Trust Abstraction? Not so simple

Trust is a big part of TLS. The Server is expected to authenticate itself to the client, so that the client can trust the server. And optionally the Client should also prove its trustworthiness to the server. Out of this trust-negotiation process both parties can obtain suitable session keys with which to encrypt their communications.

Ephemeral key exchange, sign the transcript

This is the current TLS1.3 way of doing it. It seems to put the cart-before-the-horse in that it calculates a session key first using ephemeral Diffie-Hellman, and only then establishes trust, the clever idea being that the sooner we have a shared session key, the sooner we can start encrypting things. A shared (unauthenticated) key is established, and then one or both parties digitally sign the transcript of the key exchange, which authenticates the original shared key after-the-fact.

In a post quantum world the simple solution would be a drop in replacement PQ key exchange followed by a replacement PQ digital signature. What's wrong with that? Well part of the pain of going PQ is that everything gets bigger. In hindsight elliptic curves had us spoiled as they allowed minimal exchanges of data and minimal signature sizes. Unfortunately PQ signatures tend to be massively bigger, which calls into question this whole approach.

KEMTLS – KEM for key exchange, KEM for signatures

Enter KEMTLS. But first what is a KEM – a Key Encapsulation Mechanism? It is a more generic way of doing what Diffie-Hellman does, but is a little more elaborate. It is also not symmetrical like classic DH, but that is not a problem, as DH is not used symmetrically in TLS1.3 anyway.

Alice generates a public/private key pair, PK and SK respectively. She sends PK to Bob who generates a random session key SS, which Bob encrypts (or *encapsulates*) with PK to create a ciphertext CT(PK) and sends this back to Alice who decrypts (*decapsulates*) it using SK to recover SS. Now both share SS. But there is also randomness involved. Not only are Alice's ephemeral key pair PK and SK generated randomly, the encryption by Bob also often involves randomness, so that the same SS does not encrypt the same way using the same PK.

Now a KEM can obviously be substituted for the first TLS1.3 key exchange. However it can also be used to implement a kind of implicit signature. Recall that a verified digital signature proves that the signer possesses the long term private key that matches the public key. Similarly a successful KEM based key exchange (this time using Alice's long term public/private key pair), where both parties end up with the same SS, also proves that Alice possesses the matching private key. She can prove this by encrypting something to Bob that decrypts correctly on his end. Unfortunately this "something" requires an extra flow of data from Alice to Bob. But it does implicitly authenticate Alice.

What's the point? As of now proposed PQ KEMs are a lot smaller than PQ signatures! However its not really TLS1.3 any more, it is at best a tweaked version of TLS1.3 as it involves slightly different data flows and a slightly modified state machine. But maybe TLS1.3 *needs* tweaking in a PQ world. To accommodate this, we need ours to be a tweakable TLS (TwiiTLS?)

TLS resumption?

It would appear that TLS resumption/preshared key is little impacted by this, as a digital signature is not required for resumption. So here only the DH key exchange needs to be substituted by a KEM in a straightforward way. But no extra flows, no changes to the state machine.

Over and backs – lets count them

In classic TLS, it goes like this (very simplifed). Here LTPK is the Server's Long Term Public Key. Note that a session key can be derived from a received key share or when a CT is created or decapsulated, and can be used to encrypt anything that follows.

The client is transmitting left-to-right, and the server right-to-left.

```
clientHello+key share ------>
<------serverHello+key share+{LTPK+certificate chain+Signature+Finished}+[application data]
Finished+application data ----->
```

Note that the server can start blasting away with encrypted application data almost immediately. So even as a client is finishing off the handshake its input buffer can be filling up with application data. The effect is to make TLS appear to be very responsive.

In KEMTLS, it goes like this (here STPK is a short term Public Key)

```
clientHello+STPK ------>
<------- serverHello+CT(STPK)+{LTPK+certificate chain}
{CT(LTPK)+Finished}+[application data] ------>
<------- {Finished}+[application data]
```

Note the extra communication, and that the server must delay sending application data. Those are the major downsides of KEMTLS.

Finally lets consider the IBE-based method discussed in an earlier blog. Here IDS is the server's URL. The client is assumed to hold a copy of the Trusted authority's public key.

```
CT(IDS)+{clientHello+STPK} ------>
<-------- {serverHello+CT(STPK)+Finished}+[application data]
{Finished}+[application data] ----->
```

Trust Abstraction

So it would appear that trust management can not be abstracted out from TLS in a simple way, as the way it is done may imply some structural changes to TLS.

Next moves

A good starting point would be to put up a public-facing KEMTLS test server. Such a thing does not exist AFAIK. We also need a tweakable state machine to support the various possibilities.