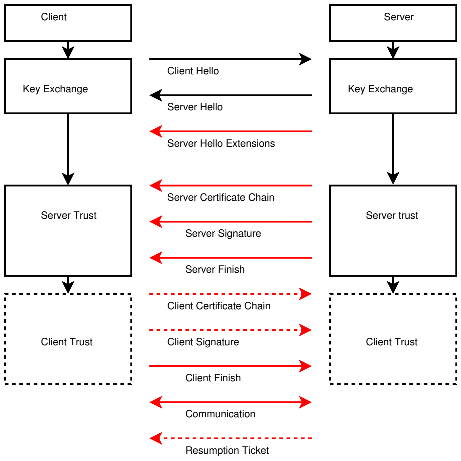

Fig. 1. Full TLS Protocol (simplified)

Primarily there will be an impact on

- execution time
- dynamic memory requirement
- bandwidth

Given the much larger public key and signature sizes associated with the soon to be standardised post-quantum methods, the amount of data transferred between client and server is going to blow up substantially. There will clearly be a price associated with this. This bandwidth issue has already been explored experimentally by Westerban [28], and indeed in some settings it may lead to insurmountable problems. However in this paper the focus will be more on the first two issues, which are of particular concern in the context of the Internet of Things (IoT).

One implication of going post-quantum is that the classic key exchange will most likely need to be replaced by a KEM (Key Encapsulation Method). As it turns out for TLS this is mostly a technical issue which will not effect the protocol as described. In a classic Diffie-Hellman key exchange both parties can simultaneously swap ephemeral values, and then calculate a mutual key. Using a KEM the client generates an ephemeral key pair and sends the public key to the server, who uses it to encapsulate a random key which is sent back to be client to be decapsulated using the private key. In all cases ephemeral values are deleted after use. In TLS the Diffie-Hellman exchange does not take place simultaneously, as the client share is sent in the client Hello, and the server response comes back in the server Hello. So the basic flow remains the same in both cases, and the outcome is a shared, and as yet unauthenticated, mutual key.

It is commonly recommended that a wiser path for transition to post-quantum cryptography will be to initially use a so-called hybrid approach. The intuition is simple, perform key exchange and signature using both pre and post quantum methods. If either fails, then that registers as an overall failure. In the key exchange context the agreed key is an amalgam of keys generated using both methods [26]. This approach mitigates against either the invention of a CRQC, or the cryptanalysis of the newer post-quantum primitives at any time prior to such an invention.

2 TiigerTLS

As a first step it was decided to write a clean-room open-source implementation of TLS 1.3, while abandoning any ambitions to maintain backward compatibility with earlier versions. This allowed us to ruthlessly clean out all deprecated cryptographic primitives (a process already encouraged in version 1.3), and also obviously greatly simplified the software development process. As a new implementation we also had the luxury to choose our preferred programming language. In line with a growing trend among crypto developers we decided to implement both client and server in the Rust language. Rust has the benefit of catching more bugs at compiler time, rather than at runtime, and is particularly strong for preventing the kind of buffer overflow attacks which have caused problems for earlier implementations of SSL/TLS [15]. However since Rust is not currently ideal for small embedded environments, it was also decided to develop a compatible TLS 1.3 client in C++.

While wanting to minimize the footprint of both server and client, our assumption was that it would be the client which would most often need to fit into the smaller space. The single largest dynamic memory requirement in TLS is for an input buffer to receive protocol communications. Nonetheless the overall requirement can still be kept quite small, as cryptographic primitives are not usually major consumers of memory. Therefore it makes sense to allocate memory exclusively from the stack where possible, rather than from the heap. As can be seen from figure 1, from the client point of view, the protocol can be divided into three main phases (key exchange, authenticating the server, and optionally authenticating the client) each of which can then automatically deallocate much of its memory from the stack when finished, to allow memory to be re-used by the next phase. The biggest consumer in our experience are those phases required to process a certificate chain.

Using stack allocation implies that maximum fixed sizes must be assumed for all allocated arrays. Unfortunately such maximums are often not dictated by the TLS standard, so worst case assumptions must be made. This in turn implies that on occasion our implementation will fail when attempting interoperation against other TLS implementations. However we have no ambitions to write a browser, as our implementation is mainly targeted at non-browser IoT applications, and maximum sizes can be adjusted where needed at compile time for each particular application.

Our implementation is open sourced and can be accessed here [1]. Read the multiple blogs for some unfiltered commentary on the issues that arose when implementing TLS1.3 from scratch. When the C++ and Rust clients are tested against the TLS Anvil framework [20], they currently achieve a compatibility rating of 98.2% at strength 3. The Rust server achieves 96.81% compatibility also at strength 3. These results compare favourably with competitive products [20].

