Article

The Butterfly Protocol: Secure Symmetric Key Exchange and Mutual Authentication via Remote QKD Nodes †

Sergejs Kozlovičs *, Elīna Kalniņa, Juris Vīksna, Krišjānis Petručeņa and Edgars Rencis

Institute of Mathematics and Computer Science, University of Latvia, LV-1459 Riga, Latvia; elina.kalnina@lumii.lv (E.K.); juris.viksna@lumii.lv (J.V.); krisjanis.petrucena@lumii.lv (K.P.); edgars.rencis@lumii.lv (E.R.)

* Correspondence: sergejs.kozlovics@lumii.lv; Tel.: +371-29513256

† The article builds on the foundation of our original paper: Kozlovičs, S.; Petručeņa, K.; Lāriņš, D.; Vīksna, J. Quantum Key Distribution as a Service and Its Injection into TLS. In *Information Security Practice and Experience*; ISPEC 2023. Lecture Notes in Computer Science; Meng, W.; Yan, Z.; Piuri, V., Eds.; Springer: Singapore, 2023; Volume 14341, pp. 527–545. https://doi.org/10.1007/978-981-99-7032-2_31.

Abstract

Quantum Key Distribution (QKD) is a process to establish a symmetric key between two parties using the principles of quantum mechanics. Currently, commercial QKD systems are still expensive, they require specific infrastructure, and they are impractical for deployment in portable or resource-constrained devices. In this article, we introduce the Butterfly Protocol (and its extended version) that enables QKD to be offered as a service to non-QKD-capable (portable or IoT) devices. Our key contributions include (1) resilience to the compromise of any single classical link, (2) protection against malicious QKD users, (3) implicit mutual authentication between users without relying on large post-quantum certificates, and (4) the Double Butterfly extension, which secures communication even when the underlying QKD network cannot be fully trusted. We also demonstrate how to integrate the Butterfly Protocol into TLS 1.3 and provide its initial security analysis. We present preliminary performance results and discuss the main bottlenecks in the Butterfly Protocol implementation. We believe that our solution represents a practical step toward integrating QKD into classical networks and extending its use to portable devices.

Keywords: quantum key distribution; QKD; post-quantum cryptography; TLS; butterfly protocol

1. Introduction

Symmetry appears everywhere. In cryptography, the first examples that come to mind are symmetric ciphers. The one-time pad (OTP) offers information-theoretic security but requires a truly random key as long as the message. There are also symmetric stream ciphers (e.g., AES in GCM mode) that use relatively short keys (e.g., 128, 192, or 256 bits in AES) to encrypt an entire communication session. AES-256 is also considered quantum-safe, offering $128 - \frac{1}{2}\log P$ bits of security in the post-quantum era when accounting for Grover's search using P parallel quantum processors.

Quantum Key Distribution (QKD) is a process for generating a secret symmetric key shared by two parties (Alice and Bob) that relies on the laws of physics and ensures that the keys have not been eavesdropped on or modified by a third party. While traditional key exchange algorithms rely on computationally-heavy tasks, QKD provides perfect forward secrecy and is resistant to "harvest-now, decrypt-later" attacks. QKD is impossible to

Academic Editor: Hsien-Chung Wu

Received: 11 December 2025
Revised: 1 January 2026
Accepted: 7 January 2026
Published: 14 January 2026

Copyright: © 2026 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license.

break algorithmically because is it based on principles of quantum mechanics (although implementation-specific and side-channel attacks are still possible). QKD is also the first step towards a universal Quantum Internet [1].

QKD keys are typically 256 bits long and used directly as AES-256 keys. QKD may also be used to generate a potentially infinite, future-proof OTP key, albeit with a very low bit rate using current technology.

In this paper, we propose the Butterfly Protocol and its extended version. The protocol enables non-QKD-capable devices (such as portable or IoT devices) to access a QKD network via classical links. Naturally, these classical links must be secured. In fact, our protocol makes them even more secure than PQC (Post-Quantum Cryptography)—a non-quantum alternative for post-quantum security. Interestingly, symmetry emerges throughout our construction: the links of the Butterfly Protocol resemble the symmetric wings of a butterfly, while the Double Butterfly Protocol exhibits even greater symmetry. We also show how to achieve mutual authentication (which is inherently symmetric) between communicating users. In addition, we demonstrate how a JWT token storing a symmetric key can be used to authenticate users accessing QKD keys.

Although the initial version of the Butterfly Protocol was published in 2023 [2,3], this article introduces new important features:

1. Additional security against other QKD users, who could act as a man (or woman) in the middle. For that, we utilise an additional authenticated classical link.
2. Implicit mutual authentication between both communicating users (Users 1 and 2) without the need to rely on PQC certificates (which may be quite substantial).
3. In addition, we propose an extended version of the Butterfly Protocol (the "Double Butterfly"), where we make it secure against an untrusted QKD network.

We proceed by justifying the need for the Butterfly Protocol (Section 2). Then we outline the required QKD network setup (Section 3), after which we present a bird's-eye view on the Butterfly Protocol (Section 4). Afterwards, we give a detailed protocol explanation, non-PQC-based mutual authentication, and mention some technical issues (Sections 5–9), followed by the extended Butterfly Protocol version, which is resilient against a single malicious QKD node or link (Section 10).

2. QKD Status Quo and the Need for the Butterfly Protocol

There are numerous QKD protocols with different properties. The very first QKD protocol was a discrete-variable prepare-and-measure BB84, with the security proof appearing only in 2000 [4,5]. Other popular protocols are B92, SARG04, Lo05, BBM92, and the Ekert protocol [6–10]. Some are prepare-and-measure protocols, some are entanglement-based, and some have both variations. Distributed-phase-reference (DPR) protocols is a specific subclass of discrete-variable protocols, with COW (Coherent One-Way protocol, patented by IDQ, uses weak coherent pulses in time bins) being a famous representative [11].

There are also continuous-variable protocols, such as GG02 and Leverrier–Grangier [12,13]. They use laser pulses and homodyne or heterodyne detectors for estimating leakage.

Sadly, specific side-channel attacks are possible for discrete variable protocols, such as the photon number splitting (PNS) attack, the detector blinding attack, and time-shift attacks, among others [14]. The initially claimed security of the DPR COW protocol has been shown to be unsuitable for long-distance QKD [15]. There are various attacks on continuous variable protocols, as well, including beam-splitting, intercept-resend, time-shift, coherence attacks, and others. Entanglement-based protocols, however, have less potential for a man-in-the-middle attack, since the source of entangled Bell states does not need to be trusted. Nevertheless, proper implementation is important to mitigate side-channel and entanglement-related attacks.

The good news is that these attacks are either difficult to exploit over short distances or can be impeded by more precise photon receivers. Thus, for practical purposes, we can assume a short- to mid-term security of modern QKD technology with the hope for near-perfect security of future QKD devices.

However, we face another issue: while commercial QKD devices are available on the market, they are expensive, require specific infrastructure, and have high operational expenses. Thus, they are not affordable for everyone. Furthermore, it is impossible to integrate QKD-specific hardware components (such as single photon detectors and cooling systems) into physically small or low-resource devices such as smartphones and IoT devices.

To address these limitations, we propose using classical Internet channels between the two communicating parties (User 1 and User 2) and two QKD Key Distribution Centres (KDCs), with each KDC responsible for one QKD node in the underlying QKD network. Unlike traditional QKD architectures, however, each communicating party interacts with both KDC nodes. This design is at the core of the Butterfly Protocol, where no single classical communication channel ever carries the entire key. As a result, an active adversary would have to wiretap and decrypt at least two independent classical links to obtain the full key. Furthermore, all classical communication between users and KDCs is protected by quantum-secure post-quantum cryptographic (PQC) algorithms, making decryption infeasible even in the quantum setting.

