
# 31-08-2023
# Deliverable D7.1
# System specification of the encryptor software

| | |
|---|---|
| Contractual Date: | 31-08-2023 |
| Actual Date: | 31-08-2023 |
| Grant Agreement No.: | 101081247 |
| Work Package: | WP7 |
| Task Item: | Task 7.1 |
| Nature of Deliverable: | R (Report) |
| Dissemination Level: | PU (Public) |
| Lead Partner: | Eötvös Loránd University (ELTE) |
| Authors: | Ottó Hanyecz (ELTE); Gergely Zsolt Kovács (ELTE); Tamás Kozsik (ELTE); Dániel Lukács (ELTE); Péter Ligeti (ELTE); Ádám Nagy (ELTE) |

## Abstract

The QCIHungary project aims to establish a national quantum communication infrastructure in Hungary as part of the EuroQCI initiative. This deliverable outlines a modular Encryptor software, which combines quantum key distribution with post-quantum cryptography. The software is supposed to be a possible Secure Application Entity in a QKD network according to the ETSI GS QKD 014 specification.


KIFÜ on behalf of the QCIHungary project. The research leading to these results has received funding from the European Union's Digital Europe Programme under Grant Agreement No. 101081247 (QCIHungary).

# Table of Contents

# Table of Figures

# Executive Summary

The QCIHungary project aims to develop a modularized software product, an encryptor, using the keys distributed over a quantum communication infrastructure. This infrastructure is the primary goal of the QCIHungary and the EuroQCI projects.

Quantum key distribution (QKD) is a method of secret key exchange between two distant participants, Alice and Bob. The laws of quantum mechanics guarantee the security of the distributed keys, which makes eavesdropping detectable. This is not possible over classical communication channels. The software product presented in this deliverable is a separate software that sits on top of the QKD layer. Its purpose is to encrypt messages (and decrypt them later) with additional keys generated using post-quantum cryptographic algorithms. To enhance security, our software product establishes an authenticated (classical) channel between the sender and the receiver of the (encrypted) message.

Our encryptor software can interact with various standard security services, for example, MACSec, IPsec, TLS, etc., which sits between the QKD layer and an arbitrary application layer, see Figure 2.1. The modularized fashion of the encryptor makes it possible to extend further the list of the available services above for later use.

The modules of the encryptor work together asynchronously in a producer-consumer fashion, which keeps the complexity minimal and makes it easy to integrate existing software libraries of tools. Our encryptor consists of five modules (Figure 3.1). The central part is the Controller, which orchestrates other modules and message routing. The Encryptor module encrypts (and decrypts) messages, and the Key-Manager provides keys for the Encryptor and information of the Controller. The Tunnel and Facade modules are simply connectors to the outside world and implement the necessary interface for communication.

The Encryptor module is stateless (Figure 3.2) and consists of other modules. Based on the number of available keys in the Key-Manager module and the type of Message, it either encrypts the Message or initiates the creation of new keys via the Key-Agreement module. These new keys will be stored in one of the Vaults via the Key-Manager.

The Key-Manager is responsible for storing private and public keys and their related information. The Controller module can only request public information and receive messages from the Key-Manager. The Encryptor module can query public and private keys from the Key-Manager.

We assume that the channel between our encryptor software and the key manager of each QKD device is authenticated. We use the ETSI 014 standard [2] for requesting keys from the key managers. We have no assumptions about the channels between the message sender and receiver.

After the authentication between two encryptor software, the message is sent through an authenticated channel. Once the channel is authenticated, any (post-quantum) key exchange happens through this channel. However, the key delivery interface of ETSI 014 standard relies on other communication protocols.

In this document, we present a modularized encryptor software solution design that uses QKD keys in combination with post-quantum keys to establish secure message transfer between Alice and Bob. We describe the environment and describe each module of the encryptor. Finally, we analyze the security of each module and discuss the assumptions that we make during the development of the software.

# 1 Introduction

The QCIHungary project aims to establish the foundations of a national quantum communication infrastructure. The present deliverable is the first product of the work package WP7 ”Software stack over a quantum communication channel”, which aims to develop a software product using the keys distributed over the infrastructure.

Quantum communication is based on Quantum key distribution (QKD), which allows two participants to exchange a secret key using a quantum communication channel. The security of QKD relies on the hypothesis that the laws of quantum mechanics bind the information that an eavesdropper might acquire on the key. The eavesdropper must interact with the quantum system to learn something about the key. This will inevitably disturb the quantum states that the two participants use, which can be detected.

Since the secret key is ultimately known by the two participants only, the security of the standardized classical algorithms is somewhat enhanced since it relies on the secret key. We aim to exploit this by creating a modularized software package that uses the secret key distributed by QKD. In this work, we assume that the security of a QKD protocol is sound and the communication channels are authenticated between the QKD devices and their key managers. We also assume that the keys are securely stored in the QKD devices (and in the key managers), meaning only the two participants can know the secret keys.

There are commercially available encryptor solutions for similar purposes, e.g. Adva, Thales, and Toshiba, but our software solution could expand these on three points.

