POLITECNICO DI TORINO

Master degree course in Computer Engineering

Master degree thesis

Applications of Quantum Key Distribution to security protocols

Supervisors
prof. Antonio Lioy
dott. Ignazio Pedone

Candidate
Orazio Lucio Terranova

July 2021

A Martina
Alla mia famiglia
Agli amici vicini e lontani

Contents

Chapter 1

Introduction

Quantum computing represents today an interesting research subject, piquing the interest of many engineering communities. Indeed, thanks to achievements in quantum devices design and realization, obtained during the last decade, quantum computers and quantum networks seem to be a feasible reality to be accomplished in the foreseeable future.
While a network of quantum computers is still a faraway milestone, simple quantum computers are already existent and their sole existence adds to informative systems a new set of vulnerabilities, that cyber-security engineers must take into account while designing their system's security.
Quantum computing can solve challenges that would take way longer times to be solved by standard computing. These kind of mathematical problems are the basis of actual public key security protocols, that are now considered secure just because standard computers are not able to quickly perform these calculations. Quantum computers, instead, would compute way more easily the solutions, obtaining in such a way the cryptographic keys used in the exchanges.
Cyber-security research communities are then considering quantum computers a threat to modern cryptographic systems and are researching possible countermeasures. These can be summarised in two main categories: post-quantum security and quantum security. The latter gathers solutions that exploit quantum computing to enforce security against quantum computing, the former, instead, collect all the countermeasures offering security leveraging on standard computing.

On the other hand, quantum computers are not the only threat menacing modern informative systems, and while standard ones can count on several pieces of research carried out over the last decades, new system models, that are now becoming popular, present new vulnerabilities that are still topic of many studies. For example, Cloud infrastructures, that are now on the verge to implement and offer technologies like Software-defined Networking (SDN) and Network function virtualization (NFV), are characterised by a new set of vulnerabilities.
Cloud infrastructures, indeed have been chosen as a scenario to further investigate and to evaluate the possibility of integration of the QKD protocol. This choice is because nowadays, the actual trend in terms of informative systems, makes it clear that most of the agencies are migrating to this kind of infrastructure. It allows obtaining efficiently resources on-demand, increasing and lowering their amount based on actual needs. This feature represents an ideal solution for agencies that in such a way can minimize their expenses, paying only for resources they need to use.
Anyway, cloud infrastructures, as described in this work, present a new set of vulnerabilities, that led cyber-security researchers to focus on them, and to find strategies to dampen or fully remove them.

This work objective is to prove the feasibility of the integration of QKD into a cloud infrastructure. To do so, it was exploited a Quantum Key Server (QKS), which is a software stack able to exchange keys using the QKD and without prior knowledge of underlying quantum devices characteristics. A QKS has already been developed by the TORSEC group, and our task was to integrate it as a service into OpenStack, which is an open-source cloud infrastructure, and eventually find fitting use cases considering other already existing services.
Analyzing services offered by OpenStack, two of them distinguished themselves as valid options: Octavia and Virtual Private Network-as-a-Service (VPNaaS). The former one is a service offering

Load Balancer-as-a-Service (LBaaS), while the latter provide VPNs establishment on demand. Both of them rely on two common secure channel protocols, respectively the Transport Layer Protocol (TLS) and the IPsec protocol (using the Internet Key Exchange, or IKE, protocol).

To integrate the QKS with these services, a valid strategy could be to integrate it directly within these two protocols, and indeed, that is what we decided to focus our studies on. TLS integration has been realized using two different scenarios: the first one considering the use of Stunnel, an automatic tool able to provide TLS security to applications without changing their sources; the second one instead assumes the use of WolfSSL, a toolkit offering methods to implement TLS protocols, that was properly modified to be integrated with the QKS. Two scenarios were needed since Stunnel, which leverage OpenSSL, and WolfSSL do not offer the same set of options in terms of TLS configuration. Indeed, since we needed to enhance TLS using the QKD keys, we opted to exploit TLS-PSK (where PSK stands for Pre-Shared Key) suites that have been added and described in TLSv1.3.
The standard, though, offers two modalities to use the PSKs, one mixing PSK and Diffie-Hellman Exchange (DHE) to provide authentication and generate key material; and another one allowing to rely completely just on PSKs. While WolfSSL offers to developers primitives to implement both of these modes, OpenSSL and therefore Stunnel, offer only the mode that includes DHE in the process. We decided then to implement and test both modes using these two scenarios.

Moving now the focus on IKE, the integration was slightly different. Also in this case two different possibilities are represented by the two IKE protocols used: IKEv1 and IKEv2. Integrating QKD keys and IKEv1 is a straightforward process since the process used by IKEv1 to generate key material, does not impact the security properties that quantum keys guarantee and therefore the protocol can be used as-is.
IKEv2 instead needs to use a new extension, specifically designed to handle quantum keys, called in this case Post-Quantum Pre-shared Key (PPK). We decided to not include it in the implementation and to not test IKEv1 since it is considered obsolete and administrators are now encouraged to migrate to its second version. Looking for a toolkit that already offered IPsec implementation, we discovered that LibreSwan not only offers the possibility to automatically set up VPNs but also already offers a working implementation of the PPK extension. So, we introduced small changes into its source code to insert code logic needed to integrate it with the QKS.

During the test phase, we proved that these integrations are working as described in related standards and the new resulting workflows introduce an overhead only during the setup phase, which has to be performed for each new connection or rekey operation. So, it has been proven that there is no impact whatsoever on the performance of the communication itself. Moreover, the integration designed shows how easily these protocols can be integrated, requiring just a minimum effort from administrators. The same is true also for the QKS, which can be quickly deployed in a distributed environment and boot up after a quick configuration.

Chapter 2

Quantum computing and QKD

2.1 Quantum computing threat

Researchers groups from all over the world are nowadays increasing their effort to study and understand how to build defences against quantum computers. Indeed, quantum computers represent a real threat to actual cryptographic systems that are based on asymmetric key algorithms. These kinds of algorithms prove their security on the assumption that, still today, no classical computer has been able to perform factorization operations in polynomial time. This does not stand true when you take quantum computers into account, though. Indeed, Shor's algorithm did prove that factorization can be executed in polynomial time (and only one step out of five, actually needs a quantum computer) [1].
While it cannot be clearly predicted how fast quantum technology will scale and spread worldwide, it should still be a concern for those who are relying on actual cryptography systems and need to protect today secrets also for the years to come [2].
In the following, sections will be then introduced both general concepts regarding quantum technologies, both solutions that have been already found.

2.2 Quantum information

Differences between classical and quantum information already start at their basic unit. Indeed, while for the classical computer it is the bit; qubit is the basic unit for quantum information. The main differences among them are summarized in the table 2.1.
A qubit can be physically implemented using an atom, nuclear spin or photon. Taking into account this latter case, its base states are referred to one of its features: *polarization*. In particular horizontal polarization represents state $|0\rangle$, while vertical polarization stands for state $|1\rangle$ (notation used here is called *bra-ket* and highlights that these two states are two vectors).
Qubits state can be represented also as a linear combination of these base states, this feature is called *superposition*. Differently from standard bit strings, qubit strings state not only depends on states regarding each individual qubit but also on states where qubits are combined together. These states, where single qubits cannot be defined without defining remaining involved qubits, are called *entangled* [3].
Superposition and entanglement provide some useful consequences regarding computation, communication and security. Qubit ability to exist in superposition leads to a huge speed-up in terms of information computing efficiency. Quantum elementary gates, indeed, due to this property, can affect all possible states at the same time.
Moreover, since it is possible to aggregate multiple qubits, possible qubits state number grows exponentially, while it only grows linearly for classical bits. So with fewer qubits, you are able to represent many more states.
Entanglement, instead, provides a kind of correlation where each qubit cannot be fully described without accounting for the other one, and whenever you measure one of the two qubits, while

its resulting state still is not deterministic, the returned value will be identical for both of them (feature exploited in quantum communication). This correlation is also non-local, so it will still works no matter what it is the distance between the two of them. Finally, by measuring qubits, you consume its entanglement obtaining two fully separated qubits with no correlation.
Entanglement also guarantees that, whenever two qubits are maximally entangled, no other qubit can share this correlation, obtaining in such a way a connection inherently private [4].
Another handy property that qubits own is related to the Heisenberg uncertainty principle. This principle states that it is not possible to measure a quantum state without modifying it. This concept still stands for qubits as proven by the no-cloning theorem [5].

bit	qubit
generally a macroscopic system, described by voltage	generally a microscopic system, described by an electron spin
its value "0" or "1" depends on a well-defined region where the continuous value, representing it, was measured	its state $\|0\rangle$ or $\|1\rangle$ corresponds to a fixed pair of quantum states that can be distinguished. This pair depends on how the qubit is physically implemented, whether it is an atom or a photon. Qubit intermediate states are indistinguishable from their base states. Indeed, only states that are orthogonal may be distinguished
it can be only "0" or "1" and multiple bit together do not influence each others' value	it can be in its state $\|0\rangle$ or $\|1\rangle$ or even in a combination of all possible intermediate states. Qubits together influence each other and their values can be represented not only by their individual states but also by the tensor product of their states.

Table 2.1. bit vs qubit (source: [3])

2.3 Quantum computing use cases

Quantum computing is a technology that offers many new possibilities to solve problems that classical computers are not able to tackle (or at least, not in feasible time spans). You will now be introduced to some of the most interesting use cases where quantum computing may be implemented [6]:

- *Quantum simulation*: quantum computing makes possible the simulation of chemical processes that need precision, impossible to achieve by classical computers. Thanks to quantum technology it would be possible to simulate microscopic processes such as nitrogen or carbon fixation. To give an idea of the importance of these results, this latter process cited consists of carbon dioxide conversion to an organic compound, a process that could be really useful to fight against global warming;
- *Quantum learning*: quantum computing has been proved to be able to perform machine learning model training and evaluation faster than any classical computer may be able to do. In particular, this speed-up is applicable to different kinds of machine learning models such as k-means or clustering ones. Right now, quantum learning major challenge concerns

how datasets are loaded into memory. Indeed, to fully experience quantum computing advantages, also data loading techniques should not leverage on classical ones (for example, to load data from a classical database, a QRAM should be implemented);

- *Quantum cryptography*: quantum computing can provide new cryptography systems featuring security features that cannot be provided by any standard computer. Since this application is the one investigated in this thesis, it will be discussed in detail in the following sections.

2.4 Quantum and post-quantum cryptography

As said in section 2.1, quantum computers are able to solve challenges impossible to be solved, in feasible time, by standard computers. On the other hand, other kinds of algorithms, even if solved faster, still remain a difficult challenge even for quantum computers.
For example, Grover's algorithm, the task to search roots for a function f, can be solved by standard computers after N evaluation. Quantum ones, instead, need only $\sqrt{N}$ evaluations. This speed-up, anyway, is not comparable to the performance increase gained with Shor's algorithm.
Performance benchmark about Grover's algorithm performance is a way to measure the ability to crack symmetric algorithms (by simply substituting function f with the one appearing in cryptography algorithms). Given that the implemented system is using algorithms with keys equal to or less than 128 bits, then quantum computers may be able to threaten it. Increasing key length to 256 bit, is enough to obtain a quantum-safe system, though [7].
Since most actual informative systems do not rely only on symmetric cryptography, that as you saw are relatively quantum-safe, but also on asymmetric algorithms, that are not quantum-safe, research groups and organizations are working towards the future of system security in a quantum information environment. Until now, two main answers have been provided and are actively researched as well as tested: post-quantum cryptography and quantum cryptography.
The former one comprehends all those asymmetric algorithms that have been proven to be resistant to Shor's algorithm, the latter one, instead, comprehends cryptography systems that leverage new features provided by quantum computers.

2.4.1 Post-quantum cryptography

While, as you will see in the following sections, quantum cryptography is today synonymous with its only algorithm proved in real use cases: Quantum Key Distribution. Post-quantum cryptography already provides several possibilities. Now the most popular ones will be introduced [7]:

- *Code-based encryption*: some computers achieve a higher degree of reliability due to an error-correcting code. It consists of the encoding of 64 bits of data into 72 bits of physical memory. It is then computed and stored as a generator matrix used to check its correctness. McEliece proposed an asymmetric system where the generator matrix is the public key. Classical attacks against this system have been proven to be too slow, and differently from symmetric key algorithms, quantum computers power does not affect its complexity enough to be considered a threat.
  The key length needed, especially when high-level security is a constraint (about a megabyte), represents the main concern regarding this system;

- *Lattice-based cryptography*:

  - *Lattice-based encryption*: introduced by Hoffstein, Pipher and Silverman, NTRU, a cryptography system that needs a much smaller key (8173 bits, using standard parameters). These kinds of systems are based on the challenge to find a lattice point close to a given point. While NTRU is still uncracked, it is true that these kinds of systems have not been researched enough to guarantee a certain degree of security;

  - *Lattice-based signature*: today most promising system is Lyubashevsky's one. It is based on the challenge to find, given a matrix A, a small vector v that solves $Av = 0$ [8]. This is a relatively new system (published in 2012), so its security degree is still under research. In particular researchers efforts are focused on limiting possible side-channel attacks as well as analysing system challenge complexity;

- *Multivariate-quadratic-equation signature*: there are many systems that provide signature-based on multivariate-quadratic-equation, anyway here only $HFE^{(v-)}$ will be cited since it is until now unbroken and it is proven to be quantum-safe (assuming it is executed using correct parameters). This system also rose interest since its generated signatures are quite short (considering standard parameters it is only 34 bytes long);

- *Hash-based signature*: these systems leverage the computational time needed to obtain input given to hash functions from their output. Lamport was first to propose to exploit it as a signature system. It works as follow: signer picks two random strings x_0, x_1 and stores them as its private key, then it computes $h(x_0), h(x_1)$ and distributes it as the public key. Whenever a signature is required, whether 0 or 1 should be signed then respectively x_0 or x_1 is revealed, so the verifier can check its correctness recalculating its hash value and comparing it with the corresponding half of the public key.
  Of course, this kind of system are valid for one-time signatures, thus major concern regarding these systems is that large public keys are used only once.
  Merkle proposed a solution to this issue that needs 2^k public key two to generate a public key able to verify 2^k signatures. To enable this scheme must be populated a binary tree where there is a leaf for each signature and its root is the generated public key calculated while building the tree. Therefore signer must store additional data to allow verifiers to check signatures.
  Another big concern relative to hash-based signatures is that each secret should be used only once, then each signer should keep track of secrets already used. This may represent an issue for all those signers hosted on stateless systems. There are already some alternatives to implement stateless hash-based signature, but they come at cost of performance in terms of key length and signature generation time.

2.4.2 Quantum cryptography

Quantum cryptography right now is one of the most promising applications for quantum information. Its potentiality is expressed by Quantum Key Distribution protocols that already are implemented in some real use cases [9]. This kind of protocol is based on the assumption that it is possible to secure communication between two peers that are linked by a quantum channel and a classical channel. The former one is needed to send and receive qubits, such as photons polarization states, while the latter is used to negotiate data regarding the key agreement. The overall process costs of four main tasks:

- Raw Key: key obtained from qubits transmitted;
- Key sifting: key reconciliation between peers using classical channel;
- Key distillation: if key produced after error correction is good enough to be distilled, then privacy amplification is performed;
- Usable key size: conclusive agreement on resulting key.

While these steps remain common among all QKD protocols proposed until now, these protocols can differ about the base conceptual scheme used. Indeed they can be prepare-and-measured-based, or entanglement-based. Former ones leverage on the idea to prepare the raw key and then send it to the corresponding peer that measures it; the latter scheme, instead, exploits the entanglement feature to distribute the key. Even if entanglement-based schemes are the ones that most take advantage of quantum information advantages, they are difficult to implement in a real use case, then nowadays prepare-and-measured-based protocol is the most researched and used [10].

Protocols BB84 and E91 will be briefly introduced to represent respectively prepare-and-measured-based and entanglement-based protocols:

- BB84 is the most implemented QKD protocol due to its relatively low requirements. It uses photon polarization to exchange data over a quantum channel, and as with other prepare-and-measured-based protocols, its security is based upon the handy feature of quantum states regarding the impossibility to measure them without being noticed (as already said in 2.2).
  BB84 protocol workflow can be summarized in the following steps:

  1. Alice will generate k random bits with value 0 or 1, then it will send a corresponding sequence of photons to Bob, encoding the random bit string using polarization to represent 0 or 1. In particular bit 1 will be represented by polarization vertical(90) and diagonal(+45), while bit 0 will be represented by polarization horizontal(0) and anti-diagonal(-45);
  2. Bob will receive each photon and then it will measure them choosing randomly between two bases: the rectilinear basis or the diagonal basis. When it is done measuring photons, it will finally have a raw key;
  3. Bob will then send back to Alice the time slot in which photons were received and a string containing the basis used to measure each photon;
  4. So Alice is able to compare the basis string sent by Bob with the photons sent in the declared time slot. Finally, it will communicate to Bob the list of basis it correctly chooses. Doing so a sifted key is obtained. In this phase, they should also measure the quantum bit error rate to evaluate if a third party (usually called Eve) is eavesdropping. If the resulting error rate is below a certain threshold they can obtain the final key after performing error correction and privacy amplification, otherwise, they should drop the key.

- E91 is an entanglement-based protocol since peers involved in key distribution own a particle belonging to an entangled pair. Similarly to BB84 then a key is prepared and transmitted exploited entanglement. E91 security is based on peers ability to recognize an eavesdropper presence by using Bell's inequality test (a test introduced to prove that Einstein hypothesis regarding particles behaviour leads to algebraic conclusion contradicted by quantum mechanics [11]). Indeed, two photons maximally entangled violate Bell's inequality test by its maximum value, and since a third actor, interacting with this pair, would decrease its entanglement, Bell's inequality violation degree may be used to tell whether or not someone is interfering [12].

2.5 QKD theoretical and implementation security

From a theoretical point of view, QKD represents a novelty since, for the first time in cryptography history, its theoretical security is not affected by eventual eavesdropper resources. It also has been proven that failures probability in QKD can be approximated and arbitrarily reduced (at cost of secure key rate).
QKD has, moreover, been proven to be universally composable, then QKD keys can be used with other cryptography techniques without losing their features.
Finally, QKD keys are said to be "everlasting", because, differently from standard keys, they cannot be broken retrospectively.
Anyway, like any other protocol, QKD is vulnerable to implementation errors which lead to vulnerabilities that are not taken into account while analyzing it from a theoretical point of view. It is indeed much more usual that an attacker would try to take advantage of implementation errors rather than try to crack the actual protocol [13].

2.5.1 Attacks exploiting QKD implementation vulnerabilities

In this section, the most known attacks against QKD will be briefly listed. As you will see, they all exploit implementation vulnerabilities rather than focusing on protocol possible faults [13]:

- Side-channel: this kind of attack happens when valuable data regarding the system, or the information itself, is leaked due to bad implementation. In QKD this could happen, for example, whenever a sending unit may be emitting lights over multiple frequencies. In this case, an eavesdropper may be able to capture signals without compromising the original one and therefore without being noted;
- Trojan-horse: it is possible for an attacker to proactively inject some light into sending units and then capture the light that is inevitably reflected due to electro-optic devices nature. It is possible, though, to minimize the amount of information leakage by taking advantage of isolators and privacy amplification;
- Multi-photon emission: QKD systems should be able to send one single photon per emission. Anyway, this is nowadays not possible since there are no devices able to do so, yet. This condition represents an implementation vulnerability that can be exploited by an eavesdropper who could conduct a photon-number-splitting attack: the attacker captures all pulses, dropping pulses containing single photons and keeping for themselves one of the multiple photons present in remaining pulses, forwarding remaining ones. This vulnerability can be anyway highly limited, exploiting correctly privacy amplification and modelling correctly pulses emitted. However, doing so, the secure key rate decreases severely;
- Imperfect encoding: as seen previously, QKD needs to transmit data over an insecure public channel to instruct another peer about how quantum states should be prepared. Sometimes though, due to physical devices limits, prepared states could slightly different from those ones described by the protocol. This variation, combined with typical losses experienced in communication channels, could lead to high key rate reduction;
- Phase correlation between signal pulses: since emitting units are usually implemented as attenuated lasers, it is common that subsequent pulses should share a certain degree of phase correlation. Even if this is not directly considered a security issue, it has been proven that setting random phase among adjacent pulses leads to better performances;
- Bright-light attack: QKD systems always need single-photon light detectors to work properly. These devices must enter a mode, called linear, anytime they need to reset back to their initial state. It has been observed that an attacker could exploit this linear mode to obtain bits of the final key, without being detected. As a countermeasure, administrators should correctly set these devices and concurrently monitor their parameters;
- Efficiency mismatch and time-shift attack: Usually QKD systems need two detectors, one for each bit (0 or 1). Since it is impossible to find two detectors identical, an attacker may try to understand which detector responds to each pulse in order to learn which bit was received. To do so, two possible attacks may be carried: efficiency mismatch and time-shift. The former exploits eventual differences in detectors efficiency (depending on different wavelengths); the latter, instead, takes advantage of differences in time responses.
  Similarly to other implementations vulnerabilities, privacy amplification represents a quite good choice to limit differences between detectors. Detectors symmetrisation is another approach proven to be effective and it consists of random bit assignment among the two detectors. Doing so, from the attackers' point of view, detectors are almost identical;
- Back-flash attack: Also this attack exploits the detectors nature and attacker's ability to gain knowledge related to detection events. Indeed, whenever a detection event happens, secondary photons are emitted and may travel back through the transmission channel and reach attackers. This kind of attack is said back-flash attack and can be prevented by designing correctly QKD systems (inserting isolators, spectral filters and preferring short gates over fast gates). Detectors symmetrisation is also a valuable countermeasure against this vulnerability;

- Manipulation of local oscillator reference: In some QKD systems it is needed to transmit an intense phase reference. It can be vulnerable to attacks that aim to bias noise estimation to remain unnoticed. Moreover, attackers may try to modify local oscillator pulse or amplitude, getting control of clock signals.
  Real-time local oscillator monitoring, as well as implementing systems where local oscillator can be regenerated at the destination, are valid countermeasures;

2.6 QKD use cases

Infrastructures and systems widely spread and usual nowadays are menaced by quantum computers. Indeed, as long as they implement security by means of not quantum-safe algorithms, such as public-key algorithms, attackers may exploit quantum computers to crack them.
In the following list are collected common use cases, that depict common infrastructure, as well as systems whose infection could lead to catastrophic consequences, and which security could be enhanced by means of QKD [14]:

- Encryption and authentication of endpoint devices: endpoint devices are any kind of device linked to a distributed computing system or network. Most common devices, such as personal computers and smartphones belong to this category. All of them usually protects storage encrypting them to a certain degree. But, even in the case of full-disk encryption, and even when symmetrical cryptography algorithms are used, usually keys are generated by means of asymmetric algorithms which are not quantum-safe.
  Security breaches in endpoints do not affect only them but also infrastructures where these devices are authenticated. Attackers could, indeed, access data and resources meant for the cracked device, and also possibly inject new vulnerabilities;

- Network infrastructure encryption: obviously endpoint devices are not the only ones managing data that need to be encrypted. It stands similarly for networks and their transient data. Actual typical defences, in public networks, are based on not quantum-safe algorithms, such as TLS that leverage on RSA and Diffie-Hellman respectively for authentication and key agreement.
  Similarly, organizations protect their networks using layer 2 or layer 3 encryption techniques. Layer 3 encryption usually takes advantage of IPsec, therefore IKE, which in its standard implementation is not quantum-safe. Regarding layer 2, instead, there are already some commercial products that exploit QKD to provide encryption;

- Cloud storage and computing: these kinds of systems provide resources hardware and software to clients who prefer to outsource them. They guarantee some advantages such as highly reducing the complexity of IT management for their clients, such as provide resources only when actually needed, limiting then possible expenses.
  These kinds of infrastructures are said to be quantum-safe when both servers, both networks linking them, are quantum-safe;

- Big data, data mining and machine learning: These concepts, which are rising interest among research communities as well as private companies, leverage the ability to extract patterns and valuable data from huge datasets. This amount of data must be both stored and moved between several nodes. Differently from standard networks and data, in this case, attackers may obtain a large amount of data that could be exploited both to gain generic knowledge (patterns and data of interest for the victim, that usually is economically valuable), both to get really specific insights on individuals (such as user usual locations, habits, etc.). Like the previous case, even in this one, achieving quantum security means giving access to quantum-safe algorithms to servers and networks;

- SCADA systems: Supervisory Control and Data Acquisition systems are those implemented to monitor and control industrial systems. In this case, vulnerabilities may let attackers gain access and control, for example, of real factories, electrical grids or airports. It is obvious then that, in this case, security breaches may also endanger human lives. Despite this, these

systems are relatively recent and with no standard available, their security is usually still managed by obscurity.
This means also that future analysis should lead to standards that already will take into account quantum computing threats.

Chapter 3

IKE Protocol

3.1 Introduction

In order to achieve IP security, it is needed that a shared state is maintained between two endpoints. By doing so, it is possible to guarantee confidentiality, authentication and integrity. Since establishing manually these kinds of states is not feasible, it must be implemented a protocol responsible for this operation. While OAKLEY and ISAKMP were initially designed and introduced for this purpose, they were then combined in a more comprehensive protocol named IKE.

The Internet Key Exchange protocol is the one now that is nowadays used to establish these states, called *Security Associations* (SA). By means of SAs, nodes can establish several communication channels that can leverage on either textitAuthentication Header (AH) or *Encapsulating Security Payload* (ESP), and, in the meanwhile, keep track of the cryptographic suites negotiated for each of them.
There are two possible versions available of this protocol: IKEv1 and IKEv2. Since IKEv2 is all around a better protocol than its previous version, it is the one picked to be analyzed in this chapter, as well as implemented and tested in this work.
Initially, a general overview of the protocol will be presented, introducing the four main kinds of exchanges that may occur. Then, some of the most interesting variations from the usual protocol flow will be described.
In the following paragraph, it will be explained how the protocol prevents some kind of attacks and which are its main flaws. Finally, the main differences between the two existent protocol versions will be highlighted.
It must be stressed that all data collected and shown here has been gathered from the RFC 7296 [15], the document published by IETF where IKEv2 is described.

3.2 Protocol overview

IKE communications are always meant to happen in pairs. The peer starting the message flow is called *Initiator*, while the node responding is called *Responder*. Each pair of messages, request and its correspondent reply, composes an *exchange*. As defined by the protocol, the first two exchanges type must be IKE_SA_INIT and IKE_AUTH, and no other order is allowed. Only after that, these two have been processed, then other kinds of exchanges may occur. The main ones are CREATE_CHILD_SA and INFORMATIONAL_EXCHANGE.
We will take now a closer look at each of these communications, introducing also packet formats whenever they appear in an exchange.

3.2.1 IKE_SA_INIT

The first exchange's role is to declare the will to establish a new SA with the recipient while announcing the cryptographic suites supported by the initiator. In the same message, it is sent the initiator's *Diffie Hellman* (DH) value as well as its nonce.

Figure 3.1. IKE_SA_INIT request

It is then the recipient responsibility to choose its preferred cryptographic suite, among the received set, and then reply accordingly including also its DH value and nonce. Optionally, it can also request the initiator's certificate including the request certificate payload (CERTREQ in the image **??**).

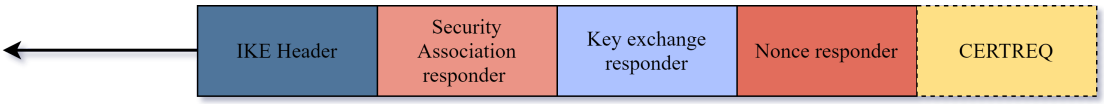

Figure 3.2. IKE_SA_INIT reply

IKE Header

IKE exchanges are transmitted over UDP and the standard ports are 500 and 4500. UDP packet pieces of information are discarded, except for the peer's IP and port. Each IKE message must always start with an IKE Header. Whenever port 4500 is used for any exchange, four zeroed octets must be put before the message.
Let's breakdown the packet field by field:

1. *Initiator and Responder SPIs [2 * 8 octets]:* instead of IP address and ports, SPIs are used to indexing IKE SAs. An SPI value should be zero only when the Initiator does not know Responder's SPI yet.
2. *Next Payload [1 octet]:* field used to declare next payload's type.
3. *Major and Minor version [2 * 4 bits]:* while the former can be set to "1" or "2", depending on the implemented version, the latter must be set to zero and ignored upon reception.
4. *Exchange type [1 octet]:* here is defined the type of exchange. By means of this value, it is possible to know which kind of payloads should be expected.
5. *Options flags [1 octet]:* Except for bits in positions 2, 3 and 4, all bits must be cleared when this message is composed and sent and must be ignored if it is received. The bit in the second position is used to understand if the message is a request or a response (0 = request, 1 = response). The following bit is instead exploited to declare that the sender is able to use a higher version number of the protocol. Finally, the bit in the fourth position is set in all messages sent by the original initiator and cleared in those sent by the responder. This information is useful to pick the correct SPI to be read.
6. *Message ID [4 octets]:* Used to match in each exchange the request with its response, it is exploited also to prevent to a certain degree replay attacks.