Importantly, unlike all classical cryptographic systems, even QKD implementations that are vulnerable to side-channel attacks retain a crucial security property: side-channel attacks can succeed only if they are performed actively during the key-exchange process. If an attacker fails to intercept the key at the moment of transmission, the generated keys remain information-theoretically secure and cannot be recovered later with any feasible amount of computation (other than brute-force search over the entire key space).

3. The Network Architecture

Figure 1 depicts the network setup assumed by the Butterfly Protocol.

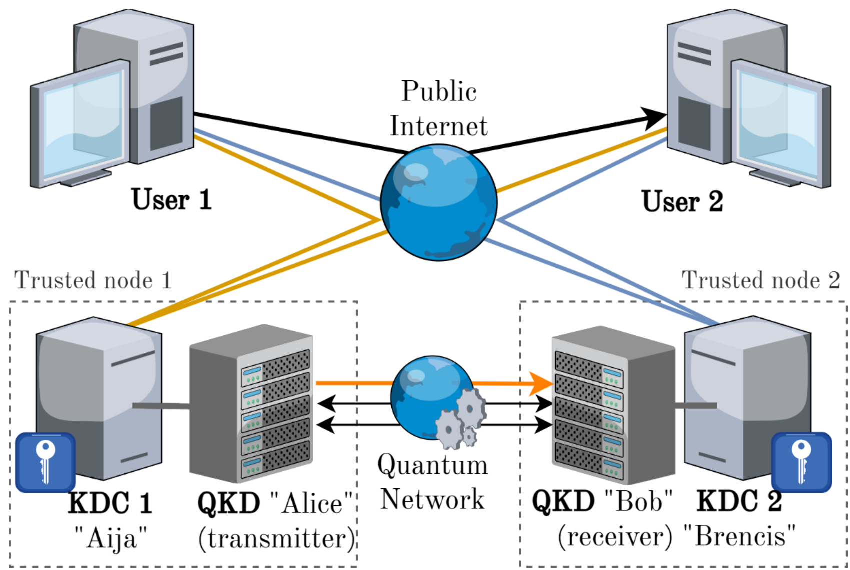

Figure 1. Prerequisites for the Butterfly Protocol. Physical QKD nodes implement a quantum channel (directly, or via a QKD network, depicted as a globe with cogs). QKD devices are accessed via KDCs (which we refer to as Aija and Brencis to distinguish them from direct QKD hardware). The protocol will use all four links (through the classical Internet) between Users 1 and 2 and KDCs 1 and 2.

Two QKD devices (Alice and Bob) are connected by multiple links (classical and quantum). The classical links (usually bidirectional) are needed for service data and for

implementing an authenticated channel for QKD protocols. Quantum links (one or more) can be unidirectional, depending on the QKD protocol. Usually, QKD keys are not obtained directly from quantum devices, but via some web service implementing a protocol for retrieving keys (such as ETSI GS QKD 014 or CISCO SKIP), which can be proxied or secured behind a firewall [16,17]. Hereinafter, we assume that key distribution centres (KDCs) provide a secure means to retrieve QKD keys. KDCs may implement key relay, proactive buffering, and other useful services such as authentication and authorization.

Due to distance limitations of a single QKD link, trusted nodes are introduced to extend the QKD range. According to the ETSI GS QKD 014 standard, a trusted node is a secured site that situates one or more QKD Entities together with a Key Management Entity (KME). In this work, we refer to KMEs as Key Distribution Centres (KDCs), re-using the term from Kerberos and other non-QKD systems. KDCs can be interconnected to a QKD network with a non-trivial topology, implementing some key relay mechanism. Typically, in such topologies, users of QKD keys can request a QKD key from any node (e.g., the geographically closest node) within the topology. In this paper, we use the term "trusted node" to denote all such nodes regardless of their function within the QKD network. Communication with trusted nodes is done through KDCs.

4. The Essence of the Butterfly Protocol

Traditionally, each of the users (Users 1 and 2) relies on a single geographically closest QKD node (Alice or Bob). Since the communication between them takes place over a classical encrypted channel (such as TLS), the overall security is limited to that of the classical encryption scheme in use. Still, using QKD-generated keys for long-distance connections, while relying on less secure local links for key acquisition, can enhance overall security [18].

Figure 2 introduces the Butterfly Protocol, which addresses the "weakest link" problem. When two users want to communicate, each connects to both KDCs (Aija and Brencis), which are linked to the paired QKD devices (Alice or Bob). The protocol establishes a QKD-keyed TLS session between User 1 and User 2 (the topmost link, referred to as the "user link" after performing key-establishing routines during the TLS handshake), with encryption keys provided by the KDCs. Each KDC only sends part of the QKD session key—either the left part (L) or the right one (R) to both users (via the "butterfly links"). Key halves are then verified against each other using the hash values H(R) and H(L), which are sent as complementary material. The added security of the Butterfly Protocol relies on the assumption that an attacker would need to compromise at least two independent TLS connections to succeed. Connections between Users 1 and 2 and the KDCs use standard TLS (v1.3), however, in our setup, we enhanced security using post-quantum cryptography (PQC) for key exchange. Besides, all classical links from Figure 2 must be authenticated, which can be ensured via PQC certificates or hash-based JWT tokens, as explained in the next sections.

An important addition to our original Butterfly Protocol is the "KDC link" at the bottom of Figure 2 (the quantum link is hidden behind the KDCs). The KDC link is an *authenticated* classical channel, which is required to prevent other QKD users—who possess valid KDC credentials—from impersonating User 2. For example, if User 3 impersonates User 2, then he may act as a man in the middle between User 1 and User 2 (though he will need to rely on two QKD key pairs: User 1 $\leftrightarrow$ User 3, and User 3 $\leftrightarrow$ User 2). We strongly argue that the KDC link must also be encrypted. Although it is not essential for the security of the protocol, this property ensures privacy: third parties will not be able to find out that communication has occurred between User 1 and User 2.

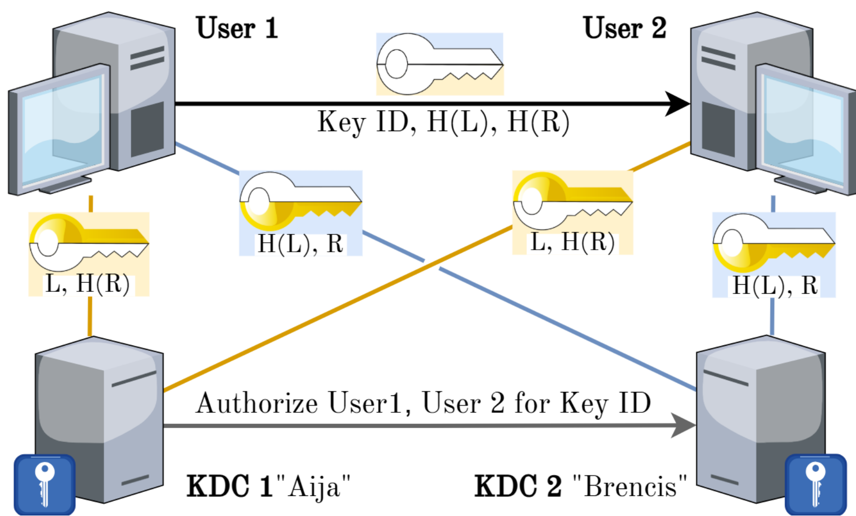

Figure 2. The six classical links in the Butterfly Protocol: the data link (top-most, TLS with QKD keys), the four butterfly links (PQC), and the link between KDCs.