1. *Adding PQ layer.* Most of the commercial solutions[1] only use the QKD keys and miss the opportunity to enhance the security provided by the recently standardized post-quantum (PQ) key exchange algorithms. In our system, the PQ key adds another encryption layer on the message in case the QKD protocol or the infrastructure gets compromised.

2. *Improved key management* Based on the used technology, there are restrictions on the rate QKD key bytes can be generated, which could create problems for some applications where the key generation should be on-demand and fast, or the demanded keys are rather long. Our solution has an independent Key Manager module that allows smart key management and simultaneous communication with various key storage standards.

3. *Flexibility* Our software is a modularized middleware, which creates a software layer where devices from different vendors and different infrastructures can be used together with minimal effort.

The encryptor software is designed in a modularized fashion. This document will define the modules, specify their roles, and describe the communication between them without the exact interface specification. The encryptor software is a middleware, so we will use the *user* and *user application* as synonyms as the outer interfaces not intended to be used by a person directly.

The goal of this deliverable is to present the environment in which the software will be used (Section 2), and the architecture (Section 3). Finally, we discuss the security analysis of the software (Section 4).

[1] There are vendors (for example Nokia) providing solutions that are combining PQC and QKD.

# 2 Environment

The goal of this work package is to develop a software system that enables applications to use standard security services enhanced with modern, QKD-based key exchange. In this section, we enlist external entities with which the developed software system interacts. We do not intend this list, nor the description of its items to be exhaustive; rather, this section is meant to illustrate the expected requirements for the developed system and to give a high-level view of the potential use cases of the system.

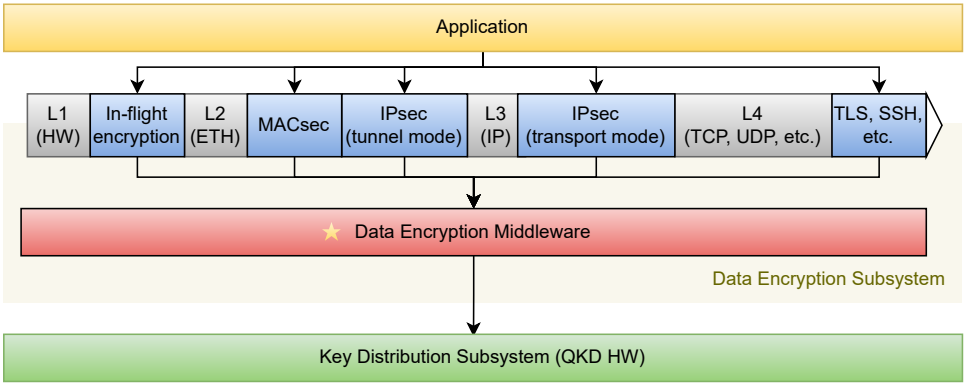


**Figure 2.1:** The system environment in which our encryptor software (Data Encryption Middleware) is located.

Figure 2.1 depicts the system environment, with the external software components (including network protocols) with which the Data Encryption Subsystem must interact. Arrows show dependencies between the components from the point of view of an application requiring a secure point-to-point communication channel.

The encryptor software – called Data Encryption Middleware (DEM) in the figure – can be considered a middleware between selected security protocols and the QKD-based Key Distribution Subsystem (KDS).

In the figure, our partitioning of the components is conceptual rather than physical. For example, a thin implementation of the Encryption Subsystem may consist strictly of the DEM, making the application responsible for instantiating the protected communication channel and the KDS driver and wiring them with the middleware. Alternatively, another Encryption Subsystem implementation may include the mechanism for instantiating and wiring together the DEM, the communication channel, and the KDS driver. We detail the proposed system architecture in Section 3.

WP2 Deliverable 2.1 lists IPsec, MACsec, and in-flight (also known as in-line) encryptor types (security protocols) that the Encryption Subsystem should provide. These protocols – described below in more detail – add security features (e.g. PDU encryption) to OSI networking layers. Their standard implementations usually use classical key exchange protocols (e.g. Diffie-Hellman protocol, in the case of IPsec). In contrast, Deliverable 2.1 specifies that their implementations in the Encryption Subsystem must use the KDS for key exchange.

WP2 Deliverable 2.1 does not expect changes in other features of the security protocols, such as encryption or authentication. On the other hand, it may be practical for the Encryption Subsystem to support or incorporate such security methods as well in order to enable secure communication over other insecure protocols (e.g. TCP).

KDS is the physical system implementing QKD and is responsible for generating and exchanging keys. DEM must be designed to support accepting keys from various drivers (adapters) of QKD hardware. The European Telecommunications Standards Institute (ETSI) has developed a standard for a key delivery application programming interface (API), the ETSI GS QKD 014 standard, for requesting a secret key from the QKD hardware. Most commercially available QKD hardware implements this standard; therefore, our software also uses it.

In the following, we describe our understanding of the encryption protocols listed in WP2 Deliverable 2.1. and propose ways to integrate them with the KDS.

