An ETSI GS QKD compliant TLS implementation[a]

Thomas Prévost[1] [b], Bruno Martin[1] [c] and Olivier Alibart[2] [d]

[1]I3S, Université Côte d'Azur, CNRS, Sophia-Antipolis, France

[2]InPhyNi, Université Côte d'Azur, CNRS, Nice, France

{thomas.prevost, bruno.martin, olivier.alibart}@univ-cotedazur.fr

Abstract: A modification of the TLS protocol is presented, using our implementation of the Quantum Key Distribution (QKD) standard ETSI GS QKD 014 v1.1.1. We rely on the Rustls library for this. The TLS protocol is modified while maintaining backward compatibility on the client and server side. We thus wish to participate in the effort to generalize the use of QKD on the Internet. We used our protocol for a video conference call encrypted by QKD. Finally, we analyze the performance of our protocol, comparing the time needed to establish a handshake to that of TLS 1.3.

1 INTRODUCTION

Quantum computers threaten current public key cryptosystems like RSA and ECC, which are expected to be broken once such machines are operational (Bhatia and Ramkumar, 2020). This has prompted concerns about "*harvest now, decrypt later*" attacks, where adversaries store encrypted data to decrypt in the future (Paul, 2022).

Post-quantum cryptography offers alternatives based on quantum-resistant problems, but new attacks continue to emerge (Kaluderovic, 2022), raising doubts about their long-term viability. While PKC is still standard for key exchange, we propose replacing it with Quantum Key Distribution (QKD).

QKD enables theoretically perfect forward secrecy by using quantum principles—specifically the no-cloning theorem— to detect eavesdropping in real time (Zygelman, 2018). Keys are exchanged using qubits (typically single photons), and any interception alters their state, alerting the participants. Authentication remains reliant on classical PKC.

Though offering strong security guarantees, QKD faces practical challenges. Device imperfections may allow attacks (Huang et al., 2019), and the need for dedicated infrastructure limits its scalability. It's best suited for high-security environments like inter-datacenter links or governmental networks.

Due to fiber loss and the no-cloning theorem, QKD is limited to a few hundred kilometers (Huttner et al., 2022). Multipath QKD protocols address this with trusted intermediaries (Liu et al., 2024; Prévost et al., 2025). ETSI GS QKD 014 v1.1.1 defines a standard interface for managing QKD keys (ETSI, 2019), which we previously verified with ProVerif under specific assumptions (Prévost et al., 2024).

We present a practical implementation of this standard by integrating QKD into TLS. Our "*TLS-QKD*" protocol replaces the handshake's public key exchange with a request to a local QKD manager, secured via HTTPS with bilateral authentication. The ETSI standard assumes local networks can safely use classical public key cryptosystems. Once a quantum key is received, symmetric encryption ensures message confidentiality.

TLS-QKD is fully backward compatible: it can interoperate with standard TLS clients and servers. We designed the implementation to align with our formal proof and developed supporting QKD key management tools, including a video conferencing demo using TLS-QKD.

Related work

The use of pre-shared keys (PSK) in TLS is already described in RFCs 9257 and 9258 (Housley et al., 2022; Benjamin and Wood, 2022). However, we do not rely on this standard, but on a custom protocol that has been formally verified (Prévost et al., 2024)..

[b] https://orcid.org/0009-0000-2224-8574

[c] https://orcid.org/0000-0002-0048-5197

[d] https://orcid.org/0000-0003-4404-4067

(Tankovic et al., 2024a) tested an implementation of the ETSI GS QKD 014 standard in real conditions, to measure latency and transmission rate, as well as a complete analysis of the standard's performance in a multi-user environment (Tankovic et al., 2024b), and a simulation of the protocol's use cases, with an estimation of possible key rates depending on the location of the sites (Dervisevic et al., 2024).

(Martin et al., 2024a) also proposes an implementation of the ETSI standard on the Madrid QKD network. As many countries develop their QKD network independently, we cannot exhaustively cite all the implementations of the ETSI standard. The final goal would be to set up large QKD networks, for example on the scale of the European continent (Martin et al., 2024b).

Interestingly, the authors of (Buruaga et al., 2025), who wrote their paper at the same time as us, propose a modification of the TLS protocol based on the ETSI GS QKD 004 standard[1].

The paper is organized as follow: section 2 introduces the ETSI standard proposal for QKD. Section 3 presents our implementation, explaining the operation mode of our protocol and the changes to TLS. We evaluate the performance tests performed on our protocol, measuring the time for the handshake with TLS-QKD, compared to that of TLS 1.3. 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 which 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 LAN), where PKC is allowed, and outside communication (e.g. over a WAN), where communications require QKD.