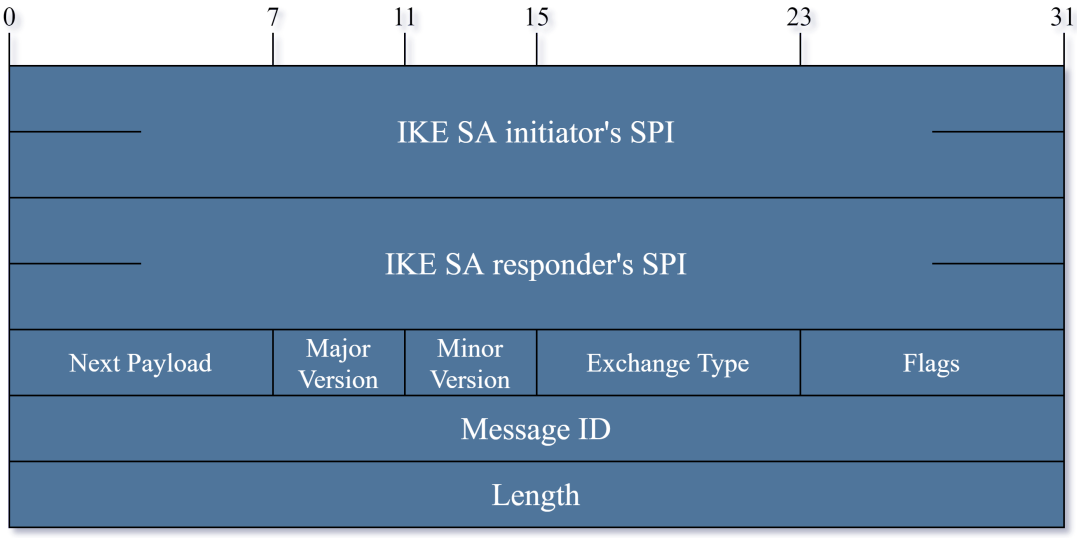

Figure 3.3. IKE Header

7. *Length [4 octets]:* Here it is stored the total length of the message in octets. The sum comprehends the header and the payloads. The zeroed octets prefix inserted when the message is sent to port 4500 are not considered.

Generic Payload Header

Figure 3.4. Generic Payload Header

Before introducing payloads expected in the IKE_SA_INIT, we will analyze the generic header format that they all share:

1. *Next Payload [1 octet]:* Field used to declare the payload's type of the next one in the chain. When this value is set to zero then the payload following this header is the last one. The encrypted payload represents an exception, though. Indeed, even if it always must be the last payload, it contains data structures that are considered payloads. Then this field change accordingly until the last "contained payload" is reached, then it is set to zero.

2. *Critical [1 bit]:* This bit is used to express to the recipient the behaviour expected when it cannot understand the payload type. It must be underlined that this bit is not referred to the next payload in the chain, but to the payload following the header. If it is set to "1" and the recipient cannot understand the payload, it must drop the entire message. If it is set to "0", instead, the recipient can skip it. It is must be set to "0" for all payload types defined in RFC-7296 [15] since those payloads are meant to be understood by all nodes implementing this protocol.

3. *Reserved [7 bits]:* Set to "0" by the sender and ignored by the receiver.

4. *Payload length [2 octets]:* Length in octets of the payload following this header (including the generic header).

Security Association Payload

This payload's role is to convey the preferred cryptographic suites that the initiator intends to use. It is needed whenever any kind of new SA is established, whether it is a new IKE, AH or ESP SA.
Its structure is composed of the generic header followed by at least one proposal. Each proposal contains transforms and attributes, needed to define the algorithm intended to use for encryption or integrity and its attributes like the key length. It is possible to offer encryption and integrity by means of two different algorithms or to implement them in a combined fashion. Two different proposals must be encoded to do so.
Transforms define, for example, the algorithm offered for encryption or integrity. Exploiting transforms' type number, the recipient can understand if the algorithm is intended to be used for encryption or integrity. Several transforms sharing the same type number are to be considered as options from which the recipient can choose. The recipient must choose one transform for each type. For example, if two transforms for encryption and two transforms for integrity are encoded, then the recipient must choose a pair picking one transform for each type. Thanks to this structure, several proposes are encoded together. Indeed, in the proposed example, four pairs can be chosen.

Key Exchange Payload

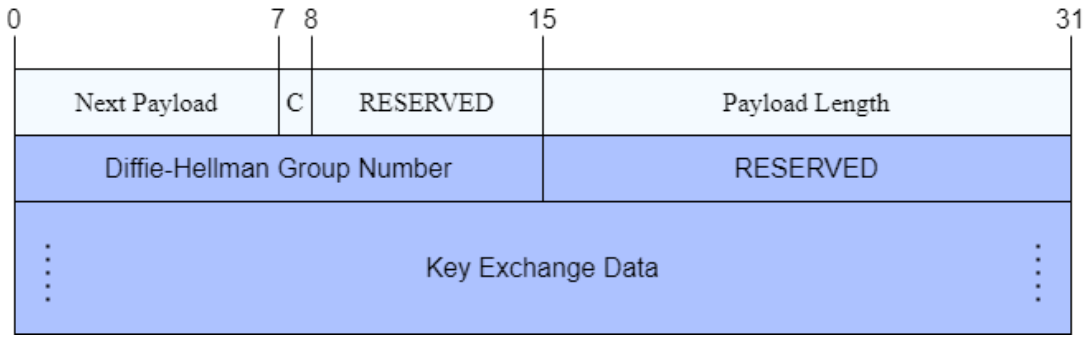

Figure 3.5. Key Exchange Payload

By means of the Key Exchange Payload, the Diffie-Hellman public values are exchanged.
It is composed by:

1. **Diffie-Hellman Group Number [2 octets]:** It define the group number used to compute the value exchanged. It must match the group number negotiated in the SA payload.
2. **Key Exchange Data [variable]:** It is the actual public value. Its length depends on the group number chosen.

Nonce Payload

This payload purpose is to protect against replay attacks and useful to check communication's liveliness. It is simply composed of the generic header plus the randomly generated data, which length is variable.

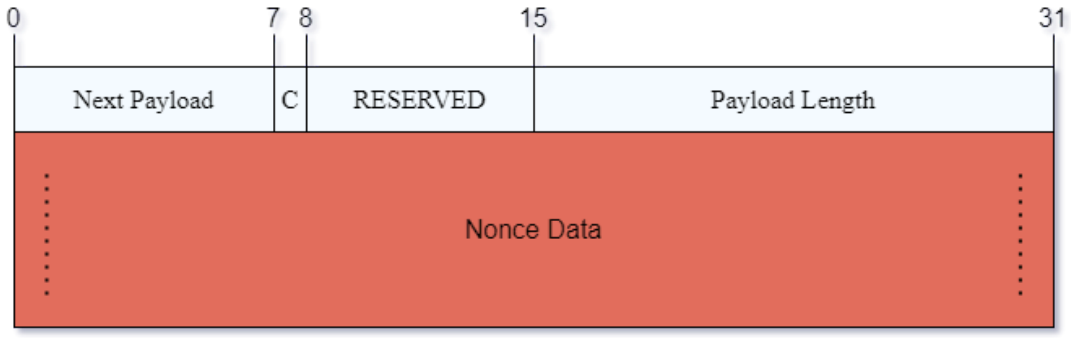

Figure 3.6. Nonce Payload

Certificate Payload

The sender, encoding this payload, can transmit data meant to be used for authentication. Its name is quite misleading since this payload can carry not only actual certificates. The type and structure of data present in the payload are defined by the Certificate Encoding field.

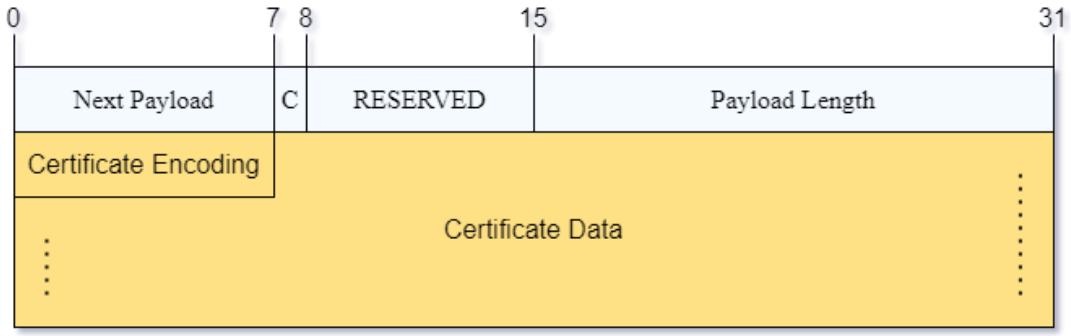

Figure 3.7. Certificate Payload

Certificate Request Payload

The certificate request payload, as it can be easily imagined, is sent to request a particular subset of certificates that are preferred by the sender. Instead of the certificate's data, the payload conveys the encoding of a certification authority considered acceptable.

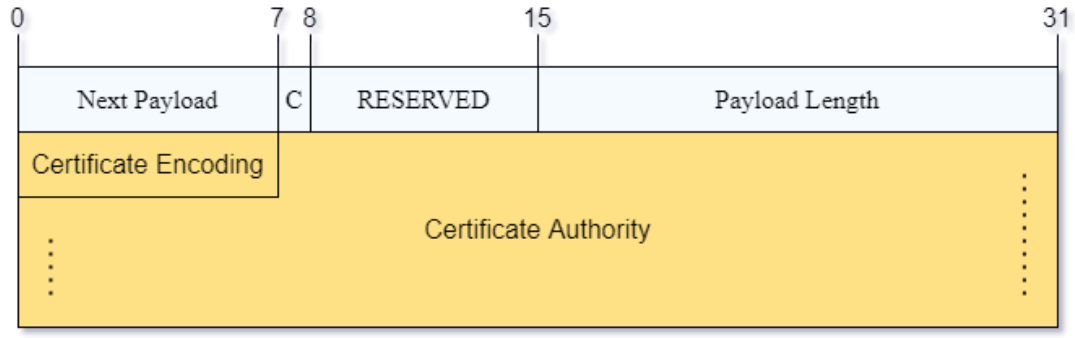

Figure 3.8. Certificate Request Payload

3.2.2 IKE_AUTH

After the IKE_SA_INIT exchange, peers can generate a value called SKEYSEED. Using this value, SK_a, SK_e and SK_d keys are computed. They are used respectively for authentication, encryption and computation of new keys for Child SAs. For the following images, the encryption and integrity protection of some payloads will be shown using the keyword *SK*. It is important to keep in mind that SK values are computed by each peer, then SK_a - SK_e pair used will be different depending on the direction of the communication.
Next to the IKE Header, the Identification payload is encoded and used to announce the initiator's identity. This statement will be then proved by means of the AUTH payload, where is stored the signed version of the first message (IKE_SA_INIT initiator's message). Optionally, it can include a Certificate payload, a Certificate request payload and/or an Identification payload. The latter one can be included to ask to communicate to a specific identity owned by the responder.
During IKE_AUTH exchange the first Child SA is established. Indeed, cryptographic suites are negotiated by means of SA payload, which carries the number "2" as Message-ID, since it is the first half of the second exchange (the first one was in the IKE_SA_INIT exchange).
Finally Traffic Selectors payloads are encoded. These two payloads' purpose is to declare the source IP address range and destination IP address range used to filter the packets that must be routed through the channel established and protected as negotiated in the related SA.

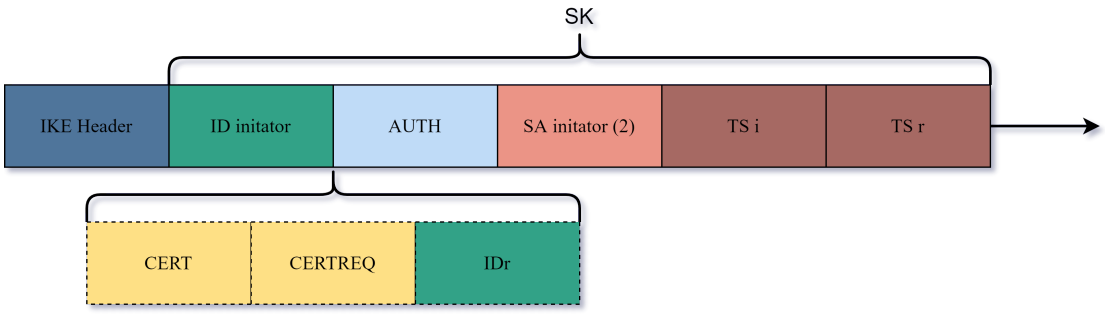

Figure 3.9. IKE AUTH request

The responder replies with a quite similar message, except for the missing CERTREQ and IDr payloads (the former one should be encoded in its first message). Similarly to the IKE_AUTH request, the responder declares its identity, proved in the AUTH payload, and selects its preferred cryptographic suite for the communication. Moreover, it will accept or select a subset of the IP address ranges sent in the request TSi and TSr payloads.
Keep in mind that it is each peer responsibility to check signatures and MACs correctness. If CERT payloads are included, then the first certificate contained must be the public key used to sign the AUTH payload.

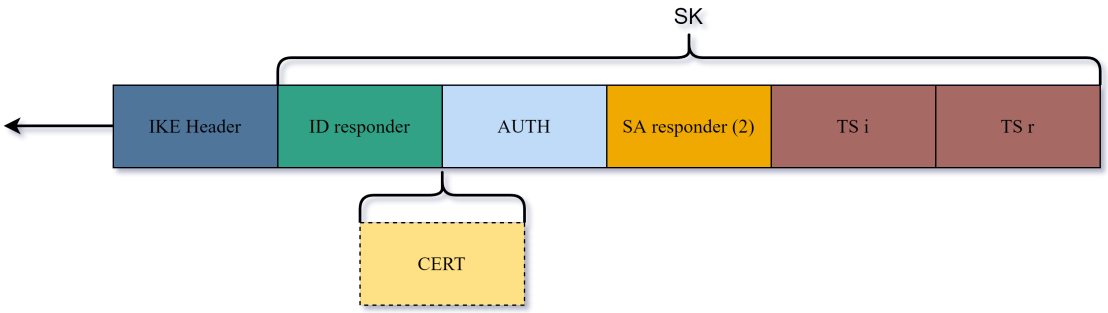

Figure 3.10. IKE AUTH reply

ID Payload

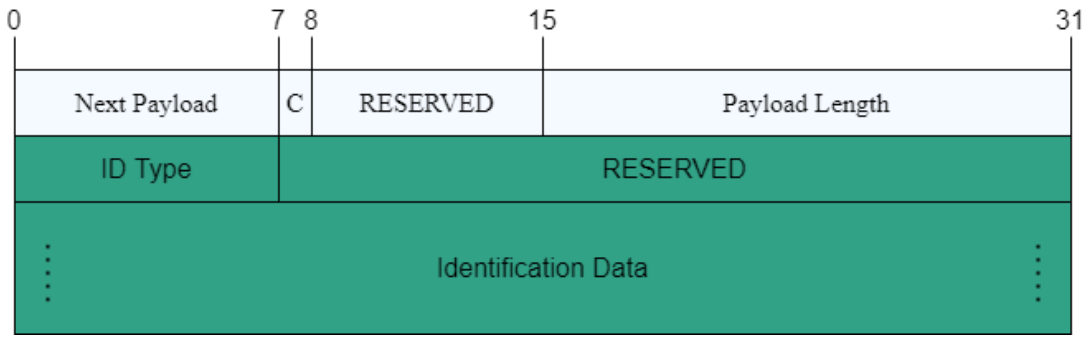

Figure 3.11. ID Payload

This payload carries the identity declared by its sender. The type of identity encoded in Identification Data is defined by the ID Type field. The main types are IPv4 address, IPv6 address, e-mail address, full domain name.

AUTH Payload

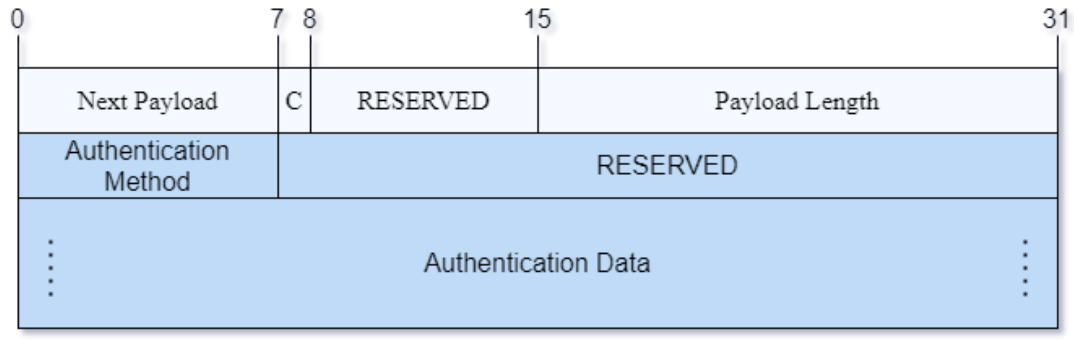

Figure 3.12. AUTH Payload

As explained before, this payload contains data needed to authenticate the sender. The authentication method chosen informs the receiver about the Authentication Data type. Possible types are:

1. **RSA Digital Signature:** data signed using an RSA private key (with RSASSA-PKCS1-v1_5).

2. **Shared Key Message Integrity Code:** the shared key is the one associated with the identity declared in the ID payload, while the PRF is the one negotiated before.

3. **DSS:** digital signature using a DSS private key.

Traffic Selector Payload

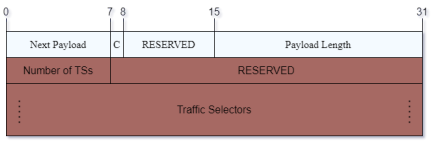

Figure 3.13. Traffic Selector Payload

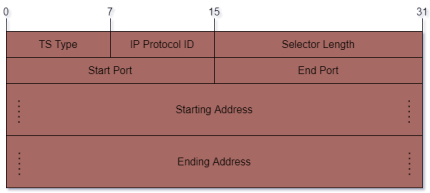

Figure 3.14. Traffic Selector Type

Traffic selector payload can contain one or more selectors. The total count of selectors is stored in the field "Number of TSs". Since multiple traffic selectors can be selected, both for source and destination address ranges, when a packet is filtered, its source and destination addresses are checked against these lists. As long as they fall inside at least one of the TSi ranges and one of the TSr ranges, they are accepted. For each traffic selector, there are several possible types defined by the TS type field. Moreover, the IP protocol (TCP, UDP, etc.) is declared in the IP Protocol ID field. Regarding the port range selected, two fields store this information: Start Port and End Port. It is possible to define an undefined port range storing a zero in the start port and the value 65535 in the end port. Similarly, IP address ranges are defined in starting address and ending address fields.

3.2.3 CREATE_CHILD_SA

This kind of exchange can happen only when IKE_SA_INIT and IKE_AUTH have been correctly performed. It usually consists of one single pair of messages. It can be used both to create a new Child SA under an existing IKE SA or to rekey a SA (whether it is IKE or Child).
Messages format is quite similar to the one seen in IKE_AUTH, except for the missing CERT and CERTREQ payloads. Moreover, in these messages KE payloads are optional. It is possible, indeed, to send them to score stronger forward secrecy for the Child SA that it is being established.

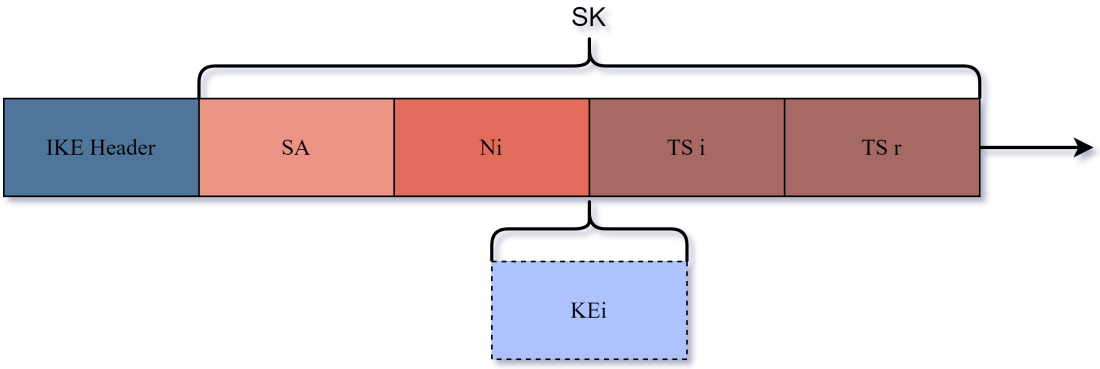

Figure 3.15. CREATE_CHILD_SA initator

The same reasoning goes for the response message format. When, instead of creating a new Child SA, one of the peers would like to rekey an IKE SA, it is sent a message similar to the one used in an IKE_SA_INIT, where the responder SPI is not zero (because the SA already exists) and the initiator SPI is a new one. The responder then can change its SPI, establishing in this way a new SA that clones the previous one. After this moment, no new Child SA can be added to the old IKE SA (that will be deleted as soon as possible).

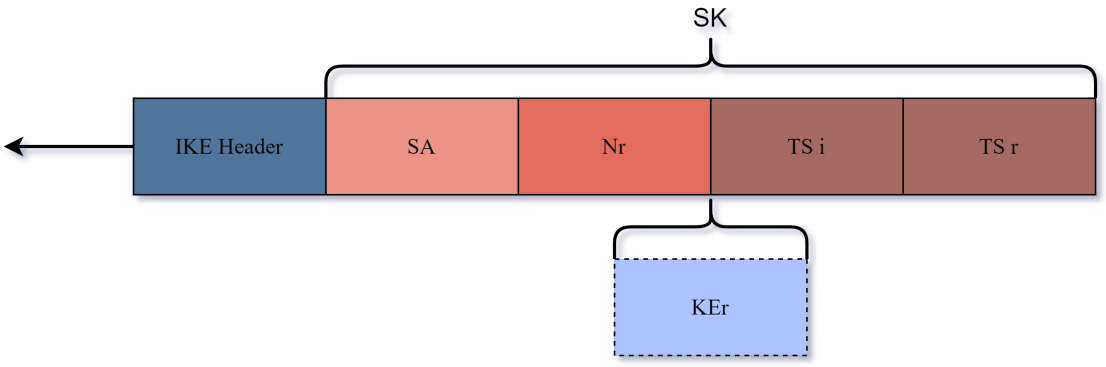

Figure 3.16. CREATE_CHILD_SA responder

To rekey a Child SA, instead, it is sent a message equal to the typical CREATE_CHILD_SA that carries a Notification payload that declares the will to rekey the referred SA. The responder's reply is exactly as the usual one in the CREATE_CHILD_SA exchange.

3.2.4 INFORMATIONAL

As for every protocol, also IKE needs at some point some sort of mechanism to exchange information about particular events or errors. These kinds of messages are called INFORMATIONAL

exchanges. They can be empty or contain more payloads containing data needed to define the event that occurred. They could be Notification payloads, Delete payloads, as well as Configuration Payloads. It is important to observe that these exchanges usually are encrypted and integrity protected, using the crypto suites negotiated for the SA interested. This is not true only in some specific cases.
For each INFORMATIONAL exchange request there must be a reply (even an empty one), otherwise, the initiator will assume that the packet got lost and will retransmit it. Thanks to this behaviour, an empty request can be used to check responder liveliness.

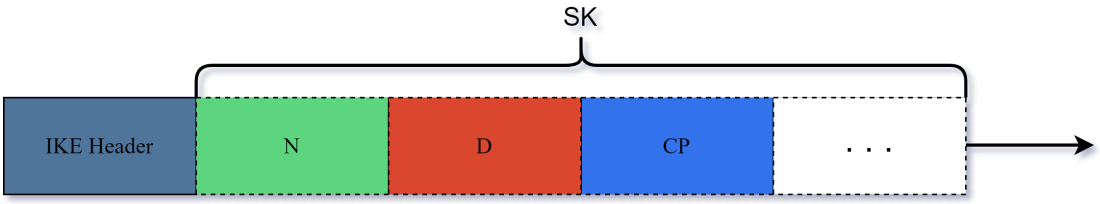

Figure 3.17. INFORMATIONAL exchange initiator

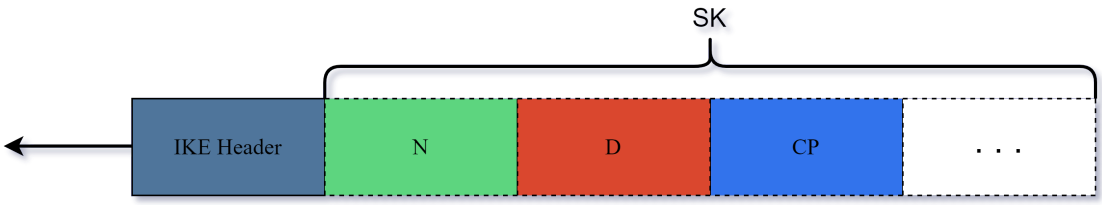

Figure 3.18. INFORMATIONAL exchange responder

Notify Payload

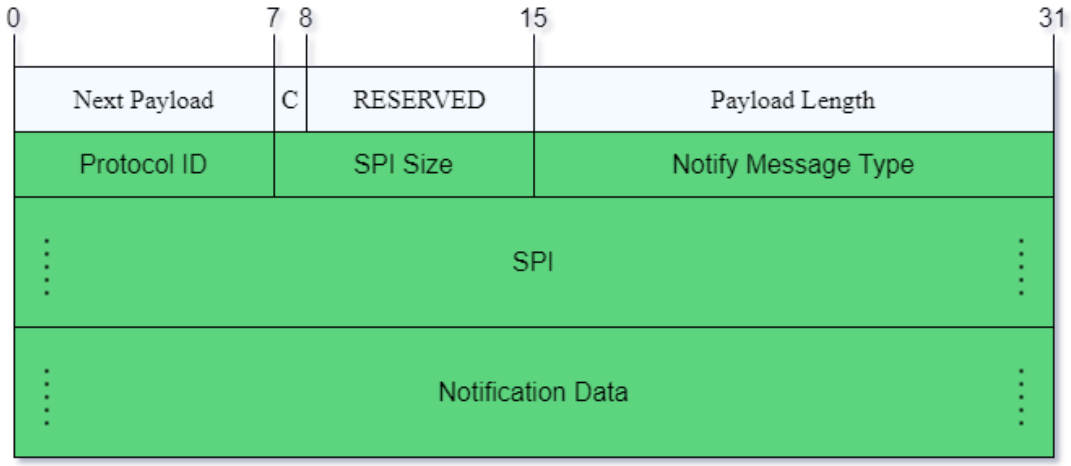

Figure 3.19. Notify payload

This payload is sent when an error occurs and the sender wants to convey the reason why it occurred in the first place. It can be also used to add data about peer's capabilities or, as seen in

paragraph 1.2.3, it can modify how a message should be interpreted.
The Protocol ID field defines the SA type referred by the notification (IKE, AH or ESP). In some cases, there is no SPI included in the payload. When this happens the Protocol ID must be cleared and ignored.
Regarding the actual data conveyed, it depends on the notification type stored in the Notify Message Type field.

Delete Payload

Figure 3.20. Delete payload

As can be easily figured, this payload purpose is to delete an SA. Anyway, there are some constraints about how a Delete payload should be composed. In fact, it is possible to delete several SPIs together but only if they share the same protocol (AH or ESP). If one peer would like to delete multiple SA belonging to different protocols then it should send multiple delete payloads in a single request.
Deleting an IKE SA is different from deleting a Child SA. When a Child SA delete request is sent, then the responder should reply with a delete request for the same SA in the opposite direction. It is clear that since the network could fail to deliver packets, delete requests can lead to halfway open SA. Anyway, a peer should be able to audit its connections and catch these kinds of situations.
When the delete payload targets an IKE SA, instead, every Child SA under that SA must be closed too. Differently from the Child SA delete, the expected reply to an IKE SA delete should be an empty INFORMATIONAL response.

Configuration Payload

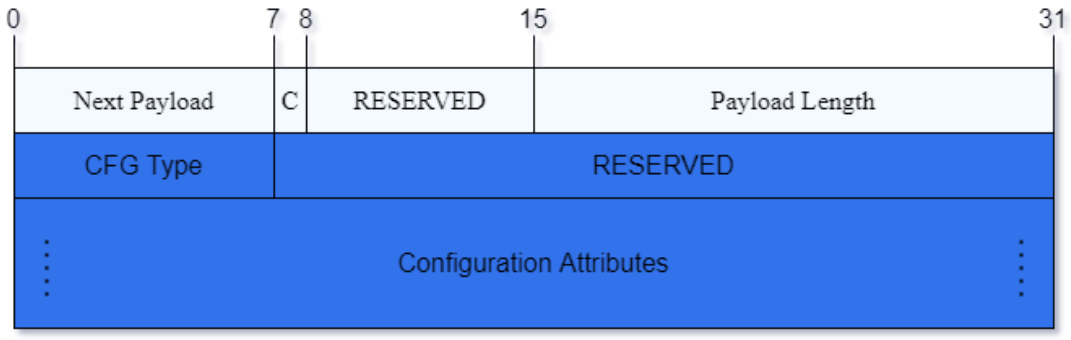

Figure 3.21. Configuration payload

This payload can be used to request an internal IP address as well as obtain other data that would be normally requested to a DHCP server.

3.3 Protocol Variations

There is a great number of protocol variations due to several specific situations and/or possible errors. Listing all of them in detail would be out of the scope and intentions of this chapter so I will highlight only a few of them. For a more in-depth view of specific cases not present here it is recommended to check the related RFC as said before in this chapter introduction.

3.3.1 EAP within IKE

Authentication can be implemented not only by means of shared secret or public key signatures. EAP is an acceptable alternative and, when it is used, it modifies the IKE_AUTH exchange flow. The initiator that desires to use EAP will not add an AUTH payload in its first IKE_AUTH message. Doing so means that identity was claimed (using IDi payload) and never proven. The responder, if able, will reply by adding an EAP payload, without sending SA and TS payloads. After this message, the initiator and responder will exchange data until the EAP will end successfully. Finally, the initiator will send an AUTH payload, which content will be produced using the shared key obtained from EAP and the responder will reply also with a similar AUTH payload followed by its SA and TS payloads. It is possible to use EAP methods that do not produce a shared key but it is not recommended because they are vulnerable to MITM attacks.