3 Agility

In a changing cryptographic landscape a key notion is that of Agility – that is the ability to quickly switch cryptographic primitives with minimal disruption to the protocol implementation. Such agility should ideally be designed-in from the start. Unfortunately TLS has a particularly poor history in this regard, and in many consumer IoT products it is implemented in a way that lags behind current best practise. In particular it is often not regularly updated, which can in part be put down to a lack of agility in the design [22].

To achieve agility in our implementation we use a SAL, a Security Abstraction Layer. The concept is simple – all cryptography is implemented inside of the SAL. Hence in a context where a primitive needs to be replaced, only the SAL is impacted.

This strict division between the protocol state machine and the cryptography that is uses is baked into our implementation of TLS 1.3. To be concrete, in our C++ client the SAL is implemented by the files `tls_sal.h` and `tls_sal.cpp`. The fixed and unchanging SAL API is defined in `tls_sal.h`. The implementation in `tls_sal.cpp` is however left to the developer. It will probably source its cryptography from one or more open or closed resources. In our implementation the default SAL uses the MIRACL Core library [2], as it supports most of the cryptography required by TLS1.3 in both C++ and Rust, and supports both 32 and 64 bit architectures. But we also provide a SAL which mixes functions from MIRACL and the well known libsodium library[3], to exploit its nice random number generator, and its fast and secure implementations of the TLS1.3 supported CHACHA20 cipher, and the X25519 elliptic curve. Yet another SAL exploits hardware support for random number generation and elliptic curve cryptography where it is available.

With a structure that has agility as part of its design, it is reasonable to assume that a switch to post quantum primitives could take place with minimal disruption. Indeed the key exchange phase of TLS can already be regarded as agility-friendly. The client and server Hellos simply need to update their lists of acceptable primitives, and old primitives will be deprecated and a new primitive will automatically be agreed. However as indicated above it is not so simple when it comes to updating PKI. The ideal solution would be to issue every web

[1] `https://github.com/Crypto-TII/TLS1.3`

[2] `https://github.com/miracl/core`

[3] `https://github.com/jedisct1/libsodium`

connected device with an updated certificate chain. However in a setting where long-lived authentication credentials are used, then major disruption seems to be inevitable in the face of a catastrophic failure of the underlying cryptography, as would be the case if CRQCs suddenly became widely available. Hopefully updated credentials can be rolled out well in advance of such an event. Here we can regard this as primarily an issue for PKI, rather than as one that a TLS implementation can solve [16].

4 Trust Management

A big component of TLS is the authentication of the identities of the two parties involved. At a minimum the true server identity must be established, while the client may, for now, remain anonymous. Authentication is established by a party processing the proffered certificate chain and establishing its authenticity. Therefore the Trust Management API is conceptually quite simple – it inputs a certificate chain along with the claimed identity, and returns true or false. However certificate processing is potentially quite a complex procedure, and so its makes sense to isolate this functionality behind another abstracted layer, a Trust Management Layer (TML). Note that the name does not preclude the possibility of, in the future, implementing an alternate authentication mechanism other than PKI.

The TML will need access to the SAL, as it will require some cryptographic functionality. If using PKI, cryptographic hashes and digital signatures will need to be calculated and verified.

In our implementation we currently deploy a very simple TML which does basic certificate chain testing, which may be sufficient for certain IoT applications. It does not support certificate revocation or the important mechanisms of Certificate Transparency and Certificate Stapling. A more functional TML is under active development.

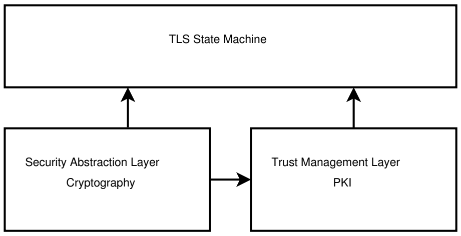

Fig. 2. TLS program structure

The overall implementation structure of both client and server can be simply visualised as in figure 2.

5 Session Resumption

As can be seen in figure 1 at any time after the secure TLS1.3 tunnel is established, the server may issue the client with a "resumption ticket". Basically the server takes a snapshot of its current cryptographic state (keys etc), encrypts it with a long term Session Ticket Encryption Key (STEK), and sends it over to the client for safe keeping. The idea being that if in the future the client wants to connect again to the same server, it starts by handing back this ticket. This simple mechanism allows the server, if it wishes, to remain stateless. That is it does not need to remember anything about prior client connections – everything it needs for a safe reconnection is stored in the ticket. And since the keys used are already authenticated, a resumption handshake does not require either the server or the client to resubmit its certificate chain. So a resumption handshake is potentially much quicker. As a bonus, since mutual keys are already available from the very start, the client may send encrypted "early data" in its first protocol pass, potentially improving the client experience by allowing the server to be more immediately responsive. See figure 3

