MASTER'S THESIS

An approach to securing IPsec with Quantum Key Distribution (QKD) using the AIT QKD software

carried out at

Course Programme
Information Technologies and IT Marketing

By: Stefan Marksteiner
PID:1210320006

Graz, 26th March 2014

.....
Stefan Marksteiner

Statutory Declaration

I hereby declare that I have authored this thesis independently, that I have not used other than the declared sources and resources and that I have explicitly marked all material which has been quoted either literally or by content from the used sources.

.....
Stefan Marksteiner

Ehrenwörtliche Erklärung

Ich erkläre ehrenwörtlich, dass ich die vorliegende Arbeit selbstständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen nicht benützt und die benutzten Quellen wörtlich zitiert sowie inhaltlich entnommene Stellen als solche kenntlich gemacht habe.

.....
Stefan Marksteiner

Acknowledgements

I like to use this space to thank this thesis' advisor Oliver Maurhart for spending his precious time. I also like to thank Christian Kollmitzer for initiating this thesis and advice with cryptographic algorithms as well as Arno Hollosi for the latter. I further thank my friend Hannes Fritz for his help in the final phase of this thesis. Most of all, I like to thank my wife Astrid Marksteiner for her ongoing help, care and support and above all endless patience, without which this thesis would not have been possible.

Stefan Marksteiner

Graz, 26th March 2014

Abstract

Quantum Key Distribution (QKD) is a cutting-edge security technology that provides mathematically proven security by using quantum physical effects and information theoretical axioms to generate a guaranteed non-disclosed stream of encryption keys. Although it has been a field of theoretical research for some time, it has only been producing market-ready solutions for a short period of time, and there are very few research groups working in this field, including the Optical Quantum Technologies group of the Austrian Institute of Technology (AIT). The downside of this technology is that its key generation rate is only around 12,500 key bits per second. As this rate limits the data throughput to the same rate, it is substandard for normal modern communications, especially for securely interconnecting networks. Internet Protocol security (IPsec), on the other hand, is a well-known security protocol that uses classical encryption and is capable of exactly creating site-to-site virtual private networks. This thesis provides an overview of the fundamentals of both technologies and presents a solution which combines the performance advantages of IPsec with QKD. The combination sacrifices only a small portion of QKD security by using the generated keys a limited number of times instead of just once. As a part of this, the thesis answers the question of how many data bits per key bit make sensible upper and lower boundaries to yield high performance while maintaining high security. While previous approaches complement the Internet Key Exchange protocol (IKE), this thesis simplifies the implementation with a new key synchronization concept. Furthermore, it provides a Linux-based module for the AIT QKD software using the Netlink XFRM Application Programers Interface, for which it also provides a detailed explanation, to feed the quantum key to the IPsec cipher. This enables wire-speed, QKD-secured communication links for business applications.

Kurzfassung

Quantum Key Distribution (QKD) ist eine Sicherheitstechnologie, die mathematisch beweisbar sichere Verschlüsselung mit Hilfe von physikalischen Effekten und informationstheoretischen Axiomen durch zur Verfgügung stellt. Obwohl bereits seit einiger Zeit Gegenstand theoretischer Forschung, produziert QKD erst seit kurzer Zeit marktgerechte Lösungen. Es gibt nur weltweit nur wenige Forschungsgruppen in diesem Feld wie die Gruppe der Optical Quantum Technologies Gruppe des Austrian Institute of Technology (AIT). Der Nachteil dieser Technologie ist, dass die Schlüsselrate nur bei rund 12.500 Schlüsselbits pro Sekunde liegt. Da dies die Datenrate auf denselben Wert limitiert, ist dies Substandard für moderne Kommunikation, speziell für Site-to-Site-Verbindungen. Internet Protocol security (IPsec), andererseits, ist ein weit verbreiteteter Standard, der mittels klassicher Verschlüssellung genau diesen Zweck erfüllt. Die vorliegende Arbeit gibt einen Überblick über die Grundlagen beider Technologien und zeigt eine Lösung um die Vorteile beider zu kombinieren. Diese opfert einen kleines Quantum an Sicherheit indem der Schlüssel einige wenige Male statt nur einmal verwendet wird. Die Arbeit behandelt dabei die Frage wie viele Datenbits pro Schlüsselbits sinnvoll für hohe Leistung bei hoher Sicherheit sind. Während frühere Ansätze das Internet Key Exchange protocol (IKE) ergänzen, vereinfacht die vorliegende Arbeit die Schlüsselsynchronisation mit einem neuen Konzept. Darüber hinaus liefert sie ein Linux-basiertes Modul für die AIT QKD Software, dass mittels dem Netlink XFRM Application Programmers Interface, welches detailiert beschrieben wird, IPsec mit einem Quantenschlüssel versorgt. Dies ermöglicht Wire-Speed QDK-gesicherte Verbindung für kommerzielle Anwendungen.

Contents

1 Introduction

The main purpose of cryptographic protocols is to secure confidentiality for the communications between two partners. Contemporary classical (as opposed to quantum) cryptography, however, relies on assumptions about the computing power of an attacker and the difficulty of the underlying mathematical problem. Both factors are subject to a degree of uncertainty, as no proof exists for these problems and a hypothetical quantum computer may solve them in reasonable time. (Zbinden, Bechmann-Pasquinucci, Gisin, & Ribordy, 1998, p.743) Quantum Key Distribution (QKD), on the other hand, is information-theoretically secure, as chapter 2 will show. QKD provides two systems, connected by a quantum channel, with keys (more precisely a continuous key stream) to encrypt data to be sent over a public network (for instance, the Internet). (Nielsen & Chuang, 2000, p.583) Internet Protocol security (IPsec) is a widespread security protocol suite which provides integrity, authenticity and confidentiality for data connections (Kent & Seo, 2005, p.4). The purpose of this thesis is to interconnect these two systems by using QKD to provide IPsec with the cryptographic keys necessary for its operation.

1.1 Motivation

The Austrian Institute of Technology (AIT) has developed a QKD solution consisting of hardware devices and a multi-platform hardware-independent software. This solution secures the traffic of QKD using peers by applying the derived keys with a One-Time Pad (OTP) on the data on the mutual communications channel.

Since physical and technical factors limit the key generation rate of QKD to around twelve kilobits per second (Treiber et al., 2009, p.9), which is substantially lower than current data rates, QKD-derived key material is considered *expensive*. In IPsec, the QKD keys will be reused, therby saving key material. This fact, along with the proliferation of IPsec, serves as motivation to integrate an IPsec module into the AIT QKD software, as this is expected to raise the market acceptance of the solution.

1.2 Objectives

The main objective of this thesis is to answer the following research question:

How is it possible, to quantum-cryptographically secure IPsec with the assistance of a state-of-the-art QKD stack?

The ultimate goal is to deliver a fully operational IPsec module for the AIT QKD software. In order to fulfill this objective, the following subquestions need to be clarified:

- What is the minimum acceptable frequency of changing the IPsec key that will ensure sufficient security?
- What is the maximum acceptable frequency of changing the IPsec key to save QKD key material?
- Is the native Linux kernel implementation suitable for this task?

Secondary objectives of this thesis include descriptions of

- The fundamentals of QKD;
- The AIT QKD software;
- The IPsec protocol suite;
- The development tools used;
- The complete solution (QKDIPsec) including prototyping and testing.

1.3 Approach and Methodology

The thesis is divided into a theoretical and a practical part. The explanations of QKD, IPsec and the used development tools, which were derived from an examination of primary materials, form chapters 2, 3 and 4. These explanations, worked out by uses of primary materials form the theoretical part. The structure and references of the theoretical QKD part are partially taken from Kollmitzer and Pivk (2010). The practical part begins with the answer of the sub-questions in section 1.2. The first two questions include the definition of suitable security goals for the solution and the definiton of upper and lower boundaries for the IPsec key changes. These limits were completely undefined at the starting time of this thesis. The third question involves the survey of said implementation and, if the answer turns out to be no, the search for a suitable alternative. With this questions solved, the development of an IPsec module for the AIT QKD software follows, using C++ programming under Linux similar to the base software itself. The module consists of interfaces and management processes between the software and the system's IPsec protocol stack. Subsequently the module is integrated into the main software, accompanied by prototyping and testing, including an appropriate evaluation. The relationship between development, integration and testing is not to be seen as a strict sequential, but rather an iterative and agile process. This solution relies on a QKD engine that is capable of handling multiple clients at once with the help of application identifiers. Since this feature will first be implemented into the AIT QKD software in a future step, the present thesis simulates the QKD key delivery behavior with an own class. The very same class will serve as an adapter for this module to the QKD engine when the relevant functions are implemented.

1.4 Distinction from Previous Approaches

Although QKD is a relatively new field for practical research, there have been some efforts to combine IPsec and QKD. In summary, all of them supplement Internet Key Exchange (IKE) in some way or use an approach to combine classical and quantum-derived keys. This thesis presents its own key synchronization mechanism, completely omitting IKE. Also, some of the other approaches work at significantly lower key change rates. A more detailed description of previous work in this field will follow later in section 5.1, after essential preliminary information has been introduced.

1.5 General Preliminary Remarks and Definitions of Terms

This thesis assumes a certain degree of general technical and scientific knowledge. It does not, however, assume that the reader is an expert in the fields covered by this thesis. Special notations, such as the Dirac notation, are explained in respective sections, yet the reader is expected to have knowledge of general mathematical notations. The glossary contains all abbreviations which are not considered common knowledge. The italic terms *must, must not, should, should not* and may correspond to the capital terms MUST, MUST NOT, SHOULD, SHOULD NOT and MAY as defined by Brandner (1997, p1.). Non-italic uses of these words correspond to their normal use in everyday, non-technical language.

2 Fundamentals of Quantum Key Distribution

The purpose of Quantum Key Distribution (QKD) is to distribute a cryptographic key which is used to encrypt communications. These communications use a classical public channel to the communication partner, who subsequently decrypts it. Therefore QKD-secured communication uses two channels: one communications channel, called public channel, and one key distribution channel, called quantum channel (see figure 2.1). (Fehr, 2010, pp.504 - 506) The method to secure the public channel consists of symmetric encryption algorithms of which the most secure but also most costly form is One-Time Pad (OTP) (see section 2.1.2). The quantum channel, continuously providing key material for the symmetric encryption, is information theoretically secure (see section 2.1.1), as long as the key itself is truly secret (Nielsen & Chuang, 2000, p.583).

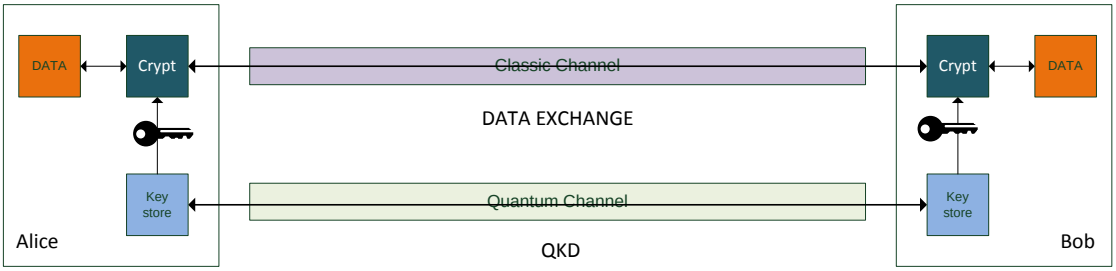


Figure 2.1: Basic QKD Architecture

A quantum channel consists ordinarily of a connection upheld by single photons. Usually the carrying media is an optical fiber cable (Zbinden et al., 1998, p.744) but there are also setups which use optical free space links (Ursin et al., 2007, p.1). The distance between the peers, however, is limited. The afore mentioned (experimental) free space link achieved a distance between the two peers of up to 144 km (Ursin et al., 2007, p.1), while fiber setups have, due to the absorption in the fiber, shorter ranges between 45 km and 70 km (Zbinden et al., 1998, p.778). This chapter describes the fundamentals of QKD including information theoretical and physical basics, descriptions of QKD protocols and efficiency and security enhancements and closes with a short description of the Austrian Institute of Technology (AIT) QKD software.

2.1 Information Theoretical Fundamentals

This section describes elementary concepts of information theory needed to understand the rest of the chapter. This includes information theoretic meaning of security, universal hash classes and the term of entropy.

2.1.1 Information-Theoretic Security

Many modern cryptographic algorithms, for instance the *Rivest/Shamir/Adlemann* (RSA) and *Diffie/Hellman* (DH) algorithms, rely on the assumption that there is no fast algorithm to solve certain mathematical problems and is therefore called *computational secure*. As for now, no such algorithm for the relevant problems is known, but the lack of it is also unproven. These algorithms are always only conditionally secure, as progress in both solving the underlying mathematical problems and computational power might compromise it. (Wolf, 1999, pp.217 - 218) An *information-theoretically secure*[1] system means that a system is still secure if an attacker has infinite resources and time at his disposal to cryptographically analyze it. (Shannon, 1949, pp.659) This type of security is also called unconditional security (Cachin & Maurer, 1997, p.292)

Perfect security additionally means that an interceptor of an encrypted message has the same amount of information after the interception *a posteriori* than before the interception *a priori* (Shannon, 1949, p.680). This means that an encrypted message E may contain any arbitrary clear text message M with the same probability $P_M(E)$, or that the probability of the occurence of any given cipher text $P(E)$ is independent of M as stated in equation 2.1. (Shannon, 1949, pp.680 - 681)

$$\forall M, E : P_M(E) = P(E) \tag{2.1}$$

The uncertainty principle of quantum mechanics guarantees the unconditional secrecy of the key in the quantum channel. (Cachin & Maurer, 1997, p.293) The secrecy on the public channel depends on the used mechanism. OTP (see section 2.1.2), guarantees perfect security.

2.1.2 One-Time Pad

One-Time Pad (OTP) is the only known unconditionally secure encryption algorithm. (Schartner & Kollmitzer, 2010, pp.177 - 178) The method suggests using the same randomly generated key sequence on both sides of a communications channel. (Vernam, 1926, p.13). The sending system performs an Exclusive Or (XOR)-operation [2] ($\oplus$) on the clear text message to encrypt

[1] Shannon used the term *secrecy* instead of security. In cryptography, more secrecy means more security (Zbinden et al., 1998, p.1). Thus, the two terms are synonymous in this context.

[2] Although it was not explicitly called XOR by Vernam, his algorithm for telegraph systems resembles this operation.

it, while the receiving system uses the same operation and sequence (or key) on the cipher text message to decrypt it (Vernam, 1919, p.3). To provide full secrecy, the key length has to be at least as long as the message to prevent key repetition (Vernam, 1926, p.15). Equation 2.2 illustrates the procedure, whereby m be the clear text message, c be the cypher text message and k be the key.

$$c_i = m_i \oplus k_i (mod 26) \tag{2.2}$$

OTP implements perfect security (see section 2.1.1), as the key length L_K and the number of letters in the key R_K is equal or higher than the message length L_M and the alphabet R_M (see equation 2.3). (Shannon, 1949, p.682)

$$R_M L_M \leq R_K L_K \tag{2.3}$$

QKD-links use OTP with the key stream distributed over the quantum channel to encrypt the data to be sent over the public channel. (Fehr, 2010, p.498)

2.1.3 Universal Hash Classes

Hash functions produce a short, fixed sized digest from a message of arbitrary length. These functions should be *preimage resistant*, so that it is not possible to compute a value x from z, where z is the hash value of x ($h(x) = z$), which means the function is *one-way*. (Pieprzyk, Hardjono, & Seberry, 2003, p.243) So, mathematically speaking, hash functions (f) are image functions which map values (x) from a large domain (A) into values (z) from a smaller or equally sized codomain (B) as described by equation 2.4. (Carter & Wegman, 1979, p.107)

$$h : A \to B, x \to h(x) = z\,;\ |A| \geq |B| \tag{2.4}$$

Naturally when the codomain is smaller than the domain, not every value from A has its unique counterpart in B, so that *collisions* occur, that is two values (x, y) from A receive the same value z from B ($h(x) = h(y)$). This is undesirable but inevitable. The ability of a hash algorithm to mitigate attempts to find an y which collides with a given x is called *second preimage resistance* or *weak collision resistance*, while the inability to find any arbitrary colliding pair of x and y is called *strong collision resistance* (Pieprzyk et al., 2003, pp.243-244).

Universal hash classes try to address this by choosing a random algorithm from a set of hash functions, whereby the members of this hash class have to be properly chosen (Carter & Wegman, 1979, p.106). The value of δ_h characterizes a hash function by its behavior regarding to collisions of a given x and y, whereby a value of *0* means no collision and *1* means a collision (see equation 2.5). (Carter & Wegman, 1979, p.107)

$$\delta_h(x,y) = \begin{cases} 1 & \dots \quad h(x) = h(y) \\ 0 & \dots \quad h(x) \neq h(y) \end{cases} \tag{2.5}$$

To characterize a set of hash functions (called *hash class*), the value δ_H sums up the δ_h values of its members. (Stinson, 1992, pp.77-78)[3]

$$\delta_H(x,y) = \sum_{f \in H} \delta_h(x,y) \tag{2.6}$$

A hash class is now called *universal*$_2$ if δ_H between x and y is lower or equal than the number of hash functions divided by the number of elements on which they map (see equation 2.7). (Carter & Wegman, 1979, p.107)

$$\delta_H(x,y) \leq \frac{|H|}{|B|} \tag{2.7}$$

This means that the overall probability for a collision between x) and y) is less than one $|B^{th}|$ ($P(h(x)) = h(y) \leq \frac{1}{|B|}$). In other words less than $\frac{1}{|B|}$ of the functions result in a collision for these values. (Carter & Wegman, 1979, p.107) The probability of a collision ($P(h(x)) = h(y)$ is $\frac{\delta_H(x,y)}{|H|}$) (Stinson, 1992, p.78). The class is further *strongly universal*$_n$ if the a randomly chosen function from that class distributes the probability of a distinctive value x_n to be mapped to a hash value z_n equally (probability = $\frac{1}{n}$). (Wegman & Carter, 1981, p.267)

$$\{h(x_1) = z_1, h(x_2) = z_2, .., h(x_n) = z_n\} = \frac{|H|}{B^n} \tag{2.8}$$

For a strongly universal$_2$ (SU$_2$) class this equation equals $\frac{|H|}{B^2}$. If the probability of $h(x_2) = z_2$ with knowledge of a given $h(x_1) = z_1$ has a value ε higher than $\frac{1}{n}$, the class is ε-almost strongly universal$_2$ (ε-ASU$_2$) (see equation 2.9). (Stinson, 1992, p.78)

$$\{h(x_1) = z_1, h(x_2) = z_2\} = \varepsilon \frac{|H|}{B} \tag{2.9}$$

The purpose of using only almost-strongly universal classes instead of strongly universal classes is practicability. Strongly universal classes contain many hash functions, requiring $\mathcal{O}(n)$ bits as function index to hash n-bit long names. In comparison, there are almost-strongly universal functions which require only $\log(n)$ bits. (Wegman & Carter, 1981, p.266)

[3] Originally, (Carter & Wegman, 1979, p.107) sum over over a set of ys to find every colliding y for a given x within a hash function. Thus $\delta_H(x,S) = \sum_{f \in H} \sum_{y \in S} \delta_h(x,y)$, where $y \in S$ and $S \subset A$.

2.1.3.1 The Use of Universal Hash Classes in Quantum Key Distribution

Universal Hash Classes may provide *unconditional security* in terms of authentication (Stinson, 1992, p.1). QKD uses ε-ASU$_2$ hash classes for authentication purposes. Two parties (*Alice* and *Bob*) share a common secret key k. This key indexes a distinctive hash function h to use out of a previously agreed, ordered ε-ASU$_2$ hash class (H), mapping $A \to B$. *Alice* sends an $x \in A$ and the accordingly indexed $h_k(x) = z \in B$ to *Bob*, who (possessing k) can easily verify the coherence of x and z (see equation 2.10).

$$z = h_k(x) \tag{2.10}$$

An attacker basically has to guess which function h of H is used by *Alice* and *Bob*. The attacker has a chance of $1/|B|$, as in a well-chosen class H the possibility of any x being mapped to a distinctive y is evenly distributed over the functions h. If the attacker brute-force calculates all matching functions h to given x/y-pair, the possibility is still only ε. (Pivk, 2010a, p.16) Therefore, ε has to be small (where $\frac{1}{B}$ is the natural boundary). There are functions which fulfill this requirement by splitting up messages or the key by giving a ε of $\frac{2}{|B|}$. (Wegman & Carter, 1981, pp.272-274)

They also find their use in the *privacy amplification* (see section 2.5.3).

2.1.4 Entropy and the Shannon Index

The *Shannon index* H measures the entropy in a discrete system. It computes logarithmically from the number of possible events in a set n and the distribution of the probability of occurrence of these events p_i (see equation 2.11). (Shannon, 1948, p.11).

$$H = -\sum_{i=1}^{n} p_i \ln p_i \tag{2.11}$$

The index fulfills the following criteria specified by Shannon (1948, p.10):

- Continuity within the p_i;
- Monotonic growth with n if the distribution is equal;
- Summability in the sense that it could form a weighted sum of sub-indices.