3.3.2 NAT Traversal

This paragraph will generally introduce the problem since it is quite often present in common usage scenarios, but since it is quite complex I will just give a quick overview of the most evident variations needed to allow IKE correct functioning when nodes are hidden behind NATs.
Because of how NATs work, IKE and ESP messages must be encapsulated under UDP. Also, since NATs can translate also port numbers, even if IKE packets should be received only by port 500 and 4500, they must anyway be accepted from any port.
IKE_SA_INIT exchange must be modified too. Indeed, both initiator and responder add to their messages two payloads:

- NAT_DETECTION_SOURCE_IP
- NAT_DETECTION_DESTINATION_IP

Main problems that this case rises lie in the TS payloads encoding since address ranges must be correctly handled.

3.3.3 Error handling

To understand how to correctly react to an error in IKE, the first thing to check is the state of the communication under which the error occurred. Different approaches are to be taken if it happens during IKE_SA_INIT, IKE_AUTH, or under an IKE SA not yet established. The rationale behind this approach is that responding to an error could be as useful as dangerous, so choices of whether or not to reply must be taken with caution. There are also several situations where this choice is left to who implements the protocol.
During IKE_SA_INIT exchange, for example, every error leads to creation failure, but the peer receiving these errors should also be aware that these notifications could be forged (in this exchange there is no authentication), so it should not react immediately but keep trying for a bit.
In IKE_AUTH exchanges, instead, errors should lead to a specific notification declaring the authentication failure. In this phase, it is also important which is the peer receiving the error since

after the first message the responder still is not authenticated to the initiator. Anyway, even if the IKE_AUTH is performed correctly, new Child SA establishments could fail. This failure should not lead to the deletion of the IKE SA under which it happened though.
Finally whenever an error is reported outside of an IKE SA, peers should be particularly cautious. If able to audit they should make note of the event and reply only if the error is not marked as a response. Also, they should reply only to a limited number of errors in a certain span of time.

3.4 Usage scenarios

In the first use case, two subnets are being protected by two different endpoints linked by an IPsec Tunnel. In this case, security is transparent to the subnets and only the two endpoints need to implement IKE and correctly route the packets.

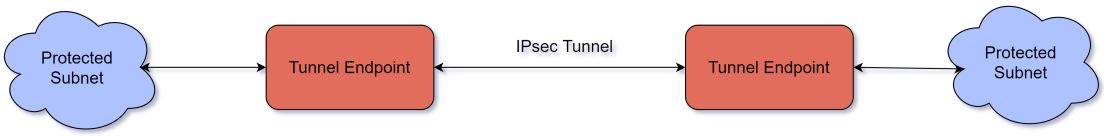

Figure 3.22. Use case #1

The second case represents an end-to-end security implementation. In this situation, of course, each node must implement IKE and then decide the SA to be used for the communication whether it is AH or ESP.

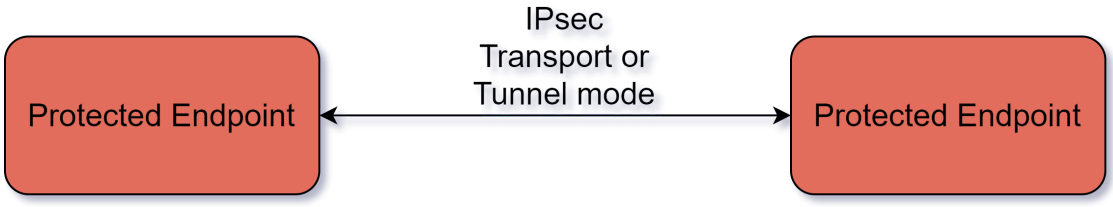

Figure 3.23. Use case #2

This last case is quite interesting and practical since it is usual for a dependent who wants to communicate with their company's network and also exploit its security mechanisms while communicating with the Internet. To be correctly implemented, the protected endpoint should get an address internal to the protected subnet and then route its packet to the endpoint when the public network must be reached.

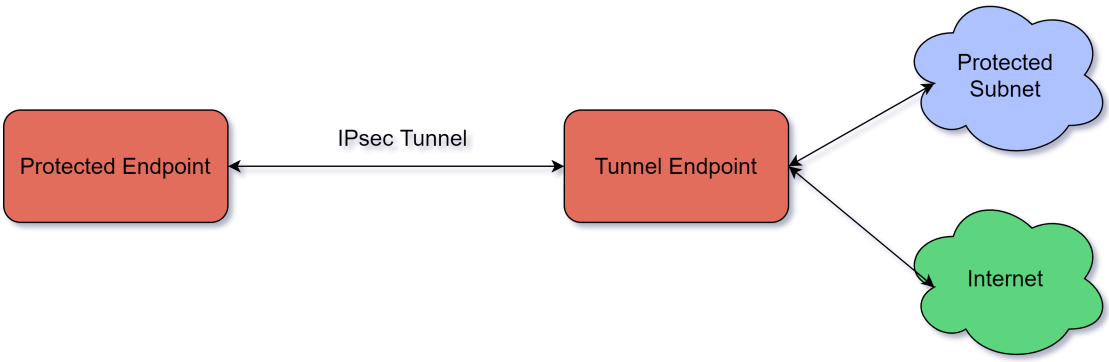

Figure 3.24. Use case #3

3.5 Security mechanisms and vulnerabilities

3.5.1 Replay attack

As previously seen, in the IKE Header there is a value, the Message-ID, needed to match up each request with its response. The same value N can be assigned to two different pairs of messages: the request/response started by the initiator and the symmetrical exchange started instead by the responder. So each peer must store two values: the current Message-ID to be used for the next request, and the current Message-ID to be expected in a request from the other peer. Retransmissions of a request must use the same Message-ID present on the original ones.
Message ID is useful as a protection mechanism from replay attacks since it is cryptographically protected. Anyway due to the fact that each peer is allowed to forget a response, it could receive a request with a Message-ID corresponding to a previous response already forgotten. In this case, it must ignore the request.

3.5.2 DoS

A known attack to IKE is a DoS which objective is to flood a peer with IKE_SA_INIT requests provoking a CPU and state exhaustion. To limit the impact of this attack, when a peer acknowledges an abnormal number of half-open SAs, it should respond to future requests using mechanisms that allow checking whether or not the initiator is using an IP address corresponding to a node really capable to receive packets.
The responder should compose a response that contains only a cookie notification payload (generated in such a way that it can be recomputed in the future without storing it). Then, an initiator receiving this kind of reply should reissue the IKE_SA_INIT request, prepending this time the cookie received. In conclusion, a responder, receiving this new request, will check the cookie and if it is correct, it will reply with the usual IKE_SA_INIT response.

3.5.3 Vulnerabilities

Many vulnerabilities of this protocol lie in how it is actually implemented and trade-offs that nodes owners would like to take. Indeed, while for interoperability's sake many decisions are left to system administrators, it should always be kept in mind how each implementation detail could affect the overall security of the protocol.
A classic example is given by the randomness that should be achieved while generating keys and nonces since it is not defined as a standard pseudo-random function. The same goes for preshared keys that are used while authenticating peers. Maybe an operator would like to derive these keys

from passwords, enabling in this way vulnerabilities to attacks like a dictionary attack and/or a social engineering one.
Another possible issue is caused by the crypto suites and Diffie-Hellman groups that a node is willing to use. It is the administrator's responsibility to constantly update through years the node ability to exploit suites that are to be considered secure.
Administrators must consider also the rekeying frequency for IKE and Child SA, that they are willing to achieve. Also, as seen previously, while creating a new Child SA it is possible to avoid generating newer keys, leaving in such a way all SAs security under the protection of a single key.
While these vulnerabilities are somewhat under administrators' control, the protocol itself cannot for example always ensure secrecy for node's configuration. Indeed, the initiator, or the responder (when EAP is used), must send some data regarding its identity or can learn which kind of certificate the initiator is willing to use, before being authenticated.

3.6 Variatons among protocol versions

In the following table, I gathered main variations among IKEv1 and IKEv2, as described in their respective RFCs, adding a brief description when needed, focusing on differences that can lead to security issues, as well as changes in the workflow which in the second version was optimized to enhance performances.

IKEv1	**IKEv2**
RFC 2409	RFC 7296
Setup is completed by means of two phases (Phase 1 and Phase 2) and can be carried in several different modes depending on the trade-off between efficiency and security that an administrator wants to achieve	Setup is usually done with only four messages, disclosing just some configuration data
No reliability	Reliability is ensured to a certain extent by means of retransmission timers as well as message identifiers which pair up each request with its reply
No protection against DoS	Cookie mechanism to protect against IKE_SA_INIT request flooding limiting CPU and memory usage
No alternative authentication methods supported	EAP is supported
Negotiation is complex and leaves no freedom to peers	Negotiation is more organized and hierarchical structured. Peers can individually choose parameters like connection lifetime

Table 3.1. IKEv1 vs IKEv2

Chapter 4

TLS protocol

Cybersecurity does not only face challenges regarding attacks against individual nodes. Indeed, since nodes need to be connected to each other to exchange data and provide services, it is a cybersecurity interest to offer strategies to protect data while it is transmitted from one node to its peer. Moreover, since you usually do not own or control any channel between two peers, proposed solutions must assume that the channel itself is not safe [16].
Protocols such as TLS solve this issue by establishing a secure communication channel between peers. Channels can be defined as secure if the following security properties are guaranteed [17]:

- Authentication: server authentication is mandatory, while the client's one is optional. It can be conveyed using asymmetric encryption, digital signature or symmetric encryption (using PSKs).
- Confidentiality: data exchanged is encrypted. Data length is not hidden by the protocol, but the user can pad data in order to do so.
- Integrity: data cannot be modified without leaving a trace, so an attacker cannot modify packets remaining unnoticed.

TLS protocol establishes secure communication channels by means of two sub-protocols:

- TLS handshake: it is the first to be executed. Its role is to authenticate peers (client authentication is optional) and negotiate cryptography suites, as well as relative attributes. It is designed to be correctly completed even while communication is under attack;
- TLS record: when a handshake has been correctly completed, traffic data is split and encapsulated using records. Each of them is protected using negotiated crypto-suites and keys.

TLS is designed in such a way that allows it to encapsulate and therefore protects protocols belonging to a higher layer. HTTPS (HTTP over TLS) is a common example widely used today.
TLS protocol counts several versions, and while version 1.3 is the one that will be discussed in this chapter, it must be underlined, though, that its previous version (1.2) is still commonly used. Version 1.3 was preferred since it introduces crypto-suites that allow using PSKs with no certificate-based authentication mechanism.
While TLS versions are not compatible with precedent ones, they share some mechanisms that allow peers to negotiate lower versions if needed. Major differences among 1.2 and 1.3 versions are collected in tables 4.1 and 4.2.

TLSv1.2	TLSv1.3
contains legacy algorithms	legacy algorithms have been pruned, remaining ones are all Authenticated Encryption with Associated Data (AEAD) algorithms
negotiated cipher suites algorithms are to be intended both for exchange keys, authentication and record protection. Same stands for hash functions that are shared by key derivation function as well as handshake message authentication code (MAC)	cipher suites are designed in such a way that peers can negotiate different algorithms for authentication, exchange keys and record protection. Also, hash functions can be differentiated whether it is intended for the key derivation or MAC
no 0-RTT mode available	introduced possibility to use a zero round trip mode (0-RTT) trading some security properties off for enhanced performance

Table 4.1. TLSv1.2 vs TLSv1.3 (source: [17])

TLSv1.2	TLSv1.3
	key derivation function and handshake state machine were redesigned to allow easier analysis and, for the latter, to prune superfluous messages
no state-of-the-art crypto-suites	Elliptic curve algorithms and Elliptic curve signature have been added to base specification. Point format negotiation was removed. Static RSA and Diffie-Hellman suites were removed in favour of suites providing forward secrecy. RSA Probabilistic Signature Scheme, as well as DSA and custom ephemeral Diffie-Hellman groups, were added.
	PSK-based mechansims substituted by a new single PSK exchange

Table 4.2. TLSv1.2 vs TLSv1.3 (source: [17])

4.1 TLS handshake

TLS handshake tasks are to negotiate protocol parameters such as its version and crypto-suites to be used. Moreover, the key material is exchanged to establish keys to be used to protect traffic data. There are three available modes to exchange keys: DHE/ECDHE (Diffie-hellman over finite fields or elliptic curves), PSK with DHE/ECDHE and PSK-only.
As you can see in figures 4.1 and 4.2, TLS handshake consists of three phases:

- Key exchange: in this phase key materials are exchanged and cryptography parameters are set. It starts when a client transmits a ClientHello containing a random nonce, protocol version desires, symmetric algorithms and hash functions preferred and finally key material that could be a list of PSKs id or key material for DHE or both.
  The server must reply with a ServerHello where it conveys to the client which algorithms and hash were chosen. It also sends key material to complete key establishment. Both client and server exploit specific extensions to transmit key material: if DHE is in use, then client and server messages will carry key_share extension; if instead PSK was selected, then you will find appended pre_shared_key extension. Finally, if the client and server negotiated PSK with DHE as an exchange mechanism, then both these extensions must be present.
  Since the key exchange is completed and keys have been established, any following message will be encrypted. It must be underlined, though, that TLS derives two different keys: one used to encrypt handshake traffic, and another one to encrypt application traffic;

- Server parameters: as soon as ServerHello is sent, the server transmits two messages. Former is an EncryptedExtensions message, used to establish some parameters not related to cryptography algorithms; latter is a CertificateRequests message, sent to request client authentication, specifying preferred certificate parameters. This last message is optional since in TLS client authentication is not mandatory;

- Authentication: finally, during the last phase, the server and optionally client will authenticate themselves. Authentication is provided by means of three messages: Certificate, CertificateVerify and Finished. Following messages descriptions stand true both for server and client.
  The certificate message contains the actual certificate provided by the authenticating node. This message may not be sent since it is allowed to provide authentication with no certificate. If a certificate is indeed sent, then a CertificateVerify must be transmitted. Inside this message, it is stored a signature over all handshake messages, signed using a private key corresponding to the public key tied to the certificate previously sent. Of course, if authentication is not provided by means of a certificate, either CertificateVerify should not be transmitted.
  Finally, the Finished message is sent. It contains a Message Authentication Code (MAC) over the whole handshake. So, it provides key confirmation, identity binding and, if the PSK mechanism was chosen as an authentication system, then it also serves as an authentication tool.

When the authentication phase is correctly completed, the client and server can derive their keys to encrypt application data and therefore secure communication may start.
I will now introduce more in-depth extensions related to PSK modes since those are the ones that QKD can exploit to provide quantum security to TLS (without modifying protocol workflow).

4.1.1 Pre-shared key exchange mode

Whenever a client intends to use PSKs, it always must send this extension. The server, instead, must never reply back using it. This extension is meant to be used to declare how the PSKs are going to be used by the client. Actually, only two possible modalities are provided:

- psk_ke: in this case no key derivation values are transmitted and PSKs must be used as is;

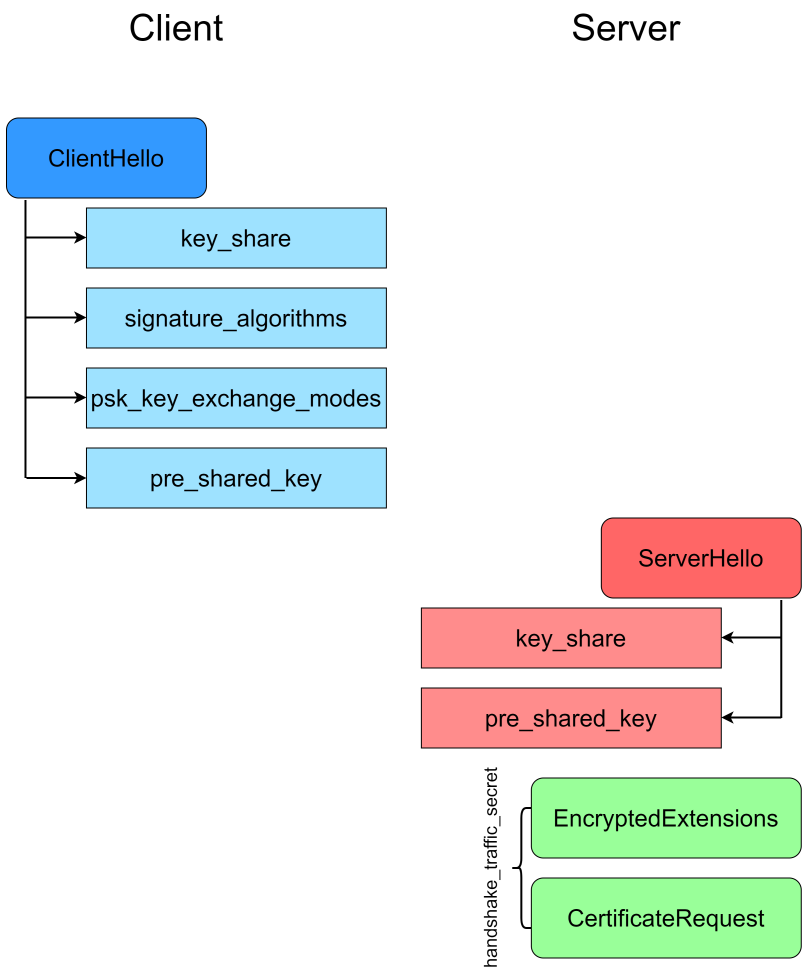

Figure 4.1. TLS handshake - key exchange and server parameters phases (source: [17])

- psk_dhe_ke: in this case key derivation values must also be sent and PSKs are used to derivate actual shared secrets.

It must be underlined that whenever a PSK is used as-is (first case), there is no forward secrecy guaranteed.

4.1.2 Pre-shared key extension

This is the actual extension used to negotiate the PSK to be used. It contains the following fields:

- identity: a label for a key obtained externally;
- obfuscated_ticket_age: a value expressing the age of the key. For keys obtained OOB, it should be 0;
- identities: list of identities the client desires to negotiate;
- binders: list of HMACs, one for each identity;
- selected_identity: the actual identity that the server selects.

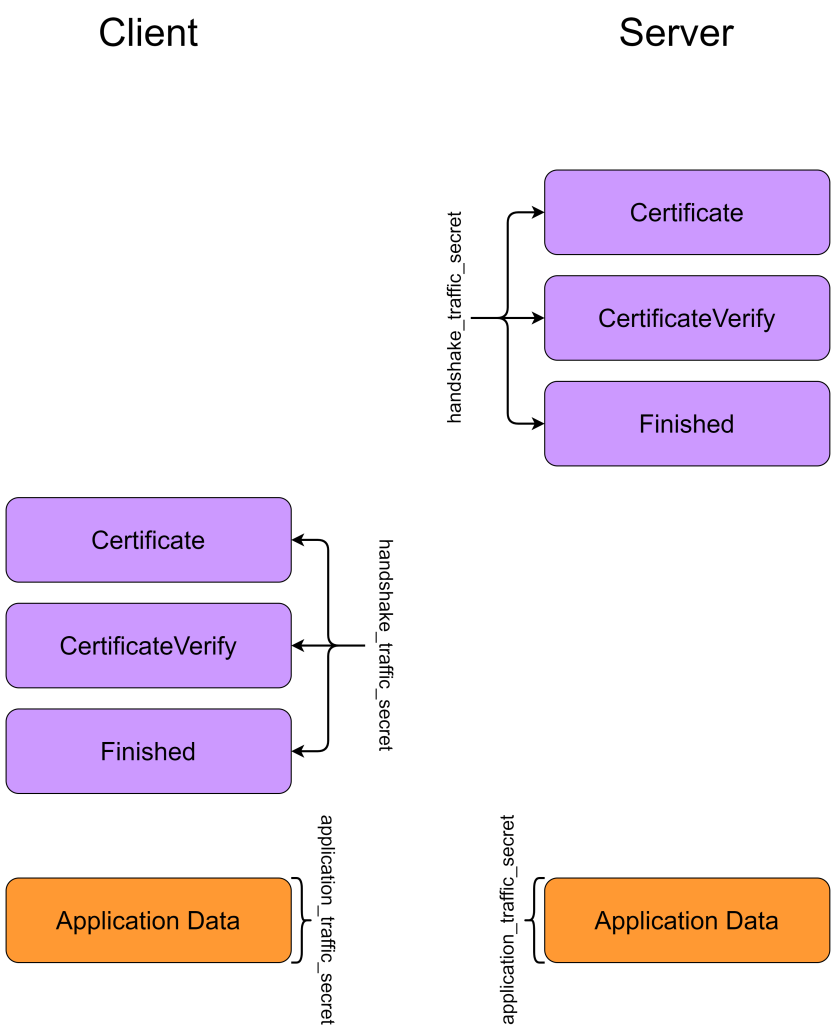

Figure 4.2. TLS handshake - authentication phase and communication start (source: [17])

Each PSK is linked to a Hash algorithm and, in particular, in the case of external keys, if it is not defined, it defaults to SHA-256.
There are also some constraints that the server and client must check. Regarding the former one, it must select one identity and then check its binder validity. If this test fails, the whole handshake must be abort. The client, instead, must verify that the selected identity belongs to the range previously offered, that the cypher suite selected comprehend a hash associated with the PSK, and finally that eventual key_share extension presence is coherent with the modality requested in the pre_shared_key_exchange_mode extension.
Both client and server must always also check that the pre_shared_key extension is the last extension sent along with the Client/Server Hello.

4.2 TLS Record

TLS Record protocol role is to manage data to be sent, splitting it into data chunks, called records, and protect those records according to parameters established during the handshake. It does make difference among four available record types: handshake, application data, alert and change_cipher_spec. This latter one is used only to negotiate former TLS protocol versions.

Record protocol workflow consists of a first phase where data is split into TLSplaintext records, and then into a second one (optional for some records) where these TLSplaintext are encrypted thus being translated into TLSciphertext.
Starting from the first phase, each record type comes with a set of rules that should be followed:

- handshake record: data regarding handshake can be collected into a single record or, if needed, it can be split among several records. Anyway, there must not be any record labelled with a different type between handshake records. Moreover handshake records with zero length must not be sent. Finally, a handshake record cannot span a key change. Indeed, users should check that all records preceding a key change align with record boundaries. If that is not the case then the connection must be dropped;
- application data record: application data can be inserted in a single record, as well as be split among different ones. It must always be protected (then it must always contain TLSciphertext) and there can be application data records with zero length. Because of how TLS is designed (to be exploited by higher layer protocols), application data is opaque to TLS;
- alert record: differently from previous record types, alert records must not fragment and every single record must contain data regarding only one alert message.

To provide protection against replay attacks, each record contains also a nonce. This value is generated starting from a sequence number and an Initialization Vector (IV). Each node, indeed, whenever a new communication is established, keeps track of two sequence numbers, respectively corresponding to incoming messages and outgoing ones. In the unusual case where this number should wrap (they are stored as 64-bit values), a rekey must be issued.
Both server and clients, when the first phase is completed, encrypt their records translating TLSplaintext into TLSciphertext.
As shown in the table 4.1 In TLSv1.3 all available algorithms are AEAD, therefore authentication and encryption are provided by means of a single operation. This process needs as input a key (established after handshake), a nonce, a plaintext and a record header.
While AEAD algorithms increase record lengths by different amounts, it is allowed to insert padding to obfuscate clear length differences that could be exploited by an attacker to gain knowledge regarding the occurring communication. It is also possible to send zero-length application data records and add a pad to them, whenever communication presence or absence is also considered information to be protected.
Finally, depending on the algorithms negotiated to protect records, it must be considered how much plain-text can be encrypted under a certain key without compromising its security, and eventually issue a rekey if desired security degree threshold cannot be guaranteed.

4.3 TLS Alert

Error and closure report messages are managed in TLS as alert records. It is also defined how peers should react to all possible alert records, providing in such a way another sub-protocol tied to TLS. Each alert record contains a description of the event that occurred, as well as a field where the severity level is declared. This field, though, is obsolete in TLSv1.3 and it is kept only for compatibility reasons. Indeed, in TLSv1.3, there is a severity level tied to each available alert so this legacy field can be simply ignored.
TLS alert protocol defines two main alert types and how to behave when they are received:

- Closure alerts: these kinds of alerts are sent to communicate that the sender has no more data to send and therefore the connection can be closed. This information must be correctly handled since otherwise, peers could be subject to truncation attacks. In TLSv1.3 it is possible to receive a close_notify or a user_canceled alert. The former means that all data is sent and no more data will be transmitted from the peer who sent this alert. Latter one, instead, is used to signal that one peer dropped the connection for reasons not related to

protocol errors.
Peers receiving close_notify alert should ignore new data records incoming and, if they are sending new records, they should not drop them (how it was instead defined in prior TLS version);

- Error alerts: TLSv1.3 defines several error alerts, all to be considered fatal. Error management in TLS is straightforward: whenever a fatal error alert is sent, both peers must drop the connection.

4.4 Security considerations

Since TLSv1.3 is relatively new, not much effort was done to research vulnerabilities, and since it is not widely implemented, there are no examples of new known attacks. Then, while considering security issues right now, it is suggested to focus research effort to limit possible vulnerabilities that may enable downgrade attacks. Indeed, an attacker, after a successful downgrade attack, could exploit known vulnerabilities related to prior versions.
Anyway, TLSv1.3 introduce some new vulnerabilities since it allows users to use PSK crypto-suites. Therefore, all known vulnerabilities related to this mechanism are introduced in TLS:

- Forward secrecy: using PSKs, with no DH exchange, will result in communication where forward secrecy is not guaranteed. Then, if the PSK used will be compromised, every preceding conversation could be decrypted;
- Brute-force and dictionary attacks: PSKs that are too short or human-generated are vulnerable to brute-force and dictionary attacks. These kinds of attacks can be performed both off-line (on a captured TLS packet), both online (attempting to connect to a server). Moreover, when PSKs are used as is, without DH exchanges, an attacker should simply eavesdrop on the handshake to get all data needed to perform its attacks. Instead, if DH exchanges have been conducted, an attacker must also trick the client by impersonating the server;

4.5 TLS-PSK use cases

While TLS is still widely implemented exploiting public-key algorithms, as we said before TLSv1.3 introduces new suites requiring only PSKs. Such novelty in TLS latest version proves how much interest PSKs suites rose during the last decade. Indeed, new technologies involving devices characterized by low computational power, such as the Internet of Things (IoT), exploit PSK to provide security, avoiding overheads related to certificate-based crypto-suites.
Moreover, TLS-PSK is providing new possibilities to obtain quantum-safe security free or at almost no cost. In fact, TLS may face quantum computers threat exploiting PSKs generated by means of QKD.
At the time of writing, two major proposals have been made:

- An extension to TLSv1.3 that would allow combining certificate-based authentication with PSK-authentication. This design would allow an easy way to improve overall security while maintaining older infrastructures not considered quantum-safe. To enable this extension, some minor change has to be made to the standard protocol, though. So at cost of some effort to implement this extension, it is possible to provide a higher degree of security without dropping well researched and known public-key suites [18];
- straightforward use of QKD generated PSKs using TLS-PSK crypto-suites. This proposal does not need any extension since it exploits standard TLS features. At cost of its simple implementation, it drops certificate-based suites, meaning a major change for actual systems. This is the proposal chosen to be tested in this thesis and thus it will be introduced more in-depth in the following chapters [19].

Chapter 5

Cloud computing and OpenStack

5.1 Cloud computing

Cloud computing is an abstract concept that includes several possible definitions. It could be pragmatically defined as a group of hardware resources that are virtualized and then offered to the user who can exploit them in several ways. In particular, depending on the degree of abstraction of these resources, three main possibilities are commonly offered [20]:

- Infrastructure as a Service (IaaS): it means the on-demand offering of computing resources, like processing or storage, to multiple clients. Usually, these resources are virtualized and can be accessed by users by means of related API. Users that interact with these layers must manage storages, nodes and networks that link them. Anyway, since these resources are virtualized, they only interact with abstract nodes without having to understand which is the actual underlying hardware;
- Platform as a Service (PaaS): this, instead, is a service offered usually to developers to allow them to develop their applications on the desired platform, without the need to actually manage the infrastructure itself. It is also particularly useful since it can provide handy services like automatic up-scaling as the usage of the application grows;
- Software as a Service (SaaS): it is the most known layer since it is the layer offered to the end-user. In this case, users do not know anything about the underlying infrastructure or even the platform where the applications run. They can simply access applications services by means of clients such as web browsers. A typical example of SaaS could be Google applications like Google Mail and Google Docs.

This technology is here introduced because, in this thesis, cloud computing networks were picked as a possible use case where implement and test QKD. In particular, OpenStack, an open-source IaaS, was researched for this purpose. Moreover, to enable future researches on the topic, the other two technologies, commonly associated with cloud computing, will be briefly discussed: Network Function Virtualization (NFV) and Software Defined Network (SDN). Indeed, they can enhance both the security and performance of a cloud network [21], while also blending with networks exploiting Quantum Key Distribution [22].

5.2 Software-defined network

Software-defined networks are networks where the control plane is decoupled from the data plane and is dynamically managed by means of specific software. It is the possibility to configure forwarding rules at run-time that particularly rose the community's interest.
SDN networks architecture is defined by three layers:

- Application Layer: This layer collects several possible applications that communicate with the underlying control layer (such as security, QoS, etc.);
- Control Layer: the core of SDN, is a centralized controller which keeps a global view of the underlying network and opportunely manage network devices as a response to applications requests;
- Data-plane Layer: the collection of all actual network devices, such as routers and switches, which must be programmable and must offer standard interfaces.

Moreover, to enable this architecture, there must be two interfaces that link these three layers: the Northbound interfaces which allow communications between the application layer and the control layer; and the Southbound interfaces, which enable the control layer to manage network devices [23].

