An ETSI GS QKD compliant TLS implementation*

Thomas Prévost[1][0009−0000−2224−8574], Bruno Martin[1][0000−0002−0048−5197], and Olivier Alibart[2][0000−0003−4404−4067]

[1] Université Côte d'Azur, CNRS, I3S, France
{thomas.prevost,bruno.martin}@univ-cotedazur.fr
[2] Université Côte d'Azur, CNRS, InPhyNi, France
olivier.alibart@univ-cotedazur.fr

Abstract. This paper presents our implementation of the Quantum Key Distribution standard ETSI GS QKD 014 v1.1.1, which required a modification of the Rustls library. We modified the TLS protocol while maintaining backward compatibility on the client and server side. We thus wish to participate in the effort to generalize the use of Quantum Key Distribution on the Internet. Finally we used this library for a video conference call encrypted by QKD.

Keywords: TLS · Quantum Key Distribution · Rust · ETSI.

1 Introduction

The public key cryptosystems used today, such as RSA or ECC, will indeed not be able to resist the advent of the quantum computer [4,17]. It is therefore possible that a malicious actor is listening to encrypted communications, in the hope of being able to decrypt them once the quantum computer is operational. This type of attack is called "*harvest now, decrypt later*" [14].

Post-quantum cryptography offers a response to the risk posed by quantum computers. These public key cryptosystems are based on problems deemed difficult to solve by a quantum computer. However, attacks are regularly discovered on these cryptosystems [12], and it is feared that these algorithms may not stand the test of time. If public key encryption is today relevant for key transfer, we aim to replace it with Quantum Key Distribution.

Quantum Key Distribution (QKD) is a key exchange mechanism offering theoretically perfect forward secrecy. It is based on very different mechanisms. All security relies on the no-cloning theorem from quantum physics, which states that it is impossible to copy the state of a qubit without modifying it [18]. Participants exchange the key in the form of qubits, most often single photons. An attacker who attempts to intercept communications would therefore necessarily

* This work has been supported by a government grant managed by the Agence Nationale de la Recherche under the Investissement d'avenir program, reference ANR-17-EURE-004

modify the state of the qubits and could therefore be detected on the fly by the participants, who would then be able to interrupt the communication. The authenticity of messages is guaranteed by the authentication of participants, which mostly remains based on a public key cryptosystem.

QKD therefore theoretically offers perfect forward secrecy. Security is no longer based on the computational difficulty of finding the message for an adversary who has intercepted the communications, but on the on-the-fly detection of this adversary when listening to communications. In practice some attacks on QKD could occur due to physical constraints of the devices used [9].

Since Quantum Key Distribution requires complex and expensive devices, as well as a direct link between the two participants (for example a dedicated optical fiber), it is especially interesting for cross-datacenter communication, as well as for organizations such as banks or governments.

The ETSI GS QKD 014 v1.1.1 protocol [7] provides an interface standard for managing QKD keys. In a previous paper we proved the security of this standard using the ProVerif tool [5], under certain conditions [15]. We present here a concrete implementation of this standard via a modification of the TLS protocol. Our TLS implementation no longer performs the handshake thanks to public key encryption. Instead, both TLS participants make a key request to their local datacenter's QKD manager. This request is secured by classic HTTPS encapsulation with a bilateral authentication. Indeed we consider public key cryptosystems reliable in a local network. Once the quantum key is obtained, the two TLS participants encrypt their messages with a classic secret key cryptosystem.

In the rest of this paper, we will call this protocol "**TLS-QKD**". Our protocol is intended to be backward compatible in both directions, that is to say that a TLS-QKD client can connect to a classic TLS server, and a TLS-QKD server can accept a connection from a classic client. We actually implemented a version of TLS with an external Pre-Shared-Key (PSK). Other implementations of this type already exist, and have been formally proven, as described for example by the RFCs 9257 and 9258 [8,1]. We chose to design our own implementation because we started by writing the formal proof of our protocol based on ETSI GS QKD 014 v1.1.1 using ProVerif. We therefore wanted to adapt our protocol to our formal proof. We also developed QKD key management software, and implemented a video conference demonstration using the TLS-QKD protocol.

