Introduction

This document aims to explain some details of the KEMTLS implementations available in the Rustls implementation. We hope that this document will clarify and help understand how the implementation is structured; and hopefully aid when considering making changes or when trying to implement KEMTLS variant protocols in other environments. It's intended that this document is read side-by-side with the code, as the level of detail required here would otherwise need to be overwhelming.

Protip for working with Rust: Although any editor is of course fine, we've got great experiences with Visual Studio Code with the rust-analyzer language server plugin (instead of the "Rust" plugin). It is able to provide lots of insight into Rust code. It should also be compatible with Vim or, god forbid, emacs, though we find that it's slightly easier to be navigate VSCode. The Rust compiler and linters do have considerable hunger for computer resources; if you're running into trouble with running everything on your ultralight laptop in reasonable time you could consider looking into VSCode Remote, which runs the program over SSH, which allows you to offload the work to any (powerful) machine. It's really super easy to set up as well.

Rustls

Rustls ("rustles") is a TLS library implemented in Rust. It's open source and available from https://github.com/ctz/rustls. Thom Wiggers implemented the original experimental versions of KEMTLS on top of this library [KEMTLS, KEMTLS-PDK]. The implementation of Patrick Towa's variant with semi-static epoch keys (which we'll call PDK-SS here, but it's also been referred to as 1rtt or ss1rtt) [PDK-SS] has been done on top of this work.

When looking at the code, it is important to realize that these variants of the protocol are, at various places, intersecting. Unfortunately this can make things a little bit messy at times. It's especially important to separate the KEMTLS-PDK code and the PDK-SS code, as although there are many similarities and in eded PDK-SS has been implemented as an extension of the PDK mechanism, due to the different key schedule and presence of update mechanisms for the semi-static keys, they in many places have different flows in the code.

Running the Rustls examples and the measurements

There are example programs in the `rustls-mio` folder that have been prepared for use with PDK-SS. The software can be compiled and run with `cargo run --example tlsserver -- --help`, similarly for the client. There is a sample PKI, using Kyber, in the `certificates` folder. See also `README-simon.txt`.

For setting up other combinations of algorithms, I recommend using the `measuring` folder, though that also needs Docker on your machine. The script `run_experiment.sh` can be used to run the experiments, for that you also need to have sudo/root on a Linux machine and to run the `setup_ns.sh` script first. It will only run a single experiment of every type unless you give it the `full` argument. See also its README.md. The experiments are run and managed by `experiment.py`, which also contains a lot of definitions for experiments. Make sure to adapt the level parallelism to the number of cores that your machine has.

If you just want to build a server, client with a certain combination of algorithms, run the `create-experimental-setup.sh` script with the arguments as defined in the start of that file. It will create a `bin/<your experiment>` folder which has the relevant algorithms.

Structure of the code

It's important to realise that the implementations build upon TLS 1.3 code. If you are trying to implement this on top of TLS 1.2, you're going to have a bad time: TLS 1.2 has significantly different behaviour, even if the

message flow is similar to that of KEMTLS. Notably, it doesn't encrypt any handshake messages and the key schedule is very different.

State machine

The Rustls TLS implementation has a fairly explicitly defined state machine. Each incoming message of the handshake is represented by a struct; the handshake in a certain state is represented by that struct. The State trait defines a `handle` method that is used for the processing of the incoming messages. Typically you will see that after a certain message is processed, say a `Certificate` message, it will call a method called `into_<next message>`, like `into_certificate_verify`, which prepares the state machine for this next method. The fields of the structs are used to carry forward state information.

New TLS Messages

There are a few new TLS handshake extensions and messages that are included as part of the implementation of PDK(-SS). Their encoding and parameters can be found through the structs that are used to represent these in the handshake protocol — the `src/msgs` folder of Rustls contains the logic used to process and generate these, but it's probably more helpful to just grep for individual names if ever necessary.

Resolvers for semistatic keys

It is somewhat common in TLS libraries that part of the configuration provided to the client and the server is the mechanisms through which the certificates are provided. Rustls uses such a "resolver" interface for its certificates as well and arbitrary functions can be provided through this mechanism.
We implemented this approach for the semistatic keys as well. For the client, this mechanism consists of two functions, one which fetches the currently active epoch key, and the other which handles an update message from the server. **The included default handler just stores this update message in memory.**
For the server, it has a function that provides the current epoch key, and a function that fetches the next key to be given to the client.
In the tlsclient.rs / tlsserver.rs programs these keys are passed via flags.

Client's handshake

ClientHello

The client starts the handshake with the client hello message. This message is constructed in the `rustls/src/client/hs.rs` file, in the function `emit_client_hello_for_retry` (the name reveals that the same function is used for HRR).

In the ClientHello message, the encapsulation to the server's long term (static) secret is included in the "ProactiveCiphertext" extension (originally from PDK). The additional semi-static encapsulation for PDK-SS is included in another client extension called "ProactiveCiphertextKEMTLS". Currently this is only done if client authentication is enabled.