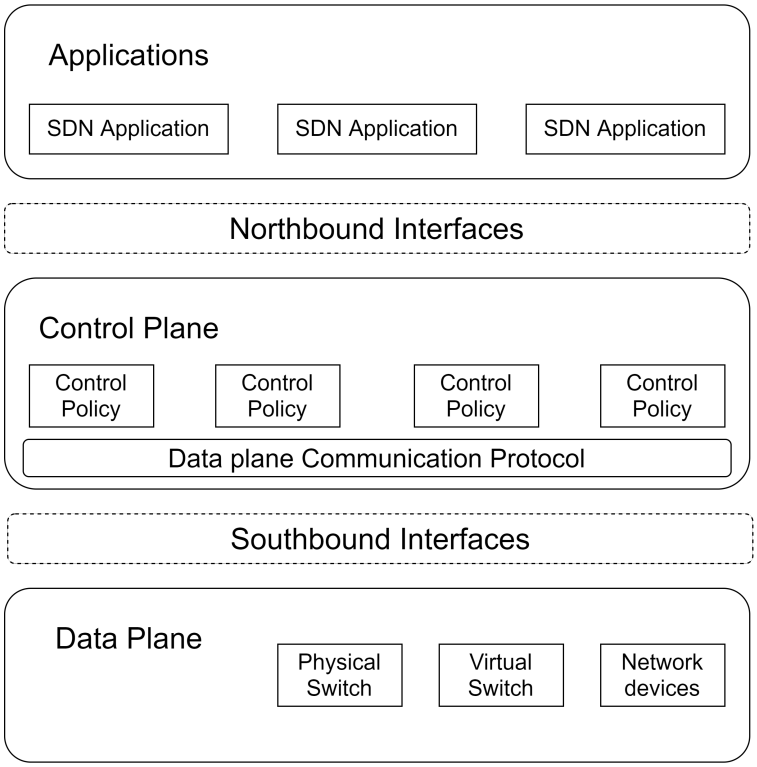

Figure 5.1. SDN architecture (source: [23])

SDN security vulnerabilities already have been discussed in some papers, and their major issues are related to the SDN framework itself. They are subdivided due to how they affect in different ways layers and interfaces provided [24]:

- Unauthorized access: unauthorized controller access can affect layers from controller to data, while an unauthenticated application ranges from application layer up to controller's one;
- Data leakage and/or modification: data layer is vulnerable to attackers who may apprehend flow rules and forwarding policy in order to analyze data or to modify it. This kind of vulnerability impacts layers between control and data;
- Malicious application: attackers may insert fraudulent rules to affect application and controller layers or they also may hijack the controller to impact on layers from the controller's up to data;

- Denial of service: achieved by means of controller-switch communication flood or switch flow table flooding;
- Configuration issues: they mainly regards error in security policies enforcement as well as lack of any authentication technologies such as TLS.

5.3 Network Functions Virtualizations

The idea behind Network Functions Virtualizations is to decouple network functions from the special purpose hardware devices usually needed. Doing so it is possible to perform these functions by means of software that can run on standard IT devices. This framework allows then to obtain an elastic network where functions can be deployed on devices that can execute them, reducing in the meanwhile costs related to dedicated network hardware.
ETSI defines the NFV framework as a Network Functions Virtualization Infrastructure which is able to execute all the NFVs needed. The NFVI is defined as the collection of hardware resources that are virtualized and that can be accessed by VMs. Usually to each VM is associated with an NFV, and all of them are orchestrated by means of an NFV Orchestrator [23].

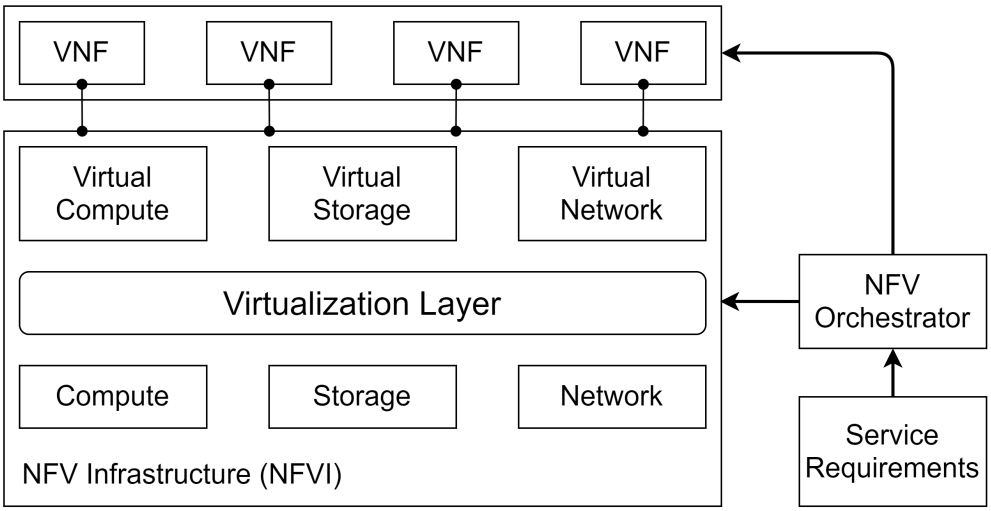

Figure 5.2. NFV framework (source: [23])

Similarly to SDN, NFV security issues arise from its infrastructure, as well as from its components which, even if virtualized, present vulnerabilities similar to their physical counterparts. Since the scope of this thesis is to present NFV associated with IaaS networks, the following vulnerabilities are to be referred to contexts where NFV is implemented over IaaS [25]:

- VMs security issues:
  - Infected images: virtual machines are usually stored as images ready to be distanced on demand. An attacker may for example modify or replace some of these images with infected ones, obtaining so easy access into VMs and possibly to the rest of the system as well;
  - VM migration: to optimize system performances, often VM migration is used to relocate VMs from one node to another. This transmission represents a vulnerability since it could be sniffed by an attacker who may simply gather data or even tamper the VM;
  - VM hopping and VM escape: an attacker who got access to VMs may exploit vulnerabilities or run malicious code to get access to another VM (hopping) or even to reach and control the underlying host (escape);

  - VM DoS: due to performance reasons, VMs share resources of the host, so attacking a VM by means of a DoS could eventually exhaust the host's resources compromising also all other VMs that share that host;

- Hypervisor security issues:
  - Hyperjacking: even if it is rare and difficult to achieve, hyperjacking is a dangerous possible attack that can be performed against this framework. Attackers can achieve it by means of three different strategies: inject a rogue hypervisor beneath the original one; run a rogue hypervisor on top of it, or finally take control of the original hypervisor;
  - Breach of isolation: theoretically VMs should be isolated from each other and should use only the amount of resources granted. Bad configurations though can lead to possible VM escape or DoS attacks;

- Insecure management interfaces: even if, as previously said, ETSI introduced a standard infrastructure, interface designs and implementations are not described. So, poor interface design, for example, could lead to events where attackers may exploit them and gather users' data;

- Virtual network components: they are both vulnerable to software vulnerabilities (since they are virtualized), both vulnerable to all standard attacks known against networks (IP spoofing, ARP spoofing, DoS, etc.);

- Security policies: security policies enforcement is crucial for the whole safety of the infrastructure and as they already are complex to be achieved correctly in a standard environment, they represent an even more challenging task in a context like NFV. Indeed, in this case, security policies are usually defined by administrators that are offering the infrastructure as a service to several clients, and each of them may require a different degree of performance and security. Bad configurations may then result from possible misunderstandings, leading to new vulnerabilities;

- Service composition: NFV offers service composition, a feature that on-demand compose several services gathered from several providers. To achieve this safely, components should authenticate each other beforehand. Usually, these components dependencies are complex and resulting trust schemes are quite twisted, so bad configurations are not so unusual. Finally, each component vulnerabilities could put the whole service at risk;

- Malicious insiders: even if it is not the most common case, attacks can be performed from the inside. Administrators, indeed, could breach users privacy, thanks to their access to the hypervisor, and gather, for example, data like users ID and SSH keys.

5.4 SDN and NFV

While SDN and NFV are different concepts that aim to solve different challenges, they can be implemented concurrently obtaining higher efficiency [23]. Indeed, NFV can abstract the SDN controller as a virtual network function and dynamically migrate it to optimal locations, while SDN improves connectivity efficiency among VNFs thanks to the programmable network that it offers.
The resulting architecture is composed of a control module, an NFV platform and forwarding devices:

- The control module consists of an SDN controller, which is in charge of determining and enforcing packet forwarding logic; and an NFV orchestrator which offers a northbound interface to the SDN controller, and is in charge of managing VNFs deployed on the NFV platform.

- NFV platform is the collection of commodity servers where VMs associated with VNFs are hosted.

- Forwarding devices are programmable network devices that link VMs belonging to the NFV platform and their routing tables are dynamically managed by the SDN controller.

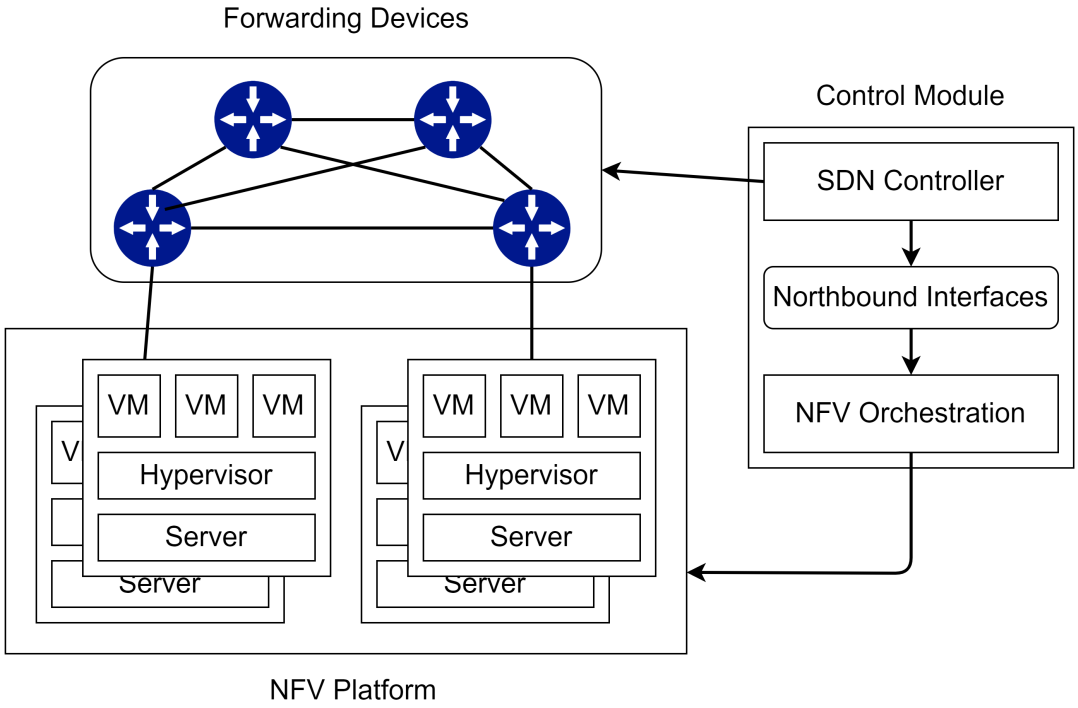

Figure 5.3. Software-defined NFV system (source: [23])

There are several advantages that an SDN-NFV network can offer over a traditional one also when we are talking about cloud computing networks [21]:

- Network management centralization provided by SDN can provide the possibility to offer several Virtual LANs over a physical LAN, as well as providing, as usual, greater efficiency in data forwarding;
- Complexity introduced by virtualization in network management is eased by SDN that can enforce opportune security and information policies. As a result network functions, such as firewalls, can be more easily managed;
- The ability to dynamically control the data plane offers the possibility to achieve greater results for applications that may require a certain QoS (VoIP, multimedia transmissions, etc.);
- NFV allows to further improve efficiency gained exploiting SDN, reducing in the meanwhile costs needed to manage the actual infrastructure.

5.5 OpenStack

In July 2010 Nasa and Rackspace Hosting launched OpenStack as a project whose aim was to offer an open-source cloud computing platform for both public and private clouds. In particular, it offers IaaS by means of different open source projects that provide several services. OpenStack let users build their clouds, letting them decide how each service is deployed and how the network among nodes is mapped [26]. It is thought to be easily scaled horizontally and it allows each user to decide which service is needed, avoiding deploying superfluous ones.

Openstack provides the administrator with the possibility to enable TLS communications over all their networks. Anyway, to limit excessive overheads, some possible configurations are recommended [27]:

- SSL*n*TLS proxy in front: in this case, there is a proxy in front of the API endpoint. Communication is secured up to the proxy and then transmitted in clear afterwards;
- SSL*n*TLS on same physical hosts as API endpoints: similar to the previous case but this time TLS proxy is hosted on the same node hosting API endpoint;

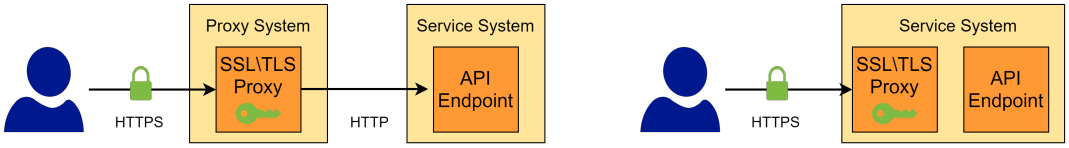

Figure 5.4. SSL \TLS with proxy in front, or on same physical host (source: [27])

- SSL*n*TLS over load balancer: architecture is equal to the previous case but this time there is a load balancer in front of it. Doing so packets can be analyzed for routing purposes without decrypting it, increasing, in such a way, system efficiency;
- Cryptographic separation of external and internal environments: this last case is the most complex one. It differs from the last case about how data is handled. Indeed, the load balancer system now decrypt packets using its certificates (used by external users) and encrypt them before sending them to endpoints using inner issued certificates.
  Doing so, data is encrypted both outside the cloud both inside it, up to endpoints. This higher security, however, has a great impact on system performance.

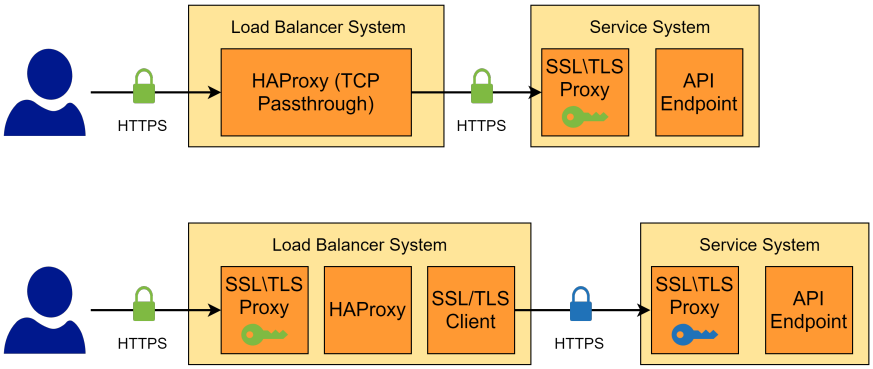

Figure 5.5. SSL \TLS over load balancer, or cryptography separation (source: [27])

OpenStack handles computing, network and storage resources through several components, one for each service. Some of those services will be now briefly introduced citing only main services that offer core functionalities, and some optional services that are of interest in this thesis:

- Nova: OpenStack's core, Nova is a distributed application that automatically manages computing resources and provisions VMs instances. It uses no virtualization software while it exploits the existing hypervisor such as KVM;

- Neutron: while initially, it was Nova to manage OpenStack's network component, after some years this role was decoupled from Nova in favour of a new service called Neutron;
- Glance: this service works as image storage, mainly needed to provide images for VM instances that are deployed on the platform;
- Cinder: similarly to Neutron, also this service manages a role previously carried out by Nova. Indeed, it serves as block storage and manages volumes that are linked to VMs;
- Horizon: this service is OpenStack's dashboard. It offers users and administrators a GUI to access, configure and launch VMs. Through the dashboard, they can also configure networks, as well as network components.
- Keystone: this is the identity service. It manages the database where users and their credentials are stored;
- Tacker: based on ETSI MANO, it provides a generic VNF Manager and a VNF Orchestrator to manage and offer Virtual Network Functions;
- Octavia: born as Neutron extension, it was then decoupled and moved to a standalone project. It offers load-balancing services deploying on-demand specific VMs (usual amphorae) which can automatically handle incoming requests.

Since OpenStack is composed of so many services, that all need appropriate configurations to work properly, its deployment is quite challenging. In order to ease this process, the community developed some tools to automate it. Thanks to them, administrators can now deploy a fully functioning OpenStack cloud in a matter of hours or even less.
For this thesis, indeed, OpenStack Cloud, used for tests, was deployed using Kolla-Ansible. It allows deploying an OpenStack cloud where each service has been containerized.

5.5.1 OpenStack networking - Neutron

OpenStack solves every need regarding networks setup and management through its service called Neutron. Indeed, it provides a feature that let users create complex virtual network topologies, such as several network types, subnets or routers.
In particular, it allows to set up an external network and several internal ones. Former one is a slice of the actual physical network, external to OpenStack installation, the latter, instead is a fully virtual network, not accessible from the outside without proper configurations. Moreover, Neutron provides three different types of networks, depending on connectivity or services they offer: provider networks, routed provider networks and self-service networks.
A standard OpenStack networking architecture contains four different node types, linked by different networks. As you can see in the figure 5.6, there are a dashboard and a cloud controller node that are reachable from the Internet and offer API to allow requests. Moreover there is at least one network node and one compute node, where are respectively hosted neutron and nova main components. These latter nodes are linked by means of a guest network, used by VMs to interact with each other, and they are also linked to all other nodes by means of a management network that offers connectivity to OpenStack services. Finally, the Network node is linked to the physical network through the external one, providing Internet connectivity to VMs that are spawned there, or even VMs linked to the guest network that owns an assigned external IP (known as floating IP). It is possible to deploy each of these nodes on a different physical host or deploy them on few hosts. They may even be deployed all on a singular host. This latter is the actual implementation modality that was chosen to conduct tests as you will see in later chapters.

Neutron is composed of three main components [29]:

- API server: layer 2 connectivity is provided through this component. While standard networks and routers are already provided, it is possible to extend this capability due to several plug-ins that can allow interoperability also with modern new technologies such as SDN controllers. Indeed, in the OpenStack environment, a standard serviceable to introduce the SDN

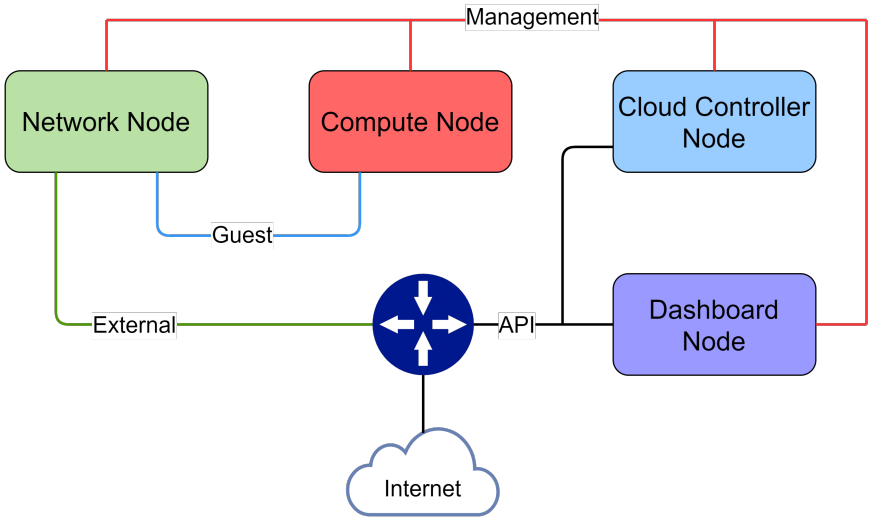

Figure 5.6. Standard OpenStack architecture (source: [28])

concept does not exist yet. From a theoretical point of view, to implement SDN, an SDN controller should be placed between Neutron and the underlying hardware infrastructure. Any needed change to the network would be requested from ad-hoc neutron extension to the SDN controller. Then the controller would be responsible to reach and manage network components as well as VMs.
Communication between Neutron and SDN controller should be provided by the Northbound API, while the SDN controller could reach hardware and VMs by means of the Southbound API. Several plugins are able to allow interactions between Neutron and an SDN controller that already exist such as Open vSwitch, Cisco UCSlNexus, Linux Bridge, Modular Layer 2, REST Proxy Plugin, RUY. The same still stands for communications between the SDN controller and Neutron (OpenDaylight, Ryu, and Floodlight, NOX) [30];

- OpenStack Networking plug-in and agents: create networks, subnets and manage ports and IP assignment. There are several possible plug-ins that can be chosen but they can be used only one at a time;

- Messaging queue: this is needed to correctly deliver messages from the Neutron server to its agents and vice versa.

Neutron also allows administrators to enforce firewall rules by means of security groups. These can be linked to VMs to define which kind of traffic must be forwarded to or from them [29]. VPNaaS is also offered as a Neutron extension, that administrators may decide to deploy. This extension provides the only site to site VPN and asks indeed to define a virtual router as a VPN endpoint. Messages coming from VMs linked to the private network behind this router will be protected as long as they are directed towards another virtual network that is linked by an active IPsec site to site connection. Several configurations options are provided while configuring this connection, in particular administrators are able to set also IKE version to be used (v1 or v2) [31].

5.5.2 Loadbalancing as-a-service - Octavia

As said before, in OpenStack LB-as-a-Service (LBaaS) is offered by Octavia and its amphorae. It should be anyway underlined that Octavia allows using any image as an amphorae substitute, as long as all requested interfaces have been implemented.
It will be introduced briefly now Octavia's architecture, as well as some of OpenStack's services

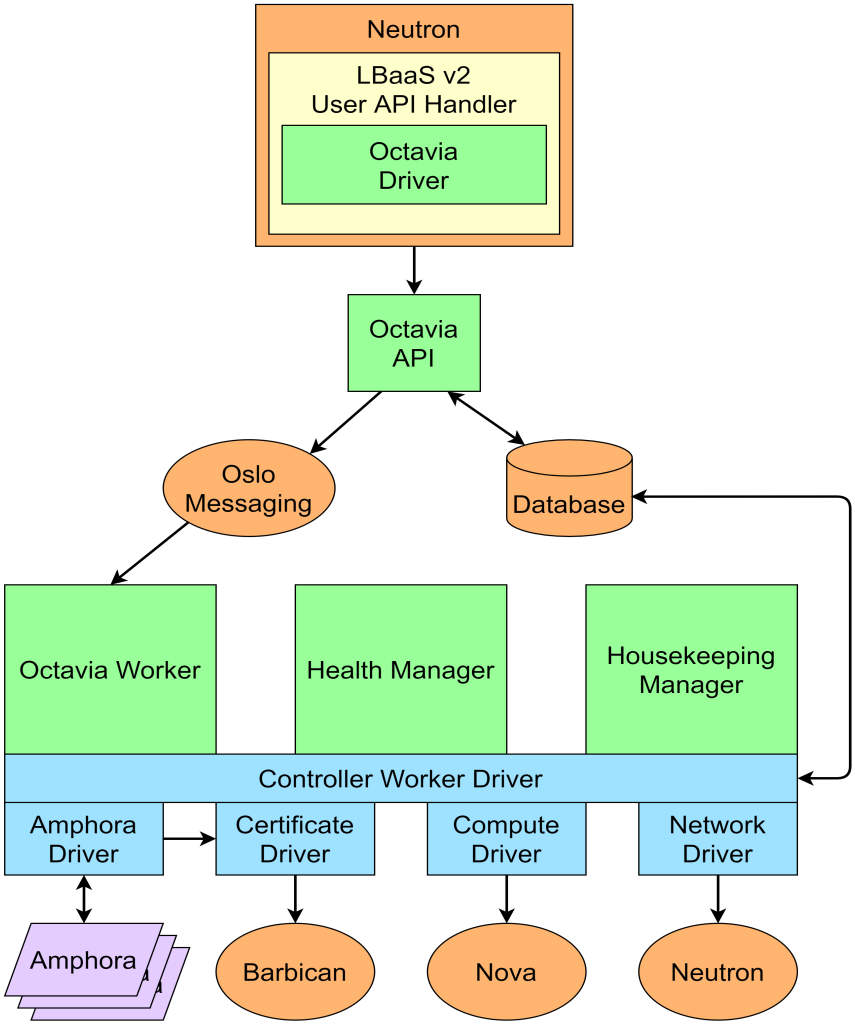

Figure 5.7. Octavia infrastructure (source: [32])

which were neglected in previous chapters, but are needed for Octavia to work properly.
All needed services are:

- Nova: to manage amphorae and computing resources;
- Neutron: to provide network connectivity among amphorae, tenant and external networks;
- Barbican: needed only when TLS communication is configured, it serves as storage for TLS certificates and credentials;
- Keystone: to manage authentication against Octavia API, as well as to authenticate Octavia with other OpenStack services;
- Glance: to store amphorae images;
- Oslo: to enable communication among Octavia controller components. Moreover, Octavia uses intensively Taskflow to manage back-end configuration.

It is crucial to know that Octavia is designed to interact with each of these services by means of a relative driver. This design principle allows administrators to easily substitute these services with custom ones when desired, without modifying Octavia itself. Here are the major Octavia components:

- amphorae: virtual machines whose role is to provide actual load-balancing services to tenants environments. In Octavia 0.8, an ubuntu virtual machine running HAproxy is the standard amphorae reference;
- controller: it is the service manager, composed of four components (that are daemons), which can run on separate back-end infrastructure:
  - API Controller: as it can be easily imagined, this component takes in API requests, sanitize them and if they are acceptable, it delivers them by means of the Oslo messaging bus;
  - Controller Worker: this component gets requests forwarded by the API controller and take actions needed to fulfil them;
  - Health Manager: it checks amphorae status and healthiness, taking as well actions if they fail unexpectedly;
  - Housekeeping Manager: it cleanup database periodically, manages spares pool and amphorae certificate rotation;
- network: Octavia need to impact the network to operate correctly, in particular, it bootstraps a load-balancing network where amphorae are spawn and concurrently manages connections with tenants environments to reach back-end servers that serve load-balanced requests.

Chapter 6

Quantum Key Server

ETSI QKD committee work aims to define a standard architecture, layer by layer, to allow the use of the QKD protocol. The proposed work focused on one of the several aspects described by this committee: the QKD Application Interface. This standard can be found in the ETSI GS QKD 004 paper [33].
This paper it is introduced the concept of QKD manager, a component needed to manage and deliver identical sets of keys to registered applications that may request them. These QKD managers can also be implemented at different levels (i.e. link or network).
QKD managers do not own resources and capabilities to work in a scenario with multiple authorized applications asking for keys or even a scenario where multiple quantum devices are available. To tackle this issue, the ETSI committee defined also another device: the Key Server.
A Key Server(KS) is a device, inserted in a QKD architecture as shown in figure 6.1, that encapsulates a QKD manager, components (that can be virtual) offering services needed to manage the server's state, keys exchanged and perform authentication of incoming requests. It also offers two interfaces, a northern one that enables the interaction with authorized applications and a southern one, used to communicate with the underlying quantum devices, called also QKD modules.
Those that we addressed as authorized applications are defined in the ETSI standard as Secure Application Entity (SAE). With this term, though, it is intended a more general concept, addressing a generic authorized entity that on behalf of one or more applications, has the right to issue requests to the Key Server to obtain one or more keys. Moreover, these queries must carry data to let the KS understand which is the destination SAE that the querying SAE is willing to reach.
QKD modules too deserve a more accurate description. These are defined in the ETSI standard as QKD Entity (QKDE), giving also in this case a generic definition to guarantee a greater degree of freedom to operators willing to implement them. A QKDE is then an entity which role consists of exchanging keys with other QKDE by means of the QKD protocol.

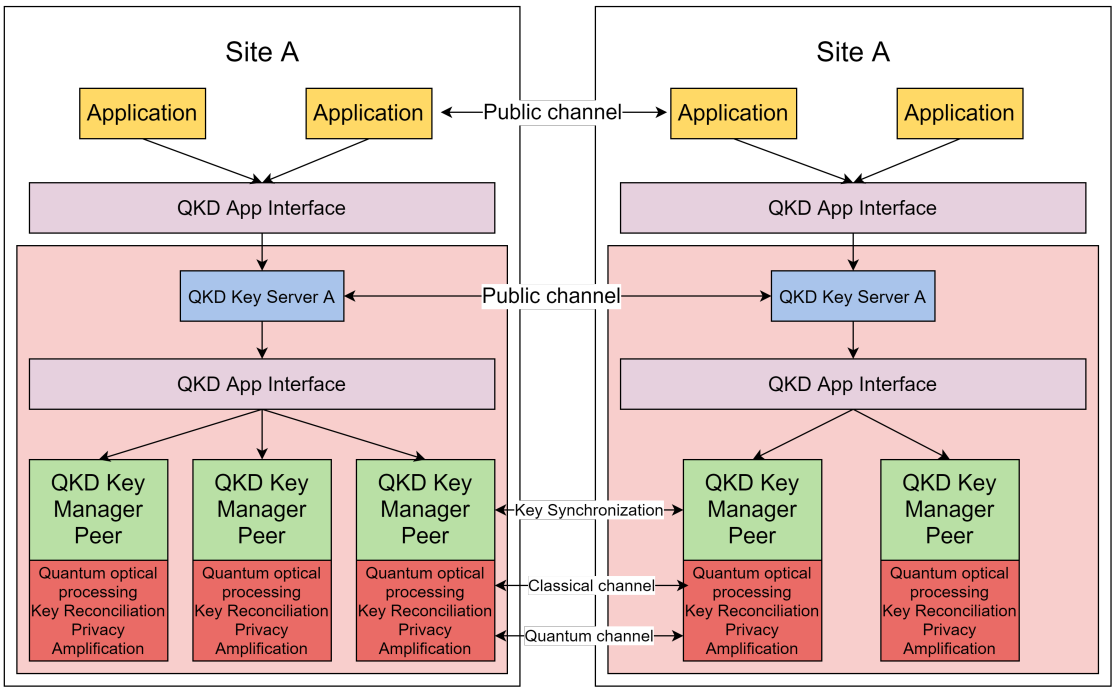

Figure 6.1. QKD architecture #2 (source: [33])

6.1 TORSEC Quantum Key Server

TORSEC group, starting from ETSI QKD standards, developed a prototype of the Key Server, that from now on, for clarity's sake, will be called Quantum Key Server(QKS). It is based on a design which aim is to offer a product, ready to be integrated with an informative system and also able to provide its functionalities inside a cloud infrastructure.
Our analysis and researches build on top of this QKS, proposing several strategies about how these QKS could be exploited in a distributed environment and eventually also how easily they could be integrated with standard security protocols
While in the previous chapter you were introduced to the KS generic architecture described in the ETSI standard, now it will be introduced a brief description of the QKS architecture, detailing how QKS components and functionalities have been implemented by the TORSEC group.

