Quantum Key Distribution as a Service and Its Injection into TLS

Sergejs Kozlovičs(✉), Krišjānis Petručeņa, Dāvis Lāriņš, and Juris Vīksna

Institute of Mathematics and Computer Science, University of Latvia, Riga, Latvia
{sergejs.kozlovics,krisjanis.petrucena,davis.larins,juris.viksna}@lumii.lv

Abstract. Quantum key distribution (QKD) is a key agreement method that relies on the laws of physics and ensures that the keys have not been eavesdropped on or modified by a third party. While commercial QKD devices are available, they are expensive, require specific infrastructure, and have high operational expenses. In this paper, we propose an architecture and a set of protocols that allow us to implement QKD as a service (QaaS). End users communicate with QaaS via classical TLS channels secured with post-quantum cryptography (PQC). We show how to further strengthen the security of these classical links to make them sustainable to active attacks (classical and quantum) on any single segment of QaaS. We also show how to integrate QaaS into the state-of-the-art TLS 1.3 protocol. As a result, QKD becomes available for a larger community of end-users. Furthermore, we show how QaaS can reduce the number of digital signatures within a TLS 1.3 handshake, which is essential since post-quantum signatures are much longer than the conventional RSA/ECC-based ones.

Keywords: quantum key distribution · post-quantum cryptography · transport layer security · PQC · QKD · TLS · QaaS

1 Introduction

Quantum key distribution (QKD) is the first step on our way to a universal Quantum Internet. QKD is a state-of-the-art technology that allows two distant parties to agree on encryption keys. The key distribution process involves a quantum channel (usually implemented via optical fiber transmitting photons), but the agreed keys are intended to be used in classical internet communication.

The properties of quantum mechanics (in particular, the no-cloning theorem) along with quantum key distribution protocols such as BB84 (and its successors B92, SARG04, Lo05) and COW[1] ensure that if some key has been eavesdropped on or altered, the parties can notice that and discard the key [2,6].

Since it is difficult to emit single photons and to deal with attenuation over long distances, certain attacks (such as photon number splitting, PNS, and other

[1] Coherent One-Way protocol, patented by IDQ.

W. Meng et al. (Eds.): ISPEC 2023, LNCS 14341, pp. 527–545, 2023.
https://doi.org/10.1007/978-981-99-7032-2_31

side-channel attacks against physical QKD implementations) are theoretically possible [17]. However, these attacks are either hard to exploit on short distances or can be impeded by modified versions of algorithms such as BB84 Decoy State [16]. Thus, for practical purposes, we can assume a short- to midterm security of modern QKD technology with the hope for near-to-perfect security of future QKD devices.

Sadly, the state-of-the-art commercial QKD devices are expensive, require specific infrastructure (high-quality optical fiber links), and have high operational expenses (such as energy costs for cooling down the devices) [13]. In order to make QKD available to a wider community of users, we deliver QKD as a service (QaaS). With QaaS, end users are able to securely obtain a shared secret from two remote key distribution centers (KDCs), where each KDC is directly connected to the corresponding endpoint of the QKD link. KDCs may be located in two cities with an established quantum channel between them. With QaaS, the inhabitants of both cities have the ability to obtain quantumly distributed keys without the need for a direct connection to QKD equipment. QaaS is a technology for connecting end users to existing QKD networks that are now being deployed all over the world [19,22].

The end users connect to KDCs via classical TLS channels, which, from the QKD point of view, are the weakest links in the key distribution process. In order to strengthen the security of classical links used in QaaS, we use post-quantum cryptography (PQC). However, PQC algorithms still have to withstand the test of time,—new attacks are constantly emerging, and the NIST standardization process is not yet finished [1,3,5,7,8]. Thus, we strengthen the security even further by proposing the architecture and a set of protocols that make QaaS sustainable to active attacks (classical and quantum) on any single communication segment. In particular, the full key is not sent via any single classical channel. Thus, a successful man-in-the middle attack would require compromising two independent TLS communication links.

The QaaS architecture and a set of underlying protocols are described in Sects. 2 and 3. In Sect. 4, we offer QaaS-specific authentication options for all involved parties. This is a noticeable contribution, since pure QKD does not offer any authentication mechanism. In Sect. 5, we give insight into some implementation detail and show how to integrate the proposed QaaS into the state-of-the-art TLS 1.3 protocol. We also show how QaaS can be used to reduce the number of digital signatures within a TLS 1.3 handshake, which is essential since post-quantum signatures are much longer than the conventional RSA/ECC-based ones. We conclude by discussing the related work and further research directions (Sects. 6 and 7).

2 The Overall QaaS Architecture

Figure 1 depicts the overall architecture of the proposed QaaS.

At the bottom of Fig. 1, two QKD devices (called Alice and Bob) are connected by multiple links implementing the quantum channel and the service

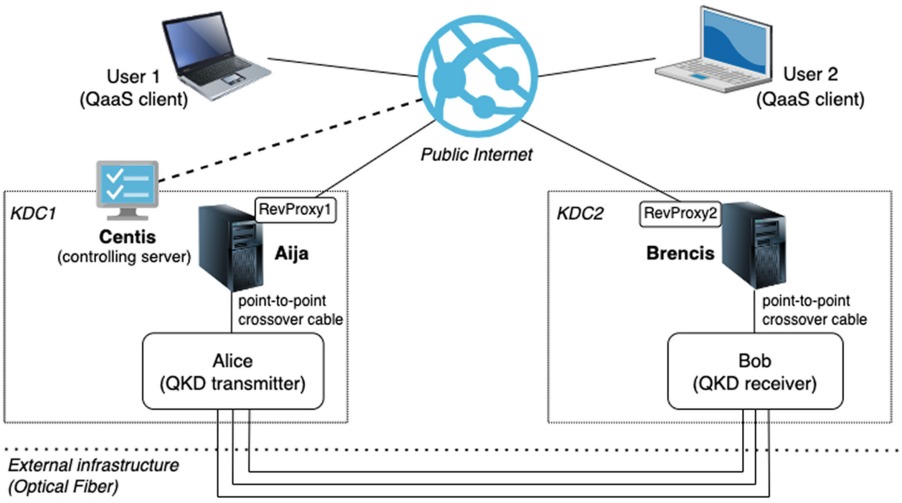

Fig. 1. The architecture of the proposed QaaS.

channel (both channels are needed in most QKD protocols). Depending on the protocol and hardware choice, there can be 2–3 optical links or a mix of a direct optical link and a classical (routed) internet connection [11].