The paper is organized as follow: section 2 introduces the ETSI standard proposal for Quantum Key Distribution, section 3 presents our implementation. Finally we suggest avenues for improving and we discuss about a possible future for Quantum Key Distribution protocols.

2 ETSI GS QKD 014 v1.1.1

In this section, we briefly recall the operative mode of the ETSI GS QKD 014 v1.1.1 standard proposal.

The ETSI standard proposal is mainly focused around a REST interface, through which the different actors interact. The standard defines two types of communication:

- Communication within "secure zones" (e.g. inside a datacenter's LAN), where public-key cryptography is allowed.
- Outside communication (e.g. between datacenters), where communications have to be secured with QKD.

Two types of actors interact in the ETSI GS QKD protocol:

- **KME**: Key Management Entities, that manage keys within the datacenter's private network and exchange keys with KMEs in remote datacenters using QKD.
- **SAE**: Secure Application Entities, applications that request keys to KMEs for communication.

SAEs make requests to their datacenter's KME via a REST API, secured by HTTPS. The KME is therefore authenticated by the server certificate. To authenticate itself, the SAE presents a client TLS certificate. This certificate also uniquely identifies the SAE.

Each actor, KME and SAE, is identified by a unique identifier in the network. The keys are identified by their UUID fingerprint, which is also supposed to be unique. Fig. 1 shows an example of a key exchange between two SAEs on remote data centers.

Here is an example of a request allowing the initiating SAE ("master") to request a key from the KME of its datacenter:

https://{KME_hostname}/api/v1/keys/{slave_SAE_ID}/enc_keys

The ETSI standard defines the interface and the order of communications. It does not go into cryptographic details, e.g. how QKD is performed or key identifiers are transmitted are considered "*outside the scope of the document*".

When we formally verified the standard using ProVerif [15], we determined that the ETSI prototype standard guaranteed the confidentiality and authenticity of the key with the following constraints on the implementation:

- The connection between the two KMEs must be authenticated (this is already a prerequisite for the operation of QKD).
- The second SAE ("slave") must send a cryptographic challenge to the initiating SAE ("master"), in order to authenticate the latter by ensuring that it has the correct quantum key.
- The key UUID can be transferred in plain text between the two SAEs.
- The quantum symmetric key exchanged between the KMEs must remain secret and have high entropy.

The protocol verification done by ProVerif is available at the following URL: `https://gist.github.com/thomasarmel/c2bfc851bb3b19348bf1df90ed041fac`. There are other frameworks dedicated precisely to the verification of this kind of hybrid protocol, notably [6]. However, we chose to use ProVerif because it is well documented, and its soundness property has been widely proven.