- Authentication and authorization: the QKS employs Keycloak to enforce authentication and authorization to requesting SAEs.
  By design then SAEs and Quantum Key Server must be both registered on Keycloak and, in such a way, whenever a new request is issued, SAEs can query Keycloak to obtain an authentication token that will then be inserted in the request sent towards the QKS. This latter will be in charge of querying then Keycloak about the validity of the received token.

- Secure storage: QKSs continuously exchange keys using their modules. These keys must be stored in secure storage so that later they can be retrieved on demand. Vault [34] was chosen as a component to solve this duty. It must be underlined that there is not a vault for each available module, instead, single secure storage holds keys coming from all of them, letting, though, access only to their respective area of memory.

- Database: The QKS is stateless, as any server should be. It needs a database where data needed to retrieve its state is stored. In this case, it has been used a MySql database. It is used both by the key server, to collect data such as the number of keys stored, the number of attached modules as well as the URI of the known authorised SAEs; both by the QKD modules. Indeed, this database holds also server and modules logs. QKD modules can

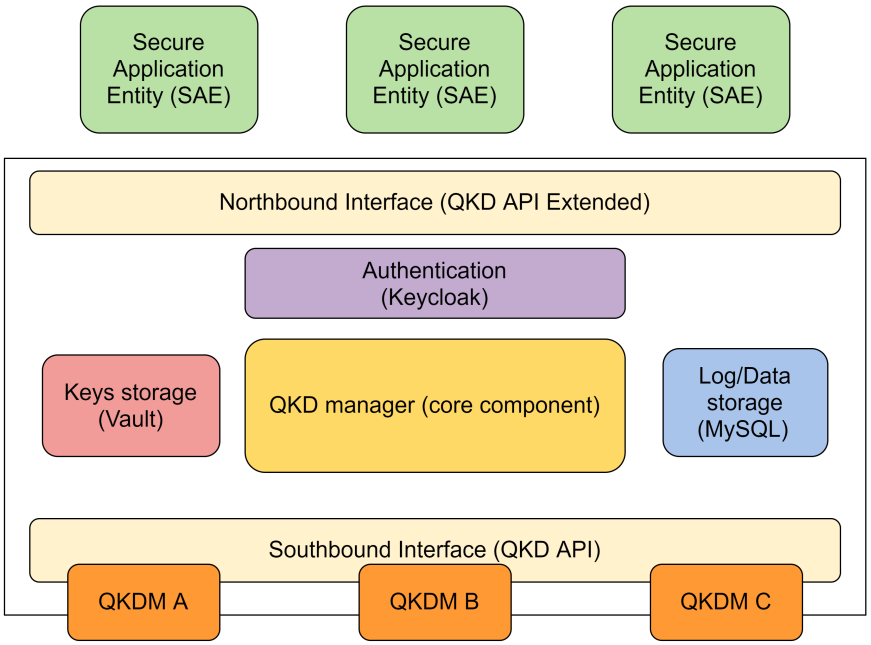

Figure 6.2. QKD architecture #2

access it similarly to the secure storage since they have reference to the portion of memories dedicated to them.

- Core component: It is the component in charge of resources initialization. It also has to offer QKD modules a way to access their related storage. Finally, it is needed to handle communication between different key servers. In particular, even if the communication is over a public channel, it offers authentication thanks to the continuous exchange of keys that is happening in the layer below.

6.2 Interfaces and SAEs interactions

As said in 6.1, the QKS offers two interfaces: the Northbound Interface and the Southbound Interface. There is also a third interface to be considered that allows intercommunication between several Quantum Key Servers: the Synchronization Interface.
These interfaces play a fundamental role in terms of services and overall usability of a QKS and must be described more in-depth. Anyway, due to the scope of this proposed work, the Northbound Interface is the one more interesting for our analysis, then while it will be analysed in detail, the other two interfaces will be simply introduced, giving a high-level description.

The Northbound interface is the one needed to allow communications between the QKS and the SAEs. First things first, it must be stressed that the Northbound Interface implemented in the QKS, does not exactly correspond to the one described in the ETSI standard. Indeed, new methods were introduced to enhance the range of customization given to subscribing applications. In particular, *GET_PREFERENCES()*. *SET_PREFERENCES()* and *GET_INFO* are the three new methods introduced. The former ones allow respectively to read parameters that can be modified by SAEs (QKD protocol, key exchange timeout and logging level) and to modify them accordingly to each SAE's need. The latter simply provide access to information about QKD modules (how many of them are currently attached and which protocol they support), as well as collect logs, specifying the logging level desired.
Focusing now on methods that are described in ETSI standard, the interface offers *GET_STATUS()*, *GET_KEY()* and *GET_KEY_WITH_KEY_ID()* methods. The former one is intended to be used by SAEs to gain knowledge about the number of keys if there are any, ready to be used for the

desired destination. Moreover, it also let them know the length of the key already present.
GET_KEY() and *GET_KEY_WITH_ID()* are the most crucial methods of this interface. Indeed, as you can see in the following paragraph, they represent the core of SAEs interactions. Using *GET_KEY()*, an SAE can query the key server to obtain one or more keys, specifying also their length.
The QKS, before replying, will send the handles related to the requested key(s) to the destination's QKS, so that it can reserve them for the destination SAE. If this phase is correctly performed, the querying key will obtain both keys and handles and it will be its responsibility to communicate the handles to its destination SAE. It is now, in this step of the exchange, that the destination SAE will use *GET_KEY_WITH_ID()* to obtain the identical set of keys, previously reserved.
In table REF these methods are briefly listed, specifying the URL to query and the HTTP method to be used (methods are REST-based).

Method	URL	Access Method
getStatus	https://{KME_hostname}/api/v1/keys/{slave_SAE_ID}/status	GET
getKey	https://{KME_hostname}/api/v1/keys/{slave_SAE_ID}/enc_keys	POST
getKeyWithKeyIDs	https://{KME_hostname}/api/v1/keys/{master_SAE_ID}/dec_keys	POST
getPrferences	https://{KME_hostname}/api/v1/preferences	GET
setPreference	https://{KME_hostname}/api/v1/preferences/preference	POST
getInfo	https://{KME_hostname}/api/v1/information/info	GET

Table 6.1. Northbound interface methods

QKS and QKDEs, mainly communicate using the interface offered by every single entity. However, a Southbound Interface is needed to attach new QKDEs to the QKS.
This interface offers then a method that has the responsibility to initialize resources and exchange minimum QKDE's data needed by the server to properly communicate with it (i.e. the object itself and its address). Moreover, in this phase, each QKDE declares the protocol it can support so that the server can choose properly QKDE to exploit, accordingly to SAEs needs.

The internal Synchronization Interface is the one that allows several operations related to QKSs inter-communications. It offers methods to bootstrap and authenticates the connection, as well as methods to give key servers the ability to look for peer SAEs destinations, previously unknown, or even reserve keys for a given SAE so that they will be available when in a second moment it will request them. Finally, it implements two methods needed by QKDEs to set up a new key exchange stream and to keep synchronization so that the two key servers query modules for identical sets of keys.

6.3 Interactions among SAEs and QKS

One of the challenges that we had to face to implement and test a fully functional scenario, assuming the presence of two QKS, deployed into two separate sites, was to design a communication protocol between two SAEs, one for each site, that desire to establish a secure channel. Indeed the standard ETSI paper [33] does not describe a protocol to tackle this exchange.

You can find a graphical representation of the proposed solution in figure 6.3. As you can see, the first step consists of the *GET_KEY()* call from SAE A to QKS A. During this call the SAE A let the server know which is the desired SAE to be reached, as well as, the number of keys requested and their minimum length.

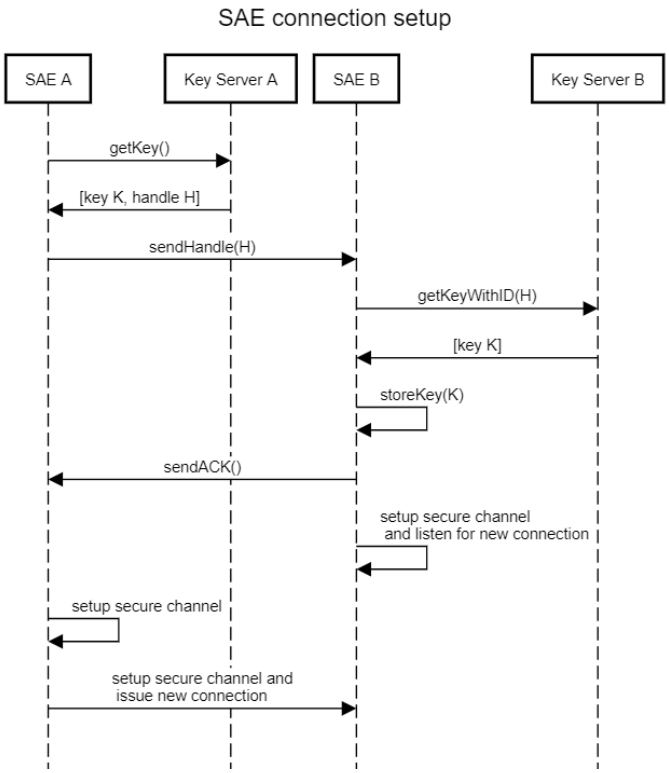

Figure 6.3. SAEs setup protocol #2

After some inner communications between QKS A and QKS B (not shown in the figure since we are focusing now only on interactions involving SAEs), SAE A will receive the QKD generated key together with its handle. The same handle will be then sent to the destination SAE, in this case, SAE B.

SAE B will be waiting for incoming key handles and when it will receive one from SAE A, it will invoke the *GET_KEY_WITH_ID()* method, using the handle as a parameter, to retrieve the corresponding key from the QKS B. As you will see in chapters 8 and 9, after the step where SAE B receives the reserved key, there will be two possible consequent steps to be followed:

- the key will be stored on a dedicated file, assigning to it an ID that must be unique for each connection, and usually carries data related to the two peers establishing the connection. For example, a valid ID would be a string where SAE A IP address and SAE B IP address are concatenated and distinguished by a proper delimiter (usually the colon character);
- the key will be directly used without storing it and the workflow will skip to the next step down the line.

In any case, then SAE B will send an ACK to let know SAE A that the key was correctly retrieved and then the process to establish the secure channel may be started. Similarly to SAE B, also SAE A may decide to store the key on a file, or simply start the channel establishment exploiting it.
As it can be easily guessed this last step, regarding the channel setup, varies depending on the protocol and tool to be used to perform it.

Chapter 7

OpenStack integration

In this chapter strategies to integrate the QKS in an IaaS will be described. Since OpenStack is commonly used as IaaS, it was chosen as a platform where design, implement and finally test our strategies.
In chapter 5 OpenStack, an open-source IaaS was introduced to the reader. Now, instead, we will describe how OpenStack was deployed during our research, and which are the proposed strategies that we formulated to integrate the QKS. The strategy chosen as optimal for this proposed work, corresponds, indeed, to the one implemented in our testbed scenario, so every test conducted and described in chapter 10 was carried out considering this architecture.

7.1 OpenStack deployment

OpenStack deployment is a challenging operation and that is why many deployment tools were developed. Anyway, even after that, the deployment phase is done, it is equally difficult to operate management on it, whether it is regarding a service upgrade, deletion or addition, whether it is the instantiation of new nodes.
Kolla-Ansible [35], which from now on will be called Kolla for brevity's sake, is one of the existing tools developed with the exact intent to simplify all these operations. It will now be introduced briefly its deployment model and main design concepts since, to conduct researches on KeyServer implementation and tests over OpenStack, the testbed environment was deployed employing Kolla. Differently from other deployment tools, it has been designed as a deployment tool able to deploy natively only OpenStack [36]. The main concept behind Kolla is to deploy OpenStack following the micro-services design pattern. Briefly, in micro-services architectures, each service is self-contained, autonomous and it should implement a single business capability [37].
To do so, Kolla exploits Docker technology, packaging each service into a related Docker container. In terms of configuration, instead, it heavily relies on Ansible, a management tool able to submit tasks to several hosts. Kolla's deployment model is illustrated in the figure 7.1 below. Boxes coloured in light red are those steps that need the intervention of an operator, while those in light blue are carried out automatically. As you can also see, Kolla isn't able to take care of the allocation and provisioning of resources, so they must be managed using a different tool.
It is interesting stressing the use of Docker images that are collected from a Docker registry. This implementation allows to easily upgrade a single service, just updating the Docker image present in the registry.

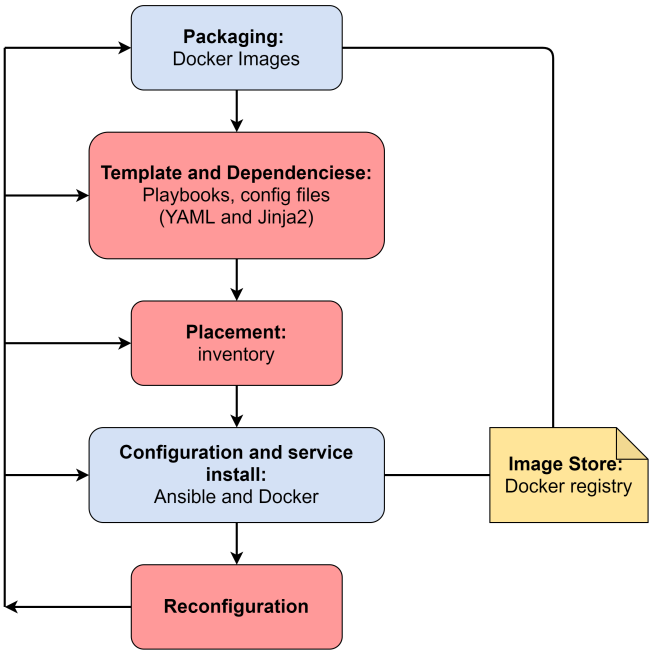

Figure 7.1. Kolla deployment model (source: [36])

7.2 QKS integration strategies

After the first phase of research, performing trials and solving errors to deploy OpenStack through Kolla, Research's focus was moved towards the challenge represented by QKS integration.
Our strategies design revolved around how challenging would have been to integrate the QKS as an Openstack service, and on the other hand, which would have been the best strategy to allow independent evolution of the QKS, avoiding impacts on other services.
As it can be easily guessed, there are multiple possible answers to these issues, each one carrying its set of pros and cons. Three main possibilities individuated will be introduced in the following subsections and while we will describe our chosen strategy, we will also add possible issues and optimizations that may be addressed by future works.

7.2.1 QKS as OpenStack service

The first strategy proposed was to add the QKS as a new OpenStack service, considering a scenario where it could be added to the list of containerized services deployed now by Kolla (this proposed architecture is shown in figure 7.2)
Even if it seems to be an intuitive solution, it was soon discarded as a possible approach, since it was considered not a feasible option for the proposed work. The first step to implement this approach would be to develop an ad hoc API server that would be hosted on the Docker network where all OpenStack services are linked, and that would be able to manage the underlying QKS. It must be stressed that the development of a proper API is not an easy challenge, indeed OpenStack services need to be compliant to many patterns and interfaces to be correctly integrated. Moreover, OpenStack enforces mandatory authentication among services, making developers face yet another challenge to integrate their custom services.
As of last observation on this strategy, we would add that an integrated and perfectly automated approach would be ideal in a production scenario, but at the same time counterproductive in a scenario where researches are still being conducted, since employing changes to the source code would lead with higher probability to unforeseen bugs.

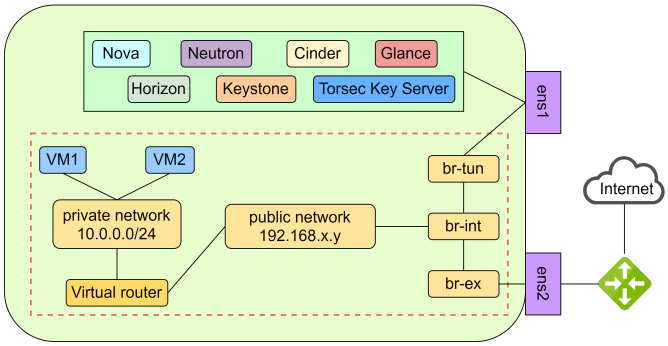

Figure 7.2. QKS as OpenStack service

7.2.2 QKS integration inspired by Octavia model

Looking for possible use cases for the Key Server in an OpenStack environment, a service meant to offer load balancing, named Octavia, piqued our interest. Not only because it could provide a good example of a use case, but also for the peculiar strategy its developers used to implement it. Indeed, instead of being fully implemented as a service, it offers a server, hosted on a Docker container, which offers API to configure and deploy load balancers where the administrator needs them. Every new load balancer creation request corresponds to the spawning of a new VM (predefined by Octavia and named *amphora*) which will feature two network adapters: one connected to the provider network, to be reached from the outside, and a second one linked to the subnet where backends are connected.
Ideally, they should be connected to the provider network and a management network reserved respectively for the Octavia API server and the amphoraes. Due to our experiences, while analyzing this implementation, we can say that it easily conflicts with OpenStack network management, and therefore it is easier to consider a scenario where management and provider network is the same network.

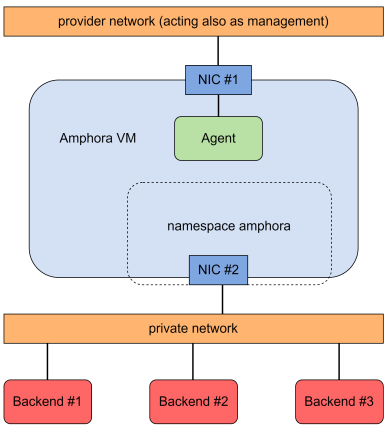

Figure 7.3. Octavia integration model (source: [38])

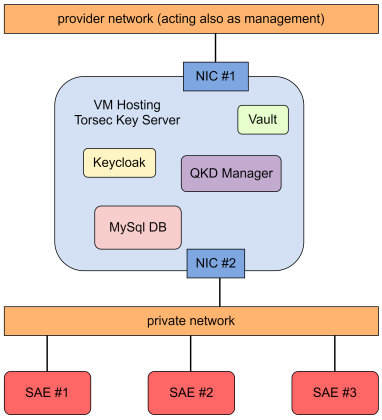

Figure 7.4. VM hosting QKS, resembling Amphora VM

This strategy could be easily adapted for the Key Server. Indeed, it must be implemented simply a new API server offering methods to perform operations such as QKSs spawning and to manage their lifetime. This API server can be then added as an OpenStack service, similarly to the Octavia API server.
Anyway, as already said while discussing the previous strategy, the amount of effort needed to make a service ready to be considered an OpenStack service and correctly authorized is not trivial. Since the main focus of this thesis is the analysis and test of integration of QKD keys in standard

protocols, this integration strategy is left as a possible opportunity to be further investigated in future works.

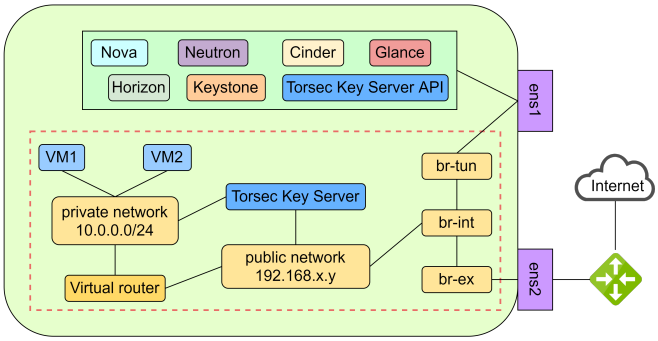

Figure 7.5. QKS integration inspired by Octavia model

7.2.3 QKS hosted on an independent VM

This last strategy, which corresponds to the chosen one, represents a compromise in terms of the effort needed to implement it and the desired design.
Starting from the last strategy discussed, we opted for the deployment of a Key Server over an ubuntu VM spawned over OpenStack, featuring two network adapters: one linked to the provider network (reachable from the outside) and the second one linked to a private subnet where SAEs may be hosted.

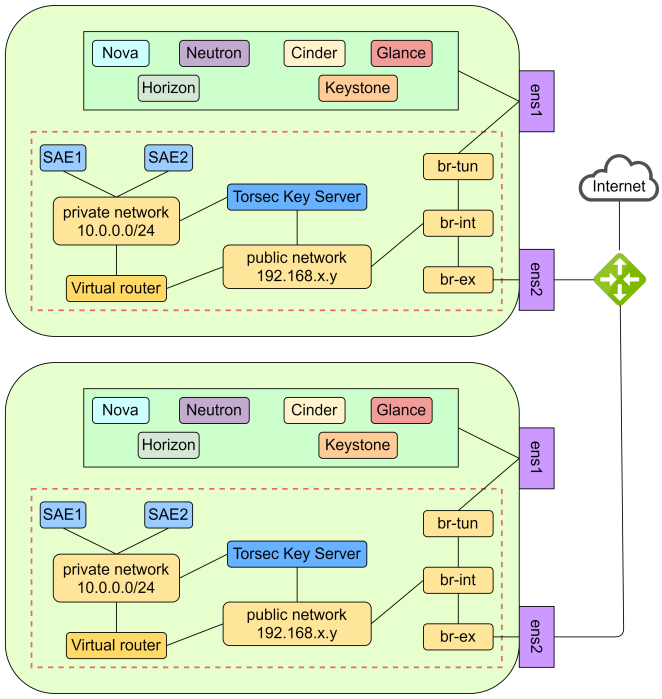

Figure 7.6. Implementation strategy chosen

It was also discussed the possibility to enhance even more proposed integration's capabilities, inserting an Nginx server [39] acting as a reverse proxy, directly linked to the Key Server, so that messages to backend SAEs could be correctly delivered. This option allows linking SAEs only to

a private network, without the need to expose them to external calls. Anyway, it was considered as an effort out of the scope of this work and it was left as possible future work.
The final architecture, indeed, assumes that SAEs have access to the provider network so that they could be able to exchange messages with target SAEs.

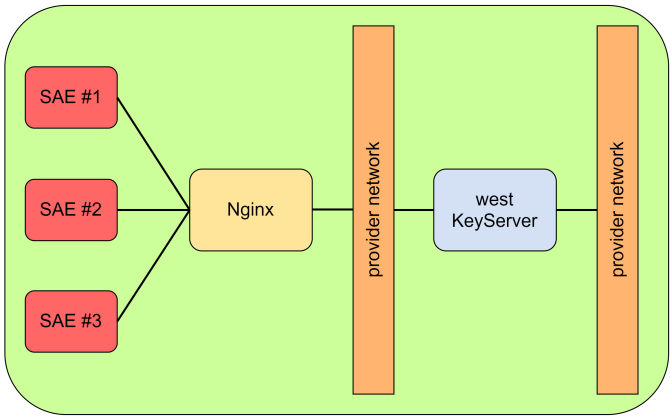

Figure 7.7. Implementation strategy proposed as future work

7.3 Use cases

Now that possible integration strategies, as well as the chosen one, have been discussed, we will focus on two interesting use cases we decided to study more in-depth when considering the integration of a QKS in an OpenStack cloud.
While analyzing OpenStack and its services, even if security properties are considered and implemented in all offered services, two particular ones stood out as perfect candidates to be Key Server's clients: Octavia and VPNaaS.

QKS and Octavia

Octavia as described in 5.5.2, is an Openstack service offering load balancing as-a-service and, as said in the previous paragraph, it offered inspiration for one of the possible QKS's integrations strategies.
At the time of writing, Octavia offers the possibility to function as a TLS terminator. It means that up to the node where the load balancer is hosted (the Amphora VM), every message will be protected employing TLS protocol. Messages exchanged, instead, between the load balancer and related backends, are sent in clear.
The QKS would represent then a useful resource for Octavia, assuming though a certain set of changes that must be performed to allow a correct integration between the two of them.
Indeed, to exploit QKD keys offered by the QKS, Octavia should allow the use of TLSv1.3 (including PSK suites). It should also be opportunely modified to support the same set of methods that SAEs are supposed to provide to interact correctly with the QKS and to exchange key handles with target SAEs.
It must be stressed that this proposed workflow would not be much different from the actual one used by Octavia when it is configured as a TLS terminator. In fact, in this case, Octavia contacts another service, *Barbican*, to retrieve its TLS certificate. The QKS should ideally substitute Barbican in this workflow, and Octavia would then retrieve a QKD key instead of a TLS certificate.
These workflows and their main differences have been represented in figures 7.8 and 7.9.

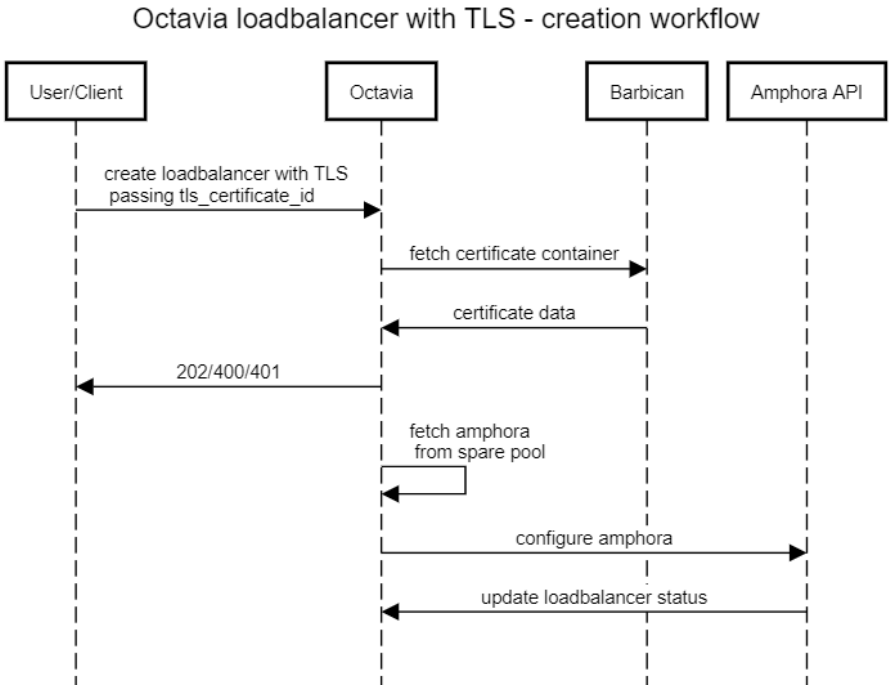

Figure 7.8. Octavia loadbalancer with TLS - creation workflow (source [40])

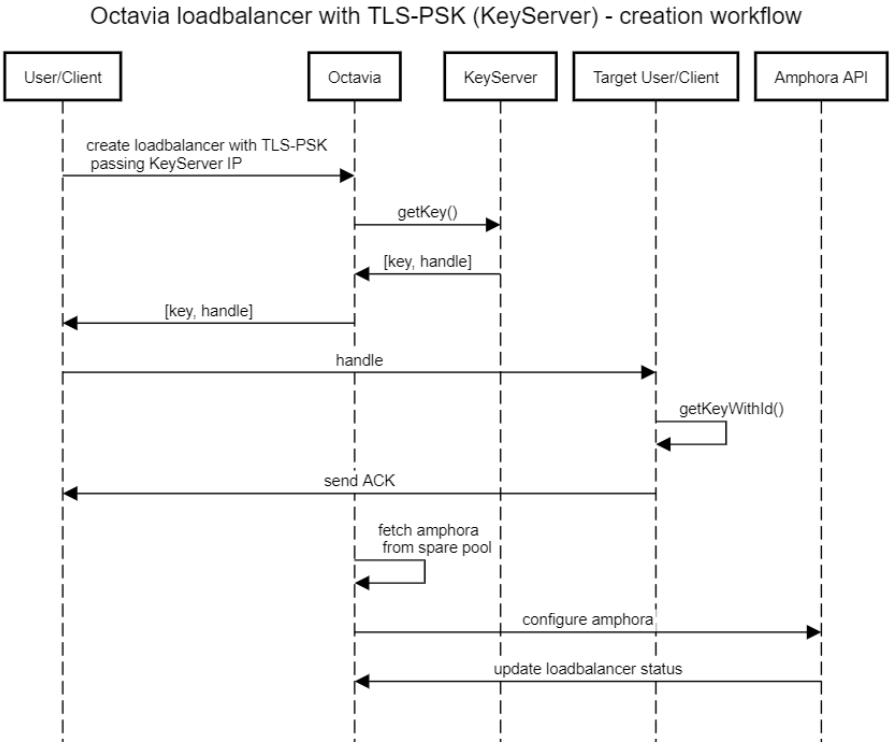

Figure 7.9. Octavia loadbalancer with TLS-PSK (KeyServer) - creation workflow

7.3.1 QKS and VPNaaS

On the other hand, while Octavia leverage TLS protocol to offer security, OpenStack provides another service to offer to its users the possibility to set up a site-to-site VPN connection. This service does not offer simply the possibility to configure a single VPN tunnel, instead, it lets users

enable an extension, included in the Neutron service, implementing VPNaaS.
To enable this service properly, administrators must configure a set of policies and details needed to define properties related to the actual protocols and the endpoints interested in the communication (you can find a more detailed explanation at [31]).

