Post Quantum TLS for the IoT

Now that is going to be challenging. In my last blog I pointed out that PQ certificate chains are going to get very big, which raises not only bandwidth issues, but also memory resource issues for a small IoT computing node.

So lets look at how to minimize the TLS1.3 memory requirement. Basically there will be a requirement to host an input buffer capable of storing a complete record, and also various arrays to store certificates, signatures and public keys.

One mitigation is to avoid making copies of these things without a good reason. For example certificates can be parsed in situ, in the input buffer. In theory signatures and public keys can also be accessed in a similar fashion. And the crypto itself should be keep as memory conserving as possible – for a good example of what can be achieved in the context of Dilithium see https://eprint.iacr.org/2022/323

However there are limits to this approach, as more extreme memory saving tricks start to impact on code readability, performance, and good software engineering practices.

Here we will concentrate on the TLS client, which is the party most likely to be implemented on a constrained device.

The TLS protocol

One good idea is to divide the main TLS protocol into relatively independent phases, so that memory can be released from an earlier phase, before more is required for the next. The main (non-resuming) TLS protocol in fact splits nicely into 4 phases.

The first is the exchange of Client and Server hellos. Typically this requires memory for the implementation of a cryptographic KEM, as ephemeral keys are generated, transmitted, and used to calculate a shared key. In our experimental Post Quantum scenario this requires an activation of the Kyber KEM algorithm. But once this is completed, most of the assigned memory can be released.

The second is the handling and parsing of the certificate chain received from the server. This will require access to large signatures and public keys, and an activation of the Dilithium signature verification code. But once all of that completes successfully, again the memory can be released.

Stage 3 is the optional preparation and transmission of the clients certificate chain, if the server should request client-side authentication. This requires a Dilithium signature.

Finally in stage 4 we wrap it all up. This phase has only minimal memory requirements.

Of course in practise many TLS client connections will be resumptions, which require much less memory, as only a KEM is needed, and no certificate processing is required.

The overall memory requirement depends on which of these phases requires most memory, that is which is on the *critical stack path*, and from our experiments this occurs in stage 2 of the main TLS protocol. Big certificates, large signatures and public keys, and the potentially memory-resource-intensive Dilithium code, all contribute.

Now the original Dilithium verification code required 56K of RAM. The clever authors of the above paper get this down to an impressive 2.7K. We decided to take a middle path and implement just the more obvious optimizations (with no performance impact) and got it down to around 14K.

So in the end we implemented our own C++ versions of Dilithium and Kyber rather than using the OQS versions, as these appear to have been written with little concern for memory consumption.

Stack or Heap?

Ideally all memory allocations should be from the stack, as it is well known that a stack-based architecture automatically takes care of the assignment, release and reuse of memory. However some IoT operating systems place a restriction on the maximum stack size, enforcing a rather shallow stack. Therefore we have made it an option for the TLS protocol to either allocate the larger arrays from the stack or the heap. However we would prefer if the low level crypto code still used only the stack.

One of the implications of this approach is that there is little point in going after savings that are not on the *stack critical path*. And as pointed out above the critical path arises in phase 2 of the full protocol.

Result

The IoT nodes we used in our experimentation were the TinyPICO, and the Arduino Nano RP2040 connect. The former uses the ESP32 processor clocked at 240MHz, and has the useful ability to report the stack high-water-mark. The latter uses an ARM M0+ processor clocked at 133MHz. Both support independent WiFi. Naturally enough neither at this stage has any hardware support for PQ primitives.

We have implemented this approach, and currently the maximum memory requirement comes in under 64K. By assigning protocol memory from the heap rather than the stack, the stack requirement can be reduced to ~24K.

And in case you were wondering, what about performance? Well no timings taken yet, but it certainly looks acceptably fast, especially on resumptions. The most telling comparison will be against a 256-bit elliptic curve based solution, for which the Arduino board does have hardware support.

As regards performance it wouldn’t hurt to have highly optimised assembly language versions of Dilithium and Kyber. And these do exist for the x86 and ARM M4 architectures, but not as far as we know for M0+ or the ESP32.