Implementing and Measuring KEMTLS

Sofía Celi[1](✉), Armando Faz-Hernández[1,2], Nick Sullivan[1,2], Goutam Tamvada[3], Luke Valenta[1,2], Thom Wiggers[4], Bas Westerbaan[5], and Christopher A. Wood[1,2]

[1] Cloudflare, Inc., Lisbon, Portugal
{sceli,armfazh,nick,lvalenta,chriswood}@cloudflare.com
[2] Cloudflare, Inc., San Francisco, United States
[3] University of Waterloo, Waterloo, Canada
goutam.tamvada@uwaterloo.ca
[4] Radboud University, Nijmegen, Netherlands
thom@thomwiggers.nl
[5] PQShield, Ltd, Oxford, UK
bas@westerbaan.name

Abstract. KEMTLS is a novel alternative to the Transport Layer Security (TLS) handshake that integrates post-quantum algorithms. It uses key encapsulation mechanisms (KEMs) for both confidentiality and authentication, achieving post-quantum security while obviating the need for expensive post-quantum signatures. The original KEMTLS paper presents a security analysis, Rust implementation, and benchmarks over emulated networks. In this work, we provide full Go implementations of KEMTLS and other post-quantum handshake alternatives, describe their integration into a distributed system, and provide performance evaluations over real network conditions. We compare the standard (non-quantum-resistant) TLS 1.3 handshake with three alternatives: one that uses post-quantum signatures in combination with post-quantum KEMs (PQTLS), one that uses KEMTLS, and one that is a reduced round trip version of KEMTLS (KEMTLS-PDK). In addition to the performance evaluations, we discuss how the design of these protocols impacts TLS from an implementation and configuration perspective.

Keywords: Post-quantum cryptography · KEMTLS · Transport Layer Security · Cryptographic engineering

1 Introduction

Transport Layer Security (TLS) is one of the most widely used protocols on the Internet today [11,22], and provides confidentiality, integrity, and authenticity to communications between two parties. The most recent version, TLS 1.3 [29], uses ephemeral (elliptic curve) Diffie-Hellman (-EC-DH) to establish keys, which are used to encrypt parts of the handshake and the traffic that will be sent in

Bas Westerbaan—Cloudflare, Inc, Amsterdam, Netherlands

P. Longa and C. Ràfols (Eds.): LATINCRYPT 2021, LNCS 12912, pp. 88–107, 2021.
https://doi.org/10.1007/978-3-030-88238-9_5

the connection. Authentication of the server and (optionally) of the client can be achieved by using digital signatures. The corresponding public keys for those signatures are transmitted during the handshake in digital certificates, which are signed by a certificate authority (CA).

Given that TLS 1.3 is the most widely used protocol today to secure connections [22], it is vital to start thinking about how to integrate post-quantum cryptography into it to protect from the imminent threat of quantum computing. Advances in quantum computing are promising and motivate a swift move to quantum-resistant algorithms. However, widespread adoption and protocol standardization are slow processes that can take several years to reach consensus among the parties[1]. In fact, the National Institute of Standards and Technologies (NIST) is organizing a multi-year competition to select post-quantum algorithms for standardization [27]. Several proposals on how to integrate post-quantum cryptography into TLS have already been suggested in the form of specifications, implementations, and experiments.

Related Work. Many early experiments focused on *transitional security* to protect against adversaries capable of recording today's communications with the hope of decrypting them in the future with a quantum computer. They focus on the key exchange phase of the handshake and add quantum-resistant confidentiality. This latter property is achieved by replacing the (EC-)DH key exchange by one based on a post-quantum Key Encapsulation Mechanism (KEM). However, this strategy does not address quantum-resistant authentication.

In 2016, a post-quantum experimentation project was initiated by Google [7], and was later expanded to a large-network scale in collaboration with Cloudflare in 2019 [21,24]. In the latter experiment, connections made from experimental versions of the Chrome browser to Cloudflare's edge servers used post-quantum key exchange algorithms in the TLS 1.3 handshake to secure connections and provide quantum-resistant confidentiality. The handshake used a "hybrid" key exchange protocol that combined post-quantum key exchange algorithms with traditional algorithms in order to safely use experimental cryptography without sacrificing any security guarantees. The experiment included two hybrid post-quantum key exchange protocols: X25519 [5] with the lattice-based KEM NTRU-HRSS [14] and X25519 with the supersingular-isogeny-based KEM SIKE [16]. These experiments focused on post-quantum *confidentiality*, but still relied on traditional authentication using non-quantum-resistant digital signature algorithms.

From a specification level, these works on quantum-resistant *confidentiality* mechanisms have taken priority [8,13,19,31,32,37,42], without much actual integration into real-world systems.

While these previous experiments provided valuable insights about the performance impact of post-quantum cryptography in real networks, post-quantum confidentiality is only one part of the picture: full post-quantum security also requires post-quantum authentication. In this sense, there are some research efforts towards this goal by using post-quantum signatures. But, most post-quantum signature schemes participating in the NIST competition have large

[1] It took, for example, 5 years to standardize TLS 1.3 [39].

public keys or signatures, and/or have significant performance considerations in their cryptographic operations. Sikeridis et al. [35] suggest that only the lattice-based candidates Dilithium [25] and Falcon [28] are viable contenders to be used in the TLS handshake, taking into account the trade-off between lengthy signatures and computationally heavy cryptographic operations.

Post-quantum KEM operations are, in practice, more efficient than post-quantum signature operations. A new approach, called KEMTLS [33], achieves authentication using KEMs instead of relying on digital signatures. This technique consists of encapsulating under the long-term KEM public key advertised in the peer's certificate, obtaining a shared secret in the process. Only the peer that has the private key corresponding to the public key in the advertised certificate can decapsulate the shared secret and decrypt any encrypted data sent under that key. Thus, KEMTLS uses post-quantum KEMs for both *confidentiality* and *authentication* to achieve full post-quantum security. A tweaked version of KEMTLS, called KEMTLS-PDK [34], achieves the same properties while reducing the number of round-trips needed.

Contributions. The focus of this paper is analyzing how the integration of post-quantum cryptography impacts the TLS 1.3 handshake from a performance, implementation, and configuration perspective. We developed a framework for establishing TLS 1.3 handshakes using post-quantum algorithms on a real-world system: a distributed network that is subject to actual Internet traffic conditions and that spans two continents. We examined several handshake configurations: one that uses KEMs for confidentiality and post-quantum signature schemes for authentication, which we called PQTLS; and we evaluate the KEMTLS protocol and its reduced round trip version called KEMTLS-PDK. We measured the latency of these handshakes and compare them against the baseline TLS 1.3 handshake by considering both server-only and mutual authentication. Additionally, we touch upon the engineering process of implementing all these protocols in the Go language, and report some constraints found in the design of KEMTLS. Our implementations are publicly available for further experimentation.