QKD uses the *conditional Shannon index* for error correction purposes (see section 2.5.2). This specialized index (shown in equation[4] 2.12) displays the distribution of the joint probability $p(x, y)$ of two events x and y. (Brassard & Salvail, 1994, p.411)

[4]The original equation was written in the form $H = -\sum_{i,j} P_i p_i(j) log p_i(j)$ (Shannon, 1948, p.13)

$$H(X|Y) = -\sum_{y\in Y}\sum_{x\in X} p(y)p(x|y)log_2 p(x|y) \tag{2.12}$$

Particularly, in a Binary Symmetric Channel (BSC)[5] the conditional entropy X given Y is the entropy of their XOR ($\oplus$) product. In other words the entropy of a set of *Bernoulli trials* with the probability operator p is multiplied by the number of events $n = |X| = |Y|$ as shown in equation (Brassard & Salvail, 1994, p.411)

$$H(X|Y) = H(X \oplus Y) = nh(p) \equiv |X| = |Y| \tag{2.13}$$

2.1.5 Low-Density Parity Checks

A Low-Density Partiy Check (LDPC) defines essentially a matrix. A sender, *Alice*, multiplies (modulo 2, since the operands are in binary) a sequence to send (*information digits*) with that matrix in order to get a number of parity bits (*check digits*). The triplet (n, j, k) defines the form of the matrix, where n is the block length of the information digits and defines the number of columns, j is a small number ≥ 3 and determines the number of ones in a column, whereas k is a small number $\geq j$ and determines the numbers of ones in a row. (Gallager, 1962, p.21) Those parameters determine also the number of rows, which can be no lower than $\frac{n}{k}j$. A matrix thus contains a low number of ones compared to zeros, hence the name *low-density*. Chosen beforehand and known to the receiver, *Bob*, the matrix defines which of the information digits are combined in a modulo 2 sum to determine the check digits. In a LDPC matrix, the element (x_a, x_b) contains a one if an information digit x_a contributes to a check digit x_b and a zero if it does not. The (however not realistic[6]) example below illustrates the operation mode of a LDPC, whereby bits x_1 through x_4 are information and x_5 through x_7 are check digits. (Gallager, 1962, p.21)

$$\begin{aligned} x_5 &= x_1 \oplus x_2 \oplus x_3 \\ x_6 &= x_1 \oplus x_2 \oplus x_4 \\ x_7 &= x_1 \oplus x_3 \oplus x_4 \end{aligned}$$

$$\begin{array}{c} \\ x_5 \\ x_6 \\ x_7 \end{array} \begin{array}{c} \begin{array}{cccc} x_1 & x_2 & x_3 & x_4 \end{array} \\ \begin{pmatrix} 1 & 1 & 1 & 0 \\ 1 & 1 & 0 & 1 \\ 1 & 0 & 1 & 1 \end{pmatrix} \end{array}$$

[5] a channel to transmit discrete bits, each attribute to noise with probability p (Brassard & Salvail, 1994, p.411)
[6] As j is < 3

$$\begin{pmatrix} 1 & 1 & 1 & 0 \\ 1 & 1 & 0 & 1 \\ 1 & 0 & 1 & 1 \end{pmatrix} \cdot \begin{pmatrix} 1 & 1 & 1 & 0 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 \end{pmatrix}$$

$$\begin{array}{cccc|ccc} x_1 & x_2 & x_3 & x_4 & x_5 & x_6 & x_7 \\ 1 & 1 & 1 & 0 & 1 & 0 & 0 \end{array}$$

LDPC matrices resemble bipartite graphs with information and check digits as node types (for a graph of the example matrix above see figure 2.2).

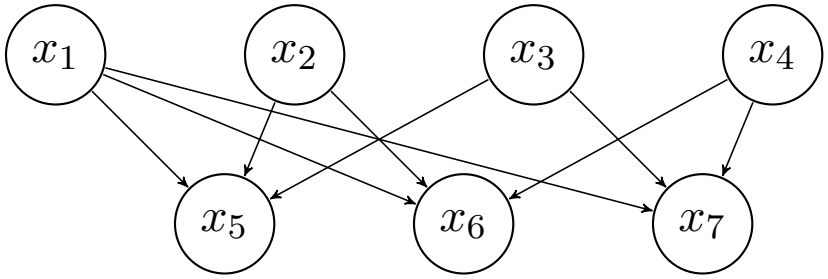


Figure 2.2: Graph of a Low-Density Parity Check Matrix

A matrix in the form (n, j, k) is filled with ones by dividing it into j groups. In the first group, each row i contains ones in at the positions from $(i-1)k+1$ to ik. The other groups contain random permutations of the first. (Gallager, 1962, p.22)

The receiver, *Bob*, tests whether his received sequence complies to that matrix. For instance, if the first bit in the above example contains an error, this circumstance will violate the equations of all of its according check digits, but only $k-1$ of the other equations in this block. *Bob* may now calculate the check digits on his own and compare it to the received ones. Eventually, the combination of information and check digits forms a system of linear equations. Contradictions in this system reveal erroneous bits. (Gallager, 1962, p.23) The efficiency of this algorithm can, however, be increased by enriching it with statistical data of single digit combinations occur with a probability $P = 1$, which decreases the ambiguity of each digit and decreases the average iteration number required to decode a block to a minimum of $\mathcal{O}(\log \log n)$. (Gallager, 1962, pp.23-24) QKD uses this scheme for error correction (see section 2.5.2.4) purposes.

2.2 Physical Fundamentals

This section describes the physical basics of QKD, which are fundamental concepts of quantum mechanics. It starts off with an explanation of the *Dirac notation*, which is the basis for this section, and explains then the concepts of quantum bits, pure states and quantum entanglement. It also describes briefly the principles of uncertainty, the no-cloning theorem and Bell's theorem. The terms described in this section are a prerequisite for the following sections.

2.2.1 The Dirac Notation

The *Dirac notation* is the standard notation for describing quantum states (Nielsen & Chuang, 2000, p.13). It is also named as *Bra-Ket*, as it derives from the brackets which describe a scalar product within the notation. The Dirac notation describes quantum states ψ as vectors in a Hilbert space $\mathcal{H}$ with a right angle ($\rangle$). Further it describes a certain quantum state a with a leading vertical bar: $\psi_a = |a\rangle$ (for an example $|a\rangle = \binom{1}{0}$). This vector is also called a *Ket*. Every *Ket* has a *dual space* equivalent $\bar{\psi}$, which is denoted by a left angle $\langle$. This so called *Bra* has, analogous to a *Ket*, a trailing vertical bar for a certain state: $\bar{\varphi} = \langle b|$. As a result, the eponymous notation for a scalar product denotes $\langle b|a\rangle$. The Dirac notation also makes use of tensor products, noted with the $\otimes$ operator. Linear operators (in the following sections denoted as α and β) use the usual arithmetical notation. (Dirac, 1939, pp.416-418)

Table 2.1 gives an overview over the *Bra-Ket* notations used in the rest of the chapter:

$\psi_a = \|a\rangle$	Quantum state ψ_a (*Ket*)
$\bar{\varphi}_b = \langle b\|$	Conjugate state $\bar{\varphi}_b$ (*Bra*)
$\alpha\psi = \alpha\|\psi\rangle$	Scalar multiplication of ψ with a linear operator α
$\psi_a\bar{\varphi}_b = \langle b\|a\rangle$	Inner product of ψ_a and $\bar{\varphi}_b$
$\psi_a \times \bar{\varphi}_b = \|a\rangle\langle b\|$	Outer product of ψ_a and $\bar{\varphi}_b$
$\|\psi\rangle \otimes \|\varphi\rangle$	Tensor product of ψ and φ

Table 2.1: Overview over used *Dirac* notation items

2.2.2 Quantum Bits

Quantum Bits (qubits) are the quantum mechanical representations of classical bits. The difference between a qubit and an ordinary bit is that it cannot only acquire the position of 0 and 1 but also positions in between, so called *superpositions*. (Nielsen & Chuang, 2000, p.13) Depending on the physical representation (called base) of a qubit (which is the kind of state that determines the information a particle contains) a quantum particle (for instance a photon) may acquire an arbitrary superposition. For an example, the base of a qubit could be a horizontally polarized photon for 0 and a vertically polarized for 1. As stated above, the photon could also acquire different states like $+45°$ and $-45°$, but also any other angle or other kinds of polarization like circular polarization. (Gisin, Ribordy, Tittel, & Zbinden, 2002, p.147) In fact, there is an infinite number of superpositions a qubit could take. (Nielsen & Chuang, 2000, p.15) The equation 2.14 describes the state of a qubits (ψ) formally, using the *Dirac notation*. (Nielsen & Chuang, 2000, p.13)

$$\alpha, \beta \in \mathbb{C} : |\psi\rangle = \alpha|0\rangle + \beta|1\rangle \tag{2.14}$$

Qubits are *unit vectors* where $|0\rangle$ and $|1\rangle$ are orthonormal to each other (for example $|0\rangle = \binom{1}{0}$ and $|1\rangle = \binom{0}{1}$) (Pivk, 2010a, pp.3-4). Horizontal and vertical polarization or diametral spins are examples for physical representations of a qubit (Wootters, 1998, p.1719). The coefficients

α and β are complex ($\mathbb{C}$) values, whose squared moduli are the probability of a qubit actually acquires $|0\rangle$ or $|1\rangle$ when measured. Naturally those squares must sum to 1, as the combined probability of acquiring any one of these two states when measured is 1 (see equation 2.15). (Nielsen & Chuang, 2000, p.13)

$$|\alpha|^2 + |\beta|^2 = 1 \tag{2.15}$$

QKD protocols use this characteristic and the principle that the measurement influences the outcome to exchange keys (see sections 2.2.5 and 2.4 for details).

2.2.3 Pure and Mixed States

A *pure state* means that the state is maximally specified. That means the probabilities of a distinguished state is 0 or 1 instead of a fraction, which would be called *mixed state*. (Wootters, 1998, p.1718) One may only distinguish orthogonal subsets of states perfectly and, thus, encode information onto the states to be retrieved with likewise perfect precision. (Bennett et al., 1994, p.7) An example of these states are diametral spin directions. Only diametral spin directions may be distinguished with certainty. Equation 2.16 shows the probability $P(\theta)$ of measuring direction $|y\rangle$ to get a result $|x\rangle$, where θ is the angle between the direction vectors. (Wootters, 1998, p.1718)

$$P(\theta) = \cos^2(\frac{1}{2}\theta) \tag{2.16}$$

This equation means that a state close to the measured has a high probability of being the actual state, while the only scenario where $P = 0$ is the complete opposite vector, thus the diametral direction. Figure 2.3 exemplifies the relation between actual and measured states: $|y_i\rangle$ are actual quantum states, while $|x\rangle$ is the measured state. The P-value next to an $|y_i\rangle$ shows the probability of being measured as state $|x\rangle$.

Despite the stated, there are QKD protocols which use nonorthogonal states. Some use special conditions of nonorthogonal states, like the one described in section 2.4.3. Others use measurement techniques which are yet able to distinguish between states, with the disadvantage of occasional inconclusive results as in section 2.4.7 (Scarani, Acín, Ribordy, & Gisin, 2004, p.2) Given the mentioned disadvantage, the determination is not perfect. (Wootters, 1998, p.1718)

A pure state of two qubits may now be denoted as in equation 2.17, where α, β, γ and δ are complex probability coefficients as described in section 2.2.2. (Nielsen & Chuang, 2000, p.16)

$$|\psi\rangle = \alpha|00\rangle + \beta|01\rangle + \gamma|10\rangle + \delta|11\rangle \tag{2.17}$$

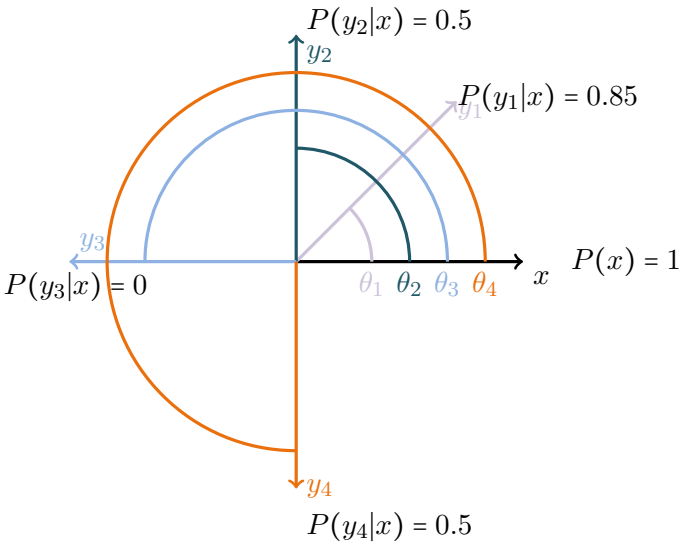


Figure 2.3: Probabilities of Measured and Actual Quantum Spin Directions

These two states may only be factorized into separate states via the *tensor product* (see section 2.2.1) if the coefficients for the pure and mixed states are even $\alpha\delta - \beta\gamma = 0$, so most states may not. These non-factorizable states are called *entangled states* (see section 2.2.6) and share a common pure state. (Wootters, 1998, pp.1718)

2.2.4 The No-Cloning Theorem

The *no-cloning theorem* was first published by Wootters and Zurek (1982) and says that a *quantum multiplier* (or quantum cloning device) is not consistent with quantum mechanics (Dieks, 1982, p.272). It presumes a unitary cloning function U and demonstrates the impossibility of such a function for arbitrary quantum states. The initial situation consists of two pure quantum states $|\psi\rangle$ and $|s\rangle$, where the first is to be copied onto the latter (see equation 2.18).

$$|\psi\rangle \otimes |s\rangle \xrightarrow{U} U(|\psi\rangle \otimes |s\rangle) = |\psi\rangle \otimes |\psi\rangle \tag{2.18}$$

Then, U is applied to another pure state φ (see equation 2.19).

$$\begin{aligned} U(|\psi\rangle \otimes |s\rangle) &= |\psi\rangle \otimes |\psi\rangle \\ U(|\varphi\rangle \otimes |s\rangle) &= |\varphi\rangle \otimes |\varphi\rangle \end{aligned} \tag{2.19}$$

The inner product applied on equation 2.19 gives equation 2.20 as a result.

$$\langle\psi|\varphi\rangle) = (\langle\psi|\varphi\rangle)^2 \tag{2.20}$$

The only two x where $x = x^2$ are $x = 0$ and $x = 1$. In this case, this means that $|\psi$ and $|\varphi$ are either identical or orthogonal. Thus, one may only clone orthogonal or identical states. (Nielsen & Chuang, 2000, p.532) This attribute of quantum information prevents an eavesdropper from *perfect eavesdropping* (that is without being noticed). (Gisin et al., 2002, p.150)

2.2.5 The Uncertainty Principle

The uncertainty principle says that the simultaneous measurement of two quantum mechanical observables cannot be exact. This relates in its original form to the simultaneous measurement of position and impulse of an electron. To observe the electron, it is necessary to illuminate it in some kind. As a consequence, the light particles interact with the electron and alter its impulse. Light at smaller wavelength makes the determination of an electron more accurate but has also a higher impact on the impulse. Thus, the more accurate the determination of the position, the less knowledge has an observer about the impulse. (Heisenberg, 1927, pp.174-175) The measurement of the impulse (or velocity), on the other hand, is based on the *Doppler Effect* and becomes more accurate with longer wavelengths. This of course conflicts with the accuracy of the measurement of the position. (Heisenberg, 1927, pp.177) Equation 2.21 shows the relation of the accuracy of the measurement of the impulse p_1 and the position q_1 is proportional to the *Planck constant* h, which means that a higher accuracy of the one results in a lower accuracy of the other. (Heisenberg, 1927, pp.174)

$$p_1 q_1 \sim \hbar \tag{2.21}$$

This was later generalized by Schrödinger (1930) and Robertson (1929) to that for no two commuting observables in any quantum state the outcome of a measurement is certain (Fehr, 2010, p.526). Nielsen and Chuang (2000, p.89) give a *Dirac notated* form of the uncertainty principle (see equation 2.22), where C and D are observables and $[C, D]$ is the commutator applied to them. The commutator calculates from $[A, B] \equiv AB - BA$ and shows the commutativity C and D. As, in quantum mechanics, properties of a quantum state are vectors, the commutator is zero if and only if C and D are diagonal with the same orthonormal basis. (Nielsen & Chuang, 2000, pp.76-77) This means that the ability to observe C and D depends on the quantum state of ψ.

$$\Delta(C)\Delta(D) \geq \frac{|\langle\psi|[C, D]|\psi\rangle|}{2} \tag{2.22}$$

This does, however, not mean that measurement of observable C disturbs the value of D, only that C and D may only be measured to a certain degree of accuracy. (Nielsen & Chuang, 2000, p.89) Section 2.4 discusses protocols, which use this factor. For instance, polarizations

of single photons only reveal a certain amount of information when measured and behave ambiguously beyond that. (Bennett & Brassard, 1984, p.175) The uncertainty principle (see section 2.2.5) may be reformulated to statements about the entropy of observables using the *Shannon index* (see section 2.1.4). This results in equation 2.23. (Nielsen & Chuang, 2000, p.503)

$$H(C)H(D) \geq 2log\frac{1}{max_{c,d}|\langle c|d\rangle|} \tag{2.23}$$

This may be transferred to the available information of a quantum state, where it sets a boundary to what a receiver *Bob* and an eavesdropper *Eve* may jointly know about a QKD derived key of a sender *Alice*. It basically says that if *Eve* listens to the transmission from *Alice* to *Bob*, the perturbation that comes with the measurement has necessarily a limiting effect on *Bob*'s information. (Gisin et al., 2002, p.186) In the single photon example stated above, *Eve* would involuntarily alter the polarization in a random way in order to gain information, thus reducing *Bob*'s (Bennett & Brassard, 1984, p.175). In that way, the uncertainty principle helps to ensure the security of QKD protocols.

2.2.6 Quantum Entanglement

Entangled quantums are connected particles, so that their state is correlated when measured. (Nielsen & Chuang, 2000, p.17) As stated in section 2.2.3, those two particles share a common *pure state* instead of having a pure state on each own. (Wootters, 1998, p.1718) An important example of an entangled two qubit state is the *Bell state* as described in equation 2.24. In this state, the first measurement defines the second one. Each has the probability of $\frac{1}{2}$ of measuring 1 or 0, leaving the pair in a state ϕ' of $|00\rangle$ or $|11\rangle$. This means that the second qubit always measures the same result as the first one. (Nielsen & Chuang, 2000, pp.16-17)

$$|\psi\rangle = \frac{|00\rangle + |11\rangle}{\sqrt{2}} \tag{2.24}$$

Another important state (with similar properties) is the *singlet state*. In this entangled state, the combined spin of a pair of particles with spin $\frac{1}{2}$ is 0 as shows equation 2.25. In this case, the measurement of the second bit always shows the opposite value than that of the first. (Biham, Huttner, & Mor, 1996, p.4)

$$|\psi\rangle = \frac{|\uparrow_A\downarrow_B\rangle - |\downarrow_A\uparrow_B\rangle}{\sqrt{2}} = \frac{|\leftarrow_A\rightarrow_B\rangle - |\rightarrow_A\leftarrow_B\rangle}{\sqrt{2}} \tag{2.25}$$

Entangled particles are quite common, but mostly in close spatial proximity to each other. Spatially separated entangled particles are rare and difficult to generate, but there are still

successful examples. (Wootters, 1998, pp.1718-1719) Some QKD protocols rely on entangled states, as described in section 2.4.

2.2.7 The Einstein-Podolsky-Rosen Paradox and Bell's Theorem

Albert Einstein, Boris Podolsky and Nathan Rosen rejected the idea of entangled quantum states. They argued that a complete theory must describe each quantifiable element of reality individually[7] and make reliable predictions without perturbations of the measurement influencing the outcome. (Einstein et al., 1935, p.777) This is not true for entangled states, where the outcome of one state measurement determines the outcome of the other; therefore this is sometimes called the Einstein-Podolsky-Rosen (EPR) paradox. They concluded that quantum mechanics is incomplete (Einstein et al., 1935, p.780). In their opinion, quantum mechanics should be supplemented by *hidden variables* (Bell, 1964, p.195). *Bell's theorem* shows that such a supplement does not complete quantum mechanics in the sense the EPR paradox requires, in particular the theorem shows that complementary parameters λ cannot describe the original *singlet state* of two $\frac{1}{2}$ spin particles. (Bell, 1964, pp.195-199) The theorem concludes that such an amendment to determine individual properties without changing statistic properties requires means of one measurement influencing the other regardless of the locality, thus rendering quantum mechanics as a non-local theory. (Bell, 1964, p.199) The theorem contains the *Bell inequality* which determines that in a *classical* complete theory, as EPR suggests, the average correlation of two state vectors $\bar{P}(\vec{a}, \vec{b})$ plus the values of the two vectors plus their scalar product must be smaller than a certain limit ϵ (see equation 2.26). The values in the Bell inequality cannot be arbitrary small due to their co-dependency. In simple words, the Bell inequality shows that the sum of average single components cannot be more than the sum of their maximum values. Certain quantum states exhibit combined average values bigger than the shown minimum in the Bell inequality, and therefore do not satisfy it. (Bell, 1964, p.198)

$$|\bar{P}(\vec{a}, \vec{b}) + \vec{a}\vec{b}| \leq \epsilon \tag{2.26}$$

Later authors generalized the Bell inequality and analyzed existing experiments (and proposed further) to investigate wether the Bell inequality is violated, which they predicted (Clauser, Horne, Shimony, & Holt, 1969). The quintessence of these studies is that quantum mechanics either violates the concept of *realism* (existence of properties from observation) or *locality* (mutual influence of distant events). (Nielsen & Chuang, 2000, p.117) The satisfaction of the Bell inequality is a necessary (though not sufficient) condition for the separability of a quantum state (Keyl, 2002, p.448). This means in reverse that violating the Bell inequality is a sufficient (though not necessary) criterion to determine the presence of an entangled state. QKD protocols (like the one described in section 2.4.2) use this behavior to determine the presence of an entangled quantum state.

[7] "every element of the physical reality must have a counterpart in the physical theory." (Einstein, Podolsky, & Rosen, 1935, p.777)

2.3 Physical Implementations of Quantum Channels

There are basically four kinds of QKD implementations (Hughes, 2004, p.6):

- Weak laser pulse;
- Single-photon sources;
- Entangled pairs;
- Continuous Variables.