There are multiple QKD devices available in the market.[2] While we conduct our experiments with IDQ Clavis[3] devices, our architecture can be applied to other devices as soon as they meet the following assumptions:

- Alice and Bob are pre-paired at the factory (e.g., with several one-time symmetric keys) and are able to establish a secure service channel as well as the quantum channel (the process of synchronizing the quantum channel usually takes several minutes).
- Once the secure channels are established, both Alice and Bob are able to generate two potentially infinite[3] identical streams of symmetric keys. Some of the shared keys can be used by Alice and Bob for technical purposes, e.g., to replace the pre-paired factory keys for subsequent re-initializations.
- Each key is of the same bit length (256 bit for IDQ Clavis[3]) and has an associated unique truly random key ID, which is near to impossible to guess (IDQ Clavis (See footnote 3) uses 128-bit key IDs; keys and IDs are generated using the built-in QRNG[4] chip).

[2] e.g., Toshiba Multiplexed and Long Distance, IDQ Clavis and Cerberis series, QTI Quell-X, LuxQuanta NOVA LQ, KEEQUANT Andariel, SeQre Aurora and Eclipse.

[3] unless the link is physically broken, a hardware failure occurs, or there is constant eavesdropping or intrusion.

[4] quantum random number generator.

From the architectural point of view, Alice and Bob are black boxes that simultaneously produce synchronized key-identifier pairs (K_{id}, id) that are secure against man-in-the-middle attacks.

We place Alice and Bob at two physically distant key distribution centers (KDCs). Each KDC also has a physical server that is directly attached (e.g., by a short crossover cable) to the corresponding QKD device. We call these servers Aija and Brencis (in order to distinguish them from the QKD devices, Alice and Bob); we also call them **KDC endpoints**. Both Aija and Brencis run QaaS server software that takes the stream of quantumly exchanged keys from Alice and Bob, respectively, and implements the QaaS protocols (discussed in Sect. 3), which securely forward the keys to end users, User 1 and User 2. In order to simplify the QaaS server software and strengthen the security of Aija and Brencis, we introduce two reverse proxies, RevProxy1 and RevProxy2. The reverse proxies authenticate end users and ensure encrypted TLS connections with them via the public internet. For such TLS connections, we utilize quantum-safe key exchange methods and signature algorithms.

All backend connections within the boundaries of a KDC (e.g., RevProxy1 $\leftrightarrow$ Aija $\leftrightarrow$ Alice) are not encrypted; however, we assume that the corresponding physical links are isolated from the external world, and no wiretapping is possible within a KDC.

For technical reasons, we need also a controlling server (called Centis in Fig. 1) that synchronizes key reservations at Aija and Brencis. Centis can be an internal server (located at the premises of one of the KDCs) or an external (cloud) server. Centis needs specific user credentials to pass through RevProxy1 and RevProxy2.

3 QaaS Protocols

The purpose of QaaS is to ensure that end users (User 1 and User 2 in Fig. 1) obtain a shared key that has been quantumly exchanged between Alice and Bob. The main issue is that, in QaaS, we are able to use only classical (i.e., non-quantum) channels between end users and KDCs.

In this section, we introduce two protocols: the Butterfly Protocol and the Control Protocol. The former allows QaaS to tolerate active attacks on any single classical link (even if TLS is decrypted). The latter is used by the controlling server Centis in order to manage key reservations at Aija and Brencis.

3.1 The Butterfly Protocol

Figure 2 depicts message flows between the users and KDCs used in the Butterfly Protocol (hence, the name).

The protocol allows User 1 and User 2 to agree on a shared key and ensures that the full key is not transmitted via any of the classical links. During the protocol, one KDC endpoint (Aija) sends only the first half of the key, and the other (Brencis) sends the second half.

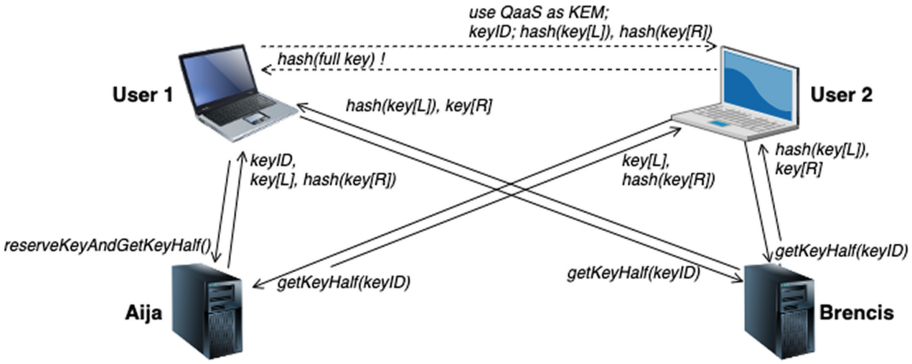

Fig. 2. Message flows in the Butterfly protocol.

There are two types of connections: **the butterfly connections** (straight lines in Fig. 2) and **the user connection** (dashed lines between User 1 and User 2).

All butterfly connections are implemented as bidirectional TLS sockets secured with PQC key exchange mechanisms (KEMs) and digital signature algorithms (for authentication). The butterfly connections require both server and client authentication (see Sect. 4).

The user connection is also implemented as a bidirectional socket. However, instead of a PQC KEM, we use the message flow of the Butterfly Protocol. Since no key material but hashes are sent via the user connection, KEM-like (Diffie-Hellman-like) encryption is unnecessary (although possible). Like in the traditional client-server architecture, client authentication is optional for the user connection. Server authentication can be performed either by a PQC signature algorithm or our novel approach described in Sect. 4.2.