The shared secrets generated in these operations are mixed in to the key schedule at the end of this function. The static shared secret is used to derive ES; the semi-static shared secret is used here to immediately derive HS. HS is also stored to use again later.
The client's certificate is encrypted under CHTS in PDK-SS. This key is set up for writing here. (This is one of the places where the difference with PDK is evident: PDK uses ETS to transmit the client certificate)

Submitting the certificate

When submitting the certificate, it's not included in the transcript for PDK-SS. This prevents a transcript synchronisation issue when the server can not decrypt the message. **This implementation detail requires some careful review with regards to the consequences for any proof of security.** It might be required that the client keep track of the handshake transcript with and without the certificate message, in case it gets rejected by the server.

Handling ServerHello

The client should carefully handle the ServerHello message, as in this message the server indicates if the PDK-SS mode was accepted. The initial part of the handling is done in `hs.rs`, until it is clear we are speaking TLS 1.3 or a KEMTLS-variant. The notable code in this file is in the `if sess.common.is_tls13()` block; the key schedule is started here and control is passed into the `src/client/tls13.rs` file.

The relevant handling of the ServerHello message is done in the `start_handshake_traffic` function in the `tls13.rs` file. Here, the ephemeral key exchange is completed by decapsulating the ct_e from the server. Next, the ProactiveCiphertext and ProactiveCiphertextKEMTLSAccepted extensions, which the server uses to indicate that the ciphertexts have been accepted, are detected. In PDK-SS the ciphertext encapsulated to the client's certificate is included in ProactiveCiphertextKEMTLSAccepted, so the client decapsulates it and computes the CAHTS and SAHTS keys The start_handshake_traffic function returns the updated key schedule, and briefly returns control to `hs.rs`.

In `hs.rs`, if PDK-SS is enabled, it will always try to handle a SPK message updating the epoch keys. Otherwise it will expect EncryptedExtensions: for PDK-SS these will be encrypted under SAHTS.

Handling ServerPublicKey

This message should be parsed and passed along the state machine until the SFIN is handled. If it's omitted by the server, the state machine will immediately continue to try to parse the incoming message as EncryptedExtensions.

Handling EncryptedExtensions

This message carries some TLS parameters. Here, for PDK-SS, if the client realises that the epochs did not match, it will transmit its certificate again and wait for the ciphertext. Otherwise, it will continue with `expect_finished_resume`.

Handling ServerFinished and optionally updating the semi-static key

The client makes sure to select the right Finished keys to verify the SFIN message. After verifying the hmac, any SPK message that's been received should be handled as it's now been authenticated. To do this, the update message is passed to the `update` method of the semi-static key resolver.

Server's handshake

Naturally, the server's behaviour is largely complementary to the messages sent and received by the client.

Receiving ClientHello

Again, this first message is handled in `src/server/hs.rs` while most of the logic is in `src/server/tls13.rs`. The part in `hs.rs` is not very interesting; control is quickly transferred to `tls13.rs`'s `handle_client_hello` method. This method is split in two parts, as before the server can send the SH reply, it must receive a certificate from the client in PDK and PDKSS modes.

Here, the client extensions for PDK and PDK-SS are detected. If the epochs match up, the server decapsulates the ciphertext from PDK-SS and derives the right keys. It then transitions to waiting for the client's certificate. After taking in the certificate, the server calls `emit_server_hello`, which will set up the key schedule. This function also does things in one of several ways depending on which type of handshake is done. If the PDK-SS ciphertext was accepted, the server encapsulates to the client's certificate and includes that ciphertext in the SH reply as an extension. Don't forget to verify the client's certificate here!
If the PDK-SS ciphertext was rejected, it leaves out this extension.
It will then send a ServerPublicKey message if necessary, ie. if the epochs did not match or if the server has a newer key available, followed by the EncryptedExtensions message which is processed as normal.

If the epochs did not match, the server again prepares to receive a certificate from the client. Otherwise, it sends SFIN and waits for the client's finished message.

Receiving the certificate again

This function receives the client's certificate and emits the ciphertext only if it's this second attempt. It now proceeds to send SFIN and waits for the client's finished message.

Key Schedule

The key schedule is defined in `key_schedule.rs`.
Unfortunately, it's one of the messier parts of the implementation due to the way it is set up as more of a state machine for the different keys in the TLS 1.3 handshake, rather than the "ratchet" structure e.g. used in the Go std/crypto/tls implementation of TLS 1.3. To make sense of what's going on, it's recommended to first take a good look at the key schedule in RFC 8446.

The key schedule of PDK-SS differs depending on if the semi-static key has been accepted. This makes it a bit messy at times to decide which key needs to be used, and you'll see in the code throughout the handshake that there are different states of the key schedule (optionally) carried forward in the handshake state structs.

Gotcha's

- The client needs to make an assumption about the server supporting a ciphersuite that's used to transmit the encrypted certificate. If the server doesn't understand this ciphersuite, the handshake will probably fail. Of course, when you can configure the public key on the client side, you may also set up this ciphersuite correctly. The Rustls implementation just hardcodes the ciphersuite to ChaCha20_Poly1305.

[KEMTLS]: https://thomwiggers.nl/publication/kemtls/
[KEMTLS-PDK]: https://thomwiggers.nl/publication/kemtls-pdk/
[PDK-SS]: https://eprint.iacr.org/2021/725
[RFC8446]: https://datatracker.ietf.org/doc/html/rfc8446