## 2.1 IPsec

IPsec provides data integrity, authentication, encryption, and more for Layer 3 protocols and above [3]. IPsec can be used in two modes [4]. In transport mode, only the payload is encrypted and authenticated. In tunnel mode

(used e.g. for creating VPNs), the entire IP packet is encrypted and authenticated and then encapsulated into a new IP header. IPsec has several robust, open-source implementations, including LibreSwan, OpenSwan, and StrongSwan (all derivatives of the now-defunct FreeSwan project). IPsec allows custom key exchange methods, and we discuss this next.

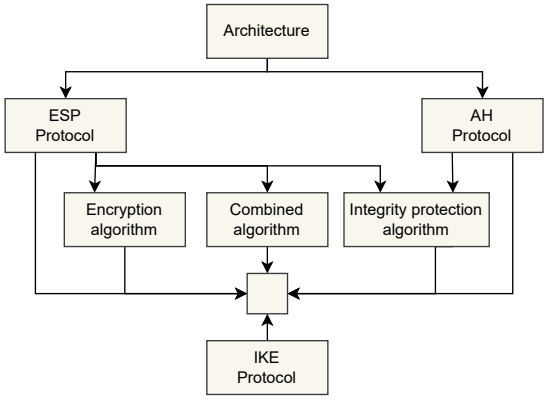


**Figure 2.2:** "Roadmap" of IPsec standards, based on RFC 6071 [1]

Figure 2.2 depicts the standards specifying IPsec and their dependencies. ESP and AH implement security features, while Key Management is responsible for key exchange mechanisms (algorithms, authentication, etc.). RFC 2408 (historic) specified a protocol for Key Management, called ISAKMP. ISAKMP allowed users to use their own key exchange protocols, although RFC 2409 (obsolete) also described the IKEv1 protocol for key exchange purposes. RFC 7296 [5] describes IKEv2, the current standard both for management of security associations and for key exchange, replacing both ISAKMP and IKEv1. IKEv2 has two phases. In the first phase, the hosts establish a secure channel using X.509 PKI certificates or PSKs to authenticate each other. In the second phase, security association management messages can be exchanged.

In this report, we do not discuss how DEM should interact with concrete implementations of IPsec, but as an example, user-generated keys can be passed to StrongSwan by setting up the `/etc/ipsec.conf` configuration file and including the PSK as plaintext in `/etc/ipsec.secret` before the secure link is initialized.

Based on this, IPsec provides at least two use cases for DEM:

1. *Manual keying.* DEM can act as the key exchange protocol for a configurable IPsec protocol (such as ISAKMP, in the case of legacy systems). In this case, the keys provided by the KDS are used for security services over IP PDUs in AH and ESP. Two separate keys should be generated: one for authentication and one for data encryption purposes.

2. *Automatic keying.* An IKEv2 daemon can act as the key exchange protocol, while DEM can provide the authentication PSKs for IKEv2. Here, the KDS-generated key is used in the first phase of IKE, namely as the PSK used in creating the secure channel. In this case, the QKD-based KDS is only used for sharing PSKs, while the actual key exchange (generating the keys that are securing the channel) is performed by IKE.

## 2.2 MACsec

MACsec provides data integrity, authentication, encryption, and more (see IEEE 802.1AE [6]) for all protocols above Layer 2 (Ethernet), including ARP and DHCP. The MACsec header wraps the payload of an Ethernet frame. Since MACsec also encrypts the IP header, it cannot operate between routers, only between hosts in the same LAN. At the time of writing, the only robust, open-source MACsec implementation is embedded in the Linux kernel.

MACsec allows custom key exchange methods, and we discuss this next.

As specified in IEEE 802.1AE, MACsec is only responsible for the above security services, while the key exchange is handled by the MACsec Key Agreement (MKA) protocol, described in IEEE 802.1X. In the MKA, the user supplies

a PSK (a pair of two keys called CAK and CKN) to both hosts. The hosts authenticate each other and share the key for encrypting the actual traffic (SAK). As part of the process, one of them generates the SAK and then uses a key exchange protocol (involving the PSK) to share this key with the other host. The MACsec security services then use this SAK.

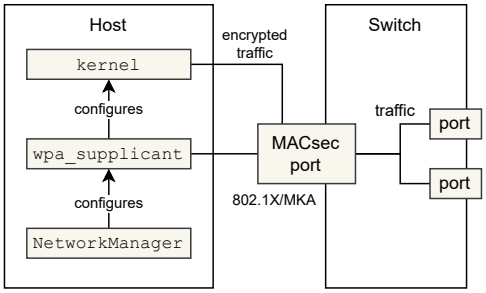


**Figure 2.3:** MACsec in Linux

Figure 2.3 depicts the MACsec architecture implemented in Linux.

MACsec (IEEE 802.1AE, without MKA) is implemented inside the Linux kernel and can be accessed using the `iproute2` tool. Using this interface, one can create a MACsec link directly and supply the encryption/decryption keys manually (e.g., using the command line). However, the manual explicitly recommends against this:

> This tool is thus mostly for debugging and testing, or in combination with a user-space application that reconfigures the keys. It is wrong to just configure the keys statically and assume them to work indefinitely. The suggested and standardized way for key management is 802.1X-2010, which is implemented by wpa_supplicant.