Organization. In Sect. 2, we describe the integration of post-quantum algorithms into the TLS 1.3 handshake. Section 3 covers details of our implementation and our integration into the testbed network used for experimentation. In Sect. 4, we discuss our experimental methodology and measurement results, and finally in Sect. 5, we state our conclusions.

2 Post-quantum Cryptography in TLS 1.3

We first give an overview of the TLS 1.3 handshake, and then discuss proposed specifications, implementations, and experiments for integrating the PQTLS, KEMTLS and KEMTLS-PDK post-quantum handshakes.

2.1 Reviewing the TLS 1.3 Protocol

Standardized in 2018, the TLS 1.3 protocol emerged in response to dissatisfaction with the outdated design of the TLS 1.2 handshake, its two-round-trip overhead, and the increasing number of practical attacks on older versions of TLS [1–3,6]. The pressure to increase efficiency also motivated the creation of alternative protocols such as the QUIC protocol [15]. In light of this, the main improvements of TLS 1.3 are: reducing the handshake's latency, encrypting as many messages as possible of the handshake itself, improving resilience to cross-protocol attacks, and removing legacy features [39]. It achieves a one-round-trip time (1-RTT) handshake and even a 0-RTT handshake through a resumption mode.

The default[2] mode of the protocol uses certificates for authentication and (EC-)DH for shared secret generation. In this mode, the handshake starts with the client sending a `ClientHello` (`CH`) message to the server. This message advertises the supported (EC-)DH groups and the ephemeral (EC-)DH keyshares offered by the client and specified in the `supported_groups` and `key_shares` extensions, respectively. The `CH` message also advertises the signature algorithms supported in the `signature_algorithms` extension. It also contains a nonce and a list of supported symmetric-key algorithms (ciphersuites).

The server processes the `ClientHello` message and chooses the appropriate cryptographic parameters to be used in the connection. If (EC-)DH key exchange is in use (meaning the client sent the `key_shares` extension), the server sends a `ServerHello` (`SH`) message containing a `key_share` extension with the server's (EC-)DH key corresponding to one of the `key_shares` advertised by the client. The `SH` message also contains a server-generated nonce and the ciphersuite chosen.

An ephemeral shared secret is then computed at both ends (the client computes it when it receives `SH`). After this point, all subsequent handshake messages are encrypted using keys derived from this secret.

The server then sends a certificate chain (`ServerCertificate` message) and a message that contains a proof that the server possesses the private key corresponding to the public key advertised in its leaf certificate. This proof is a signature over the handshake transcript and it is sent in the `ServerCertificateVerify` message. The advertised `signature_algorithms` in `CH` are used to decide which algorithms can be used to generate this signature. The goal of this message is to provide proof of possession of the server's private key, which is essential for achieving authentication. The server also sends the `ServerFinished` message that provides integrity of the handshake up to this point. It contains a message authentication code (MAC) over the entire transcript providing key confirmation and binding the server's identity to any computed keys.

Optionally, the server can send a `CertificateRequest` message, prior to sending its `ServerCertificate` message, requesting a certificate from the client for authentication. At this point, the server can immediately send application data to the unauthenticated client. Upon receiving the server's messages, the client verifies the signature of the `ServerCertificateVerify` message and the MAC of the `ServerFinished` message. If requested, the client must respond with their own authentication messages, `ClientCertificate` and `ClientCertificateVerify`, to achieve mutual authentication. Finally, the client

[2] Advanced modes of the TLS 1.3 handshake can also use a pre-shared key (PSK) exchange, PSK with ephemeral key exchange, and password-based authentication.

must confirm their view of the handshake by sending a MAC over the handshake transcript in the `ClientFinished` message.

It is only after this process that the handshake is completed, and the client and server can derive the keying material required by the record layer to exchange application data protected with authenticated encryption.

2.2 PQTLS: Signed Post-quantum TLS 1.3

A variety of specifications, implementations and experiments explain how to integrate post-quantum cryptography into the TLS 1.3 handshake. Regarding the post-quantum key exchange phase of TLS 1.3 (without addressing post-quantum authentication), several Internet-Drafts are proposed [13,19,31,37,42], as well as some experimental demonstrations [9,21,23,24]. On the other hand, fewer works have focused on post-quantum authentication. In [18,35], the authors recommended that the adoption of at least two post-quantum signature algorithms is viable for the TLS 1.3 handshake.

There are no theoretical obstacles for transitioning TLS 1.3 to a post-quantum world. One can use post-quantum signature algorithms for authentication and the (EC-)DH key exchange can be replaced by a post-quantum KEM; we call this approach PQTLS.

In practice, however, this replacement is not so simple. CAs must adapt their software to include post-quantum signatures, and, historically, the Web Public Key Infrastructure (PKI) and other X.509 PKIs have limited which algorithms can be used. It could take a long time until new algorithms are widely deployed. These changes may occur in the future but, for the purpose of experimentation and rapid deployment, these issues become limitations.

We propose a practical approach for overcoming this problem. Specifically, we rely on a delegation mechanism for credentials. A Delegated credential (DC) is an authenticated credential valid for a short period (at most 7 days) that can be used to decouple the handshake authentication algorithm from the authentication algorithms used in the certificate chain: a delegated credential can contain an algorithm to be used in the handshake and, in turn, it is cryptographically bound to the end-entity certificate as it is authenticated by it. The process of authenticating the DC is executed at the TLS stack level.

Using DCs in itself does not give us full post-quantum security, but it allows us to support post-quantum authentication algorithms that are not supported by existing CAs. An existing certificate is used to authenticate this delegated credential (by signing in a classical way in our experiments), and the advertised algorithm in the DC is used to authenticate the handshake.[3] The Internet

[3] Authentication is as strong as its weakest link, so until the entire certificate chain has post-quantum security we do not have a fully post-quantum authenticated protocol. However, the approach suffices for the purpose of our experiments.

Engineering Task Force (IETF) draft describing this technique, "Delegated Credentials for TLS" [4], is on track for standardization.[4]

Using delegated credentials comes with other advantages for our cases. Unlike a regular certificate, a delegated credential is smaller and has no other extensions, such as revocation lists and certificate statuses, which makes it a perfect fit for experiments where the size of parameters is important. Also, DCs are validated only at the TLS stack level, which reduces the number of codebases or systems where we needed to roll out new algorithms.