Weak laser pulse systems use highly attenuated diode lasers over fiber optics. They try to send only ≈ 1 photon per pulse and are relatively low-cost. Their field application areas are metropolitan areas. (Bethune & Elliott, 2004, pp.1-2) Single photon sources try to utilize single molecules, Nitrogen Vacancy-Centers (NVCs) in diamond, semiconductor quantum-wells and dots spontaneous parametric down-conversion to generate single photons (Nam, 2004, p.1). These sources, however, may be unpractical in QKD (Nam, 2004, p.4). Entangled-pair QKD is discussed in section 2.4.2 and continuous-variable QKD in section 2.4.6.

2.3.1 Disturbances on Quantum Channels

In single-photon systems (weak laser and single-photon sources), the probability that a photon out of a laser pulse actually reaches the receiver is only a fraction of the sending rate. Therefore the disturbance limits the key material per second. Equation 2.27 shows the computation of the *quantum channel gain* (g_q), which describes the combined losses of the quantum channel. When multiplied by the number of photons emitted by the laser emitter of the sender (f_{data}), it determines the average rate of received key material per second ($g_q \cdot f_{data}$). (Pivk, 2010b, p.26)

$$g_q = \mu \cdot \alpha_f \cdot \alpha_e \cdot \eta_{det} \cdot k_{dead} \tag{2.27}$$

In this equation μ describes the *mean photon number* emitted by a single photon source. This source usually consists of a laser (sending at f_{data}) and a device to weaken the laser pulse to point where it only emits a single photon. The mean photon number calculates by the average number of times the weakening device emits photons after its application. Usual μ rates are 0.1, which means that only ten percent of the pulses actually emit a photon after the weakening. From this ten percent, still five percent yield in more than one photon. (Pivk, 2010b, p.24) The values α_f and α_e describe losses in the fiber and other system losses, while η_{det} describes the efficiency rate of the receiver's photon detector. Finally, the value k_{dead} is a factor of the unavoidable hold-off time of the receiver's photon detector between a successfully made detection and the possibility of a subsequent one. (Pivk, 2010b, p.26) For weak laser pulse systems the raw detection rate is approximately $\eta_{det}\eta_l\mu$, where η_l is a factor for the losses due to the line length. (Scarani et al., 2004, p.1)

2.4 Quantum Key Distribution Protocols

QKD protocols use between two and six nonorthogonal states (more is thinkable but not practicable) to encode a single qubit in order to distribute a common key between two parties, *Alice* and *Bob*. There are two groups of protocols: one where one party prepares particles in a random state and sends them to the other (relying on the uncertainty principle (see section 2.2.5 for eavesdropper detection) and a second where the measurement of an entangled particle state of one party determines the outcome of the other (where wiretap detection relies on the Bell inequality see section 2.2.7). The measurement itself is similar. (Bennett, 1992, p.3121) As the measurement determines the outcome in both types of protocols, one could also speak of key generation instead of mere distribution in QKD. (Nielsen & Chuang, 2000, p.591)

There is a number of different protocols which ensure secure key distribution over the quantum channel. The following sections give an overview over these protocols.

2.4.1 Bennet and Brassard 1984

Bennett/Brassard 1984 (BB84) is the oldest (Bennett, Brassard, Ekert, Fuchs, & Preskill, 2004, p.1) and best known QKD protocol (Pivk, 2010b, p.23). It uses two different bases which each represent two qubit states ($|0\rangle$ and $|1\rangle$). Although other methods are possible, an established method to represent the two bases of qubit is the originally proposed one of using photon polarizations. The two bases consist each of two orthogonal polarizations: horizontally/vertically (+), which consists of the polarizations $\leftrightarrow$,$\updownarrow$ and diagonally($\times$), which consists of the polarizations $\nearrow$, $\nwarrow$, representing binary states. The polarizations in each of the base have to be orthogonal as this is the only case where the two polarizations where one can certainly distinguish between the two states. Naturally the two sets of polarizations are nonorthogonal to each other. In other words: a qubit does not reveal any information if measured in a nonorthogonal wrong polarisation. (Bennett & Brassard, 1984, pp.175-176) The original specification proposes a polarization base pair of $0°$ for 0 and $90°$ for 1 and a second base pair consisting of $45°$ for 0 and $135°$ for 1. (Bennett & Brassard, 1984, p.175)

To distribute a secure key, a call initiator *Alice*, who has the ability to generate the four described states, sends a stream of polarized photons to and a call receiver *Bob*, who has the ability to measure each base pair with a distinguished filter. *Alice* randomly choses a random bit(0 and 1) and a random base ($+, \times$) resulting in a random photon polarization ($\leftrightarrow$,$\updownarrow$,$\nearrow$, $\nwarrow$). *Bob* chooses on the other end of the line randomly and independently from *Alice* one of the two filters ($+, \times$). If the bases match, the photon is measured correctly by *Bob*, thus resulting in a successfully transferred bit. If the bases do not match, however, there is a probability of $\frac{1}{2}$ for each of the polarizations of *Bob*'s base to match them, as the two bases are conjugate to each other. (Bennett & Brassard, 1984, pp. 176-177) Non-matching bases therefore mean that *Bob* does not get any information from *Alice* as a probability of $\frac{1}{2}$ in two states (0 or 1) does not contain any.

The next step is that *Alice* and *Bob publicly* (that is over the *public channel*) compare their bases on order to determine which of the sent photons have been measured wrong (which is statis-

tically half of the photons). Section 2.5.1 discusses this so called *sifting* in more detail. Both of the partners throw the wrongly measured bits away gaining a correct key.

To exemplify this: imagine *Alice* randomly[8] chooses the following sequence of ten photons: ↘ ↙ ↘ ↕ ↙ ↔ ↕ ↔ ↕ ↔, while *Bob* choses, randomly and independently from *Alice*, the following ten bases: × × × + + + × × +×. The qubits 1 through 4, 6 and 9 show matching bases, the others are discarded. According to the values mentioned earlier, this results in a key sequence of 101101. Figure 2.4 shows the exchange of data via the quantum and public channels. Above it shows the original random bit sequence chosen by *Alice* (wich is subsequently encoded into quantum polarizations) and below the resulting sifted common key.

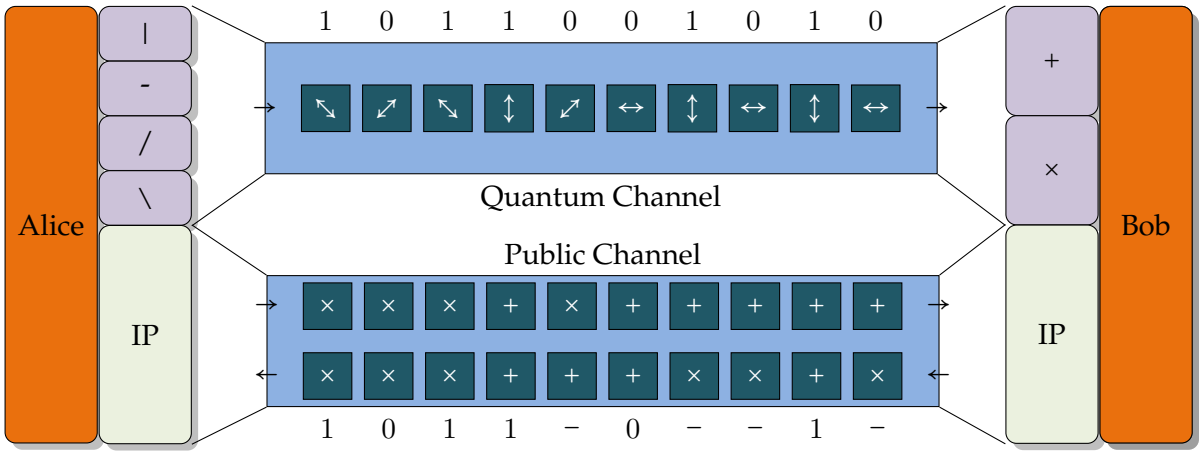


Figure 2.4: The *Bennett/Brassard 1984* (BB84) Protocol

The exchange of bases on the public channel is secure, if authentication in the manner section 2.1.3.1 suggests is in place. (Bennett & Brassard, 1984, p.177) Section 2.5 discusses further details of sifting and eavesdropper detection.

2.4.2 Ekert 1991

The *Ekert 1991* (E91) protocol uses *entangled states* (see section 2.2.6) for security. A trustful source generates pairs of $\frac{1}{2}$-spin particles in a singlet (which means an entangled, see section 2.2.7) state and sends them to *Alice* and *Bob*. Of each particle, the two of them measure an observable, chosen randomly and independently from each other from a set of complementary attributes. The original proposal suggests three components of the spin of a particle. *Alice* measures with angles $\phi_a i = \{0, \frac{1}{4}\pi, \frac{1}{2}\pi\}$, while *Bob* measures with $\phi_b j = \{\frac{1}{4}\pi, \frac{1}{2}\pi, \frac{3}{4}\pi\}$. Each of the random measurements can result in a spin state of +1 (*spin up*) or −1 (*spin down*), whereby they show an correlation coefficient E of a certain observable to have a certain value calculated by equation 2.28. (Ekert, 1991, p.662)

$$E(a_i, b_j) = P_{++}(a_i, b_j) + P_{--}(a_i, b_j) - P_{+-}(a_i, b_j) - P_{-+}(a_i, b_j) \tag{2.28}$$

[8]The random numbers for this example have been taken from `http://www.random.org`

After that, they exchange the bases of their measurement over the public channel and divide the results in two groups: one with according bases and one with disaccording bases. Figure 2.5 depicts this operating principle. The ones with according bases determine the key as they are *anticorrelated* (which means they have the exact opposite value) due to their entangled state ($E(a_2, b_3) = E(a_3, b_2) = -1$). (Ekert, 1991, p.662)

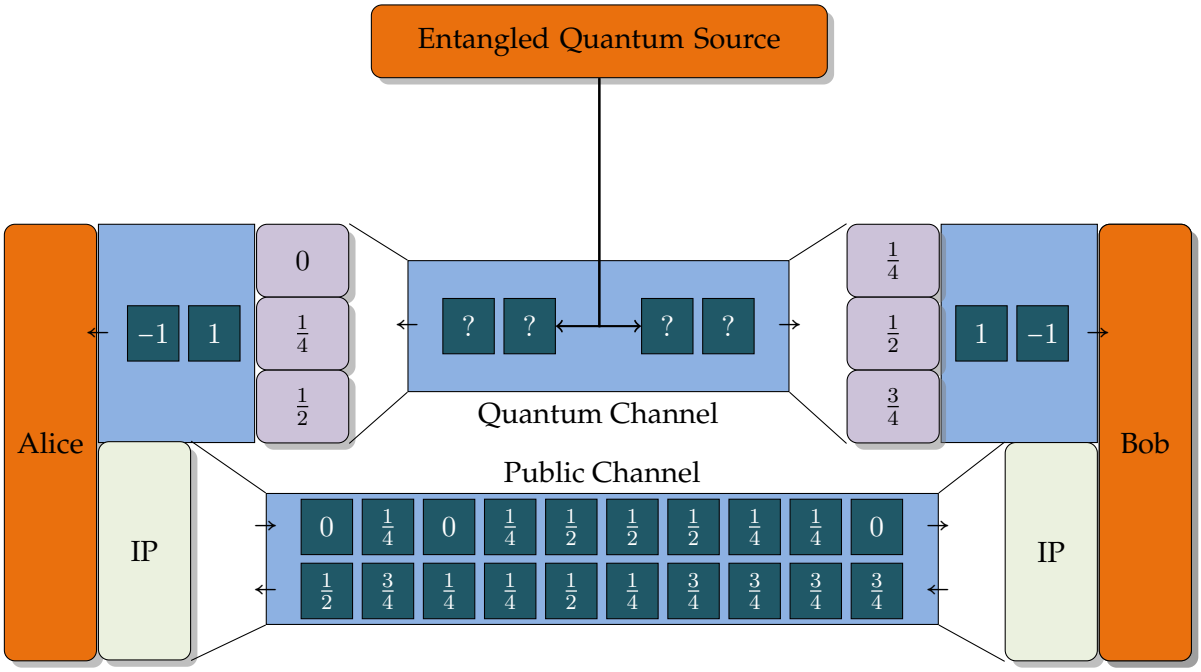


Figure 2.5: The *Ekert 1991* (E91) Protocol

The disaccording particles can still be used to test if an eavesdropper *Eve* listens to the qubits on the quantum channel. As stated in section 2.2.7, the *Bell inequality* is able to test the presence of entangled states. The protocol uses a variant by Clauser et al. (1969) to test the presence of an entangled state. The average probability E that a certain observable has a certain state value calculates by $E(a_i, b_j) = -a_i b_j$. Quantum mechanics predicts that entangled states fulfill equation 2.29 (which violates the Bell inequality as a classical theory would set -2 as a limit). (Ekert, 1991, p.662)

$$S = E(a_1, b_1) - E(a_1, b_3) + E(a_3, b_1) + E(a_3, b_3) = -2\sqrt{2} \tag{2.29}$$

Alice and *Bob* can now exchange the disaccording particles openly (over the public channel), as they do not participate in the composition of the key. These values determine whether or not equation 2.29 is fulfilled. If so, an entangled state is present and the key derived from the group of corresponding results is legitimate. If not, *Eve* may have altered some or all of the particles and the key is not secure. (Ekert, 1991, p.662)

An advancement and simplification to the the protocol works with the scheme from the BB84 protocol (see section 2.4.1), also known as *Bennett/Brassard/Mermin 1992* (BBM92) (Bennett, Brassard, & Mermin, 1992, p.3). Other types for entangled measuring include the energy-time

entanglement (Tittel, Brendel, Zbinden, & Gisin, 2000) or polarization-entangled photons. In the latter case the photons have a polarization of $\alpha_{1-4} = \{45°, 90°, 135°, 180°\}$ for *Alice* and $\beta_{1-4} = \{0,°, 45°, 90°, 135°\}$ for *Bob*. (Naik, Peterson, White, Berglund, & Kwiat, 2000, p.1) Some propose to not use a third party source, but instead to let *Alice* herself generate the entangled particles (Kwiat & Rarity, 2004, p.2).

2.4.3 Bennett 1992

The *Bennett 1992* (B92) protocol is an advancement of the BB84 protocol, only that it uses only two states instead of four states. The two states ($|u_0\rangle$ and $|u_1\rangle$), however, have to be nonorthogonal and possess two projection operators P_0 and P_1 which depict $|u_1\rangle$ and $|u_0\rangle$ onto an orthogonal plane (see equation 2.30). (Bennett, 1992, p.3122) In contrast, BB84 uses two sets of states orthogonal to each other, where the sets

$$P_i = 1 - |u_i\rangle\langle u_i| \,;\; i = \{0, 1\} \tag{2.30}$$

The operator P_0 annihilates $|u_1\rangle$ and successfully reads $|u_0\rangle$ with a probability of $1-|\langle u_0|u_1\rangle|^2 > 0$. Analogous, the operator P_1 annihilates $|u_0\rangle$ and successfully reads $|u_1\rangle$ with the same probability. The sender *Alice* now sends randomly chosen qubits encoded in $|u_0\rangle$ and $|u_1\rangle$, while the receiver *Bob* measures them with the randomly chosen operators P_0 and P_1. Subsequently, *Bob* tells *Alice* publicly which measurements have been successful. These qubits are correlated and can be used as a common key, while *Alice* discards the others. Afterwards, they publicly exchange some of the key bits for error correction and eavesdropper detection. (Bennett, 1992, p.3122) An implementation of this protocol uses phase shifting to encode $|0\rangle$ and $|1\rangle$. (Zbinden et al., 1998, p.745)

2.4.4 Time-Reversed Einstein-Podolsky-Rosen Scheme

In this protocol, as in BB84, two communication partners *Alice* and *Bob* use four states to encode information. *Alice* encodes her key value into the state $|\alpha\rangle_a$ and Bob encodes his value into $|\beta\rangle_b$, where a and b are the bases + and ×, respectively. They subsequently send the encoded qubits to a third party, the *center*.

This *center*, which is an untrusted party, measures the total-spin operator ($(\hat{S}_total)^2$) by projection. The *center* determines then whether or not the projection results in a singlet state (see equation 2.25 in section 2.2.6), which it does if the total spin equals 0. If so, the two particles sent by *Alice* and *Bob* are *anticorrelated* (similar to the E91 protocol). After the notification of the *center* whether the particles are in a singlet sate, *Alice* and *Bob* have to exchange their bases a and b (similar to the BB84 protocol) over the public channel. If the bases are equal ($a = b$), the encoded information is the exact opposite and *Bob* may obtain *Alice*'s key by reversing his key bits, resulting in a common key. The probability of obtaining a singlet state is 0 if the state is equivalent and $\frac{1}{2}$ if it is opposite. Both states have equal probability ($P = \frac{1}{2}$). This is further reduced by the probability of *Alice* and *Bob* sharing the same base which is also $\frac{1}{2}$. The total

probability of aquiring a usable key bit out of a sent and received state therefore is $\frac{1}{2}\frac{1}{2}\frac{1}{2} = \frac{1}{8}$. (Biham et al., 1996, p.5) Figure 2.6 depicts this scheme.

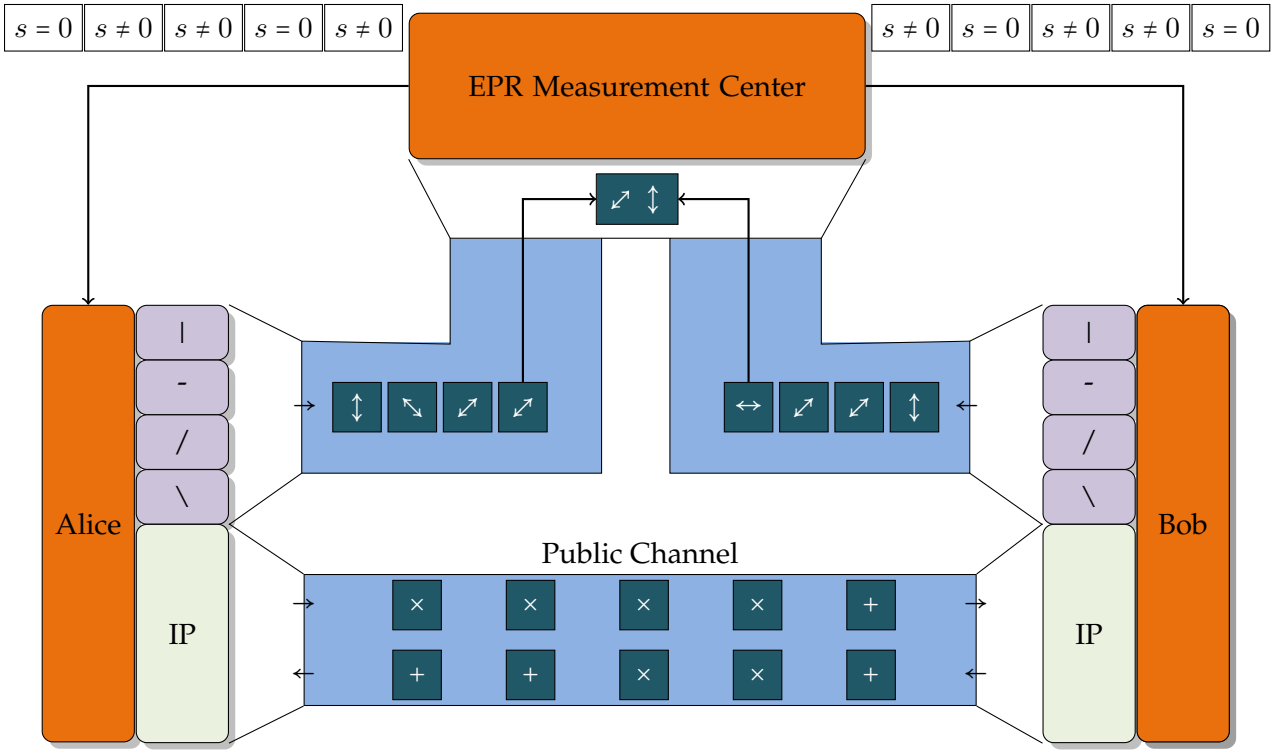


Figure 2.6: The Time-Reversed Einstein-Podolsky-Rosen Scheme

The *center*, however, does not know which of the parties had which value as, it is only able to measure the combined state of a pair of $|\alpha\rangle_a$ and $|\beta\rangle_b$ undetected, otherwise it would necessarily introduce errors (Biham et al., 1996, pp.5,7). Inamori (2002, p.344) shows a test which a malicious *center* cannot pass and still extract information out of the qubits sent by *Alice* and *Bob*.