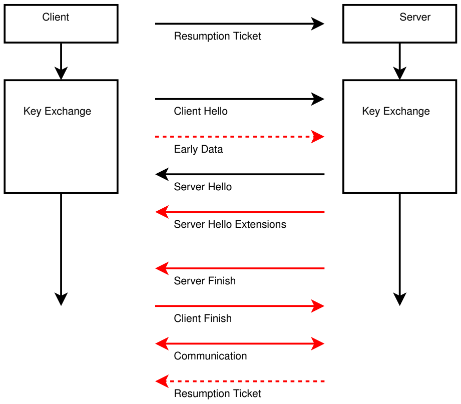

Fig. 3. Resumption TLS Protocol (simplified)

Observe that after a resumption handshake, yet another resumption ticket can be issued. TLS 1.3 has in effect adopted the popular idea of the double ratchet, as made famous by the popular Signal messaging application.

There are some subtleties to bear in mind. First tickets are not valid forever, indeed the maximum permissible lifetime in TLS 1.3 is just one week, and most often they are issued with lifetimes much shorter than that. Also if the server remains stateless, there is nothing to prevent a client from re-using the same ticket indefinitely – one reason for limiting its lifetime. Observe that we have kept the key exchange phase of the protocol, to allow new random key material to be mixed in, and hence to maintain forward secrecy, a classic requirement for an effective ratchet. However this does not take effect in time for the "early data" and hence care should be exercised over what is transmitted. The length of the early data is limited, and to be safe only relatively innocuous data should be transmitted here (like a HTTP GET message used to fetch a web page's HTML content).

Despite some valid privacy concerns as to how session resumption may be abused to track client behaviour [27], with careful use we regard session resumption as being a valuable feature of TLS. In our experience the majority of the websites that support TLS 1.3 now also support session resumption. In an Internet of Things context its reduced overhead compared to the full handshake, is particularly welcome.

6 The TLS State Machine

One major implication of "going post-quantum" is that public keys and/or signatures will get much larger. This is an unfortunate feature of all proposed post-quantum methods. If we simply use post-quantum primitives as drop-in replacements for existing methods, then this will have a major impact on the sizes of the protocol messages that will be exchanged. In particular certificate sizes will increase, maybe getting 10 times bigger. Which raises a valid and legitimate question – the TLS state machine was designed in the context of much smaller public keys and signatures. So maybe it is no longer optimal in a post-quantum world, and maybe it should therefore be modified.

Just such an idea is suggested by KEMTLS [24]. In this paper the authors suggest a way in which the transcript signature can be replaced by a KEM, as post-quantum KEMs (like Kyber [4]) are much smaller than post-quantum signatures (like Dilithium [9]). However in the case where client authentication is required this introduces an extra flow into the protocol, which is quite a high price to pay, although mitigations have been proposed [25].

However KEMTLS does not eliminate the bulk of post-quantum signatures and public keys that are bound up inside of certificate chains and which will still need to be processed. It is also not relevant to resumption handshakes. Given the effort made to agree the TLS1.3 specification we feel it unlikely that such a change will be acceptable to the community. It is therefore our view that the most likely way forward will not involve changes to the state machine.

7 Implementing TLS1.3 in a constrained environment

Unfortunately TLS1.3 is itself an unconstrained protocol. Therefore size limitations must be imposed if it is to fit inside of a constrained environment, like an IoT node. So the obvious next question must be – how constrained is that? And the less than satisfactory answer must be – it depends. In reality it may not be possible to fit TLS into the tiniest of devices, especially when it is considered that TLS is probably just a small part of what must be fitted inside of the available resources. The application itself has a legitimate expectation that it it will be able to make use of the bulk of those resources. See [1] for a discussion of these issues.

Programmers can often surprise us with their ability to squeeze cryptographic protocols into the smallest of spaces (while often ignoring the requirements of the actual application), but at some point this comes at the cost of poor performance or limited functionality. In [13] it is suggested that “An IoT device may only have 256 or 512kb of RAM and often need to conserve battery”, and this is the type of device that we choose to target here. We find that we can implement (almost) the full TLS1.3 into such devices, while only consuming a small fraction of the available resources. To achieve this constraints must be imposed on larger data objects like certificates, but we find that such constraints are not very limiting in practise.

For our experiments we will use two typical contemporary IoT node devices. See figure 4. The first device is the Arduino Nano RP2040 Connect, which uses the Raspberry Pi Pico microcontroller. This has 264kb of SRAM and up to 16MB of flash memory. It uses a dual core ARM M0+ processor clocked at 133MHz. It supports WiFi and bluetooth, has low power modes, and has a built-in ATECC608A crypto co-processor [17].

