KEMTLS vs. Post-Quantum TLS: Performance On Embedded Systems

Ruben Gonzalez[1] and Thom Wiggers[2]

[1] Neodyme AG, Garching, Germany
[2] Radboud University, Nijmegen, The Netherlands
mail+kemtls@ruben-gonzalez.de

Abstract. TLS is ubiquitous in modern computer networks. It secures transport for high-end desktops and low-end embedded devices alike. However, the public key cryptosystems currently used within TLS may soon be obsolete as large-scale quantum computers, once realized, would be able to break them. This threat has led to the development of post-quantum cryptography (PQC). The U.S. standardization body NIST is currently in the process of concluding a multi-year search for promising post-quantum signature schemes and key encapsulation mechanisms (KEMs). With the first PQC standards around the corner, TLS will have to be updated soon. However, especially for small microcontrollers, it appears the current NIST post-quantum signature finalists pose a challenge. Dilithium suffers from very large public keys and signatures; while Falcon has significant hardware requirements for efficient implementations.

KEMTLS is a proposal for an alternative TLS handshake protocol that avoids authentication through signatures in the TLS handshake. Instead, it authenticates the peers through long-term KEM keys held in the certificates. The KEMs considered for standardization are more efficient in terms of computation and/or bandwidth than the post-quantum signature schemes.

In this work, we compare KEMTLS to TLS 1.3 in an embedded setting. To gain meaningful results, we present implementations of KEMTLS and TLS 1.3 on a Cortex-M4-based platform. These implementations are based on the popular WolfSSL embedded TLS library and hence share a majority of their code. In our experiments, we consider both protocols with the remaining NIST finalist signature schemes and KEMs, except for Classic McEliece which has too large public keys. Both protocols are benchmarked and compared in terms of run-time, memory usage, traffic volume and code size. The benchmarks are performed in network settings relevant to the Internet of Things, namely low-latency broadband, LTE-M and Narrowband IoT. Our results show that KEMTLS can reduce handshake time by up to 38%, can lower peak memory consumption and can save traffic volume compared to TLS 1.3.

Keywords: Post Quantum Cryptography · KEMTLS · Transport Layer Security · Embedded Systems · Cortex-M4 · NIST PQC

1 Introduction

Transport Layer Security (TLS) is ubiquitous in modern computer networks. It adds confidentiality, authenticity and integrity to application-layer protocols. We trust it, among other things, with securing connections to websites, emails, instant messages and virtual private networks. In its most recent version, TLS 1.3 [31], ephemeral (elliptic curve) Diffie-Hellman is used to establish encryption keys. Server (and optionally client) authentication is achieved by using digital signatures. To verify the signatures, public keys are transmitted in certificates during the TLS handshake. These certificates are in turn signed by certificate authorities, which are pre-installed on the verifying device.

As TLS is an integral part of today's internet security architecture, it is vital to integrate post-quantum cryptography soon. This promises to mitigate the increasingly grave threat of large quantum computers to cryptography. The United States National Institute of Standards and Technologies (NIST) launched a multi-year standardization project for post-quantum algorithms [28]. The project is calling for key encapsulation mechanisms (KEMs) and digital signature algorithms that withstand large quantum computers.

While standardization of the primitives is ongoing, work on post-quantum TLS (PQTLS) has also begun. This began with academic experiments in 2015, demonstrating R-LWE key exchange in TLS 1.2 [5]. Like the previous work, many have focused on the ephemeral key exchange in TLS, often using so-called "hybrid" algorithms. These essentially perform a classic elliptic-curve key and a post-quantum key exchange in parallel, to increase the confidence in the security. Google and Cloudflare have already conducted large-scale industry studies employing hybrid algorithms within TLS [21–23]. Amazon already includes experimental support for post-quantum schemes in its S2N TLS implementation and its Key Management Services product [16]. While previous works have mainly focused on post-quantum confidentiality; there have been fewer experiments deploying post-quantum authentication. Sikideris et al. [36] have measured the performance of post-quantum signature schemes between servers in two data centers. They concluded that out of the (NIST Round 2) schemes they tested, only two (Falcon [30] and Dilithium [24]) seem viable for deployment in TLS 1.3. Experiments by Cloudflare [38], that added dummy data to TLS connections to measure the impact of the larger sizes of post-quantum signature schemes, seem to support these results. Still, when using Dilithium as a drop-in replacement for all of the signatures in TLS (which adds 17 kB to the handshake), Cloudflare reports an expected 60–80% slowdown for the Linux default congestion window of 10 packets. Falcon has much more favorable public key and signature sizes but requires hardware support for double-precision floating-point operations. Without this, Sikideris et al. report signing handshakes with Falcon is not viable.

There has also been some work investigating embedded devices rather than large-scale, high-performance computers. Bürstinghaus-Steinbach et al. have experimented with Kyber and stateless hash-based signature scheme SPHINCS$^+$. They integrated SPHINCS$^+$ in mbedTLS's [25] TLS 1.2 implementation and showed the performance on various Arm boards [6]. More recently, George et

al. have evaluated the performance of post-quantum TLS 1.3 on embedded systems [13]. They investigated the performance of the NIST finalist KEMs and the Dilithium and Falcon signature algorithms in WolfSSL's TLS 1.3 implementation.

To mitigate the difficulties with post-quantum signatures, Wiggers et al. proposed KEMTLS [33]. Instead of authenticating the handshake through a signature, KEMTLS performs authentication through KEM key exchange with KEM public keys in the certificates. As the KEMs currently considered for standardization are generally smaller and/or more computationally efficient than the post-quantum signature schemes, this can be more efficient. Additionally, it reduces the size of the trusted code base: the code that facilitates the ephemeral key exchange can also be used for authentication. Knowledge of the server's long-term key is imaginable in e.g. session resumption, or perhaps in IoT applications where the clients speak to a single server. We note that the certificates are still signed by a certificate authority using post-quantum signatures. We thus still need to verify post-quantum signatures; we might however choose signature algorithms that are optimized for size or verification time rather than signing time.

While KEMTLS and PQTLS have been compared, these studies focused on high-end hardware and high bandwidth connections [7, 33, 34]. However, TLS is used for more than just protecting web browsing on desktop computers. The Internet of Things (IoT) increasingly interconnects embedded devices over the internet. Especially device-to-cloud communication is an omnipresent IoT use case. New communication protocols like Matter [10] (formerly Connected Home over IP) mark a new trend by using IPv6 and forcing every embedded device to establish its own end-to-end-secure connection. From a security perspective, this makes perfect sense. However, for embedded software developers this poses a challenge. Key establishment, digital signatures and certificate transmission are already problematic for low-cost, resource-constrained devices. With the advent of post-quantum cryptography, it will become even more challenging to establish TLS connections from those embedded devices.

1.1 Contribution

This work investigates if KEMTLS' advantages transfer to the embedded realm by comparing KEMTLS and PQTLS in an embedded setting. For this purpose, KEMTLS and PQTLS were implemented including all NIST finalist signature schemes and KEMs, except for Classic McEliece which has too large public keys. As the PQTLS and KEMTLS implementation share large parts of their code base, a direct performance comparison is possible. Our analysis focuses on the relevant trade-offs embedded systems engineers face. To our knowledge, this is the first work to investigate KEMTLS in an embedded setting. We benchmark runtime, memory usage, code size and bandwidth consumption of our KEMTLS and PQTLS instantiations. The benchmark results were obtained by running our implementations on a Cortex-M4-based platform. Our experiments were conducted with a technology stack that is typically used in real-world deployments, in which the embedded device is a TLS client talking to a TLS

server running on a high-end computer. This computer also simulated different network technologies throughout the experiments. After a brief introduction of post-quantum cryptography, previous work and the ongoing standardization process, the PQTLS and KEMTLS protocols will be presented. The differences between these protocols will be outlined afterward. To support our results, the implementation and experimental setup will then be explained in detail. Finally, the results will be presented and concluded.