5. The Detailed Explanation of the Butterfly Protocol

The Butterfly Protocol allows User 1 and User 2 to agree on a shared key without transmitting the full key via any of the classical links. In our narrative, one KDC endpoint (Aija) sends only the first half of the key, and the other (Brencis) sends the second half. However, we can randomize that, since each KDC knows the full key and can be switched from returning left halves to right halves, or vice versa.

Before running the protocol, User 1 chooses one of the two KDCs, which he/she will contact first; we assume that it is Aija. A symmetric alternative (when User 1 selects Brencis instead) is also possible. It means that Aija and Brencis can share the key reservation role equally. We refer to this as the "equivalence property" of the KDCs. This property ensures that there is no benefit in targeting one KDC over the other. It also allows for designing algorithms that can intentionally choose which KDC receives the first protocol message.

The protocol starts when User 1 wants to communicate with User 2 via a TCP connection (Figure 3). A standard TLSv1.3 flow between User 1 (the client) and User 2 (the server) is executed. However, instead of running a traditional Diffie–Hellman key exchange, we perform the flow of the Butterfly protocol. In Section 7, we show how to integrate it into TLSv1.3 in a manner similar to post-quantum key exchanges.

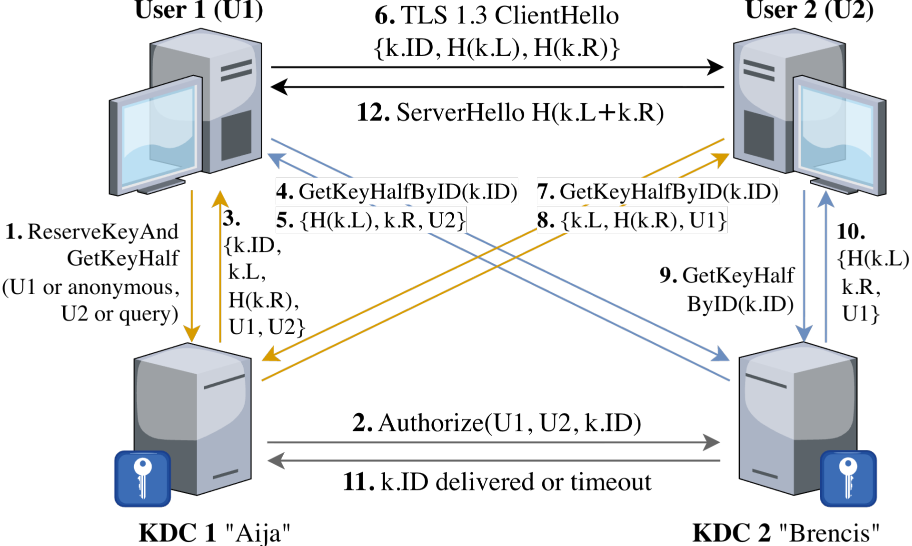

Figure 3. Message flows in the Butterfly Protocol.

We use the following shorthands: k is a QKD-generated session key, split into a left half $k.L$ and a right half $k.R$; $H(\cdot)$ denotes a cryptographic hash; and the communicating users are abbreviated as U1 and U2 (the reader may assume that U1 and U2 correspond to users' distinguished names such as specified in the X.500/X.509 standards [19]). Error handling details are omitted from this description. Authentication is addressed in Section 6.

1. U1 $\rightarrow$ Aija: sends `reserveKeyAndGetKeyHalf`, stating the intent to talk with U2. KDC1 (Aija) reserves some QKD key (from the key buffer) and associates the corresponding key ID with users U1 and U2. Aija serves only the left half of the key, $k.L$, which is returned to U1 in Step 3 and to U2 in Step 8, while KDC2 (Brencis) serves only the right half of the key ($k.R$ in Steps 5 and 10). Normally, U1 and U2 denote user identities, with some exceptions discussed in the notes below.
2. Aija $\rightarrow$ Brencis: notifies that only U1 and U2 may request the right half $k.R$
3. Aija $\rightarrow$ U1: returns $k.ID$, $k.L$, and $H(k.R)$; Aija also records U2 as the second addressee
4. U1 $\rightarrow$ Brencis: requests the second (right) half via `getKeyHalf`$(k.ID)$
5. Brencis $\rightarrow$ U1: supplies $H(k.L)$ and $k.R$; U1 cross-checks $H(k.L)$ and $k.R$ with $k.L$ and $H(k.R)$ obtained in Step 3;
6. U1 $\rightarrow$ U2: starts TLSv1.3 handshake (*ClientHello*): U1 ID, $k.ID$, $H(k.L)$, $H(k.R)$
7. U2 $\rightarrow$ Aija: requests the first (left) half `getKeyHalf`$(k.ID)$
8. Aija $\rightarrow$ U2: returns $k.L$ and $H(k.R)$;
9. U2 $\rightarrow$ Brencis: requests the second half `getKeyHalf`$(k.ID)$
10. Brencis $\rightarrow$ U2: sends $H(k.L)$ and $k.R$; U2 cross-checks $H(k.L)$ and $k.R$ with $k.L$ and $H(k.R)$ obtained in Step 8; U2 also validates the hashes with $H(k.L)$ and $H(k.R)$ received in Step 6;
11. Brencis $\rightarrow$ Aija: reports whether $k.R$ reached both users or timed out
12. U2 $\rightarrow$ U1: sends $H(k.L||k.R) = H(full\ key)$ in ServerHello; U1 verifies it

Some notes:

- Steps 1 and 2 bind User 1 and 2 identities to the reserved key. This binding is established within KDC1 and transferred to KDC2 via an authenticated channel. Thus, we do not need to send users' identities in Step 6.
- Steps 4 and 6, as well as steps 7 and 9, can be launched in parallel. Aija communicates with Brencis via an authenticated KDC link.
- In the end, User 1 validates its $full\ key$ against the hash provided by User 2. Since User 2 cannot efficiently compute $hash(full\ key)$ without interacting with Aija and Brencis (the knowledge of $H(k.L)$ and $H(k.R)$ does not help), and because Steps 1 and 2 guarantee that only Users 1 and 2 can obtain the key halves from the KDCs, User 1 can conclude that UserZ2 has been successfully authenticated by both KDC1 and KDC2 (we use this property in Section 6).
- Similarly, Steps 1 and 2—together with $H(k.L)$ and $H(k.R)$—serve as proof to User 2 that User 1 has been authenticated by both KDC1 and KDC2. However, User 2 needs to know the identity of the counterparty (User 1), which is provided by both KDCs in replies for `GetKeyHalfByID`.
- We intentionally omit sending U1 in Step 6 because the ClientHello message is unencrypted, and including U1 would create a privacy leak for User 1. However, if User 1 wishes to remain anonymous also to User 2 (e.g., when client authentication is not needed at the User 2 side), he may request that KDC1 generate a random U1 value in Step 1. Likewise, the U2 sent in Step 1 can represent a query used to identify User 2. For this reason, `reserveKeyAndGetKeyHalf` returns both U1 (which may be a newly generated random identifier) and U2 (which may correspond to the distinguished name of User 2 that satisfies the query).

- User1 sends $H(k.L)$ and $H(k.R)$ to User2, while User2 replies with $H(\text{full key})$. That ensures mutual and *independent* verification that the other party has the knowledge of the full key.
- The reply sent in Step 12 may be transmitted unencrypted. If the reply is forged, User 1 will simply discard the hash. If it is eavesdropped, the attacker still gains no information about the QKD key, which remains secret.

