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

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

List of Figures

List of Tables

Chapter 1

Post-quantum TLS on embedded systems

1.1 An overview of TLS 1.3

Transport Layer Security (TLS) is a communication protocol with which two parties can securely exchange data over an insecure transport layer. It was first developed at Netscape in 1994 and went through many iterations. The latest revision is TLS 1.3, formally standardized by IETF in RFC8446 [?] in 2018. TLS 1.3 made significant improvements over TLS 1.2 [?] and prior versions, including deprecating weak symmetric cipher suites, removing static key exchange schemes that do not provide forward secrecy, and restructuring the handshake state machine in order to simplify the handshake workflow. According to Cloudflare [?], TLS 1.3 reached more than 93% adoption on the Internet. Given TLS 1.3's superior design and nearly universal adoption over other versions of TLS, this work will focus on this version unless otherwise specified.

A new TLS 1.3 connection begins with a handshake in which the two parties exchange a specific sequence of messages to negotiate cryptographic parameters, establish shared secrets, and authenticate each other. If the handshake is successful, then the connection transitions to exchanging application data using the encryption scheme and secret keys obtained from the handshake phase. In TLS 1.3, the handshake can be further divided into two phases [1]:

[1]We omitted parameter negotiation messages like `HelloRetryRequest` and `EncryptedExtensions`. They are either optional or inconsequential to our argument

1. **Key exchange**
    (a) `ClientHello`: client publishes its supported symmetric cipher suites (AEAD, hash functions), signature algorithms, a random nonce, and key exchange public keys.
    (b) `ServerHello`: server chooses its preferred symmetric cipher suite, as well as publishes its own random nonce and key exchange public keys.
2. **Authentication**
    (a) `Certificates`: server sends a chain of X.509 certificates that bind its identity to a digital signature public key; optionally, server can request client authentication (`CertificateRequest`), in which case the client must also send a chain of certificates.
    (b) `CertificatesVerify`: server proves ownership fo the corresponding digital signature secret key by producing a signature over the handshake transcript; client will do the same upon server's request
    (c) `Finished`: both parties prove to each other the integrity of the handshake by producing an HMAC over the handshake transcript

Figure 1.1 illustrates a simple TLS 1.3 handshake:

1.1.1 Certificate-based authentication

1.1.2 Key schedule and HKDF

1.1.3 Transition to post-quantum

1.2 Embedded device security

1.2.1 Implementing (post-quantum) TLS on embedded device

1.2.2 Authenticating the embedded device

Prior works [?, ?, ?] evaluating post-quantum TLS on embedded systems primarily focused on configurations in which the constrained device is an anonymous TLS client who does