2 Background

In this section, we will give some background on the development of post-quantum cryptography, providing some history and summarizing the NIST standardization process. We will also detail the impact post-quantum cryptography has on TLS 1.3 and the development of KEMTLS.

2.1 Post-Quantum Cryptography

In 1994 Peter Shor published his famous quantum algorithms for discrete logarithms and factoring [35]. Virtually all of today's deployed public-key cryptography is based on the difficulty of computing discrete logarithms or integer factorization. Shor's algorithm, therefore, poses a severe threat to information security. This affects key-exchange methods and signature algorithms alike. Unfortunately, all of today's TLS key exchange and signature algorithms would be broken once an adversary has access to a large quantum computer. Moreover, advances in quantum computing give reason to believe that the arrival of large quantum computers is on the horizon [26,27]. Since development, standardization and adaptation of cryptographic algorithms is a slow process, preparations against quantum computers have to be started now.

The NIST standardization project for post-quantum cryptography started in 2017 [28]. From over 60 proposed candidates, four KEMs and three signature algorithms have proceeded as *finalists* to the competition's third round. There are an additional five KEMs and three signature algorithms still in the competition as *alternate candidates*. NIST has announced that they will select at most one of the lattice-based KEMs Kyber, SABER or NTRU as a standard, as well as one of the two lattice-based signature schemes Falcon and Dilithium. They will also be opening up an on-ramp for new signature schemes in the next round for signature schemes based on other assumptions.

The algorithms in the NIST competition propose parameters at three security levels, called *I*, *III* and *V*. Algorithms in these security levels should be at least as hard to break as AES-128, AES-192 and AES-256. Due to the resource constraints of embedded devices, we will only consider parameters of security level I.

NIST Finalists Three signature schemes remained as finalists in the third round of the NIST standardization project. Two of them are lattice-based constructions, and both were selected for standardization at the end of Round 3 in July 2022.

Falcon [30]'s security assumptions are based on NTRU, while **Dilithium** [24]'s assumptions are based on the Module-LWE and the Short Integer Solution problems. **Rainbow** [12] is a multivariate signature algorithm that is a variant of UOV [18]. Its security is based on the hardness of finding solutions to systems of equations in many variables over finite fields. Rainbow was recently broken by Beullens [4], and later eliminated from the standardization process. However, Rainbow is a good representative of UOV-based multivariate signature schemes in terms of size and performance. In these schemes, public keys are very large, but the very small signature sizes lead to interesting trade-offs. Since other UOV-like multivariate schemes will likely be proposed during NIST's next call for post-quantum signatures, we choose to include Rainbow in this our analysis.

Key encapsulation mechanisms (KEMs) are used for key exchange. Within TLS they can serve the same role as the Diffie-Hellman ephemeral key exchange. Similar to public-key encryption (PKE), a KEM public key is used to generate an encapsulated shared secret key. Only the corresponding private key can then decrypt this ciphertext into the right shared secret key.

Four KEMs proceeded as finalists into the third round of NIST's PQC competition. Among them are the lattice-based schemes **Kyber** [32], **Saber** [11] and **NTRU** [8]. Although these schemes are all based on lattices, their underlying lattice structure and implementation details differ substantially. **Classic McEliece** [3] is the only non-lattice-based finalist; its security relies on the hardness of decoding random linear codes instead. Code-based cryptography has been around since the 1970s. Therefore, it has a longer history of cryptanalysis than lattice-based cryptography. Because of this body of literature around Classic McEliece, it is often considered the most conservative choice. However, Classic McEliece's parameter choices make for very large public key sizes. Also in terms of speed, it can not compete with the lattice-based algorithms. Kyber was chosen to be the next NIST standard for key exchange in July 2022, while Classic McEliece was moved forward into the fourth round of the competition [2].

PQC on embedded devices Public-key cryptography was already challenging for embedded systems in a pre-quantum setting. The more expensive post-quantum algorithms will make this worse. To gain a better understanding of PQC algorithm performance on embedded systems, the PQM4 [17] project collects implementations for the Cortex-M4 platform and benchmarks them. Table 1 shows size and performance tradeoffs between the NIST PQC finalists based on numbers from [17] and [9]. Here it is important to mention that these numbers are accomplished on a clocked-down Cortex-M4. Using such a slowed-down embedded processor is customary for measuring algorithm run times because it avoids flash wait states. In a real deployment, code would be fetched from a fast ROM instead of a flash. For our experiments, we are not exclusively interested in PQC algorithm run-time, but in the performance of the overall system. Therefore, we do not clock down our CPU. The ramifications of this are detailed in subsection 3.1.

Table 1. Comparison of NIST PQC Round 3 finalists at security level I. We show the size (in bytes) of data transmitted during a handshake (public key, signature and ciphertext), offline data (secret keys) and operation timings (from [9,17]) on M4.

	bytes transmitted			**stored**	**computation (≈Kcycles)**		
Signatures	pubkey	sig.	sum	secret	keygen	sign	verify
Dilithium ★	1 312	2 420	3 732	2 528	1 597	4 095	1 572
Falcon ★	897	690	1 587	1 281	163 994	39 014	473
Rainbow †	161 600	66	161 666	103 648	94	907	238
KEMs	pubkey	ciph.	sum	secret	keygen	encaps	decaps
Kyber ★	800	768	1 568	1 632	440	539	490
NTRU †	699	699	1 398	953	2 867	565	538
SABER †	672	736	1 408	1 568	352	481	453

★: Scheme was selected for standardization.
†: Scheme was eliminated from the NIST standardization project.

2.2 Post-Quantum TLS

We will briefly explain how TLS 1.3 post-quantum can be made post-quantum and summarize the KEMTLS proposal for an alternatively authenticated TLS handshake.

(Post-quantum) TLS TLS is a protocol that has seen widespread deployment, famously as part of HTTPS. Its most recent iteration is TLS 1.3 [31]. In the most common uses, it offers unilateral authentication of the server to the client. Optionally, it also allows mutual, client-to-server authentication. The protocol authenticates the peers through signatures, which are in turn verified using public keys that are contained in (CA-signed) certificates. There is optional support for pre-shared, symmetric keys in place of certificate authentication as well.

The unilateral, certificate-authenticated TLS 1.3 handshake consists of an ephemeral, Diffie–Hellman (DH) key exchange followed by a signature over the handshake to authenticate. Finally, the handshake is additionally authenticated by a MAC. It is possible to make this handshake post-quantum, simply by replacing the server's DH key generation with $\mathsf{KEM_e.Encapsulate}$, to encapsulate against the client's ephemeral public key, and sending the ciphertext instead of the server's ephemeral DH public key. The client would derive the shared secret by decapsulating the ciphertext. For authentication, we simply use post-quantum signature algorithms in place of RSA or elliptic curve signatures. The TLS 1.3 handshake has been very carefully designed to be very efficient in the number of round-trips and is a single round-trip (1-RTT) protocol. As we can see on the left-hand side of Fig. 1, the server can already send data to the client in its first response flow. The client can send it first message after the server's first flow, after 1.5 RTT.

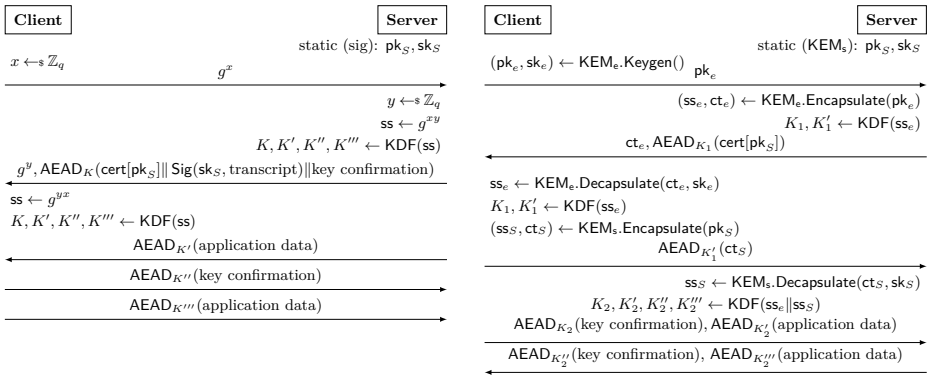


Fig. 1. Simplified protocol flow diagrams of: (left) the TLS 1.3 handshake, using signatures for server authentication; and (right) the KEMTLS handshake, using KEMs for server authentication.