The deceptively named `wpa_supplicant` implements MKA (as well as authentication protocols used by high-level encryptors such as WPA2). It performs the MKA and then instructs the kernel to set up the MACsec link. The PSK (CKA and CKN) for the MKA can be provided in its configuration file (`wpa_supplicant.conf`).

Finally, most Linux distributions also include the `NetworkManager` service unit, which provides a high-level interface to create connections. This tool can also set up MACsec with MKA, using the PSK (CKA and CKN) provided by the user (e.g., on the command line, using `nmcli`). Internally, it relies on `wpa_supplicant`.

Based on this, MACsec provides at least two use cases for DEM:

1. *Direct keying.* DEM can act as the key exchange protocol for MACsec. In this case, the keys provided by the KDS are used directly for MACsec security services in the Linux kernel.

2. *MKA.* An MKA agent (e.g. `wpa_supplicant`, `NetworkManager`) can act as the key exchange protocol for MACsec while DEM can provide the authentication PSKs for MKA. Here, the KDS-generated key is used for generating SAKs and exchanging SAKs between hosts. In this case, the QKD-based KDS is only used for sharing PSKs, while the actual key exchange (generating the keys that are actually securing the channel) is performed by the MKA.

## 2.3 In-flight

In-flight encryption provides security below the L2 layer. Implementations often promise additional security features, such as protection against traffic analysis. It is common in optical networks, as these are primarily used for high-distance connections, where such security features are relevant.

In-flight encryption encrypts the Ethernet frame itself that includes network interface identifiers (MAC addresses). This means that these PDUs are usually not transferred between Ethernet-capable software or hardware devices (e.g. NICs, switches), instead, the secured link is supposed to lay between special-purpose devices handling encrypted PDUs.

As in-flight encryption is – by definition – outside the traditional network layers, there are no commonly accepted standards. Instead, each vendor provides its own proprietary solutions, which must be evaluated on a case-by-case basis.

Nonetheless, we expect similar options to those of IPsec and MACsec. For example, WaveLogic Encryption by Ciena performs first a PSK-based based ITU X.509 based authentication, which is followed by data encryption. Here, DEM may be utilized to deliver KDS-generated PSKs to the device.

In this report, we do not discuss how DEM should interact with concrete implementations of in-flight encryption. Still, it seems clear that DEM must be designed to support transmitting keys to various drivers (adapters) of in-flight encryptor hardware.

# 3 Architecture

With the emergence of quantum computers (or at least their promise) and Shor's algorithm [7], a good part of our well-established cryptographic tools become obsolete. However, we don't want to reestablish a completely new system but use solutions that do not compromise security. The relatively simple solution to accomplish this is modularization, where the architecture enables to apply widely used IT security standards and tools. Furthermore, the additional work to create interfaces pays off in flexibility, which is very important in a research project (like EuroQCI) where multiple groups work parallel on similar solutions.

## 3.1 Aim

Our goal in this section is to define the architecture of an application that can be employed on various OSI layers (based on the implemented connector modules[2]) and can work with existing solutions (like post-quantum SSL, key storage, etc.). To achieve this, we outline modules and interfaces between them where each module has a distinctive role, while the interfaces are as simple as possible. Here, the modules can be seen as active entities asynchronously working together in a producer-consumer fashion, which keeps the complexity minimal and makes easy to create envelop modules for integrating existing software libraries or tools.

## 3.2 Modules

The proposed architecture consists of several modules (see Figure 3.1) which can be organized based on their connections and roles. In the middle, the **Controller** module is responsible for orchestrating other modules and for message routing (including routing to Encryptor). The **Tunnel** and **Facade** components are the connectors and implement the interfaces that communicate with the outside. The stateless **Encryptor** encrypts messages, while the **Key-Manager** provides keys for the Encryptor and information for the Controller.

### 3.2.1 Controller

The Controller is the core module of the architecture, and when it starts it

- reads a configurational object and, based on that
- starts the other modules[3],
- checks the state of the system and
- commences to route the messages between the communication modules and the Encryptor.

*Note:* The Controller is a module and not provides a regular application interface; instead, the module itself should be used or enveloped in an actual application.

[2] In the case of the Application Layer, the software can behave like a VPN client, providing a socket or network interface as a network tunnel.

[3] The Controller starts the listed submodules like Vaults or Authonticators too. These modules are passed to their logical parent during the initialization process.

