Where to get the crypto

Recall that in TLS1.3 there are three phases where public key cryptography can potentially impose a large CPU penalty.

1. The ephemeral key exchange
2. The transcript signature verification
3. Certificate chain signature verifications

There are subtle differences between the requirements of each phase. For example certificate chain checking requires support for ECC and RSA, with RSA mostly using old-school padding methods like PKCS1.5. The ephemeral key exchange on the other hand can be, and often is, bang up to date using superior X25519 and X448 elliptic curves. The transcript signature verification may use ECC or RSA (the public key used for verification comes from the servers certificate), but can improve things by, for example, using RSA with modern padding methods like RSA-PSS. Note that RSA verification is still an order of magnitude faster than elliptic curve verification, and since trust in RSA is still critical for TLS (due to legacy certificate use), its quite valid to exploit this property.

When it comes to going post-quantum, the key exchange is easiest to update as it just requires a KEM, and post-quantum KEMs are quite efficient. One the other hand the transcript signature is more problematical as it requires a good PQ signature scheme (is there one?). Rolling out fully PQ certificate chains is certainly going to be a big complex task.

Given the range of these requirements it is unlikely that they can all be efficiently fulfilled from a single source. Our MIRACL library for example provides more coverage than most, but it is constrained performance-wise as it is written entirely in high level languages (C, C++ and Rust). It would be great if there were a single library that implemented all of our current and future TLS public key requirements in optimal assembly language for every processor out there. But that is not going to happen.

Therefore when using our Security Abstraction layer architecture we should leverage the fact that we can source our crypto implementations from across a multitude of available resources.

In the IoT

For example on the Arduino Nano rp2040 (based on the Raspberry Pi Pico) there is hardware support for the NIST secp256r1 elliptic curve, which is a curve that a TLS1.3 implementation MUST support. Using the ECC608A hardware we get the following timings

```
10:53:27.994 -> Cryptography by MIRACL Core + ECC608A hardware
10:53:33.423 -> SAL supported Key Exchange groups
10:53:33.423 ->     X25519
10:53:34.548 ->         Key Generation (ms)= 139
10:53:35.626 ->         Shared Secret (ms)= 139
10:53:35.626 ->     SECP256R1
10:53:36.704 ->         Key Generation (ms)= 81
10:53:37.735 ->         Shared Secret (ms)= 50
10:53:37.735 ->     SECP384R1
10:53:43.447 ->         Key Generation (ms)= 1427
10:53:49.124 ->         Shared Secret (ms)= 1427
```

Observe a few things. First this SAL supports the three most common TLS1.3 key exchange groups (there are a few others). The MIRACL software x25519 timings are pretty good! Note that the Pico uses the ARM M0+ processor and I am unaware of any optimized assembly language

implementation for this particular processor – implementers tend to concentrate on the more crypto-friendly M4 processor. Clearly the hardware supported secp256r1 timings are the best. We could try and exploit this by starting the key exchange in this group in our clientHello. However surprisingly many servers do not support it, and we end up having to respond to a Handshake Retry Request (HRR), forcing us back to x25519. So this particular hardware speed-up is hard to exploit in practice. Finally note that the software secp384r1 timings are very poor, but no hardware support is available.

Now lets try that again, this time not using the hardware, implementing support for secp256r1 using MIRACL software.

```
13:12:38.653 -> Cryptography by MIRACL Core
13:12:38.653 -> SAL supported Key Exchange groups
13:12:38.653 ->     X25519
13:12:39.746 ->         Key Generation (ms)= 139
13:12:40.860 ->         Shared Secret (ms)= 139
13:12:40.860 ->     SECP256R1
13:12:42.668 ->         Key Generation (ms)= 446
13:12:44.459 ->         Shared Secret (ms)= 445
13:12:44.459 ->     SECP384R1
13:12:50.298 ->         Key Generation (ms)= 1462
13:12:56.142 ->         Shared Secret (ms)= 1461
```

As can be seen the software secp256r1 implementation is now significantly slower. One can conclude that the designers of the ECC608A chip might have been better advised to support x25519 rather than secp256r1. But wait! We can also usefully use secp256r1 in phases 2 and 3 (see above), unlike x25519 which only applies to phase 1. (Therefore a bespoke TLS1.3 implementation that used secp256r1 for everything would be very fast on this device!)