Two types of actors interact in the this protocol: **KME**: Key Management Entities managing keys within the LAN and exchanging keys with remote KMEs using QKD, and **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 there-

---

[1]https://www.etsi.org/deliver/etsi_gs/QKD/001_099/004/02.01.01_60/gs_qkd004v020101p.pdf

Figure 1: This diagram shows a typical quantum key exchange between the initiator SAE 1 ("master") and the SAE 2 "slave", as defined in the standard proposal. SAE 1 makes an authenticated key request to its local KME (KME 1), which communicates the key enciphered within a TLS response to the remote KME (KME 2). SAE 1 then transfers the key identifier to its SAE 2 peer, which requests the key from its local KME.

Figure 2: Flow of a key exchange using the verified implementation of the ETSI protocol.

fore 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 a key exchange between two SAEs on remote data centers.

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 (Prévost et al., 2024), 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") to authenticate the latter by ensuring that it owns the correct quantum key.

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

3 OUR IMPLEMENTATION

Let us present our Key Management software, as well as our implementation of the custom 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 by QKD. For our experiments, we used BBM92 (Bennett et al., 1992b), based on the entanglement of photons pairs. Any other QKD protocol, however, would have given similar results. After receiving all the photons, the two remote KMEs share a sequence of random bits. It is necessary to add a Privacy Amplification (PA) step on both sides in order to extract the maximum entropy from these shared bits (Bennett et al., 1992a). This step ensures the uniformity of the random bits of the symmetric key, and therefore its security. At the end of the protocol, the 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 (the keys). If new keys are generated during operation of the KME server, the latter will 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.

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.

We added this route to allow an administrator to detect a failure or an attack in QKD.

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 X.509 certificates. Note that we use a regular HTTPS protocol to notify the KME of the use of a key, to protect against a "Harvest now, decrypt later" attacker. So we assume that *at the time of the exchange*, the attacker is not able to break the public key encryption.

The KME key manager source code can be retrieved at: 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. 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 classical TLS connections. We could then fear that a malicious actor could carry out a "*downgrade attack*" to force the protagonists to use classic TLS to weaken the protocol.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.

Our implementation of TLS-QKD is a modification of the TLS 1.3 protocol (Rescorla, 2018). Here are the changes we made to the protocol:

3.2.1 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 their local KME. 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.

3.2.2 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**.

3.2.3 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.

The TLS clients will also request their SAE identifier from their KME:

`https://{KME_hostname}/api/v1/sae/info/me`

3.2.4 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 (the IV could be the output of a key derivation function like PBKDF2). Each type of extension is associated with a 2-byte number. We arbitrarily chose **0xFEA6** for this ClientHello extension. If the TLS server detects this extension in ClientHello, it determines that the client supports TLS-QKD.

3.2.5 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.

3.2.6 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.

3.2.7 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.

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 (Prévost et al., 2024). It is also no longer necessary to send a Finished message to authenticate the handshake.

3.2.8 Symmetric encryption.

For symmetric encryption, we use AES-AEAD. This is not, however, a security necessity, as authentication is already guaranteed by the protocol. 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.

3.2.9 Implementation.

Fig. 3 shows a TLS-QKD handshake. Code is available at: https://github.com/thomasarmel/rustls/tree/qkd.

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. This software is separated into two parts: a server managing the audio and video display, and a client, which captures the video stream from the

Figure 3: Handshake on TLS-QKD

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 is available at: 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) at 25km distance, 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.

We also tested the operation of backward compatibility towards classic TLS, the code is available at https://github.com/thomasarmel/rustls/blob/qkd/rustls/tests/qkd.rs.

3.4   Protocol performance

We tested the performance of our protocol on the same sites that were used during the videoconference, 25 kilometers apart. These two sites are connected by a direct optical fiber with a particularly low latency (approximately 1 ms).

We performed 10 measurements of the time required to perform a handshake between the two SAEs with our protocol (on the "TLS-QKD" classical link), it takes on average 38.2 ms, with a minimum of 25 ms and a maximum of 66 ms. For comparison, we performed 10 measurements of the latency time of a handshake in TLS 1.3, on the same network: the average delay is 21.5 ms, with a minimum of 17 ms and a maximum of 28 ms. The code used to test the handshake duration is available at https://github.com/thomasarmel/test_https_qkd.

It should be noted, however, that our network is particularly favorable to our protocol, which requires a lot of message exchanges during the handshake, because the ping latency between the two sites is about 1 ms. A network with higher latency would show a much larger gap in delay between the handshake of our protocol and that of TLS 1.3.

4   FURTHER IMPROVEMENTS

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

A total of 29 messages is exchanged between the different actors (SAEs and KMEs) during handshake, including TCP SYN and ACK messages. 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, 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.