6. Authentication Without PQC

There are two authentication "axes":

- The "vertical" authentication between users and KDCs. Users 1 and 2 must first verify the authenticity of both KDCs before accepting any key material; otherwise, a forged or hijacked KDC could inject malicious values. Conversely, KDCs must also authenticate clients — not just for billing in a commercial "QKD as a Service" (QaaS) setting, but also to prevent key-starvation, DDoS floods, and rogue-insider man-in-the-middle (MITM) attacks in general.
- The "horizontal" authentication between User 1 (the "client") and User 2 (the "server").

While it is possible to use PQC certificates to authenticate all the classical links from Figures 2 or 3, we propose a way to authenticate all parties (in both axes) without long PQC signatures. Here, we do not consider the authentication of the classical channel between QKD devices (Alice and Bob), as they are "black boxes" from our perspective.

Our first idea is to introduce a Butterfly Protocol extension for implicit mutual horizontal authentication, which aborts the Butterfly Protocol (and thus the entire TLS handshake between Users 1 and 2) in the event of forgery. This would remove the need for PQC certificates on the user link. Our second approach is to eliminate PQC certificates on vertical links by replacing them with hash-based signatures, such as HMAC-SHA256–based JWT tokens (https://jwt.io, accessed on 6 January 2026). Notice that JWT variants other than hash-based are not quantum-safe.

Recall (Section 5) that hash values for the full QKD key and its halves can be used by both User 1 and User 2 as proofs that the counterparty has been authenticated within one or both KDCs (Steps 10 and 12 of the Butterfly Protocol). We use this property to extend the Butterfly Protocol with support for mutual client and server authentication.

First, we assume that each KDC possesses a unique secret key used to sign the header and payload (e.g., the client distinguished name + token expiration date + salt) of JWT tokens. Each user (User 1, User 2, etc.) is given *two* tokens—signed by KDC1 and KDC2, respectively. Otherwise, compromising just one link (for example, the connection between User 1 and KDC1) would enable an attacker to impersonate that user when communicating with the second KDC. Although JWTs do not need a database, they still require a way to revoke old tokens, similar to a certificate revocation list (CRL).

Second, we update the Butterfly Protocol by sending the user's JWT token with any request to Aija and Brencis. However, since the token is a private resource, every TLS connection to Aija and Brencis starts with server-only authentication (which can be done via a pre-shared key, generated for each user individually, and distributed together with a JWT token). After a TLS connection with the server has been established, the client sends its JWT token, bitwise XOR-ed with the pre-shared key (or some function on the pre-shared key) in the encrypted application data, thus preventing direct eavesdropping on the token. Bitwise XOR is used here to conceal the token in case an active eavesdropper compromises one of the butterfly channels.

Third, we update the Butterfly Protocol in the following way: every time Aija or Brencis receives a client message (and, hence, a JWT token), they verify it by performing a database lookup or by verifying the hash-based JWT signature. In case of failure, KDC

drops the connection, which terminates the Butterfly Protocol (and, hence, TLS) with an exception. Thus, if the Butterfly Protocol indeed establishes a TLS connection between users, both users can be sure that they are mutually authenticated and posses the same QKD key.

While JWT-based authentication can reduce latency by approximately 30%, as we can conclude from our experiments, it comes at the cost of relying on a non-standard approach (in contrast to digital certificates, which are part of the Public Key Infrastructure, PKI) and a more complex deployment, requiring distributing two JWT tokens and a pre-shared key to each user.

7. Integration into TLS

In order to integrate the Butterfly Protocol, we extend TLSv1.3 protocol implementation with our original TLS Injection Mechanism. It provides the ability to add virtually any KEM (Key Encapsulation Mechanism) and digital signature algorithm to the TLS handshake process. KEM is an abstraction for Diffie–Hellman key exchange, used for PQC algorithms. Each KEM must implement three API functions: $keyGen() \rightarrow pk1, sk1$ (by User 1), $encapsulate(secret, pk1) \rightarrow ct2$ (by User2), and $decapsulate(ct2) \rightarrow secret$.

During the TLSv1.3 handshake, the client (User 1) sends the ClientHello message to the server (User 2). We are interested in the following lists within the message:

- The list of ciphersuites (e.g., AES-256 in GCM mode). We do not alter this list, since asymmetric ciphers with 256 bits are considered quantum safe.
- The list of supported KEMs (the “supported_groups” extension in TLSv1.3). We alter it by “injecting” the Butterfly Protocol as the only supported (and the default) KEM.
- The list of supported digital signature algorithms (the “signature_algorithms” extension). We alter it by specifying PQC signature algorithms or our own horizontal authentication method described earlier in Section 6.

The lists are represented by 16-bit integers, which we refer to as “code points” (the term used by the OpenQuantumSafe project, analogous to Unicode code points). As of late 2025, standard code points exist only for certain PQC KEM algorithms, while no code points have yet been standardized for PQC digital signature algorithms. For signature schemes, we also require algorithm OIDs (object identifiers) for use in certificates, but standardized OIDs for PQC digital signature algorithms are not yet available. However, we can rely on temporary code points and OIDs defined by OpenQuantumSafe (see https://github.com/open-quantum-safe/oqs-provider/blob/main/ALGORITHMS.md, accessed on 6 January 2026). BouncyCastle v1.8+ has implemented their own solution to support PQC in TLS. However, their code points and OIDs are hardcoded, and there can be compatibility issues with other libraries (like OpenSSL) depending on which version of a PQC standard (or draft) they rely on.. The temporary code points lie in the reserved-for-private-use ranges, i.e., 0xFE00..0xFEFF for KEMs and 0xFE00..0xFFFF for signature schemes. For Butterfly Protocol (implemented as a KEM), we reserve the 0xFEFB code point. For the mutual horizontal authentication via the Butterfly protocol, we reserve the 0xFFFB code point (the last hexadecimal digit “B” stands for “butterfly”.)

In order to support PQC algorithms, we had to rely on some open-source TLSv1.3 implementation and modify it to be able to inject non-standard key exchange and digital signature algorithms. We chose the BouncyCastle (https://www.bouncycastle.org/download/bouncy-castle-java/, accessed on 6 January 2026) library as a code base for our TLS Injection Mechanism (the mechanism itself is outside the scope of this paper). BouncyCastle is distributed under the MIT license (with some Apache 2.0-licenses modules), and provides pure Java implementations of TLS and cryptographic primitives (including PQC algorithms and certificate management).

In order to inject the Butterfly Protocol into TLSv1.3, we define a "virtual KEM", called ButterflyKEM, which implements the same PQC KEM API (*keyGen*, *encapsulate*, and *decapsulate*) but does not encapsulate/decapsulate the key. Instead, ButterflyKEM runs the flow of the Butterfly Protocol. The ephemeral User 1 public key (in TLS terms) is just a plain message sent in Step 6 (see Figure 4). The encapsulated value, sent by User 2, is just the message sent in Step 12.