KEMTLS The post-quantum KEMs and signature algorithms are further apart than their classic variants were. KEMTLS is an alternative proposal for a PQTLS handshake, which allows using KEMs (which are typically much smaller and/or more computationally efficient than post-quantum signatures) instead of signatures in the online handshake. In KEMTLS, the certificates contain public keys for a KEM instead of a signature scheme. There are still signatures for the verification of the certificate chain, but these only need to be verified. As those signatures are done offline, it is also possible to use algorithms optimized for public key and signature size, rather than signing time. For example, [33, Appendix D] gives parameters for such a variant of XMSS$^{\mathsf{MT}}$.

KEMTLS is inspired by the OPTLS proposal by Krawczyk and Wee [19]. OPTLS was an early proposal for TLS 1.3, where the authentication would be done via Diffie–Hellman key exchange. However, as observed by Kuhnen [20], OPTLS requires the non-interactive key exchange properties of DH, which KEMs do not offer. To authenticate a server via KEM, the client encapsulates a ciphertext to the long-term public key contained in the server's certificate. This would naively result in a 2-RTT protocol. KEMTLS avoids the performance penalty this would imply by observing that, in many applications including HTTP, for a *useful* response from the server, it first needs to receive a request from the client. For example, which page the client is requesting from the server. KEMTLS allows the client to send its request in the same flow as it would in TLS 1.3, returning the protocol to 1.5-RTT. This is achieved by encrypting the data with a key that is derived from both the ephemeral key exchange and the shared secret encapsulated to the server's long-term key. This key is *implicitly* authenticated, as the client can not be sure of the server's presence before it receives a message (`ServerFinished`) in which the server uses the encapsulated secret. The right-hand side of Fig. 1 describes a simplified version of a (unilaterally authenticated) KEMTLS handshake.

3 Experimental Setup

The following section describes the experimental setup used to acquire our results. Both protocols were benchmarked for handshake times, run-times of algorithms, peak memory usage, code size, and network traffic. Handshake times were measured in three network environments relevant to the IoT domain. This includes regular "broadband" internet, as well as two low-power wide-area network standards, LTE-Machine Type Communication (LTE-M) and Narrowband-IoT (NB-IoT), developed by the 3rd Generation network Partnership Project (3GPP). We give the characteristics employed for these environments in Table 2. While the performance characteristics of LTE-M and NB-IoT are based on numbers of the 3GPP [1], the broadband scenario is based on realistic round-trip times of client-to-cloud communication within West Europe using a consumer-grade connection [29].

Table 2. Connection Characteristics according to 3GPP [1]

Name	Abbrev.	Bandwidth	RTT time
Broadband	BB	1 Mbit	26 ms
LTE Machine Type Communication	LTE-M	1 Mbit	120 ms
Narrowband-IoT	NB-IoT	46 kbit	3 s

Cryptographic Primitives As KEMTLS is a post-quantum protocol, it is not specifically designed for transitional security. Although KEMTLS does not preclude their use, we do not consider mixed classic/post-quantum certificates or hybrid (post-quantum plus elliptic curve) key-exchange methods in our experiments. For comparability, our PQTLS implementation is also exclusively using post-quantum algorithms. We evaluated all combinations of NIST finalists, except for the KEM *Classic McEliece*. Classic McEliece's public keys are too large to fit into memory and do not fit in the `ClientHello`'s `KeyShareEntry` extension [31, Sec. 4.2.8].

Both KEMTLS and PQTLS make use of a certificate authority (CA) that signs certificates. The CA's certificate, containing the CA's public key used for signature verification, is stored on the client device. Only leaf certificates, transmitted by the server during the handshake, differ in KEMTLS and TLS 1.3. For PQTLS they include the public key of a signature algorithm, in KEMTLS a KEM public key.

We only evaluate primitives at the lowest security level, NIST level I. These are the smallest and most efficient parametersets.

3.1 Implementation

All benchmarks were conducted on a Silicon Labs STK3701A board, also known as the "Giant Gecko". This board was chosen because it features a 72 MHz ARM

Cortex-M4F embedded processor and offers large enough memory (2 MB flash storage, 512 kB SRAM) to fit Rainbow public keys. As Cortex-M4 is the designated NIST PQC reference platform for embedded devices, there are optimized assembly implementations available for most finalist algorithms. The PQM4 project collects these implementations and provides extensive benchmarks [17]. All PQC implementations used for benchmarking were taken from the PQM4 project. Only minor modifications, such as adding *verify* functions to signature schemes, fixing alignment issues and name-spacing symbol names had to be conducted. The code was compiled using GCC version 11.1, with the `-O3` speed optimization flag. In contrast to experiments run within the PQM4 project, we do not clock down the processor to avoid wait states. Instead, the processor runs at full speed. This makes sense since we are not exclusively interested in the run times of the primitives, but the performance of the overall system. Running the processor at full speed makes the PQC algorithms consume more cycles due to flash wait states and higher costs of memory accesses. However, since the PQC algorithms do not consume more wall-clock time, the actual handshake durations are not negatively affected. The Giant Gecko board exclusively takes on the role of an embedded (KEM)TLS client, wanting to connect to a backend server. To validate certificates send in the handshake, we flash the CA's root certificate into the Giant Gecko's persistent memory during setup. For efficiency, the CA directly signs the server's certificate. This avoids the need for transmitting intermediate CAs, reducing the size of the certificate chain. As both endpoints in embedded scenarios are usually under some level of manufacturer control, this is a common deployment. Communication to the backend server is done via the Giant Gecko's Ethernet port, which is directly connected to a high-end computer. This host computer simulates different network environments by using Linux's *netem* network emulation framework [15]. The network emulation framework is set up to throttle bandwidth and delay round trip times (RTTs) according to the aforementioned network environments. Wiggers' original KEMTLS implementation [33, 34] is used as server software. When running an iteration of the experiment, the corresponding PQC algorithms and the CA certificate are linked into the binary using Zephyr's *West* build tool. We then flash the binary onto the board via JLink. Benchmark results are received via serial communication.

Platform To have a realistic setup, we employed a typical embedded systems software stack. In our case that includes an embedded real-time operating system (RTOS) with an open-source TCP/IP stack and added TLS support. For reproducibility, we used the Apache-licensed Zephyr RTOS [39]. Zephyr supports over 200 boards and is backed by the Linux Foundation and multiple large corporations involved with developing embedded systems, such as NXP, NORDIC or Memfault. It provides its own optimized embedded network stack and allows cycle-accurate run time measurements (given a board's hardware supports it). Our application code runs as the exclusive Zephyr thread, eliminating scheduling costs. PQTLS and KEMTLS support was added to the operating system via a custom WolfSSL module. All code, including reproducible build system and server software, used

in this work is publicly available.[3] KEMTLS certificate generation was conducted using a customized Python script based on Wiggers et al. code. Post-quantum certificates for PQTLS were generated using a fork of OpenSSL's command-line tool maintained by the Open Quantum Safe project [37]. The TLS 1.3 cipher suite `TLS_CHACHA20_POLY1305_SHA256` was used in all experiments.