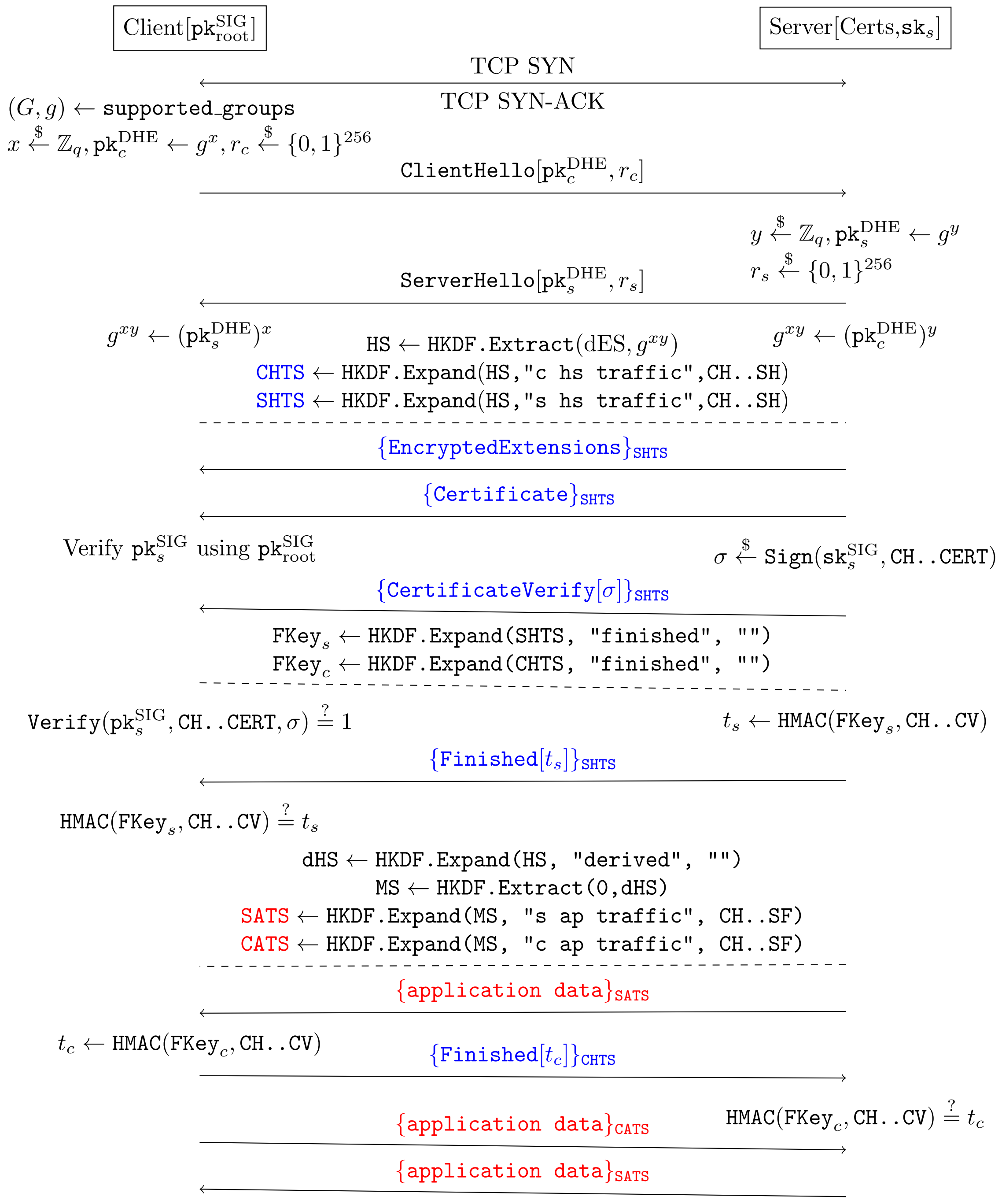
Client[$\mathsf{pk}_{\mathrm{root}}^{\mathrm{SIG}}$]
Server[Certs,sk_s]
TCP SYN
TCP SYN-ACK
$(G, g) \leftarrow$ supported_groups
$x \xleftarrow{\$} \mathbb{Z}_q, \mathsf{pk}_c^{\mathrm{DHE}} \leftarrow g^x, r_c \xleftarrow{\$} \{0,1\}^{256}$
ClientHello[$\mathsf{pk}_c^{\mathrm{DHE}}, r_c$]
$y \xleftarrow{\$} \mathbb{Z}_q, \mathsf{pk}_s^{\mathrm{DHE}} \leftarrow g^y$
$r_s \xleftarrow{\$} \{0,1\}^{256}$
ServerHello[$\mathsf{pk}_s^{\mathrm{DHE}}, r_s$]
$g^{xy} \leftarrow (\mathsf{pk}_s^{\mathrm{DHE}})^x$
HS $\leftarrow$ HKDF.Extract(dES, g^{xy})
$g^{xy} \leftarrow (\mathsf{pk}_c^{\mathrm{DHE}})^y$
CHTS $\leftarrow$ HKDF.Expand(HS,"c hs traffic",CH..SH)
SHTS $\leftarrow$ HKDF.Expand(HS,"s hs traffic",CH..SH)
{EncryptedExtensions}$_{\mathsf{SHTS}}$
{Certificate}$_{\mathsf{SHTS}}$
Verify $\mathsf{pk}_s^{\mathrm{SIG}}$ using $\mathsf{pk}_{\mathrm{root}}^{\mathrm{SIG}}$
$\sigma \xleftarrow{\$}$ Sign($\mathsf{sk}_s^{\mathrm{SIG}}$, CH..CERT)
{CertificateVerify[σ]}$_{\mathsf{SHTS}}$
$\mathsf{FKey}_s \leftarrow$ HKDF.Expand(SHTS, "finished", "")
$\mathsf{FKey}_c \leftarrow$ HKDF.Expand(CHTS, "finished", "")
Verify($\mathsf{pk}_s^{\mathrm{SIG}}$, CH..CERT, σ) $\overset{?}{=} 1$
$t_s \leftarrow$ HMAC(FKey_s, CH..CV)
{Finished[t_s]}$_{\mathsf{SHTS}}$
HMAC(FKey_s, CH..CV) $\overset{?}{=} t_s$
dHS $\leftarrow$ HKDF.Expand(HS, "derived", "")
MS $\leftarrow$ HKDF.Extract(0,dHS)
SATS $\leftarrow$ HKDF.Expand(MS, "s ap traffic", CH..SF)
CATS $\leftarrow$ HKDF.Expand(MS, "c ap traffic", CH..SF)
{application data}$_{\mathsf{SATS}}$
$t_c \leftarrow$ HMAC(FKey_c, CH..CV)
{Finished[t_c]}$_{\mathsf{CHTS}}$
{application data}$_{\mathsf{CATS}}$
HMAC(FKey_c, CH..CV) $\overset{?}{=} t_c$
{application data}$_{\mathsf{SATS}}$

Figure 1.1: TLS 1.3 handshake with (EC)DHE and signature-based authentication

not need to authenticate itself to the server. While the TLS 1.3 (and KEMTLS) supports mutual authentication, implementing it on an embedded device posts unique challenges. The same set of challenges also apply to using embedded device as TLS server.