If full post-quantum security is wanted, the whole certificate chain will need to contain post-quantum algorithms. A peer wanting to authenticate another peer with its certificate (and the public key in it) in the TLS 1.3 handshake requires confidence that the associated private key is owned by the certificate owner's peer. This confidence is obtained through the use of public key certificates that bind these values to an identity. A CA signs certificates after asserting proof of possession of the private key. If the peer does not hold the public key of the CA that signed the other peer's certificate, then it might need an additional certificate to obtain that public key. These certificates are called 'intermediates'.

For a client to authenticate a server it uses this chain of certificates: a root CA's one, followed by at least one intermediate CA certificate, and then the leaf certificate of the server. Certificates can be cached, pre-installed or suppressed, which means that less data needs to be transmitted during the handshake; but these mechanisms are not widely deployed. In turn what this means is that for a full post-quantum TLS 1.3 handshake, peers will need to transmit the whole certificate chain and verify all their authentication proofs (at least three signatures or other proofs of authentication). If a DC is used in this scenario, data transmitted is increased, as well of the number of authentication operations.

2.3 KEMTLS: KEMs Everywhere

Using post-quantum signatures for authentication comes with another challenge. The proposed signature schemes participating in the NIST post-quantum competition have public keys or signatures much larger than their classical counterparts. For most algorithms, this size increase for post-quantum signatures is bigger than for post-quantum KEMs. The large size of cryptographic material can become an issue in the PQTLS scenario.

KEMTLS suggests the use of KEMs as the primary asymmetric building block for both the key exchange and authentication phases of the TLS 1.3 handshake. Its goal is to achieve a TLS 1.3 handshake that provides full post-quantum security (confidentiality and authentication) in an efficient way. KEMs instead of signatures are used for authentication because the KEM's public keys and ciphertexts are smaller.

[4] While it is stated in the draft that the DC signature algorithm *"is expected to be the same as the sender's CertificateVerify.algorithm"*, this is not a hard requirement, and in KEMTLS the Certificate Verify messages are not sent.

Like in PQTLS, the client advertises their support of post-quantum KEMs as part of the `supported_groups` extension, and their supported ephemeral KEM public keys as part of their `key_shares` extension. Support for KEMTLS authentication, via KEM leaf certificates or DCs with KEMs, is indicated by including KEMs in the `signature_algorithms` extension.

The server, in turn, determines the appropriate cryptographic parameters to be used in the connection, and replies with a ciphertext: an encapsulation against one of the advertised ephemeral KEM public keys of the `ClientHello` message. The encapsulation generates a second output: an unauthenticated ephemeral shared secret. From this point onward, all subsequent messages will be encrypted under the secret, after applying the appropriate key schedule operations. The server also sends its certificate chain (`ServerCertificate` message): the leaf certificate (or DC) should advertise a post-quantum KEM public key. Optionally, the server can send a `CertificateRequest` message, which is sent prior to the `ServerCertificate` message, asking the client to authenticate.

Contrary to TLS 1.3, the server cannot provide explicit proof of possession (using digital signatures) of the private key corresponding to the public key advertised as part of the leaf certificate (or DC). Instead, in KEMTLS, the client must receive the `ServerCertificate` message first, and reply with the encapsulation of the public key advertised in it. This encapsulation (a ciphertext) is sent as part of a new TLS message called `ClientKEMCiphertext`. The KEMTLS handshake diverges from the TLS 1.3 standard, as the server must wait for this message adding another flight or half round-trip to the protocol.

The second output of the client key encapsulation is an implicitly authenticated shared secret. This secret is mixed into the key schedule operations and will afterwards be used to encrypt all subsequent messages. Only the intended server can decrypt any messages encrypted under this key. By being able to do so, the server proves possession of the private key corresponding to the public key in it's certificate. If the server did not request client authentication (server-only authentication), the client can immediately send their `ClientFinished` message in this flight, which contains a MAC over the entire transcript. The client can also send at this point application data, which is implicitly authenticated, and has slightly weaker downgrade resilience and forward secrecy compared to when digital signatures are used.

When receiving the `ClientKEMCiphertext` message and decapsulating their parameters, the server can send their confirmation message `ServerFinished`, authenticating the handshake transcript. In the same flight, the server can now send application data encrypted by the shared secret of the decapsulation mechanism. Once the client receives and verifies the `ServerFinished` message, the server is explicitly authenticated, and the handshake has full downgrade resilience and strong forward secrecy.

Ciphersuite Negotiation and Middlebox Compatibility. TLS 1.3 allows clients and servers to negotiate the used algorithms. For key exchange, the supported algorithms are advertised in the `supported_groups` extension. For authentication, the mandatory `signature_algorithms` extension contains a list of algorithms that can be used by the peer to pick the appropriate certificate advertised by the corresponding peer. Post-quantum KEMs can simply be added to these lists and negotiated accordingly.

Any compliant TLS 1.3 implementation that does not understand or wish to negotiate KEMTLS will simply ignore any advertised post-quantum KEMs for the key exchange, and will not send a leaf certificate (or DC) with a KEM public key. As all messages following `ServerHello` are encrypted, changes in the protocol should be opaque to any non-decryption traffic interception; otherwise, a barrier on its adoption will be observed, similar to the "Middlebox" issues that arose when moving from TLS 1.2 to TLS 1.3 [20,38]. Issues may still arise if traffic interception servers enforce stricter constraints on key sizes than those required by the TLS 1.3 standard; these kinds of issues are harder to control.

Mutual Authentication. TLS 1.3 requires that *"the client's identity should be protected against both passive and active attackers"* [29, Sec. E.1]. Thus, both TLS 1.3 and KEMTLS cannot send the client's certificate (its identity) before the server has been authenticated. In TLS 1.3, the client can authenticate to the server, after receiving a request to do so from it, by providing its certificate and a signature over the handshake transcript.

In the sketch of client authentication in KEMTLS [33, App. C], upon request from the server, the client responds with the `ClientCertificate` message, where the leaf certificate (or DC) must contain a post-quantum KEM public key. This message must be sent in the same flight as when the `ClientKEMCiphertext` message is sent (but after it). In turn, the server sends the `ServerKEMCiphertext` message containing an encapsulation against the client certificate's KEM public key after processing the `ClientKEMCiphertext` and `ClientCertificate` messages. The client must wait for a `ServerKEMCiphertext` message from the server prior to sending their `ClientFinished` or any other message. Therefore, the client proves their identity by showing that both sides can arrive to the same shared key: the output of the encapsulation of the client's public key sent in the leaf certificate (or DC). Finally, once the server receives the `ClientFinished`, it can send `ServerFinished`, which achieves full downgrade resilience and forward secrecy.

The straightforward addition of these messages adds a round-trip to the handshake, as they can not be sent until the server has been authenticated. This extra round does not occur in the TLS 1.3 handshake because an explicit proof of authentication (the signature) is sent in the same flight as the certificate.