WolfSSL Integration Previous work [6,13] also uses WolfSSL for running benchmarks on embedded systems. We decided to use WolfSSL for the same reasons as the mentioned works and to make comparisons with our results easier. WolfSSL is designed to be memory efficient and fast on embedded systems. On top, it already supports TLS 1.3 and has a clean implementation of TLS's state machine. This makes it an ideal basis for implementing PQTLS and KEMTLS. Adding post-quantum algorithms to WolfSSL is straightforward. WolfSSL's crypto provider, called WolfCrypt, has a clean API that can be extended easily. As the KEM Kyber was already included in WolfSSL by [6], we did not need to make changes to the TLS 1.3 state machine. Apart from including the relevant ASN.1 object identifiers for KEMs and post-quantum signatures, only small changes such as increasing the maximum size of certificates had to be applied. Our embedded KEMTLS implementation is based on the same WolfSSL version as our PQTLS implementation. The majority of the code is identical in the PQTLS and KEMTLS implementation. However, adding support for KEMTLS to WolfSSL still required significant effort. Apart from altering the certificate/ASN.1 parser to allow KEM keys in certificates (and using those), WolfSSL's internal state machine, key derivations and state structures had to be modified. In both our PQTLS and KEMTLS experiments the client only performs signature verification, so no code for signing was linked into the final binary.

4 Results

For developers of embedded systems, the trade-offs between ROM (code size), RAM (memory usage), network traffic and CPU time (run-time of code) are most crucial. In this section, we present our findings regarding the consumption of these resources by KEMTLS and TLS 1.3 using NIST PQC finalists.

The run-time of algorithms impacts the device's energy consumption. This is especially relevant for battery-powered devices that rely on the possibility to hibernate when inactive. Network traffic also affects energy consumption, as operating an antenna is usually a very energy-consuming operation. Depending on the underlying wireless technology, network traffic can also be expensive in terms of network provider fees. Our results are representative for Cortex-M4-based platforms in general. Hence we focus on benchmarks that are independent of our specific evaluation board. As energy consumption varies heavily based on a board's design, choice of peripherals and transmission technology we did not

[3] Source code is available at https://github.com/rugo/wolfssl-kemtls-experiments/tree/paperv1.

include direct energy measurements into our results. Instead, we present code size, consumed memory, handshake traffic, handshake duration and run-time of PQC primitives. All KEMTLS and PQTLS instantiations were run 1000 times, with each run using a different CA and leaf certificate. The presented benchmarks are averaged over all runs. The NIST signature finalist Rainbow, which is included as a representative for multivariate-based cryptography, is only present in the KEMTLS results. This is because Rainbow public keys are very large. There was not enough memory to fit Rainbow as well as another signature scheme. It could therefore not be included into the PQTLS benchmarks. We emphasize that all employed PQC algorithms were optimized for speed, and not stack consumption.

4.1 Storage and Memory Consumption

Both protocol implementations are roughly the same size. Without post-quantum primitives, they have a code size of around 111 kB. Table 3 shows combinations of PQC algorithms with their measured code size. For KEMTLS, only instantiations with one KEM used for both ephemeral key exchange and authentication are shown. Including two KEMs does not give an advantage, but increases code size. However, for completeness, a table with all combinations can be found in Appendix A. Similarly, PQTLS instantiations with the same signature algorithm used for CA and leaf certificates are shown. Additionally, we include the combination of Dilithium and Falcon, where Dilithium is used as the handshake signature algorithm. This combination was suggested by Sikideris et al. [36] to make use of Dilithiums faster signing times for servers without hardware support for Falcon's double-precision floating-point operations.

The table also shows the PQC code's share of the overall code size as a percentage. Also included in Table 3 is memory consumption. Shown is the peak of consumed memory, in both heap and stack, during the handshake. This includes the memory consumed by the protocol implementation and PQC primitives.

In contrast to TLS 1.3, KEMTLS uses a KEM encapsulation instead of a signature verification to authenticate the connection. KEMTLS, therefore, needs code for KEM encapsulation, whereas TLS 1.3 does not. TLS 1.3 on the other hand needs the code for two distinct verification algorithms if different signature algorithms are used for CA and leaf certificates. Instantiations with NTRU ephemeral key exchange are notable outliers in terms of code size, requiring over 200 kB of code. This is in line with results reported by PQM4 [17]. Interestingly, this big increase in code size can not be observed when NTRU is used exclusively for authentication. This is because the client requires key generation and decapsulation code for ephemeral key exchange, whereas authentication via KEM only requires encapsulation functionality. Whenever Rainbow is used, the CA certificate containing a Rainbow public key takes up between 33% and 53% of the overall consumed storage space. This, however, does not disqualify Rainbow from usage on embedded systems, due to its small signature and very fast verification times (see Section 4.2).

Further, the results show that the lattice-based schemes perform well in terms of memory consumption. The consumed memory is mainly driven by stack usage

Table 3. Code and CA certificate sizes (and as percentage of total ROM size), and peak memory usage in the experiments. Parametersets used are NIST level I.

	KEX	Auth.	CA	PQC code (%)	CA size (%)	Memory
KEMTLS	Kyber	Kyber	Dilithium	29.0 kB (20.1%)	3.9 kB (2.7%)	49.7 kB
	Kyber	Kyber	Falcon	25.7 kB (18.6%)	1.7 kB (1.2%)	52.8 kB
	Kyber	Kyber	Rainbow	29.8 kB (9.8%)	161.8 kB (53.4%)	167.0 kB
	NTRU	NTRU	Dilithium	203.4 kB (63.9%)	3.9 kB (1.2%)	49.7 kB
	NTRU	NTRU	Falcon	200.0 kB (63.9%)	1.7 kB (0.6%)	52.8 kB
	NTRU	NTRU	Rainbow	204.0 kB (42.8%)	161.8 kB (33.9%)	182.9 kB
	SABER	SABER	Dilithium	31.5 kB (21.5%)	3.9 kB (2.7%)	49.7 kB
	SABER	SABER	Falcon	28.2 kB (20.0%)	1.7 kB (1.2%)	52.8 kB
	SABER	SABER	Rainbow	32.2 kB (10.5%)	161.8 kB (53.0%)	167.9 kB
PQTLS	Kyber	Dilithium	Dilithium	29.0 kB (20.1%)	4.0 kB (2.8%)	58.0 kB
	Kyber	Falcon	Dilithium	34.4 kB (23.0%)	4.0 kB (2.7%)	60.7 kB
	Kyber	Falcon	Falcon	25.8 kB (18.6%)	1.8 kB (1.3%)	56.2 kB
	NTRU	Dilithium	Dilithium	203.4 kB (63.8%)	4.0 kB (1.3%)	56.6 kB
	NTRU	Falcon	Dilithium	208.7 kB (64.4%)	4.0 kB (1.2%)	59.3 kB
	NTRU	Falcon	Falcon	200.1 kB (63.9%)	1.8 kB (0.6%)	54.8 kB
	SABER	Dilithium	Dilithium	31.5 kB (21.5%)	4.0 kB (2.7%)	58.0 kB
	SABER	Falcon	Dilithium	36.8 kB (24.2%)	4.0 kB (2.6%)	60.7 kB
	SABER	Falcon	Falcon	28.2 kB (20.0%)	1.8 kB (1.3%)	56.2 kB

of the PQC signature algorithms. Only Rainbow is an exception here. With a Rainbow-powered CA certificate, the very large public key has to be loaded into memory and held during signature verification. This requires a large allocation of heap space. In a custom certificate loader implementation it would be possible to store the public key in an already usable form in flash. Then the public key could directly be streamed in from flash (similar to [14]), without the need to hold it in memory entirely. However, since we present comparable results of reusable code, we did not include this kind of optimization for an individual algorithm.

4.2 Handshake Times

Apart from storage and memory consumption, handshake times are key in an embedded environment. Table 4 shows handshake times for different transmission technologies measured in millions of cycles. A complete table, with all possible instantiations, can be found in Appendix A. In Fig. 2 we show the handshake times and traffic for the broadband and NB-IoT scenarios. In a real deployment, the device would likely go into a low power mode or sleep instead of actively polling data during a slow transmission. This behavior however depends highly on the specifics of the embedded system and its transmission technology. Therefore, to achieve reproducible results, the CPU was running at a constant speed of 72MHz during all experiments. This also makes a direct translation to wall time possible. The table also shows the percentage of cycles spend on the underlying

PQC primitives. The remaining cycles are spent in the TLS state machine, memory operations or waiting for I/O.