2.4.5 Six-State Protocol

The Six-State Protocol (SSP) rests upon the BB84 protocol. In contrast to the latter, SSP operates with three base pairs (resulting in six states likewise to the original specification of the E91 protocol) instead of two. The benefit of this method over the four states of BB84 is that an eavesdropper *Eve* may obtain less information out of an information flow in a quantum channel. In both protocols the eavesdropping causes disturbance on the quantum line allowing the communication partners *Alice* and *Bob* to detect *Eve*. (Bruß, 1998, p.3018) Compared to BB84, the mutual information shared by *Alice* and *Eve* is lower. This means that it is more difficult for *Eve* to obtain information about the key without being detected. In conclusion SSP is more secure than BB84. (Bruß, 1998, p.3020) The disadvantage of this protocol is that, using three states instead of two, the probability of *Alice* and *Bob* choosing the same base statistically sinks from $\frac{1}{2}$ in BB84 to $\frac{1}{3}$ in SSP. This reduces the ratio of usable key material from the sent qubits. (Bruß, 1998, p.3018) Nonetheless, a successful entanglement-based implementation showed

that, for low error rates, the amount of key material may be even higher after all stages of key generation are complete. (Enzer, Hadley, Hughes, Peterson, & Kwiat, 2002, p.45.7)

2.4.6 Continuous Quantum Variables

In contrast to other QKD protocols, *continuous quantum variable* protocols use multi-photon effects and are, thus, independent from single-photon sources. In a certain protocol the amplitude and phase quadratures carry the key information. These observables of a particle beam underlie, similar to single particles, the uncertainty principle, as they are conjugate variables. Although these variables can be measured simultaneously by splitting a beam, this process will inevitable introduce noise onto the beam. The uncertainty principle thus limits the information which can be taken from a beam. A sender, *Alice*, now modulates two independent streams of random numbers on the beam, one onto the phase quadrature and one onto the amplitude quadrature. The receiver of the transmission, *Bob*, measures the beam with a randomly chosen quadrature. Beforehand they agree to one of the quadratures as the key channel and the other as the test channel. The quadrature measured changes in random intervals. Similar to BB84, *Bob* notifies *Alice* of quadrature changes over the public channel. *Alice* takes the part of the information from the key stream which is respective to the time when *Bob* measured the key quadrature. *Bob* then public sends the measured values from the test quadrature to *Alice* who compares it to the respective time slot on the test stream. If the error rate exceeds a certain acceptable value or rises erratically, they may assume that noise introduced by an eavesdropper is responsible and consequently regard the key as insecure. (Ralph, 1999, pp.1-2)

2.4.7 Scarani, Acín, Ribordy and Gisin 2000

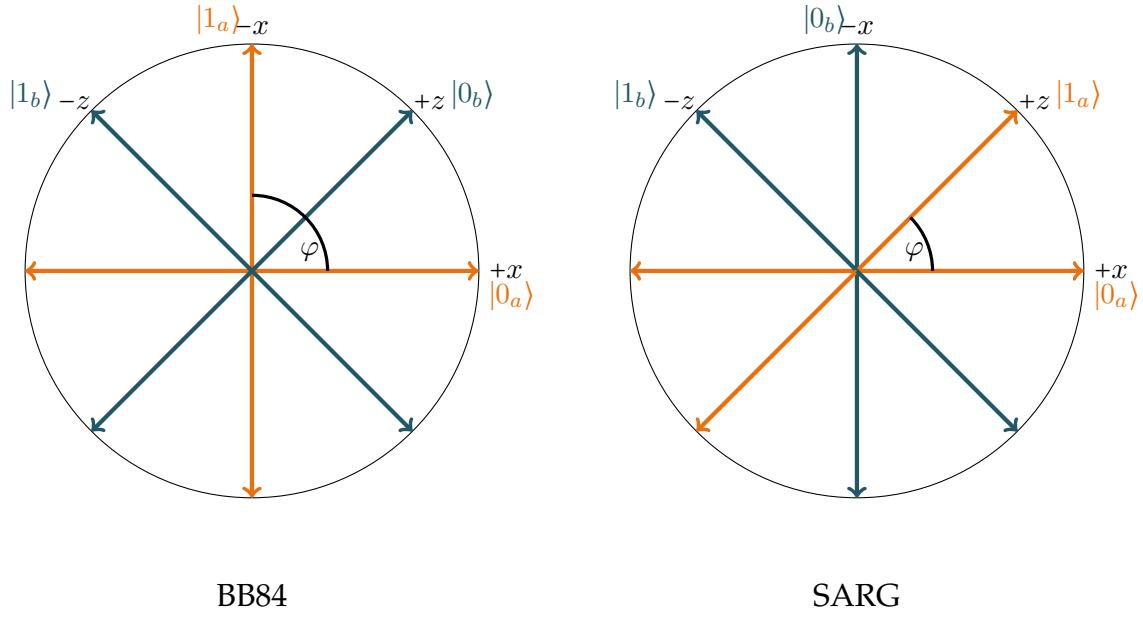
The *Scarani/Acín/Ribordy/Gisin* (SARG) protocol addresses some issues of the BB84 protocol when implemented with weak laser pulses (which is widely proliferated). The problem is that using a weak laser pulse, a sender, *Alice*, does not necessarily send single photons to a receiver, *Bob*, but a set of photons with equal polarisation. An eavesdropper, *Eve*, does theoretically not have to clone qubits (and is, thus, not bound to the no-cloning theorem depicted in section 2.2.4), as she may split off a part of the set and measure it sending the rest of the set to *Bob* in order to remain undetected. (Scarani et al., 2004, p.1) To decrease the information of *Eve*, SARG uses two sets of each two nonorthogonal states. This means that $\langle 0|1\rangle \neq 0$ as the $\cos(\varphi) \neq 0$ for an angle φ between the the two states $|0\rangle$ and $|1\rangle$. Despite the states not being orthogonal, there is a type of filter with the ability to certainly distinguish between the states, but has the disadvantage that only a fraction of the photons pass the filter, even if it is correctly chosen. (Scarani et al., 2004, p.2)

The SARG protocol tries to take advantage of nonorthogonal states only by modifying the sifting procedure. To establish a key, *Alice* sends qubits randomly chosen from two pairs consisting each of two states denoted as $|\pm x\rangle$ and $|\pm z\rangle$ (exactly as in BB84). The difference to BB84 lies in two factors. First, $|\pm x\rangle$ represents a 0 and $|\pm z\rangle$ represents a 1, so a $(0, 1)$ pair lies adjecent instead of orthogonal. Second the announcement of the bases is different: *Alice*

announces publicly one of the four possible state pairs $\{|\pm x\rangle, |\pm z\rangle\}$. In conclusion the overlap χ in between a pair of states with angle φ is $\frac{1}{\sqrt{2}}$ as shown in equation 2.31.

$$\chi = |\langle 0_a|1_a\rangle| = \frac{1}{\sqrt{2}} \tag{2.31}$$

Alice uses the correct sign for the group of the actually sent qubit and an arbitrary for the other, *Bob* announces in return whether or not his measurement was unambiguous. (Scarani et al., 2004, p.3) Figure 2.7 depicts the differences between the bases of BB84 and SARG. Note that the pairings in SARG according to the graph could also be between $|0_a\rangle$ and $|1_b\rangle$, depending on which pair *Alice* announces, resulting in the same φ (see the example following below).



BB84 SARG

Figure 2.7: Differences between the Bases in the BB84 and SARG Protocols

As an example, if the sent qubit was $|x\rangle$, *Alice* may announce $\{|+x\rangle, |+z\rangle\}$ or $\{|+x\rangle, |-z\rangle\}$. In the first case, if *Bob* measured the x base, he will correctly measure $|+x\rangle$. If he measured the y base he will either get $|+z\rangle$ or $|-z\rangle$ with the same probability. Only if he measured with the wrong base and got $|-z\rangle$ he will get an unambiguous result. As this is not in the announced set, *Bob* could conclude with certainty, that he had measured the wrong (y) base, so for the correct (x) base he may add a 0 to the key bits. The second case is analogous to the first. Again a measurement of base x brings an ambiguous result. The measurement of the z bases only results conclusively the result was positive ($|+z\rangle$) as it contradicts the announced negative result for z. Afterwards, *Bob* announces the conclusiveness of his results in order for *Alice* to incorporate or discard the respective key bit. Naturally, the procedure is analogous for the other three qubit states. In conclusion, SARG does rather perform a negative test onto the bases in the sifting phase instead of distinguishing between two states. This approach makes the protocol significantly more resilient against a variety of attacks against QKD implementations using weak laser pulses including photon-number splitting and cloning attacks. (Scarani et al., 2004, p.4)

2.5 Techniques Used to Increase Efficiency and Security

The preceding sections describe QKD protocols to provide a secure distribution of a shared key. However, they omitted a crucial attribute of real-world quantum channels, namely errors which inadvertently occur. This section describes the schemes to correct these errors. It also takes a closer look at how diverse QKD protocols conduct the sifting process. In general, this section elaborates the mechanisms carried out over the public channel.

2.5.1 Sifting

The term *sifting* describes the process of synchronizing the key streams of a sender *Alice* and a receiver *Bob* by ensuring that what *Alice* has sent over the quantum channel is the same as *Bob* has received. As described in section 2.3.1 physical implementations of quantum channels feature disturbances resulting in losses of qubits. The sifting phase of QKD is responsible for ensuring that *Alice* and *Bob* have the same knowledge which qubits are lost. To do so, *Bob* simply has to tell *Alice* which of the qubits he has received. (Pivk, 2010b, p.27) The knowledge which qubits have gone lost may come from a synchronized clock and the expectation of receptions in a given time slot. (Lütkenhaus, 1999, p.3)

Furthermore, all of the protocols described in section 2.4 use several different bases to encode qubits. Sifting makes sure both communication partners use the same bases (qubit-wise) by exchanging them over the public channel. Therefore the two partners send each other a bit string of the uses bases (for example 0 for base $+$ and 1 for base $\times$). Eventually, both partners get a common *sifted key*, without the qubits where they used non-matching bases. This key, however, still contains errors (see sections 2.5.2.1 and 2.5.2). (Pivk, 2010b, p.28) Naturally the average rate of lost key bits due to measurement base mismatch in a four-state protocol (according to bases) is $\frac{1}{2}$ if the choice of bases happens randomly. If *Alice* and *Bob* agree to a bias for one of the bases they will be able to level down this rate (hence this is called *biased sifting*. This means that they agree on a number $0 < p < \frac{1}{2}$ and uses the two bases with the probabilities of p for one and $1-p$ for the other. (Lo, Chau, & Ardehali, 2005, p.151) Of course, an eavesdropper has also a higher probability of yielding a better result on guessing key bits. As the purposes of error estimation also include eavesdropper detection, the algorithm for that must be adapted (see section 2.5.2). In summary, the *sifted key* consists of the send qubits minus quantum and base mismatch losses.

As the sifting runs unsecured over the public channel, it needs authentication. As discussed in section 2.1.3.1, authentication by ε-ASU_2 hash classes comes close to unconditional security and is therefore used to authenticate the sifting. Unfortunately, this form of authentication uses a common key for *Alice* and *Bob*. Therefore some of the key bits generated by QKD must be spared to authenticate the next round of sifting. Initially (before any QKD key is available), *Alice* and *Bob* must distribute a key beforehand through a different secure channel. (Pivk, 2010b, p.30)

2.5.2 Reconciliation

Reconciliation is the process of estimating the rate of and correcting errors on a quantum channel to boost the rate of key bits, which is limited by errors (see 2.3.1). The first step is to estimate the number of errors, the Quantum Bit Error Rate (QBER). The error rate tells a sender/receiver pair *Alice* and *Bob* how many of the key bits do not match despite the sifting process (which means that they are still wrong even if the bases matched), as there are also physical error causes, besides the chosen bases. This means that on *Bob*'s side, a sifted key bit might exhibit the opposite value compared the corresponding bit on the *Alice*'s side with the probability p and preserves its value with $1 - p$. (Nielsen & Chuang, 2000, p.355) Figure 2.8 depicts a *Markov chain* of this set of facts.

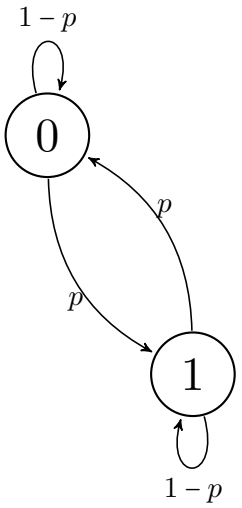


Figure 2.8: Markov Chain of State Probabilities

2.5.2.1 Eavesdropper Detection

If an eavesdropper *Eve* listens to both the quantum channel (where the qubits are transmitted) and the public channel (where the bases are compared) she would potentially gain the same knowledge as *Alice* and *Bob*, thus compromising the key. *Eve*, however, generates errors on the quantum channel as she is not able to measure the photon without influencing its state (the combined knowledge of *Eve* and *Bob* of *Alice*'s emission is limited - see section 2.2.5). If she chooses the wrong base while measuring , she polarizes the photon with the wrong base before sending it forth to *Bob*. Because of the *no-cloning theorem* she is also not able to duplicate it before the measurement. *Alice* and *Bob* can now detect *Eve* by exchanging random bits from the generated key and compare if they do not match despite having chosen a matching base. (Bennett & Brassard, 1984, p.177) The probability for Eve of introducing an error in a system with two bases is $\frac{1}{8}$. There are $(n = 2)^3 = 8$ possible combinations of *Alice*, *Bob* and *Eve* chosing their bases and *Eve* introduces an error only in the $m = \frac{2}{8}$ states where *Alice* and *Bob* agree to each other but she chooses a different base. Equation 2.32 shows *Eve*'s error introduction rate, where n is the number of bases (as section 2.4.5 shows, there may be more

than two). The explanation is as follows: the denominator is the total number of combinations between the three parties and the numerator is the total number reduced by one power (as the combinations of *Alice* and *Bob* collapse to a single factor, due to their correlation) minus the n times where *Eve* also chooses the same base. This is further reduced by $\frac{1}{2}$ as the photon gets its original polarization again at *Bob*'s detector half of the time.

$$P_e(n) = \frac{n^2 - n}{2n^3} \tag{2.32}$$

The error rate p of the forward approach to exchange random bits computes by the number of errors e divided by the length of chosen bits r (see equation 2.33). (Pivk, 2010b, p.31) The bits used for error correction are naturally disclosed and have, thus, to be discarded.

$$p = \frac{e}{r} \tag{2.33}$$

Eve, however, may use a *biased attack* analogous to the biased sifting in section 2.5.1. This means that *Eve* measures one base with a higher probability p_1 than the other p_2 (and does nothing with the probability $p = 1 - p_1 - p_2$). There is also the chance of p for each of *Alice* and *Bob* to choose a certain base (if they don't use biased sifting, it is $p = \frac{1}{2}$). The average rate (or probability) of introducing an error on one of the two bases for if is now as shown in equation 2.34. (Lo et al., 2005, p.152)

$$\bar{e} = \frac{p^2 p_2 + (1-p)^2 p_1}{2(p^2 + (1-p^2))} \tag{2.34}$$

If *Eve* listens to only one base, equation 2.34 becomes dependent only to p. If further *Alice* and *Bob* use biased sifting with $p \to 0$ and the base *Eve* listens to is the prefered one, the error rate for the other base tends to zero. On the chosen base, *Eve* naturally does not introduce any errors at all. Therefore the overall error rate is near zero and *Eve* remains undetected, while gaining much knowledge about the key. To circumvent this case, *Alice* and *Bob* might split the error rate and calculate it separately for each base. The distributed key is secure if both error rates are sufficiently low. This is because if *Eve* listens only to the one highly favored base, the error rate on the other base will be $\frac{1}{2}$, which is generally sufficient for detection. (Lo et al., 2005, p.152)

Eve could achieve perfect eavesdropping if she had a quantum cloning device. However, as described in section 2.2.4, the *no-cloning theorem* says that perfect cloning of arbitrary states is impossible. Such a device could only clone identical or orthogonal states, which would not do any good. *Eve* would still have to know the right base beforehand to ensure the correctness of the clone, as the two bases are nonorthogonal to each other. (Bennett & Brassard, 1984, p.176) Furthermore they have to exchange bits for error correction which occur on the line for other reasons.

2.5.2.2 Error Correction

The second part of reconciliation is the error correction. A protocol to correct errors will have to disclose some information in order to correct the key string. Let A be a key string sent by *Alice* and B a sifted key string received by *Bob* (A will likely be $\neq B$, making error correction necessary after all). Equation 2.35 defines reconciliation protocol R^p, where S is the produced corrected key string and Q is the amount of information exchanged (and thus to be discarded) on the public channel. A reconciliation protocol is further ε-robust if the probability to fail is $< \varepsilon$. (Brassard & Salvail, 1994, p.413)

$$R^p(A, B) = [S, Q] \tag{2.35}$$

The leaked information after the conclusion of the protocol $I_E(S|Q)$ should naturally be as small as possible. However, the *noiseless coding theorem* sets lower boundary of 1 in conjunction with the *conditional Shannon index* (confer to equation 2.13 in section 2.1.4). If a reconciliation protocol reaches this boundary, one can speak of a *optimal reconciliation protocol* as depicted in equation 2.36. (Brassard & Salvail, 1994, p.413) One could conclude from this equation, that the more equal the distribution between erroneous and non-erroneous $H(X \oplus Y) = nh(p)$ bits in a key string is, the more disclosure is acceptable from an optimal protocol.

$$\lim_{n\to\infty} \frac{I_E(S|Q)}{nh(p)} = 1 \tag{2.36}$$

Furthermore, a protocol is *efficient*, if the expected runtime for an input string length n is lower than a polynomial time $t(n)$. Said protocol is ideal, if it is both optimal and efficient. As optimality is a theoretical condition, in practice the target is to reach a certain boundary for the information leakage. This boundary over the theoretical limit is called ζ. A protocol ${R_\zeta}^p$, which remains below this boundary (so that equation 2.36 fulfills $\leq 1 + \zeta$ instead of $= 1$) and fulfills all other aspects of ideality is called *almost-ideal*. (Brassard & Salvail, 1994, p.417)

2.5.2.3 Cascade

The *Cascade* protocol represents an *almost-ideal* reconciliation protocol, as it is close to fulfilling equation 2.36 in a BSC[9] with $p = 0.015$. Cascade is an iterative protocol and *Alice* and *Bob* agree to the number of iterations beforehand. In iteration one, they divide the key bit string into chunks of size k_1 (the bit position is defined by $K\upsilon^1\{l|(\upsilon - 1)k_1 < l \leq \upsilon k_1\}$), and exchange the parities of each of the blocks and examine the differences. (Brassard & Salvail, 1994, pp.419-420) For each block with a deviating parity the perform essentially a binary search for that bit, by exchanging parities of key string portions of of continuously decreasing size, and correct it. (Brassard & Salvail, 1994, p.417) In conclusion the whole key string has an even number

[9] see footnote 5

(including zero) of errors. The next iterations (i), divide the string into randomized chunks of size k_i and repeat the procedure. Equation 2.37 shows the function to randomly choose the bit positions for i iterations. After a sufficient number of iterations i, each of the k_i blocks has an even number of errors (including zero). (Brassard & Salvail, 1994, p.420)

$$\begin{aligned} {K_j}^i &= l | f_i(l) = j \\ fi &: [l..n] \rightarrow [1..\frac{n}{k_i}]. \end{aligned} \tag{2.37}$$

This results in a key string with a very low error probability. (Pivk, 2010b, p.39) Figure 2.9 depicts the protocol's mode of operation and the segmentation of the blocks. It makes obvious how randomizing portions of the key strings help to detect erroneous bits.

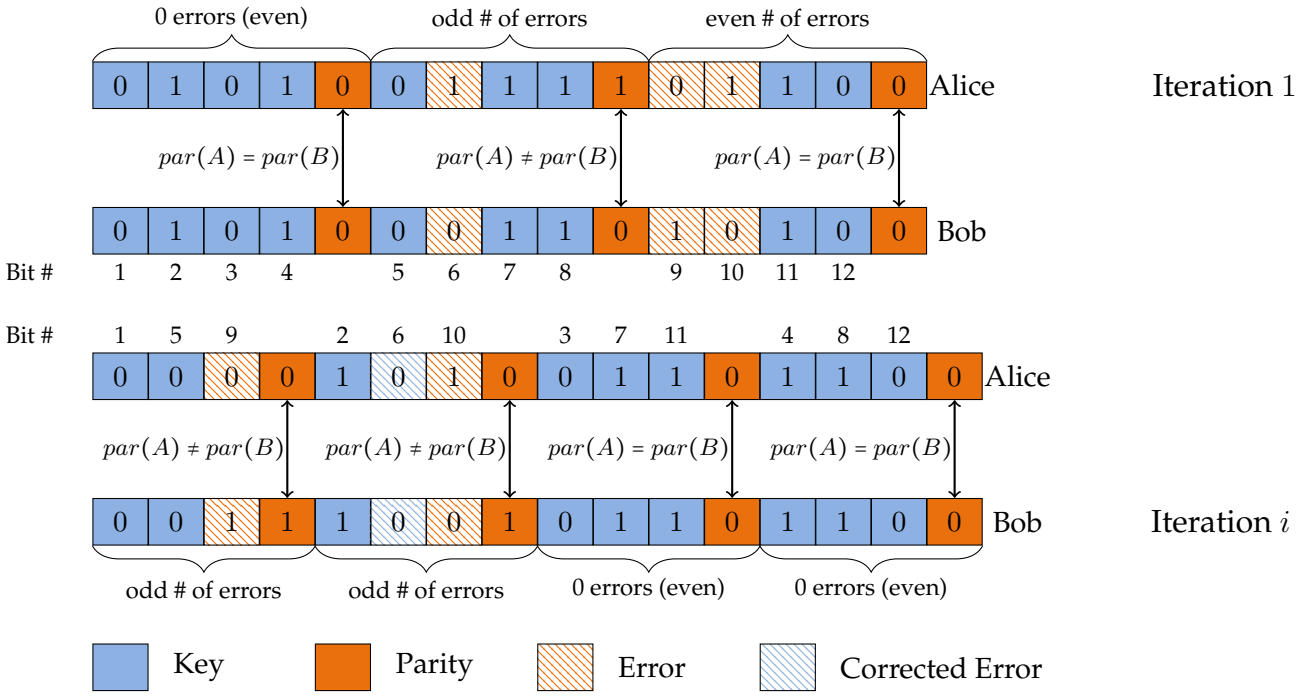


Figure 2.9: Error Correction with Cascade

2.5.2.4 Reconciliation of Continuous Variable Systems

In Continuous Variable QKD systems (see section 2.4.6), both the sender, *Alice*, and the receiver, *Bob*, use a quantizer to extract data from the particle beam. *Alice* subsequently sends *Bob* a compressed version of her quantized stream, while *Bob* has his own quantized stream as side information (thus, this process resembles source coding with side information). (Bloch, Thangaraj, McLaughlin, & Merolla, 2006, p.2) The quantization splits the continuous stream in an even number of k intervals (Bloch et al., 2006, p.4). A label with length $l = \log_2 k$ describes each digit. The optimal method for reconciliation is to apply l different LDPC codes (see section 2.1.5) to each of the l binary sequences - this method is called Multilevel Coding / Multistage Decoding (MLC/MSD) - whereas another method ,Binary Interleaved Code Mod-

ulation (BICM), uses a single LDPC code on an combined interleaved version of all sequences. (Bloch et al., 2006, p.3-4) In this system, LDPC codes combined with the MLC/MSD method performed well, using unusually long block sizes of $200,000$ (Bloch et al., 2006, p.8). A practical implementation over 25 km fiber also relied on this reconciliation method (Lodewyck et al., 2007, p.16).

