University of Waterloo E-Thesis Template for LaTeX

by

Pat Neugraad

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Doctor of Philosophy
in
Philosophy of Zoology

Waterloo, Ontario, Canada, 2023

Examining Committee Membership

The following served on the Examining Committee for this thesis. The decision of the Examining Committee is by majority vote.

External Examiner:	Bruce Bruce Professor, Dept. of Philosophy of Zoology, University of Wallamaloo
Supervisor(s):	Ann Elk Professor, Dept. of Zoology, University of Waterloo Andrea Anaconda Professor Emeritus, Dept. of Zoology, University of Waterloo
Internal Member:	Pamela Python Professor, Dept. of Zoology, University of Waterloo
Internal-External Member:	Meta Meta Professor, Dept. of Philosophy, University of Waterloo
Other Member(s):	Leeping Fang Professor, Dept. of Fine Art, University of Waterloo

Author's Declaration

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

This is the abstract.

Acknowledgements

I would like to thank all the little people who made this thesis possible.

Dedication

This is dedicated to the one I love.

Table of Contents

Examining Committee ii

Author's Declaration iii

Abstract iv

Acknowledgements v

Dedication vi

List of Figures viii

List of Tables ix

1 Implementation and benchmarking 1

1.1 WolfSSL . . . 1

1.1.1 Integrating post-quantum algorithms . . . 1

1.1.2 Implementing KEMTLS . . . 2

1.1.3 Miscellaneous comments . . . 4

References 5

APPENDICES 6

List of Figures

List of Tables

Chapter 1

Implementation and benchmarking

To evaluate post-quantum TLS 1.3 and KEMTLS on constrained devices, we implemented post-quantum TLS 1.3 and KEMTLS on top of WolfSSL, a TLS library written in the C programming language. We then implemented a minimal TLS client on the Raspberry Pi Pico 2 W, a microcontroller with two ARM Cortex-M33 cores [1] and 512kB of SRAM, and measures the time it takes for the Pico client to complete a TLS 1.3 or KEMTLS handshake with a server. This chapter describes some of the implementation details, the benchmarking methodology, and the performance measurements.

1.1 WolfSSL

WolfSSL is a modern open-source TLS library written in C. In addition to a complete TLS stack, WolfSSL also includes its own cryptography library called WolfCrypt. Both WolfSSL and WolfCrypt are optimized for code size, speed, and memory footprint, and it portability and ease of configuration greatly simplifies managing multiple build targets using a single code base.

1.1.1 Integrating post-quantum algorithms

As of June 2025, WolfCrypt contains an in-house implementation of ML-KEM and ML-DSA. Both implementation are skillfully optimized, achieving at least 2x speedup on the

[1] The Pico 2 W also has 2 RISC-V cores, though we did not use them in this project.

Pico compared to the reference implementation [2]. Unfortunately, this makes the comparison across different schemes unfair. Instead, I chose to integrate with PQClean's clean implementations.

While PQClean is not specifically optimized for embedded system builds, all of its clean implementations are trivially portable to ARM. One non-trivial challenge is adapting the `randombytes` API in PQClean to am embedded system build with no operating system. Fortunately, WolfCrypt's `WC_RNG` API provides a common abstraction that works on both a desktop build (where random bits can be sourced from `/dev/urandom`) and Pico (where random bits are collected from various peripherals by the SDK). In the end, we expanded the `randombytes` API so it can be told to source random bits from user specified instance of `WC_RNG`.

Expanding the selection of KEMs for the initial key exchange (i.e. `ClientHello` and `ServerHello`) is straightforward, thanks to the fact that WolfSSL already supports ML-KEM for key exchange. The `NamedGroup` enum is trivially captured using a single 16-bit integer, and the logic for branching into the correct KEM allows for a simple `switch-case` block.

Expanding the selection of post-quantum signatures is trivial thanks to previous efforts to integrate `liboqs` into WolfSSL. We only need to replace all uses of `liboqs` with their equivalents in PQClean.

1.1.2 Implementing KEMTLS

Generating certificate chain and private keys

WolfCrypt's `asn.h` API provides a nearly complete collection of tools needed to generate certificate chains, encode certificates and private keys according to DER, then further encode them to PEM format. At the time of writing this thesis, WolfCrypt does not support signing a certificate signing request (CSR), but for benchmarking purposes I control the entire chain, and WolfCrypt does support directly signing the body of a certificate.