The primary obstacle to implementing authentication on embedded device is the threat of physical intrusion. For example, servers in a data center are physically secured within a building guarded by human with guns, so they often do not require extra hardware security measures. In some critical use cases where extra security is required (e.g. banking), a hardware security module (HSM) can be deployed on premise. By comparison, embedded devices are commonly deployed in unprotected if not hostile environments, where adversaries can easily capture the device and execute sophisticated physical attacks.

There are numerous hardware and software measures that can be deployed to protect secrets on an embedded devices, including but not limited to physical unclonable functions (PUF), trusted platform modules (TPM), cryptocoprocessors, and secure boot. In addition to cost and power consumption concerns, however, the existence of a “secure storage” means one can simply deploy a 256-bit symmetric key and use symmetric primitives (i.e. HMAC) to authenticate the device. In fact, TLS 1.3 supports handshake using pre-shared key (PSK)[**?**] in environments with no public key infrastructure (PKI). PSK automatically provides mutual authentication and also offers vastly superior performance by eliminating all asymmetric cryptographic operations. In other words, regardless of whether private keys can be securely stored or not, there is no scenario in which authenticating an embedded device using public-key cryptography makes sense.

Chapter 2

Implementation and performance measurements

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.

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

2.1.1 Integrating additional 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 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.

2.1.2 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.

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

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 `2.16.840.1.101.3.4.4.1`. 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 unilaterally authenticated 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 flows first diverge 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` containing a signature over the handshake transcript. In KEMTLS, client will not receive additional message. Instead, it uses the KEM public key to encapsulate the **authentication secret**, then sends the ciphertext to the server in a `KemCiphertext` message. We followed the original KEMTLS implementation's `KemCiphertext` format as a handshake message whose payload contains the raw ciphertext and no additional metadata. Within the context of this project, each server instance will only load one private key, so `KemCiphertext` not carrying metadata on the ciphertext will not cause confusion. However, WolfSSL supports loading multiple private keys for authentication, in which case it might be necessary for `KemCiphertext` to carry metadata such as an OID.

Client's `KemCiphertext` is encrypted under handshake traffic keys derived from the unauthenticated handshake secret (HS). After sending `KemCiphertext`, client must update the key schedule by mixing in the authentication secret and deriving the authenticated handshake secret (AHS). From AHS, the client will derive new handshake traffic key for encrypting additional handshake messages, as well as application traffic key. This key schedule update is necessary for subsequent messages to provide implicit authentication: no adversary, even if it compromises the handshake secret, can decrypt subsequent handshake message or application data without the long-term secret key.

Client's `Finished` follows the same format as TLS 1.3's `Finished`: a handshake message whose payload contains a raw HMAC tag computed under the `finished_key` against the handshake transcript. The MAC key is derived from the `MasterSecret`, which is itself derived from AHS. After sending `Finished`, the client can start sending application data without receiving server's `Finished` first. Because application data will be encrypted under application traffic key derived from authenticated handshake secret, any server successfully decrypting them is implicitly authenticated. Finally, client explicitly authenticates the server after receiving server's `Finished`.

On the server side, KEMTLS and TLS 1.3 handshake flow first diverges after server finishes sending its `Certificate`. If either of `haveMlKemAuth` and `haveHqcAuth` flag is set, then after exiting `SendTls13Certificate`, instead of constructing and sending `CertificateVerify`, server will receive and process `KemCiphertext`. After decapsulating client's `KemCiphertext`, server similarly needs to update the key schedule: first derive the authenticated handshake secret using the authentication secret, then derive new handshake traffic keys, HMAC keys for processing client's `Finished` and constructing server's `Finished`, and finally the application traffic keys. Last but not least, server needs to construct and send its `Finished` before it can start sending application data.

In our implementation, we used the handshake message type for `client_key_exchange` when constructing `KemCiphertext`. `client_key_exchange` is a handshake message type that exists only in TLS 1.2 or prior but not in TLS 1.3. This upsets some sanity checks in WolfSSL's TLS 1.3 state machine. Similarly, the different order of messages, particularly the order of `Finished`, can also cause the same set of sanity checks to fail. For the purpose of benchmarking performance only, we modified the these sanity checks to ignore message orders when `haveMlKemAuth` or `haveHqcAuth` is set. However, in production use, KEMTLS will definitely require a distinct set of checks to ensure the integrity of the handshake state machine.