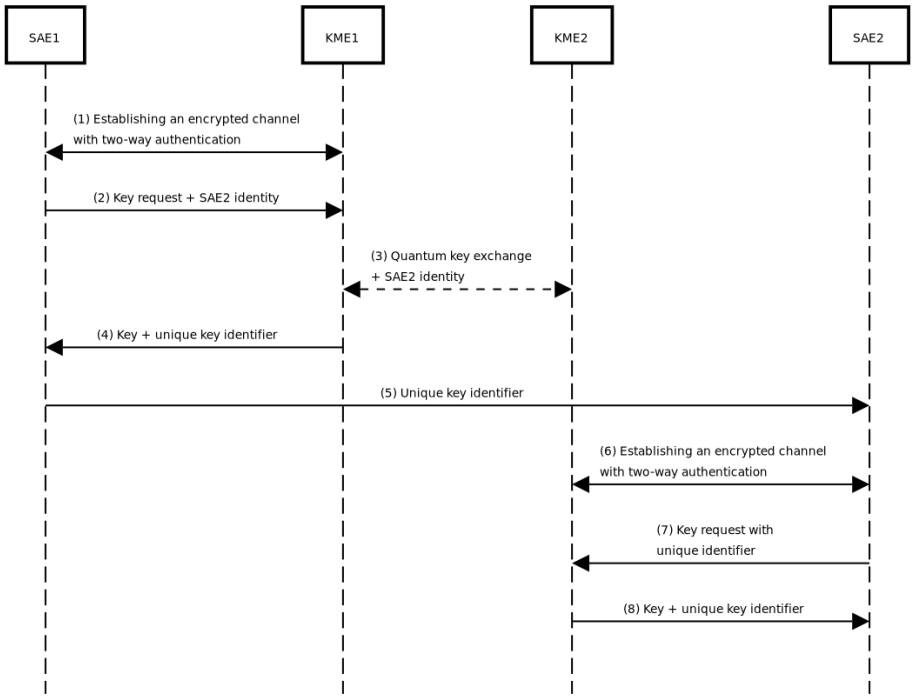


Fig. 1: This diagram shows a typical quantum key exchange between the initiator SAE 1 (“master”) and the SAE 2 ”slave”, as defined in the theoretical standard proposal. The SAE 1 makes an authenticated key request to the KME of its data center (KME 1), which will communicate the key enciphered within a TLS response to the KME of the remote data center (KME 2). SAE 1 then transfers the key identifier to its SAE 2 peer, which can then request the key from its data center’s KME.

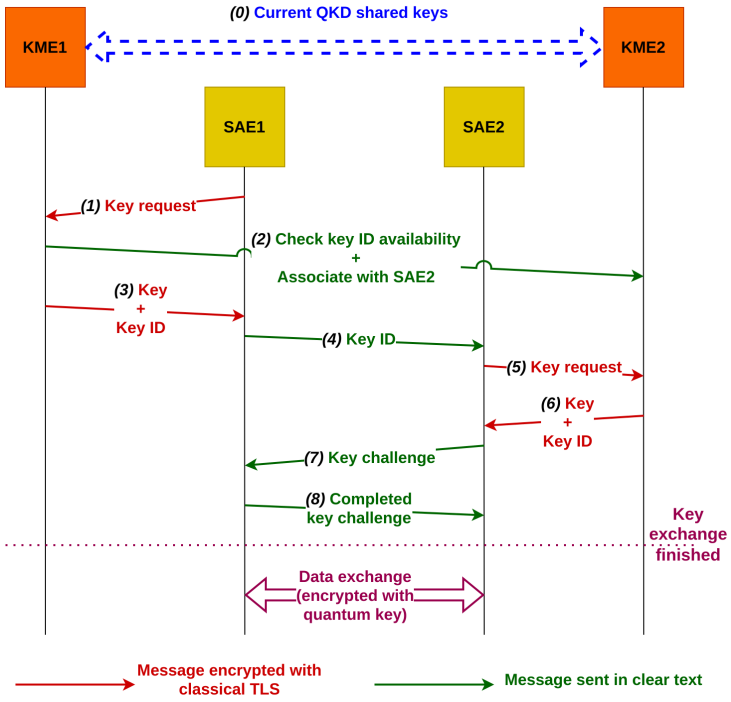


Fig. 2: This diagram shows the flow of a key exchange using the verified implementation of the ETSI protocol. This is a realistic and functional implementation, whose security has been formally verified. The SAE 2 sends a cryptographic challenge to the initiating SAE 1 to ensure its authentication. Here we assume that the KMEs have exchanged QKD keys before the start of the protocol, in order to get a latency compatible with the smooth running of an IP protocol.

Fig. 2 shows an example protocol following an implementation compatible with these security requirements.

3 Our implementation

In this section, we present our Key Management software, as well as our implementation of the modified Rustls library, and our video conferencing software based on TLS-QKD.

3.1 KME key manager

Key Management Entities (KMEs) are responsible for managing keys within the data center and exchanging keys with their remote counterparts, via QKD. The protocol used for Quantum Key Distribution during our experiments is BBM92 [3], based on the entanglement of photons pairs. Any other QKD protocol, however, would have given the same results (other protocols might have a different maximum geographic distance between remote KMEs). After receiving all the photons, the two remote KMEs share a sequence of random bits. Indeed, the same measurement of two maximally entangled photons will give the same result. It is necessary to add a Privacy Amplification (PA) step on both sides in order to extract the maximum entropy from these shared bits [2]. This step allows us to ensure the uniformity of the random distribution of the bits of the symmetric key, and therefore the security of the latter. At the end of the protocol, the two protagonists share a secret perfectly random bit string.