Modifying WolfCrypt's `asn.h` module to support KEM public key in a certificate is relatively straightforward. The only non-trivial obstacle comes from how WolfSSL handles OIDs. Object Identifier (OID) is a variable-length sequence of integers used to identify individual cryptographic primitives. For example, the OID for ML-KEM-512 is

[2] The Keccak implementation in WolfCrypt is only roughly 10% faster than PQClean's implementation, so the optimization must have come elsewhere

`2.16.840.1.101.3.4.4.1`. OID is included in an certificate to identify the public key and the signature; it is also included in the DER encoding of private keys. Having variable length makes OID tedious to work with when programming in C: unlike `NamedGroup`, which has fixed length that can be captured in a 16-bit integer, OID cannot be easily abstracted using a fixed-sized enum type. WolfSSL works around this limitation by using an "OID sum" algorithm, which computes an "hash" of an OID that fit into a 32-bit integer. Compressing variable-length integer sequence into a 32-bit integer carries with it the risk of collision, and in fact the first version of the OID summing algorithm indeed ran into a collision between SPHINCS-192-fast and SPHINCS-128-fast. A newer OID summing algortihm provided stronger collision resistance and resolved this issue. All OID sums are stored in a header file `oid_sum.h`, which is generated by a Perl script.

Implementing KEMTLS handshake

KEMTLS handshake workflow is identical to TLS 1.3's handshake flow from the beginning until client starts processing server's `Certificate` message. Even after KEMTLS and TLS 1.3's handshake flow diverges, they still share the format of the `Finished` message (which contains exactly one HMAC tag). Finally, once the handshake is complete, TLS 1.3 and KEMTLS exchange application data in identical fashion. The similarity between TLS 1.3 and KEMTLS handshake workflow allows us to reuse a significant part of WolfSSL's TLS 1.3 implementation, diverging at only a handful of places that are easy to reason about. While working with a TLS library written in C is intimidating at first, this implementation strategy proved successful, and I was able to finish implementing KEMTLS in less than a month using only around 4600 lines of code change.

The `WOLFSSL` struct is used on both client-side and server-side and encodes the pair of client and server state as a global TLS state. We begin modifying the TLS state machine by adding two flags `haveMlKemAuth` and `haveHqcAuth` to the main `WOLFSSL` struct. In a unilaterally authenticated KEMTLS handshake, the two flags are set on the server side when the server loads a KEM private key. Detecting a KEM private key is cleanly accomplished because at certificate generation, private keys are encoded according to DER, and the OID of the KEM scheme is included. If the OID belongs to one of ML-KEM's variants, then `haveMlKemAuth` is set, and if the OID belongs to one of HQC's variants, then `haveHqcAuth` is set. On the client side, these two flags are set when the client finds a ML-KEM or HQC public key in the certificate chain sent by the server. The combination of these two flags is sufficient for deciding when the two peers are performing a KEMTLS or TLS 1.3 handshake, and all divergence between KEMTLS and TLS 1.3 handshake flow will be controlled by these two flags.

On the client side, KEMTLS and TLS 1.3 handshake flow first diverges after the client finishes processing server's `Certificate` message. In signature-based TLS 1.3, client's immediate next step is to receive and process server's `CertificateVerify`, which contains a signature over the handshake transcript. In KEMTLS, client's immediate next step is to construct and send a `KemCiphertext` message. We followed the format used in the original KEMTLS implementation and let `KemCiphertext` contain only the ciphertext data with no metadata. This is acceptable within the context of this project since the peer loads at most one private key at a time, though in production use a server might load multiple private key and may require `KemCiphertext` to contain metadata indicating which keypair should be used.

1.1.3 Miscellaneous comments

References

[1] Michel Goossens, Frank Mittelbach, and Alexander Samarin. *The LaTeX Companion.* Addison-Wesley, Reading, Massachusetts, 1994.

[2] Donald Knuth. *The TeXbook.* Addison-Wesley, Reading, Massachusetts, 1986.

[3] Leslie Lamport. *LaTeX — A Document Preparation System.* Addison-Wesley, Reading, Massachusetts, second edition, 1994.

APPENDICES