Doing it all over again in Rust

The original TIITLS client was written in C++ (actually very close to C). However it was always the plan that the Server would be written in Rust, and that there might be a Rust client as well. The reason being that Rust is seen as a better (and above all safer) overall language for this type of application.

The reason for the original C++ version was that Rust is still quite immature for use in the IoT space. It will get better over time, but for now C is still king when it comes to good performance and fitting into small spaces. Also there are a lot more open source crypto libraries written in C/Assembly than there are in Rust/Assembly.

So I translated the client from C++ into Rust. An interesting experience. And there were a few unexpected bonuses that arose from doing this exercise. First it made me look again and think again about the overall program structure. Second it caught quite a few bugs and other issues in the original C++ code. Finally since Rust does a lot of compile time checks for memory issues, I could be more confident that the C++ code was free of such problems.

One surprise was that the crypto code (none of which currently uses assembly language) was 10-20% faster in Rust.

The original C++ code was careful to make exclusive use of the stack, and only use the heap in extremis. Not all languages make it clear where memory comes from, and in Rust it took me a while to figure this out.

One very useful feature of Rust (and indeed many other modern languages) is the “slice”. Basically an object in stack memory can have its various parts “sliced” out. Take for example a certificate chain. Each certificate can be accessed as a slice. Each individual certificate can then have its signature and its embedded public key “sliced” out. So these components can be used in a read-only fashion without any further memory allocation.

A big part of our overall design is the Security Abstraction API Layer, which sits above all of the crypto. And this is common between client and server, as they both need to support the same primitives Also common is the certificate chain processing code. (Since clients can optionally be asked to authenticate by supplying their own certificate chain, both sides need this capability). So the rust code that supports these functionalities can be shared by both server and client.