2.5.3 Privacy Amplification

Privacy Amplification means to extract a non-disclosed part out of partly disclosed information. This is necessary, as an eavesdropper *Eve* may obtain information about the key bits exchanged by a sender *Alice* and a receiver *Bob*. *Eve* may obtain a set V consisting of a most t bits from a key string W with length n. The goal of the privacy amplification is to provide a function $g : \{0,1\}^n \rightarrow \{0,1\}^r$ that keeps *Eve*'s knowledge about a g-derivate of W, namely $K = g(W)$, arbitrarily small in order for *Alice* and *Bob* to use it as a key secret, from *Eve*'s knowledge. (Bennett, Brassard, Crepeau, & Maurer, 1994, p.1915) In other words, the knowledge of *Eve* about K should be less than an arbitrary small ε, or $I(K;GV) \leq \varepsilon$. *Alice* and *Bob* have no information about the distribution of *Eve*'s knowledge P_{VW} except that it has to fulfill some constraints. Some constraints derive from quantum physics, while others derive from the type of information[10] V about W that *Eve* obtains. The latter determine the length r of the resulting key K: the closer the ties between the V and W, the smaller is r and vice versa. (Bennett et al., 1994, p.1918) To achieve the desired compression, *Alice* and *Bob* may use *universal hash classes* as described in section 2.1.3 (Bennett et al., 1994, p.1919).

2.6 The Austrian Institute of Technology Quantum Key Distribution System

The AIT uses a hardware system that generates EPR entangled quantum pairs for QKD via fiber cable or free space. Yet, it utilizes the BB84 protocol using photon polarization ($0/90°$ and $\pm 45°$) as qubit bases forming the quantum channel. The AIT QKD software handles all necessary steps to post-process to exchange of a QKD key over the BB84 protocol. (Austrian Institute of Technology, 2013) It therefore manages all QKD tasks outside the quantum channel, which means all tasks in the sender (*Alice*) and receiver (*Bob*) systems and the public channel.

The software architecture consists of these components:

- Quantum event;
- QKD stack;
- QKD network;
- Desktop Bus (DBUS) management.

[10] This type may be direct bits from W, parity bits or mapping functions results (Pivk, 2010b, p.41)

The software itself is designed to be hardware agnostic and therefore not bound to the AIT QKD hardware. The quantum event part represents the exchange on the quantum channel. The QKD stack handles the necessary steps for a complete QKD negotiation:

- Hardware pickup and presifting;
- Sifting (BB84);
- Error estimation;
- Error correction (Cascade and LDPC);
- Privacy amplification;
- Confirmation.

After these steps, the two parties, *Alice* and *Bob*, share a common key. The QKD network provides now a key store, a crypto engine (to use the quantum-distributed on data to be sent over the public channel) and the Quantum Point-to-Point Protocol (Q3P) (see below). Eventually, the DBUS management provides system management via DBUS[11] (Maurhart, 2013, pp.9-11).

Q3P rests on the classical Point-to-Point Protocol (PPP) and provides a logical QKD connection between two nodes. It consists of the Q3P node, which represents a peer in a connection, and the Q3P link, which represents a connection between exactly two QKD nodes. If a node maintains Q3P connections to more than one adjacent node, every connection has its own Q3P link. Furthermore, every Q3P link maintains its own key store and crypto engine. (Maurhart, 2010, pp.156-157)

Section lists the software requirements for the AIT QKD software.

[11] DBUS is a free inter-process communication software. See `http://freedesktop.org/wiki/Software/dbus/` for details.

3 Fundamentals of Internet Protocol Security

Internet Protocol security (IPsec) is a protocol suite with the purpose of ensuring integrity, authenticity and confidentiality for connections over the public internet. The original specification dates from 1998 (updated 2005) (Kent & Atkinson, 1998; Kent & Seo, 2005). IPsec operates at the Internet Protocol (IP) layer (Kent & Seo, 2005, p.1). It functions as a perimeter between protected and unprotected network interfaces by requiring a protection level, letting pass unprotected or discarding traffic transiting from the unprotected to the protected network. Both host and gateway systems may implement IPsec. (Kent & Seo, 2005, pp.7-8) Therefore, the objectives to use IPsec may be either to secure point-to-point or site-to-site connections.

3.1 General Architecture

A Security Association (SA) has the purpose to enforce a defined security policy (Kent & Seo, 2005, p.19). Therefore, IPsec uses two databases to determine how to handle packets:

- The Security Policy Database (SPD), containing the policies;
- The Security Association Database (SAD), containing the associations.

The subsequent sections describe these two databases and their function in the IPsec protocol suite.

3.1.1 Security Policy Database

The SPD determines which of three actions (*PROTECT, BYPASS, DISCARD*) an IPsec host has to take with a transiting packet (Kent & Seo, 2005, p.20). An entry into this database is either labeled *SPD-O* for outbound direction (traffic from protected to unprotected) and *SPD-I* inbound direction (vice versa) for bypassed or discarded traffic and *SPD-S* for IPsec-protected (*s* stands for secure) traffic. (Kent & Seo, 2005, p.21) These entries may contain the following criteria to identify the connections for which to apply the above actions (Kent & Seo, 2005, pp.26 - 29):

- The local IP (v4 or v6) address;
- The remote IP (v4 or v6) address;

- The next layer protocol (may also be *ANY* or *OPAQUE*[1] and may contain protocol information such as port numbers);
- A name if addresses are unsuitable, like a fully qualified user[2] or domain name.

These so-called *selectors* determine the SPD entries which contain also values, which of the selectors may be derived from an IP packet instead of the database and the processing information (consisting of the three action mentioned above). If the processing requires to be *PROTECT* it holds additional information of how to conduct the IPsec protection (Kent & Seo, 2005, pp.31-32):

- IPsec mode (see section 3.3);
- IPsec protocol (see section 3.2);
- Cryptographic algorithms to use;
- Additional addresses if in tunnel mode;
- The use of extended sequence numbers and stateful fragment checking (see below);
- Bypass options for certain flags[3].

The handling of fragments needs particular attention in IPsec, as non-initial fragments[4] may not contain every value needed to allocate it to a corresponding SA. (Kent & Seo, 2005, pp.66) This may raise problems when a host sends a non-initial fragment to an IPsec gateway which does not have any record of the fragment sequence and the SA that would actually correspond requires information not included in the fragment. If the SPD entry corresponding to the fragments addresses contains next layer information other than *ANY* or *OPAQUE* (for instance a port number), and the fragment does not contain the corresponding information (in this case port number), the gateway will not be able to assign the fragment to the right SA. Therefore, a gateway might perform some sort of stateful fragment checking (this means keeping track of initial fragments and mapping subsequent fragments to the corresponding SA). (Kent & Seo, 2005, p.68)

A typical SPD entry (Linux 3.10.7 example) exhibits a structure as shown in listing 3.1.

```
10.0.11.0/24[any] 10.0.11.33[any] 255
out prio def ipsec
esp/tunnel/192.168.0.1-192.168.1.2/require
created: Aug 27 22:01:03 2013  lastused:
lifetime: 0(s) validtime: 0(s)
spid=17 seq=0 pid=8765
refcnt=1
```

Listing 3.1: Linux Example SPD Entry

[1] For IPv6 only; this means that the next layer protocol may not be determinated
[2] For instance an email address
[3] The Don't Fragment (DF) and Differentiated Services Codepoint (DSCP) bits
[4] Fragments other than the first one of a fragmented packet

3.1.2 Security Association Database

An SA corresponds to a directed link between two nodes. It is therefore necessary for bidirectional communication to establish two SAs, one for each direction. (Kent & Seo, 2005, p.12)

The SA *must* contain the following items (Kent & Seo, 2005, p.36):

- Security Parameters Index (SPI) (see below);
- Sequence Number Counter [5] and Counter Overflow flag[6];
- Anti-Replay Window [7];
- Algorithm, key and parameters for Authentication Header (AH)[8] (see section 3.2.1);
- Algorithms, keys and parameters for Encapsulating security Payload (ESP)[9] (see section 3.2.2);
- SA lifetime (see below);
- IPsec mode (see section 3.3);
- Use of stateful fragment checking (see section 3.1.1);
- DSCP values;
- Bypass options for certain flags[10];
- Observed path Maximum Transmission Unit (MTU) and aging variables;
- Tunnel header IP source and destination (see section 3.3.2).

Every SA may have a lifetime after whose expiration the SA is invalid. The lifetime may be measured in time or data volume or both (every node *must* support all three possibilities). After the expiration, a new SA must replace the expired or the connection must terminate. (Kent & Seo, 2005, p.37)

The SPI is an arbitrary, mandatory 32-bit value in an IPsec packet and, correspondingly, in a SAD. The SPI identifies solely or in conjunction with IP address and IPsec protocol information, which SA corresponds to an IPsec packet. The receiver of an SA generates the SPI, therefore a central Group Controller has to perform this task in multicast environments. (Kent, 2005b, p.10)

[5] A counter for generating sequence numbers
[6] Which determines whether the counter may overflow or not
[7] a counter to identify replay packets
[8] If AH is supported; this refers mainly to the authentication algorithm and key
[9] If ESP is supported; this refers mainly to the encryption, integrity or combined mode algorithm, key, mode and initial vector
[10] See footnote 3

A typical SAD entry (Linux 3.10.7 example) shows a structure as shown in listing 3.2.

```
10.0.11.41 10.0.11.33
esp mode=transport spi=65537(0x00010001) reqid=0(0x00000000)
E: des-cbc  3ffe0501 4819ffff
A: hmac-md5  61757468 656e7469 63617469 6f6e2121
seq=0x00000000 replay=0 flags=0x00000000 state=mature
created: Aug 27 22:01:03 2013   current: Aug 27 22:05:12 2013
diff: 249(s)    hard: 0(s)      soft: 0(s)
last:                           hard: 0(s)      soft: 0(s)
current: 0(bytes)       hard: 0(bytes)  soft: 0(bytes)
allocated: 0    hard: 0 soft: 0
sadb_seq=0 pid=8873 refcnt=0
```

Listing 3.2: Linux Example SAD Entry

3.2 Security Protocols

IPsec provides two security protocols with overlapping purposes (with their respective IP protocol numbers in parentheses) (Kent & Seo, 2005, p.9):

- Authentication Header (AH) (51);
- Encapsulating security Payload (ESP) (50).

As IPsec is an integral component of Internet Protocol version 6 (IPv6), the headers for AH and ESP where amongst the six Extension Headers (EHs) in the original IPv6 specification (Deering & Hinden, 1998, p.7).

3.2.1 Authentication Header

The AH provides authentication (hence the name), integrity and replay protection services for IP traffic. It does, however, not provide encryption, which is the objective of the ESP (see section 3.2.2). The protection concerns the whole IP datagram, with the exception of some mutable fields in the IP header, such as Time-to-Live (TTL) (this distinguishes the protocol from ESP). These fields are not sensible to protect, as intermediate devices (for instance routers) could (and actually do) alter them as the packet traverses along the path between two end hosts. (Kent, 2005a, p.3) The AH contains the following components with the number of bytes in parentheses (Kent, 2005a, p.4):

- Next header field (1 byte);
- Payload length (1 byte);
- Bits reserved for future use (2 bytes);

- Security Parameters Index (SPI) (4 bytes) - see section 3.1.2;
- Sequence number (4 bytes);
- Integrity Check Value (ICV) (variable length).

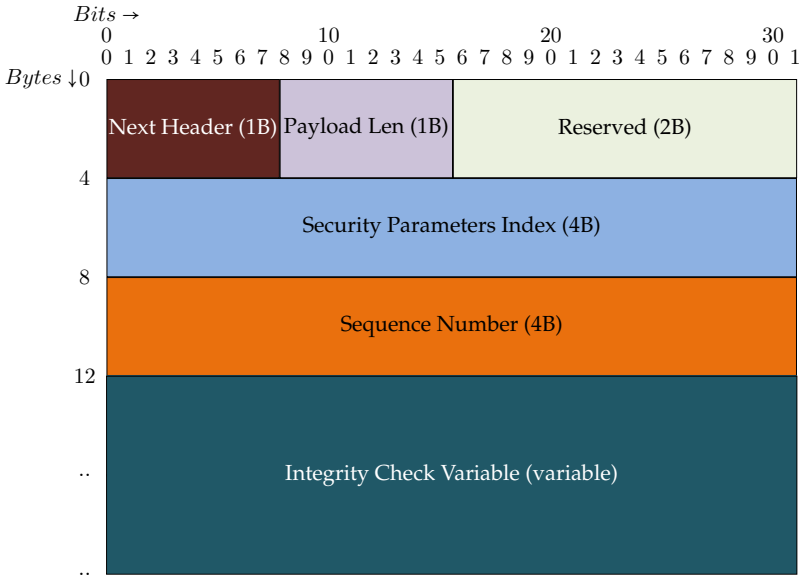


Figure 3.1: AH Packet Structure (Kent, 2005a, p.4)

Figure 3.1 depicts the AH. The position of the AH depends on the IPsec mode of operation (see section 3.3). The *next header* field indicates the following protocol or IPv6 EH, according to the IP protocols list maintained by the Internet Assigned Numbers Authority (IANA)[11]. The payload length indicates the length of the AH. This is necessary, as the Integrity Check Value (ICV) may have a variable length. The value calculates the header length in 32 bit (4 byte) blocks minus 2 ($\frac{Hlen[byte]}{4} - 2$). (Kent, 2005a, p.5) The sequence number counts forward (starting with zero) for each packet that is sent via the corresponding SA. The receiver of an IPsec transmission *may* check the sequence number for anti-replay purposes. (Kent, 2005a, pp.12,19) For high-speed transmissions, the communication partners may agree to extend the sequence number to 64 bits. In this case, the packet contains only the low-order 32 bits. (Kent, 2005a, pp.8,12)

The ICV contains a cryptographically generated checksum over the whole sent packet except for the parts of the IP header which are neither immutable nor predictable. This also includes implicit parts like the (unsent) high-order parts of an extended (64 bits) sequence number and implicit paddings. (Kent, 2005a, p.13) IPv6 EHs must have length of a multiple of 64 bits. Some algorithms may not fulfill this requirement, therefore their output may need padding. (Kent, 2005a, p.16) The receiver of a packet recalculates the value and compares it with the sent one. If they do not match, he *must* discard the packet. (Kent, 2005a, p.20) The SA defines the algorithm for the computation for the value. This may be a keyed Message Authentication Code (MAC). (Kent, 2005a, p.11) The IPsec standard requires that implementations *must* support *HMAC-SHA1-96*, *should* support *AES-XCBC-MAC-96* and *may* support

[11] see `http://www.iana.org/assignments/protocol-numbers/` for the list

HMAC-MD5-96 (Manral, 2007, p.6). A later work, however, discourages the use of Hashed Message Authentication Codes (HMACs) based on *MD5* and *SHA1* (Hoffman, 2007, p.3). Moreover, the National Institute of Standards and Technology (NIST) disapproves with *AES-XCBC-MAC-96* - see section 5.5.2). Ultimately, the protective services of AH rest on the ICV and its computation.

3.2.2 Encapsulating Security Payload

The ESP provides, in conjunction with an AH or solitary, confidentially, integrity and authenticity as well as replay protection and Traffic Flow Confidentiality (TFC) for IP datagrams. Using the confidentiality without integrity, however, is discouraged. (Kent, 2005b, p.3) Apart from that, it is also valid to use ESP for integrity checking without encryption but not to use it without both. (Kent, 2005b, p.20) The packet format consists of the following components with the number of bytes in parentheses (Kent, 2005b, p.9):

- Security Parameters Index (SPI) (4 bytes) - see section 3.1.2;
- Sequence Number (4 bytes)- see section 3.2.1;
- Initialization Vector (IV) (optional, variable length);
- Payload data (variable length);
- Traffic Flow Confidentiality (TFC) padding (optional, variable length);
- Padding (0-255 bytes);
- Padding length (1 byte);
- Next header field (1 byte) - see section 3.2.1;
- Integrity Check Value (ICV) (variable length) - see section 3.2.1.

Figure 3.2 depicts the ESP packet structure. Apart from the components already described in other sections, the ESP header contains an optional Initialization Vector (IV). Some algorithms may use this vector as starting point for cryptographic calculations (see section 3.5.2). Though the IV, is not an explicit field in ESP, algorithms that use an IVs *must* specify the location, length, structure and conjunction with ESP in an Internet Request for Comments (RFC). Immediately afterwards the actual payload data follows, which means the contained datagram (which may be an upper layer or IP datagram, depending on the operation - see section 3.3). (Kent, 2005b, p.13) Additionally to disguise the structure of sent packets, an ESP peer may add supplementary padding to the data. This feature is called TFC. For advanced traffic flow protection, an ESP implementation is able to generate bogus packets. (Kent, 2005b, p.17) It uses the next header field to indicate subsequent bogus packets, which do not need to contain well formatted payloads, yet have exhibit valid ESP fields. (Kent, 2005b, p.16). Both IV and TFC padding are transparent to ESP and, thus, considered as part of the payload. Furthermore, the IV is usually unencrypted. (Kent, 2005b, p.9, 13) Figure 3.3 shows this possible inner payload structure.

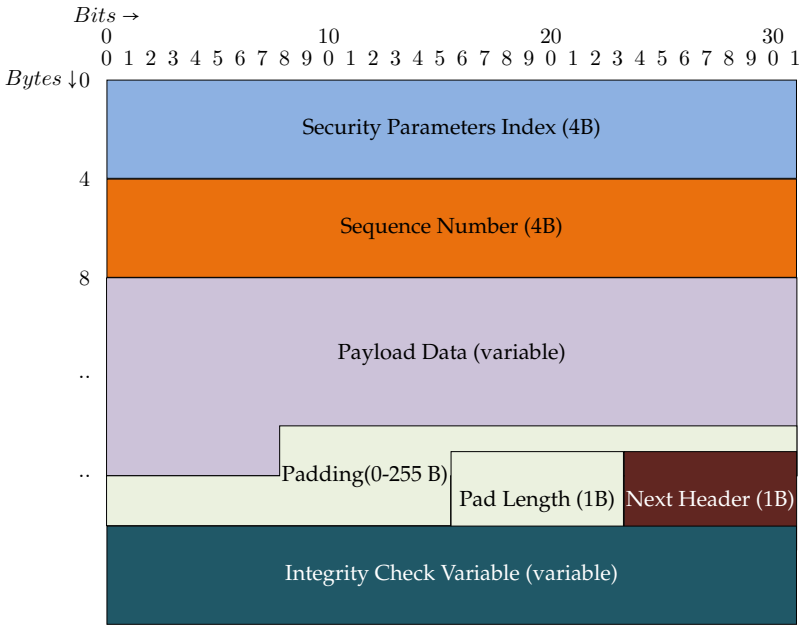


Figure 3.2: ESP Packet Structure (Kent, 2005b, p.7)

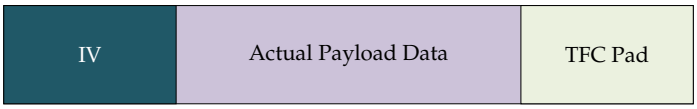


Figure 3.3: Possible Inner IP Payload Structure in ESP

Some cryptographic algorithms may also use padding (no to be confused with the TFC padding) to adjust the payload data to a specific block size needed to encrypt the data. The padding may serve also the purpose to guarantee a valid length of a resulting ciphertext, which must be a multiple of 4 bytes. The subsequent padding specifies the padding length in bytes. (Kent, 2005b, p.14) The packet encryption in ESP covers the payload (except the IV), padding and next header fields, while the integrity check covers the whole packet, except the ICV (if separate encryption and integrity algorithms are used). Combined algorithms encrypt and check the integrity of data at once. They however only perform checks on the encrypted parts and therefore the SPI and sequence number are unchecked. Thus, security requires such an algorithm to replicate these values into the encrypted part in order to ensure their integrity (Kent, 2005b, p.6). An ESP implementation *must* support the encryption algorithms *AES-CBC*, *TripleDES-CBC* and the *NULL* algorithm, it *should* support *AES-CTR*, while it *should not* support *DES-CBC*. For authentication, it *must* support *HMAC-SHA1-96*, *should* support *AES-XCBC-MAC-96* and *may* support *HMAC-MD5-96* and the *NULL* algorithm. (Manral, 2007, pp.4-5) Nevertheless, as with AH, a more recent paper discourages the use of *MD5* and *SHA1* HMAC functions (Hoffman, 2007, p.3). The *NULL* algorithm is mathematically defined by the *identity function* ($NULL(b) = I(b) = b$) and does not alter the data. (Glenn & Kent, 1998, p.1-2) Authentication and encryption *must not* use the *NULL* algorithm simultaneously. (Manral, 2007, p.5)

3.3 Modes of Operation

There are two principal modes of operation for the IPsec protocol (Kent & Seo, 2005, p.9):

- The tunnel mode;
- The transport mode.