Time spent in crypto operations is significant in the broadband and LTE-M setting. Whereas the NB-IoT transmission is so slow, that the share of cycles spent in cryptographic operations is very low (0.8% - 1.7%). In low-bandwidth/high-RTT settings like NB-IoT, the transmission size of certificates and public keys is the main driving factor of run time. Loading large public keys from storage into memory is a relevant factor as well, slowing down the otherwise fast Rainbow signature algorithm. Cycles spent to access memory and storage also become increasingly negligible when using slow transportation mediums. This is visible in Fig. 2b, where the instantiations with similarly sized handshake traffic clearly form clusters.

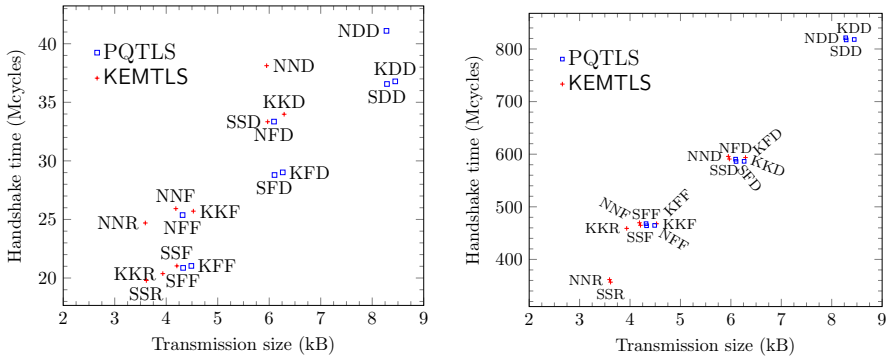
Table 4. TLS handshake traffic and runtime for various scenarios. Parametersets used are NIST level I.

				Handshake	Handshake time in Mcycles (% of crypto)		
	KEX	**Auth.**	**CA**	**traffic**	**BB (%)**	**LTE-M (%)**	**NB-IoT (%)**
KEMTLS	Kyber	Kyber	Dilithium	6.3 kB	17.1 (30.2%)	34.0 (15.2%)	593.6 (0.9%)
	Kyber	Kyber	Falcon	4.5 kB	12.3 (27.2%)	25.7 (13.0%)	467.8 (0.7%)
	Kyber	Kyber	Rainbow	3.9 kB	11.3 (25.1%)	20.4 (13.9%)	459.0 (0.6%)
	NTRU	NTRU	Dilithium	6.0 kB	21.3 (46.0%)	38.1 (25.6%)	595.8 (1.6%)
	NTRU	NTRU	Falcon	4.2 kB	16.6 (47.8%)	25.9 (30.6%)	469.7 (1.7%)
	NTRU	NTRU	Rainbow	3.6 kB	15.7 (47.4%)	24.7 (30.1%)	361.6 (2.1%)
	SABER	SABER	Dilithium	6.0 kB	16.3 (29.4%)	33.3 (14.4%)	590.8 (0.8%)
	SABER	SABER	Falcon	4.2 kB	11.6 (25.5%)	21.0 (14.1%)	464.8 (0.6%)
	SABER	SABER	Rainbow	3.6 kB	10.7 (23.1%)	19.8 (12.5%)	356.8 (0.7%)
PQTLS	Kyber	Dilithium	Dilithium	8.4 kB	19.9 (35.9%)	36.8 (19.5%)	818.1 (0.9%)
	Kyber	Falcon	Dilithium	6.3 kB	15.5 (33.0%)	29.0 (17.6%)	586.4 (0.9%)
	Kyber	Falcon	Falcon	4.5 kB	10.9 (30.1%)	21.0 (15.6%)	464.6 (0.7%)
	NTRU	Dilithium	Dilithium	8.3 kB	24.3 (47.6%)	41.1 (28.1%)	821.3 (1.4%)
	NTRU	Falcon	Dilithium	6.1 kB	19.9 (47.8%)	33.4 (28.5%)	590.6 (1.6%)
	NTRU	Falcon	Falcon	4.3 kB	15.2 (50.3%)	25.4 (30.2%)	468.0 (1.6%)
	SABER	Dilithium	Dilithium	8.3 kB	19.7 (35.2%)	36.6 (19.0%)	817.3 (0.8%)
	SABER	Falcon	Dilithium	6.1 kB	15.3 (32.0%)	28.8 (17.0%)	586.2 (0.8%)
	SABER	Falcon	Falcon	4.3 kB	10.7 (28.5%)	20.9 (14.6%)	464.0 (0.7%)

Both PQTLS and KEMTLS use a KEM for key exchange. While the performance of the module lattice KEMs Kyber and SABER is similar, they both outperform NTRU for this task. This is mainly due to the rather slow key generation of NTRU increasing handshake time. Slow key generation is also the reason why PQTLS and KEMTLS instantiations using NTRU have the highest percentage of cycles spent in PQC operations.

All KEMs outperform Dilithium when used for authentication. This makes sense as Dilithium's verify routine is slower than the encapsulation routine of all investigated KEMs. Dilithium's performance also suffers from its large public key and signature that increase the required transmission size. In slow,

bandwidth-constrained network environments, such as NB-IoT, this drawback becomes even more apparent. Rainbow performs well in terms of handshake times when used as a CA certificate. Not only because it has a fast, bitsliced Cortex-M4 implementation. Since the large Rainbow public key is stored on the client device, only the small signature has to be transmitted during the handshake. Rainbow's small signature and fast runtime make it a good fit for CA certificates if the storage and memory demands can be afforded. The instantiations with Rainbow offer the fastest KEMTLS handshake times throughout all transmission mediums. Additionally, the shortest NB-IoT handshake times use KEMTLS with Rainbow and SABER. Falcon on the other hand performs very well on the Cortex-M4 platform in our experiments. In terms of runtime, it even outperforms KEMs for server authentication. However, this is only true for the client side. Signing operations using Falcon are considerably more expensive than KEM decapsulations. But these operations are conducted on the server side, increasing server load, which is not part of our measurements. Additionally, Falcon's public key and signature sizes are comparable to the sizes of the KEM's public keys and ciphertexts. So it is not surprising that PQTLS instantiations using Falcon perform well. In the broadband and LTE-M setting, PQTLS with Falcon and SABER performs as well as KEMTLS with Rainbow and SABER.



(a) Broadband (b) Narrowband-IoT

Fig. 2. Handshake times and traffic for instantiations of KEMTLS and PQTLS. Letters represent the algorithms Dilithium, Falcon, Kyber, NTRU, Rainbow, and SABER in the roles of ephemeral key exchange, handshake authentication and CA, in that order.

5 Discussion

Our results show that KEMTLS with server-only authentication uses less memory than PQTLS and has similar code sizes. Due to Falcon's verification algorithm being very efficient, in terms of bandwidth and computation time, PQTLS with

Falcon performs as well as or better than any KEMTLS instantiation. The only exception are the KEMTLS instantiations using SABER or NTRU with Rainbow, where the ability of KEMTLS to use Rainbow due to lower memory usage saves a few bytes and thus become the best-performing instantiations in the NB-IoT scenario. Falcon also performs better than Dilithium on the client side, in any scenario.

Although we have not measured client authentication or an embedded server, we can extrapolate from our results. As reported by PQM4 [17] and Sikideris et al. [36], Falcon's signing algorithm, especially without hardware support, is significantly more costly than Dilithium's or any of the KEM operations. This suggests that Falcon is perhaps not generically suitable for post-quantum authentication.