The second device is the even physically smaller TinyPICO. It uses a dual core Xtensa LX7 processor clocked at 240MHz and has 4MB of flash memory and up to 4MB of SRAM. It also supports WiFi. In its low power mode it might consume as little as 20uA.

Given the generous allocation of flash memory for both devices, we found that code size was not a significant constraint. It was practical therefore for the client executable to include a full database of root certificates.

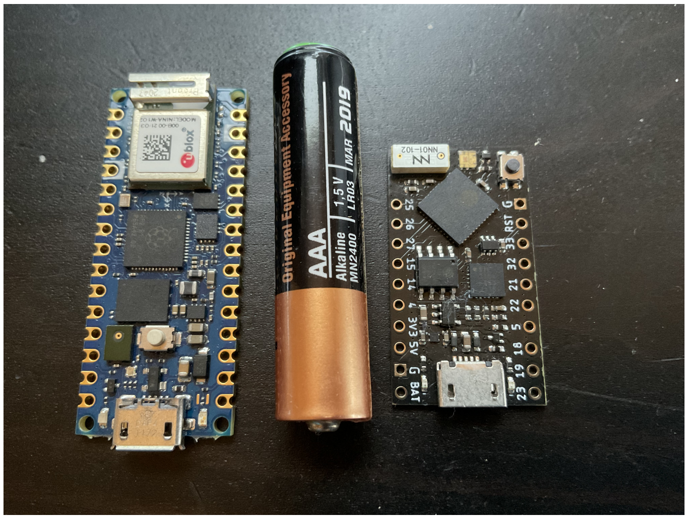

Fig. 4. RP2040 Connect, AAA battery, TinyPICO

8 Results

Both devices are supported on the well-known Arduino Platform. In all cases our Rust TLS server ran on a PC, and the C++ client ran on the IoT device. They communicated wirelessly using standard WiFi over a local area network. Protocol progress could be observed on the Arduino serial monitor and in a PC command window. Since we control both sides of the protocol we can generate our own certificate chains, and add our own root certificate to the root certificate stores. One minor isue arises – it is not entirely straightforward to determine when TLS finishes and the application takes over the communications channel. Handshake and application data can overlap. So here we assume that a TLS handshake has completed after the client Finish message, but before any resumption tickets are transmitted.

First we measure the bandwidth requirement as observed by the client, in terms of the number of bytes sent and received in order to complete the full handshake and resumption handshakes. These results must be considered as indicative, as they obviously depend on the variable lengths of certificate chains, random padding etc.

If client authentication is to be supported in the context of a resumption handshake, then the client identity must be embedded in that part of a resumption ticket which forms the pre-shared key identity which is transmitted back from client to server in the pre-shared key extension in its client Hello. This could be a full copy of the clients X.509 certificate chain, or just a subset, an option alluded to in RFC5077. Here we choose just to include the client's authenticated identity. This reduces the extension size significantly.

	Bytes read	Bytes sent/no client auth	Bytes sent/client auth
ECC Full	1059	365	1157
ECC Resume	267	508	534
PQ Full	15632	1519	10425
PQ Resume	1316	1651	1663
HYBRID Full	12034	1553	8294
HYBRID Resume	1359	1667	1691

Table 1. Bytes sent and received by client

8.1 Pre-Quantum TLS 1.3

Our first measurements were taken in a scenario where standard Elliptic Curve Cryptography (ECC) methods were the preferred technology for both key exchange and certificate public keys and signatures. This is a setting very favourable for constrained IoT deployments, as ECC key sizes and signatures are close to being the smallest possible. For key exchange the client favoured the well known X25519 curve, which was also supported by the server. The ECDSA algorithm was used for the transcript signature using the secp256r1 standard elliptic curve and the certificate chain used the same algorithm throughout. The cipher suite negotiated by the initial exchange of Hellos was TLS_AES_128_GCM_SHA256 [29]. The idea was to compare the impact of going post-quantum against this best case scenario.

Fortuitously the TinyPICO reports the stack high water mark, and from this it is quite easy to obtain a good estimate of stack memory usage. In this case it was just less than 16kb. As remarked above one major memory requirement is for an input buffer to hold protocol messages. Since ECC based certificates can be very small, we found that this input buffer could be safely reduced to just 4kb, which helped to keep the stack requirement relatively modest.

The Arduino Nano RP2040 uses the well known ARM M0+ processor and it was therefore possible to make use of Thomas Pornin's fast assembly language implementation of X25519 [23]. The Nano RP2040 also has a crypto co-processor which does fast hardware ECDSA using the same secp256r1 curve. The device was initialised to generate its own secret signing key inside of its secure memory, and to emit a self-signed certificate to allow it to authenticate as a client. For this device it was therefore possible to create an optimised tailored SAL to make full use of all of these resources. An alternative SAL using only portable C++ software was also available using the MIRACL library. The MIRACL library also provided support for all of the other less time-critical cryptographic primitives.