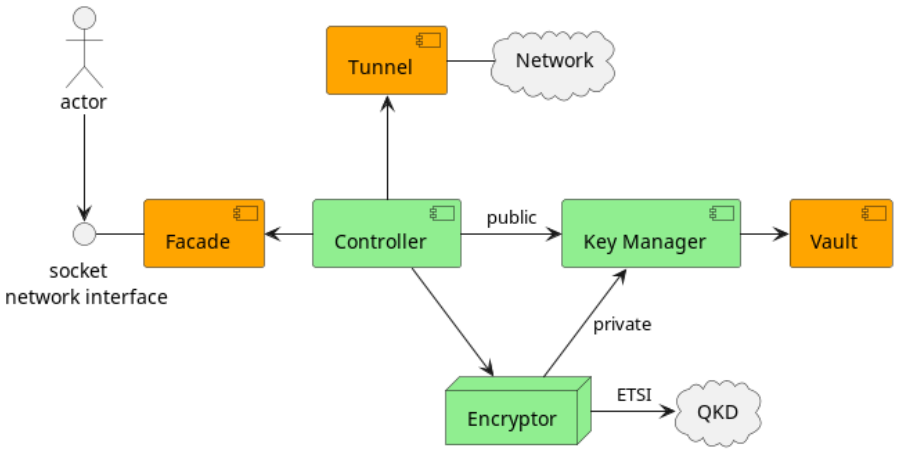


**Figure 3.1:** Modules (green – unique, orange – several instances supported)

### 3.2.2 Communcation - Facade and Tunnel

A Facade transfers messages between the Controller and the user, which usually, but not necessarily[4], means the Facade provides a standard socket or network interface that the user application can use.

A Tunnel provides a network tunnel for communication with other Controllers; in most cases, it connects to a socket or network interface.

The Controller can handle multiple instances of Facades and Tunnels. Still, from the viewpoint of behavior, a Tunnel-Controller-Facade trio is like a pipeline with the Controller in the middle, responsible for securing the messages.

### 3.2.3 Encryptor

The Encryptor module encapsulates and organizes other cryptographic submodules (see Figure 3.2) to provide a stateless module that can secure messages. Based on the Message (maintenance, raw or encrypted) and the available keys in the Key-Manager, the Encryptor can decide what to do with the Message. For raw and encrypted messages, if there is no key available for encryption or decryption, then the module can initiate the creation of new keys via the Key-Agreement module. The new key will be created by one of the Key-Update modules based on what type of key was requested; and will be stored in one of the Vaults via the Key-Manager. An authenticator is responsible for authenticating messages based on preshared data (stored in KM), which is crucial, for example, if the key-update process require multiple authenticated data transfer.

It's important to note that the structure of the Encryptor module enables us to use standard encryption tools. For example, a new key agreement or authentication protocol can be added to the system by enveloping it in a Key-Agreement or Authentication module. Furthermore, if there is a standard, wisely used package like SSL, we can encapsulate it in an encryptor and use it in our system without changing the other modules.

### 3.2.4 Key-Manager

The Key-Manager is responsible for storing private (symmetric and asymmetric) and public keys and their related information. As a module, it has two different interfaces. Via the interface between the Controller and the Key-Manager

- the Controller can only query public information,
- the Key-Manager can transfer messages to the Controller, which can be addressed to the user (answer to user query) or to the Tunnel.

[4] In case, the Controller is part of some special application or network infrastructure, the Facade can play a more specific connector role.

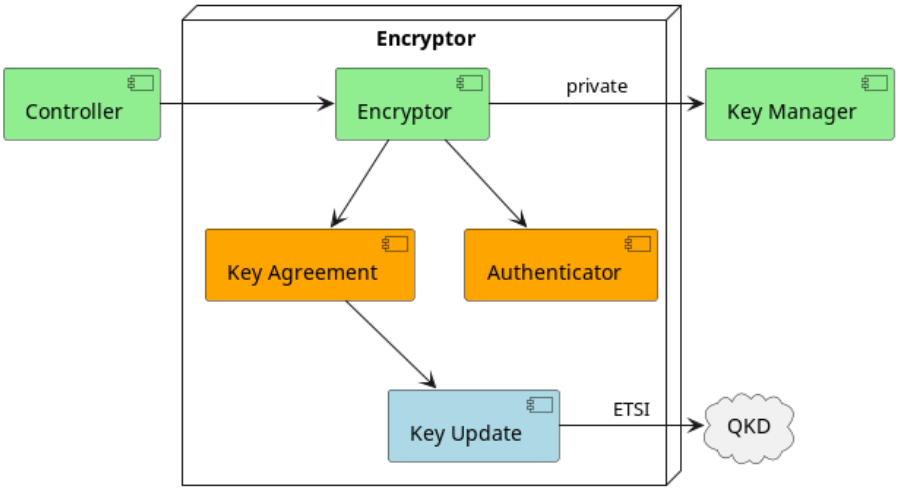


**Figure 3.2:** Encryptor modules

The interface between the Encryptor and the Controller is unidirectional; the Encryptor can

- query public and private keys,
- store and update keys.

As the Encryptor module is stateless, and the Controller has no control over the encryption process, the KeyManager contains all the security-related data in Vault modules.

#### 3.2.4.1 Vault

The Key Manager does not provide storage; instead, it uses at least one Vault module for this purpose. Separating the management and the storage has two essential benefits and one concern. On one hand, this way the Key Manager can use multiple key Vaults that may use different standards and hardware, which makes the data separation clearer and results in better security. On the other hand, the management part can be generalized, and making the management smarter depends on the data and its behavior (keys), not on how it is stored. The concern is the performance as we add another layer to the key query process, but we think the overhead will be small.