5   CONCLUSION

In this paper, we presented a modified TLS protocol which uses keys exchanged by QKD, compliant with the ETSI GS QKD 014 v1.1.1 standard proposal. Our protocol offers a solution against "harvest now-

decrypt later" attacks.

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 MITM attack. However, this type of scenario seems unlikely today. Indeed, if we still use PKC or PQC for inter-KMEs authentication (for QKD and key requests), it is very unlikely that an attacker would have a quantum computer capable of breaking such a cryptosystem in a short time.

It is backward compatible in both directions with TLS. We have deliberately chosen to leave this backward compatibility despite the risk of "downgrade attack", in order to facilitate a potential adoption. 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

Benjamin, D. and Wood, C. (2022). RFC 9258: Importing external pre-shared keys (PSKs) for TLS 1.3.

Bennett, C., Bessette, F., Brassard, G., Salvail, L., and Smolin, J. (1992a). Experimental quantum cryptography. *Journal of cryptology*.

Bennett, C., Brassard, G., and Mermin, N. (1992b). Quantum cryptography without Bell's theorem. *Physical review letters*.

Bhatia, V. and Ramkumar, K. (2020). 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.

Buruaga, J. S., Méndez, R. B., Brito, J. P., and Martin, V. (2025). Quantum-safe integration of TLS in SDN networks. *arXiv preprint arXiv:2502.17202*.

Dervisevic, E., Fehrić, M., Pivac, D., Fazio, P., Voznak, M., and Mehic, M. (2024). Simulations of selected quantum key distribution network use-cases. In *2024 International Conference on Quantum Communications, Networking, and Computing (QCNC)*. IEEE.

ETSI, G. (2019). 014. Quantum Key Distribution (QKD); protocol and data format of REST-based key delivery API.

Housley, R., Hoyland, J., Sethi, M., and Wood, C. (2022). RFC 9257: Guidance for external pre-shared key (PSK) usage in TLS.

Huang, A., Navarrete, Á., Sun, S., Chaiwongkhot, P., Curty, M., and Makarov, V. (2019). Laser-seeding attack in quantum key distribution. *Physical Review Applied*.

Huttner, B., Alléaume, R., Diamanti, E., Fröwis, F., Grangier, P., Hübel, H., Martin, V., Poppe, A., Slater, J. A., Spiller, T., et al. (2022). Long-range QKD without trusted nodes is not possible with current technology. *npj Quantum Information*.

Kaluderovic, N. (2022). Attacks on some post-quantum cryptographic protocols: The case of the Legendre PRF and SIKE. Technical report, EPFL.

Liu, C., Che, X., Xie, J., and Dong, Y. (2024). A multi-path QKD algorithm with multiple segments. *Journal of Cyber Security and Mobility*.

Martin, V., Brito, J. P., Ortíz, L., Mendez, R., Buruaga, J., Vicente, R., Sebastian-Lombrana, A., Rincon, D., Perez, F., Sanchez, C., et al. (2024a). Madqci: a heterogeneous and scalable SDN-QKD network deployed in production facilities. *npj Quantum Information*.

Martin, V., Peev, M., Brito, J., Ortíz, L., Brito-Méndez, R., Vicente, R., Saez-Buruaga, J., Sebastian-Lombraña, A., García-Cid, M., Faba, J., et al. (2024b). Towards large scale QKD networks. In *2024 24th International Conference on Transparent Optical Networks (ICTON)*. IEEE.

Paul, S. (2022). On the transition to post-quantum cryptography in the industrial Internet of things.

Prévost, T., Martin, B., and Alibart, O. (2024). Formal verification of the ETSI proposal on a standard QKD protocol. *GTMFS*.

Prévost, T., Martin, B., and Alibart, O. (2025). A secret key spreading protocol for extending ETSI quantum key distribution. In *Proceedings of the 11th International Conference on Information Systems Security and Privacy - Volume 2: ICISSP*. INSTICC, SciTePress.

Rescorla, E. (2018). The transport layer security (TLS) protocol version 1.3. Technical report.

Tankovic, A., Burdiak, P., Dervisevic, E., Voznak, M., Mehic, M., and Kaljic, E. (2024a). Performance analysis of ETSI GS QKD 014 protocol in 5G/6G networks. In *Future of Information and Communication Conference*. Springer.

Tankovic, A., Dervisevic, E., Voznak, M., Mehic, M., and Kaljic, E. (2024b). Performance analysis of ETSI GS QKD 014 protocol in multi-user environment. In *2024 23rd International Symposium INFOTEH-JAHORINA (INFOTEH)*. IEEE.

Zygelman, B. (2018). No-cloning theorem, quantum teleportation and spooky correlations. *A First Introduction to Quantum Computing and Information*.