Sikideris et al. also suggested a combination of Dilithium and Falcon for PQTLS, in scenarios where there is no hardware support for Falcon's double-precision floating-point operations. Dilithium would be put in the leaf certificate, to make use of its efficient signing times for online handshake signatures. Falcon's smaller public key and signature sizes would be beneficial for the CA certificate algorithm, which signs the leaf certificate only once, but the signature is transmitted many times. However, our results show that for embedded clients that only need to do signature validation Falcon is preferable over Dilithium, especially in very low bandwidth scenarios like NB-IoT.

6 Conclusion and Future Work

In this paper, we compared the performance of KEMTLS and TLS 1.3 using NIST PQC finalists in an embedded environment. This environment was represented by a Cortex-M4-based client communicating with a desktop-class server. We showed that a KEMTLS client consumes less memory than TLS 1.3, due to the smaller memory footprint of KEMs. The code size did not differ between KEMTLS and TLS 1.3. Since only server authentication was used, both protocols require a signature verify function and KEM for key exchange. Our run times show that in both protocols PQC primitives require a significant amount of computational time during the handshake, sometimes requiring over 50% of the entire handshake time. Even in the LTE-M setting, the percentage of cycles spent in PQC computations is considerable. However, in the bandwidth-constrained NB-IoT setting, handshake times are mostly driven by handshake size. In these conditions, Rainbow's very small signatures are an advantage. While Dilithium is generally outperformed by KEMs when used for authentication, Falcon performs very well due to its efficient verification algorithm. However, signing in Falcon is a very costly operation. Future work should therefore investigate KEMTLS and TLS 1.3 using client authentication, and embedded KEMTLS and post-quantum TLS 1.3 servers. In both of these applications, the embedded TLS 1.3 client needs to produce handshake signatures. This would increase the cost of using signatures instead of KEMs significantly, leading to new trade-offs. Another avenue of research is the pre-distributed key setting, where the client already

knows the server's public key. In this setting, bandwidth can be reduced even further, which may be compelling for the NB-IoT application.

Acknowledgements This work has been supported by Neodyme AG, the European Research Council through Starting Grant No. 805031 (EPOQUE) and by an NLnet Assure grant for the project "Standardizing KEMTLS".

References

1. 3rd Generation Partnership Project (3GPP): The mobile broadband standard specification release 13. Tech. rep., 3GPP (Sep 2015), https://www.3gpp.org/ftp/Information/WORK_PLAN/Description_Releases/Rel-13_description_20150917.zip
2. Alagic, G., Apon, D., Cooper, D., Dang, Q., Dang, T., Kelsey, J., Lichtinger, J., Miller, C., Moody, D., Peralta, R., Perlner, R., Robinson, A., Smith-Tone, D., Liu, Y.K.: Status report on the third round of the nist post-quantum cryptography standardization process. Tech. Rep. NISTIR 8413, National Institute of Standards and Technology (2022). https://doi.org/https://doi.org/10.6028/NIST.IR.8413
3. Albrecht, M.R., Bernstein, D.J., Chou, T., Cid, C., Gilcher, J., Lange, T., Maram, V., von Maurich, I., Misoczki, R., Niederhagen, R., Paterson, K.G., Persichetti, E., Peters, C., Schwabe, P., Sendrier, N., Szefer, J., Tjhai, C.J., Tomlinson, M., Wang, W.: Classic McEliece. Tech. rep., National Institute of Standards and Technology (2020), available at https://csrc.nist.gov/projects/post-quantum-cryptography/round-3-submissions
4. Beullens, W.: Breaking Rainbow takes a weekend on a laptop. Cryptology ePrint Archive, Report 2022/214 (2022), https://eprint.iacr.org/2022/214
5. Bos, J.W., Costello, C., Naehrig, M., Stebila, D.: Post-quantum key exchange for the TLS protocol from the ring learning with errors problem. In: 2015 IEEE Symposium on Security and Privacy. pp. 553–570. IEEE Computer Society Press (May 2015). https://doi.org/10.1109/SP.2015.40
6. Bürstinghaus-Steinbach, K., Krauß, C., Niederhagen, R., Schneider, M.: Post-quantum TLS on embedded systems: Integrating and evaluating kyber and SPHINCS+ with mbed TLS. In: Sun, H.M., Shieh, S.P., Gu, G., Ateniese, G. (eds.) ASIACCS 20. pp. 841–852. ACM Press (Oct 2020). https://doi.org/10.1145/3320269.3384725
7. Celi, S., Faz-Hernández, A., Sullivan, N., Tamvada, G., Valenta, L., Wiggers, T., Westerbaan, B., Wood, C.A.: Implementing and measuring KEMTLS. In: Longa, P., Ràfols, C. (eds.) LATINCRYPT 2021. LNCS, vol. 12912, pp. 88–107. Springer, Heidelberg (Oct 2021). https://doi.org/10.1007/978-3-030-88238-9_5
8. Chen, C., Danba, O., Hoffstein, J., Hulsing, A., Rijneveld, J., Schanck, J.M., Schwabe, P., Whyte, W., Zhang, Z., Saito, T., Yamakawa, T., Xagawa, K.: NTRU. Tech. rep., National Institute of Standards and Technology (2020), available at https://csrc.nist.gov/projects/post-quantum-cryptography/round-3-submissions
9. Chou, T., Kannwischer, M.J., Yang, B.Y.: Rainbow on cortex-M4. IACR TCHES **2021**(4), 650–675 (2021). https://doi.org/10.46586/tches.v2021.i4.650-675, https://tches.iacr.org/index.php/TCHES/article/view/9078
10. Connectivity Standards Alliance: Build with Matter (2022), https://buildwithmatter.com, [accessed 2022-05-16]

11. D'Anvers, J.P., Karmakar, A., Roy, S.S., Vercauteren, F., Mera, J.M.B., Beirendonck, M.V., Basso, A.: SABER. Tech. rep., National Institute of Standards and Technology (2020), available at https://csrc.nist.gov/projects/post-quantum-cryptography/round-3-submissions
12. Ding, J., Chen, M.S., Petzoldt, A., Schmidt, D., Yang, B.Y., Kannwischer, M., Patarin, J.: Rainbow. Tech. rep., National Institute of Standards and Technology (2020), available at https://csrc.nist.gov/projects/post-quantum-cryptography/round-3-submissions
13. George, T., Li, J., Fournaris, A.P., Zhao, R.K., Sakzad, A., Steinfeld, R.: Performance evaluation of post-quantum TLS 1.3 on embedded systems. Cryptology ePrint Archive, Report 2021/1553 (2021), https://eprint.iacr.org/2021/1553
14. Gonzalez, R., Hülsing, A., Kannwischer, M.J., Krämer, J., Lange, T., Stöttinger, M., Waitz, E., Wiggers, T., Yang, B.Y.: Verifying post-quantum signatures in 8 kB of RAM. In: Cheon, J.H., Tillich, J.P. (eds.) Post-Quantum Cryptography - 12th International Workshop, PQCrypto 2021. pp. 215–233. Springer, Heidelberg (2021). https://doi.org/10.1007/978-3-030-81293-5_12
15. Hemminger, S., Ludovici, F., Pfeiffer, H.P.: (Nov 2011), https://man7.org/linux/man-pages/man8/tc-netem.8.html, `man ip netem`
16. Hopkins, A.: Post-quantum tls now supported in AWS KMS. Amazon AWS Security Blog (Nov 2019), https://aws.amazon.com/blogs/security/post-quantum-tls-now-supported-in-aws-kms/, [accessed 2022-05-20]
17. Kannwischer, M.J., Rijneveld, J., Schwabe, P., Stoffelen, K.: PQM4: Post-quantum crypto library for the ARM Cortex-M4, https://github.com/mupq/pqm4
18. Kipnis, A., Patarin, J., Goubin, L.: Unbalanced Oil and Vinegar signature schemes. In: Stern, J. (ed.) EUROCRYPT'99. LNCS, vol. 1592, pp. 206–222. Springer, Heidelberg (May 1999). https://doi.org/10.1007/3-540-48910-X_15
19. Krawczyk, H., Wee, H.: The OPTLS protocol and TLS 1.3. In: 2016 IEEE European Symposium on Security and Privacy (EuroS&P). pp. 81–96 (2016). https://doi.org/10.1109/EuroSP.2016.18
20. Kuhnen, W.: OPTLS revisited. Master's thesis, Radboud University (2018), https://www.ru.nl/publish/pages/769526/thesis-final.pdf
21. Kwiatkowski, K., Langley, A., Sullivan, N., Levin, D., Mislove, A., Valenta, L.: Measuring TLS key exchange with post-quantum KEM (Aug 2019), https://csrc.nist.gov/Presentations/2019/measuring-tls-key-exchange-with-post-quantum-kem
22. Langley, A.: CECPQ2. ImperialViolet (Dec 2018), https://www.imperialviolet.org/2018/12/12/cecpq2.html, [accessed 2021-02-16]
23. Langley, A.: Real-world measurements of structured-lattices and supersingular isogenies in TLS. ImperialViolet (Oct 2019), https://www.imperialviolet.org/2019/10/30/pqsivssl.html, [accessed 2021-02-16]
24. Lyubashevsky, V., Ducas, L., Kiltz, E., Lepoint, T., Schwabe, P., Seiler, G., Stehlé, D., Bai, S.: CRYSTALS-DILITHIUM. Tech. rep., National Institute of Standards and Technology (2020), available at https://csrc.nist.gov/projects/post-quantum-cryptography/round-3-submissions
25. mbed TLS, https://www.trustedfirmware.org/projects/mbed-tls/, [accessed 2022-04-29]
26. Mosca, M.: Cybersecurity in an era with quantum computers: Will we be ready? Cryptology ePrint Archive, Report 2015/1075 (2015), https://eprint.iacr.org/2015/1075
27. Mosca, M., Piani, M.: Quantum threat timeline. Tech. rep., Globval Risk Institute (Oct 2019), https://globalriskinstitute.org/publications/quantum-threat-timeline/