On a PC

In a desktop environment crypto resources are plentiful. First consider an implementation that uses only MIRACL Core.

```
Cryptography by MIRACL Core
SAL supported Key Exchange groups
    X25519
        Key Generation     0.21 ms
        Shared Secret      0.23 ms
    SECP256R1
        Key Generation     0.42 ms
        Shared Secret      0.40 ms
    SECP384R1
        Key Generation     0.97 ms
        Shared Secret      0.96 ms
SAL supported Cipher suites
    TLS_AES_128_GCM_SHA256
    TLS_AES_256_GCM_SHA384
SAL supported TLS signatures
    ECDSA_SECP256R1_SHA256
    RSA_PSS_RSAE_SHA256
    ECDSA_SECP384R1_SHA384
SAL supported Certificate signatures
    ECDSA_SECP256R1_SHA256
    RSA_PKCS1_SHA256
    ECDSA_SECP384R1_SHA384
    RSA_PKCS1_SHA384
    RSA_PKCS1_SHA512
```

But we are aware of the Libsodium library which has a highly optimized x25519 implementation – the result of many man years of heroic optimization of this curve for the x86 architecture. Using a SAL based on a combination of MIRACL and Libsodium we get these timings

```
Cryptography by MIRACL Core + LibSodium
SAL supported Key Exchange groups
    X25519
        Key Generation     0.03 ms
        Shared Secret      0.05 ms
    SECP256R1
        Key Generation     0.40 ms
        Shared Secret      0.37 ms
    SECP384R1
        Key Generation     0.91 ms
        Shared Secret      0.96 ms
SAL supported Cipher suites
    TLS_CHACHA20_POLY1305_SHA256
    TLS_AES_128_GCM_SHA256
    TLS_AES_256_GCM_SHA384
SAL supported TLS signatures
    ECDSA_SECP256R1_SHA256
    RSA_PSS_RSAE_SHA256
    ECDSA_SECP384R1_SHA384
    ED25519
SAL supported Certificate signatures
    ECDSA_SECP256R1_SHA256
    RSA_PKCS1_SHA256
    ECDSA_SECP384R1_SHA384
    RSA_PKCS1_SHA384
    RSA_PKCS1_SHA512
    ED25519
```

Yes, your eyes do not deceive you, the x25519 timings are around 5 times faster. Indeed on certain processors like the x86 and ARM M4 assembly language implementation of these kind of crypto primitives can often be 3-5 times faster. We would be foolish not to use of it (licensing permitting). Also note that the Libsodium library while not supporting the other NIST elliptic curves, does provide support for CHACHA20 encryption and the Ed25519 elliptic curves (as usable in phases 2 and 3). These are also optional primitives for TLS1.3, which are not supported by MIRACL. So by using Libsodium we can grab them as well.

In a PQ world

Final observation: When it comes to testing PQ primitives in our TLS1.3 implementation, rather than implement these ourselves, we should simply use the existing optimized implementations that are already out there. Of course in practise it may not be quite as simple as that. For example in the constrained IoT context we may have to modify them to keep their memory requirements within reason.

Does any of this matter?

Now it could be that in the overall context of the TLS1.3 protocol maybe public key operations are not in fact that important. Maybe communication latencies contribute more to protocol delays. On the client side it is certainly less likely to matter, as a client may not be under either CPU or time pressure. However for a server that needs to maintain multiple client connections simultaneously then it would be important that such operations are executed as quickly as possible.

Bottom line. Its complicated!