The transport mode is intended to protect traffic between end hosts. Although it may tunnel (which means to encapsulate) traffic (*in-IP tunneling*) and be used by intermediate systems, these systems must not process traffic of interfaces not belonging to themselves in this manner. (Kent & Seo, 2005, p.14) In contrast, the tunnel mode serves the purpose to encapsulate traffic from and to a security gateway. A gateway should use tunnel mode only for connections, for which the gateway itself serves as an end host (for instance a management connection). (Kent & Seo, 2005, p.15)

3.3.1 Transport Mode

In IPsec transport mode, the respective Header comes immediately after the IP header. The AH has precedence over the ESP header and comes first (if both headers are present). (Kent, 2005a, p.10) The ESP header comes before the next layer protocol. In this manner, the AH fully authenticates the ESP header. (Kent, 2005b, p.18) In IPv6, the EH order is as follows: Hop-by-Hop Options, Routing (Type 0), Fragment, Destination Options, Authentication, Encapsulating Security Payload. (Deering & Hinden, 1998, p.7) Thus, both IPsec headers reside at the end of an EH chain, again with AH before ESP. The sections 3.2.1 and 3.2.2 discuss the packet parts protected by both protocols. Figure 3.4 depicts the packet construction and protection areas of them in transport mode.

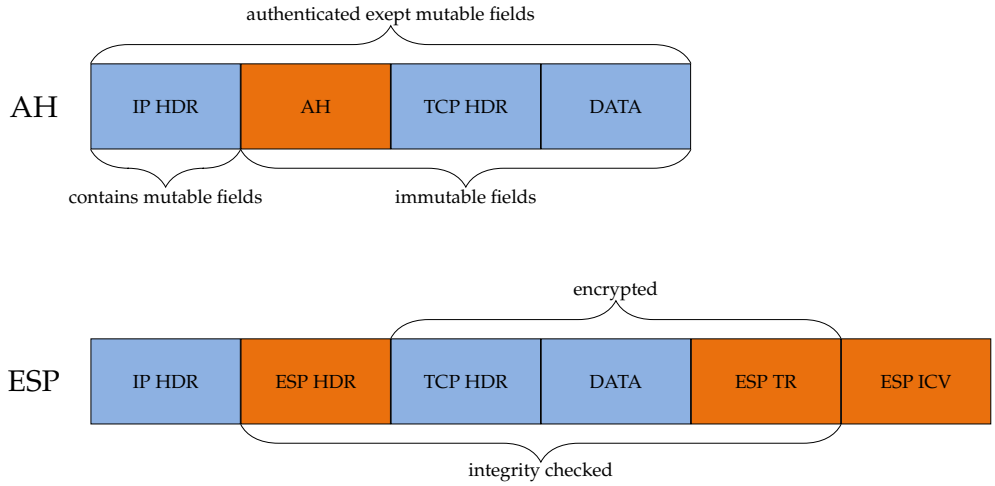


Figure 3.4: IPsec Transport Mode (Kent, 2005a, p.10) and (Kent, 2005b, p.10)

3.3.2 Tunnel Mode

The tunnel mode encapsulates an IP packet into another. Therefore, there are two IP headers: an inner and an outer. While the inner represents the original header (including unimpaired IPv6 EHs) of the packet, the outer header addresses to the tunnel terminators, which encapsulate and decapsulate the tunneled packet and subsequently forward it. (Kent & Seo, 2005, pp.55-56) An IPsec gateway in tunnel mode alters only three fields of the inner header. The Explicit Congestion Notification (ECN) field moves to the outer header and is replaced by 0 or 1, the TTL field is decremented as usually by intermediate systems and the checksum must be recalculated in order to match the new field values. As IPv6 has no checksum field, this alteration is omitted in that protocol. (Kent & Seo, 2005, pp.57-59) The protection level is similar to the transport mode, only that it concerns the whole encapsulated packet. The AH protects also the before mutable fields of the inner header, which will not be altered by intermediate device while encapsulated (only the ones of the outer header) (Kent, 2005a, p.11). In ESP, the encryption and/or authentication concerns now also the (inner) IP header (Kent, 2005b, p.19). Figure 3.5 depicts the packet construction and protection areas of the AH and ESP in tunnel mode.

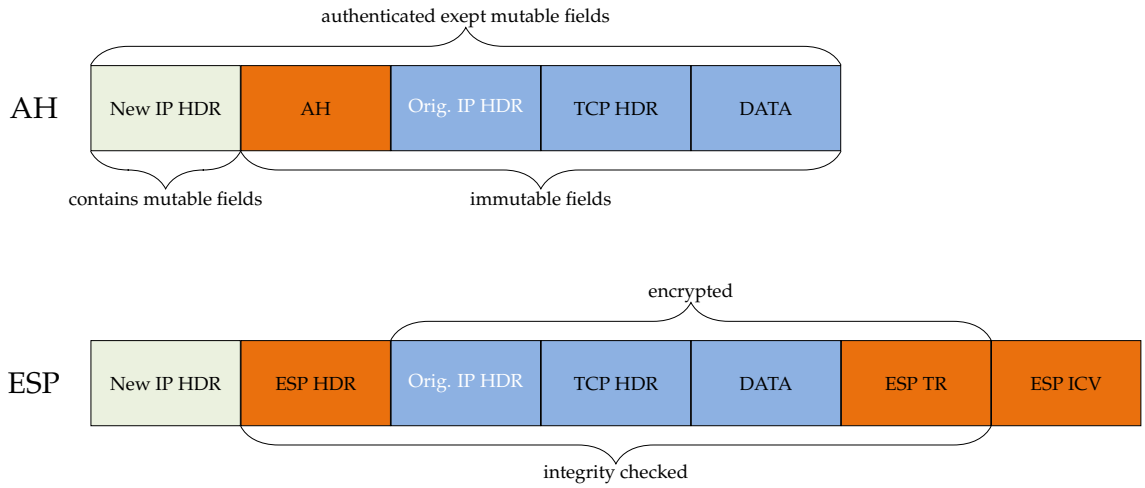


Figure 3.5: IPsec Tunnel Mode (Kent, 2005a, p.11) and (Kent, 2005b, p.20)

3.4 Key management

IPsec uses two methods to exchange secret keys to protect data with cryptographic algorithms:

- Manual keying;
- Automatic keying.

Manual keying means that a person enters the necessary secrets onto the peers by hand. This is only sensible for manageable installations and involves mostly symmetric keys. Bigger installations with the demand of a secure connection as needed require feasible rekeying (which

involves the establishment of a new SA) and, thus, an automated method for key exchange. Although an IPsec implementation *may* provide other means for automatic keying, the default is Internet Key Exchange (IKE). (Kent & Seo, 2005, p.48) The following sections describe this protocol and its successor Internet Key Exchange version 2 (IKEv2).

3.4.1 Internet Key Exchange Version 1

The purpose of IKE is to provide authenticated secret key material for SAs. (Harkins & Carrel, 1998, p.2) The protocol takes place in two phases. The first phase establishes an authenticated, bidirectional, secure link - the Internet Security Association and Key Management Protocol (ISAKMP) SA- which the second phase subsequently uses to negotiate the IPsec SAs. (Harkins & Carrel, 1998, pp.5-6) This thesis, however, does not discuss this protocol in-depth as there is a newer version (see section 3.4.2), which obsoletes IKE (Kaufman, 2005, p.1). This protocol, IKEv2, possesses itself already an overhauled version (Kaufman, Hoffman, Nir, & Eronen, 2010, p.1).

3.4.2 Internet Key Exchange Version 2

IKEv2 manages SAs between IPsec peers. (Kaufman et al., 2010, p.1) The protocol establishes own SAs (IKE SAs) in order to negotiate parameters for the IPsec SAs. It therefore serves the same purpose and works similarily as IKEv1, yet the two protocols are not compatible [12]. The protocol consists of four commonly named pairs (request and response) of messages, which are called message types (Kaufman et al., 2010, p.5):

- *IKE_SA_INIT*;
- *IKE_AUTH*;
- *CREATE_CHILD*;
- *INFORMATIONAL*.

IKEv2 is encapsulated in a User Datagram Protocol (UDP) packet (port number *500* or *4500*). The protocol has its own header which consists of (Kaufman et al., 2010, p.71):

- IKE SPI for the initiator (8B) - see below;
- IKE SPI for the responder (8B) - see below;
- Next Payload (8B) - the immediatly following payload;
- Major Version (4b) - 2 for IKEv2;
- Minor Version (4b) - 0 for IKEv2;

[12] From this point forward mentions of IKE refer to IKEv2 unless stated otherwise

- Exchange Type (1B) - see above;
- Flags (1B) - see below;
- Message ID (4B) - used for flow control;
- Length (4B) - total message length including header and payloads;

The IKE SPI is similar to the IPsec SPI (see section 3.1.2), only that it is longer and the SPIs for both directions are in the same header. The responder SPI must naturally be zero for the first initiating packet. There are also three defined Flags (residing in the third, fourth and fifth bit of the flag field): *R* (response), *V* (version) and *I* (initiator). The *R* and *I* flags serve the purpose to correlate the SPIs and a set *R* flag prohibits a further response (as a response may follow only after a request). The *V* flag indicates that the sender of the packet is able to speak a higher version of IKE than the actually spoken. (Kaufman et al., 2010, pp.71-72) Figure 3.6 depicts the IKEv2 header.

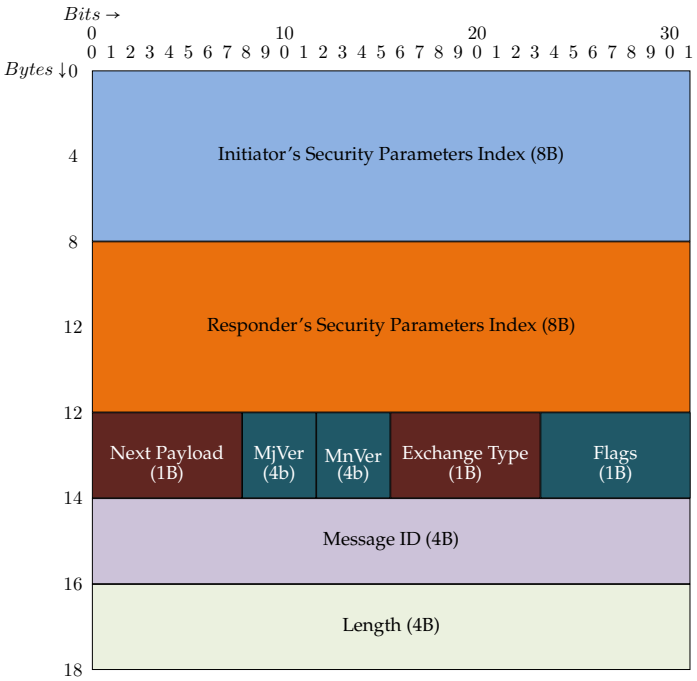


Figure 3.6: IKEv2 Packet Header (Kaufman et al., 2010, p.71)

As an IKEv2 packet may contain numerous payloads, each of them has its own payload header. This header includes a field for the next payload, which is 0 if the current header is the last, while the first is determined by the next header field of the IKEv2 packet header. Please refer to table 3.1 for a list of the possible payloads. The *critical* bit determines whether (0) or not (1) to ignore the payload if it is unknown to the recipient. (Kaufman et al., 2010, pp.73-74) The other fields are self-explanatory. Figure 3.7 depicts the generic payload header.

Ordinarily, IKEv2 uses one *IKE_SA_INIT* and one *IKE_AUTH* - resulting in four messages - to establish an IKE SA. (Kaufman et al., 2010, p.5) The first message, an *IKE_SA_INIT* request, contains (apart from the header) a proposal of supported cryptographic algorithms to

No Next Payload		0
Security Association	*SA*	33
Key Exchange	*KE*	34
Identification - Initiator	*IDi*	35
Identification - Responder	*IDr*	36
Certificate	*CERT*	37
Certificate Request	*CERTREQ*	38
Authentication	*AUTH*	39
Nonce	*Ni,Nr*	40
Notify	*N*	41
Delete	*D*	42
Vendor ID	*V*	43
Traffic Selector - Initiator	*TSi*	44
Traffic Selector - Responder	*TSr*	45
Encrypted and Authenticated	*SK*	46
Configuration	*CP*	47
Extensible Authentication	*EAP*	48

Table 3.1: Overview over used IKEv2 payloads

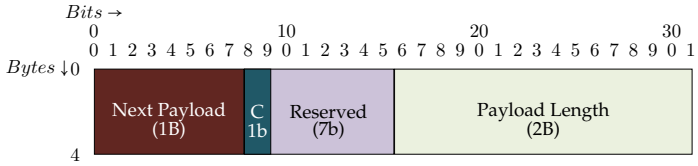


Figure 3.7: IKEv2 Generic Payload Header (Kaufman et al., 2010, p.73)

establish the IKE SA (payload *SAi1*[13]) - for the meanings of further payload values please refer to table 3.1), the initiator's *Diffie/Hellman* (DH) (*KEi*) value (see section 3.5.1) and the initiator's nonce (*Ni*). The subsequent reply contains the same items from the responder (*SAr1, KEr, Nr*) and a certificate request (*CERTREQ*). (Kaufman et al., 2010, p.10) The peers are now able to calculate a shared secret value with the DH algorithm. This secret, the *SKEYSEED*, is the basis for all keys in the IKE SA. Both communication directions obtain their own keys for encryption (*SK_e*) and authentication (*SK_a*). Additionally, the peers derive key for the establishment of subsequent SAs (*SK_c*). Please note that the peers encrypt and authenticate all subsequent messages (*SK*). The next following request consists of an authentication packet. The appropriate response is (also *SK*) . After that, a IKE SA is established. (Kaufman et al., 2010, pp.11-12) With a successfully established IKE SA, the protocol is able to create a *ChildSA*, which means an IPsec AH or ESP SA (Kaufman et al., 2010, p.5). The initiator (either of the two peers) sends therefore the sequence a *CREATE_CHILD_SA* request and receives an appropriate reply. If the negotiation fails, the IKE SA still remains established. Rekeying both IKE and IPsec SAs works in a similar way with *CREATE_CHILD_SA*. (Kaufman et al., 2010, pp.14-16) Table 3.2 gives an overview over the composition of these negotiation packets[14].

[13] In this context, suffix small letters *i* and *r* refer to the initiator and responder

[14] Conversations starting with the second one are encrypted and authenticated (*SK*); the payload references are as in table 3.1

Purpose	Initiator	Responder
Initial	HDR, SAi1, KEi, Ni	HDR, SAr1, KEr, Nr, [CERTREQ]
Initial (AUTH)	IDi, [CERT], [CERTREQ], IDr, AUTH, SAr2,TSi, TSr	IDr,[CERT] AUTH, SAr2, TSi, TSr
Establish ChildSA	SA, Ni, [KEi],TSi, TSr	SA, Nr, [KEr],TSi, TSr
IKE Rekeying	SA, Ni, KEi	SA, Nr, KEr
ChildSA Rekeying	N(REKEY_SA), SA, Ni,[KEi],TSi, TSr	SA, Nr, [KEr],TSi, TSr

Table 3.2: Overview over the Most Important IKEv2 Message Compositions

3.5 Cryptographic Algorithms

This section gives an overview over cryptographic methods for authentication and encryption in IPsec. The overview comprises the DH key exchange (used in IKE), the construction of MACs and symmetric encryption cipher modes and gives an overview of the used algorithms. The explanations do not describe the operational method of the cryptographic algorithms themselves, as this would be to extensive for the scope of this thesis. The respective sections for ESP (3.2.2) and AH (3.2.1) include the required and recommended algorithms to use as stated by Manral (2007).

The IANA maintains also a list of *cipher suites*, which represent a bundle of cryptographic algorithms and modes to be used for IPsec. A suite contains ESP authentication and encryption (none of the suites uses the AH) and IKE authentication, encryption, pseudo-random number generation, as well as the DH group to use. (Hoffman, 2005, p.1,6) Currently defined suites are (in brackets, the defining RFC) (Internet Assigned Numbers Authority, 2011):

- VPN-A [RFC4308];
- VPN-B [RFC4308];
- Suite-B-GCM-128 [RFC6379];
- Suite-B-GCM-256 [RFC6379];
- Suite-B-GMAC-128 [RFC6379];
- Suite-B-GMAC-256 [RFC6379].

While the first two are current corporate Virtual Private Network (VPN) choices, the latter four rest on implementations of the National Security Agency (NSA). (Law & Solinas, 2011, p.2) However, the NIST discourages the use of *VPN-A* and *VPN-B* (see section 5.5.2).

3.5.1 Diffie Hellman Key Exchange

The DH key exchange algorithm is able to compute a commonly shared secret for two parties by exchanging information publicly, while retaining the secret. Each one of two communication partners, *Alice* and *Bob*, compute a value K as shows equation 3.1. (Diffie & Hellman, 1976, pp.648-649)

$$K = \alpha^{xy} mod q \tag{3.1}$$

Hereby all values come from a finite field $GF(q)$ with a number of q elements. q is a prime number and α is a primitive element of GF, while x and y are random numbers from $1..q$. In order to independently calculate the key, *Alice* and *Bob* calculate the values X and Y as equation 3.2 shows. They may disclose them along with q and α, enabling them to compute K. (Diffie & Hellman, 1976, p.649)

$$\begin{array}{lllll} Alice & X = \alpha^x \mod q & & Bob: & Y = \alpha^y \mod q \\ & K = Y^x \mod q & & & K = X^y \mod q \end{array} \tag{3.2}$$

As long as x and y remain secret, an attacker has to solve equation 3.3 or a similar problem. This equation is not easily solved and as long this is the case, the system is secure. (Diffie & Hellman, 1976, p.649)

$$K = X^{(\log_\alpha Y)} \mod q \tag{3.3}$$

The method is an Internet Engineering Task Force (IETF) standard which also describes the generation method for α and q. (Rescorla, 1999, p.9)

3.5.2 Symmetric Encryption

Symmetric encryption algorithms enable two parties, *Alice* and *Bob*, to encrypt and decrypt a message with the same key. The ESP protocol uses symmetric encryption to keep its payload confidential. The corresponding SAs for each directional IPsec channel between *Alice* and *Bob* contain the keys for the encryption and IKE handles the necessary preliminary exchange of the keys. Table 3.3 gives an overview of the currently defined IPsec encryption ciphers and their respective block sizes and key lengths (Internet Assigned Numbers Authority, 2012b)[15].

These algorithms are *block ciphers* (that means that they dissect data into blocks of a certain size and encrypt each block separately) there are certain modes of operation which ensure that the individual blocks depend on each other, as this straight-forward approach - called Electronic Code Book (ECB) - is unsuitable for most applications. (Menezes, van Oorschot,

[15] Although the IANA's list also contains RC4, Piper (1998, p.11) just reserves an identifier value for this algorithm. There is no known implementation.

Algorithm	Block Size	Key Length	Source
AES	128	128, 192, 256	(National Institute of Standards and Technology, 2001, p.1)
DES	64	64	(National Institute of Standards and Technology, 1999, p.8)
3DES	64	64	(Barker & Barker, 2012, p.3)
RC5	<=128	<=256	(Baldwin & Rivest, 1997, pp.2-3)
IDEA	64	128	(Lai & Massey, 1991, p.1)
CAST	64	<=128	(Adams, 1997, p.8)
BLOWFISH	64	<=448	(Schneier, 1994, p.191)
SEED	128	128	(Lee et al., 2005, p.1)
CAMELLIA	128	128, 192, 256	(Matsui et al., 2004, p.1)

Table 3.3: Overview over ESP cipher block and key lengths

& Vanstone, 1996, p.228) An obvious reason is that identical plaintexts yield identical ciphertexts, which may ease cryptoanalysis. One of these modes used with the standardized IPsec algorithms is Cipher Block Chaining (CBC). This mode incorporates the preceding block into the cryptographic calculation of the current. For the first block has no predecessor, it needs an IV (Dworkin, 2001, p.10). This value needs to be unpredictable but does not have to be secret (Dworkin, 2001, p.27). Equation 3.4 shows the encryption and decryption calculations, where C means ciphertext and P plaintext with the block number as subscript. The functions f and f^{-1} are the encryption and decryption functions with key k, $\oplus$ is an Exclusive Or (XOR) operation and n the number of blocks. (Dworkin, 2001, pp.10-11) Figure 3.8 shows the CBC mode.

$$\begin{aligned} &Encryption: && C_1 = f_k(P_1 \oplus IV) \\ & && C_j = f_k(P_j \oplus C_{j-1}); \quad j = 2...n \\ &Decryption: && P_1 = f_k^{-1}(C_1) \oplus IV \\ & && P_j = f_k^{-1}(C_j) \oplus C_{j-1}; \quad j = 2...n \end{aligned} \tag{3.4}$$

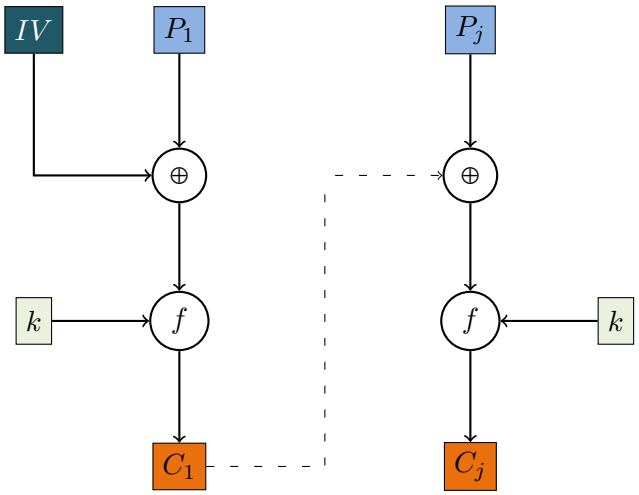


Figure 3.8: CBC block cipher mode (Dworkin, 2001, p.10)

A second cipher mode is Counter (CTR). In this mode, the cryptographic function encrypts a counter value which is a constantly changing input block with the only restriction that a counter block must not repeat with the same key value[16]. An XOR function combines the plaintext with the encrypted counter value to the ciphertext. The last block may not be full, therefore the last round takes only so many bits from the encrypted counter value as there are unencrypted plaintext bist left. In this mode, the same function (instead of an inverse) decrypts the encrypted data. Furthermore, as the blocks are independently encrypted, a possible error destroys only the very block of its occurence. Equation 3.5 shows the encryption and decryption, where T_j are the counter values, u is the length of the last block and MSB_u are the u most significant bits. The other values are as defined above. (Dworkin, 2001, p.15) Figure 3.9 gives an overview over the CTR mode.

$$\begin{aligned} Encryption: \quad & O_j = f_k(T_j); && j = 1, 2...n \\ & C_j = P_j \oplus O_j; && j = 1, 2...n \\ & P_n = C_n \oplus MSB_u(On) \\ Decryption: \quad & O_j = f_k(Tj), && j = 1, 2...n \\ & P_j = C_j \oplus O_j && j = 1, 2...n \\ & P_n = C_n \oplus MSB_u(O_n) \end{aligned} \tag{3.5}$$

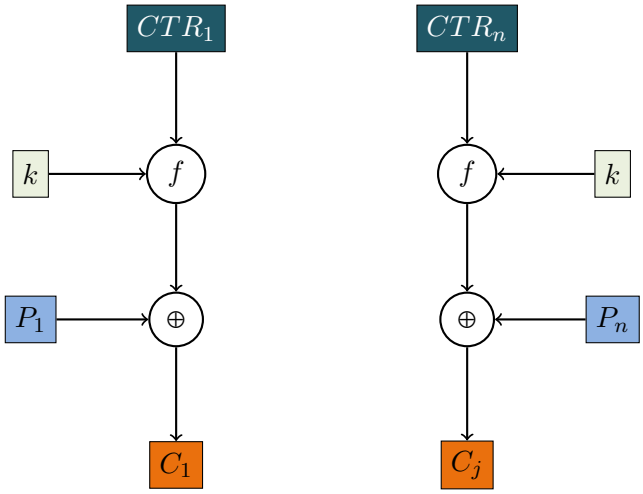


Figure 3.9: CTR block cipher mode (Dworkin, 2001, p.16)

In ESP, two more cipher modes are in use. Both of them combine encryption and authentication (Internet Assigned Numbers Authority, 2012b). The first one, Counter with CBC-MAC (CCM), is a combination of CTR and an authentication with a MAC (see section 3.5.3) in combination with the CBC mode (Dworkin, 2004, p.iii). The second one, Galois/Counter Mode (GCM), is a variation of CTR that uses a universal hash function (see section 2.1.3) distributed over a *Galois field* (Dworkin, 2007, p.1).