2.1.3 Miscellaneous comments

We appreciate the many thoughtful design choices made in both WolfSSL and WolfCrypt. Using C preprocessing macros defined in a easily manageable `user_settings.h` file allowed us to build for three different platforms (Apple Silicon on the author's laptop running MacOS, `x86_64` on the test server running Linux, and 32-bit ARM on baremetal) using a single codebase. The modular I/O callback API made it possible to run WolfSSL on the Pi Pico without any RTOS. WolfSSL's repository even provided ready-made integration with the `Pico-SDK` so that using Pico's hardware RNG with WolfCrypt's `WC_RNG` struct requires no additional effort from the authors.

Another difficulty with WolfSSL comes from the need for manual memory management. This is especially the case when building for the Pico, which has only 512kB SRAM and can be easily overwhelmed by memory leaks since post-quantum keys, ciphertexts, and signatures all consume 1-10 kilobytes of memories each. Fortunately, there are only a handful of places where WolfSSL requires dynamic memory allocations, and they are either freed within the function scope or freed at the end of the handshake as part of `wolfSSL_shutdown` or `wolfSSL_free`. Other instances of dynamic memory allocation were flawlessly managed in WolfSSL's existing source code, allowing the Pico to continuously perform thousands of handshakes without exhausting its SRAM or needing assistance from an RTOS.

2.2 Hardware setup

We chose to test TLS 1.3 and KEMTLS performance on a Raspberry Pi Pico 2 W and a Linux desktop. The Raspberry Pi Pico 2 W is a microcontroller developed by the Raspberry Pi Foundation and released in late 2024. At the heart of the Pico 2 W is the RP2350 chip, which contains two ARM Cortex-M33 cores and two Hazard3 RISC-V cores, though only one architecture can be enabled at boot time. The chip is clocked at 150MHz. The Pico 2 W also has 520KB of SRAM and 4MB of flash storage. For connectivity, the Pico 2 W has a CYW43439 module that supports 2.4 GHz Wifi and Bluetooth 5.4.

For developing on the Pico 2 W, we used the GNU ARM toolchain (`arm-none-eabi-gcc` version 8.5.0) and the Pico-SDK, which comes with a standard library as well as integration with third-party open source projects, most notably `lwip` (lightweight IP). `lwip` is a TCP/IP library designed for embedded system, and the Pico-SDK provided tight integration between `lwip` and the `cyw43` driver, which allowed us to implement a simple TCP stream on top of the raw TCP API. We also used `lwip` to implement DNS and time synchronization over NTP, the latter being necessary for WolfSSL to validate the expiration dates of peer's certificates.

Because of the lack of a file system, certificates must be baked into the firmware at compilation. While it is possible to define certificates as raw bytes in C preprocessing macro, we found it much easier to encode the certificates in PEM format, which consists entirely of ASCII characters. On the other hand, while private keys can also be encoded in DER or PEM format and included into the firmware at compilation, without further protection they can be easily extracted from the devices via a firmware dump. The Pico 2 W features a one-time programmable (OTP) ROM that, once enabled and written to, will permanently be protected by secure boot. In December 2024, Raspberry Pi Ltd. launched a hacking challenge[3] in which the participants try to extract a secret from a secure-boot-enaled Pico 2 W. While some participants have successfully extracted the secret, all publicly known winners used intrusive methods that required extensive physical access and non-trivial external hardware. Another issue with using the OTP for private key is size, since on RP2350 the OTP has a limited capacity of 8KB, although it is possible to store a seed that can then be expanded into the private key at bootup. Embedded devices that need to store private keys should consider using a peripheral device, such as a trusted platform module (TPM) or secure hardware module (HSM).

The Linux desktop has an 8-core AMD Ryzen 7 1700X with 12GB of RAM, running Ubuntu 24.04 LTS, and compiling with `GCC 13.3.0`.

[3]https://github.com/raspberrypi/rp2350_hacking_challenge

Our network setup takes advantage of Pico 2 W's wireless capabilities. We used a netowrk router that is connected to University of Waterloo's internal network via Ethernet, then connect the Pico 2 W to the router via Wifi. The Linux desktop was connected to the internal network via Ethernet as well. **I need more metrics on the network :(**

2.3 Performance of TLS 1.3 and KEMTLS

In this section we will compare the performance of post-quantum TLS 1.3, KEMTLS, and classical TLS 1.3.