For a practical instantiation for our experiments, we use classically signed DCs that wrap KEM public keys to provide certified KEM keys.

2.4 KEMTLS-PDK: Reducing Round Trips

KEMTLS-PDK is a technique that relies on pre-distributed keys and has the goal of improving KEMTLS round-trips. It assumes the client knows the server's public key beforehand. This is not an uncommon situation as, for example, web browsers cache certificates of frequently accessed servers, mobile apps pin certificates, or server certificates are pre-distributed through DNS [17].

During the handshake, servers can authenticate earlier to the client, when KEM authentication keys are pre-distributed. We implement this mechanism using the TLS cached information extension[5] [30], so the client sends an encapsulation against the server's public KEM key in the first flight (alongside the `ClientHello` message: either as a separate message or as an extension for it). This allows the server to be explicitly authenticated by sending `ServerFinished` in the first message to the client and to immediately send application data.

On the other hand, the situation is more complex for achieving earlier client authentication since the client has to *proactively know* that the server will ask for its authentication. Nonetheless, this assumption does occur in certain applications such as in virtual private networks (VPN), where the client could send the certificate as early as possible.

Recall that for privacy reasons, TLS 1.3 requires that the server must be authenticated prior to transmitting the client certificate, and that this certificate must be sent encrypted. For the former requirement, KEMTLS-PDK assumes the client knows the server's certificate so it is sent after the `ClientKEMCiphertext` message in the first flight (as a separate message from the `ClientHello` one). For the latter requirement, the client certificate is encrypted under the shared secret resulting from the encapsulation mechanism used for `ClientKEMCiphertext`. Thus, it is possible to remove a full round-trip from KEMTLS with mutual authentication.

Early client authentication can be secured by caching a `CertificateRequest` message using the TLS cached information extension. The client certificate will then contain a key with an authentication algorithm that is likely known to be supported by the server. However, further investigation is needed for coming with a mechanism to encrypt the client's certificate.

3 Implementation Details

3.1 Implementation in Go

Go is a high-level programming language with support for the TLS protocol (including version 1.3). Its standard library is open, which allowed us to made modifications to its internals without requiring third-party libraries. While Go is well-known for developing web server applications, it also has mechanisms to interact with low-level features of the computer architecture. This is particularly useful for accessing architecture-specific capabilities, which are only available through assembler code.

Some implementations of post-quantum algorithms are available. The teams currently contending at the ongoing NIST's post-quantum competition provide implementations in C/C++. The Open-Quantum Safe [36] project wraps C implementations to run in Go through the cgo programming interface. However, performance degradation in it can be observed due to this wrapping procedure. The CIRCL [10] library implements a number of post-quantum algorithms natively in

[5] This extension is only available for TLS 1.2, so we adapted it to be used in TLS 1.3.

Go, including SIDH and SIKE [16]. As part of our contributions, we integrate to the CIRCL library AVX2-optimized implementations of the Dilithium signature scheme (round 2) and the Kyber key encapsulation mechanism.

Go provides a clean implementation of TLS 1.3. However, the implementation is conservative in regards to the type of extensions and algorithms that it supports. Changing the TLS 1.3 implementation to include delegated credentials and PQTLS required including some extensions and adding certain algorithm identifiers. It also meant adding a way for generating and validating delegated credentials, as well as adding the ability to include the delegated credentials X.509 extension to generated certificates. We also added the cached information extension [30] and modified it to work with TLS 1.3 for KEMTLS-PDK.

Integrating KEMTLS and KEMTLS-PDK was more challenging. Doing so required the interruption of the handshake's flow depending on whether there is cached information, whether it is server-only authentication, or whether it is mutual authentication. As noted, the flow of messages in KEMTLS and KEMTLS-PDK is different depending on the authentication modes: server-only, mutual or with cached information. This differs from the standard TLS 1.3 handshake that follows the same flow of messages regardless if server-only or mutual authentication is performed. These differences were an important lesson learned during our implementation as it was often a source of errors.

We made available all of these modifications in a fork of Go at https://github.com/cloudflare/go/tree/cf-pq-kemtls. This code integrates CIRCL and can be used as a replacement of the standard Go to compile other Go programs. Hence, anyone wanting to use post-quantum algorithms or the new handshake protocols can benefit from our code by compiling programs with our modified Go.

3.2 A Testbed Network

To test and measure TLS connections, we looked for a service that operates under common Internet conditions and spans across different geographical locations. We chose Drand [40], a distributed randomness beacon written in Go, as the target of our experimentation. In this network, Drand servers are linked so they can collectively produce publicly-verifiable random numbers at fixed intervals of time. A threshold signature scheme prevents collusion or biasing the generation of numbers. Nodes in the network communicate with one another using a gRPC protocol [26] with TLS authentication. Additionally, the Drand service exposes public randomness through an HTTPS endpoint.

Changes in the Drand code base are minimal. We needed to provide and configure a certificate with the DCs extension enabled for servers and clients. We also needed to state which protocol will be initiated (KEMTLS or PQTLS) by stating so at the TLS configuration level. If KEMTLS-PDK wanted to be used, a "regular" KEMTLS handshake is first run, information is cached (the `ServerCertificate` message), and then cached information is used in a fresh KEMTLS-PDK handshake by configuring it at the TLS configuration level. We added those configuration options for ease of experimentation: in a more realistic scenario stating which key exchange and signature algorithms are supported should be enough to trigger the appropriate protocol execution.

At run time, fresh delegated credentials are generated each time that a request arrives. However, these credentials can be further cached and stored so they can be reused between connections. A mechanism that routinely checks the validity of these credentials can also be implemented. This shows that delegated credentials can be easily implemented and used without needing to constantly modify certificate storage or retrieval. It is worth noting that adding delegated credentials increases the number of validations that need to be executed: the certificate has to be validated, the delegated credential has to be validated and the handshake has to be validated.

4 Measurement Experiment and Discussion

The goal of our experiment is to analyze the effects on the TLS handshake when using post-quantum algorithms. To do that, we measure the time it takes for a TLS 1.3 handshake using certificate-based authentication to complete, and compare all experiments to this standard measure.

4.1 Experiment Setup

We build a Drand cluster with one leader node and three worker peers. Each node independently ran in a data center located in Portland, USA. The connection of each internal node and the external `HTTPS` interface are configured to support post-quantum handshake protocols.

A Drand client retrieves randomness from the Drand network. We opted for locating the client far from the Drand network itself, so it is located in Lisbon, Portugal. With this setup our experiment faces the same traffic conditions found in transatlantic connections. Source codes of the client program are available at https://github.com/claucece/KEMTLS-local-measurements.