At the time of writing, the VPNaaS extension does not implement any particular process to store and retrieve the keys to be used by the underlying IKE protocol (whether it is its first or second version). Instead, it is the administrator's responsibility to insert a key when configuring the extension.
The key server would fit then perfectly in this schema, taking keys management's responsibility away from administrators, providing QKD keys to be used as PSKs. The best way to implement this proposed workflow would require minor changes to the VPNaaS service's source code, to allow it to contact the QKS when needed, as well as interact with the destination SAE.
Anyway, this is not the only change required to obtain a fully functional integration. Indeed, VPNaaS supports both IKEv1 and IKEv2. While when using IKEv1, as you will see in chapter 8, thanks to this version key generation process, QKD keys can be used as-is, and the resulting key materials and consequently channel may be considered quantum-safe; on the other hand, it cannot be said the same for its second version.
As it will be described in chapter 8, IKEv2 implements a new key generation process that invalidates the security properties that QKD keys feature. Then, to obtain a correct integration, VPNaaS source code should be modified to exploit an IKEv2 version that leverages the use of special packets that were defined just to support quantum-safe keys.

While analyzing OpenStack services, looking for valid use cases, and after having chosen Octavia and VPNaaS, we considered how interesting would be to evaluate an integration from a more generic point of view. In particular, while we could put our efforts into the QKS implementation with these two specific services, we decided that work to obtain an integration with the actual underlying protocols themselves would have been a more efficient strategy.
Indeed, by integrating the QKS directly with IKE and TLS, we can offer a solution that can be easily adapted not only to Octavia and VPNaaS but even to any service that leverages these two protocols.
As you will see in the following chapters, we were able to obtain a satisfying integration, and we leave then as possible future work a more specific integration with Octavia and VPNaaS.

Chapter 8

QKD-based IKE protocol

8.1 IKE and the quantum threat

As already said in chapter 2, quantum computers represent a serious threat to actual cryptographical algorithms. Then, as obvious consequences, modern security protocols, that leverage those algorithms, are vulnerable as well. During our work, we analyzed how TLS and IKE are facing this issue, and which is the strategy that has been suggested to overcome them. In this chapter, the focus will be on the IKE protocol, and then it will be moved on the TLS one in chapter 9.

As described in chapter 3, IKE protocol is available by means of two different versions. These two handle differently the key generation process and, therefore, could be made quantum-safe using different strategies.
IKEv1 is already compatible with the use of quantum keys. Indeed, if administrators would like to use QKD keys as secrets to establish an IKEv1 connection, the connection itself can be considered safe against quantum computers. This is possible since the key generation process that IKEv1 implements do not alter QKD keys properties.
The same cannot be said for IKEv2. To use QKD keys when creating IKEv2 connections, some minor changes are necessary and have already been described in an IETF draft that specifically addresses the issue [41].
In this draft, it is stated that the IKEv2 protocol must be extended through the addition of a Post-quantum Preshared Key (PPK) that is used as an additional secret in a key material generation. Systems interested in setting up IKEv2 PPK-based connections must own a list of PPKs and also store a related ID for each one of them. Moreover, it must be kept the information that specifies which is peer-related to this tuple (key, key id) and whether or not the use of PPK is to be considered mandatory.
It must be stressed that integrating QKD keys in IKE, using them as PPKs, is a pragmatic solution which security must be still formally proven.

8.1.1 IKEv2 setup phase with PPK

In terms of protocol exchanges, this extension requires the addition of some payloads to the standard RFC [41]:

- USE_PPK: notification payload corresponding to type "16435", carrying protocol ID equal to "0" and no SPI. It must be sent by an initiator willing to use PPKs, in its first message. Responders receiving this payload, and that are both able and willing to use PPKs, must include it in the reply.
- PPK_IDENTITY: this notification payload is sent by initiators during the IKE_AUTH phase. It corresponds to a payload of type "16436", protocol ID equal to "0" and with no SPI. Its role is to deliver the PPK_ID chosen by the initiator to the responder.

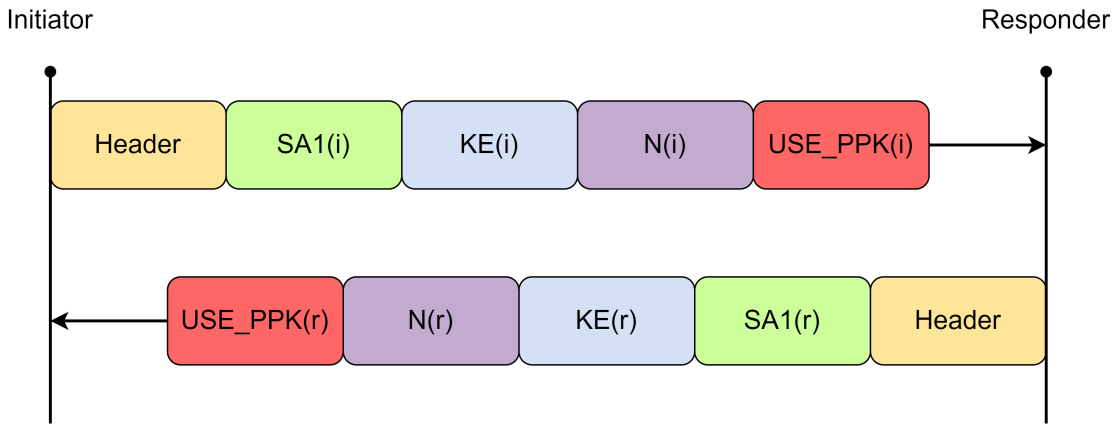

Figure 8.1. IKEv2 - IKE_SA_INIT with PPK (source: [41])

In the scenario where the responder recognizes the chosen PPK_ID, then it must append in its reply an empty PPK_IDENTITY payload.

- NO_PPK_AUTH: this last one is an optional payload that may be included when the initiator is sending its PPK_IDENTITY payload. When NO_PPK_AUTH payload is included, responders can consider the use of PPK optional. So if for any reason it is not possible to conclude the PPK setup (missing PPK_ID for example) the responder can continue the IKEv2 setup as standard RFC describes.

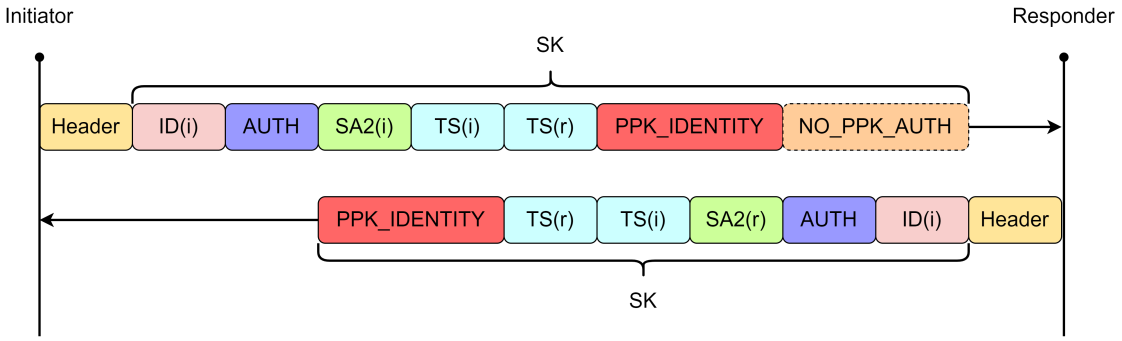

Figure 8.2. IKEv2 - IKE_AUTH with PPK (source: [41])

This draft considers the possibility to offer PPK as optional to ease the upgrade procedure needed to implement it. The concept, indeed, is to slowly upgrade nodes, configuring them to enable the use of PPKs and adding PPKs lists. Their functionality can be tested one by one, simply enabling the use of PPK, defining it as optional. When the upgrade procedure will be fully completed, finally all nodes may be configured to define PPK use as mandatory, being sure that they will work with no issues. It must be stressed that this last step is needed to protect the system from downgrade attacks.
Finally, to fully enable quantum security based only on PPK, excluding public key infrastructure, IKEv2 must be implemented using the NULL authentication method described in its standard. This is possible only in scenarios where nodes consider PPK use as mandatory, otherwise, no authentication would be employed at all.
Security considerations have also been discussed and the main issues arise from the actual security of the PPK used. In particular, what is crucial for the overall security it's its degree of randomness. Since PPKs are meant to be used in an environment where quantum computers exist, when

considering security issues, attacking algorithms that quantum computers can perform, must be taken into account.
Indeed, a quantum computer may be able to perform an attack exploiting Grover's algorithm, halving key size. So it is advised to use a key of at least 256 bits of entropy so that it can be considered to have 128 bits of post-quantum security.

Now that the PPK extension has been introduced in detail, it will be presented the toolkit used to test it. Indeed, as you will see in the next paragraph, Libreswan already provides a functional implementation of the PPK extension, exactly how it is described in the IETF draft [41].

8.2 PPK implementation in Libreswan

LibreSwan is free software that has been in development since 1997. It offers a software implementation of the IPsec protocol, as well as the IKE protocol (both in its first and second version), described by IETF RFC [15].
Even if Libreswan does not represent the only option while choosing a tool to set up a VPN connection, it was the one that we chose to be tested in this work, since it already offers, out of the box, and implementation of the IETF draft where PPKs were introduced [41].
Libreswan is based on the NSS library which carries out IPSec crypto operations in the userspace. Indeed, Libreswan uses a daemon called *Pluto* that provides needed pieces of information to NSS, establishing secure connections, never managing directly keys. This kind of solution gives to Libreswan the great advantage to be able to work without needing any knowledge of the underlying cryptographic device.
It can provide several VPN scenarios, contemplating both hosts to host connection, as well as subnet to subnet, both exploiting the certificate-based or PSK suites. Moreover, it is given the possibility to set up connections between a VPN server and a VPN client, or even VPN connections towards cloud services.
To correctly configure and use Libreswan, administrators must properly edit two files: ipsec.conf and ipsec.secrets. The former contains two main sections: a *conn* section, where data related to connections can be defined (such as subnet addresses, host addresses, protocol to be used as well as details about keying); and a *config* section where data needed to the process startup are configured (i.e. port to bind, logging options or policies to increase resistance against DDoS attacks). The latter, instead, contains the secrets to be used to protect communications. Libreswan supports as secrets standard PSKs, XAUTH passwords and as said before, PPKs. For each PPK entry in this file, there must be a line formatted in the following way: {host A} {host B} : PPKS {PPK_ID} {PPK}.
As you can see, administrators are supposed to insert here static PPKs, but as you will see in the following paragraph, we used these parameters field differently, allowing the possibility to request to a Quantum Key Server a new QKD key whenever Pluto is boot up and then fetches its configurations.
Once the configuration is done, Libreswan offers a set of commands to manage its daemon process through an interface. First things first, administrators must enable and start the ipsec service. In this phase, *ipsec.conf* and *ipsec.secrets* are read and as consequence, the service starts exchanging messages with destinations to set up a public channel that will be used later to configure the VPN. Indeed administrators must then issue a command to explicitly populate the internal database with the connection's data and finally boot the connection. After this step, if all the process was correctly performed, two hosts/subnets will be able to communicate safely through a VPN.

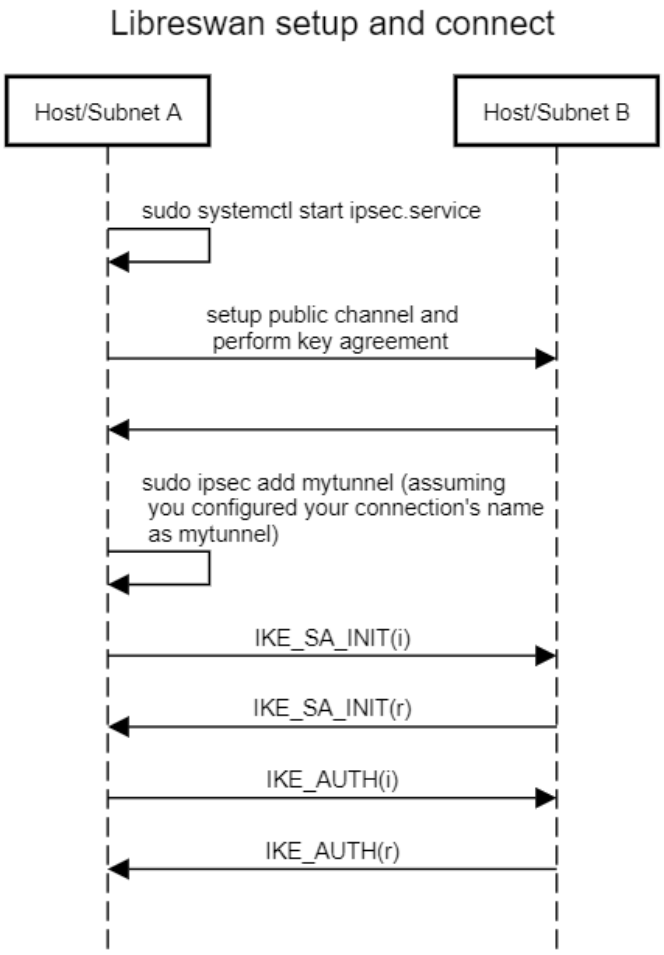

Figure 8.3. Libreswan setup and connect (source:)

8.3 Quantum Key Server integration and workflow

QKD keys could be exploited as PPKs. Since a Quantum Key Server produces constantly new keys, it seemed an obvious consequence to design a workflow where QKD keys are frequently refreshed. As you saw before, though, Libreswan considers secrets as static, so the implementation offered out of the box was not fitting our desired design.
To let interoperate correctly Libreswan and Quantum Key Servers, we chose to intervene on its workflow, acting on the step where secrets are parsed.
As you can see in figure 8.4, after that an administrator issue the command to start the ipsec service, it contacts the TORSEC Key Server, querying for a new key through the GET_KEY() method.

What is happening under the hood, is that the ipsec service is parsing the ipsec.secrets file, scanning it line by line. In its original implementation, this process consists of a simple read from a file, checking how the line retrieved is formatted and analyzing its inner fields, to detect which is the kind of secret retrieved and how it must be exploited.
We modified opportunely this parsing process, changing the code logic to be performed whenever a new PPK key is found. In brief, we implemented the workflow shown in figure 6.3, using data collected from ipsec.secrets file to correctly perform it. We did not hardcode any information related to this exchange, instead, we exploited the fact that LibreSwan parse keys from a file, modifying the meaning on the fields present in a PPK entry.
The new meaning defined for the PPK entry is represented in figure 8.5. As you can see where previously was stored the PPK ID, now data related to the QKS is stored. In particular QKS

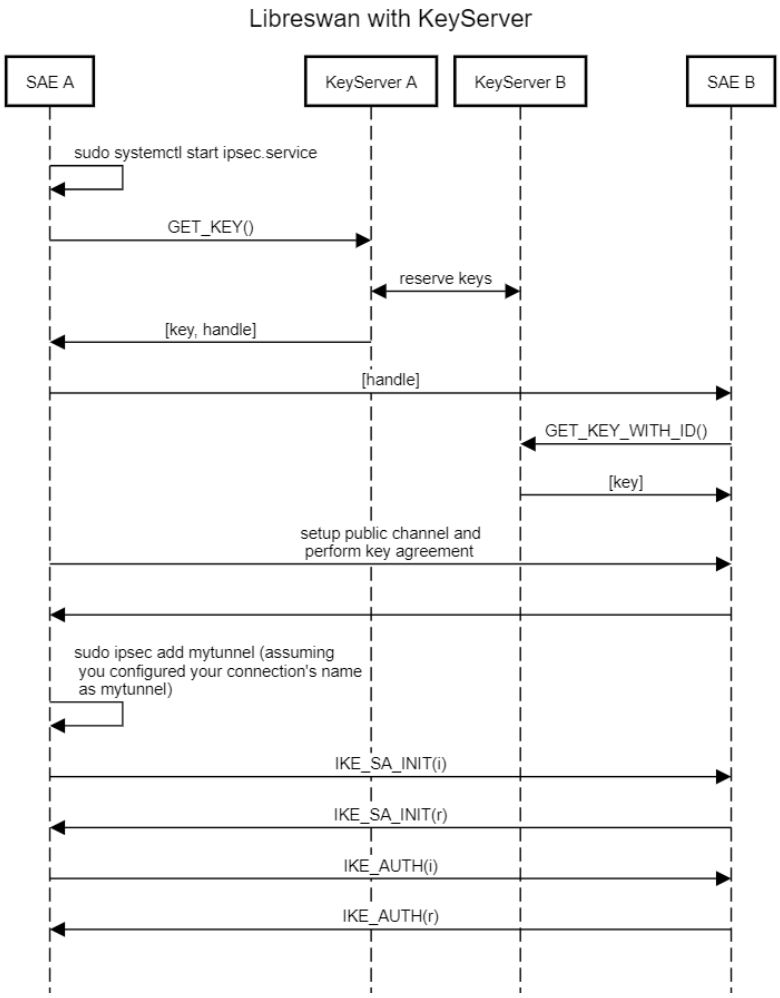

Figure 8.4. Libreswan setup and connect with KeyServer (source:)

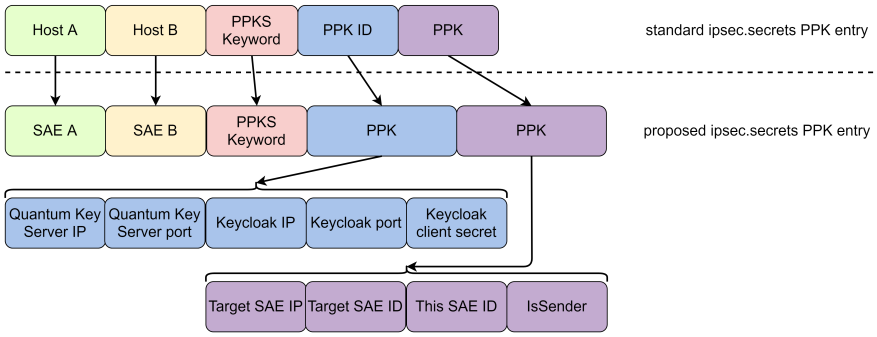

Figure 8.5. Comparison between old and new ipsec.secrets PPK entry

and Keycload addresses and ports are stored, as well as the master token to be used by the SAE while requesting a token from Keycloak.
The old PPK field, instead, contains now data related to the SAEs involved in the connection. You can see, indeed, that both SAEs IDs are stored as well as the IP address of the target SAE. Finally it is stored a boolean value is used to understand if the actual side is the initiator or the responder one. It means that depending on this value is decided which SAE is the one in charge to perform the GET_KEY() request and which one instead must wait for the key handle to use

for the GET_KEY_WITH_ID() request.

We must stress that after these changes, the resulting process, that previously did not consider external requests and was fast to perform, now became a blocking process that put the ipsec service startup on hold for several seconds.

Chapter 9

TLS-PSK over QKD keys

TLS is one of the most common protocols nowadays used to establish secure channels. In the vast majority of modern scenarios where it has been implemented, it is configured to exploit the public key infrastructure.
The introduction of quantum computers represent a major threat for all asymmetric algorithms, then the proposal of solutions to dampen this vulnerability, is a task becoming more and more important. Since this thesis focuses on the use of the QKD protocol, we will not describe here post-quantum solutions, but instead, we will discuss possible integrations that assume the use of QKD keys to be exploited by TLS-PSK suites.
As previously described in chapter 4, TLSv1.3 supports the use of PSKs through dedicated PSK suites. It, indeed, offers two modes to implement a PSK exchange. An operator, configuring a TLS-PSK connection has to choose if they want to employ Diffie-Hellman Exchanges (DHE) during the key exchange process, and then consequently choose the proper mode to be used: *psk_ke* if DHE is not needed, *psk_dhe_ke* otherwise.
While looking for tools to be used to implement TLS protocol to conduct proper tests, we found two toolkits that implement IETF RFC standards and that are the most common ones in modern systems: *OpenSSL* and *WolfSSL*. Moreover, we also considered a third-party tool, named Stunnel, which allows users to employ TLS security without modifying their applications' source code.
What must be stressed, though, is that only WolfSSLl offers both TLS-PSK modalities described before, while OpenSSL, and therefore Stunnel since it leverages on OpenSSL, offer only the psk_dhe_ke mode.

9.1 TLS-PSK using stunnel over openssl

Initially developed by Michal Trojnara, Stunnel is free software (but not a community project) designed to be portable and scalable. Thanks to this design, it can be easily implemented in a distributed environment.
It acts as a proxy that allows creating a TLS tunnel between two applications that do not employ TLS natively, without needing any modification in their source code. Indeed, the TLS connection is set up among two Stunnel applications, one acting as server and the other one as a client. Messages are then sent in clear between applications and their related and Stunnel client/server, while they are encrypted between the two Stunnel applications.
The stunnel approach in terms of configuration and secrets management is similar to Libreswan's. Indeed, administrators must configure properly Stunnel defining its role (Server/Client), the address where to bind/connect, the address of the source application, the protocol to be used and finally the path where secrets are stored.
Differently from Libreswan, though, Stunnel is an application and not a service, so its lifecycle should be handled by the administrator. Moreover, it does not handle the key agreement setup phase, leaving this responsibility to the administrator employing it.

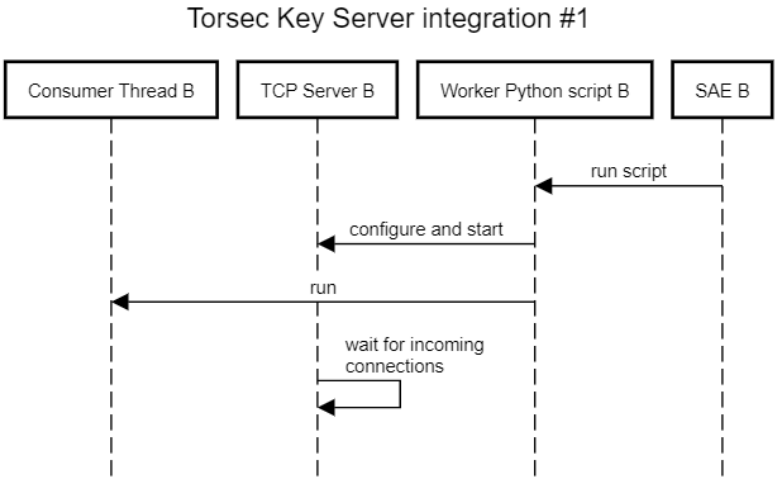

Figure 9.1. KeyServer integration with Stunnel, part #1

To add a certain degree of automatization, in this case, we chose to develop a Python script that would allow users to boot TLS communication over Stunnel without any further input other than the launch of the Python script itself.
In brief, this script configures and launch a TCP server that will be used to handle the key agreement phase and, when the two peers own both the same PSK key, which will be a QKD key, in this case, they store it in the file where Stunnel looks for secrets in its configuration phase. Finally, it forces Stunnel to perform a reconfiguration, sending to the Stunnel process a *SIGHUP* signal.
This design allows indeed both to configure Stunnel for its first communication over this QKD key, as well as, to offer the possibility to issue a key refresh on-demand.
Now that the workflow has been described, we will briefly focus on the actual underlying TLS-PSK implementation. Indeed, Stunnel, in its standard configuration, exploits OpenSSL libraries to carry out any TLS related operation.
While OpenSSL is widely used in informative systems that need TLS security, it is not always on the same page with IETF RFC describing the protocol. In fact, at the time of writing, it does not offer TLS-PSK in its psk_ke mode. WolfSSL, instead, offers both modalities.
we must add that Stunnel actually provides the opportunity to use WolfSSL libraries, instead of OpenSSL's ones. That being said, configuring Stunnel in such a way is a challenging task, and requires further investigation in terms of which are the constraints related to this option. For time constraints, this option was not implemented in the test phase, and it is left as future work.
In any case, Stunnel and OpenSSL represent still a valuable use case to be tested, since many modern systems use them. It must be considered also that in the ETSI report addressing the use of QKD keys in TLS, it is stated that hybrid approaches between quantum security and actual standard protocols, may be considered acceptable during this decade.

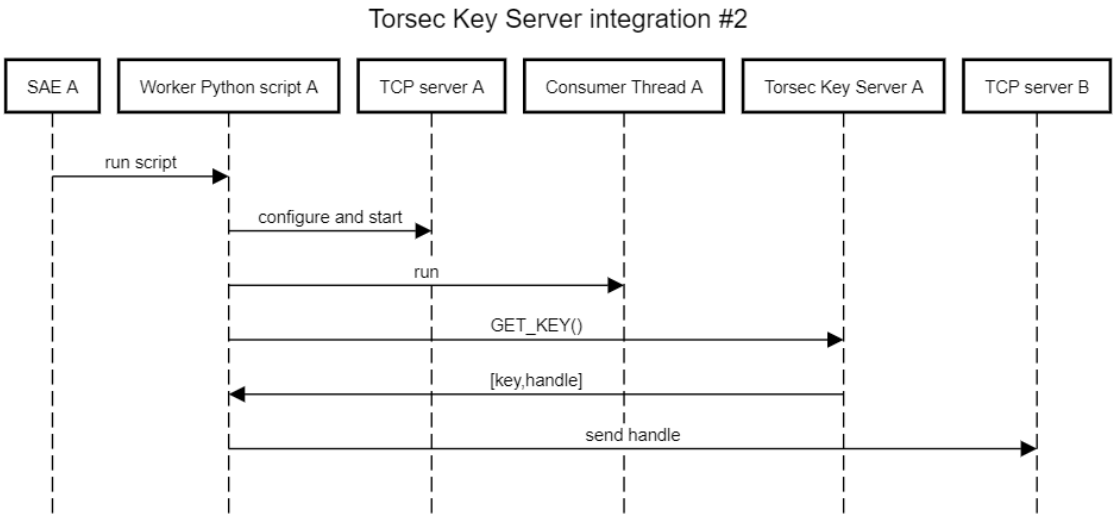

Figure 9.2. KeyServer integration with Stunnel, part #2

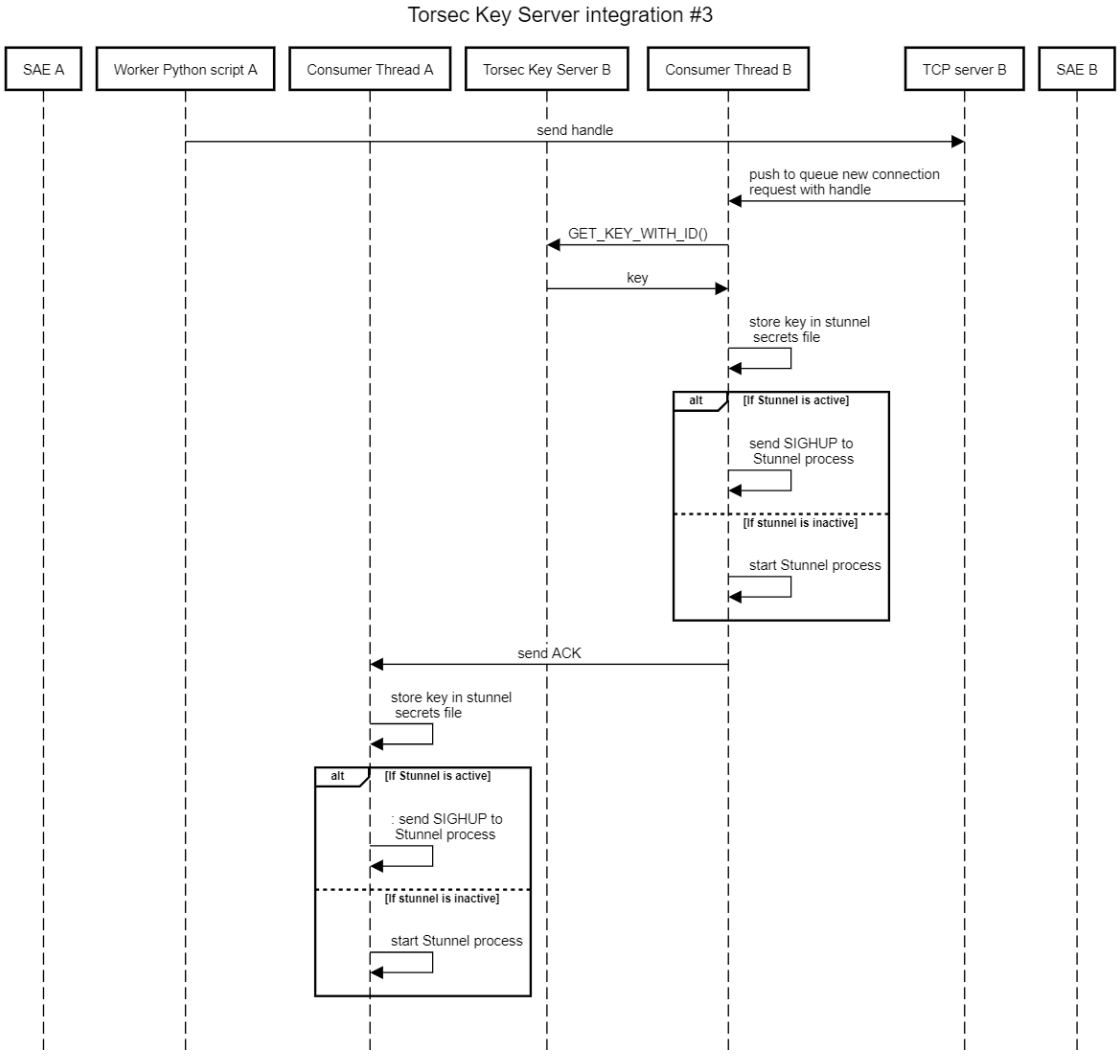

Figure 9.3. KeyServer integration with Stunnel, part #3

9.2 TLS-PSK using wolfssl

Initially developed in 2004 by Larry Stefonic and Todd Ouska, WolfSSL is a toolkit providing primitives to use TLS suites. Like OpenSSL, it is an open-source project but, differently from it, it was designed keeping as main focus portability, performances and lightweight. Today it is a commonly used toolkit, representing the main rival of Openssl, and it offers an ever-growing list of features to its users. Indeed, as said before, psk_ke mode is actually offered only by WolfSSL. Since in this case, our task was to integrate the TORSEC Key Server, directly in a toolkit, we had a certain degree of freedom in terms of design. Even if in this scenario, to not store keys in a dedicated file was an available design option, we decided to follow anyway the same pattern used by Libreswan and Stunnel, providing, indeed, a file where exchanged keys must be stored.