The TinyPICO uses a much less well known processor for which we could not find optimal assembly language implementations, and it does not support a crypto co-processor. So for this device only the fully portable SAL was viable.

Timings are given in table 2. Note that 3 ECDSA signature verifications are calculated by the TLS client (one to verify the transcript signature, and two more to verify the certificate chain), and key exchange timings are for both the generation of ephemeral values and the calculation of the shared secret.

Device	Nano RP2040		TinyPICO
Elliptic Curve Cryptography	Optimal	Portable	Portable
Full TLS Handshake	584ms	1609ms	701ms
Full TLS H/S + Client Auth	646ms	1908ms	781ms
Resumption H/S + Early data	251ms	473ms	183ms
X25519 key exchange	51ms	207ms	58ms
ECDSA Verification (x3)	68ms	380ms	110ms
ECDSA Signature	109ms	325ms	98ms

Table 2. Timings using standard ECC cryptography

8.2 Post-quantum TLS 1.3

As indicated above we will just drop-in soon-to-be-standardised post-quantum primitives as replacements for the ECC methods, which are broken in a post-quantum world. The key exchange uses the Kyber768 KEM [4], and for digital signature/verification we will use Dilithium 3 [9]. We use the portable implementations from the MIRACL library, which minimise memory while not compromising on performance. We acknowledge however that a much smaller implementation is always possible given enough programmer ingenuity [5]. Security for both schemes is set at the conservatively chosen AES192 level, which may be overkill, but seems prudent given the relative novelty of these methods. To create a suitable server certificate chain and a client certificate plus private key, we used the tweaked version of OpenSSL as provided by the Open Quantum Safe project [7]. The OpenSSL instructions required to generate a standard 3-link certificate chain are given in the appendix.

As well as a SAL supporting Kyber and Dilithium, our main TLS and TML code needed to be augmented to accept these new primitives. Provisional OIDs have been assigned to the Dilithium scheme, and so our X.509 parsing code has only to be patched to recognise the new OID.

But the elephant in the room is the size of those elephantine Dilithium 3 certificates. Each one is signed and each one embeds a public key. And the public key size is 1184 bytes, and the signature is 3293 bytes. Recall that an elliptic curve public key can be as short as 32 bytes, and a signature 64 bytes.

As expected much more memory is required, just under 58k of RAM. That is bad, but not in our view a show-stopper, certainly not for our chosen devices. Note that Dilithium signing timings vary significantly, as multiple attempts are normally required before a safe signature can be created [9]. So the timings given for signing should be treated with caution.

Device	Nano RP2040	TinyPICO
Post-Quantum Cryptography	Portable	Portable
Full TLS Handshake	771ms	376ms
Full TLS H/S + Client Auth	1435ms	717ms
Resumption H/S + Early data	512ms	347ms
Kyber key exchange	45ms	27ms
Dilithium Verification (x3)	88ms	40ms
Dilithium Signature	560ms	224ms

Table 3. Timings using Post-Quantum cryptography

8.3 Hybrid TLS1.3

Here we follow [26] for the key exchange, combining X25519 with Kyber768. We also calculate all signatures using both the elliptic curve secp256r1 combined with

Dilithium 2. Note that here we use Dilithium at a slightly weaker security level, as we assume that overall security benefits from the use of two signatures. In all cases verification is performed twice, and both must succeed for the signature to be validated. Again we generate our test certificates using Open Quantum Safe [7], which generates certificates and certificate chains in the required format.

Device	Nano RP2040	TinyPICO
Hybrid Cryptography	Portable	Portable
Full TLS Handshake	2006ms	792ms
Full TLS H/S + Client Auth	2528ms	1064ms
Resumption H/S + Early data	685ms	378ms
Kyber key exchange	264ms	109ms
Dilithium Verification (x3)	440ms	131ms
Dilithium Signature	498ms	199ms

Table 4. Timings using Hybrid cryptography

9 Using Identity-Based Encryption

The use of Identity-Based encryption (IBE) in the context of TLS as an alternative to PKI has been suggested by Banerjee and Chandrakasan [2]. Established methods for implementing IBE have been proposed using both classic cryptogaphic pairings [3] and using post-quantum lattices [10].

Using standard PKI, a client must wait to learn the public key of the server prior to sending it any data. The client also needs to have stored somewhere the public key of the certificate authority that validated the certificate chain which conveys the server's public key to the client. The advantage of using IBE is that data can be transmitted immediately using only the known address of the server, as that address, or identity, is its public key. However there is again a need to store a public key associated with the Trusted Authority (TA) that issued the server with its IBE private key. And of course that is also the well-known downside of using IBE – the private key is issued directly to the server by the Trusted authority. However in an IoT context where the TLS server and the TA may have a particularly close and trusting relationship, this key-escrow issue may not be so important.