We choose a combination of cryptographic algorithms for setting up the following handshake configurations:

TLS 1.3 handshake using Ed25519 certificates for authentication (baseline).

TLS 1.3+DC handshake with Ed25519 certificate and delegated credentials either using Ed25519 or Ed448 algorithms for authentication.

PQTLS handshake with SIKEp434 and Kyber512 for key exchange, and hybrid signatures using round-two Dilithium mode 3 and mode 4, respectively, paired with Ed25519 and Ed448 for authentication (the authentication algorithms are advertised in DCs).

KEMTLS handshake with SIKEp434 and Kyber512 for both key exchange and authentication (the authentication algorithms are advertised in DCs).

KEMTLS-PDK handshake using the same configuration as KEMTLS (server authentication only).

4.2 Measurements

For each client to server connection, we measured the time elapsed until completion of the TLS handshake, that is until the client can send encrypted application data, for each different handshake configuration. We also measured the elapsed time for each flight of the handshake, i.e., the time elapsed that a peer (server or client) waits for receiving messages from their counterpart. We initiated two timers: one for the client (which started when the `CH` message was constructed and sent) and one for the server (which started when the `CH` message is received). Therefore, the first and second flight, as seen in the tables, do not include network latency, as the timer is started prior to the message being sent or just when it is received, respectively. Note that the round trip times (RTT) from the third flight onward are affected by the conditions of the state of the network. We tested the scenarios over an average-latency network.

To reduce the effects caused by the state of the network, the Drand client was instructed to fetch randomness from the Drand server consecutively during one hour. The total number of connections during this period amounts to approximately 5×10^3 connections. From them, we calculated the average time of the connections and report the timings in Table 1 and Table 2. We also measured the total average time until the handshake is completed (note that these times include the sending and receiving of encrypted application data). These measures are listed in Table 3 and Table 4.

In server-only authentication, the handshake performs the following flights:

1st $(C \Rightarrow S)$ Sending `ClientHello` for all cases.
KEMTLS-PDK: this message includes the `ClientKEMCiphertext` message, and a hash of the cached server's `ServerCertificate` message.
2nd $(C \Leftarrow S)$ Processing of `ClientHello`.
Standard and PQTLS: reply with the `ServerHello`, `ServerCertificate`, `ServerCertificateVerify` and `ServerFinished` messages.
KEMTLS: reply with the `ServerHello` and `ServerCertificate`.
KEMTLS-PDK: reply with the `ServerHello` and `ServerFinished` messages.
3rd $(C \Rightarrow S)$ Processing of received messages based on the protocol.
Standard and PQTLS: processing of `ServerHello`, `ServerCertificate`, `ServerCertificateVerify` and `ServerFinished` messages.
Reply with `ClientFinished` and immediate sending of encrypted application data.
KEMTLS: processing of `ServerHello` and `ServerCertificate`. Reply with `ClientKEMCiphertext` and `ClientFinished` messages and immediate sending of encrypted application data.
KEMTLS-PDK: processing of `ServerHello` and `ServerFinished` messages. Reply with `ClientFinished` and immediate sending of encrypted application data.
4th $(C \Leftarrow S)$ Processing of received messages based on the protocol.
Standard and PQTLS: processing of `ClientFinished` message and of encrypted application data.

KEMTLS: processing of `ClientKEMCiphertext` and `ClientFinished` messages. Reply with `ServerFinished` message.
KEMTLS-PDK: processing of `ClientFinished` message and of encrypted application data.

In mutual authentication, the handshake performs the following flights:

1st ($C \Rightarrow S$) Sending `ClientHello` for all cases.
2nd ($C \Leftarrow S$) Processing of `ClientHello`.
Standard and PQTLS: reply with the `ServerHello`, `ServerCertificate`, `ServerCertificateVerify`, `CertificateRequest` messages followed by the `ServerFinished` message.
KEMTLS: reply with the `ServerHello`, the `ServerCertificate` and the `CertificateRequest` messages.
3rd ($C \Rightarrow S$) Processing of received messages based on the protocol.
Standard and PQTLS: processing of `ServerHello`, `ServerCertificate`, `ServerCertificateVerify`, `CertificateRequest` messages followed by the `ServerFinished` message.
Reply with the `ClientCertificate`, the `ClientCertificateVerify` and the `ClientFinished` messages, and immediate sending of encrypted application data.
KEMTLS: processing of the `ServerHello`, the `ServerCertificate` and the `CertificateRequest` messages.
Reply with `ClientKEMCiphertext` and `ClientCertificate` messages.
4th ($C \Leftarrow S$) Processing of received messages based on the protocol.
Standard and PQTLS: processing of the received `ClientCertificate`, `ClientCertificateVerify` and `ClientFinished` messages, and received encrypted application data.
KEMTLS: processing of `ClientKEMCiphertext` and `ClientCertificate` messages. Reply with `ServerKEMCiphertext` message.
5th ($C \Rightarrow S$) This case only happens in KEMTLS. It includes the processing of `ServerKEMCiphertext` message and sending of the `ClientFinished` message. Immediate sending of encrypted application data.
($C \Leftarrow S$) This case only happens in KEMTLS. It includes the processing of `ClientFinished` message and any application data. Sending of the `ServerFinished` message.

4.3 Discussion

As noted, we initiated two timers for our measurements: one for the client (which started when the `CH` message was constructed and sent) and one for the server (which started when the `CH` message is received). This is the reason why the first and second flights see small timings as they do not take into account network latency. Starting from the third flight, the impact of network latency can be seen. An important point to note as well is that encrypted application data is sent already on the 3rd flight of all experiments except for KEMTLS for mutual

Table 1. Average time in 10^{-3} s of messages for server-only authentication. Note that timings are measured per-client and per-server: each one has its own timer. The 'KEX' label refers to the Key Exchange and the 'Auth' label refers to authentication.

Handshake	KEX	Auth	Handshake flight			
			1st	2nd	3rd	4th
TLS 1.3	X25519	Ed25519	0.227	0.436	123.838	180.202
TLS 1.3+DC	X25519	Ed25519	0.243	0.489	156.954	186.868
TLS 1.3+DC	X25519	Ed448	0.242	0.907	165.395	183.124
PQTLS	Kyber512	Dilithium3	0.350	0.701	173.814	198.256
PQTLS	SIKEp434	Dilithium4	2.533	4.856	441.732	212.924
KEMTLS	Kyber512	Kyber512	0.412	0.217	157.123	187.147
KEMTLS	SIKEp434	SIKEp434	3.058	7.215	352.840	291.592
KEMTLS-PDK	Kyber512	Kyber512	0.623	0.327	181.132	189.442
KEMTLS-PDK	SIKEp434	SIKEp434	9.573	12.507	396.818	287.550

authentication (as the client has to wait two flights in order to be able to send application data), which can increase the timing numbers.