The overall proposed workflow is divided into two main steps: first SAEs must agree on the QKD keys to be used, storing them in a file; then, the same keys are retrieved by the process establishing the TLS channel, so that they can be passed to the dedicated TLS callback offered by WolfSSL.
From now on we will refer to these phases as the *key agreement* phase and the *connection* phase. Both phases were designed in a client/server fashion. Indeed, even if a peer-to-peer pattern would have been optimal in terms of optimization, we preferred to choose a pattern resembling the one usually exploited by similar tools (such as Stunnel).
We will focus now on the connection phase since the key agreement phase is identical to the one presented for Stunnel. To correctly implement this second phase, we had to slightly modify WolfSSL callbacks in order to let them accept non-static keys, that are instead retrieved at run-time from the file where they are stored. Moreover, from the client-side, we needed to modify the method offered by WolfSSL that manages the creation of a TCP connection.
In fact, since our design allows to set up and connect two SAE with a single input from the user, we needed to make sure that a client connecting to a server yet not ready, would retry to open the connection for a minimum number of trials, while before, it simply would return an error in case of an unsuccessful connection.

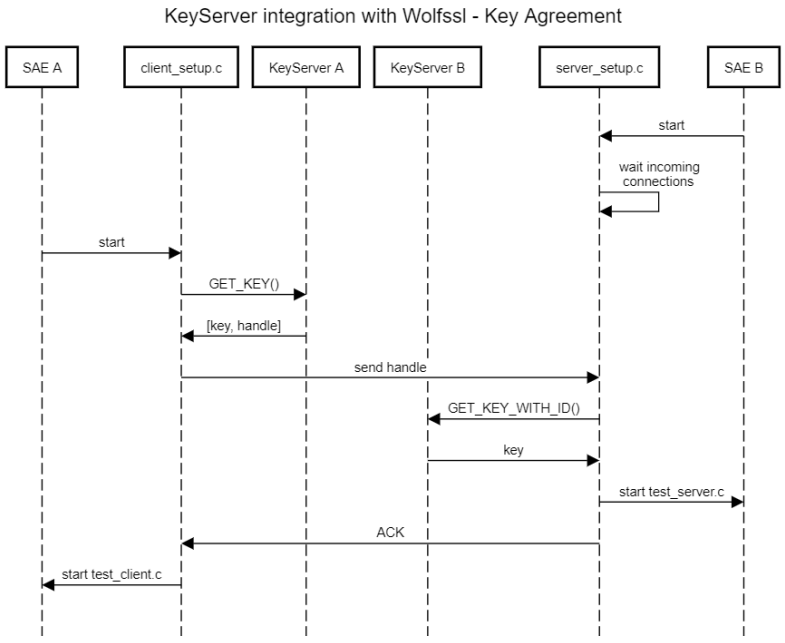

Figure 9.4. KeyServer integration with Wolfssl - Key Agreement

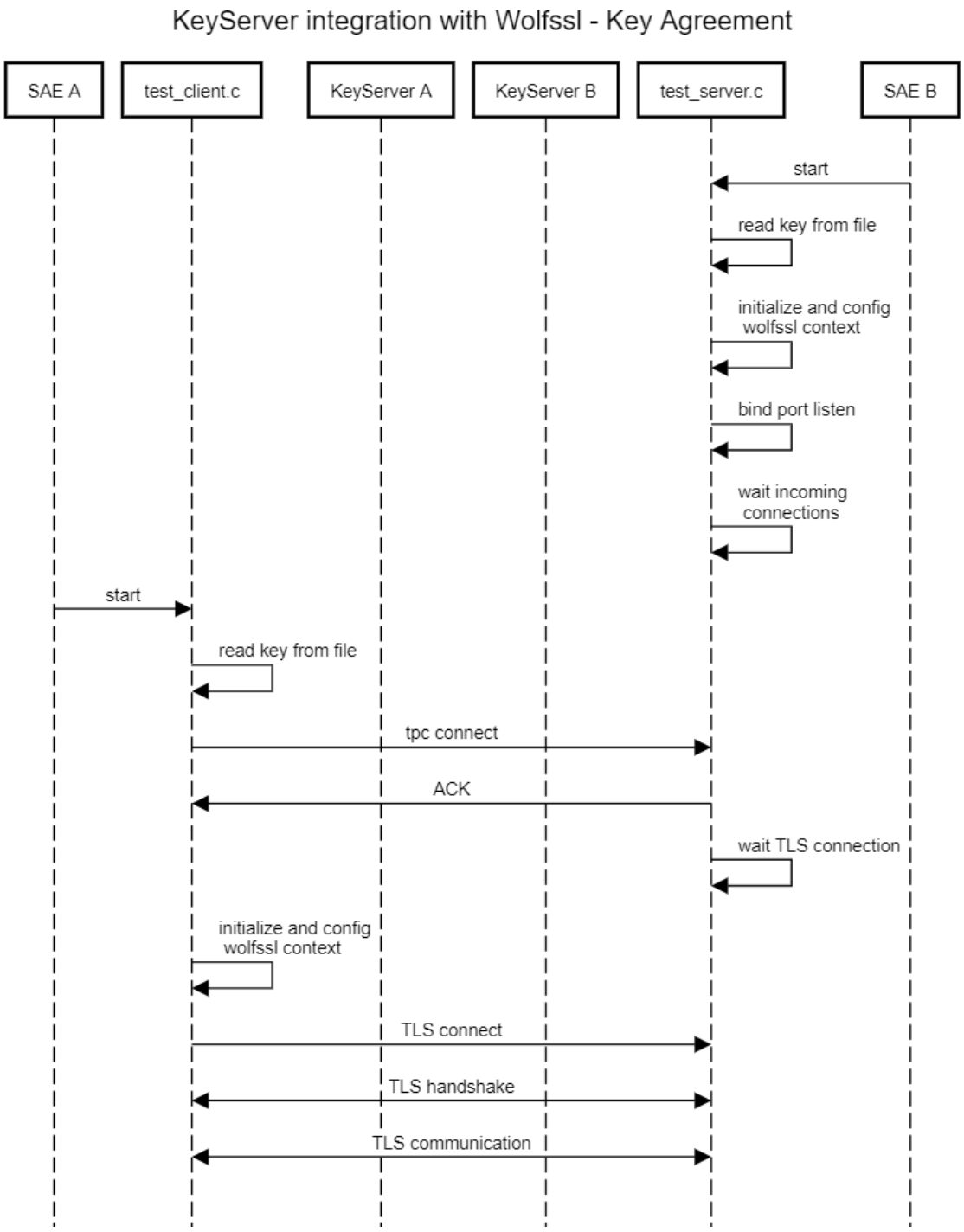

Figure 9.5. KeyServer integration with Wolfssl - Connection

Chapter 10

Test

Testbed used to perform tests and evaluate results presented in this chapter is composed of two Openstack sites, deployed in an All-in-one (AIO) fashion through Kolla-Ansible, and hosted on two Intel NUC (Next Unit of Computing). Each of these Intel NUC features the following characteristics:

- CPU: Intel Core i5-5300U - 2.30 GHz
- RAM: 16 GiB
- OS: Ubuntu 18.04.5 LTS (Bionic Beaver)
- Storage: SSD 50 GB (free space size recommended)
- Two Network Interface Controllers (NICs)

The two Intel NUC and, therefore, the two Openstack sites, have been hooked to the same switch device. Then, communications between the two hosts did not need to pass through an external network.
Focusing now on the internal architecture of each site, we deployed Openstack and integrated the QKS as shown in figure 7.6, spawning then three VMs, each one configured differently depending on the test we performed exploiting them:

- Ubuntu 18.04.5 LTS (Bionic Beaver) VM featuring our custom version of the LibreSwan toolkit
- Ubuntu 18.04.5 LTS (Bionic Beaver) VM featuring the standard installation of the WolfSSL library (with TLSv1.3 and PSK enabled) and hosting our programs interacting with each other and the QKS
- Ubuntu 18.04.5 LTS (Bionic Beaver) VM featuring the standard installation of the Stunnel tool, as well as our custom Python script managing QKS and SAEs interactions

This architecture was identical in both sites so that SAEs from site A would be able to interact with their mirror SAEs from site B, reaching then a total of six VMs (actually eight if we count also those that hosted the two QKSs). In total, we conducted three tests: one to evaluate IKE integration and the other two considering TLS-PSK. In all of these cases, we checked the functional aspect of the exchanges performed, as well as the overall overhead introduced by interactions with the QKS. we also checked that, as expected, these integrations would not be affecting the communication phase happening after that the secure channel was established. It must be added that measurements and values provided in this chapter represent an average calculated over around a hundred iteration for each test
Since, from a functional point of view, all of these three cases share the same protocol to interact with the QKS and perform a key agreement, we will before introduce to you the results found in this common phase and then we will move to analyze in detail each test case.

10.1 SAEs and QKS interactions test

Gathering execution times needed to perform several operations from the first exchange up to the exchange of messages over the secure channel established, we decided to distinct two phases: a first one considering the time needed to complete the workflow defined in figure 6.3; and a second one including operations needed to exchange secure messages over the channel. The execution time needed to fully perform the former phase will be addressed from now on as *setup time*.

The setup time is different for each of the test cases in the exam since each of them perform different operations to implement the full workflow. Then their measurements will be introduced when addressing individually each test. Moreover it must be made distinction between the setup time needed by the node acting as initiator, and the setup time needed by the one acting as responder.
In next sections, times needed to perform single operations will be presented and it will be proved that the setup times can be calculated with good approximation as follows:

$$t_{initiator\_setup} = t_{get\_token} + t_{get\_key} + t_{send\_handle\_receive\_ack} \tag{10.1}$$

The latter, instead, can be measured as:

$$t_{responder\_setup} = t_{get\_token} + t_{get\_key\_with\_id} + t_{send\_ack} \tag{10.2}$$

Now, the setup phase will be shown from a functional point of view. Indeed, while testing our integration, we collected packets exchanged among SAEs and QKSs to prove that the workflow was being executed as expected. In figures 10.1 and 10.2 are shown packets captured from the two different QKSs presents in the testbed, that will now be called QKS A and QKS B (with the QKS A taking the role of client, and the QKS B instead acts as a server).
The first packet is received by the QKS A and contains the request sent by SAE A to obtain a token from Keycloak. When the token is obtained, then SAE A sends a GET_KEY() request as you can see in the third row of the image (any request here is shown by the REST method matching it, and it can be found in table 6.1).
When the QKS A receives a GET_KEY(), it begins a sequence of exchanges among the two QKSs (IP addresses are respectively: 192.168.0.249 and 192.168.0.236). If it ends successfully then the QKS A replies back to SAE A with an HTTP message with code 200, carrying the requested key.

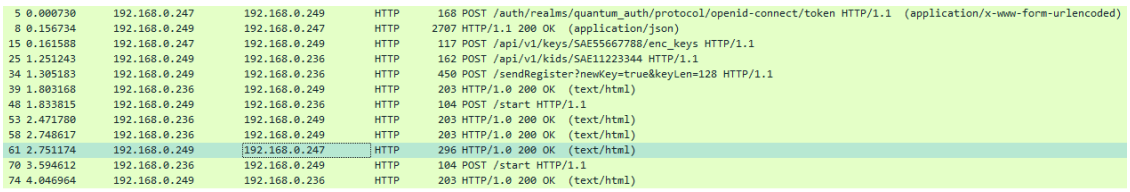

Figure 10.1. Key agreement using KeyServer - client side

In figure 10.2 you see instead of the perspective from the QKS B point of view, starting from the moment when the SAE B received the key handle and it is now contacting its QKS. Similarly to the previous case, even here it beforehand contacts Keycloak to obtain a token. Then, this time, the SAE B sends a GET_KEY_WITH_ID() request (line 57 in the figure), and since there is no communication needed with its peer QKS, the QKS B can return right away the key linked to the provided handle.

5	0.000641	192.168.0.237	192.168.0.236	HTTP	168	POST /auth/realms/quantum_auth/protocol/openid-connect/token HTTP/1.1 (application/x-www-form-urlencoded)
8	0.035709	192.168.0.236	192.168.0.237	HTTP	2707	HTTP/1.1 200 OK (application/json)
17	6.657614	192.168.0.249	192.168.0.236	HTTP	162	POST /api/v1/kids/SAE11223344 HTTP/1.1
26	6.674781	192.168.0.249	192.168.0.236	HTTP	450	POST /sendRegister?newKey=true&keyLen=128 HTTP/1.1
30	7.172205	192.168.0.236	192.168.0.249	HTTP	203	HTTP/1.0 200 OK (text/html)
41	7.203391	192.168.0.249	192.168.0.236	HTTP	104	POST /start HTTP/1.1
44	7.840809	192.168.0.236	192.168.0.249	HTTP	203	HTTP/1.0 200 OK (text/html)
49	8.117638	192.168.0.236	192.168.0.249	HTTP	203	HTTP/1.0 200 OK (text/html)
57	8.304950	192.168.0.237	192.168.0.236	HTTP	135	POST /api/v1/keys/SAE11223344/dec_keys HTTP/1.1
64	8.963415	192.168.0.236	192.168.0.249	HTTP	104	POST /start HTTP/1.1
68	9.219362	192.168.0.236	192.168.0.237	HTTP	307	HTTP/1.0 200 OK (text/html)
73	9.430905	192.168.0.249	192.168.0.236	HTTP	203	HTTP/1.0 200 OK (text/html)

Figure 10.2. Key agreement using KeyServer - server side

10.2 IKE integration test

To test the IKE integration we set up two VMs, one for each site, installing initially a standard version of the LibreSwan toolkit, replacing it then in a second moment with our custom version. In both cases, LibreSwan was configured to establish a secure channel using a PPK.
In the first case, the daemon pluto, managed by LibreSwan, simply collected a static PPK from a file and use it to carry out the authentication and configuration of the channel. In the second case, instead, it used data found in the file where usually PPK are stored, to contact the QKS and carry out the key agreement with the mirror SAE hosted on the other site.

The first test performed was a functional one. Indeed we needed to prove that in both our test cases, the daemon pluto was correctly performing IKE exchanges as described in the related RFC [41]. As you can see in figure 10.3 is represented a capture collecting packets exchanged by SAE A and SAE B to set up the channel. What is interesting to notice is that, exactly as expected, you can find in the IKE_SA_INIT the notification payload used to declare the will to use PPKs to authenticate the connection. It can be distinguished by the field carrying its status type, which corresponds to "16435".
The same result was obtained in both test cases conducted, independently from our changes to LibreSwan

37	13.903994	192.168.0.239	192.168.0.245	ISAKMP	901	IKE_SA_INIT MID=00 Initiator Request
38	13.909768	192.168.0.245	192.168.0.239	ISAKMP	505	IKE_SA_INIT MID=00 Responder Response
39	13.960645	192.168.0.239	192.168.0.245	ISAKMP	445	IKE_AUTH MID=01 Initiator Request
40	14.002553	192.168.0.245	192.168.0.239	ISAKMP	275	IKE_AUTH MID=01 Responder Response
41	27.172081	192.168.0.239	192.168.0.245	ESP	154	ESP (SPI=0xb5e87706)
42	27.172081	192.168.0.239	192.168.0.245	ICMP	98	Echo (ping) request id=0x4394, seq=1/256, ttl=64
43	27.172208	192.168.0.245	192.168.0.239	ESP	154	ESP (SPI=0x0efea8d4)
44	28.209276	192.168.0.239	192.168.0.245	ESP	154	ESP (SPI=0xb5e87706)
45	28.209276	192.168.0.239	192.168.0.245	ICMP	98	Echo (ping) request id=0x4394, seq=2/512, ttl=64
46	28.209384	192.168.0.245	192.168.0.239	ESP	154	ESP (SPI=0x0efea8d4)
47	29.210707	192.168.0.239	192.168.0.245	ESP	154	ESP (SPI=0xb5e87706)
48	29.210707	192.168.0.239	192.168.0.245	ICMP	98	Echo (ping) request id=0x4394, seq=3/768, ttl=64
49	29.210773	192.168.0.245	192.168.0.239	ESP	154	ESP (SPI=0x0efea8d4)

```
Payload: Notify (41) - RESERVED TO IANA - STATUS TYPES
    Next payload: Notify (41)
    0... .... = Critical Bit: Not Critical
    .000 0000 = Reserved: 0x00
    Payload length: 8
    Protocol ID: RESERVED (0)
    SPI Size: 0
    Notify Message Type: RESERVED TO IANA - STATUS TYPES (16435)
    Notification DATA: <MISSING>
```

Figure 10.3. IKE_SA_INIT carrying PPK notification payload

The second test aim was instead to measure the impact of QKS interactions on the overall time needed to set up a secure channel. The difference between the two cases tested (standard and custom LibreSwan) reside indeed in how the PPKs are fetched. In the first case, the execution time mainly consists of I
O operations needed to read the key from the dedicated file. In the second one, instead, there are several exchanges involved, both among SAEs and QKSs, therefore the time needed to process

this workflow is way higher than the standard one, jumping from execution times in the standard version lying in the realm of milliseconds, up to multiple seconds needed to perform exchanges in our custom version.
Since these measurements belong to magnitudes too different, they cannot be described well by any graph. Therefore we collected and showed them in tables 10.1 and 10.2 where you can see how, on average, the overhead introduced in the setup phase corresponds to more than one second (measured from the SAE B side).

	Custom LibreSwan	Standard LibreSwan
Configuration read and load	2.9×10^{-5} s	N/A
GET_TOKEN()	3×10^{-2} s	N/A
Paese token	1.2×10^{-4} s	N/A
GET_KEY	1.2 s	N/A
Parse key	4.6×10^{-5} s	N/A
Receive ACK from Server	8.4×10^{-1} s	N/A
IKE initiator setup time	3.9 s	1×10^{-5} s

Table 10.1. Custom LibreSwan vs Standard LibreSwan (Initiator)

	Custom LibreSwan	Standard LibreSwan
Read handle	2×10^{-4} s	N/A
GET_TOKEN()	3×10^{-2} s	N/A
GET_KEY_WITH_ID()	1.2 s	N/A
IKE responder setup time	1.2 s	1×10^{-5} s

Table 10.2. Custom LibreSwan vs Standard LibreSwan (Responder)

10.3 TLS-PSK integration test

To test TLS-PSK integration, we considered two main test cases: a first one to test the TLS-PSK mode using also Diffie-Hellman exchange (psk_dhe_ke), and a second one using instead the mode exploiting only the PSK (psk_ke).
To do so we spawned two couples of Ubuntu VMs, in such a way to have one VM for each test case on each site. After having layout this architecture we performed similar tests to those one seen in the IKE case: first, a functional one to be sure that our integration was compliant to RFCs describing it and then a test focused on the performances and, in particular again on the setup time measured.

10.3.1 PSK with DH - OpenSSL

For this test case, we installed the standard version of Stunnel (configured by default to use OpenSSL) and then we configured our Python script to handle SAEs and QKSs interactions.
With the functional test we conducted, we proved that exchanges happening between SAEs were indeed matching the expected ones. In particular, you should notice that in the Client Hello represented in figure 10.4, both the key_share and the psk_key_exchange mode are present.
Moreover, as expected in this specific case, since we are using OpenSSL, the PSK mode offered inside the psk_key_exchange is the psk_dhe_ke mode.

Measurements conducted to evaluate the overhead introduced by QKS in the setup phase were performed comparing the scenario where Stunnel simply fetches keys from a file and then the scenario where QKS are taken into account.
These tests show similar results to those seen in the IKE use case. Indeed, again QKSs and SAEs

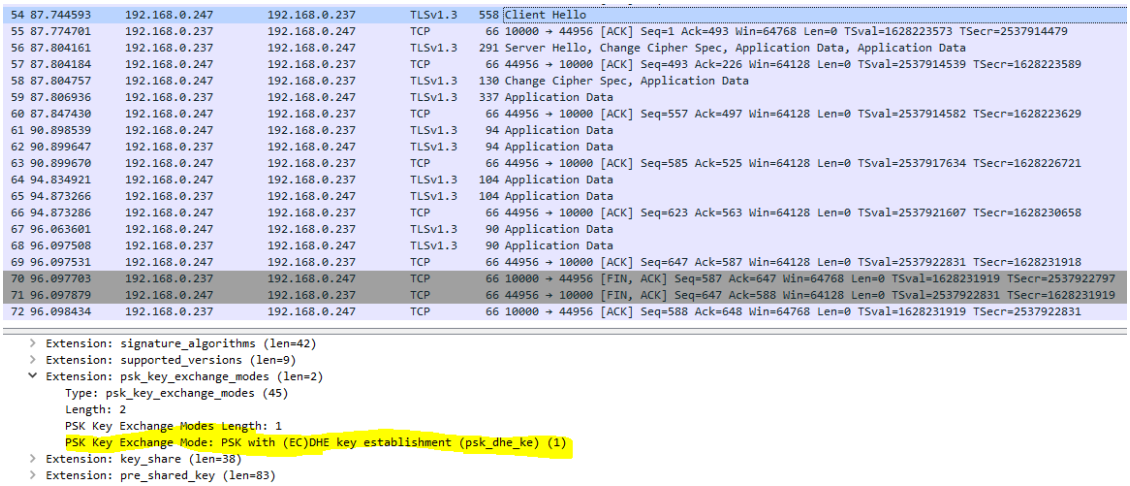

Figure 10.4. Stunnel Client hello with PSK - psk_dhe_ke mode

interactions process time are those driving the overall time needed to complete the setup phase, as you can see in tables 10.3 and 10.4

	Stunnel exploiting QKS	Stunnel retrieving keys from file
Load Config file	2×10^{-4} s	2×10^{-4} s
Create/Load secrets file	2×10^{-4} s	2×10^{-4} s
GET_TOKEN()	7×10^{-2} s	N/A
GET_KEY()	2.7 s	N/A
Key parsed and stored	1.1×10^{-2} s	6.7×10^{-4} s
Receive ACK	1.26 s	4.7×10^{-2} s
Stunnel initiator setup time	4 s	5.4×10^{-2} s

Table 10.3. Stunnel with QKS vs Standard Stunnel workflow (Initiator)

	Stunnel exploiting QKS	Stunnel retrieving keys from file
Load Config file	2×10^{-4} s	2×10^{-4} s
Create/Load secrets file	2×10^{-4} s	2×10^{-4} s
Read handle	7×10^{-4} s	N/A
GET_KEY_WITH_ID()	1.2 s	N/A
Key parsed and stored	1.5×10^{-2} s	6.7×10^{-4} s
Start/Reconfigure Stunnel	5.4×10^{-2} s	4.7×10^{-2} s
Stunnel responder setup time	1.23 s	5.4×10^{-2} s

Table 10.4. Stunnel with QKS vs Standard Stunnel workflow (Responder)

10.3.2 PSK only - WolfSSL

VMs exploited to perform the second test case, instead, featured the standard version of WolfSSL and hosted our two custom scripts used respectively to handle key agreement and then used the fetched key to set up the TLS-PSK secure channel.
Even in this case, the functional test was the first one performed and similar to the previous one,

we captured exchanges among SAEs to check that the Client Hello was carrying the expected extension. As you can see in figure 10.5, the psk_key_exchange extension is indeed present and it is now carrying the request to use the psk_ke mode.

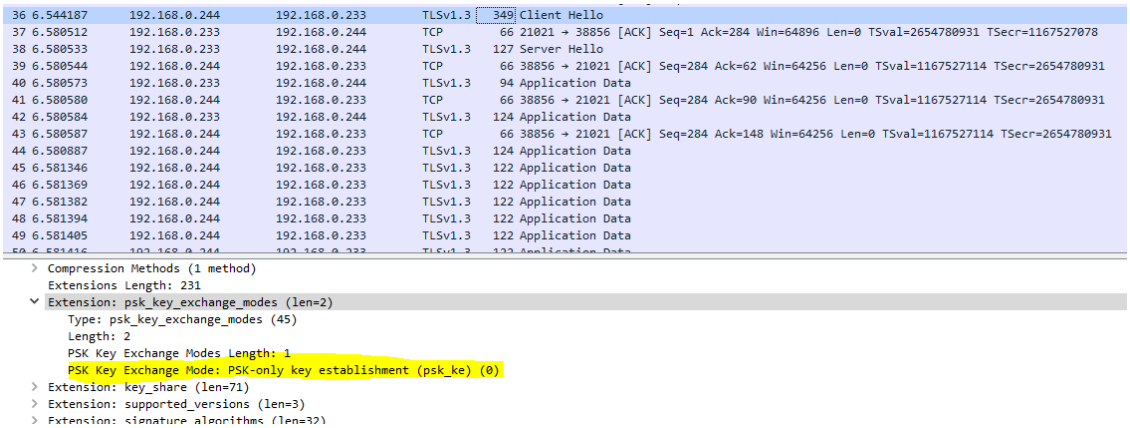

Figure 10.5. Wolfssl Client hello with PSK - psk_ke mode

Finally, introduced overhead was evaluated also for this test case, comparing the use of WolfSSL exploiting the QKS and the standard use of WolfSSL passing in static PSKs.
As expected, results are similar to those observed in previous test cases. In this case, though we developed two scripts: one to handle key agreement and a second one to handle the channel setup and the actual exchanges over the channel.
Then in terms of overhead introduced it is enough to consider the time needed to process the first script which in the case of standard WolfSSL use is not executed in the first place. Times to operate for completing the setup phase are then collected and shown in tables 10.5 and 10.6.

	WolfSSL setup with QKS
GET_TOKEN()	4.5×10^{-2} s
GET_KEY	2.5 s
Sent handle and received ACK	1.1 s
WolfSSL initiator setup time	3.7 s

Table 10.5. WolfSSL setup operations execution times

	WolfSSL setup with QKS
Received handle from client	2×10^{-2} s
GET_KEY_WITH_ID	1.1 s
Key stored on file	2×10^{-4} s
WolfSSL responder setup time	1.1 s

Table 10.6. WolfSSL setup operations execution times

10.4 Considerations about results

Some considerations regarding results obtained must be presented. In particular, it is important to stress both the actual average times of overhead measured, both a couple of improvements that could be operated on the testbed.
As you can see in table 10.7 the average time needed to complete the setup phases in the various

test cases are close. Indeed main overheads are not due to new flows inserted in each toolkit used, but instead, they are caused by the QKS which is the common factor in all three test cases.
Improvements to the QKS would then result in optimization of these overheads. After all the QKS employed in this work was a beta version of the final product that is still developed by the TORSEC group, therefore big improvements may be expected, especially since the final product will not only more optimized but also will feature the option to perform multiple requests in parallel.
Moreover, it must be considered that real quantum devices, at the time of writing, can exchange keys with a throughput close to 1 kbps, so overheads measured in this work are on the same page with the state of the art.

	QKS with IKE	QKS with Stunnel	QKS with WolfSSL
Average initiator setup time	1.2 s	1.23 s	1.1 s
Average responder setup time	3.9 s	4 s	3.7 s

Table 10.7. Setup time of the proposed use cases

It should be also considered that results are measured on a testbed where no external network is involved. Therefore no network latency has been taken into account while measuring overall execution time overheads. Anyway, while testing on sites physically separated by an external network may be interesting, the QKS indeed allows asking for several keys with a single request. Exploiting this possibility, overheads due to network latency may be greatly optimized. This approach allows also to reduce the impact of measured overheads added during the setup phase.

The testbed could be improved also considering a scenario closer to the reality of a distributed environment, where hundreds, if not thousands, of nodes, may be flooding the bandwidth and the QKS with requests.
Even if it is a recommended scenario to test in future works, it is already expected that the QKS, due to its design based on containers, can be easily scaled horizontally, being in such a way able to face a distributed environment without any issues. Moreover, it must be stressed that the testbed provided already is based on two sites featuring the deployment of an IaaS, so they are not so much different from a real scenario.

Chapter 11

Conclusions

This work main objective was to offer a valid integration strategy for the Torsec Key Server in a distributed environment. After several analysis conducted on the opensource IaaS OpenStack, have been given two possible strategies, with related pros and cons in terms of complexity or possible automation, to properly integrate the Torsec Key Server.
Moreover, this work also had the purpose to find possible use cases in said scenario. Indeed, OpenStack services like Octavia and Neutron extension offering VPNaaS, fit perfectly as use cases. While studying this integration, though, it seemed clear that it was more efficient to take a more general approach, not focusing on an integration designed specifically for these services, but instead putting efforts in the analysis and possible integration of the protocols those services leverage on IPSec and in particular IKE, for the VPNaaS; and TLS-PSK for Octavia.
So addressing these protocols and their modes one by one I can conclude the following:

- IKEv1: it is supported by the VPNaaS extension and it can exploit QKD keys without any modification needed. The main problem, though, is that it is an old version with many known vulnerabilities, so it would be better to not use it. Anyway, it could prove as a valid option for a limited time, while administrators work on the IKEv2 implementation.
- IKEv2: it is supported by the VPNaaS extension but it needs minor changes to include the PPK extension, at the time of the writing offered only by LibreSwan. It has been proven to work correctly with a Quantum Key Server, considering the integration within LibreSwan, but the integration with the actual Neutron extension may be challenging.
- TLS-PSK (psk_dhe_ke): not supported natively by Octavia, it can be easily implemented and exploited by several applications thanks to the fact that it is supported by Stunnel. The main drawback of this option is that it uses Diffie-Hellman exchanges, not to be considered then fully quantum-safe. Indeed, this hybrid approach has to be considered a valid option in the short term but to be replaced before the next decade.
- TLS-PSK (psk_ke): not supported natively by Octavia, it surely represents a more challenging integration, both for Octavia, both for any other application, since they have to change their source code to call WolfSSL primitives. This cost is compensated, though, by the higher degree of security offered. Since it is leveraging only on quantum keys, as long as keys' entropy is sufficient (256 bits), it can be considered quantum-safe.