The protocol starts when User 1 wants to communicate with User 2 via a TLS socket.

1→ Before initiating a TLS handshake with User 2, User 1 chooses one of the two KDCs (say, Aija) and sends the *reserveKeyAndGetKeyHalf* message to it. An alternative (symmetric) scenario, when User 1 chooses Brencis, is also possible. Thus, Aija and Brencis can participate equally in key reservations. We call it the **equivalence property** of KDCs.[5]

1← Aija chooses one of the quantumly shared keys key and replies with $keyID$, the first (left) half of the key $key[L]$, and the hash of the second (right) half of the key $hash(key[R])$.

2→ User 1 asks Brencis for the second half of the key by sending the *getKeyHalf(keyID)* message.

[5] One of the benefits of the equivalence property is that there is no advantage in attacking either of KDCs. Another benefit is the ability to design algorithms and protocols that can purposely choose the first receiver of the *reserveKeyAndGetKeyHalf* message.

2← Brencis replies with $hash(key[L])$ and the second half of the key $key[R]$. User 1 validates $key[L]$ (received from Aija) against $hash(key[L])$ received from Brencis. User 1 also validates $key[R]$ (received from Brencis) against $hash(key[R])$ received from Aija.

3→ User 1 initializes the TLS handshake of the user connection by sending the Client Hello message to User 2. In the handshake, User 1 sends $keyID$, $hash(key[L])$, and $hash(key[R])$ to User 2. (User 2 will use these hashes as proof that User 1 has been authenticated within KDC1).

4→ User 2 asks Aija for the first half of the key by sending the *getKeyHalf(keyID)* message.

5→ User 2 asks Brencis for the second half of the key by sending the *getKeyHalf(keyID)* message.

4← Aija replies with $key[L]$ and $hash(key[R])$.

5← Brencis replies with $hash(key[L])$ and $key[R]$. User 2 validates $key[L]$ against the two copies of $hash(key[L])$ (received from User 1 and from Brencis) and $key[R]$ against the two copies of $hash(key[R])$ (received from User 1 and from Brencis).

3← User 2 computes $hash(full\ key)$ and sends it in the Server Hello message to User 1. User 1 validates its $full\ key$ against this hash. Since $hash(full\ key)$ cannot be efficiently computed by User 2 without communicating to Aija and Brencis, it serves as proof for User 1 that User 2 has been authenticated within both KDC1 and KDC2.

Step 3 can be launched in parallel with step 2; steps 4 and 5 can be launched in parallel as well.

The strength of the protocol relies on the underlying hash function (e.g., SHA-256 or SHAKE-128, used in our experiments) and PQC algorithms used for the butterfly connections. Besides, the protocol is able to sustain an active attack on any single link. By an active attack we mean the ability to decrypt or wiretap the TLS session key.

- The quantum link is assumed to be secure against wiretapping by the law of Physics.
- If Eve can decrypt *one* of the butterfly links (say User 1 to Aija), she can get the key ID and the first half of the key. However, Eve would be unable to connect also to Brencis (without breaking another butterfly link) since a new connection to Brencis needs client authentication, and Eve lacks the private keys owed by Users 1 and 2.
- If Eve attacks the user link (between User 1 and 2), she cannot wiretap the session key since only key ID and hashes are transmitted there. If Eve alters the key ID that is being sent to User 2, Aija and Brencis won't reply to User 2 for a non-reserved key ID. In an unlikely case when the modified key ID has also been reserved (e.g., by other QaaS users), the hashes won't match.

3.2 User Connection Management

The user connection is a TLS 1.3 connection with the distinction that we inject the Butterfly Protocol as a new key share "group"[6] in the TLS Key Share extension (see more detail on page 14 in Sect. 5). If all butterfly connections finish successfully, and all Butterfly Protocol checks (e.g., hash validations) are passed, both User 1 and 2 get the keys. On any failure (TCP, TLS, or Butterfly Protocol error) within any of the butterfly connections between User1/User2 and Aija/Brencis, the TLS between User 1 and User 2 closes with an exception. That may happen due to security checks (e.g., Aija or Brencis could not authenticate the user, TLS error) or when Aija or Brencis is temporarily down (TCP error). Besides, Aija or Brencis can reply with an error when they have been (re-)launched but are not serving the keys yet (more on that in Sect. 3.4).

3.3 Key Reservation in the Butterfly Protocol

Since both Aija and Brencis can be used for key reservation (Step 1), they need some distributed algorithm that resolves conflicts between them. Besides, if some key has been reserved at one KDC, both KDCs must be ready to send their key halves to Users 1 and 2 and to delete the used key afterward (even if the protocol has started but not finished, e.g., due to a network interruption).

Our idea is to divide the keys into two classes depending on their parity ($keyID$ bit sum). Aija is allowed to reserve those keys for which parity is even, and Brencis—those with odd parity; thus, no collision is possible. However, in order to ensure the eventual consistency of key sets between Aija and Brencis, we also need the following time constants:

- ε is a small time interval ($\varepsilon < 1$ second) that must pass before a newly quantumly exchanged key can be reserved by end users. The need for ε arises from the fact that new keys do not appear in the QaaS software of Aija and Brencis simultaneously. Waiting for the time ε ensures that both KDC endpoints receive the key (thus, ε can be compared to the cycle time in CPUs);
- the key reservation timeout T ($T \approx 90$ seconds)[7]. If a key has been reserved for User 1, but no $getKeyHalf$ message has been received from User 2 during the time T, the key is deleted. At the other KDC endpoint, T is the maximal waiting time between the two $getKeyHalf$ messages expected from Users 1 and 2;
- TTL (time-to-live $\approx$ 1 day) is the maximal time the key is available for reservation. TTL limits the size of the key buffer and ensures that "zombie" keys are eventually deleted.[8]

[6] TLS 1.3 terminology; actually, it is a key exchange method.

[7] We choose T such that each of the three connections at the longest path (Aija $\leftrightarrow$ User 1 $\leftrightarrow$ User 2 $\leftrightarrow$ Brencis) can survive the maximal TCP back-off; $T \approx 3 \times 30$ s $\approx 3\times$ TCP re-transmission timeout for five tries.