When adding delegated credentials to the TLS 1.3 handshake, a peer receiving a delegated credential must validate that it was signed by the appropriate end-entity certificate (which is sent as part of the handshake) and must validate the certificate chain, as well. In our measurements, we observed a short increase in the latency of the flights when DCs are added; but the impact is almost negligible (specially, in the second flight when the DCs are received).

This is not the case when adding either post-quantum signatures or post-quantum KEMs for certain algorithms. The first observable difference appears in the `ClientHello` in both server-only authentication and mutual authentication: this message advertises both classic and post-quantum key exchange algorithms because this could be the realistic scenario for systems when transitioning to post-quantum cryptography. The timings increase specially when using SIKEp434 as a KEM in both KEMTLS and PQTLS, because its KEM decapsulation time takes in average 8.92 ms (when using the implementation of the CIRCL library). The predominant factor that slows down PQTLS is the number of signature validations; but this is similar (when using Kyber512) to using Ed448.

In regards to KEMTLS, its biggest drawback is the number of round-trips that it has to perform, specially when performing mutual authentication. The KEM cryptographic operations do not seem to heavily impact the connection if the underlying algorithm operations are fast. An ideal scenario for post-quantum cryptography is the use of KEMs for both confidentiality and authentication provided that the number of round trips do not increase, which is the case of KEMTLS-PDK for server authentication. This prediction matches with the

Table 2. Average time in 10^{-3} s of messages for mutual authentication. Note that timings are measured per-client and per-server: each one has its own timer. The 'KEX' label refers to the Key Exchange and the 'Auth' label refers to authentication.

Handshake	KEX	Auth	Handshake flight					
			1st	2nd	3rd	4th	5th	6th
TLS 1.3	X25519	Ed25519	0.113	0.420	111.358	121.349		
TLS 1.3+DC	X25519	Ed25519	0.148	0.546	129.638	178.90		
TLS 1.3+DC	X25519	Ed448	0.154	0.221	137.131	192.283		
PQTLS	Kyber512	Dilithium3	0.125	1.326	231.232	191.187		
PQTLS	SIKEp434	Dilithium4	3.324	7.294	459.888	216.077		
KEMTLS	Kyber512	Kyber512	0.244	0.303	231.752	175.490	375.202	346.308
KEMTLS	SIKEp434	SIKEp434	2.450	6.206	431.445	228.414	510.591	436.301

timings in tables: note that the best scenario is KEMTLS-PDK for server-only authentication, specifically, when it is used with Kyber512.

Let's look now at the measurements in regards to the kind of peer authentication they perform:

In the case of server-only authentication, KEMTLS performs faster than PQTLS and, in both cases, a client can immediately send application data on the third flight (when the client sends its ClientFinished). Nevertheless, for KEMTLS the server still has to wait for the ClientFinished to arrive and to send their ServerFinished in turn, in order to be able to send application data. Sending of the ServerFinished completes the handshake for the server, and provides full downgrade-resilience and forward-secrecy for the whole connection. However, this extra half-round trip forces the server to wait for a time before sending application data, which could not be an ideal scenario for real-world systems. In contrast, the client can send application data after sending their ClientFinished (as noted in the measurements) but it has weaker security protections (weak downgrade-resilience and forward-secrecy), and, therefore, a client might also wait until receiving the ServerFinished message to send its data in turn. This adds an extra round-trip which is not noted in the measurements. If we look at Fig. 1, we see that the best protocol to use is KEMTLS, if we don't take into consideration that application data sent at that point has weaker security properties. The ideal case is using KEMTLS-PDK which allows the sending of application data much earlier and with the stronger notions of the security properties.

For mutual authentication, KEMTLS has the biggest impact on the handshake completion timings, as an extra flight is needed prior to be able to send encrypted application data, as seen in Fig. 1. SIKEp434, on average, increases the handshake timings by approximate 10ms compared with Kyber512 for the verification of the peer's Certificate in both cases. For this reason, the PQTLS completion time is also slowed down when using SIKEp434 even without the extra round-trip addition. Although, we do not provide timings for the KEMTLS-

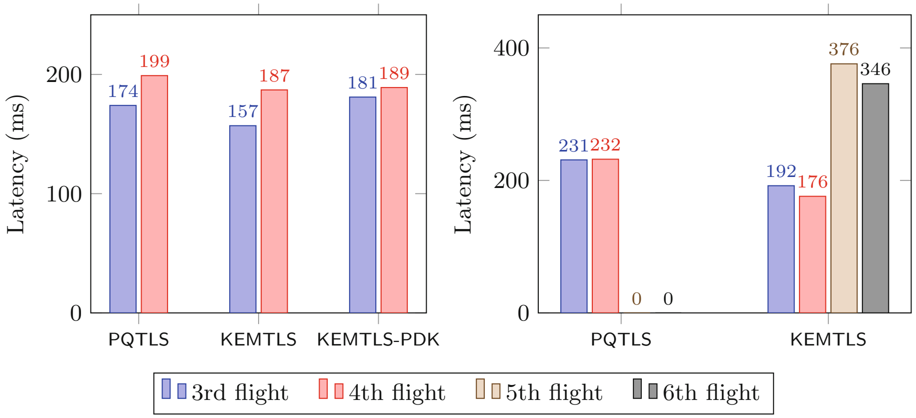

Fig. 1. Comparison of: on the left, server authentication flows for the 3rd, and 4th flights; on the right, mutual authentication flows for the 5th and 6th flights. Both using Kyber512.

Table 3. Total average handshake completion time (in 10^{-3} s) for server-only authentication.

Handshake	Key Exchange	Authentication	Handshake time	
			Server	Client
TLS 1.3	X25519	Ed25519	187.296	552.518
TLS 1.3+DC	X25519	Ed25519	197.568	578.097
TLS 1.3+DC	X25519	Ed448	220.576	614.366
PQTLS	Kyber512	Dilithium3	199.025	556.203
PQTLS	SIKEp434	Dilithium4	219.401	634.546
KEMTLS	Kyber512	Kyber512	200.237	792.168
KEMTLS	SIKEp434	SIKEp434	277.304	901.292
KEMTLS-PDK	Kyber512	Kyber512	209.872	583.582
KEMTLS-PDK	SIKEp434	SIKEp434	200.126	561.068

PDK handshake with mutual authentication, our timings can provide an insight about the cost of the operations and the relevance of the algorithm selection.

