More on Dilithium

The basic structure of a program to do signature verification looks a little like this (and key generation and signature are broadly similar)

```
.. start off
for (i=0;i<6;i++)
     .. do stuff
     for (j=0;j<5;i++)
          .. do more stuff
… finish off
```

In the inner loop a matrix-vector multiplication is taking place, with a 6x5 matrix A of polynomials multiplying a vector of 5 polynomials. Each polynomial is itself represented by an array of 256 integer coefficients. The elements of the matrix A are calculated from a hash of an input of an array ρ of 32 pseudo-random bytes combined with the (i,j) coordinates of the element. Clearly each element of the array can be calculated independently as needed on-the-fly.

Now you don't have to a genius on asymptotics to appreciate that almost all of the execution time takes place inside the inner loop. And what is happening there is the construction of the array element $A(i,j)$, followed by some multiplication and addition of polynomials. But due to the magic of the Number Theoretic Transform (NTT) these multiplications and additions are incredibly fast. And everything inside that inner loop is already in NTT form. So that time critical inner loop is spending almost all of its time… hashing!

The reference implementation uses SHA-3 as its hashing algorithm – well known to be slow in software (but fast in hardware). Those 64-bit rotations it requires are not efficient on a 32-bit architecture.

So when timing Dilithium, you are actually timing SHA-3. Who knew...

Pre-computation?

In the reference implementation the matrix A is precomputed outside of the main loop. But this saves no time as ρ is part of the public key, and so the full matrix A needs to be calculated afresh for each signature that needs to be verified. However pre-computation of A would help with signature, as recall that each signature takes on average 5 attempts to succeed, and there is clearly no need to re-compute A each time. But for key generation and verification it is not necessary, and such a pre-computation consumes memory to no purpose. Which just goes to show that memory consumption is not a priority for many implementers. All that matters is execution time (or rather contrived "clock cycles" – see below). There are implementations which achieve faster execution time by bundling a precomputed A with the public key, but of course that blows up its size, and we want to keep it small.

Speed

Now lattice based crypto is known to be fast – faster indeed than most pre-quantum competitors. When looking for a post-quantum alternative we need one that ideally is not slower, and not bigger (in terms of public key and signature sizes), and also not bigger in terms of its dynamic memory requirement. With Dilithium we get the first but not the second or the third. We can probably live

with bulked up public keys and signatures, but if it requires more memory than our IoT node has – then its useless.

However most implementers still prioritise execution speed over everything else. Why? Its to do with the distortion caused by the requirement for academic publication – at least that's my theory. Write a paper describing an implementation which is 5% faster, and you are guaranteed a publication, irrespective of how useful it is in the real world. And since the point is to be faster than the competition there is a requirement to have some kind of level playing field. So its not actually execution time which is quoted (which is what the real world is interested in), instead performance is measured in clock cycles. But clearly that's not enough either, as other architectural factors can influence that (clock speed, cache memory etc). So the comparison requires a particularly contrived environment. So if you want to play this game you need the very specific ST32F407 discovery board which uses a ARM Cortex M4 chip. This naturally clocks at 168MHz, but since higher clock speeds require more wait states the board needs to be artificially slowed down to 24MHz so that the flash memory requires 0 wait states, which makes your cycle count look better.

So you end up with a number completely divorced from reality, and its not at all easy to work back from these cycle counts to real time. As it happens I have an ST32F407 board so I can play these games. I also have a couple of other boards based on the same ARM Cortex M4 chip, which give completely different cycle counts for the same implementation. And its all pretty pointless as Dilithium is clearly fast enough, and that is all that matters. So can we please get back to the other issues, signature/public key sizes and dynamic memory requirement?

Since I am operating in a space with no competitors I did easily get the memory requirement down to around 14K bytes for key pair generation, 24K for signature generation and 16K bytes for signature verification. Which is about 3 times smaller than the reference implementation (or any other implementation I have seen). This was at a small cost to performance, which since we already have a good margin to play with there, that really does not matter. The memory requirement is still a lot more than an elliptic curve implementation would need, as those polynomials are bulky, however I would regard it as acceptable.

Hashing

Some things really should be implemented in hardware, and hashing algorithms are one of them. These use no secrets, and hence have no requirements for constant time implementation or any of those other requirements that would arise if for example implementing a block cipher in hardware. The IoT node of the future will I am sure implement SHA-3 in hardware, which will completely blow away any lingering concerns over execution time issues.