[16]Naturally, the value has to be the same for both communication partners. Therefore, the value must be easily calculated by them. It does, however, not have to be secret, a the secrecy derives from the key.

3.5.3 Message Authentication Codes

Message Authentication Codes (MACs) authenticate messages by using a keyed cryptographic one-way function on a message. They do not encrypt it but instead generate a hash value. (Frankel & Herbert, 2003, p.1) Table 3.4 shows currently defined IPsec hashing algorithms for use in the AH with their respective block and output sizes (Internet Assigned Numbers Authority, 2012a)[17].

Algorithm	Block Size	Output Size	Source
MD5	512	128	(Rivest, 1992, pp.1,3)
SHA	512	160	(National Institute of Standards and Technology, 2002, p.3)
DES	64	128	(National Institute of Standards and Technology, 1999, p.8)
SHA2	512/1024	128-512	(National Institute of Standards and Technology, 2002, p.3)
RIPEMD	512	128	(Keromytis & Provos, 2000, p.2)
AES	128	32-128	(Dworkin, 2007, pp.7,9)

Table 3.4: Overview over AH hash block and output sizes

As with symmetric encryption, there are different modes of how to apply cryptographic algorithms to generate MACs. Standard IPsec authentication modes use *XCBC* and HMAC (Manral, 2007, pp.4-5). The Extended Cipher Block Chaining (XCBC) rests on the CBC mode (see section 3.5.2). To authenticate instead of encrypt a message, the CBC mode splits the message of a multiple of the cryptographic algorithm block length message into blocks and encrypts the first. On all subsequent blocks, it performs an XOR operation with its encrypted predecessor and encrypts the result until all blocks are processed. The final result is the MAC. The disadvantage of this method is that it is insecure of messages of varying length. The *XCBC* mode extends this by using two additional keys which are n =block length long, each of which is applied via XOR on the last plaintext block before the encryption: one if the message length is a multiple of the algorithm block length, the other if not. In the latter case, the last block is additionally padded with a single 1 and the number of zeros needed to fill the block up to the algorithm block size. Listing 3.3 shows a pseudo code of the algorithm. (Black & Rogaway, 2000, p.201) In this listing, E is the encryption function, K are the keys, M is the message, P is a plaintext variable for the padded message and C is the ciphertext.

```
Algorithm XCBC E_{K1},K_2,K_3(M)
if M ∈ (∑n)+
        then K ← K_2, and P ← M
        else K ← K_3, and P ← M||10^i , where i ← n − 1 − |M| mod n
Let P = P_1...P_m , where |P_1| = ... = |P_m| = n
C_0 ← 0^n
for i ← 1 to m − 1 do
        C_i ← E_{K1}(Pi ⊕ C_{i−1})
return E_{K1}(P_m ⊕ C_{m−1} ⊕ K)
```

[17] DES is also defined for AH, but here applies the same thing as for RC4 in ESP (see footnote 15)

Listing 3.3: XCBC Pseudo Code

Whereas a *XCBC* uses a symmetric cipher algorithm, there is also an approach which uses one-way hash (such as Message Digest (MD) or Secure Hash Algorithm (SHA)) functions to authenticate messages. This approach is called HMAC and concatenates two hash functions. As input for the first hash serves the message and a key XORed with a constant called Inner Padding (ipad). The second function gets the result from the first and the same key XORed with a second constant called Outer Padding (opad) as input. The output from the second hash is the HMAC. The IETF defines the constants as follows:

- ipad: *0x36* repeated block length of the hash function times
- opad: *0x5C* repeated block length of the hash function times

Equation 3.6 depicts this function with hash function H and key K, where $\oplus$ indicates an XOR operation and $||$ indicates a concatenation. (Krawczyk, Bellare, & Canetti, 1997, p.3)

$$H(K \oplus opad||H(K \oplus ipad||message)) \tag{3.6}$$

Apart from the two described ones, currently defined also use the Galois Message Authentication Code (GMAC) mode (Internet Assigned Numbers Authority, 2012a). This mode of operation is a authentication-only version of GCM (Dworkin, 2007, p.2).

4 Basics of Used Tools

This chapter describes the Austrian Institute of Technology (AIT) Quantum Key Distribution (QKD) system and its principal architecture, as well as additional tools used in this thesis.

4.1 Description of the used Quantum Key Distribution System

The AIT QKD solution uses the *Bennett/Brassard/Mermin 1992* (BBM92) QKD scheme (see section 2.4.2), which is a modified version of *Bennett/Brassard 1984* (BB84) for entangled states. Due to the entanglement, the key is generated on both peers simultaneously. The solution then uses a sifting phase (see section 2.5.1), error correction with the *CASCADE* protocol (see section 2.5.2.3) and privacy amplification (see section 2.5.3) to complete the QKD process. (Treiber et al., 2009, pp.5-6) The latter parts are the task of the AIT QKD software, an extensible, hardware-agnostic and open QKD system. (Maurhart, 2013, pp.9-11) Section 4.5 gives an overview of the reference system specifications and used libraries (apart from GCC and *Make*), while figure 4.1 depicts the software architecture.

It is also noteworthy that with this setup, the randomness of the quantum-derived key is secured by the measurement itself (Poppe et al., 2004). Universal_2 hash class functions (see section 2.1.3.1) provide the authentication for the QKD system.(Treiber et al., 2009, p.6)

4.2 Racoon and ipsectools

The Linux Internet Protocol security (IPsec) stack derives from the *USAGI* project (which ported the Berkeley Software Distribution (BSD) IPsec stack named *KAME* to Linux) and was incorporated into the kernel in version 2.6. (Roy, 2004) It uses the Transform (XFRM) configuration architecture and the *PF_KEY* interface to manage the Security Policy Database (SPD) and Security Association Database (SAD) entries. (Kanda, Miyazawa, & Esaki, 2004, p.159) *PF_KEY* is an Application Programmer Interface (API) to enable key management applications to communicate with key engines (for example the SAD) (McDonald, Metz, & Phan, 1998, p.1).

Due to its heritage, the *PF_KEY* interface is compatible with the one of *KAME*. Therefore the *setkey* utility can be used to manipulate the SPD and SAD and the *Racoon* demon for Internet Key Exchange (IKE). (Kanda et al., 2004, p.161) With the library *ipsectools*[1], a porting of these BSD tools is available for Linux.

[1]Please see `http://ipsec-tools.sourceforge.net/` for the project's website

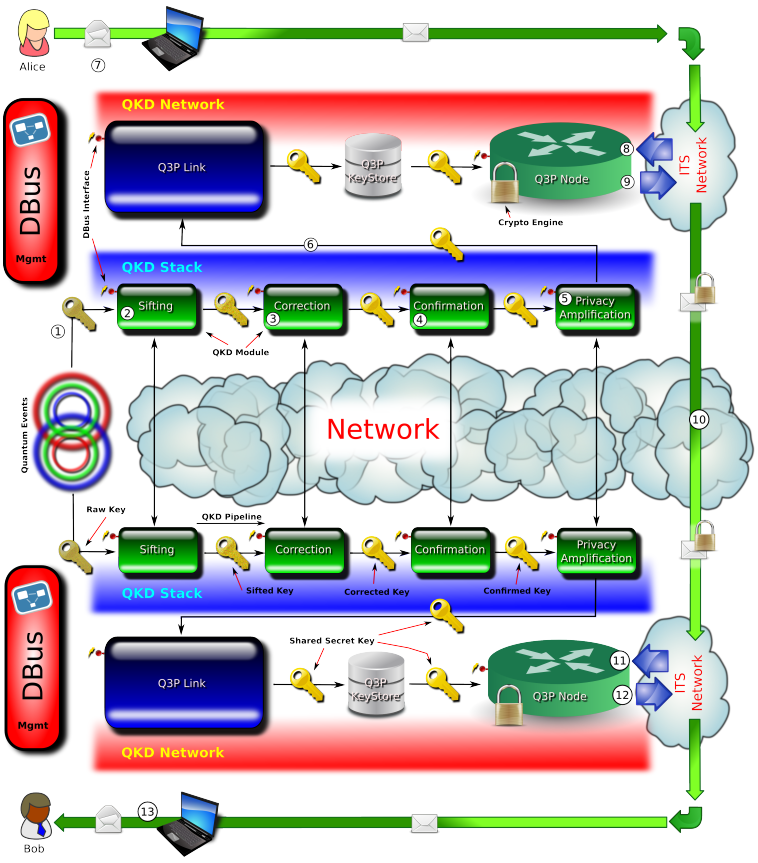


Figure 4.1: AIT QKD software architecture by Maurhart (2013, p.10)

The simplest way to establish an IPsec connection is to use *setkey* with manual keying. The *setkey* syntax uses the configuration command *spdadd* to add entries into the SPD and *add* to add entries into the SAD. The *setkey* command accepts these configuration commands within a file (invoked by *setkey -f <file>*). The *spdadd* command accepts the following parameters: *source ip address range, destination ip address range, upper layer protocol, and policy*. The latter is preluded by a *-P* and consists of the direction (*in/out*), action (*discard, none, ipsec*) and, if *ipsec*, a string specifying the protocol (*esp/ah*), mode (*transport/tunnel*) and the level (*default,use, require, unique*). The *add* command contains again the *source ip address range* and *destination ip address ranges*, the protocol (*esp/ah*), a Security Parameters Index (SPI) number and the cryptographic algorithm (for example *des-cbc*) and key (for example *0xAABBCCDDEEFF0011* preceded by *-E* Listing 4.1 shows a sample configuration for *setkey*. To review the configuration the *setkey* command switches *-D* and *-PD* show the SAD and SPD while *-F* and *-PF* flush them. (BSD, 2004)

```
spdadd 10.0.11.0 10.0.11.33 any -P out ipsec esp/transport//↩
   require;
add 10.0.11.0 10.0.11.33  esp 24501 -E des-cbc 0↩
   xAABBCCDDEEFF0011;
spdadd 10.0.11.33 10.0.11.0 any -P out ipsec esp/transport//↩
   require;
```

```
add 10.0.11.33 10.0.11.0  esp 24501 -E des-cbc 0↩
   xAABBCCDDEEFF0022;
```

Listing 4.1: Example configuration for setkey

This thesis used the *setkey* command with configuration files similar to listing 4.1 for proof of concept tests regarding rapidly changing IPsec keys.

4.3 The Linux GCC Compiler

The GNU Compiler Collection (GCC) is a compiler suite currently containing support for *C, C++, Objective-C, Objective-C++, Java, Fortran, Ada* and *Go*. (R. Stallman et al., 2013, p.3) The source of GNU Compiler Collection (GCC) is GNU's Not Unix (GNU), which is a *Unix*-like operating system. GNU variants using the *Linux* kernel are widespread. These *GNU/Linux* systems are commonly abbreviated to *Linux*. (R. Stallman et al., 2013, p.709) Therefore, the GCC suite is broadly available on *GNU/Linux* systems. When using GCC for compiling *C++* source code, the suite recognizes the appropriate file extensions[2] and automatically treats them as *C++* files. As it does not automatically include the *C++* library, there is a program called *g++* which calls GCC with the appropriate linking options. (R. Stallman et al., 2013, p.29) The compiler suite complies to the *C++* standardization ISO/IEC 14822 from the International Organization for Standardization (ISO) and International Electrotechnical Commission (IEC) in versions of 1998 and 2003 (R. Stallman et al., 2013, p.321).

4.4 Linux Make

Make is a tool to manage builds. This means that it takes on the task of (re-)compiling present source codes of large software projects in the desired sequence and linking them together into an executable program. For that purpose, it reads a *makefile*, which specifies the build process. *Make* may be used to steer any update process. (R. M. Stallman, McGrath, & Smith, 2010, p.1-3) The *make* process recompiles and substitutes files according to the makefile rules only if they do not exist or the current files are of newer version than the prior existing (R. M. Stallman et al., 2010, p.5).

4.5 Additional Libraries

The AIT QKD software runs on *Linux* platforms, the reference system is a *Debian 7.0 Wheezy amd64* installation (Maurhart, 2013, p.12). Apart from the aforementioned, the AIT QKD software uses the following libraries (Maurhart, 2013, p.14):

- gcc and g++;

[2] .C, .cc, .cpp, .CPP, .c++, .cp or .cxx for C++ code files and .hh, .hpp, .H or .tcc for C++ header files

- boost;
- OpenSSL;
- UUID;
- CMake;
- GMP;
- Zero Message Queue (0MQ);
- Qt4;
- Qwt5;
- Doxygen.

5 Solution Concept

This chapter describes the principal system architecture of the presented solution and the context it resides in. Therefore, it shows by what means the QKDIPsec system keeps up Quantum Key Distribution (QKD)-fed Internet Protocol security (IPsec) connections and the keys synchronized between two peers using the system. Considerations regarding key lengths and change frequencies and about authentication and integrity conclude this chapter. Explanations of the software concepts, Application Programmer Interface (API) and other technical parts follow in chapters 6 and 7, respectively. For explanations about the behaviour of this solution regarding to the QKD system please refer to section 7.5.

5.1 Previous Approaches

This thesis is aware of some previously developed methods to combine QKD with IPsec. All of them work in conjunction with the Internet Key Exchange (IKE) or the underlying Internet Security Association and Key Management Protocol (ISAKMP) protocol. They introduce a supplement for QKD parameters or combine IKE-derived and QKD-derived keys. Opposed to this, the present thesis tries to use an approach omitting IKE and following the pivotal idea that there is no need for that protocol to exchange keys, for that is the task of QKD. The key feed from QKD therefore provides the material for manual keying in this solution, all that is left is to keep those keys synchronous. For this task, this thesis provides a more slender approach (see section 5.4). Furthermore, some of the previous approaches operate at a substantially lower speed than the key change (see section 8.5) presented in this thesis or use One-Time Pad (OTP) (see section 2.1.2) limiting the data rate to the QKD key rate. This rate today lies at maximum arround twelve kilobits per second (see section 5.5.1).

5.1.1 DARPA Quantum Network

The *DARPA Quantum Network* it the world's first end-to-end QKD network and also offers an IPsec solution. The solution uses a Berkeley Software Distribution (BSD) system and alters the kernel as well as the IKE daemon for its use with QKD. It uses two kinds of encryption. On the one hand it uses quantum keys as *seeds* to secure the exchange of classical keys (with continual refreshment), on the other for direct application with OTP. The *seed* method changes the key (with Advanced Encryption Standard (AES)) around every minute. (Elliott, Pearson, & Troxel, 2003, pp.234-235) OTP naturally lowers the data rate to the QKD key rate.

5.1.2 Secure Quantum Key Exchange Internet Protocol

The Secure Quantum Key Exchange Internet Protocol (SEQKEIP) is no supplement but a replacement for IKE. It does, however, build on ISAKMP and use the same Main Mode for phase one and Quick Mode for phase two as IKE does. The difference to IKE is that it has an additional phase zero in which it uses the *quantum mode* to exchange a shared key and it also does not exchange *Diffie/Hellman* (DH) groups but instead use the shared key. It further gives the opportunity to use OTP as a cipher. Other ciphers are only used as transition methods until switched to OTP. (Sfaxi, Ghernaouti-Hélie, Ribordy, & Gay, 2005, pp.6-8) Therefore, this solution has a relatively low data throughput.

5.1.3 QIKE

QIKE is a joint effort of Siemens, the Austrian Institute of Technology (AIT) - at that time named Austrian Research Centers (ARC) - and the Graz University of Technology to provide key management for IPsec systems and has an implemented prototype on a security gateway. Its main objective is to provide efficient IPsec key management with quantum keys. (Neppach et al., 2008, p.177) To accomplish this, the solution supplements IKE with a modified own version called *QIKE*. Its task is to negotiate Security Associations (SAs) and parameters for QKD key streams. (Neppach et al., 2008, p.180) Similar to IKE, it consists of two phases, with the first establishing an enhanced ISAKMP SA, extended by a QKD key rate and an application ID. The second phase generates an IPsec SA in *quick mode*, where multiple SAs prevent the IPsec engine from having to wait for new keying material. An expired SA does not have to be renegotiated, but instead the peer whose SA has expired sends a notification to its counterpart, instructing it to delete the concerned SA. QIKE uses PF_KEY to handle the Security Association Database (SAD). (Neppach et al., 2008, p.182)

In contrast to QIKE, the approach presented in this Thesis provides no supplement for IKE. It does not use handshakes similar to IKE, instead it lets the AIT QKD software choose which algorithms and parameters it wants to use and uses its own concept for key synchronization (see section 5.4). Furthermore it provides a pure software solution instead of specialized hardware.

5.1.4 MagiQ Technologies

MagiQ Technologies is a research and development company, that also provides an IPsec-capable Virtual Private Network (VPN) gateway. (MagiQ Technologies, 2007) The company holds a patent on integrating QKD with IPsec in which they claim a method for joining classical and quantum keys for the use with IPsec SAs. In their approach, they use classic IKE to establish a classic SA and then apply different quantum keys to create different quantum key-secured SAs which are ready to use (*final SAs*). The patent also includes variations and implementation details of this method, like using Exclusive Or (XOR) to combine classic and QKD-derived keys, the number the provided SAs, the introduction of a *maximum allowable Internet Protocol (IP) packet delay time*, encoding details into the Security Parameters Index (SPI) and others. (Berzanskis, Hakkarainen, Lee, & Hussain, 2009)

QKDIPsec, on the contrary, does not use IKE or any combination of IKE with QKD-derived keys to secure traffic, but relies on QKD keys alone.

5.1.5 Aqua

The Advancement in Quantum Architecture (AQUA) project[1] has produced an Internet Draft to enhance IPsec with QKD, which originates from a Bachelor's Thesis by Nagayama (2010). It suggests to supplement IKE by replacing the DH key exchange with QKD and defines appropriate IKE payloads. One of these payloads is intended for the exchange of a key identifier, which comes from the respective QKD engine on both sides of a QKD-enhanced IPsec connection. The second one is to determine a fallback method if there is no QKD key available. (Nagayama & Van Meter, 2009, pp.5-9) It defines three possible methods:

- Wait until a new key has been generated;
- Use *Diffie/Hellman* (DH) key exchange (use a classically generated instead of a QKD key);
- Continue with the most recent key (no new key is generated).

The AQUA approach does not only differ from this thesis in that it uses IKE as a principal key exchange method and makes supplements to use quantum-derived keys, but also that the draft does not give an apparent notation to apply these keys *rapidly* on an IPsec system. The draft therefore does not lower the number of data bits per key bit to aggravate cryptanalysis (see section 5.5.1).

5.1.6 SwissQuantum

SwissQuantum[2] is a project of various Swiss universities, research and industrial organizations to create a testing environment for QKD. Amongst other research, this project also created a solution for IPsec. This solution resembles the current approach by using a Linux distribution (*CentOS*) and *ipsectools* (which, like the present solution uses *Netlink* - see chapter 6), but differs by, like the other known previous approaches, supplementing IKE. (Swiss Quantum, 2009) It also operates at relatively low key change rates (one key per minute) compared to the ones in this thesis (Stucki et al., 2011, p.21).

