Implementing TLS 1.2

Samuel Breese

For my part in the project, I implemented a host of symmetric and asymmetric cryptosystems. In this writeup, I'll detail each cryptosystem, security considerations, and any interesting issues that came up during implementation. This will proceed in roughly chronological order following the path of project development. The ultimate goal was to connect to Google using modern ciphers, the plan being to use the `TLS_ECDHE_RSA_CHACHA20_POLY1305_SHA256`. This naturally informed many choices in the development process (foremost among them being that we never implemented AES, or indeed any block cipher).

I began by implementing RSA. As RSA is very simple, this was not too difficult a task - most of the time was spent developing Miller-Rabin and various useful helper functions (modular exponentiation, for example). At first, I chose a very simple serialization/deserialization format for RSA public key. Later, I would have to return to the RSA implementation in order to

modify this format to match the RFC, but this was expected.

Next, I moved on to hash functions. I started with SHA-1 (since it is very well-documented and specified in the project requirements) and moved on to SHA256. There isn't very much to say about these - they're simple-to-implement and decent ciphers. I was able to reuse a lot of code between the two implementations; for example, the input processing for SHA-1 and SHA256 is identical. One notable aspect of the project's design choices is that since I implemented ChaCha20 later on, it would be easy to implement the BLAKE hash function (which was a contender for SHA-3) atop the current codebase. After implementing these hashes, the group presented our current work in class on 2018/11/26. Everything past this point is novel work conducted in the past week.

Since we were aware that we would eventually need a symmetric cipher, I decided to implement ChaCha20. ChaCha20 (alongside Poly1305 for message authentication) was recently (2014) adopted by Google. ChaCha20 is easy to implement, and is much faster than AES on platforms without AES operation in hardware. It is also immune to padding-oracle attacks of the kind used to attack AES in cipher block chaining mode. Since AES implementation is somewhat complex, and since I would need to introduce more complexity by implementing non-CBC modes, ChaCha20 was an easy choice.

Implementation for both ChaCha20 and Poly1305 proceeded smoothly by following RFC 7539, which specifies pseudocode and test vectors. Note that the implementation is likely extremely vulnerable to side-channel attacks.

After implementing ChaCha20-Poly1305, I realized that I had made an error assuming that the project could simply use naive RSA or Diffie-Hellman for key exchange. The cipher suites including ChaCha20-Poly1305 only use elliptic-curve Diffie-Hellman (ephemeral or not), authenticated with either RSA or elliptic-curve DSA. Therefore, the next task was to implement the primitives necessary for elliptic-curve cryptography. To ensure that we supported the right curve, I verified that Google's ECDHE was using Curve25519 (`https://en.wikipedia.org/wiki/Curve25519`). From here, I began implementation, following RFC 7748. This was somewhat more difficult than implementing ChaCha20, since the test inputs given in the RFC are considerably less granular, making it harder to isolate bugs. Nevertheless, I completed the X25519 implementation, giving the project the ability to perform ECDH and ECDH. It took some effort to determine the exact format for ECHDE in TLS; following The Illustrated TLS Connection (`https://tls.ulfheim.net/`) in addition to WireShark traces was very useful here. Once again, the implementation probably has many side-channel vulnerabilities: no consideration was given to this due to time constraints.

At this point, I had implemented all cryptographic primitives necessary for the chosen cipher suite: ECC via X25519 for ECDHE, RSA, the ChaCha20-Poly1305 AEAD, and SHA256. The next step was integrating these primitives with the TLS protocol code. This proved to be a difficult task. We proceeded by following RFC 5246 (which describes TLS), using traces from WireShark to guide us. We were able to simplify much of the protocol due to our simplified implementation - for example, my ECC implementation only supports one curve.

Ultimately, we faced a few lingering issues that we were not able to overcome by the project deadline (e.g. alert handling). The final client only connects to our own server, although we are sure that with a bit more poking and prodding it would all come together. The outgoing traffic at least looks similar enough to TLS that WireShark is able to recognize it as such. Our problems are mostly in properly handling unexpected records and properly serializing the ChaCha20-Poly1305 AEAD output, which I have not been able to find information about in any RFC or anywhere online. I could have resorted to reading source code for (e.g.) OpenSSL, but I felt that this would have made things less fun, so I refrained. It is also likely that there are flaws in the TLS extension parsing. On the security side, we completely ignore the server's certificate (since we are using ECDHE for key exchange, we don't

even need the public key sent in the certificate message).

For our application layer, we decided to develop a simple echo server, with a twist: the server would reverse the plaintext before encrypting and returning to the client. This was uninteresting and very simple to implement; as expected, most of our time was spent on the handshake.

On a personal note: this project was very enjoyable, and made me respect the state of modern cryptographic software. In many domains of contemporary software engineering, it would be astronomically difficult for a single author or small group to build a working prototype in this short of a timeframe. For example, to interface with the modern web, one would need to implement a rendering engine, a JavaScript interpreter, etc. Here, I was able to develop working (albeit insecure) implementations of an array of modern cryptographic primitives in around 40 man-hours. I don't think an experience like this would be possible in other domains, and it was very novel and enjoyable because of this.

To conclude and summarize: I implemented a (mostly functional) implementation of TLS 1.2 supporting the `TLS_ECDHE_RSA_CHACHA20_POLY1305_SHA256` ciphersuite. Most of the cryptosystems involved are relatively modern, making the task more interesting due to a lack of good references. Our final deliverables are a client and a server that reverses whatever it receives. We

are at present unable to connect to third-party TLS servers, but this is an issue of time, not lack of trying, and we may well reach this goal within the next week. Our implementation is almost certainly highly vulnerable to side-channel attacks. We are also vulnerable to man-in-the-middle attacks since we do not verify certificates.