2.3.1 Communication cost

TLS 1.3 recommended ephemeral elliptic curve Diffie-Hellman key agreement (ECDHE) to be the default scheme for key exchange in `ClientHello` and `ServerHello`. Using ECDHE, the client and the server each sends the other an element from the agreed group of points on an elliptic curve, so round-trip communication cost is the sum of two group points. When using a post-quantum KEM, the client sends to the server an encapsulation key, and the server sends back a ciphertext obtained under said encapsulation key. The round-trip communication cost of a KEM key agreemnt is thus the sum of a public key and a ciphertext. Table 2.1 lists the costs of each key agreement scheme.

Table 2.1: Key agreement communication costs (bytes). For NIST curves we include the costs for compressed and uncompressed forms.

Category	Scheme	Public key	Ciphertext	Total
ECDHE	SECP256R1	33/65		66/130
	SECP384R1	49/97		98/194
	SECP521R1	67/133		134/266
	X25519	32		64
	X448	56		112
PQC	ML-KEM-512	800	768	1568
	ML-KEM-768	1184	1088	2272
	ML-KEM-1024	1568	1568	3136
	HQC-128	2249	4433	6682
	HQC-192	4522	9029	13551
	HQC-256	7245	14469	21714

Post-quantum digital signature schemes also have larger sizes than elliptic-curve signatures and RSA-signatures. Table 2.2 compares the sizes of public key and signature across the chosen set of signature schemes.

Table 2.2: Digital signatures communication costs (bytes)

Category	**Scheme**	**Public key**	**Signature**	**Total**
Classical	RSA-2048 (PKCS#1 v1.5)	256	256	512
	ECDSA SECP256R1	64	64	128
	ECDSA SECP384R1	96	96	192
	ECDSA SECP521R1	132	132	264
	Ed25519	32	64	96
	Ed448	57	114	171
PQC	ML-DSA-44	1312	2420	3732
	ML-DSA-65	1952	3309	5261
	ML-DSA-87	2592	4627	7219
	Falcon-512	897	666	1563
	Falcon-1024	1793	1280	3073
	SPHINCS+-128f	32	17088	17120
	SPHINCS+-192f	48	35664	35712
	SPHINCS+-256f	64	49856	49920
	SPHINCS+-128s	32	7856	7888
	SPHINCS+-192s	48	16224	16272
	SPHINCS+-256s	64	29792	29856

In our handshake setup, server's public key sits at the top of a chain of three certificates (root, intermediate, and leaf). However, the root certificate is compiled into the client's firmware, so it is not transmitted in the handshake. Table 2.3 compares the sizes of certificate chain.

2.3.2 Cryptographic operations performance

We benchmarked the speed of relevant cryptographic operations on the Pico 2 W. Table 2.4 summarizes the results.

2.3.3 Handshake performance

TLS 1.3 and KEMLS handshake can be divided into two distinct phases. The first phase is key agreement, which starts at the beginning at the handshake and ends when client

Table 2.3: Server certificate chain size (bytes). Each certificate is encoded in PEM format. For rows with one name, there are still three certificates, but they use the same scheme for all three keys.

Category	**Root**	**Intermediate**	**Leaf**	**Size**
Classical	RSA2048			2506
	ECDSA SECP256R1			1429
	ECDSA SECP384R1			1429
	ECDSA SECP521R1			1429
	Ed25519			1893
	Ed448			1893
PQ-TLS	ML-DSA-44			16763
	ML-DSA-65			???
	ML-DSA-87			???
	SPHINCS-128f	ML-DSA-44	ML-DSA-44	54734
	SPHINCS-128s	ML-DSA-44	ML-DSA-44	??
	Falcon-512	ML-DSA-44	ML-DSA-44	??
	SPHINCS-192f	ML-DSA-65	ML-DSA-65	??
	SPHINCS-192s	ML-DSA-65	ML-DSA-65	??
	SPHINCS-256f	ML-DSA-87	ML-DSA-87	??
	SPHINCS-256s	ML-DSA-87	ML-DSA-87	??
	Falcon-1024	ML-DSA-87	ML-DSA-87	??
PQ-KEMTLS	ML-DSA-44	ML-DSA-44	ML-KEM-512	???
	ML-DSA-65	ML-DSA-65	ML-KEM-768	???
	ML-DSA-87	ML-DSA-87	ML-KEM-1024	???
	ML-DSA-44	ML-DSA-44	HQC-128	???
	ML-DSA-65	ML-DSA-65	HQC-192	???
	ML-DSA-87	ML-DSA-87	HQC-256	???

Table 2.4: Cryptographic operations on RP2350 (ARM Cortex-M33 at 150MHz).