28. National Institute for Standards and Technology: Submission requirements and evaluation criteria for the post-quantum cryptography standardization process (Dec 2016), https://csrc.nist.gov/CSRC/media/Projects/Post-Quantum-Cryptography/documents/call-for-proposals-final-dec-2016.pdf
29. Paul, S., Kuzovkova, Y., Lahr, N., Niederhagen, R.: Mixed certificate chains for the transition to post-quantum authentication in TLS 1.3. Cryptology ePrint Archive, Report 2021/1447 (2021), https://eprint.iacr.org/2021/1447
30. Prest, T., Fouque, P.A., Hoffstein, J., Kirchner, P., Lyubashevsky, V., Pornin, T., Ricosset, T., Seiler, G., Whyte, W., Zhang, Z.: FALCON. Tech. rep., National Institute of Standards and Technology (2020), available at https://csrc.nist.gov/projects/post-quantum-cryptography/round-3-submissions
31. Rescorla, E.: The Transport Layer Security TLS Protocol Version 1.3. RFC 8446, RFC Editor (Aug 2018). https://doi.org/10.17487/RFC8446
32. Schwabe, P., Avanzi, R., Bos, J., Ducas, L., Kiltz, E., Lepoint, T., Lyubashevsky, V., Schanck, J.M., Seiler, G., Stehlé, D.: CRYSTALS-KYBER. Tech. rep., National Institute of Standards and Technology (2020), available at https://csrc.nist.gov/projects/post-quantum-cryptography/round-3-submissions
33. Schwabe, P., Stebila, D., Wiggers, T.: Post-quantum TLS without handshake signatures. In: Ligatti, J., Ou, X., Katz, J., Vigna, G. (eds.) ACM CCS 2020. pp. 1461–1480. ACM Press (Nov 2020). https://doi.org/10.1145/3372297.3423350
34. Schwabe, P., Stebila, D., Wiggers, T.: More efficient post-quantum KEMTLS with pre-distributed public keys. In: Bertino, E., Shulman, H., Waidner, M. (eds.) ESORICS 2021, Part I. LNCS, vol. 12972, pp. 3–22. Springer, Heidelberg (Oct 2021). https://doi.org/10.1007/978-3-030-88418-5_1
35. Shor, P.W.: Algorithms for quantum computation: Discrete logarithms and factoring. In: 35th FOCS. pp. 124–134. IEEE Computer Society Press (Nov 1994). https://doi.org/10.1109/SFCS.1994.365700
36. Sikeridis, D., Kampanakis, P., Devetsikiotis, M.: Post-quantum authentication in TLS 1.3: A performance study. In: NDSS 2020. The Internet Society (Feb 2020)
37. The Open Quantum Safe project: Open Quantum Safe, https://openquantumsafe.org, [accessed 2022-05-20]
38. Westerbaan, B.: Sizing up post-quantum signatures (Nov 2021), https://blog.cloudflare.com/sizing-up-post-quantum-signatures/
39. Zephyr Project: Zephyr project, https://www.zephyrproject.org

A Extended Benchmark Tables

In Table 5 we report code sizes, CA certificate sizes and memory usage for all experiments we ran. Table 6 provides all results for the handshake traffic and handshake timing metrics.

Table 5. Code and CA certificate sizes (and as percentage of total ROM size), and peak memory usage in the experiments.

	KEX	Auth.	CA	PQC code (%)	CA size (%)	Memory
KEMTLS	Kyber	Kyber	Dilithium	29.0 kB (20.1%)	3.9 kB (2.7%)	49.7 kB
	Kyber	Kyber	Falcon	25.7 kB (18.6%)	1.7 kB (1.2%)	52.8 kB
	Kyber	Kyber	Rainbow	29.8 kB (9.8%)	161.8 kB (53.4%)	167.0 kB
	Kyber	NTRU	Dilithium	41.0 kB (26.3%)	3.9 kB (2.5%)	49.7 kB
	Kyber	NTRU	Falcon	37.7 kB (25.0%)	1.7 kB (1.1%)	52.8 kB
	Kyber	NTRU	Rainbow	41.7 kB (13.3%)	161.8 kB (51.4%)	182.9 kB
	Kyber	SABER	Dilithium	44.9 kB (28.1%)	3.9 kB (2.4%)	49.7 kB
	Kyber	SABER	Falcon	41.7 kB (26.9%)	1.7 kB (1.1%)	52.8 kB
	Kyber	SABER	Rainbow	45.7 kB (14.3%)	161.8 kB (50.8%)	167.9 kB
	NTRU	Kyber	Dilithium	216.3 kB (65.3%)	3.9 kB (1.2%)	49.7 kB
	NTRU	Kyber	Falcon	213.0 kB (65.4%)	1.7 kB (0.5%)	52.8 kB
	NTRU	Kyber	Rainbow	217.1 kB (44.3%)	161.8 kB (33.0%)	182.9 kB
	NTRU	NTRU	Dilithium	203.4 kB (63.9%)	3.9 kB (1.2%)	49.7 kB
	NTRU	NTRU	Falcon	200.0 kB (63.9%)	1.7 kB (0.6%)	52.8 kB
	NTRU	NTRU	Rainbow	204.0 kB (42.8%)	161.8 kB (33.9%)	182.9 kB
	NTRU	SABER	Dilithium	219.7 kB (65.6%)	3.9 kB (1.2%)	49.7 kB
	NTRU	SABER	Falcon	216.4 kB (65.7%)	1.7 kB (0.5%)	52.8 kB
	NTRU	SABER	Rainbow	220.4 kB (44.7%)	161.8 kB (32.8%)	182.9 kB
	SABER	Kyber	Dilithium	44.5 kB (27.9%)	3.9 kB (2.4%)	49.7 kB
	SABER	Kyber	Falcon	41.3 kB (26.8%)	1.7 kB (1.1%)	52.8 kB
	SABER	Kyber	Rainbow	45.3 kB (14.2%)	161.8 kB (50.8%)	167.9 kB
	SABER	NTRU	Dilithium	43.9 kB (27.6%)	3.9 kB (2.5%)	49.7 kB
	SABER	NTRU	Falcon	40.6 kB (26.4%)	1.7 kB (1.1%)	52.8 kB
	SABER	NTRU	Rainbow	44.6 kB (14.0%)	161.8 kB (50.9%)	182.9 kB
	SABER	SABER	Dilithium	31.5 kB (21.5%)	3.9 kB (2.7%)	49.7 kB
	SABER	SABER	Falcon	28.2 kB (20.0%)	1.7 kB (1.2%)	52.8 kB
	SABER	SABER	Rainbow	32.2 kB (10.5%)	161.8 kB (53.0%)	167.9 kB
PQTLS	Kyber	Dilithium	Dilithium	29.0 kB (20.1%)	4.0 kB (2.8%)	58.0 kB
	Kyber	Dilithium	Falcon	34.4 kB (23.3%)	1.8 kB (1.2%)	60.0 kB
	Kyber	Falcon	Dilithium	34.4 kB (23.0%)	4.0 kB (2.7%)	60.7 kB
	Kyber	Falcon	Falcon	25.8 kB (18.6%)	1.8 kB (1.3%)	56.2 kB
	NTRU	Dilithium	Dilithium	203.4 kB (63.8%)	4.0 kB (1.3%)	56.6 kB
	NTRU	Dilithium	Falcon	208.7 kB (64.9%)	1.8 kB (0.6%)	58.6 kB
	NTRU	Falcon	Dilithium	208.7 kB (64.4%)	4.0 kB (1.2%)	59.3 kB
	NTRU	Falcon	Falcon	200.1 kB (63.9%)	1.8 kB (0.6%)	54.8 kB
	SABER	Dilithium	Dilithium	31.5 kB (21.5%)	4.0 kB (2.7%)	58.0 kB
	SABER	Dilithium	Falcon	36.8 kB (24.6%)	1.8 kB (1.2%)	60.0 kB
	SABER	Falcon	Dilithium	36.8 kB (24.2%)	4.0 kB (2.6%)	60.7 kB
	SABER	Falcon	Falcon	28.2 kB (20.0%)	1.8 kB (1.3%)	56.2 kB