Our KME software takes as a parameter a folder in which the key files are located after Privacy Amplification and cuts them into sections of 32 bytes, in order to obtain 256-bit keys. If new keys are generated during operation of the KME server, the latter will detect them and add them to its database. If the KME exchanges keys with several other KMEs, it takes as parameter the folders containing the keys exchanged with each of its counterparts.

SAEs are authenticated with their client TLS certificate, and identified with the certificate serial number. Indeed, the serial number of the certificate is chosen by the Certificate Authority, and can therefore be unique within the secure zone. The KME associates the serial number with a unique identifier on the network, a 64-bit integer. In addition to the standard, we added the following REST route on the KME, in order to allow SAEs to know their identifier on the network: *https://{KME_hostname}/api/v1/sae/info/me*.

Each KME also has a unique identifier on the network in the form of a 64-bit integer. The addressing of KMEs is independent of that of SAEs, which means that a KME can have the same address as a SAE.

The UUID of the keys is generated from their SHA-1 fingerprint.

At any time SAEs can request from the KME the total Shannon entropy of the stored keys, from the route given by the REST request *https://{KME_hostname}/api/v1/keys/entropy/total*. We added this route to allow an administrator to detect a failure in Quantum Key Distribution.

In order to inform its remote counterpart of the association between a SAE and its key, the KME also uses the REST protocol, encrypted via HTTPS and authenticated on both sides, between the two KMEs. Bilateral authentication between KMEs is done with client and server TLS X.509 certificates.

The KME key manager source code can be retrieved at the following address: `https://github.com/thomasarmel/qkd_kme_server`

3.2 Our implementation of TLS with QKD keys

Our implementation of TLS with QKD keys is a modification of the Rustls library. We chose to modify this implementation for the following reasons:

- The excellent code quality of the library simplified our development work.
- Security offered by the Rust programming language reduces the probability that our implementation is affected by certain classic security vulnerabilities, such as memory corruption.
- The library being Rust-native, it compiles without problem on most platforms; it is also easy to generate a static binary, portability is therefore greatly facilitated.

Our version of Rustls is designed to be backward compatible in both directions. Thus, a TLS-QKD client can connect to a classic TLS server, and a TLS-QKD server can receive connections from a classic TLS client. We could then fear that a malicious actor having intercepted the communications could carry out a "**downgrade attack**", that is to say force the protagonists to use classic TLS to weaken the protocol [13]. We are fully aware of this vulnerability, and believe this is an acceptable compromise at this time to facilitate adoption of the protocol. However, the user who needs strict QKD protection could easily disable TLS 1.3 backward compatibility in our implementation.

Our implementation of TLS-QKD is a modification of the TLS 1.3 protocol [16], which is the latest version of the protocol as of this writing. This version of TLS is much faster than the previous one, TLS 1.2. Indeed, TLS 1.3 handshake only requires two messages: the client sends a *ClientHello*, which also contains its ephemeral Diffie-Hellman KeyShare. The server then responds with a *ServerHello*, containing its certificate as well as its ephemeral Diffie-Hellman KeyShare. The client and the server then have enough information to establish an encrypted and authenticated communication channel. In comparison, TLS 1.2 handshake requires 4 messages to complete.

Here are the changes we made to the protocol:

Client and server configuration interface. The TLS client and server are two SAEs in the ETSI protocol. Client is the initiating ("master") SAE. They must collect the keys from the KME of their respective data center. The client and the server take the address and port number of the KME interface as parameters. SAEs authenticate with KMEs using client TLS certificates. The configuration therefore takes as parameter the path of a **.pfx** client certificate file,

as well as the password allowing this file to be unlocked. Files with .pfx extension are typically used to store X.509 certificates, and can be encrypted using a password. In addition, the client takes as parameter the unique identifier of the server, in the form of a 64-bit integer. This identifier will be used to request the key from KME.

Protocol version. TLS messages contain the protocol version number in two bytes. For example, the code associated with TLS version 1.3 is 0x0304. For our TLS-QKD implementation, we arbitrarily chose the number **0x0E00**.