ECDHE

Name	`KeyGen`			`Agree`		
	Medium	P90	P99	Medium	P90	P99
X25519						
X448						
SECP256R1						
SECP384R1						
SECP521R1						

Post-quantum KEM

	`KeyGen`			`Encap`			`Decap`		
	Medium	P90	P99	Medium	P90	P99	Medium	P90	P99
ML-KEM-512									
ML-KEM-768									
ML-KEM-1024									
OT-ML-KEM-512									
OT-ML-KEM-768									
OT-ML-KEM-1024									
HQC-128									
HQC-192									
HQC-256									

Digital signature

	`Sign`			`Verify`		
	Medium	P90	P99	Medium	P90	P99
RSA-2048						
Ed25519						
ECDSA (P-256)						
ECDSA (P-384)						
ECDSA (P-521)						
ML-DSA-44						
ML-DSA-65						
ML-DSA-87						
SPHINCS+-128f						
SPHINCS+-128s						
SPHINCS+-192f						
SPHINCS+-192s						
SPHINCS+-256f						
SPHINCS+-256s						
Falcon-512						
Falcon-1024						

finishes processing `ServerHello`. The second phase is authentication, which starts at the end of the key agreement phase and ends when the handshake is completed. We collected the timestamps at key moments from the client side:

1. `ch_start`: when the handshake starts
2. `ch_sent`: when `ClientHello` is sent
3. `sh_start`: when `ServerHello` is received
4. `sh_done`: when `ServerHello` is processed
5. `auth_start`: when `Certificate` is received
6. `auth_done`: when the handshake finishes

Using these timestamps we can clearly define the boundaries of the two phases from the client's perspective: key agreement duration is time delta between `sh_done` and `ch_start`, while authentication duration is the interval from `auth_start` to `auth_done`. These detailed timestamps also allow us to observe the impact of communication cost on handshake. For example, the interval from `sh_done` to `auth_start` reflects the time it takes for server to transmit the certificate chain.

Key agreement. We define the total key agreement time to be the length of interval from the start of the handshake to the end of processing `ServerHello`. In addition, we also track the length of time for which the Pico is doing active work: the time spent constructing `ClientHello` plus the time spent processing `ServerHello`. Table 2.5 compares the performance of various key exchange scheme using these two metrics. Figure 2.1 illustrates the distribution of data points.

Authentication. We define the *certificate transmission time* to be the length of interval from the end of key agreement to the start of processing server's certificates, *authentication time* to be from start of processing certificates to the end of the handshake. Table 2.6 summarizes these metrics. Figure 2.2 illustrates the distribution of data points.

Finally, Table 2.7 and Figure 2.3 summarize the performance of the handshake as a whole.

2.3.4 Discussion

Category	Key agreement	Total time		CPU time	
		Median	Std	Median	Std
Classical	ECDHE (P-256)	445096.0	195745.12	438639.0	182.35
	X25519	36322.5	713456.00	28813.0	31.68
	ECDHE (P-384)				
	ECDHE (P-521)				
	X448				
PQC	ML-KEM-512	27254.0	284903.41	19443.0	85.32
	ML-KEM-768				
	ML-KEM-1024				
	OT-ML-KEM-512	16938.0	5573.21	14025.0	41.26
	OT-ML-KEM-768				
	OT-ML-KEM-1024				
	HQC-128	1341054.5	303532.68	1321492.0	80.01
	HQC-192				
	HQC-256				

Table 2.5: Comparing key agreement performance (microseconds).

Category	Certificate chain			Cert TX time	Auth time
	Root	Int	Leaf		
Classical	RSA-2048				
	ECDSA (P-256)				
	Ed25519				
PQC	ML-DSA-44				
	SPHINCS+ 128f	ML-DSA-44	ML-DSA-44		
	ML-DSA-44	ML-DSA-44	ML-KEM-512		

Table 2.6: Medium authentication durations (microseconds).

Category	KA	Auth			Handshake time
		Root	Int	Leaf	
Classical	ECDHE (P-256)	RSA-2048			
	ECDHE (P-256)	ECDSA (P-256)			
	X25519	Ed25519			
PQC	ML-KEM-512	ML-DSA-44			
	ML-KEM-512	SPHINCS+ 128f	ML-DSA-44	ML-DSA-44	
	ML-KEM-512	ML-DSA-44	ML-DSA-44	ML-KEM-512	

Table 2.7: Medium handshake durations (microseconds).