In [2] it is suggested to modify the full TLS handshake to integrate an IBE-KEM exchange with the ephemeral Diffie-Hellman key exchange between the two parties. Here we suggest a different approach, whereby we exploit the Pre-Shared Key (PSK) feature support offered in TLS1.3. The intuition is that *knowing the identity of the Server, and its TA public value, is equivalent to already possessing a pre-shared key (PSK) which authenticates the server.* In this way IBE can be used in a TLS PSK handshake without making any changes to the state machine. The server recovers the PSK by performing an IBE decryption using its IBE secret key. Since an IBE PSK can be calculated off-line, before the connection is made, its cost can be amortised against other start-up costs by the client. Observe that using IBE there is no extra on-line cryptographic overhead required of a client, over and above that required of a ticket-based resumption.

In the paper [2] it is suggested that client-side authentication can also be achieved using IBE. However the suggested modification to TLS assumes that the client identity is known a priori to the server, and this is not normally the case. Indeed client identity in TLS is only optionally revealed to a server on a clients second pass, which is encrypted using the handshake keys. Only when this identity is known could IBE encryption be used by the server, and we do not see much advantage in this. Note that client authentication using PKI is commonly supported on IoT nodes using hardware (as is the case for the RP2040), including secure memory for PKI secret key storage. And the TLS1.3 standard does support a **post_handshake_auth** extension which allows a client to authenticate after a PSK handshake has concluded [29]. So we will not consider client-side authentication further in the IBE context.

9.1 Pairing-based IBE (BFIBE)

Pairing-based IBE as originally described in [3] does require the expensive calculation of a pairing for both encryption and decryption. However alternative pairing-based IBE schemes do not require a pairing for encryption, if this should be considered an issue [6]. As pointed out above IBE encryption does not contribute to the online TLS connection cost. To implement an ID-KEM-CCA [2] based on pairings we use the de facto standard BLS12381 curve and the Optimal Ate pairing, which approximately provides 128-bit of security. We implemented pairing-based IBE using the **FullIdent** CCA-secure scheme as described in [3]. The ciphertext that encapsulates the PSK is 161 bytes in length (we do not use point compression).

9.2 Post Quantum IBE (PQIBE)

Here we follow [2] and use the method of Ducas et al [10]. We choose the parameter set recommended in [14] for a very conservative 192-bits of security. Using the ID-KEM-CCA given in [2] this results in a ciphertext of exactly 4000 bytes. Some savings could be achieved by using a 24-bit rather than a 28 bit prime modulus, as suggested in [21], [2].

9.3 Results

Again we find that the lattice based implementation requires much more memory/bandwidth. But by using a post-quantum IBE based TLS implementation we find that the overall memory/bandwidth requirement is significantly reduced compared with our original post-quantum PKI based TLS.

After a successful IBE TLS connection, standard ticket-based resumption can be used for subsequent connections to the same server. Also note that using a PSK allows the transmission of encrypted "early data" as discussed above, with the potential of improving overall client-server responsiveness.

And of course there is again the possibility of going with a hybrid solution – using a combination of BFIBE and PQIBE. This is simply achieved by concatenating both PSK encapsulations, then concatenating the extracted PSKs and processing the combination using the standard KDF (key derivation function).

On final observation. Although we find that in a post-quantum world using IBE results in a faster connection and the overall bandwidth requirement is more than halved compared to PKI, if using IBE the client is required to transmit more data. This may be an issue, as IoT nodes typically consume most battery power while transmitting.

	Bytes read	Bytes sent	Timing
BFIBE/X25519	264	580	354ms
PQIBE/KYBER	1308	5555	414ms
HYBRID	1349	5774	609ms

Table 5. Bytes sent/received by client using IBE based PSK, and timings on RP2040

10 Discussion

If Kyber and Dilithium maintain their claimed security levels into the future, it may be acceptable to lower their security from AES192 to AES128, with some savings. But this is far from being assured at the time of writing.

Just to extract one comparison from our timings, we find that for the Nano RP2040, without client authentication, an optimised ECC full TLS handshake takes 584 ms, and a resumption handshake takes 251 ms. A Post-Quantum full handshake takes 771ms, and a resumption takes 512ms.

One tentative conclusion might be that once hardware and assembly language support for lattice based crypto starts appearing there will not really be a significant difference in these timings. As the TLS1.3 resumption mechanism becomes more widely used this will also lead to significant improvements. Lattice based crypto typically spends a major proportion of its time hashing, and hence hardware support for the SHA3 hash algorithm would have a significant impact.

