Smaller, Faster

First thing is to choose a platform where being smaller and faster is really quite important, and will make a significant impact. So we choose the Arduino Nano 33 IoT node. 256K of ROM, 32K of RAM, an ECC608A crypto chip, and clocked at 48MHz. The processor is the lowest powered in the 32-bit ARM range, the M0, which doesn't even support a proper 32-bit multiply instruction. My Fitbit has a much more powerful architecture, based as it is on the superior M4 chip. So this really is the low end of the Internet of Things. A raspberry pi looks like a super-computer next to this thing.

Here the constraints really hurt, all of them. But we avoid (for now) the instinct to provide a severely cut-down version of the TLS1.3 protocol, and set ourselves the task of shoe-horning our implementation into the space provided, hoping at the same time to get decent performance.

Smaller

We are in the fortunate position of being in control of the entire code-base – there are no external libraries not under our control. So while we can't blame anyone else, we can control every aspect of the overall configuration.

The crypto SAL (Security Abstraction Layer) is provided by our miracl core library, with support from the ECC608a hardware crypto chip. Fortunately the miracl core library was designed not to be a memory hog. However as required in this application it does support multiple curves and RSA implementations simultaneously, and each supported primitive has a ROM/RAM requirement. For good TLS1.3 coverage we choose to support the secp256r1, secp384r1, x25519 elliptic curves, and RSA with key lengths of 2048 and 4096 bits. The larger RSA key sizes do consume a lot of memory, but paranoid people do like to use 4096-bit keys. Fortunately the ECC608a chip fully supports the secp256r1 curve as described in our last blog, so software support for this curve can be eliminated. We discovered that this saved us about 12K of ROM.

In our implementation of RSA we make use of the fast Karatsuba method for big number multiplication. This means that for example a 4096-bit multiplication can be calculated from 3 (not 4) 2048-bit multiplications. And this can be continued recursively, as a 2048-bit multiplication can be calculated from 3 1024-bit multiplications. How far down to go? Well we find that for maximum speed, recursing any lower than 1024-bits starts slowing things down. However since we are only ever doing RSA signature verification this doesn't really matter, as RSA verification is very fast. We found that by recursing down to 512-bits for both the RSA2048 and RSA4096 code, and thus sharing the 512-bit big number code, we saved a further 50K of ROM.

But we can go further. By recursing down to 256-bits, we can share some code with the 255-bit X25519 elliptic curve! This saves a further 18K of ROM.

Faster

Exploiting the ECC608a chip definitely speeds things up. We find that it takes 3.5 seconds to do a full handshake, and 1.1 seconds to do a resumption handshake. In fact we are not currently fully making full use of the ECC608a capabilities, as the library support for this chip has a few gaps in its coverage. So performance may improve further as we plug those gaps.

Sometimes its the little things that get you. Originally we were using the ECC608a random number generator to directly generate random bytes. However its actually very slow. Surprising, but I guess it takes its time to ensure that the provided bytes are truly random. By instead just using it to seed the miracl core PRNG, things speeded up considerably.

Next I looked again at the ECCX08 library code. It included built in delays for each operation. However these delays seemed to be based on an earlier version of chip. By decreasing them the full handshake came down to 2.7 seconds, and the resumption handshake down to 0.8 seconds.