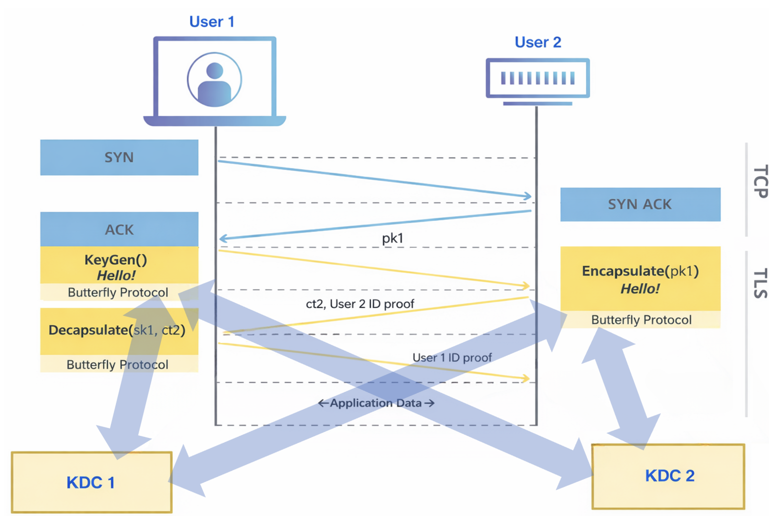

Figure 4. Butterfly Protocol integration into TLSv1.3.

8. Security of the Butterfly Protocol

We analyze the security of the Butterfly Protocol under an active attacker model, in which the adversary has access to a quantum computer. We do not consider denial-of-service attacks, as all QKD protocols fail if the attacker continuously eavesdrops on the quantum channel—an effect analogous to physically cutting the wire. Besides, we assume that the PQC private keys (or the pre-shared keys used to conceal the JWT tokens) of users 1 and 2 are not compromised. However, the adversary may attempt to decrypt any of the classical links and to impersonate User 1, User 2, Aija, or Brencis, using methods other than possession of the private key.

First, we axiomatize the security of the QKD link, where an attacker can be detected using quantum mechanics and QKD protocols.

As mentioned in Section 4, the link between KDC1 and KDC2 should be encrypted. That can be done using a symmetric key obtained from QKD devices. The key can be used to encrypt the whole channel using a quantum-safe symmetric cipher (e.g., AES-256) with some key update interval, or multiple QKD keys can be combined for OTP encryption.

All butterfly links are encrypted using PQC key exchange. We also need to authenticate them, e.g., using PQC certificates or JWT tokens, see Section 6. Although we refer to "PQC" here, state-of-the-art practical deployments should use a hybrid approach—for example, combining classical elliptic-curve cryptography with PQC algorithms—to reduce the risk posed by potential vulnerabilities in PQC schemes.

Notice that among all butterfly links, there is no link where the full shared key is sent. Since the full key is never transmitted via any single classical link, an active attacker (with the computational power for breaking PQC) must eavesdrop and decrypt at least two butterfly connections to obtain the QKD-generated key. We rely on the strength of PQC (and on classical cryptography as well, if a hybrid approach is used). Although it is possible—however unlikely—that vulnerabilities may be discovered in PQC algorithms or that new quantum attacks could emerge, we can mitigate this risk by using different PQC key-exchange algorithms for different butterfly links. Nevertheless, the practical feasibility

of such attacks remains highly limited. In the case of JWT tokens, security relies on the secrecy of the corresponding pre-shared key and the cryptographic strength of the hash function (SHA-256) used to sign the tokens.

Although compromising either KDC1 or KDC2 would reveal the full key, KDCs are generally located in physically secure, monitored facilities. Eavesdropping near a KDC's boundary still only exposes half of the key. Physically securing the KDCs also helps prevent unknown key-share (UKS) attacks, in which two parties share the same key but disagree on each other's identity. This is achieved because the reserved key ID is bound to User 1 and User 2 within KDC1 and then securely transferred to KDC2 via an authenticated channel. Even if the peers are misidentified, the Butterfly Protocol terminates with an exception, preventing any compromise of the user's TLS session.

Eavesdropping near User 1 or User 2 could be possible if both butterfly links originating from the same user are routed through the same Internet Service Provider (ISP). While a user can mitigate this risk by relying on two different ISPs—e.g., one mobile and one fiber-optic connection—even with a single ISP, simultaneously decrypting both butterfly links remains infeasible, since different session keys are used. Besides, it is possible to encrypt these butterfly links using different PQC algorithms.

The tastiest morsel for attacks is the user link, since it remains unencrypted until the Butterfly Protocol completes. Its strength relies, to a certain extent, on the underlying hash function (e.g., SHA3-256, which has proofs in the quantum random oracle model). However, even if preimage resistance for the hash function H is broken (which is unlikely for SHA3), the probability of successful hash-based attacks can be made negligible if we use, e.g., a 128-bit hash with 512-bit QKD keys: for any half of the key (256 bits), we would have $\approx 2^{128}$ preimages!

Let us estimate the max-entropy known to the attacker, if all three hashes are known. L is a 256-bit string, so without any information there are 2^{256} possibilities (entropy = 256 bits). Knowing $h(k.L)$ (a 128-bit value) pins L to about $2^{256-128} = 2^{128}$ preimages. So the uncertainty about L drops from 256 bits to 128 bits. The same is for R, keeping 256 bits of entropy. However, since the attacker knows also $H(\text{full key})$, the remaining entropy drops to 128 bits. Here we assume that L and R are uncorrelated, since they are generated using some quantum RNG embedded into a QKD device.

Still, we can assume that recovering L, R, or *full key* from $h(k.L)$, $h(k.R)$, and h(*full key*) is infeasible with current technology. Even if pre-image resistance is somehow broken, we still have 128 bits of undisclosed entropy, which is sufficient to send a key ID for the next pair of QKD keys in a secure way. Then, User1 and User2 can ask for the corresponding key directly from QKD devices (Alice and Bob).

Notice that although the protocol relies on hash values, they are used solely as checksums. Shorter hashes (or checksums) could provide stronger protection against key preimage attacks, albeit at the price of reduced confidence in User 1 and User 2 identities, when using implicit horizontal authentication from Section 6.

9. Additional Technical Challenges

Since both Aija and Brencis can be used for key reservation, they need a distributed algorithm that resolves conflicts between them and prevents race conditions, such that no key is reserved twice for different communication channels. Besides, if a key has been reserved at one KDC, both KDCs must be ready to send their key halves to Users 1 and 2 and then delete the used key or the "zombie" key (appearing when the protocol has started but not finished, e.g., due to a network interruption) afterwards.

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 considerable amount of time (usually several minutes). Besides, during re-initialization, the QKD devices consume more power than during regular operation.

To mitigate the consequences of a system interruption, KDCs maintain buffers of previously generated QKD keys. However, this imposes additional synchronization procedures to ensure that Aija and Brencis always have the same set of key ID and value pairs after partial QKD key stream failure. Moreover, keys have an incubation period before they can be reserved to ensure delivery at the other endpoint and a grace period before deletion, e.g., in the case of key rotation when disk space is finite.

All the issues just mentioned are technical and can be managed by the Control Protocol, which has already been published in our first paper [2]. Since the Control Protocol requires access to the internal data structures of KDCs, additional precautions have to be enforced when executing it (e.g., introducing separate authentication credentials, reverse proxies, and access control).

10. The Double Butterfly Protocol

So far, we have assumed that the quantum link is secure due to Quantum Mechanics, QKD protocols, and (hopefully) secure implementation. We also expect QKD device certification in the future.

However, at the initial stage, the QKD network will be in limited availability, controlled by large telecommunication providers and government agencies. Moreover, since end users do not control the network, they may wish to protect themselves from unscrupulous QKD nodes. Our idea is to further strengthen the security of the Butterfly Protocol by obtaining two pairs of keys: one from one QKD pair (Alice 1 and Bob 1) and another from the second QKD pair (Alice 2 and Bob 2). Of course, the keys are not obtained directly, but rather through KDCs, as shown in Figure 5. Now, when each user (User 1 and User 2) has two keys k_1 and k_2, they can use some cryptographically secure key derivation function $k \leftarrow KDF(k_1, k_2)$ to obtain a combined shared key k. This key will be safe even if one of the keys (k_1 or k_2) is compromised. Thus, we can argue that the Double Butterfly protocol is sustainable against one malicious or compromised QKD node (since, in that case, either k_1, or k_2 becomes compromised, but the key k is still safe).