Client request to KME. Equipped with its client TLS certificate and the SAE identifier of the TLS server, the client can make a request to the KME of its "secure zone" to request a key allowing it to communicate with the remote SAE. The remote SAE is identified by its unique identifier, a 64-bit integer. This number is specified by the programmer when establishing the connection with the KME. The KME then returns the key in base64 format as well as the UUID of this key: *https://{KME_hostname}/api/v1/keys/{slave_SAE_ID}/enc_keys*.

The TLS clients will also request their SAE identifier from their KME: *https://{KME_hostname}/api/v1/sae/info/me*.

ClientHello extension. The TLS client communicates to the server its SAE identifier as well as the UUID of the key via an extension of the ClientHello message. We add to the extension the Initialization Vector (IV) which will subsequently be used for secret key encryption (we could also have generated the IV with a key derivation function like PBKDF2 [11]). This data is binary encoded in this order in the ClientHello extension. Each type of extension is associated with a 2-byte number. We arbitrarily chose the number **0xFEA6** for this ClientHello extension. If the TLS server detects this extension in ClientHello, it will then be able to determine that the client supports TLS-QKD.

Server request to KME. The TLS server having detected that the client wishes to communicate using TLS-QKD, it makes in turn a request to the KME of its 'secure-zone", to ask for the key associated with the UUID and the identifier of the initiating SAE "master", received in ClientHello. If the response from the KME is positive, the TLS client and server then share a secret key. However, it remains to correctly authenticate the initiating SAE, which will be done later by a cryptographic challenge.

ServerHello extension. In order to authenticate the client, the TLS server must ensure that the latter is in possession of the quantum key. To do this, it will send him a cryptographic challenge, in the form of a 256 bits random token and a 256 bits random seed, encrypted with the quantum key. The TLS client must send back the same token as well as a **different** random seed, encrypted with the

same quantum key. The challenge is inserted as an extension in the ServerHello response. The 2-byte number we arbitrarily chose for this ServerHello extension is **0xFEA7**. By finding this extension in the ServerHello response, the client will have confirmation that the server supports TLS-QKD.

Client challenge acknowledgment. After having confirmed that the server supports TLS-QKD, the client must now send the cryptographic challenge back to the server in order to authenticate. After decrypting the ServerHello challenge, the client encrypts it again after changing the random seed. It sends the response to the challenge in the form of a new message type, **ChallengeAck**. TLS provides a one-byte code for each message type. For example, ApplicationData messages have the code 0x17. For ChallengeAck messages, we arbitrarily chose the code **0x50**. Once the acknowledgment has been verified by the server, both participants directly start the data transfer using the quantum symmetric key.

Checking the TLS server certificate. The client no longer checks the server's TLS certificate, since the TLS-QKD protocol is sufficient to guarantee its authentication provided that the security assumptions on the KMEs are respected, as proven by ProVerif in [15].

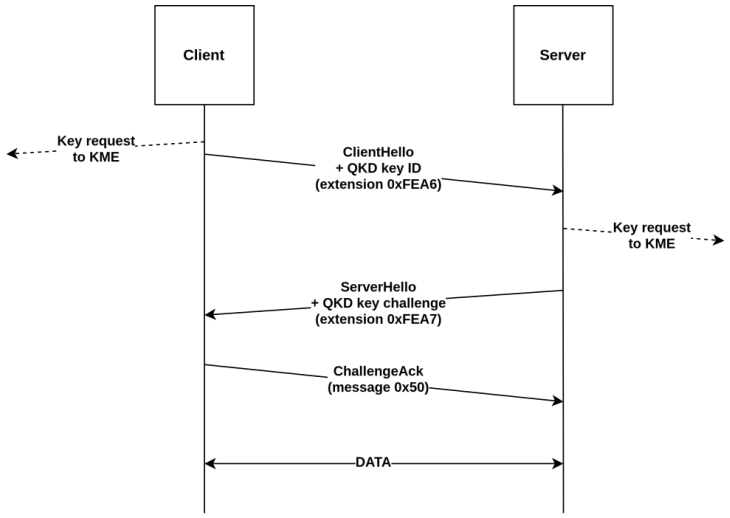
Symmetric encryption. For symmetric encryption, we use AES with the Authenticated Encryption with Associated Data (AEAD) mode. This is not, however, a security necessity, as the authentication is already guaranteed by the protocol, as proven in [15]. The key size is arbitrarily fixed to 256 bits, a standard to be quantum safe. The key size is hardcoded in our implementation. Keys are never regenerated in this implementation, we are considering this feature in future work.