[8] A key is called a "zombie" if it is being stored at one KDC endpoint but is not present at the other, i.e., it has been reserved and deleted or hasn't been received

Each KDC endpoint has three disjoint dictionaries that map $keyID$-s to keys: $PreMy$, My, and $NotMy$. New keys with the corresponding parity p are placed temporarily to $PreMy$ (for time ε) before they are moved to My. All the keys with parity $p-1$ are moved to $NotMy$ without any delay. Therefore, when Aija reserves a key, the time ε has already passed, and Brencis should contain the same key in its $NotMy$ dictionary (even if the key appeared at Brencis later than at Aija but within the allowed ε time slot).

Keys in the My dictionary are stored for the $TTL - \varepsilon$ time. Keys in the $NotMy$ dictionary are stored for the $TTL + T$ time. Thus, if a key is reserved at one KDC endpoint, it will still be available at the other endpoint while the Butterfly Protocol is running. On the other hand, the reserved keys are eventually deleted even if the protocol is aborted.

3.4 The Control Protocol

After the QKD initialization, Aija and Brencis start to receive identical streams of keys from their corresponding QKD devices (Alice and Bob). However, due to system reboots, hardware failure, or network interruptions, one or both KDC endpoints can stop receiving keys from Alice and/or Bob. While it is possible to re-initialize the QKD channels, the process takes a long time (e.g., up to 30 min in IDQ Clavis[3]), and QKD devices consume more power than during normal operation. The purpose of the Control Protocol (executed by the controlling server Centis in Fig. 1) is to ensure that Aija and Brencis have the same streams of QKD keys after they re-connect to Alice and Bob.

For Aija and Brencis, we define the following three states:

EMPTY—when there are no keys received from the QKD device yet;
RECEIVING—when at least one key has been received and put into the My map;
RUNNING—when keys can be reserved by the users. If at some point in time $My \cup PreMy$ becomes empty[9], the state is automatically changed to EMPTY.

When launched, both KDC endpoints are in the EMPTY state. They can change their state to RECEIVING or EMPTY, depending on the keystream from the QKD device. The RUNNING state can be set only by Centis, when both KDC endpoints are in the RECEIVING state.

Centis can send two types of messages to Aija and Brencis:

- **getState** is a no-argument message that asks for the current state; if the state is RECEIVING or RUNNING, the KDC endpoint also sends two key IDs in the reply: $keyID_0$ and $keyID_1$ corresponding to the first key IDs (the oldest) of each parity;

at all (due to server restart or network interruption). "Zombie" keys can also be deleted before TTL expires, e.g., by the Control Protocol.

[9] e.g., due to too many key reservation requests or due to some technical failure, when new keys stop appearing from the QKD device.

– **setState**($state, keyID_0, keyID_1$) instructs the KDC endpoint to change its state and remove the unnecessary keys depending on $keyID_0$ and $keyID_1$.

On a regular basis, Centis sends the *getState* message to both KDC endpoints. If one of them is in the EMPTY state or is not reachable, Centis sends the *setState*(EMPTY) to the other endpoint (thus, clearing the keys, if any). However, if both Aija and Brencis are in the RECEIVING state, Centis gets four key IDs (two from each endpoint): $keyID_{0,Aija}, keyID_{1,Aija}, keyID_{0,Brencis}$, and $keyID_{1,Brencis}$. Then Centis sends

– setState(RUNNING, $keyID_{0,Aija}$, $keyID_{1,Aija}$) to Brencis and
– setState(RUNNING, $keyID_{0,Brencis}$, $keyID_{1,Brencis}$) to Aija.

When receiving $keyID_{parity,opponent}$, the endpoint looks up for this key ID in the corresponding dictionary. We distinguish three cases:

– The key is not found. In this case, no keys are deleted because our endpoint has fewer keys than the opponent. Keys will be deleted at the opponent's side, where our first key will be found.
– The key is found, and the parity is ours (corresponding to the keys we can reserve). In this case, keys with the IDs received prior to the found one can be deleted. The opponent does not have them and will not be able to reply to $getKeyHalf$ messages for those key IDs.
– The key is found, and the parity is the opponent's parity. In this case, keys received prior to $currentTime - T$ can be deleted. We keep a few older keys (within the time frame T) since they could have been reserved and deleted at our opponent while the Butterfly protocol is still running (thus, Users 1 and 2 can send us $getKeyHalf$ requests for those keys).

In any case, the endpoint changes its state to RUNNING, meaning that it can start serving Butterfly protocol requests from users 1 and 2.

While communicating with Aija and Brencis, Centis uses TLS with PQC. Centis certificate is signed by CA_{Centis}, which differs from the CA that signs certificates for Users 1 and 2; thus, reverse proxies can authorize Centis to execute the Control Protocol if Centis possesses the corresponding private key. Since only four key IDs are sent in the Control Protocol, no single bit of the keys themselves is compromised. Though, an active attacker can use these key IDs to impede the Butterfly Protocol. In order to mitigate such attacks, the keys corresponding to the four key IDs are deleted and not distributed to end users.

4 Authentication

In the Butterfly and Control Protocols, User 1, User 2, and Centis act as clients, which initiate the corresponding connections. Obviously, the clients need to validate the authenticity of both KDC endpoints, which have control over all quantumly shared keys. Since the speed of generating new shared keys is limited, KDC endpoints (the servers) have to identify their clients in order to distribute

the keys between them. Client authentication is a must if QaaS is offered as a paid service, where different clients may have different payments (e.g., depending on a subscription plan or the number of keys shared).

In addition, depending on the application, User 1 and User 2 (acting as the client and the server in the user connection) may need to validate each other.

First, we show how both client and server authentication can be established by means of PQC signatures. Since PQC signatures are much longer than traditional RSA/ECC-based ones, we also show how to minimize the number of signatures used in the Butterfly and Control Protocols.