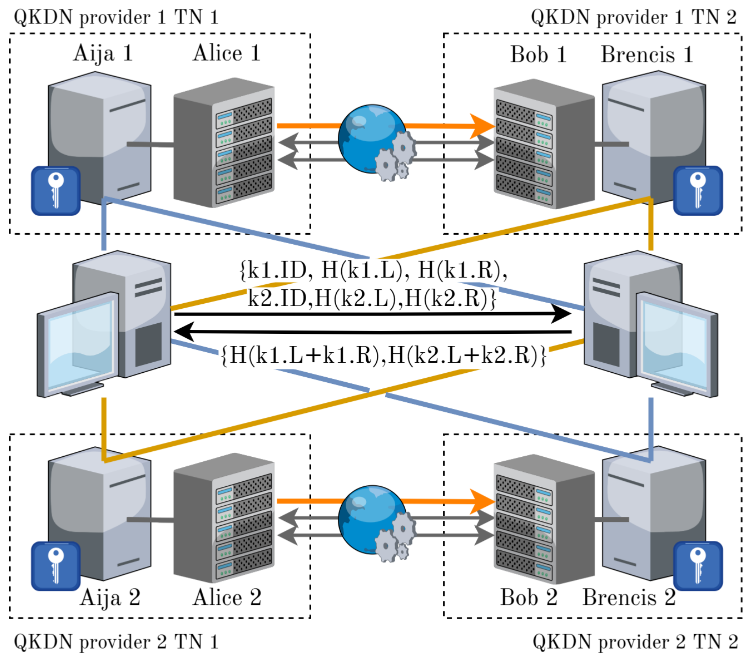

Figure 5. The conceptual schema of the Double Butterfly Protocol.

11. Results

In this paper, we have enhanced the original Butterfly Protocol (published in 2023 [2,3]) by incorporating several key improvements. First, we have strengthened its resilience against adversarial QKD users by introducing an additional authenticated classical channel. Second, we have achieved implicit mutual authentication between the two parties, removing the dependence on potentially large post-quantum certificates. Lastly, we have proposed the Double Butterfly Protocol extension, which ensures protocol security even when some nodes or links of the underlying QKD network cannot be trusted. These developments mark a significant step toward more practical and secure quantum key distribution in real-world scenarios. Our software, which implements the Butterfly Protocol and runs on Aija and Brencis, has been developed using the Go programming language, which has superb built-in concurrency support. To support QKD devices from different vendors, we implement a Go interface named *KeyGatherer* that is used to obtain streams of indexed keys, i.e., tuples $(keyID, key, timestamp)$.

Since the Go code of our KDC service does not use TLS, the post-quantum key management and TLS implementation on the server side is provided by the reverse proxies. Sadly, none of the current software that could be used as a reverse proxy (including, haproxy and nginx) can parse PQC signatures, even if compiled against OpenSSLv3 with PQC algorithms provider oqsprovider" (based on LibOQS from the OpenQuantumSafe project). Thus, we have developed our own reverse proxy, pqproxy", which uses BouncyCastle TLS implementation and digital signatures and KEMs provided by LibOQS [20]. The current implementation of the Butterfly Protocol is available at https://butterfly.quantum.lumii.lv/ (accessed on 6 January 2026).

Table 1 shows Butterfly Protocol performance in different scenarios.

Table 1. Butterfly Protocol performance (not optimized, logging enabled).

Test Scenario	Mean Time (ms)	Standard Deviation
Full Butterfly (encrypted), PQC everywhere	1542	3.3%
Full Butterfly (unencrypted), PQC for the user link	587	4.6%
No Butterfly, PQC for the user link	105	6.8%
No TLS, no butterfly links—user link unencrypted	9.8	±1.5 ms

The artifacts to reproduce the results can be obtained from GitHub repository https://github.com/LUMII-Syslab/qkd-as-a-service.git (accessed on 6 January 2026).

In all tests, User 1 initiated a connection to User 2. Both users were compiled on GraalVM CE 25.0.1 using pure-Java TLS implementation from BouncyCastle and. We used our TLS injection mechanism to extend TLS with PQC algorithms from LibOQS (a shared library, compiled for Intel x86_64 Linux and ARM64 macOS). Users 1 and 2 were running on Apple M4 Pro chip, macOS Tahoe 26.1. Butterfly KDCs Aija and Brencis were running on Intel Xeon W-2235, Ubuntu 25.10. The distance between users and the KDCs was approximately 10 km, with data transmitted via traditional internet routing. In all test scenarios, User 1 sent a simple hello text message, and User 2 replied with a modified message. Each test scenario was repeated five times. Internet communication remained stable, with TCP ensuring reliable handling of any packet loss. In case of unstable internet connection, one or more butterfly connections would be terminated (e.g., due to a TCP socket timeout), causing the protocol flow to end with an exception. Both User 1 and User 2 were implemented in Java; therefore, the typical memory consumption of the Java Virtual Machine was expected and also observed (approximately 400 MB of RAM).

We observed that receiving half of the key from Aija or Brencis takes approximately 240 ms. This latency could be reduced by switching to a more efficient reverse proxy with support for PQC certificates once such support becomes available. We anticipate HAProxy, which offers best-in-class performance. However, it relies on LibSSL, which currently lacks

support for PQC signatures in TLS via OpenSSL providers (e.g., `oqsprovider`). In addition, HAProxy pre-caches PQC keys and certificates internally and then injects them into TLS, meaning that supporting PQC certificates would require substantial modifications to both the HAProxy and OpenSSL codebases. Similar issues also exist in Nginx.

We also found that mutual authentication using PQC certificates takes around 100 ms per connection (if we use SPHINCS+ with SHA2, 128-bit fast variant). Thus, by adopting JWT-based authentication for butterfly links and implicit authentication for the user link (see Section 6), we could reduce the latency by up to an additional 500 ms in total, depending on the JWT validation method and whether database lookups are required. However, this approach is technically more complex, as it requires pre-shared keys and the management of two tokens per each user.

Although we measured end-to-end link performance, the latencies reported in Table 1 scale sub-linearly, since the total latency is distributed among all four nodes involved in the Butterfly protocol. User 1, for example, can initiate multiple TLS connections in parallel. Besides, all servers (User 2, Aija, and Brencis) can benefit from parallel execution across multiple threads in multi-user scenarios. At present, the most resource-intensive component is the PQC reverse proxy pqproxy", which we hope can eventually be replaced by a PQC-enabled HAProxy capable of effectively leveraging multi-threading. (see https://www.haproxy.com/blog/how-haproxy-takes-advantage-of-multi-core-cpus, accessed on 6 January 2026) In addition, the pure-Java TLS implementation in BouncyCastle could eventually be replaced with an OpenSSL-based pure-C implementation once a comparable TLS injection mechanism is implemented in OpenSSL.