Handshake authentication As proven with ProVerif, the two participants are authenticated as long as it is ensured that they both have the same symmetrical quantum key. It is therefore no longer necessary to send a Finished message to authenticate the handshake.

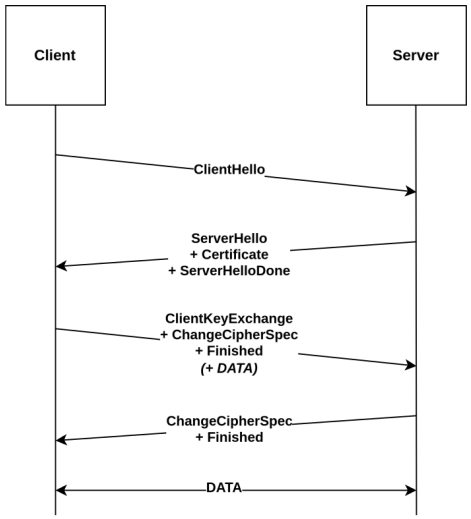
Implementation. Implementation can be found at: `https://github.com/thomasarmel/rustls/tree/qkd`. Fig. 3 shows the difference between a classic TLS 1.3 handshake and a TLS-QKD handshake.

3.3 Proof of concept: a video conference call encrypted by TLS-QKD

In order to provide a proof of concept of our protocol, we created videoconferencing software encrypted with TLS-QKD. We created our own videoconferencing



(a) Handshake on TLS-QKD.



(b) Handshake on classic TLS 1.3.

Fig. 3: Comparison of handshakes on TLS-QKD and TLS 1.3.

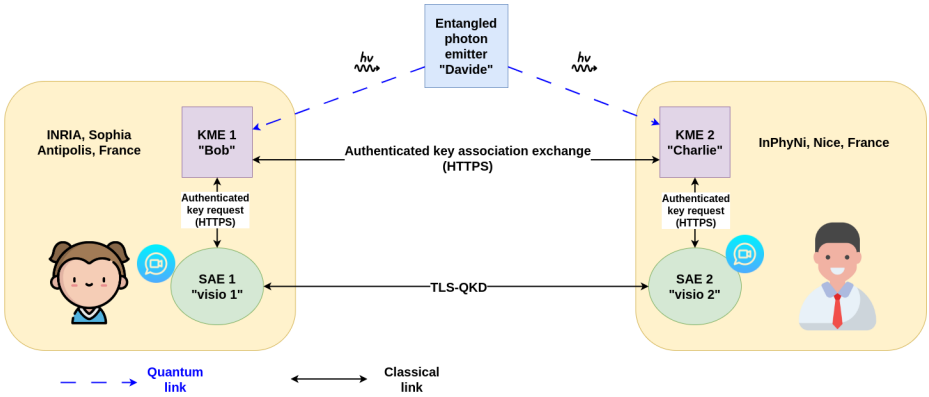


Fig. 4: Our setup for videoconferencing with TLS-QKD between the INRIA center in Sophia-Antipolis (France) and the InPhyNi physics department site in Nice (France). The two sites are separated by a distance of approximately 25 kilometers.

software because it was simpler than reusing existing code, especially because Rust is very suitable for developing this kind of applications. This software is separated into two parts:

- A server, which displays the video stream and plays the audio part.
- A client, which captures the video stream from the camera and the audio from the microphone.

To make a videoconference call, it is therefore needed to first launch the server on both machines, then the client. Code can be found at the following address: `https://github.com/thomasarmel/qkd_camera_streaming_client`.

During our tests between the INRIA center, in Sophia Antipolis (France) and the InPhyNi site, in Nice (France), we managed to set up a videoconference with a resolution of 720 pixels and 10 fps. The conversation was absolutely not hampered by sound latency. TLS-QKD typically took less than 1 second to perform the key handshake. Fig. 4 gives the network topology during our experiment.