4.1 Authentication via PQC Signatures

At one extreme, there could be a single certification authority (CA) that signs the public keys of all involved parties. At another extreme, there could be a separate CA for each node from Fig. 1 with potential intermediate CA-s. A more realistic model, though, is having two trusted root certification authorities (CA-s), CA_1 and CA_2, which are parts of KDC1 and KDC2, respectively. In this model (which we stick to), KDC1 and KDC2 are located in different places (e.g., cities) and managed by different organizations (e.g., city authorities).

Each QaaS client (User 1 and User 2 in Fig. 1) applies for a client certificate either at CA_1 or at CA_2 (e.g., depending on the client's city of residence).[10]. The chosen CA verifies the client payment and issues a client certificate valid for the time period paid up. Each of RevProxy1 and RevProxy2 from Fig. 1 accepts client certificates signed by both CA_1 and CA_2. Besides, the reverse proxies identify themselves with server certificates signed by CA_1 and CA_2, respectively.

The client key pair (generated), the client certificate (signed after receiving payment), and both server certificates (public) are delivered to the client. We call these data client bundle.

Authentication of the controlling server (Centis) is performed similarly. However, its certificate is signed by a specific CA (CA_{Centis}), which is trusted by both RevProxy1 and RevProxy2. CA_{Centis} is used to sign public keys of controlling servers only.

The servers (the reverse proxies) need only CA_1, CA_2, and CA_{Centis} to be configured as trusted root CA-s. Since client authentication is performed by validating digital signatures, no client database is required. However, in a rare case when QaaS access has to be revoked from some client, the corresponding client is added to the server-side certificate revocation list (CLR), which is delivered to both RevProxy1 and RevProxy2.

Client and server authentication is performed via the normal TLS v1.3 flow, following the traditional signed key exchange approach. However, in our case, both the client and the server negotiate a post-quantum KEM and send certificates signed with a PQC algorithm (see Sect. 5). After the handshake, TLS continues as usual using a symmetric cipher suite (e.g., AES in GCM mode).

[10] A client can generate a key pair by himself and send a certificate signing request (CSR) to the CA, or the whole process can be performed by the CA.

While the user connection (between Users 1 and 2 in Fig. 2) can also use PQC certificates to authenticate the client and the server, the following section proposes a more elegant approach.

4.2 Reducing the Number of Post-Quantum Signatures

Eliminating Signatures in Client Certificates. Client certificates used in butterfly connections can be replaced by arbitrary tokens. In this case, TLS starts with server-only authentication, and the client sends its token in the encrypted application data. Sending the token *after* the TLS handshake prevents its eavesdropping.

In the naïve approach, the issued tokens are stored in a database, shared or replicated between both KDCs. In order to ensure database consistency, a classical connection is needed between KDCs (a pre-shared symmetric key can be used for it; KDCs may also update this key with quantumly exchanged keys on a regular basis).

A more advanced approach is to rely on tokens with hash-based signatures such as HMAC-SHA256-based JSON web tokens, JWTs[11]. Each KDC has a secret key used to sign the header and payload (e.g., the client name + expiration date + salt) of JWT tokens. Signed tokens are distributed to QaaS clients. Both KDCs must have secret keys of each other in order to verify tokens signed by either KDC. While not requiring a database, JWTs need a CRL alternative in order to revoke previously issued tokens.

Since JWT tokens support only non-PQC RSA and ECDA asymmetric signature schemes, we suggest using the symmetric HMAC algorithm (considered quantum-safe), where both KDC endpoints know the symmetric keys of each other.

Reducing the Number of Server Signatures to Be Transmitted. Public keys of both KDCs can be added to the client bundle (from Sect. 4.1). Thus, the server can send only its public key instead of the full certificate chain. However, albeit rarely, the client still has to download and verify the full chain after the server key is renewed.

Eliminating Server Signatures in the User Connection. As we explained in Sect. 3.1, the hash values for the full quantumly exchanged key and its halves can be used by User 1 and User 2 as proofs that the counterparty has been authenticated within one or both KDCs. We use this property to extend the Butterfly Protocol with the support for server authentication. We consider the idea of domain-based authentication, traditionally used in TLS certificates.[12]

When requesting a client certificate (or a JWT token), a QaaS client can specify its domain name. This scenario is useful for QaaS clients that will play

[11] https://jwt.io.

[12] Technically, any string, e.g., a URI, can be used to identify the communicating parties. In this paper, we use the term "domain name" to represent such strings.

the server role in the user connection (i.e., User 2 in Fig. 2). The signing KDC associates the issued certificate (or a token) with the client domain name (it has to be done only at one KDC, which we call a "domain registrar" for the given QaaS client).

For domain name validation, we introduce the following modifications to the Butterfly Protocol (called the Butterfly Protocol with Domain Validation):

- User 1 (from Fig. 2) appends the domain name of User 2 (application server) to each *reserveKeyAndGetKeyHalf* and *getKeyHalf* request.
- User 1 sends the *reserveKeyAndGetKeyHalf* and *getKeyHalf* messages to Aija and Brencis *before* the handshake with User 2.
- After receiving the domain name in a request sent by User 1, the domain registrar for User 2 associates the reserved key with the domain name and waits for a *getKeyHalf* request from User 2. The other KDC endpoint (which is not the registrar for User 2) just replies as usual and appends the domain check result value of *false*.
- User 2 sends the *getKeyHalf* requests to Aija and Brencis. Each endpoint checks whether the key has been associated with a domain name. If no, the reply to User 2 is sent as usual. If yes, the registrar checks that the domain name is indeed associated with User 2 and sends the requested half of the key to User 2 only if the check returned *true*.[13]. In any case, the result of this domain check is sent back to User 2. Thus, if the check fails, User 2 doesn't receive one half of the key and is not able to compute $hash(full\ key)$.
  After finishing processing the *getKeyHalf* request, the registrar (which was waiting for it) can now reply to User 1 with the check result.
- After receiving both key halves from Aija and Brencis, User 2 validates both hashes received from User 1 and computes *hash(full key)* to be sent to User 1.
- (Check 1) User 1 computes the OR function on both domain check results received from Aija and Brencis. The value of *true* corresponds to the case when the domain name of User 2 has been recognized by one of the KDC endpoints.
- (Check 2) User 1 also validates *hash(full key)* received from User 2 (see Fig. 2). The correct hash value means that User 2 received key halves from both KDCs; thus, User 1 (application client) can now trust that it is talking to User 2 (application server), having the corresponding domain name.