However, the most significant bottleneck of the Butterfly Protocol setup is the QKD key rate, which depends on the QKD equipment. For instance, IDQ advertises up to 14,000 AES-256 keys per hour (4 keys a second) at 24 dB attenuation for Clavis XG (although we observed better key-rates), whereas Toshiba's QKD systems can achieve key rates of several thousand keys (approximately 5000–6000) per second. In any case, key management systems typically maintain a key buffer. In addition, our Go implementation also collects QKD keys in a buffer (currently set to 50,000 keys) to prevent key starvation. Furthermore, additional (non-QKD) keys can be generated and sent over a pre-established encrypted channel between KDCs. A QKD-capable encryptor or a VPN protocol, where symmetric QKD keys are refreshed at fixed intervals, can be used for that.

Currently, we do not advice the Butterfly Protocol for production usage, since we are still working on devising a formal security proof via the Tamarin prover. That would be published in our next paper. Besides, we are working on technical protocol improvements.

12. Discussion

Our solution is novel in that we sacrifice the idea of relying on the geographically closest QKD node (which might be challenging to implement due to per-packet routing in the global Internet) in favor of the tolerance to breaking any 1 out of 6 classical links from Figure 2. Naturally, this increases the consumption of computational and network resources, as the system relies not only on a single QKD link but also on the butterfly links and the connections between KDCs. Nevertheless, it is no longer necessary for QKD devices to be directly attached to Users 1 and 2. We believe that, with optimizations, a pair of Aija and Brencis could serve several thousand TLS connections per second, bringing their performance in line with the best QKD devices, achieving comparable key rates.

Previous proposals for "software-defined" QKD networks have contributed valuable architectural insights [21], yet most stop short of offering concrete implementation strategies. Our work moves beyond theoretical frameworks by presenting a functioning integration of

QKD into hybrid communication environments, supported by real-world implementation and testing.

There have also been various efforts to bring QKD into practical use, typically by using QKD keys to replace or augment traditional AES keys in cryptographic protocols (the ”re-keying” approach) [22,23]. Solutions range from proprietary mechanisms, such as IDQ's Dual-Key agreement, to open-source adaptations of established protocols, including TLS and IPSec. These typically involve significant changes to the protocol stack or runtime key refreshment logic. In contrast, our architecture preserves most of the TLS protocol, modifying only the list of key-encapsulation mechanisms (KEMs) and the list of digital-signature algorithms in the ClientHello message. ButterflyKEM and the Butterfly-based horizontal authentication component are modules that can be integrated through the TLS Injection Mechanism as needed, keeping the overall system clean and easily extensible.

One of the most well-known proposals aimed at removing the need for large post-quantum signatures in TLS handshakes is KEMTLS [24]. By leveraging additional KEM operations, KEMTLS achieves implicit client-server authentication, albeit at the cost of altering the standard TLS handshake. While our Butterfly Protocol targets similar goals, it takes a fundamentally different approach: rather than modifying TLS itself, we define a custom Butterfly-based KEM and rely on hash-based mechanisms for implicit authentication, rather than modifying the TLS flow. KEMTLS aims to optimize the handshake by minimizing round-trips. In contrast, our design prioritizes robustness against active attacks on any individual communication link, which increases the overall number of round-trips. Nonetheless, from the end-user's perspective, the handshake in the use link remains compatible with TLS 1.3, preserving the standard three round-trips in server-only authentication scenarios.

The high cost of QKD devices has motivated the development of QKD simulators, with QKDNetSim standing out as a notable example [22] (see also: https://www.qkdnetsim.info and http://open-qkd.eu, both accessed on 6 January 2026). This aligns well with our modular approach based on a unified Go interface, which supports both real and simulated QKD environments. While our interface allows for seamless switching between device drivers and test simulators, it currently assumes preconfigured simulation setups, without offering fine-grained parameter control as found in more advanced simulation tools.

It is important to note that practical QKD deployments universally rely on an authenticated classical channel to prevent man-in-the-middle attacks. This authentication is typically performed through information-theoretic message authentication codes (e.g., Wegman–Carter MAC [25]), with fresh MAC keys regularly derived from QKD-generated key material to maintain unconditional security [26].

Regarding authentication in QKD systems, existing methods still lack standardized solutions. QKD authentication can be performed performed through information-theoretic message authentication codes (e.g., Wegman–Carter MAC), with fresh MAC keys regularly derived from QKD-generated key material to maintain unconditional security [25,26]. However, in 2021, Wang et al. proposed a PQC-based mutual authentication scheme for QKD networks that share a common certification authority, effectively replicating a PQC-enabled TLS handshake via classical channels [27].

Emerging quantum-authentication research has begun to explore the embedding of special authentication qubits within the BB84 quantum key distribution protocol itself, providing an alternative to purely classical methods and further strengthening identity verification in QKD exchanges [28].

Another recent paper by Kon et al. proposes a Quantum Authenticated Key Expansion (QAKE) protocol that merges QKD with secure client authentication in a single procedure while enabling key recycling, meaning that authentication keys can be safely reused if the

protocol succeeds [29]. They rigorously analyze its security using a classical authenticated key-exchange framework adapted for quantum settings and demonstrate a practical implementation using commercial QKD hardware over a simulated 45-km fiber, achieving feasible key rates and low error.

While we envisage a future global QKD network, we must acknowledge that its realization will be gradual. It will require many QKD segments interconnected via trusted nodes, since long-distance QKD is not yet feasible [30]. Thus, temporary solutions are needed to compensate for missing QKD links. One option is satellite-based QKD. Regretfully, ground stations are costly, and the achievable key rates are significantly lower than those of terrestrial QKD links, although satellites can span much longer distances without relying on trusted nodes. Another option is to employ our Butterfly Protocol, which can replace any missing point-to-point QKD link with classical butterfly links. Although these links do not provide perfect forward secrecy, the overall system can still tolerate the compromise of any single link.

Author Contributions: The Butterfly Protocol idea: S.K. and J.V.; conceptualization: all the authors; methodology: E.K. and E.R.; software: S.K.; validation: K.P. and E.K.; writing—original draft preparation: S.K.; writing—review and editing: all the authors; visualization: K.P. and S.K.; discussion: S.K. and E.R., supervision: J.V. All authors have read and agreed to the published version of the manuscript.

Funding: This work has been supported by the Latvian Quantum Initiative under European Union Recovery and Resilience Facility project no. 2.3.1.1.i.0/1/22/I/CFLA/001.

Data Availability Statement: The data presented in this study are openly available in [Butterfly Protocol implementation] at [https://github.com/LUMII-Syslab/qkd-as-a-service] (accessed on 6 January 2026).

Conflicts of Interest: The authors declare no conflicts of interest.