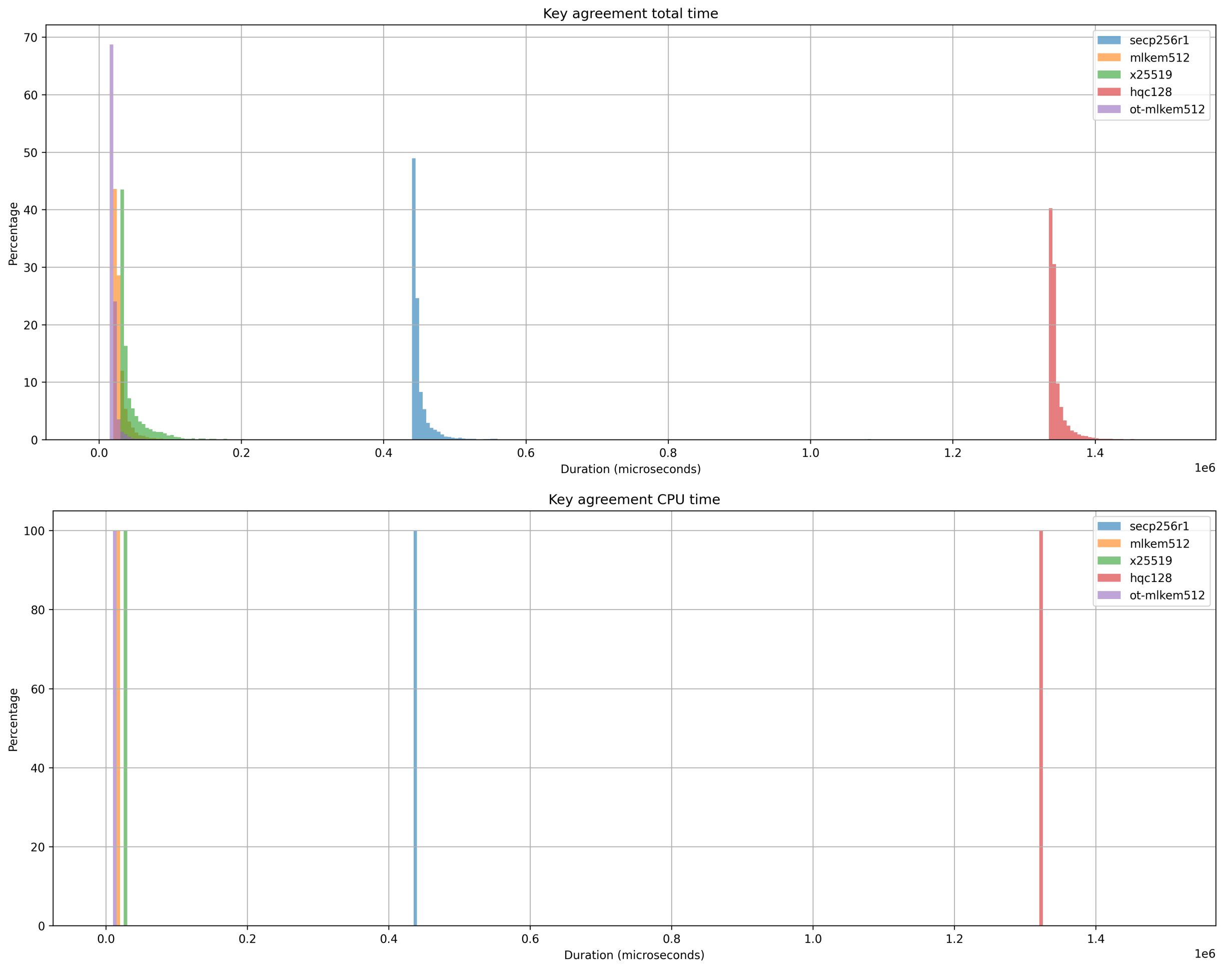

Figure 2.1: Key agreement performance (microseconds).