Another conclusion would be that implementation of post-quantum TLS1.3 on the majority of IoT nodes is likely to be quite feasible.

In this exercise we have used the same post-quantum signature algorithm at the same security level for all layers in the certificate chain. In fact alternative standard signature schemes with a different balance of signature/public key size, and at different security levels, may be a better fit. This will be the subject of further research. See also [16].

Of course its important to admit to what is not covered by this exercise, and that is the impact of the much larger bandwidth requirement on system delays and therefore on the overall user experience. More studies of the type reported by Westerban [28] are needed.

Going hybrid in a sense actually gives us the worst of both worlds – the larger keys and signature sizes associated with post-quantum methods are combined with the longer compute times associated with pre-quantum methods. However it appears to be the wisest way to progress in the immediate future.

Finally we recommend consideration of an IBE based TLS as an alternative to standard PKI based TLS for use in a closed-world IoT setting, as it significantly mitigates the cost of going post-quantum. As we have demonstrated this is quite easily integrated with the existing TLS protocol.

Acknowledgments

The author is grateful to Robert Merget for his help in setting up and using the remarkable TLS Anvil tool [20].

References

1. D. Atkins. Requirements for post-quantum cryptography on embedded devices in the IoT, 2021. `https://csrc.nist.gov/CSRC/media/Events/third-pqc-standardization-conference/documents/accepted-papers/atkins-requirements-pqc-iot-pqc2021.pdf`.
2. U. Bannerjee and A. Chandrakasan. Efficient post-quantum TLS handshakes using identity-based key exchange from lattices. In *ICC 2020 – 2020 IEEE International Conference on Communications*, pages 1–6, 2020.
3. D. Boneh and M. Franklin. Identity-based encryption from the Weil pairing. *SIAM Journal of Computing*, 32(3):586–615, 2003.
4. J. Bos, L. Ducas, E. Kiltz, T. Lepoint, V. Lyubashevsky, J. Schanck, P. Schwabe, G. Seiler, and D. Stehle. CRYSTALS – Kyber: a CCA-secure module-lattice-based KEM. Cryptology ePrint Archive, Paper 2017/634, 2017. `https://eprint.iacr.org/2017/634`.
5. J. Bos, J. Renes, and D. Sprenkels. Dilithium for memory constrained devices. Cryptology ePrint Archive, Paper 2022/323, 2022. `https://eprint.iacr.org/2022/323`.
6. L. Chen and Z. Cheng. Security proof of Sakai-Kasahara's identity-based encryption scheme. Cryptology ePrint Archive, Paper 2005/226, 2005. `https://eprint.iacr.org/2005/226`.
7. M. Mosca D. Stebila. Post-quantum key exchange for the internet and the open quantum safe project. In *Selected Areas in Cryptography*, pages 1–24, 2016. `https://openquantumsafe.org`.
8. Denis Diemert and Tibor Jager. On the tight security of TLS 1.3: Theoretically-sound cryptographic parameters for real-world deployments. Cryptology ePrint Archive, Paper 2020/726, 2020. `https://eprint.iacr.org/2020/726`.