References

1. Wehner, S.; Elkouss, D.; Hanson, R. Quantum internet: A vision for the road ahead. *Science* **2018**, *362*, eaam9288. https://doi.org/10.1126/science.aam9288.
2. Kozlovičs, S.; Petručeņa, K.; Lāriņš, D.; Vīksna, J. Quantum Key Distribution as a Service and Its Injection into TLS. In *Information Security Practice and Experience*; Series Title: Lecture Notes in Computer Science; Meng, W.; Yan, Z.; Piuri, V., Eds.; Springer Nature: Singapore, 2023; Volume 14341, pp. 527–545. https://doi.org/10.1007/978-981-99-7032-2_31.
3. Viksna, J.; Kozlovics, S.; Rencis, E. POSTER: Integrating Quantum Key Distribution into Hybrid Quantum-Classical Networks. In *Applied Cryptography and Network Security Workshops*; Series Title: Lecture Notes in Computer Science; Zhou, J.; Batina, L.; Li, Z.; Lin, J.; Losiouk, E.; Majumdar, S.; Mashima, D.; Meng, W.; Picek, S.; Rahman, M.A.; et al., Eds.; Springer Nature: Cham, Switzerland, 2023; Volume 13907, pp. 695–699. https://doi.org/10.1007/978-3-031-41181-6_42.
4. Bennett, C.H.; Brassard, G. Quantum cryptography: Public key distribution and coin tossing. In Proceedings of the IEEE International Conference on Computers, Systems and Signal Processing, Bangalore, India, 9–12 December 1984; pp. 175–179.
5. Shor, P.W.; Preskill, J. Simple Proof of Security of the BB84 Quantum Key Distribution Protocol. *Phys. Rev. Lett.* **2000**, *85*, 441–444. https://doi.org/10.1103/PhysRevLett.85.441.
6. Bennett, C.H. Quantum cryptography using any two nonorthogonal states. *Phys. Rev. Lett.* **1992**, *68*, 3121–3124. https://doi.org/10.1103/PhysRevLett.68.3121.
7. Scarani, V.; Acín, A.; Ribordy, G.; Gisin, N. Quantum Cryptography Protocols Robust against Photon Number Splitting Attacks for Weak Laser Pulse Implementations. *Phys. Rev. Lett.* **2004**, *92*, 057901. https://doi.org/10.1103/PhysRevLett.92.057901.
8. Lo, H.K.; Ma, X.; Chen, K. Decoy State Quantum Key Distribution. *Phys. Rev. Lett.* **2005**, *94*, 230504. https://doi.org/10.1103/PhysRevLett.94.230504.
9. Bennett, C.H.; Brassard, G.; Mermin, N.D. Quantum cryptography without Bell's theorem. *Phys. Rev. Lett.* **1992**, *68*, 557–559. https://doi.org/10.1103/PhysRevLett.68.557.
10. Ekert, A.K. Quantum cryptography based on Bell's theorem. *Phys. Rev. Lett.* **1991**, *67*, 661–663. https://doi.org/10.1103/PhysRevLett.67.661.

11. Gao, R.Q.; Xie, Y.M.; Gu, J.; Liu, W.B.; Weng, C.X.; Li, B.H.; Yin, H.L.; Chen, Z.B. Simple security proof of coherent-one-way quantum key distribution. *Opt. Express* **2022**, *30*, 23783–23795.
12. Grosshans, F.; Grangier, P. Continuous Variable Quantum Cryptography Using Coherent States. *Phys. Rev. Lett.* **2002**, *88*, 057902. https://doi.org/10.1103/PhysRevLett.88.057902.
13. Leverrier, A.; Grangier, P. Unconditional Security Proof of Long-Distance Continuous-Variable Quantum Key Distribution with Discrete Modulation. *Phys. Rev. Lett.* **2009**, *102*, 180504. https://doi.org/10.1103/PhysRevLett.102.180504.
14. 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. Innov. Open Solut.* **2016**, *4*, 2188–2197.
15. González-Payo, J.; Trényi, R.; Wang, W.; Curty, M. Upper Security Bounds for Coherent-One-Way Quantum Key Distribution. *Phys. Rev. Lett.* **2020**, *125*, 260510. https://doi.org/10.1103/PhysRevLett.125.260510.
16. ETSI Industry Specification Group on Quantum Key Distribution. Quantum key Distribution (QKD); Protocol and Data Format of REST-Based Key Delivery API. Group Specification GS QKD 014 V1.1.1, ETSI. 2019. Available online: https://www.etsi.org/deliver/etsi_gs/QKD/001_099/014/01.01.01_60/gs_qkd014v010101p.pdf (accessed on 6 January 2026).
17. Singh, R.; Hill, C.; Kawaguchi, S.; Lupo, J. Secure Key Integration Protocol (SKIP). Internet-Draft, Work in Progress draft-cisco-skip-02, Internet Engineering Task Force. 2025. Available online: https://datatracker.ietf.org/doc/draft-cisco-skip/02/ (accessed on 6 January 2026).
18. Cao, Y.; Zhao, Y.; Wang, Q.; Zhang, J.; Ng, S.X.; Hanzo, L. The Evolution of Quantum Key Distribution Networks: On the Road to the Qinternet. *IEEE Commun. Surv. Tutor.* **2022**, *24*, 839–894. https://doi.org/10.1109/COMST.2022.3144219.
19. Housley, R.; Ford, W.; Polk, W.; Solo, D. Internet X.509 Public Key Infrastructure Certificate and CRL Profile. Technical Report RFC2459, RFC Editor. 1999. Available online: https://www.rfc-editor.org/info/rfc2459 (accessed on 6 January 2026).
20. Stebila, D.; Mosca, M. Post-quantum Key Exchange for the Internet and the Open Quantum Safe Project. In *Selected Areas in Cryptography—SAC 2016*; Avanzi, R., Heys, H., Eds.; Springer International Publishing: Cham, Switzerland, 2017; Volume 10532, pp. 14–37.
21. Aguado, A.; Lopez, V.; Lopez, D.; Peev, M.; Poppe, A.; Pastor, A.; Folgueira, J.; Martin, V. The Engineering of Software-Defined Quantum Key Distribution Networks. *IEEE Commun. Mag.* **2019**, *57*, 20–26. https://doi.org/10.1109/MCOM.2019.1800763.
22. 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.* **2017**, *16*, 253.
23. Neppach, A.; Pfaffel-Janser, C.; Wimberger, I.; Loruenser, T.; Meyenburg, M.; Szekely, A.; Wolkerstorfer, J. Key Management of Quantum Generated Keys in IPsec. In Proceedings of the 3rd International SECRYPT Conference, Porto, Portugal, 26–29 July 2008; pp. 177–183.
24. 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, Virtual, 9–13 November 2020; pp. 1461–1480.
25. Wegman, M.N.; Carter, J. New hash functions and their use in authentication and set equality. *J. Comput. Syst. Sci.* **1981**, *22*, 265–279. https://doi.org/10.1016/0022-0000(81)90033-7.
26. Bibak, K.; Kapron, B.M.; Srinivasan, V. Authentication of variable length messages in quantum key distribution. *EPJ Quantum Technol.* **2022**, *9*, 8. https://doi.org/10.1140/epjqt/s40507-022-00127-0.
27. Wang, L.J.; Zhang, K.Y.; Wang, J.Y.; Cheng, J.; Yang, Y.H.; Tang, S.B.; Yan, D.; Tang, Y.L.; Liu, Z.; Yu, Y.; et al. Experimental authentication of quantum key distribution with post-quantum cryptography. *Npj Quantum Inf.* **2021**, *7*, 67.
28. Park, H.; Park, B.K.; Woo, M.K.; Kang, M.S.; Choi, J.W.; Kang, J.S.; Yeom, Y.; Han, S.W. Mutual entity authentication of quantum key distribution network system using authentication qubits. *EPJ Quantum Technol.* **2023**, *10*, 48. https://doi.org/10.1140/epjqt/s40507-023-00205-x.
29. Kon, W.Y.; Chu, J.; Loh, K.H.Y.; Alia, O.; Amer, O.; Pistoia, M.; Chakraborty, K.; Lim, C. Quantum Authenticated Key Expansion with Key Recycling. *arXiv* **2024**, arXiv:2409.16540. https://doi.org/10.48550/arXiv.2409.16540.
30. Huttner, B.; Alléaume, R.; Diamanti, E.; Fröwis, F.; Grangier, P.; Hübel, H.; Martin, V.; Poppe, A.; Slater, J.A.; Spiller, T.; et al. Long-range QKD without trusted nodes is not possible with current technology. *Npj Quantum Inf.* **2022**, *8*, 108. https://doi.org/10.1038/s41534-022-00613-4.

Disclaimer/Publisher's Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.