In order to test the operation of backward compatibility towards classic TLS, we also launched requests in HTTPS, in the following configurations:

- TLS-QKD client to TLS-QKD server
- TLS-QKD client to classic TLS server (`https://fr.wikipedia.org`)
- Classic client (*curl command line*) to TLS-QKD server

The code corresponding to these tests can be found in our unit tests, available at `https://github.com/thomasarmel/rustls/blob/qkd/rustls/tests/qkd.rs`.

4 Discussion

Recent attacks on post-quantum public key cryptosystems raise the question of the long-term security of asymmetric encryption. Our protocol offers a solution against "harvest now-decrypt later" attacks if the attacker is able to listen to communications on the outside network. Our protocol remains vulnerable if the attacker is able to break QKD authentication between KMEs on the fly, since she will be able to carry out a Man-In-The-Middle attack. However, this type of scenario seems unlikely to us at the moment. Indeed, if we still use classic public key or post quantum cryptography for inter-KMEs authentication (for QKD and key requests), it is very unlikely that an attacker would have been able to secretly develop a quantum computer capable of breaking such a cryptosystem in a short time.

In general, in the case where we wish to offer perfect forward-secrecy, it seems relevant to us to envisage a world where public key cryptography no longer offers long-term forward secrecy, as has been considered until now.

Finally, our protocol relies on QKD for key sharing between the two participants, but any solution allowing a random secret to be established between the two remote KMEs is possible.

5 Further improvements

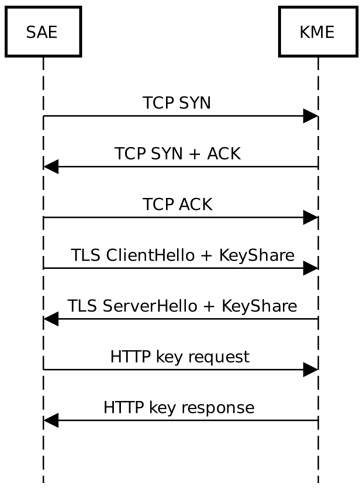


Fig. 5: All messages exchanged between SAE and KME for the first key request stage.

The main problem with TLS-QKD is its relative slowness at the handshake stage, i.e. to exchange the quantum symmetric key. In fact, each SAE must start

by establishing a secure connection with its KME. As shown in Fig. 5, a total of 7 messages is exchanged between the "master" SAE and KME at the time of the key request.

A total of 29 messages is exchanged between the different actors (SAEs and KMEs) during handshake. One way to reduce the number of messages would be to pre-establish a TLS connection between the SAEs and the KMEs. However, this would make our library much less portable, since programs running on SAEs would have to communicate with a background service responsible for keeping the connection with the KME active. Since we are targeting data center use, this compromise could be acceptable.

Another solution would be to rely on the QUIC protocol [10], which uses UDP instead of TCP. Since SAE-KME communications operate over a LAN, packet loss should not be too much of a problem. This solution would at least reduce latency in communications between SAEs and KMEs.

It would further be possible to pre-establish TLS connections in advance between KMEs. This would work at least as long as the overall network of KMEs is not too large. If the network size becomes too large, we could ensure that only the most frequent inter-KME links pre-establish the TLS connection in advance.

In this implementation, we never regenerate the symmetric key. Adding this feature would increase the security of our protocol. The property of forward secrecy is in fact not assured if an attacker were to discover the quantum key, which is more vulnerable because it is shared between four actors (the two KMEs and the two SAEs).

In our current implementation, the TLS client (initiating SAE) is required to know the TLS server SAE identifier in advance. To avoid managing a directory of correspondence between nodes and their identifiers, we could consider deriving unique identifiers from network addresses. If the IPv6 standard were to be widely adopted, then we could use it as a unique identifier, since IPv6 addresses are the same in the LAN and the WAN.

We might consider a protocol change to use Kerberos rather than bilateral TLS certificates for communication between SAE and KME. The security of communications on a local network would no longer be ensured by a public key cryptosystem but by a pre-established secret, for example a password. This would require setting up a Key Distribution Center (KDC), but it is likely that this service is already set up in most data centers. This implementation would however not be compliant with the ETSI GS QKD 014 v1.1.1 standard.