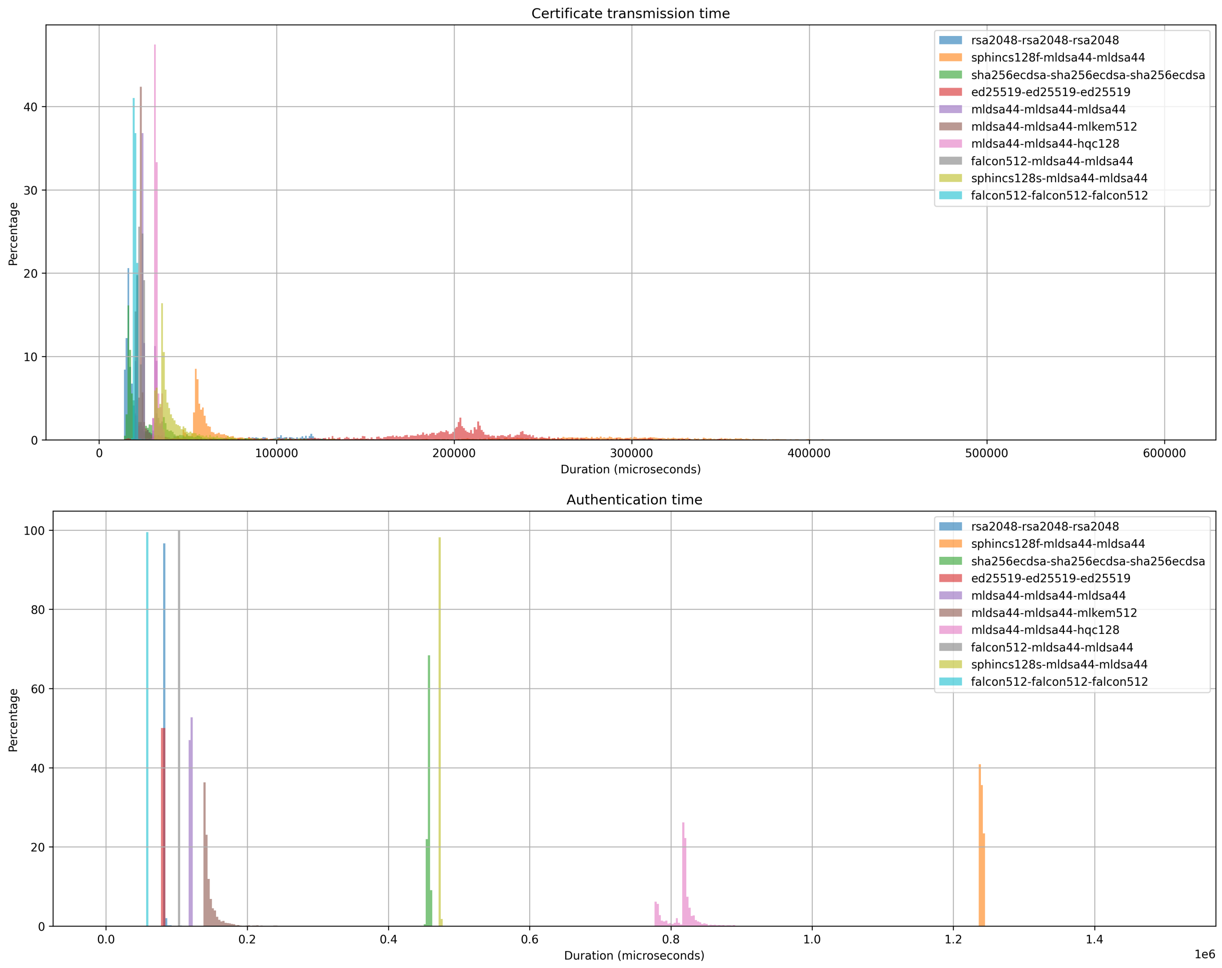

Figure 2.2: Authentication durations (microseconds).

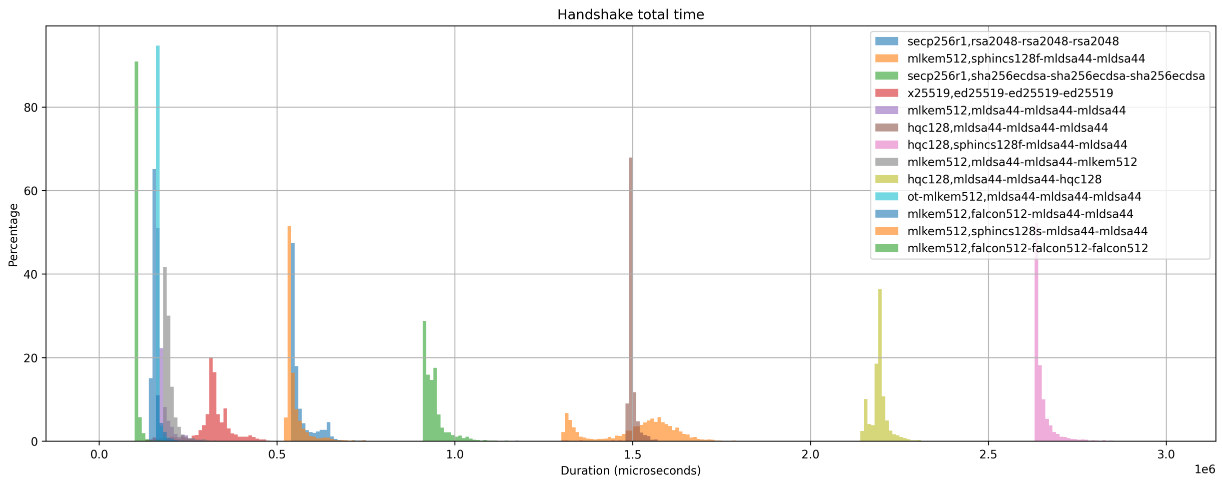

Figure 2.3: Handshake durations (microseconds).

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