#### 3.2.4.2 Keys

The keys can come from different sources, such as being generated by the system itself (for example keys used for authentication) or received from another party, such as keys used for encryption received from the quantum key distribution (QKD) device.

The keys are stored alongside some accompanying information, which may include the following:

- unique identifier
- source
- size
- time of the creation
- other properties

These can be used, for example, to search for the correct keys for a specific use case or for key refreshment.

#### 3.2.4.3 Key refreshment

Key refreshment is the process of updating or replacing the keys periodically or when needed to enhance security.

Besides providing storage (with the Vault module), the Key-Manager module implements different key refreshment strategies according to the sources and types of the keys. Apart from working with fixed expiration dates, it can observe the usage of different keys and initiate (re)create keys based on the behavior. The intent of the Key Manager is passed to the Key Agreement module as an internal Message via the Controller and the Encryptor. In case of QKD keys, the refreshment process also relies on the Key-Update module, which provides a unified interface for different sources of fresh keys, such as key generation or requesting a provably secure quantum key from the QKD device.

## 3.3 Communication between modules

To keep it as simple as possible, only two types of objects can carry data between the different modules. One is the Message, which can be maintenance, raw, or encrypted; and the other is KeyInfo, which contains different types of keys and/or information about them. As mentioned above, the modules behave asynchronously, which implies we use queues for data transfer. The transferred data, mostly the Message objects, are in raw data format, but depending on the selected (based on performance and ease of use) queueing library may be RPC packages.

## 3.4 Implementation and deployment

We use C++ as the main development language for the modules; however, the development of each module is independent, and the language can be changed later separately with minimal cost. In the module hierarchy (see arrows in Figure 3.1 and 3.2), if a parent-child module realized using different programming languages, then, naturally, we need to implement a simple control module that implements the (let's say) C++ interface of the parent. This is necessary due to the way our system deploys its modules as different components.

An application using our solution deploys the system as follows:

1. Controller starts and reads its configuration (from file).
2. Based on the configuration, the Controller initiates the Vaults, Tunnels, Facades, Key Manager, and Encryptor modules.
3. Controller passes the Vaults to the Key Manager.
4. Controller starts and observes (status only) Tunnels, Facades, Key Manager, and the Encryptor.
5. Controller starts to accept and route Messages from the modules through queues.

## 3.5 Performance

Our goal is to provide an application for real-time communication, which, depending on the chosen encryption form, can help to securely transfer text, sound, video, and general data. The critical factor regarding performance should be the selected cryptographic protocols and the availability of the keys, not the communication between the modules. We intend to choose solutions where the cost of the data transfer makes possible real-time video transfer.

# 4 Security analysis

This section is devoted to summarizing the main assumptions and the security requirements the system has to satisfy, focusing on both the communication and cryptography point of view.

## 4.1 Communication building blocks

Note that the proposed encryptor software is supposed to be in the application layer according to the ETSI 014 standard, hence we have some trivial assumptions related to the communication between the layers. The encryptor has three kinds of communication channels, see Figure 4.1:

- Channel between the encryptor and the respective KM
- Channel to other encryptors
- Channel to a user

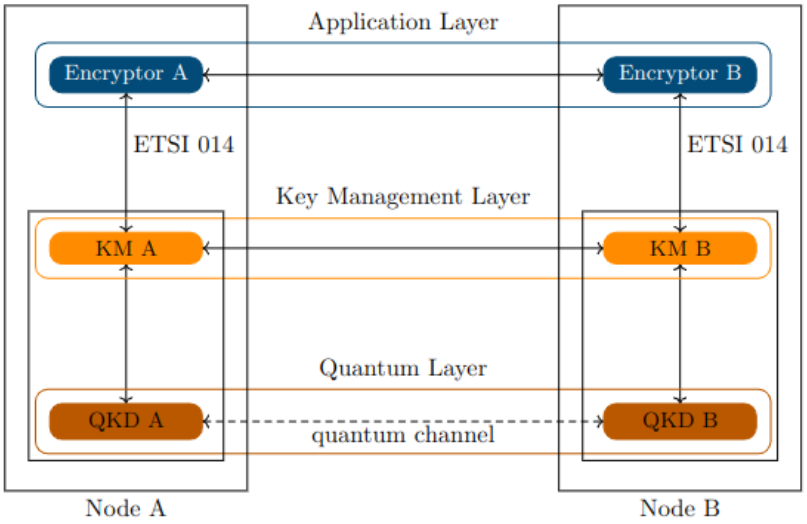


**Figure 4.1:** An example of a point-to-point QKD network

### 4.1.1 Channel between the encryptor and the KM

We assume that every communication between the encryptors and the KMs are using the ETSI 014 standard, hence it is an authenticated secure channel. Form the security point of view, this is the most convenient link, even the adversary can't eavesdrop the sent messages.

### 4.1.2 Channel between encryptors

Since the ETSI 014 assumes nothing about any links in the application layer, we also have no assumption. Hence, this is an unauthenticated open channel, the adversary is able to eavesdrop and/or modify any messages. This is the most challenging link, the encryptor has to solve the authentication and encryption as well.