9. L. Ducas, T. Lepoint, V. Lyubashevsky, P. Schwabe, G. Seiler, and D. Stehle. CRYSTALS – Dilithium: Digital signatures from module lattices. Cryptology ePrint Archive, Paper 2017/633, 2017. https://eprint.iacr.org/2017/633.
10. L. Ducas, V. Lyubashevsky, and T. Prest. Efficient identity-based encryption over NTRU lattices. In *Asiacrypt – 2014*, Lecture Notes in Computer Science, pages 22–41. Springer-Verlag, 2014.
11. A. Giron, J. Nascimento, R. Custodio, and L. Perin. Post-quantum hybrid KEMTLS performance in simulated and real network environments. Cryptology ePrint Archive, Paper 2022/1639, 2022. https://eprint.iacr.org/2022/1639.
12. R. Gonzalez and T. Wiggers. KEMTLS vs. post-quantum TLS: Performance on embedded systems. Cryptology ePrint Archive, Paper 2022/1712, 2022. https://eprint.iacr.org/2022/1712.
13. D. Grant. TLS 1.3 is going to save us all, and other reasons why IoT is still insecure, 2017. https://blog.cloudflare.com/why-iot-is-insecure/.
14. T. Guneysu and T. Oder. Towards lightweight identity-based encryption for the post-quantum-secure internet of things. In *18th International Symposium on Quality Electronic Design (ISQED)*, pages 319–324. IEEE, 2017.
15. Heartbleed. The heartbleed bug. https://heartbleed.com/.
16. P. Kampanakis, P. Panburana, E. Daw, and D. Van Geest. The viability of post-quantum X.509 certificates. Cryptology ePrint Archive, Paper 2018/063, 2018. https://eprint.iacr.org/2018/063.
17. P. Kietzmann, L. Boeckmann, L. Lanzieri, T. Schmidt, and M. Wahlisch. A performance study of crypto-hardware in the low-end IoT. Cryptology ePrint Archive, Paper 2021/058, 2021. https://eprint.iacr.org/2021/058.
18. H. Krawczyk. SIGMA: the SIGn-and-MAc approach to authenticated diffie-hellman and its use in the IKE protocols, 2003. https://webee.technion.ac.il/~hugo/sigma-pdf.pdf.
19. H. Krawczyk. Cryptographic extraction and key derivation: The HKDF scheme. In *Crypto 2010*, volume 6223 of *Lecture Notes in Computer Science*, pages 631–648. Springer-Verlag, 2010.
20. M. Maehren, P. Nieting, S. Hebrok, R. Merget, J. Somorovsky, and J. Schwenk. Tls-Anvil: Adapting combinatorial testing for TLS libraries. In *Proceedings of 31st Useneix Security Symposium*, pages 215–232, 2022.
21. S. McCarthy, N. Smyth, and E. O'Sullivan. A practical implementation of identity-based encryption over NTRU lattices. In *IMA Conference on Cryptography and Coding (IMACC)*, pages 227–246, 2017.
22. M. Paracha, D. Dubois, N. Vallina-Rodriguez, and D. Choffnes. IoTLS: Understanding TLS usage in consumer IoT devices. In *Proceedings of the 21st ACM Internet Measurement Conference*, pages 165–178, 2021.
23. T. Pornin. X25519 implementation for ARM cortex-M0/M0+, 2018. https://github.com/pornin/x25519-cm0.
24. Peter Schwabe, Douglas Stebila, and Thom Wiggers. Post-quantum TLS without handshake signatures. Cryptology ePrint Archive, Paper 2020/534, 2020. https://eprint.iacr.org/2020/534.
25. Peter Schwabe, Douglas Stebila, and Thom Wiggers. More efficient post-quantum KEMTLS with pre-distributed public keys. Cryptology ePrint Archive, Paper 2021/779, 2021. https://eprint.iacr.org/2021/779.
26. D. Stebila, S. Fluhrer, and S. Gueron. Hybrid key exchange in TLS 1.3, 2022. https://datatracker.ietf.org/doc/draft-ietf-tls-hybrid-design/.

27. Erik Sy, Christian Burkert, Hannes Federrath, and Mathias Fischer. Tracking users across the web via TLS session resumption, 2018. http://arxiv.org/abs/1810.07304.
28. Bas Westerban. Sizing up post-quantum signatures, 2021. https://blog.cloudflare.com/sizing-up-post-quantum-signatures/.
29. David Wong. A readable specification of TLS 1.3, 2018. https://www.davidwong.fr/tls13/.

Creating a post-quantum certificate chain

These OpenSSL instructions use the modified version of OpenSSL (available from the Open Quantum Safe project [7]) to create a post-quantum 3-link certificate chain based on the Dilithium signature scheme. It can easily be modified to create 3-link ECC-based and hybrid certificate chains.

```
#create Root CA
openssl req -new -x509 -days 365 -newkey dilithium3 -keyout dilithium3_CA.key -out dilithium3_CA.crt -nodes -subj "/CN=TiigerTLS root CA"
#create Intermediate CA
openssl req -new -newkey dilithium3 -keyout dilithium3_intCA.key -out dilithium3_intCA.csr -nodes -subj "/CN=TiigerTLS intermediate CA"
openssl x509 -req -CAcreateserial -days 365 -extfile myopenssl.cnf -extensions int_ca -in dilithium3_intCA.csr -CA dilithium3_CA.crt
-CAkey dilithium3_CA.key -out dilithium3_intCA.crt
#create Server certificate
openssl req -new -newkey dilithium3 -keyout dilithium3_server.key -out dilithium3_server.csr -nodes -subj "/CN=TiigerTLS server"
openssl x509 -req -in dilithium3_server.csr -CA dilithium3_intCA.crt -CAkey dilithium3_intCA.key -set_serial 01 -days 365
-out dilithium3_server.crt
#verify cert chain
openssl verify -CAfile dilithium3_CA.crt -untrusted dilithium3_intCA.crt dilithium3_server.crt
cat dilithium3_server.crt dilithium3_intCA.crt > dilithium3_certchain.pem
```

After running these instructions, provision the TLS server with files `dilithium3_server.key` (private key) and `dilithium3_certchain.pem`. Provision the TLS client with `dilithium3_CA.crt`, to be inserted into the root CA store.