Notice that two checks associate the butterfly connections with the user connection: Check 1 validates that fact of domain registration, while Check 2 validates that User 2 possesses the full key (i.e., the two butterfly links between User 2 and KDCs have been executed).

[13] In the case of client certificates, the traditional certificate-based domain name validation is performed. In the case of JWT tokens, the check is performed by a database lookup or by verifying the hash-based JWT signature.

5 Implementation and Integration into TLS 1.3

The QaaS service software that runs on Aija and Brenics has been developed using the Go programming language, which has built-in concurrency support. The QaaS software implements the server-side part of both the Butterfly Protocol and the Control Protocol. In order to support QKD devices from different vendors, we created a Go interface named *KeyGatherer* that is used to obtain streams of indexed keys, i.e., tuples $(keyID, key, timestamp)$. Currently, we have three *KeyGatherer* implementations: one for the IDQ Clavis[3] device (used in our real testbed), another for fetching keys from the file system (e.g., when keys from the QKD device are stored as files; a shared folder can also be used to simulate a QKD device), and the third one for generating random keys on-the-fly inside a single process used to simulate both Aija and Brencis.

The Go code implements the Butterfly Protocol and the Control Protocol via non-TLS web sockets. Post-quantum key and certificate management and TLS implementation on the server side are provided by reverse proxies. We have implemented our own reverse proxy in Java by relying on the TLS implementation provided by the BouncyCastle library[14]. Alternatively, HAProxy based on OpenSSL 1.1.1 with embedded PQC algorithms from the OpenQuantumSafe project can be used [23].[15]

The QaaS client library (used by User 1, User 2, and Centis) has been implemented in Java using the BouncyCastle library. A pure-Java client implementation allows us to deploy the QaaS client library for Linux, macOS, and Windows by compiling it with GraalVM Native Image [25].[16] For the PQC butterfly connections, our Java implementation is interoperable with LibOQS (written in C); thus, we can use any LibOQS-based reverse proxy to provide PQC to backend endpoints.

Sadly, BouncyCastle, out of the box, does not support PQC algorithms in TLS. Thus, we implemented a set of additional classes that allow us to inject PQC KEMs and signature schemes into TLS 1.3 flow in the BouncyCastle code.[17] We call it **TLS Injection Mechanism**. In particular, we extend the BouncyCastle PQC JCA/JCE provider and add the ability to inject and invoke new algorithms. These can be PQC algorithms from the BouncyCastle distribution, PQC algorithms implemented in LibOQS (accessible via the `liboqs-java` Java

[14] BouncyCastle provides pure Java implementations of cryptographic primitives, including the majority of PQC algorithms from NIST Rounds 3 and 4 in the latest releases. BouncyCastle can be downloaded from https://www.bouncycastle.org/java.html.

[15] Our scripts for building such HAProxy are available at https://github.com/LUMII-Syslab/oqs-haproxy.

[16] We used the same approach in our quantum random number generator service https://qrng.lumii.lv [15].

[17] We use TLS v1.3 since it supports KEMs and reduces the number of round-trips in a TLS handshake. KEMs are promoted by NIST, while TLS is an IETF standard supported by all browsers and networking libraries.

wrapper[18]), or other algorithms (such as our "virtual" KEM below). We are working hard on merging our code into the main BouncyCastle distribution. Implementation of the TLS Injection Mechanism is not straightforward and has several non-trivial pitfalls, such as:

- modifying the lists of default KEMs and signature schemes (these lists are sent in the TLS Client Hello message);
- aligning BouncyCastle KEM and signature scheme code points with those used by the OpenQuantumSafe project. Since code points for PQC algorithms are not standardized yet, we stick to the reserved-for-private-use ranges, i.e., 0xFE00..0xFEFF for KEMs and 0xFE00..0xFFFF for signature schemes;
- aligning BouncyCastle and OpenQuantumSafe X.509/X.660 object identifiers (OIDs) for PQC algorithms. These OIDs are used in binary representations of keys and certificates in the ASN.1 notation;
- creating converters between the ASN.1 notation and the internal BouncyCastle representation of keys;
- adding support for PQC keys and certificates (in the ASN.1 DER notation) retrieved from Java key stores, where client private keys and certificates, and CA certificates, are located.

For the butterfly connections, currently, we use the SPHINCS+[19] algorithm for signatures and FrodoKEM[20] as KEM in TLS 1.3. We use AES256-GCM-SHA384 as a cipher suite. For application data, instead of using pure TCP+TLS sockets, we use web sockets since 1) they can be used from client-side code running in web browsers and 2) they are compatible with HTTP(s) traffic (important when configuring firewalls and proxies). All messages in the Butterfly Protocol and Control Protocol are encoded in the ASN.1 binary notation[21], with traditional object identifiers (OIDs) for denoting hash functions.[22]

For the user connection (between User 1 and User 2), we introduce a "virtual" KEM called QKD KEM (we reserve the 0xFEFF code point for it). Unlike in traditional KEMs, no key material is sent via the user connection (hence, KEM is "virtual"); the Butterfly Protocol is executed instead. However, from the BouncyCastle point of view, QKD KEM is treated like any other KEM. We use our TLS Injection Mechanism to add QKD KEM support to BouncyCastle.

In order to implement a KEM, three KEM primitives (*KeyGen*, *Encapsulate*, *Decapsulate*) have to be provided. For some KEMs, each of these primitives is called twice (all six calls are intertwined between the client and the server). In QKD KEM, each of the three KEM primitives is needed once: *KeyGen* and *Decapsulate* on the client side and *Encapsulate* on the server side. The primitives are implemented as follows:

[18] https://github.com/open-quantum-safe/liboqs-java.
[19] NIST PQC Round 3 winner, to be standardized.
[20] NIST PQC Round 3 candidate, not participating in Round 4 but invented by renowned scientists.
[21] since it is a standard, which is already being used for keys and certificates.
[22] thus, hash functions can be upgraded in the future.