4.4 Optimizations

The cost of transmitting post-quantum parameters is tangible in our measurements. These costs can be further optimized by using a form of certificate compression [12] or of suppression of the intermediate certificates [41]. Still, the costs of post-quantum operations needed remains.

Table 4. Total average handshake completion time (in 10^{-3} s) for mutual authentication.

Handshake	Key exchange	Authentication	Handshake time	
			Server	Client
TLS 1.3	X25519	Ed25519	190.587	592.801
TLS 1.3+DC	X25519	Ed25519	179.653	549.760
TLS 1.3+DC	X25519	Ed448	222.902	541.695
PQTLS	Kyber512	Dilithium3	191.939	542.599
PQTLS	SIKEp434	Dilithium4	223.470	609.646
KEMTLS	Kyber512	Kyber512	352.448	881.928
KEMTLS	SIKEp434	SIKEp434	571.057	1096.708

5 Conclusions

Our experimental results are the first ones that integrate different post-quantum handshake alternatives to the TLS 1.3 handshake into a real-world system. These results have shown us how post-quantum algorithms can impact the handshake completion time, and, therefore, impact the establishment of real-world connections. In general, on the reliable network that we used, the different post-quantum TLS 1.3 handshake alternatives do not have a handshake completion time that is ostensibly different to a regular TLS 1.3 handshake. The only somewhat exception to this is KEMTLS, as the extra half or full round trip that is added does increase the completion time. For this reason, it is vital to think more in depth around KEMTLS-PDK, as it could reduce the completion time.

In this paper, we dive into the implementation of post-quantum algorithms in native Go language, adapt different handshake configurations and modify TLS extensions, and we explore the deployment of a test bed distributed network for enabling measurements. As a result, we developed a measurement framework that allows to perform transatlantic post-quantum TLS 1.3 connections for retrieving random numbers from a Drand network.

We remark that an important piece to achieve crypto-agility on the transition to post-quantum algorithms is the use of delegated credentials. They allowed us to advertise post-quantum KEMs or post-quantum signatures without generating new certificates or asking certificate authorities to support new algorithms.

Future work can involve increasing the number of connections tested, modifying the latency of the network, and testing with more post-quantum algorithms; we intend to continue our experiments. We further can extend our experiments to implement KEMTLS-PDK with mutual authentication, but more investigation is needed to determine the security requirements for encrypting the `ClientCertificate` message. Another interesting topic for further investigation lies around on how to properly integrate post-quantum algorithms into certificate chains and experiment with certificate authorities.

Acknowledgements. Authors wish to thank Latincrypt's reviewers for their useful suggestions. A special mention to Jonathan Hoyland for reviewing an early version of this document. Thom Wiggers was supported during this work by the European Commission through the ERC Starting Grant 805031 (EPOQUE). Goutam Tamvada was supported by the Natural Sciences and Engineering Research Council of Canada (NSERC) Discovery grant RGPIN-2016-05146.

References

1. Adrian, D., et al.: Imperfect forward secrecy: how Diffie-Hellman fails in practice. In: Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, CCS 2015, pp. 5–17. Association for Computing Machinery, New York (2015). https://doi.org/10.1145/2810103.2813707
2. Arai, K., Matsuo, S.: Formal verification of TLS 1.3 full handshake protocol using proverif (Draft-11). IETF TLS mailing list (2016). https://mailarchive.ietf.org/arch/msg/tls/NXGYUUXCD2b9WwBRWbvrccjjdyI
3. Aviram, N., et al.: DROWN: breaking TLS using SSLv2. In: 25th USENIX Security Symposium (USENIX Security 2016), pp. 689–706. USENIX Association, Austin, August 2016. https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/aviram
4. Barnes, R., Iyengar, S., Sullivan, N., Rescorla, E.: Delegated credentials for TLS. Internet-Draft draft-ietf-tls-subcerts-10, Internet Engineering Task Force, January 2021. https://datatracker.ietf.org/doc/html/draft-ietf-tls-subcerts-10. Work in Progress
5. Bernstein, D.J.: Curve25519: new Diffie-Hellman speed records. In: Yung, M., Dodis, Y., Kiayias, A., Malkin, T. (eds.) PKC 2006. LNCS, vol. 3958, pp. 207–228. Springer, Heidelberg (2006). https://doi.org/10.1007/11745853_14
6. Beurdouche, B., et al.: A messy state of the union: taming the composite state machines of TLS. In: 2015 IEEE Symposium on Security and Privacy, pp. 535–552 (2015). https://doi.org/10.1109/SP.2015.39
7. Braithwaite, M.: Experimenting with post-quantum cryptography. Google Security Blog, Google Online Security, July 2016. https://security.googleblog.com/2016/07/experimenting-with-post-quantum.html. Accessed 16 Feb 2021
8. Campagna, M., Crockett, E.: Hybrid post-quantum key encapsulation methods (PQ KEM) for transport layer security 1.2 (TLS). Internet-Draft draft-campagna-tls-bike-sike-hybrid-06, Internet Engineering Task Force, March 2021. https://datatracker.ietf.org/doc/html/draft-campagna-tls-bike-sike-hybrid-06. Work in Progress
9. Crockett, E., Paquin, C., Stebila, D.: Prototyping post-quantum and hybrid key exchange and authentication in TLS and SSH. In: Second PQC Standardization Conference, University of California, Santa Barbara, August 2019. https://csrc.nist.gov/Presentations/2019/prototyping-post-quantum-and-hybrid-key-exchange
10. Faz-Hernández, A., Kwiatkowski, K.: Introducing CIRCL: An Advanced Cryptographic Library. Cloudflare, Inc, June 2019. https://blog.cloudflare.com/introducing-circl/. Accessed Feb 2021
11. Feman, R.C., Willis, T.: Securing the web, together. Google Security Blog, March 2016. https://security.googleblog.com/2016/03/securing-web-together_15.html. Accessed 16 May 2021