Let the sender be called Alice and the receiver be called Bob as usual.

### 4.1.3 Channel between the encryptor and the user

This channel is supposed to be a standard API, hence can be protected by standard methods, like HTTPS/TLS.

## 4.2 Cryptography building blocks

The encryptor software has the following main crypto building blocks:

- Key-generation

- Authentication
- Key-agreement
- Encryption

Let us note, that, as a consequence of the modular architecture of the proposed encryptor software, our goal is not choose any arbitrary fixed algorithms for these building blocks. The purpose of this report is to summarize the requirements the particular blocks has to satisfy. In the following sections we give a detailed description of these primitives.

### 4.2.1 Key-generation

The proposed encryptor uses various keys for different purposes. Within this section, we summarize the required functionalities for each of them.

#### 4.2.1.1 QKD keys

We have a preliminary assumption, namely the existence of a QKD algorithm $Gen^{qkd}$ generating (quantum)keys in the following way:

- the `key-generation` $Gen^{qkd}$ takes the security parameter $1^n$ and a length function $l(.)$ as inputs and outputs a pair $(k_{qkd}, ID_{k_{qkd}}) \in \{0,1\}^n \times \{0,1\}^{l(n)}$ of a QKD-key and its identifier.

#### 4.2.1.2 Long-term static keys

We assume that both parties have some long-term static keypairs generated by the key generation step of a particular public-key scheme $\Pi^{pub} = (Gen^{pub}, *)$, i.e., Alice generates $(sk_A, pk_A) = Gen^{pub}(1^n)$ and Bob generates $(sk_B, pk_B) = Gen^{pub}(1^n)$.

#### 4.2.1.3 Authentication keys

We assume that both participants generate their authentication key pairs using the key generation part of the Authentication 4.2.2, i.e. it outputs key-pairs $(pk_A^{auth}, sk_A^{auth})$ and $(pk_B^{auth}, sk_B^{auth})$ for Alice and Bob, resp.

#### 4.2.1.4 Symmetric keys

The symmetric keys are temporary keys generated for every new session by a pair of parties (i.e. Alice and Bob). In fact, these keys are the outputs of the Key-agreement step 4.2.3

### 4.2.2 Authentication

This part aims to establish mutual authentication between Alice and Bob using a long-term public key, secret key pairs, and some fresh randomness.

Formally, the authentication $\Pi^{auth}$ consists of three PPT algorithms:

- the `key generation` $Gen^{auth}$ takes the security parameter $1^n$ as input/outputs key-pairs $(pk_A^{auth}, sk_A^{auth})$ and $(pk_B^{auth}, sk_B^{auth})$ for Alice and Bob, respectively.
- the `authentication` $Auth$ takes the secret keys of $sk_A^{auth}, sk_B^{auth}$ and some random strings $r_A, r_B \in_R \{0,1\}^n$ and outputs a pair of tags $(t_A, t_B)$ where $t_A = Auth_{sk_A^{auth}}(r_A, r_B, pk_B^{auth})$ and $t_B = Auth_{sk_B^{auth}}(r_A, r_B, pk_A^{auth})$
- the `verification` $Vrfy$ takes a pair of tags $(t_A, t_B)$, the strings $r_A, r_B$ and the public keys $pk_A^{auth}, pk_B^{auth}$ as inputs and outputs a bit with $b = b_A \cdot b_B = Vrfy_{pk_A^{auth}}(t_B, r_A, r_B) \cdot Vrfy_{pk_B^{auth}}(t_A, r_A, r_B)$ with $b_A (= b_B) = 1$ if the tag generated by Alice (Bob) is valid and $0$ otherwise.

The algorithms $(Gen^{auth}, Auth, Vrfy)$ can be any arbitrary authentication scheme.

From now on, we assume that every communication of the following building blocks between Alice and Bob is sent through an authenticated channel.

### 4.2.3 Key-agreement

The informal purpose of this block is to establish a common symmetric key between Alice and Bob by combining (one of) the participants' long-term keys and fresh randomness.

Formally, the key-agreement $\Pi^{kagr}$ is a pair of PPT algorithms:

- the `key-generation` $Gen^{kagr}$ takes the security parameter $1^n$ as input and outputs an ephemeral key $r_A \in_R \{0,1\}^n$ for the sender Alice
- the `key-agreement` $Kagr$ takes the static secret key $sk_B$ of the receiver Bob and the ephemeral key $r_A$ and outputs a shared key $k_{AB} = Kagr(sk_B, r_A)$

The algorithms $(Gen^{kagr}, Kagr)$ can be any arbitrary key-agreement scheme, like Diffie-Hellman or more preferably some post-quantum solution.

### 4.2.4 Encryption

Informally, this is a variant of a symmetric key encryption scheme with a special key derivation function, such that the resulting key combines quantum and classical symmetric keys.