6 Conclusion

In this paper, we presented a modified TLS protocol which uses keys exchanged by Quantum Key Distribution (QKD), compliant with the standard proposal ETSI GS QKD 014 v1.1.1.

This protocol is backward compatible in both directions, meaning that a TLS-QKD client can connect to a classic TLS server, and a TLS-QKD server

can accept a classic TLS connection. We have deliberately chosen to leave this backward compatibility despite the risk of "downgrade attack", in order to facilitate a potential adoption of our protocol. However, backward compatibility can easily be disabled in the future.

The protocol is based on TLS 1.3, but adds additional configuration for communication with Key Management Entities (KME). The information necessary for the protocol to run is sent in extensions that we added to the ClientHello and ServerHello messages. Additionally, another message is sent by the client at the end of the handshake to confirm their identity, ChallengeAck.

Finally, we showed that our protocol is usable in real application cases, such as videoconferencing. However, the time required for the handshake remains significantly longer than a classic TLS handshake, since many more messages are sent and that the application spends a lot of time waiting for the KMEs stack to return the symmetric keys.

References

1. Benjamin, D., Wood, C.: RFC 9258: Importing external pre-shared keys (PSKs) for TLS 1.3 (2022)
2. Bennett, C.H., Bessette, F., Brassard, G., Salvail, L., Smolin, J.: Experimental quantum cryptography. Journal of cryptology **5** (1992)
3. Bennett, C.H., Brassard, G., Mermin, N.D.: Quantum cryptography without Bell's theorem. Physical review letters **68**(5) (1992)
4. Bhatia, V., Ramkumar, K.: An efficient quantum computing technique for cracking RSA using Shor's algorithm. In: 2020 IEEE 5th international conference on computing communication and automation (ICCCA). IEEE (2020). https://doi.org/10.1109/ICCCA49541.2020.9250806
5. Blanchet, B., Smyth, B., Cheval, V., Sylvestre, M.: Proverif 2.00: automatic cryptographic protocol verifier, user manual and tutorial. Version from (2018)
6. Dowling, B., Hansen, T.B., Paterson, K.G.: Many a mickle makes a muckle: A framework for provably quantum-secure hybrid key exchange. In: International Conference on Post-Quantum Cryptography. Springer (2020)
7. ETSI, G.: 014. Quantum Key Distribution (QKD); protocol and data format of REST-based key delivery API (2019)
8. Housley, R., Hoyland, J., Sethi, M., Wood, C.: RFC 9257: Guidance for external pre-shared key (psk) usage in TLS (2022)
9. Huang, A., Navarrete, Á., Sun, S.H., Chaiwongkhot, P., Curty, M., Makarov, V.: Laser-seeding attack in quantum key distribution. Physical Review Applied **12**(6) (2019)
10. Iyengar, J., Thomson, M., et al.: QUIC: A UDP-based multiplexed and secure transport. In: RFC 9000. Internet Engineering Task Force (IETF) Fremont, CA, USA (2021)
11. Kaliski, B.: Password-based cryptography specification. RFC 2898 (2000)
12. Kaluderovic, N.: Attacks on some post-quantum cryptographic protocols: The case of the Legendre PRF and SIKE. Tech. rep., EPFL (2022). https://doi.org/10.5075/epfl-thesis-8974
13. Lei Zhang, H.: Three attacks in SSL protocol and their solutions. Internet: https://www. cs. auckland. ac.

nz/courses/compsci725s2c/archive/termpapers/725zhang. pdf [June, 2014] (2014)
14. Paul, S.: On the transition to post-quantum cryptography in the industrial Internet of things (2022)
15. Prévost, T., Martin, B., Alibart, O.: Formal verification of the ETSI proposal on a standard QKD protocol. GTMFS (2024)
16. Rescorla, E.: The transport layer security (TLS) protocol version 1.3. Tech. rep. (2018)
17. Wohlwend, J.: Elliptic curve cryptography: Pre and post quantum. `http://math.mit.edu/~apost/courses/18.204-2016/18.204_Jeremy_Wohlwend_final_paper.pdf` (2016)
18. Zygelman, B., Zygelman, B.: No-cloning theorem, quantum teleportation and spooky correlations. A First Introduction to Quantum Computing and Information (2018)