- **KeyGen()** at User 1: sends *reserveKeyAndGetKeyHalf* to Aija and *getKeyHalf* to Brencis (see Fig. 2).
  Returns $pk_1 = (keyID, hash(key[L]), hash(key[R]))$ as a public key and $sk_1 = full\ key$ (a concatenation of the key halves) as a secret key. The public key is sent to User 2 in a TLS Client Hello message.
- **Encapsulate(pk_1)** at User 2: sends *getKeyHalf* to Aija and Brencis.
  Returns $sk_2 = full\ key$ (a concatenation of the key halves) as a shared session key for User 2 and $ct_2 = hash(full\ key)$ as a (virtual) ciphertext to be sent back to User 1 in the reply (=TLS Server Hello message).
- **Decapsulate(sk_1, ct_2)** at User 1: validates the hash ct_2 (and performs other checks if domain validation is used).
  Returns the first argument $sk_1 = full\ key$ as is (sk_1 has already been obtained during **KeyGen**). It will serve as a shared session key for User 1. Notice that in the true KEM, sk_1 would be used to decrypt the shared key from the server cipher text (ct_2). In our "virtual" KEM, however, we do not need to perform any actions with sk_1 since the shared key has already been exchanged quantumly and ct_2 contains only the hash.

Our QaaS implementation is available at https://qkd.lumii.lv[23]. So far, we have implemented all required modules and protocols described in this paper except the extra features mentioned in Sect. 4.2.

With our current implementation, we could obtain some preliminary performance test results. Notice that our current implementation has not been optimized and contains some debug code. In our setup, we used BouncyCastle TLS implementation with injected PQC algorithms from LibOQS (via the Java wrapper) and our own pure-Java implementation of the QaaS protocols. We also used our reverse proxies written in Java.

Establishing one PQC TLS link, serializing and sending a short message (a few bytes long), and receiving the result from the server takes 1.57 s on average (on the i7-2600 CPU). TLS-related computations take 96% of that time. Establishing one TLS link with QaaS (by executing the whole Butterfly Protocol) and sending/receiving a message takes 3.75 s on average. Thus, the whole QaaS introduces the 2.38 slowdown factor compared to a single PQC TLS link. With optimizations, we plan to achieve the performance of running the whole QaaS cycle in less than a second on modern CPUs. Thus, using QaaS in the real-world setup seems realistic.

6 Related Work

Multiple attempts to apply the QKD technology in practice have been made. Most software-based solutions are based on re-keying, when the initial symmetric AES keys are replaced by or combined with the QKD keys [18,20]. Both proprietary protocols (such as IDQ Dual-Key agreement) and open modifications to TLS and IPSec have been proposed [4,10]. Such approaches introduce

[23] See also: https://github.com/LUMII-Syslab/qkd-as-a-service.

significant modifications to existing protocols or require the ability to replace AES keys at runtime. In contrast, our QaaS architecture keeps the TLS protocol almost intact (with the exception of reserving the 0xFEFF code point for QKD KEM). However, we factor out the key exchange flow (with the two proposed protocols) as a pluggable KEM.

Several attempts have been made to strengthen TLS security with hardware security modules (HSMs) such as IDQ HSMs and SafeNet Ethernet Encryptors [9]. Currently, HSMs can be integrated into QaaS by developing the corresponding drivers manually. While we anticipate more QKD-certified HSMs, the need for common APIs that ensure HSM interoperability becomes more apparent.

Since QKD devices are expensive, the idea of QKD simulation naturally appeared with QKDNetSim as a representative implementation [18].[24] It correlates with our idea of defining a common Go interface for different QKD implementations, where some implementations act as drivers for real QKD devices while others are used as simulated test environments. Technically, QKDNetSim-like simulators can be plugged into our QaaS software, though our Go interface is very simple and cannot be used to tune simulation parameters—a preconfigured simulator is expected.

KEMTLS is probably the most prominent attempt to eliminate the need for long PQC signatures from the TLS handshake [21]. In KEMTLS, additional KEM invocations are used to authenticate the client and the server implicitly at the cost of non-standard TLS flow. While pursuing the same goal, our Butterfly Protocol differs from KEMTLS in two points:

- We do not modify TLS but introduce a new KEM (QKD KEM).
- We use hash functions, not KEMs, for implicit authentication.

Another goal of KEMTLS is to reduce the number of round-trips used in a TLS handshake. In contrast, our goal was to implement an architecture that is sustainable for active attacks on any single segment. That has been achieved at the cost of a larger number of round-trips in the Butterfly protocol. Still, if we consider the user connection only, the number of round trips remains the same as in TLS 1.3 (i.e., 3 for server-only authentication).

Our idea of domain name verification (in the Butterfly Protocol with Domain Validation) involves checks performed by Aija and Brencis. That resembles the Online Certificate Status Protocol (OCSP), which eliminates the need for certificates and certificates revocation lists (CLRs) on the client side but requires an internet connection to the trusted server [12].

Currently, the QKD technology lacks its own authentication mechanism. In 2021, Wang et al. proposed a PQC-based mutual authentication of multiple QKD network users that trust the same CA [24]. The authentication mechanism is based on PQC signatures that are exchanged via a shared classical link. The process is basically the same as TLS with PQC KEMs and signatures. In our QaaS mechanism, we assume that QKD devices (Alice and Bob) authenticate each other via factory pre-shared keys that can be extended/replaced by new

[24] See also: https://www.qkdnetsim.info and http://open-qkd.eu.

quantumly exchanged keys. Otherwise, if Alice and Bob were relying on PQC, we would get a chicken and egg situation with the User 1↔User 2 connection that relies on the security of the Alice↔Bob link.

7 Conclusion

In this paper, we proposed the "QKD as a service" (QaaS) architecture, two protocols (and the Domain Validation extension to the Butterfly Protocol), the distributed key reservation algorithm, and several authentication mechanisms for QaaS, including some ideas for reducing the number of (very long) post-quantum signatures.