Table 6. TLS handshake traffic and runtime for various scenarios

	KEX	Auth.	CA	Handshake traffic	Handshake time in Mcycles (% of crypto) BB (%)	LTE-M (%)	NB-IoT (%)
KEMTLS	Kyber	Kyber	Dilithium	6.3 kB	17.1 (30.2%)	34.0 (15.2%)	593.6 (0.9%)
	Kyber	Kyber	Falcon	4.5 kB	12.3 (27.2%)	25.7 (13.0%)	467.8 (0.7%)
	Kyber	Kyber	Rainbow	3.9 kB	11.3 (25.1%)	20.4 (13.9%)	459.0 (0.6%)
	Kyber	NTRU	Dilithium	6.1 kB	17.1 (31.5%)	34.1 (15.8%)	592.2 (0.9%)
	Kyber	NTRU	Falcon	4.4 kB	12.4 (28.8%)	21.7 (16.4%)	466.2 (0.8%)
	Kyber	NTRU	Rainbow	3.8 kB	11.4 (27.0%)	20.5 (15.0%)	358.1 (0.9%)
	Kyber	SABER	Dilithium	6.1 kB	16.8 (30.0%)	33.6 (15.0%)	591.5 (0.8%)
	Kyber	SABER	Falcon	4.4 kB	12.0 (26.6%)	21.5 (14.9%)	465.4 (0.7%)
	Kyber	SABER	Rainbow	3.8 kB	11.0 (24.5%)	20.2 (13.4%)	357.4 (0.8%)
	NTRU	Kyber	Dilithium	6.1 kB	21.3 (44.8%)	38.2 (25.0%)	596.9 (1.6%)
	NTRU	Kyber	Falcon	4.4 kB	16.6 (46.4%)	25.9 (29.7%)	470.8 (1.6%)
	NTRU	Kyber	Rainbow	3.8 kB	15.5 (46.3%)	24.7 (29.1%)	462.4 (1.6%)
	NTRU	NTRU	Dilithium	6.0 kB	21.3 (46.0%)	38.1 (25.6%)	595.8 (1.6%)
	NTRU	NTRU	Falcon	4.2 kB	16.6 (47.8%)	25.9 (30.6%)	469.7 (1.7%)
	NTRU	NTRU	Rainbow	3.6 kB	15.7 (47.4%)	24.7 (30.1%)	361.6 (2.1%)
	NTRU	SABER	Dilithium	6.0 kB	20.8 (45.1%)	37.7 (24.9%)	594.9 (1.6%)
	NTRU	SABER	Falcon	4.2 kB	16.2 (46.6%)	25.6 (29.5%)	468.9 (1.6%)
	NTRU	SABER	Rainbow	3.6 kB	15.3 (46.3%)	24.3 (29.1%)	360.9 (2.0%)
	SABER	Kyber	Dilithium	6.1 kB	16.8 (29.4%)	33.6 (14.7%)	593.0 (0.8%)
	SABER	Kyber	Falcon	4.4 kB	11.9 (26.1%)	22.7 (13.7%)	466.8 (0.7%)
	SABER	Kyber	Rainbow	3.8 kB	11.0 (23.7%)	20.2 (12.8%)	458.3 (0.6%)
	SABER	NTRU	Dilithium	6.0 kB	16.8 (30.8%)	33.7 (15.3%)	591.5 (0.9%)
	SABER	NTRU	Falcon	4.2 kB	12.0 (27.9%)	21.5 (15.5%)	465.6 (0.7%)
	SABER	NTRU	Rainbow	3.6 kB	11.0 (25.8%)	20.2 (14.1%)	357.6 (0.8%)
	SABER	SABER	Dilithium	6.0 kB	16.3 (29.4%)	33.3 (14.4%)	590.8 (0.8%)
	SABER	SABER	Falcon	4.2 kB	11.6 (25.5%)	21.0 (14.1%)	464.8 (0.6%)
	SABER	SABER	Rainbow	3.6 kB	10.7 (23.1%)	19.8 (12.5%)	356.8 (0.7%)
PQTLS	Kyber	Dilithium	Dilithium	8.4 kB	19.9 (35.9%)	36.8 (19.5%)	818.1 (0.9%)
	Kyber	Dilithium	Falcon	6.7 kB	14.7 (35.4%)	31.0 (16.8%)	595.8 (0.9%)
	Kyber	Falcon	Dilithium	6.3 kB	15.5 (33.0%)	29.0 (17.6%)	586.4 (0.9%)
	Kyber	Falcon	Falcon	4.5 kB	10.9 (30.1%)	21.0 (15.6%)	464.6 (0.7%)
	NTRU	Dilithium	Dilithium	8.3 kB	24.3 (47.6%)	41.1 (28.1%)	821.3 (1.4%)
	NTRU	Dilithium	Falcon	6.5 kB	19.0 (50.3%)	35.3 (27.2%)	599.2 (1.6%)
	NTRU	Falcon	Dilithium	6.1 kB	19.9 (47.8%)	33.4 (28.5%)	590.6 (1.6%)
	NTRU	Falcon	Falcon	4.3 kB	15.2 (50.3%)	25.4 (30.2%)	468.0 (1.6%)
	SABER	Dilithium	Dilithium	8.3 kB	19.7 (35.2%)	36.6 (19.0%)	817.3 (0.8%)
	SABER	Dilithium	Falcon	6.5 kB	14.5 (34.2%)	30.7 (16.2%)	595.2 (0.8%)
	SABER	Falcon	Dilithium	6.1 kB	15.3 (32.0%)	28.8 (17.0%)	586.2 (0.8%)
	SABER	Falcon	Falcon	4.3 kB	10.7 (28.5%)	20.9 (14.6%)	464.0 (0.7%)