Tests highlighted another possible issue related to this integration. Any process execution time, related to the setup phase, has been heavily impacted by the methods that manage interactions between SAEs and Key Servers. In particular, GET_KEY() and GET_KEY_WITH_ID() calls could take alone between 2 and 4 seconds to be performed.
I concluded, though, that this cost in terms of execution time is acceptable, considering that this integration opens the possibility to a frequent key rotation, previously not contemplated. Moreover, execution times are still in the realm of times that modern quantum devices need to exchange a new key, so these overheads, at the actual state of the art, cannot be reduced.

In several chapters of this thesis, I stressed possible issues that could be solved by future works. In particular, I would like to highlight the main issue that was not solved due to time constraints but that could improve greatly the actual implementation: at the time of writing, this implementation offers the option to set up and bootstrap a secure connection among two SAEs, exploiting QKD keys. What is missing, though, is the possibility to issue a rekey on demand, or to configure a timeout to automatically invalidate actual keys and agree on a new set.
Moreover, I can also mention the possibility to fully integrate the Quantum Key Servers inside OpenStack, as described in the chapter 7, or even the option to directly integrate it with VPNaaS and Octavia. This last option is less recommended though, because OpenStack and VPNaaS, due to their nature as community projects, are continuously changing. So putting effort to natively integrate the Quantum Key Server with some OpenStack service, would probably soon be made unuseful by a newer release of the service.

Bibliography

[1] P. Shor, "Algorithms for quantum computation: discrete logarithms and factoring", Proceedings 35th Annual Symposium on Foundations of Computer Science, 1994, pp. 124–134, DOI 10.1109/SFCS.1994.365700

[2] M. Michele, "Cybersecurity in an era with quantum computers: Will we be ready?", IEEE Security Privacy, vol. 16, September 2018, pp. 38–41, DOI 10.1109/MSP.2018.3761723

[3] C. Bennett and D. DiVincenzo, "Quantum information and computation", Nature, vol. 404, March 2000, pp. 247–255, DOI 10.1038/35005001

[4] W. Kozlowski, S. Wehner, R. Van Meter, B. Rijsman, A. S. Cacciapuoti, M. Caleffi, and S. Nagayama, "Architectural principles for a quantum internet draft-irtf-qirg-principles-05", IEEE Communications Standards Magazine, vol. 4, September 2020, pp. 4–6, DOI 10.1109/mcomstd.2020.9204590

[5] W. Wootters and W. Zurek, "A single quantum cannot be cloned", Nature, vol. 299, October 1982, pp. 802–803, DOI 10.1038/299802a0

[6] M. Roetteler and K. M. Svore, "Quantum computing: Codebreaking and beyond", IEEE Security Privacy, vol. 16, September 2018, pp. 22–36, DOI 10.1109/MSP.2018.3761710

[7] D. Bernstein and T. Lange, "Post-quantum cryptography", Nature, vol. 549, September 2017, pp. 188–194, DOI 10.1038/nature23461

[8] V. Lyubashevsky, "Lattice signatures without trapdoors", vol. 7237, 2012, pp. 738–755, DOI 10.1007/978-3-642-29011-4_43

[9] H. Zhang, Z. Ji, H. Wang, and W. Wu, "Survey on quantum information security", China Communications, vol. 16, October 2019, pp. 1–36, DOI 10.23919/JCC.2019.10.001

[10] A. I. Nurhadi and N. R. Syambas, "Quantum key distribution (qkd) protocols: A survey", 2018 4th International Conference on Wireless and Telematics (ICWT), Nusa Dua, July 2018, pp. 1–5, DOI 10.1109/ICWT.2018.8527822

[11] A. Aspect, "Bell's inequality test: more ideal than ever", Nature, vol. 398, March 1999, pp. 189–190, DOI 10.1038/18296

[12] L. Alexander, M. Peloso, I. Marcikic, L. Antia, and K. Christian, "Experimental e91 quantum key distribution", Advanced Optical Concepts in Quantum Computing, Memory, and Communication (Z. U. Hasan, A. E. Craig, and P. R. Hemmer, eds.), March 2008, DOI 10.1117/12.778556

[13] European Telecommunications Standards Institute, "Implementation Security of Quantum Cryptography: Introduction, challenges, solutions", June 2015, https://www.etsi.org/images/files/ETSIWhitePapers/etsi_wp27_qkd_imp_sec_FINAL.pdf

[14] European Telecomunications Standards Institute, "Quantum Safe Cryptography and Security: An introduction, benefits, enablers and challenges", June 2015, https://www.etsi.org/images/files/ETSIWhitePapers/QuantumSafeWhitepaper.pdf

[15] C. Kaufman, P. E. Hoffman, Y. Nir, P. Eronen, and T. Kivinen, "Internet Key Exchange Protocol Version 2 (IKEv2)." RFC 7296, October 2014, DOI 10.17487/RFC7296

[16] E. Rescorla, B. Korver, and Internet Architecture Board, "Guidelines for writing rfc text on security considerations." IETF, July 2003, DOI 10.17487/rfc3552

[17] E. Rescorla, "The transport layer security (tls) protocol version 1.3." IETF, August 2018, DOI 10.17487/rfc8446

[18] R. Housley, "Tls 1.3 extension for certificate-based authentication with an external pre-shared key." IETF, March 2020, DOI 10.17487/rfc8773

[19] M. Elboukhari, M. Azizi, and A. Azizi, "Integration of quantum key distribution in eap-tls protocol used for wireless lan authentication", December 2010, DOI 10.1109/isvc.2010.5656266

[20] S. Patidar, D. Rane, and P. Jain, "A survey paper on cloud computing", 2012 Second International Conference on Advanced Computing Communication Technologies, January 2012, pp. 394–398, DOI 10.1109/acct.2012.15

[21] P. Patel, V.Tiwari, and M. Abhishek, "SDN and NFV integration in openstack cloud to improve network services and security", 2016 International Conference on Advanced Communication Control and Computing Technologies (ICACCCT), May 2016, pp. 655–660, DOI 10.1109/icaccct.2016.7831721

[22] A. Aguado, E. Hugues-Salas, P. Anthony Haigh, J. Marhuenda, A. B. Price, P. Sibson, J. E. Kennard, C. Erven, J. G. Rarity, M. G. Thompson, A. Lord, R. Nejabati, and D. Simeonidou, "Secure nfv orchestration over an sdn-controlled optical network with time-shared quantum key distribution resources", Journal of Lightwave Technology, vol. 35, April 2017, pp. 1357–1362, DOI 10.1109/jlt.2016.2646921

[23] Y. Li and M. Chen, "Software-defined network function virtualization: A survey", IEEE Access, vol. 3, 2015, pp. 2542–2553, DOI 10.1109/access.2015.2499271

[24] D. Ageyev, O. Bondarenko, W. Alfroukh, and T. Radivilova, "Provision security in SDN/NFV", 2018 14th International Conference on Advanced Trends in Radioelecrtronics, Telecommunications and Computer Engineering (TCSET), February 2018, pp. 506–509, DOI 10.1109/tcset.2018.8336252

[25] M. Pattaranantakul, R. He, Q. Song, Z. Zhang, and A. Meddahi, "NFV security survey: From use case driven threat analysis to state-of-the-art countermeasures", IEEE Communications Surveys & Tutorials, vol. 20, no. 4, 2018, pp. 3330–3368, DOI 10.1109/comst.2018.2859449

[26] R. Kumar, N. Gupta, S. Charu, K. Jain, and S. Jangir, "Open source solution for cloud computing platform using openstack", 05 2014, DOI 10.13140/2.1.1695.9043

[27] OpenStack Community, "Secure reference architectures", February 2021, https://docs.openstack.org/security-guide/secure-communication/secure-reference-architectures.html

[28] OpenStack Community, "Networking architecture", February 2021, https://docs.openstack.org/security-guide/networking/architecture.html

[29] OpenStack Community, "Openstack networking", July 2020, https://docs.openstack.org/neutron/pike/admin/intro-os-networking.html

[30] O. Tkachova, M. J. Salim, and A. R. Yahya, "An analysis of SDN-OpenStack integration", 2015 Second International Scientific-Practical Conference Problems of Infocommunications Science and Technology (PIC S&T), October 2015, pp. 60–62, DOI 10.1109/infocommst.2015.7357269

[31] OpenStack Community, "Virtual private network-as-a-service (vpnaas) scenario", July 2018, https://docs.openstack.org/neutron/latest/admin/vpnaas-scenario.html

[32] Openstack Community, "Introducing octavia", May 2020, https://docs.openstack.org/octavia/queens/reference/introduction.html

[33] "Quantum key distribution (qkd) application interface", European Telecommunications Standards Institute, August 2020. https://www.etsi.org/deliver/etsi_gs/QKD/001_099/004/02.01.01_60/gs_QKD004v020101p.pdf

[34] HashiCorp, https://www.hashicorp.com/products/vault

[35] Openstack Community, https://docs.openstack.org/kolla-ansible/latest/

[36] H. Coullon, C. Perez, and D. Pertin, "Production deployment tools for IaaSes: An overall model and survey", 2017 IEEE 5th International Conference on Future Internet of Things and Cloud (FiCloud), August 2017, pp. 183–190, DOI 10.1109/ficloud.2017.51

[37] Microsoft, "Microservices architecture style", October 2019, https://docs.microsoft.com/en-us/azure/architecture/guide/architecture-styles/microservices

[38] "Openstack octavia, creating and monitoring a load balancer", April 2020, https://leftasexercise.com/2020/05/04/openstack-octavia-creating-and-monitoring-a-load-balancer/

[39] Nginx, https://www.nginx.com/

[40] OpenStack Community, "Tls data security and barbican", June 2021, https://docs.openstack.org/octavia/latest/contributor/specs/version0.5/tls-data-security.

html

[41] S. Fluhrer, P. Kampanakis, D. McGrew, and V. Smyslov, "Mixing preshared keys in the internet key exchange protocol version 2 (ikev2) for post-quantum security", June 2020, DOI 10.17487/rfc8784

Appendix A

User's Manual

In this chapter, you can find guidelines to reproduce the testbed for each test introduced in chapter 10. It will be respected in the same order in which they were presented: first the IKE test cases and then the two TLS ones (Stunnel before and then WolfSSL).
All commands shown in the following sections assume that the reader is using *Ubuntu 18.04.5 LTS (Bionic Beaver)* as OS for theirs VMs.

A.1 IKE integration

First things first, the OS should be updated. Afterwards, dependencies to build the LibreSwan toolkit must be installed:

```
sudo apt-get update

# install dependecies to build libreswan
apt-get install libnss3-dev libnspr4-dev pkg-config libpam-dev \
       libcap-ng-dev libcap-ng-utils libselinux-dev \
       libcurl3-nss-dev flex bison gcc make libldns-dev \
       libunbound-dev libnss3-tools libevent-dev xmlto \
       libsystemd-dev
```

Then an additional library must be installed to build the custom LibreSwan containing changes proposed in this work. Indeed, new functions need to use the libjson-c library that is not exploited by the standard LibreSwan.

```
sudo apt install libjson-c-dev
```

Finally, the LibreSwan toolkit can be built and installed (assuming it is contained in a folder named *custom-libreswan*)

```
cd custom-libreswan
sudo make programs
sudo make install
```

Before running the ipsec service, though, you have to create and fill accordingly the two configuration files needed at LibreSwan startup:

```
sudo touch /etc/ipsec.conf
sudo touch /etc/ipsec.secrets
```

It can be now started by the ipsec service that will then used to add the connection defined in *ipsec.conf* to the pluto's database. The last operation to perform will be to set up the connection so that pluto will process the secrets and execute the new workflow introduced in this work. It must be stressed that this process will be on hold until the peer host's ipsec service will also be similarly configured and boot up.

```
# start ipsec service
systemctl start ipsec.service

# add connection (assuming it is named my_conn) to pluto's database
ipsec auto add my_conn

# boot up the connection
ipsec auto up my_conn
```

A.2 Stunnel integration

To reproduce the Stunnel test case you must before download and install Stunnel. Even it will be the python script in charge to manage Stunnel's lifecycle, you still have the responsibility to configure Stunnel.
In particular, your task will be to edit the *stunnel.conf* to edit the path to be followed by Stunnel to reach the file where all secrets are stored.

```
# download Stunnel
sudo wget https://www.stunnel.org/downloads/stunnel-5.59.tar.gz \
--no-check-certificate

# unzip, build and install it
sudo gzip -dc stunnel-5.59.tar.gz | tar -xvf -
cd stunnel-5.59/
./configure
sudo make
sudo make install

# create log file
cd /usr/local/var
sudo mkdir log
sudo chmod 777 log

# create a custom configuration file starting from the example one
# edit filepath to log file and to secrets file
cd /usr/local/etc/stunnel
sudo cp stunnel.conf-sample stunnel.conf
sudo nano stunnel.conf
```

After having installed Stunnel, you have to configure accordingly the python script, developed during this work, meant to handle it. In particular, you have to be sure that you are inserting fetched keys in the same secrets file you pointed to in the Stunnel configuration file.
Once the configuration is done, you can simply launch the python script acting as main:

```
python3 worker.py
```

It must be stressed that it is important to launch the script before in the responder node and then in the initiator one.

A.3 WolfSSL integration

First steps to perform consist in the installation of library dependencies by WolfSSL, considering also the libjson-c library and the libcurl library, both used by the proposed functions developed to allow interactions among SAEs and QKS.

```
# WolfSSL dependencies
sudo apt install build-essential

# new methods dependencies
sudo apt install libjson-c-dev
sudo apt-get install libcurl4-openssl-dev

# needed to unzip WolfSSL
sudo apt-get install unzip
```

Assuming you already downloaded WolfSSL, having now a zip package, you can issue the following commands to unzip, configure, build and install it (you can notice that in this work was used the 4.7 version).

```
unzip wolfssl-4.7.0.zip
cd wolfssl-4.7.0
# TLSv1.3 is enabled by default, but PSK suites must be enabled
# instead during the configuration phase
./configure --enable-psk
sudo make
sudo make install
```

This work used four scripts to perform WolfSSL test. Two of them are meant to be executed in the node acting as initiator (*client_setup.c* and *test_client.c*), while the remaining two are supposed to be executed by the responder node (*server_setup.c* and *test_server.c*).
Before executing them though, users must properly fill in the configuration file *config.ini* and finally exploit the already offered Makefile to build and compile them.
It must be stressed that there is an order of execution to be followed: first both setup scripts must be run (*client_setup.c* and *server_setup.c*) and only if they both exit succesfully, you can execute the scripts handling WolfSSL connection (*test_client.c* and *test_server.c*.

```
# Initiator node

# build and compile sources
make

# start setup phase
./client_setup

# start WolfSSL connection (to be launched after correct execution of
    previous command)
./test_client
```

The same set of commands, using the other couple of scripts, must be issued in parallel on the responder node.
In both cases, scripts will write logs in the */log* directory, that must be present in the same folder where they are executed.

Appendix B

Developer's Manual

In this chapter, changes to LibreSwan and WolfSSL toolkits, needed to reproduce the proposed solution to obtain the QKS integration, will be briefly presented. Moreover, it will be also introduced the Python script used to allow QKS integration with the Stunnel tool.

B.1 IKE integration

LibreSwan relies mainly on a daemon process, called *pluto*, which manage secure channels through their whole lifecycle. The user can interact with pluto using an interface offering methods to request the creation or deletion of a secure channel.
Beforehand, though, the user must fill two files used by LibreSwan to set up channels and collect secrets. These are the *ipsec.conf* and the *ipsec.secrets* files. The former one contains information related to the channel configuration such as endpoint addresses, protocol to be used and kind of secrets to be exploited.
In this case, it is important to configure hosts participating in the exchange to propose or even insist on the use of PPKs.
The ipsec.secrets file, instead, contains the complete list of secrets that this host owns to establish channels. Each line has its different format, and in particular, PPKs entry must feature two IP addresses separated by white space, followed by a colon, the *PPKS* keyword and finally two fields where are stored respectively the PPK ID and the PPK itself.
In the proposed implementation, we have been exploiting these last two fields to carry data needed to perform QKS and SAEs interactions.
First things first, the *secrets.c* file was modified to insert a new flow to be executed whenever a PPK entry is parsed from the ipsec.secrets file. This new workflow is guarded by a boolean variable called *debugKS*. Developers must be aware, though, that this value cannot be modified at runtime, therefore if you want to switch from the legacy workflow to the new one, and vice versa, you need to modify debugKS value and then recompile LibreSwan.

```
else if (tokeqword("ppks")) {
        llog(RC_LOG_SERIOUS, flp->logger, "PPK LINE FOUND");
        s->pks.kind = PKK_PPK;

        int debugKS = 1;
        if (!debugKS) {
                ugh = !shift(flp) ? "ERROR: unexpected end of record in static
                    PPK" : process_ppk_static_secret(flp, &s->pks.ppk,
                    &s->pks.ppk_id);
        }
        else {
                ugh = !shift(flp) ? "ERROR: unexpected end of record in static
                    PPK" : custom_process_ppk(flp, &s->pks.ppk,
                    &s->pks.ppk_id);
```

```
}
```

In the new flow, a custom function, named *custom_process_ppk* is called, passing the cursor pointing to the current position in the file, as well as two pointers that pluto is expecting to be filled by the PPK and its PPK ID.
This custom function gathers data from the file, using the current cursor obtained as a parameter. This data is needed to correctly perform QKS and SAEs interactions (as described in 8.3). If all values have been correctly parsed, then exchanges are performed and finally, the key obtained by the QKS, together with its ID are stored in the portions of memory pointed by pointers that were passed as parameters.

New flow's behaviour is slightly different whether the host performing it is acting as an initiator or as a responder. In the former case, it will perform the GET_KEY() request to the QKS, then it will send to its peer the corresponding key handle and finally, it will wait for an ACK.
In the latter case, instead, it will first open a socket to listen for an incoming connection, retrieve the key handle and then perform a GET_KEY_WITH_ID() request using it. Finally, in case of successful execution, it will send back an ACK to the client.
Both cases share a phase, precedent to the described one, where they communicate with Keycloak to obtain the token that will be used in the following request to the QKS.

```
int get_token(struct MemoryStruct* chunk, const char* keycloak_ip, const
    char* keycloak_port, const char* keycloak_client_id, const char*
    keycloak_sec);

int get_key(struct MemoryStruct* chunk, const char* token, const char*
    key_server_ip, const char* key_server_port, const char* target_sae, const
    char* this_sae_str);

int get_key_with_id(struct MemoryStruct* chunk, const char* token, const
    char* key_id, const char* key_server_ip, const char* key_server_port,
    const char* target_sae);
```

Since LibreSwan is developed in C language, we were constrained to the use of this programming language while coding methods to manage these interactions. C language does not feature built-in libraries to handle network communications over HTTP and network objects such as JSON ones. To overcome this issue, libraries libcurl and json-c were installed and added to LibreSwan.
In the following snippet, which is taken from the body of the GET_KEY() method, you can see how methods from these libraries were used to compose the HTTP message and generate valid JSON objects to attach.

```
/* init the curl session */
    curl_handle = curl_easy_init();

    /* specify URL to get */
    buffer = (char*)malloc(sizeof(char)*255);
    sprintf(buffer, "http://%s:%s/api/v1/keys/%s/enc_keys", key_server_ip,
        key_server_port, target_sae);
    curl_easy_setopt(curl_handle, CURLOPT_URL, buffer);

    /* Now specify the POST data */

    post_data = json_object_new_object();
    number = json_object_new_int(1);
    klen = json_object_new_int(128);
    this_sae = json_object_new_string(this_sae_str);
```

```
    json_object_object_add(post_data, "number", number);
    json_object_object_add(post_data, "size", klen);
    json_object_object_add(post_data, "SAE_ID", this_sae);

    curl_easy_setopt(curl_handle, CURLOPT_POSTFIELDS,
        json_object_to_json_string(post_data));
```

B.2 Stunnel integration

Stunnel license does not allow the publication of a custom version of the tool, therefore it was treated as a black box, designing and developing an external Python script to manage both interactions with QKS and SAEs, both Stunnel's lifecycle. What the developer must know is that Stunnel, during its startup phase, retrieves secrets from a file which path can be edited in its configuration settings.
Before running the Python scripts, developers must also configure properly the configuration file, named *bootConn.conf*, where data related to SAEs and QKS are stored.

```
IPaddr = 192.168.0.247     # IP where bind to listen
port = 21021      # Port to be binded for plain text exchanges
keyServIP = 192.168.0.249     # Key server IP address
keyServPort = 4000     # Key server port
dockStunnel = stunnel     # Stunnel container's name
stunnPsk = /home/ubuntu/psk.txt    # Stunnel file where ID:PSK pairs are
    stored
keycloakIP = 192.168.0.249     # IP where keycloak is hosted
keycloakPort = 8080     # port where keycloak is bind
keycloakRealm = quantum_auth    # keycloak realm name
clientSaeID = SAE11223344     # client sae ID registered on keycloak
clientCredentials = 3febe46c-718a-44ec-a730-e0ba0ac3986f   # client
    credentials got from keycloak
sender = 1      # sender is 0 -> do not start key handle exchange

kNum = 1          # num of keys requested
kLen = 128          # key length desired

targetSAE = SAE55667788       # target SAE ID
targetIP = 192.168.0.237        # target SAE IP
targetPort = 21021         # target SAE port
```

Once the configuration file is ready, developers can launch the main script called *worker.py*. This script parse configuration data and launch threads managing the various interactions. One thread manages a TCP server used to process incoming requests, retrieving key handles and inserting them in a queue shared with another thread acting as a consumer.
The latter thread performs a *GET_KEY_WITH_ID()* request, then it stores the key in the secrets file where Stunnel looks for keys. Finally, it checks if Stunnel is already running. If so, it sends a *SIGHUP* signal to Stunnel's process to force its reconfiguration; otherwise, it launches a new Stunnel instance.

```
while True:

        msg = qPlain.get()
        c, address, port, keyID = msg

        # get key from Key Server for key ID
```

```
        key = getFromKeyServer(keyServerIP, keyServerPort, keyID,
            configs.get('targetSAE'), configs.get('serverSAE'),
            configs.get('kNum'), configs.get('kLen'), oauthHeader)

        # edit secretsFp (dummy version, in actual version should store the
            key for tls-psk server)
        stunnPskLock.acquire()
        try:
            stunnPsk.addOnlyLast(keyID, key)
        finally:
            stunnPskLock.release()
        timer.stop("Key parsed and stored on file")

        timer.start()
        if (checkIfProcessRunning("stunnel")):
            lPids = findProcessIdByName("stunnel")
            stunnPid = lPids[0]['pid']
            reconfProc = subprocess.Popen(["sudo", "kill", "-SIGHUP",
                str(stunnPid)])
            reconfProc.wait()
            print('Stunnel correctly reconfigured')
            timer.stop("Stunnel reconfigured")
        else:
            stunnelThread = Thread(target = startStunnel,)
            stunnelThread.start()
            timer.stop("Stunnel started")

        timer.start()
        # reply to client using c and close connection
        reply = "OK DONE!\r\n"
        c.send(reply.encode())
        c.close()
        timer.stop("Sent ACK to client", True)

        # to debug and test only one connection configured then exits
        break
```

worker.py initial behavior depends on the value assigned to the *sender* boolean variable. This variable must be set to "1" in the host intended to be used as an initiator (the one performing the *GET_KEY()*), and to "0" in the host meant to serve as responder (performing the *GET_KEY_WITH_ID()*. In both cases, the thread managing the TCP server and the other one acting as a consumer are both running. In the initiator case, though, it is performed an additional portion of code where the first half of the workflow is executed (perform the GET_KEY() and send the key handle to the target SAE).

```
# if sender == 1 then get key and send handle to peer
   if (int(configs.get('sender'))):
        keyID = getFromKeyServer(configs.get("keyServIP"),
            configs.get("keyServPort"), -1, configs.get('targetSAE'),
            configs.get('clientSaeID'), int(configs.get('kNum')),
            int(configs.get('kLen')), oauthHeader)

        timer.start()
        s = socket.socket(socket.AF_INET, socket.SOCK_STREAM) # create TCP
            socket
        s.connect((configs.get('targetIP'), int(configs.get('targetPort'))))
        s.sendall(('SET KEY ID=' + keyID + '\r\n\r\n').encode('utf-8'))
```

```
        timer.stop("Handle sent to server")
```

B.3 WolfSSL integration

Similar to the Stunnel case, also for WolfSSL developers must beforehand fill out a configuration file, named in this case *config.ini* containing the same set of data shown in the previous case. There are two fields that are related only to the WolfSSL test and are the *retry_conn_sec* and the *ntest_msg*. The former one defines how many seconds the program should wait before trying again to establish a TCP connection. The latter one instead must be set to the number of messages desired to be exchanged after the secure channel establishment.

```
[Keycloak]
keycloak_ip = 192.168.0.236
keycloak_port = 8080
kc_client_id = SAE55667788
kc_client_secret = 1dc54431-d9c7-4ecb-8886-8eb9ee89358d

[KeyServer]
keyserver_ip = 192.168.0.236
keyserver_port = 4000

; data related to SAEs involved in key exchange
[SAEs]
target_sae_ip = 192.168.0.244
target_sae = SAE11223344
this_sae = SAE55667788

; data related to socket used to listen, accept and retrieve key handle
[Socket data]
socket_port = 21021
retry_conn_sec = 2
ntest_msg = 10
```

Once the configuration file is correctly set up, the developer must launch before a first script, to retrieve keys and perform key agreement; and then a second one to set up the secure channel and exploit it to exchange *ntest_msg* messages over it.
Initiator's side and respoonder's one are distinguished by the script launched. Indeed there are four scripts in total: two for setup and connection phase, acting as initiator, called *client_setup.c* and *test_client.c*; and then other two scripts handling similar operations, but from the responder's side, called *server_setup.c* and *test_server.c*. Operations performed and methods called in the setup phase (both in initiator and responder case) are identical to those one seen in the IKE case. It will now introduce then more in detail the scripts handling the connection phase.
Starting from the initiator side, the keys obtained in the previous phase are retrieved and then stored in a portion of memory that will be accessed in a second moment by the TLSv1.3 callback. Then it will try to create a plain TCP connection with the responder, using the *custom_tcp_connect* function. When this connection is established, the client can then set up the WolfSSL context and init a WolfSSL connection over the plain TCP socket. In the following snippet you can find the *custom_tcp_connect* (modified to re-issue new connection attempts until one succeeds), and the *custom_psk_client_tls13_cb* which set up the TLS-PSK (in psk_ke mode) using the QKD key.

```
static WC_INLINE void custom_tcp_connect(SOCKET_T* sockfd, const char* ip,
    word16 port,
                            int udp, int sctp, WOLFSSL* ssl, int retry_sec)
{
    SOCKADDR_IN_T addr;
```

```
    build_addr(&addr, ip, port, udp, sctp);
    if (udp) {
        wolfSSL_dtls_set_peer(ssl, &addr, sizeof(addr));
    }
    tcp_socket(sockfd, udp, sctp);

    if (!udp) {
        while (connect(*sockfd, (const struct sockaddr*)&addr, sizeof(addr))
            != 0) {
            log_error("TCP connect failed - retransmitting in %d...\n",
                retry_sec);
            sleep(retry_sec);
        }
    }
}

static WC_INLINE unsigned int custom_psk_client_tls13_cb(WOLFSSL* ssl,
        const char* hint, char* identity, unsigned int id_max_len,
        unsigned char* key, unsigned int key_max_len, const char**
            ciphersuite) {
    int we;
    int b = 0x01;
    const char* userCipher = (const char*)wolfSSL_get_psk_callback_ctx(ssl);

    (void)ssl;
    (void)hint;
    (void)key_max_len;

    XSTRNCPY(identity, myKeyID, strlen(myKeyID));
    XSTRNCPY(key, myKey, strlen(myKey));

    *ciphersuite = userCipher ? userCipher : "TLS13-AES128-GCM-SHA256";

    return 32; /* length of key in octets or 0 for error */
}
```

Considering now the responder side, a similar pattern of operations is performed, except for the TCP connection since in this case, it has to bind a socket to listen and accept incoming connections. Also, the TLSv1.3 callback behaves slightly differently in this case. Indeed, in this case, it has also to compare the identity declared by the client with all known identities. If it is found a matching identity, then it sets its key as the one to be used in the following exchanges.

```
static WC_INLINE unsigned int custom_psk_server_tls13_cb(WOLFSSL* ssl,
        const char* identity, unsigned char* key, unsigned int key_max_len,
        const char** ciphersuite)
{
    int we;
    int b = 0x01;
    const char* userCipher = (const char*)wolfSSL_get_psk_callback_ctx(ssl);

    (void)ssl;
    (void)key_max_len;

    if (XSTRNCMP(identity, myKeyID, XSTRLEN(myKeyID)) != 0)
        return 0;
    XSTRNCPY(key, myKey, strlen(myKey));
```

```
    *ciphersuite = userCipher ? userCipher : "TLS13-AES128-GCM-SHA256";

    return 32; /* length of key in octets or 0 for error */
}
```