We hope our work will make QKD available for a larger community of end-users. Still, a lot of work is yet to be done, including the development of formal proofs of the proposed protocols, analysis of potential threats and attacks to QaaS, standardization, supporting and extending our reference implementation by reacting to new developments and standards in the PQC field, and developing multi-hop (multi-node) QKD and QaaS (a very large field of research). Besides, we think that PQC algorithms should find their way to Java chip cards, which can be used for more secure user authentication.

We also look forward to integrating the proposed QaaS into our web application infrastructure webAppOS [14].

Acknowledgements. Research supported by the European Regional Development Fund, project No. 1.1.1.1/20/A/106 "Applications of quantum cryptography devices and software solutions in computational infrastructure framework in Latvia".

References

1. Alagic, G., et al.: Status report on the third round of the NIST post-quantum cryptography standardization process. Technical report, NISTIR 8413, NIST (2022)
2. Bennett, C.H., Brassard, G.: Quantum cryptography: public key distribution and coin tossing. In: Proceedings of IEEE International Conference on Computers, Systems and Signal Processing, vol. 175, p. 8 New York (1984)
3. Castryck, W., Decru, T.: An efficient key recovery attack on SIDH (2022). https://eprint.iacr.org/2022/975. Cryptology ePrint Archive, Paper 2022/975
4. Dervisevic, E., Mehic, M.: Overview of quantum key distribution technique within IPsec architecture. In: Proceedings of the 18th International ISCRAM Conference, pp. 391–403 (2021)
5. Dubrova, E., Ngo, K., Grtner, J.: Breaking a fifth-order masked implementation of CRYSTALS-Kyber by copy-paste (2022). https://eprint.iacr.org/2022/1713. Cryptology ePrint Archive, Paper 2022/1713
6. Gao, R.Q., et al.: Simple security proof of coherent-one-way quantum key distribution. Opt. Express **30**(13), 23783–23795 (2022)
7. Guo, Q., Johansson, A., Johansson, T.: A key-recovery side-channel attack on classic McEliece implementations. IACR Trans. Cryptographic Hardw. Embed. Syst., 800–827 (2022). https://doi.org/10.46586/tches.v2022.i4.800-827

8. Guo, Q., Nabokov, D., Nilsson, A., Johansson, T.: SCA-LDPC: a code-based framework for key-recovery side-channel attacks on post-quantum encryption schemes (2023). https://eprint.iacr.org/2023/294. Cryptology ePrint Archive, Paper 2023/294
9. IDQ: Telecom Service Provider: 100G encryption with OKD (use case brochure) (2017). https://www.idquantique.com/resource_type/quantum-safe-security/
10. IDQ: ID Quantique partners with ADVA to commercialise a quantum-safe encryption solution (press release) (2019). https://www.idquantique.com/id-quantiquepartners-with-adva-to-commercialise-a-quantum-safe-encryption-solution/
11. IDQ: Redefining Security: Clavis XG QKD System (2022). https://www.idquantique.com/quantum-safe-security/products/clavis-xg-qkdsystem/
12. IETF Standard: X.509 Internet Public Key Infrastructure: Online Certificate Status Protocol - OCSP (RFC 6960) (2013)
13. Jacak, M., Jacak, J., Jwiak, P., Jwiak, I.: Quantum cryptography: theoretical protocols for quantum key distribution and tests of selected commercial QKD systems in commercial fiber networks. Int. J. Quantum Inf. **14**(02), 1630002 (2016)
14. Kozlovis, S.: The web computer and its operating system: a new approach for creating web applications. In: Proceedings of the 15th International Conference on Web Information Systems and Technologies (WEBIST 2019), Vienna, Austria, pp. 46–57. SCITEPRESS (2019)
15. Kozlovis, S., Vksna, J.: POSTER: a transparent remote quantum random number generator over a quantum-safe link. In: Zhou, J., et al. (eds.) Applied Cryptography and Network Security Workshops. LNCS, vol. 13285, pp. 595–599. Springer International Publishing, Cham (2022). https://doi.org/10.1007/978-3-031-16815-4_32
16. Lo, H.K., Ma, X., Chen, K.: Decoy state quantum key distribution. Phys. Rev. Lett. **94**(23), 230504 (2005). https://doi.org/10.1103/PhysRevLett.94.230504
17. Mailloux, L.O., Hodson, D.D., Grimaila, M.R., Engle, R.D., Mclaughlin, C.V., Baumgartner, G.B.: Using modeling and simulation to study photon number splitting attacks. IEEE Access?: Pract. Innovations Open Solutions **4**, 2188–2197 (2016)
18. Mehic, M., Maurhart, O., Rass, S., Voznak, M.: Implementation of quantum key distribution network simulation module in the network simulator NS-3. Quantum Inf. Process. **16**(10), 253 (2017)
19. Mehic, M., et al.: Quantum key distribution: a networking perspective. ACM Comput. Surv. **53**(5), 1–41 (2021)
20. Neppach, A., et al.: Key management of quantum generated keys in IPsec. In: Proceedings of the 3rd International SECRYPT Conference, pp. 177–183 (2008)
21. Schwabe, P., Stebila, D., Wiggers, T.: Post-quantum TLS without handshake signatures. In: Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security, pp. 1461–1480. ACM, Virtual Event USA (2020)
22. Stanley, M., Gui, Y., Unnikrishnan, D., Hall, S., Fatadin, I.: Recent progress in quantum key distribution network deployments and standards. J. Phys: Conf. Ser. **2416**(1), 012001 (2022)
23. 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 International Publishing, Cham (2017). https://doi.org/10.1007/978-3-319-69453-5_2

24. Wang, L.J., et al.: Experimental authentication of quantum key distribution with post-quantum cryptography. npj Quantum Inf. **7**(1), 67 (2021)
25. Wimmer, C.: GraalVM native image: large-scale static analysis for Java (keynote). In: Proceedings of the 13th ACM SIGPLAN International Workshop on Virtual Machines and Intermediate Languages, pp. 3–3. ACM (2021)