12. Ghedini, A., Vasiliev, V.: TLS Certificate Compression. RFC 7924, RFC Editor, December 2020. https://doi.org/10.17487/RFC8879
13. Hoyland, J., Wood, C.: TLS 1.3 extended key schedule. Internet-Draft draft-jhoyla-tls-extended-key-schedule-03, Internet Engineering Task Force, December 2020. https://datatracker.ietf.org/doc/html/draft-jhoyla-tls-extended-key-schedule-03. Work in Progress
14. Hülsing, A., Rijneveld, J., Schanck, J., Schwabe, P.: High-speed key encapsulation from NTRU. In: Fischer, W., Homma, N. (eds.) CHES 2017. LNCS, vol. 10529, pp. 232–252. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-66787-4_12
15. Iyengar, J., Thomson, M.: QUIC: A UDP-Based Multiplexed and Secure Transport. RFC 9000, May 2021. https://doi.org/10.17487/RFC9000
16. Jao, D., et al.: SIKE. Technical report, National Institute of Standards and Technology (2020). https://csrc.nist.gov/projects/post-quantum-cryptography/round-3-submissions
17. Josefsson, S.: Storing Certificates in the Domain Name System (DNS). RFC 4398, RFC Editor, March 2006. https://doi.org/10.17487/RFC4398
18. Kampanakis, P., Sikeridis, D.: Two post-quantum signature use-cases: non-issues, challenges and potential solutions. In: 7th ETSI/IQC Quantum Safe Cryptography Workshop 2019, November 2019. https://eprint.iacr.org/2019/1276
19. Kiefer, F., Kwiatkowski, K.: Hybrid ECDHE-SIDH Key Exchange for TLS. Internet-Draft draft-kiefer-tls-ecdhe-sidh-00, Internet Engineering Task Force, May 2019. https://datatracker.ietf.org/doc/html/draft-kiefer-tls-ecdhe-sidh-00. Work in Progress
20. Kumar, D., et al.: Security challenges in an increasingly tangled web. In: Barrett, R., Cummings, R., Agichtein, E., Gabrilovich, E. (eds.) Proceedings of the 26th International Conference on World Wide Web, WWW 2017, Perth, Australia, 3–7 April 2017, pp. 677–684. ACM (2017). https://doi.org/10.1145/3038912.3052686
21. Kwiatkowski, K., Langley, A., Sullivan, N., Levin, D., Mislove, A., Valenta, L.: Measuring TLS key exchange with post-quantum KEM. University of California, Santa Barbara, August 2019. https://csrc.nist.gov/Presentations/2019/measuring-tls-key-exchange-with-post-quantum-kem
22. Lamik, M.: Introducing Cloudflare Radar. The Cloudflare Blog, September 2020. https://blog.cloudflare.com/introducing-cloudflare-radar. Accessed 16 May 2021
23. Langley, A.: CECPQ2. ImperialViolet, December 2018. https://www.imperialviolet.org/2018/12/12/cecpq2.html. Accessed 16 Feb 2021
24. Langley, A.: Real-world measurements of structured-lattices and supersingular isogenies in TLS. ImperialViolet, October 2019. https://www.imperialviolet.org/2019/10/30/pqsivssl.html. Accessed 16 Feb 2021
25. Lyubashevsky, V., et al.: CRYSTALS-DILITHIUM. Technical report, National Institute of Standards and Technology (2020). https://csrc.nist.gov/projects/post-quantum-cryptography/round-3-submissions
26. Marculescu, M.: Introducing gRPC, a new open source HTTP/2 RPC framework. Google Developers, February 2015. https://developers.googleblog.com/2015/02/introducing-grpc-new-open-source-http2.html
27. National Institute of Standards and Technology: Post-Quantum Cryptography Standardization, January 2017. https://csrc.nist.gov/projects/post-quantum-cryptography/post-quantum-cryptography-standardization. Accessed 16 May 2021
28. Prest, T., et al.: FALCON. Technical report, National Institute of Standards and Technology (2020). https://csrc.nist.gov/projects/post-quantum-cryptography/round-3-submissions

29. Rescorla, E.: The Transport Layer Security TLS Protocol Version 1.3. RFC 8446, RFC Editor, August 2018. https://doi.org/10.17487/RFC8446
30. Santesso, S., Tschofenig, H.: Transport Layer Security (TLS) Cached Information Extension. RFC 7924, RFC Editor, July 2016. https://doi.org/10.17487/RFC7924
31. Schanck, J.M., Stebila, D.: A Transport Layer Security (TLS) Extension For Establishing An Additional Shared Secret. Internet-Draft draft-schanck-tls-additional-keyshare-00, Internet Engineering Task Force, April 2017. https://datatracker.ietf.org/doc/html/draft-schanck-tls-additional-keyshare-00. Work in Progress
32. Schanck, J.M., Whyte, W., Zhang, Z.: Quantum-Safe Hybrid (QSH) Ciphersuite for Transport Layer Security (TLS) version 1.2. Internet-Draft draft-whyte-qsh-tls12-02, Internet Engineering Task Force, January 2017. https://datatracker.ietf.org/doc/html/draft-whyte-qsh-tls12-02. Work in Progress
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: 27th Conference on Computer and Communications Security, pp. 1461–1480. ACM Press, Virtual Event, 9–13 November 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 (2021). https://eprint.iacr.org/2021/779
35. Sikeridis, D., Kampanakis, P., Devetsikiotis, M.: Post-quantum authentication in TLS 1.3: a performance study. In: ISOC Network and Distributed System Security Symposium - NDSS 2020. The Internet Society, San Diego, 23–26 February 2020
36. Stebila, D., Mosca, M.: Post-quantum Key exchange for the internet and the open quantum safe project. In: Avanzi, R., Heys, H. (eds.) SAC 2016. LNCS, vol. 10532, pp. 14–37. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-69453-5_2
37. Steblia, D., Fluhrer, S., Gueron, S.: Hybrid key exchange in TLS 1.3. Internet-Draft draft-ietf-tls-hybrid-design-03, Internet Engineering Task Force, April 2021. https://datatracker.ietf.org/doc/html/draft-ietf-tls-hybrid-design-03. Work in Progress
38. Sullivan, N.: Why TLS 1.3 isn't in browsers yet. The Cloudflare Blog, December 2017. https://blog.cloudflare.com/why-tls-1-3-isnt-in-browsers-yet/. Accessed 15 April 2021
39. Sullivan, N.: A detailed look at RFC 8446 (a.k.a. TLS 1.3). The Cloudflare Blog, August 2018. https://blog.cloudflare.com/rfc-8446-aka-tls-1-3/. Accessed 16 February 2021
40. Syta, E., et al.: Scalable bias-resistant distributed randomness. In: 2017 IEEE Symposium on Security and Privacy (SP), pp. 444–460 (2017). https://doi.org/10.1109/SP.2017.45. https://drand.love
41. Thomson, M.: Suppressing intermediate certificates in TLS. Internet-Draft draft-thomson-tls-sic-00, Internet Engineering Task Force, March 2019. https://datatracker.ietf.org/doc/html/draft-thomson-tls-sic-00. Work in Progress
42. Whyte, W., Zhang, Z., Fluhrer, S., Garcia-Morchon, O.: Quantum-Safe Hybrid (QSH) Key Exchange for Transport Layer Security (TLS) version 1.3. Internet-Draft draft-whyte-qsh-tls13-06, Internet Engineering Task Force, October 2017. https://datatracker.ietf.org/doc/html/draft-whyte-qsh-tls13-06. Work in Progress