Formally, we let $Enc$ and $Dec$ be the encryption and the decription algoritm of some symmetric key encryption scheme (like AES for example). Then the encryptor consists of three PPT algorithms:

- the `key-generation` $Gen$ takes the security parameter $1^n$ as input and outputs a public function $mix : \{0,1\}^n \times \{0,1\}^n \to \{0,1\}^n$ and the encryptor key $k = mix(k_{qkd}, k_{AB})$ where $(k_{qkd}, *) = Gen^{qkd}(1^n)$ and $k_{AB} = Gen^{kagr}(1^n)$
- the `encryption` $Enc$ takes the encryptor key $k$, the identifier $ID_{k_{qkd}}$ and a message $m$ as input and outputs the ciphertext $c = (ID_{k_{qkd}}, Enc_k(m))$
- the `decryption` $Dec$ takes, the identifier $ID_{k_{qkd}}$ and a ciphertext $c$ as input, then first generates the encryptor key $k$ related to $ID_{k_{qkd}}$ and outputs a message $m = Dec_k(c)$

## 4.3 Security requirements

### 4.3.1 Communication building blocks

From a communication point of view, it should be noted that the key delivery interface of ETSI standards relies on other communication protocols, such as HTTPS or TLS. The security of such applications is therefore reduced to these communication protocols.

### 4.3.2 Key-generation

For various building blocks, we will use the celebrated random oracle model of Bellare and Rogaway [8], namely these functions producing random outputs. The trivial and straightforward assumption is that every key generation algorithm must be a random function. Such assumptions are that all of $Gen^{qkd}, Gen^{pub}, Gen^{auth}$ are a random oracle.

### 4.3.3 Authentication

We use the BWM model of Blake-Wilson and Menezes [9] for the authentication. There are two main reasons for choosing this model: on the one hand, the proposed protocol should work in the public key setting rather than based on pre-shared symmetric keys. On the other hand, as a consequence of the modular design of the proposed protocol, the goal of this module is mutual authentication (i.e., there is no need for key exchange in this module).

### 4.3.4 Key-agreement

For the security of the key agreement, the requirement is that the common session keys between Alice and Bob must be unpredictable and hidden from every (eavesdropping) adversary. Formally, we assume that the function $Kagr$ is a random oracle.

### 4.3.5 Encryption

Informally, here we have two main requirements: the used key must be secret and unpredictable, and the encryption must be secure.

Formally, we first assume that the function $mix : \{0,1\}^n \times \{0,1\}^n \to \{0,1\}^n$ is a random oracle. Additionally, we assume the security against the strongest type of active adversary, i.e., $\Pi$ satisfies indistinguishability under an adaptive chosen ciphertext attack (or IND-CCA2 shortly).

# Acronyms

| | |
|---|---|
| BME | Budapest University of Technology and Economics |
| DEM | Data Encryption Middleware |
| ELTE | Eötvös Loránd University |
| ETSI | European Telecommunications Standards Institute |
| HTTPS | Hypertext Transfer Protocol Secure |
| IEEE | Institute of Electrical and Electronics Engineers |
| IP | Internet Protocol |
| IPsec | Internet Protocol Security |
| KIFÜ | Governmental Agency for IT Development (Hungarian NREN) |
| KMS | Key Management Server |
| MACsec | Media Access Control security |
| OSI | Open Systems Interconnection model |
| PQ | Post-quantum |
| QKD | Quantum Key Distribution |
| SSH | Secure Shell |
| TLS | Transport Layer Security |
| Wigner RCP | Wigner Research Centre for Physics |

# Bibliography


[1] Sheila Frankel and Suresh Krishnan. IP Security (IPsec) and Internet Key Exchange (IKE) Document Roadmap. RFC 6071, February 2011.

[2] Quantum Key Distribution (QKD); Protocol and data format of REST-based key delivery API. ETSI GS QKD 014, 2019.

[3] Karen Seo and Stephen Kent. Security Architecture for the Internet Protocol (Section 2.1, Goals/Objectives/Requirements/Problem Description). RFC 4301, December 2005.

[4] Stephen Kent. IP Authentication Header (Section 3.1 Authentication Header Processing). RFC 4302, December 2005.

[5] Charlie Kaufman, Paul E. Hoffman, Yoav Nir, Pasi Eronen, and Tero Kivinen. Internet Key Exchange Protocol Version 2 (IKEv2). RFC 7296, October 2014.

[6] IEEE Standard for Local and metropolitan area networks-Media Access Control (MAC) Security (Section 6.9, Security services). *IEEE Std 802.1AE-2018 (Revision of IEEE Std 802.1AE-2006)*, pages 1–239, 2018.

[7] Peter W. Shor. Polynominal time algorithms for discrete logarithms and factoring on a quantum computer. In *Algorithmic Number Theory, First International Symposium, ANTS-I, Ithaca, NY, USA, May 6-9, 1994, Proceedings*, volume 877 of *Lecture Notes in Computer Science*, page 289, 1994.

[8] Mihir Bellare and Phillip Rogaway. Random oracles are practical: A paradigm for designing efficient protocols. In *Proceedings of the 1st ACM Conference on Computer and Communications Security*, CCS '93, page 62–73, 1993.

[9] Simon Blake-Wilson and Alfred Menezes. Entity authentication and authenticated key transport protocols employing asymmetric techniques. In *Security Protocols Workshop*, 1997.