5.2 System Context

The identified main parts surrounding this solution are the AIT QKD software, the Linux kernel IPsec subsystem (including the SA and Security Policy (SP) databases) and the network interface. The QKD software controls the QKDIPsec module (which means that it gives the

[1] `aqua.sfc.wide.ad.jp`
[2] `http://swissquantum.idquantique.com`

commands and data to create an IPsec connection with a peer) and provides the quantum keys. The Linux IPsec subsystem's databases (Security Policy Database (SPD) and SAD) contain the information the kernel uses to decide which traffic to protect with IPsec and which parameters and keys to use. The QKDIPsec module feeds these databases with keys from the QKD software and further parameters. Based on this, the Kernel subsequently encrypts the user data flowing over the network interface to the peer system. The QKDIPsec module provides the interface with parameters such as a virtual IP address and a route to the peer system and uses it for key synchronization with the peer. Figure 5.1 illustrates the relations between the user and the system components.

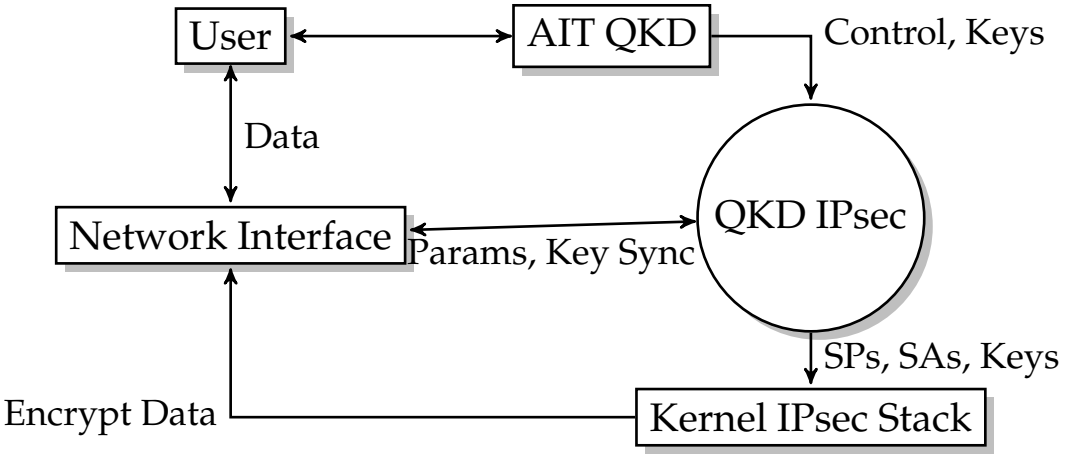


Figure 5.1: Context Diagram of the Solution

5.3 Principle Architecture

To protect network data traffic via QKD-enhanced IPsec, this concept uses rapid rekeying. The security level depends on the frequency of the key change and, accordingly, time and byte SA limits (for recommendations regarding these items see section 8.7). Currently, this solution changes the keys on a time basis (tested with up to 50 key changes per second and various key lengths - see section 8.5). To prevent waste of QKD key material, it has the opportunity to change its clock rate during operation. Therefore a system utilizing QKDIPsec might constantly adjust the key change rate to preserve quantum key economy. In future, the solution might also have the opportunity to operate based on data rates as well. In order to practically implement rapid rekeying on a compter system a software module needs to fulfill three major tasks:

- acquire QKD keys;
- feed them into the IPsec system;
- keep those keys synchronized between the communicating peers.

In order to fulfill the requirement to accomplish these main tasks, the present solution contains three main parts, each specializing in one task:

- an IPsec connection manager (controller and key synchronization);
- multiple kernel managers (IPsec and network system management);
- multiple key managers (QKD key management).

The first part manages the establishment and termination of a QKDIPsec connection and has the task to keep the keys of the communicating peer systems synchronous. The connection itself consists on the one hand of a data channel consisting of an IPsec Encapsulating security Payload (ESP) tunnel. This tunnel encrypts all traffic flowing to a virtual IP address on the remote side, routed through a virtual address on the local side. To manage the key synchronization, on the other hand, it uses a control channel consisting of a Transmission Control Protocol (TCP) connection authenticated with an IPsec Authentication Header (AH) (see section 5.4.3). For each of the channels, it uses two kernel IPsec managers responsible for handling the IPsec SPD and SAD entries for one direction (inbound and outbound). Apart from creating and deleting IPsec connections on behalf of the connection manager, they are charged with rapidly changing the encryption key with new key material from the AIT QKD software. Providing these keys is the task of the key manager. It therefore is the interface between the main QKD software and the IPsec module presented in this thesis. Each of the kernel IPsec managers possesses its own key manager which derives keys from the QKD engine based on an application identifier, which each key manager uniquely possesses. Each key manager matches (via application identifier) with a counterpart in the opposite direction of the same channel (data or control) on the peer system, therefore creating corresponding keys for the four unidirectional IPsec links on both peers. Additionally, the IPsec connection manager uses a kernel network manager to add and remove network interface addresses and routes. The kernel and key managers provide therefore interfaces to systems outside

of the module (IPsec and network through the kernel managers and QKD through the key managers, respectively), while the IPsec connection manager provides the control unit and the key synchronization, which section 5.4 discusses in more detail. In summary, the solution uses two IPsec channels and a quantum channel for the key distribution. The AIT QKD software uses an extra Quantum Point-to-Point Protocol (Q3P) channel for OTP-secured connections. Figure 5.2 depicts this channel architecture.

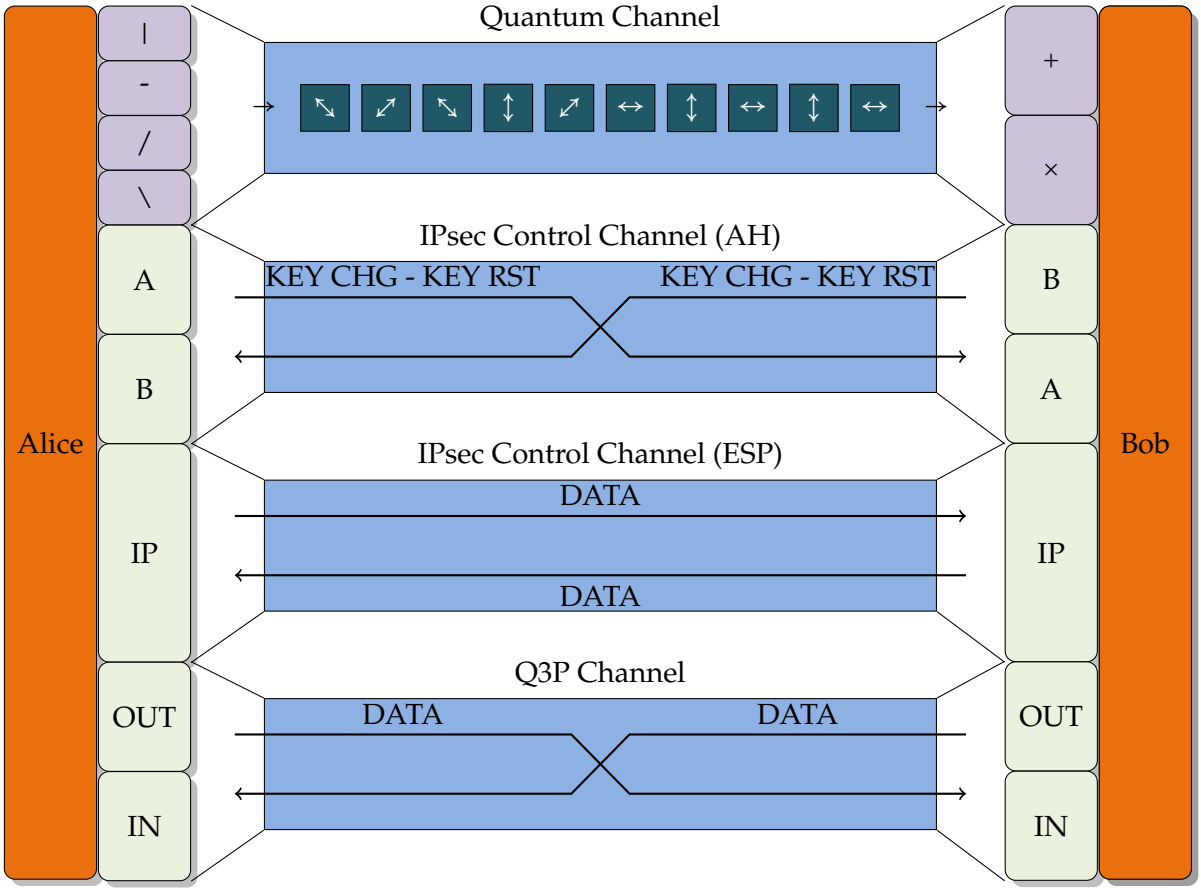


Figure 5.2: Channel Architecture of QKDIPsec

While the quantum channel is an own dedicated line, the other channels run over the Internet. For more information about Q3P, please refer to Maurhart (2013) and Maurhart (2010). For a closer description of the control channel, please refer to section 5.4.3.

5.4 Key Synchronization - Rapid Rekeying Protocol

The main aim of the rapid rekeying (key synchronization) protocol is robustness and swiftness. Therefore, even as it uses TCP connections with according recovery mechanisms, its design allows to ignore lost packets, for recovery might slow down the key change process too much. In difference to previous approaches (see section 5.1), this solution does not use any IKE-derived protocol to exchange keys as this has already happened through QKD at that time (and still is continuing to happen, for QKD delivers a key stream). Since QKD needs a

dedicated quantum channel (which has physically to be point-to-point - see the introduction of chapter 2), it can be assumed that both peer systems know each other and in further consequence not only the IP address but also other IPsec parameters like cryptographic algorithms are preshared[3]. As this solution is part of the AIT QKD software, in this case this software may choose these parameters. Therefore, everything needed to establish an IPsec connection is already present on both peers at the time of establishing an IPsec connection. The only thing left to do is to fulfill the need of keeping those items synchronous, particularly the IPsec SPIs and keys which are the main mutable parts during a connection. QKD keys do not come continuously but rather in bursts, therefore key material might not always be available. In that case, the solution suspends the communication on the data preventing data to leak through an unsafe channel. As soon as key material is available, it releases the channel again.

5.4.1 Security Parameters Index Generation

While the keys come from the QKD engine, the module calculates the SPIs by itself (except for the first one). These are no secret values, every IPsec packet contains its corresponding SPI as a plaintext value in its header. This is the reason why the SPI calculation is no high security task, in fact a simple counter might suffice. Nevertheless, a counter might allow to draw conclusions about how long and at which data and key change rates an IPsec connection is running without having to record all of the packets. To make that non-obvious, the generation of a new SPI uses a *salted* Secure Hash Algorithm (SHA)-1 algorithm derivate of the old one. The salt (s) and the first SPI value come from QKD which is not very costly in terms of quantum key consumption for this derivation occurs only on initialization. Equation 5.1 shows the SPI generation.

$$SPI_i = \lfloor \frac{SHA1(SPI_{i-1} \oplus s)}{2^{128}} \rfloor \tag{5.1}$$

While the SPI is a value of 32 bits (see section 3.1.2), the output of the SHA-1 algorithm is 160 bits (Eastlake & Jones, 2001, p.2). This algorithm takes leftmost 32 bits of the output as new SPI value. The interference with different other applications on the host platform in terms of SPI overlapping is not regarded as problematic as the SPI only partially identifies an IPsec connection (see section 3.1.2). If all of the other constituting parts match as well, the SA sensibly belongs to the very same connection. Figure 5.3 depicts the generation of an SPI.

5.4.2 Master/Slave Key Advancement

To ensure synchronous key and SPI advancement[4], the present solution uses a master/slave relation between the peers to ensure perpetually matching IPsec SAs. This approach basically uses the directional characteristic of SAs (see section 3.1.2) to constitute said relationship. As both IPsec links that are necessary for a bidirectionally secured communication channel are

[3] Although this could also be automated - similar to IKE - with initial negotiation messages.

[4] Due to the key derives from QKD with the help of *getNextKey* methods, the term key advancement seems appropriate.

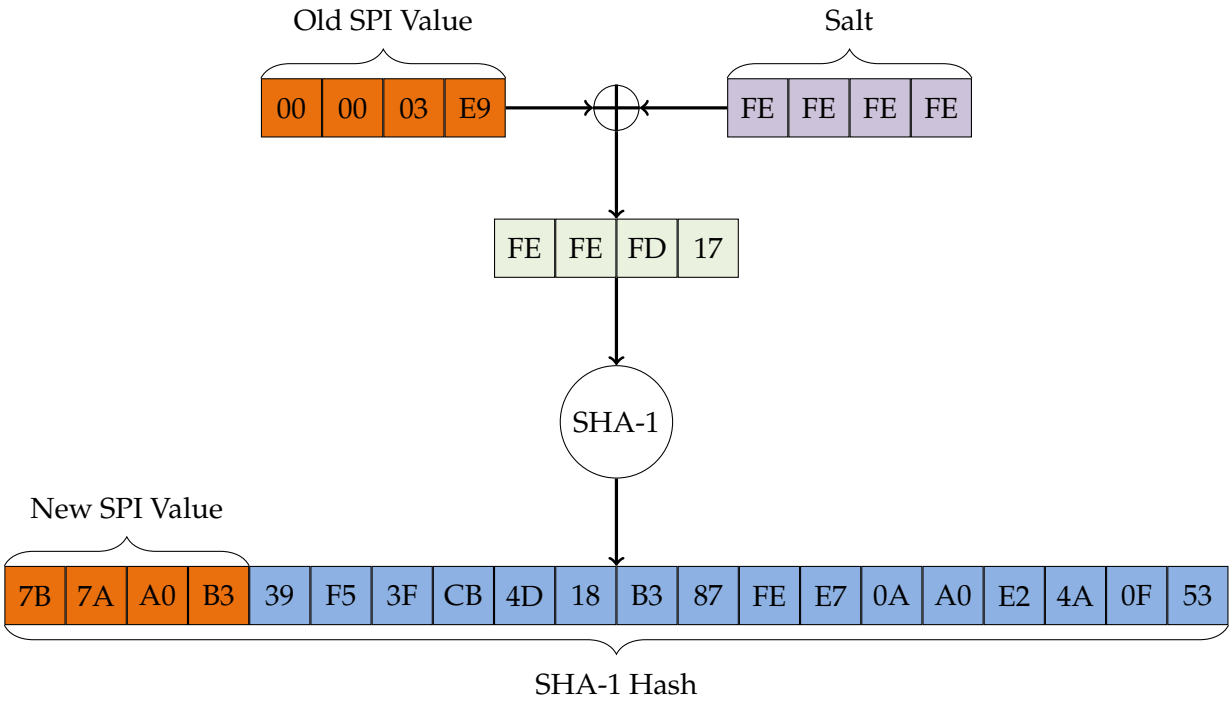


Figure 5.3: SPI Generation Procedure

completely independent, each peer takes responsibility for one of them. As there is a control channel for each data channel, the former serves as communication line for key changing an reset messages for both the latter and itself. A receiver might have multiple SAs installed for an IPsec connection and determine by the SPI of an IPsec packet which SA to use (Kent & Seo, 2005, p.10,pp.13-14). QKDIPsec uses this opportunity to mitigate packet loss, late packets and other performance decreasing issues during the key change process. A sender on the other hand, might install multiple, but logically use only one single SAs per packet as there is only one SPI per packet. Linux, for instance uses Last In First Out (LIFO) queuing. This means that it always uses the newest installed matching SA to perform outbound IPsec protection. (Neppach et al., 2008, p.6) Due to this selection process, the SPIs and their respective keys are strongly bound as they, as stated above, form the mutable part over the duration of the connection. The rest of the parameters that constitute an SA remains constant (with the possible exception of SA expiration limits). This naturally applies to the data and control channel alike.

This solution installs a new key by adding a new SA and deleting an old one. This method provides full control over which SAs are actually in the system. However, conducted tests showed that using SAs with expiration timer could yield up to five times higher key install rates[5] (see section 8.2). For the time being, this does not seem necessary on state of the art machines, but might its have use on embedded systems. A drawback of this method is that, as SA timeouts in Linux may only be set in seconds granularity, there may be a number of deprecated SAs in the system at any time given. This number linearly increases with the

[5]There is an experimental version of this solution, that principally supports this method, but is not working yet. This method might incorporated in the future if demand is given.

key frequency. Furthermore, this prevents using SAs without timeout, as they would not disappear any more.

To maintain the SAs synchronous, each peer assumes the master role for its outbound IPsec links. While the kernel automatically encrypts and decrypts the data channel, the module periodically changes the key on the master side, but also sends a key change notification (*DT_CHG_REQ*) containing the new SPI to its peer beforehand. On reception of this notification, the slave reacts by installing a new SA on its own part (the slightly different control channel mechanism is explained later in this section). The slave node also sends an acknowledgement (*DT_CHG_ACK*) containing the same received SPI. By the current design it serves only as a keepalive packet for dead peer detection (see section 5.4.5), but might serve other purposes in future usage. Figure 5.4 displays an example key change sequence.

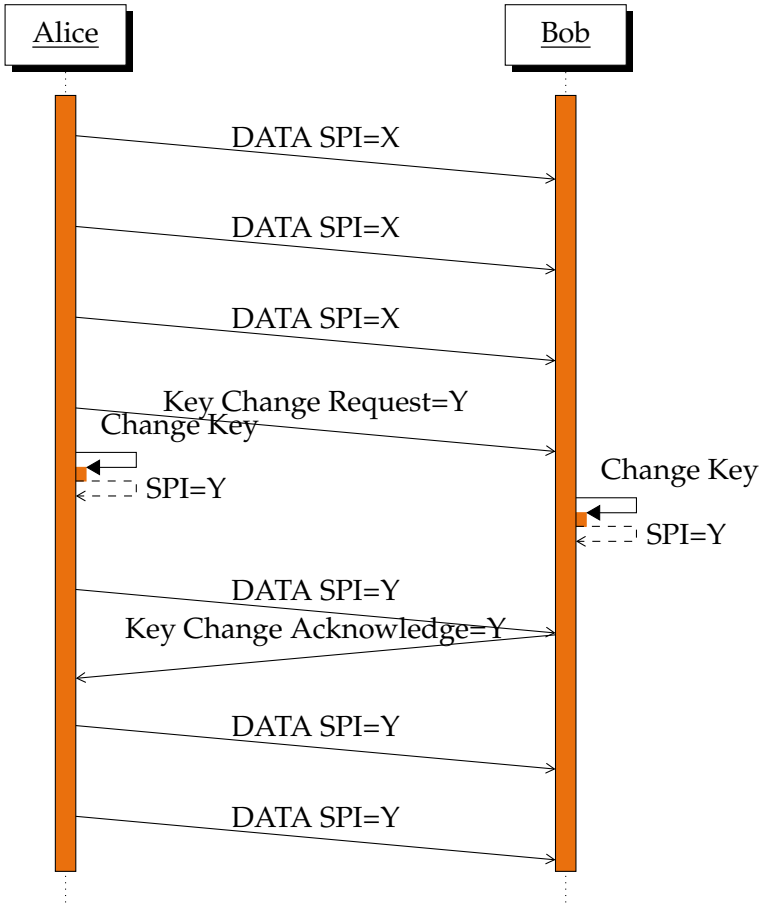


Figure 5.4: Key Change Sequence Diagram

Due to the data IPsec transformation happens in the Kernel (which is completely outside the control sphere of this module) the relation between key changing and actual data encryption is an asynchronous one. This means that data secured by a more recent SA on the sender side might match an older SA on the receiver side, causing the receiver to drop the respective packets. The cause is that network packets may not necessarily arrive the same order as they are sent (Braden, 1989, p.10).

To prevent this condition, the receiver installs a certain number of SAs in advance using SPIs and keys not yet installed on the sender side. This procedure is possible because, as mentioned above, using QKD eliminates the need to care for key exchange. The only requirement in this context is the synchronicity of the quantum keys which is in the responsibility of the QKD system. This module supports the QKD system by arranging key collection as simultaneously as possible by adequate queueing. As a result, the sender, despite being only able to install only one acting sending SA at a time, also requests keys and calculates SPIs beforehand and stores them until they come to actual use[6]. So instead of generating an SPI/key pair and just using this actual one, the slave installs it in inbound direction and deletes the eldest, while the master puts it in a queue and replaces the actually installed SA with the eldest pair from the queue. To cover the possibility of late packets arriving with an old SPI, the inbound queue (containing all inbound and therefore slave SPIs) also holds deprecated SPIs and leaves them in the system for an additional timespan. The present solution stipulates the number of past SPIs to be equal to future ones[7]. Therefore the outbound queue is half the size of the inbound, for it only stores future SPIs. The queue size is configurable and a sensible value depends on the key change rate. Figure 5.5 illustrates the SPI queuing. The notations with arrows demonstrate the actions to be taken for a new SA installation. Please note that on the masters' side is an extra queue for outbound keys. On the outbound side only the current SA is in the SAD, thus the fetched forward keys must be stored. On the inbound side, no key queuing is necessary because the forward keys are already installed and do not need to be remembered separately.

5.4.3 The Control Channel

As stated above, the control channel, responsible for all key change communication, consists of an AH-secured connection. This communication occurs over a simple, type-value paired data structure called key change message. So far, the following key change message types are declared:

- NO_MSG ;
- DT_CHG_REQ;
- DT_CHG_ACK;
- DT_CHG_FAI;
- MA_RST_REQ;
- MA_RST_ACK;
- MA_RST_RAK;

[6] If the IPsec system were using First In First Out (FIFO) queueing the sending SAs could be installed immediately and key change would happen by deleting the last SA.

[7] Due to the large possible SAD size, performance is not an issue here. On embedded devices, however, it may well be. As the probability of getting out of synchronization with the key is higher than the probability and impact of late packets, a more economic version might choose the number of future SAs higher than the past ones.

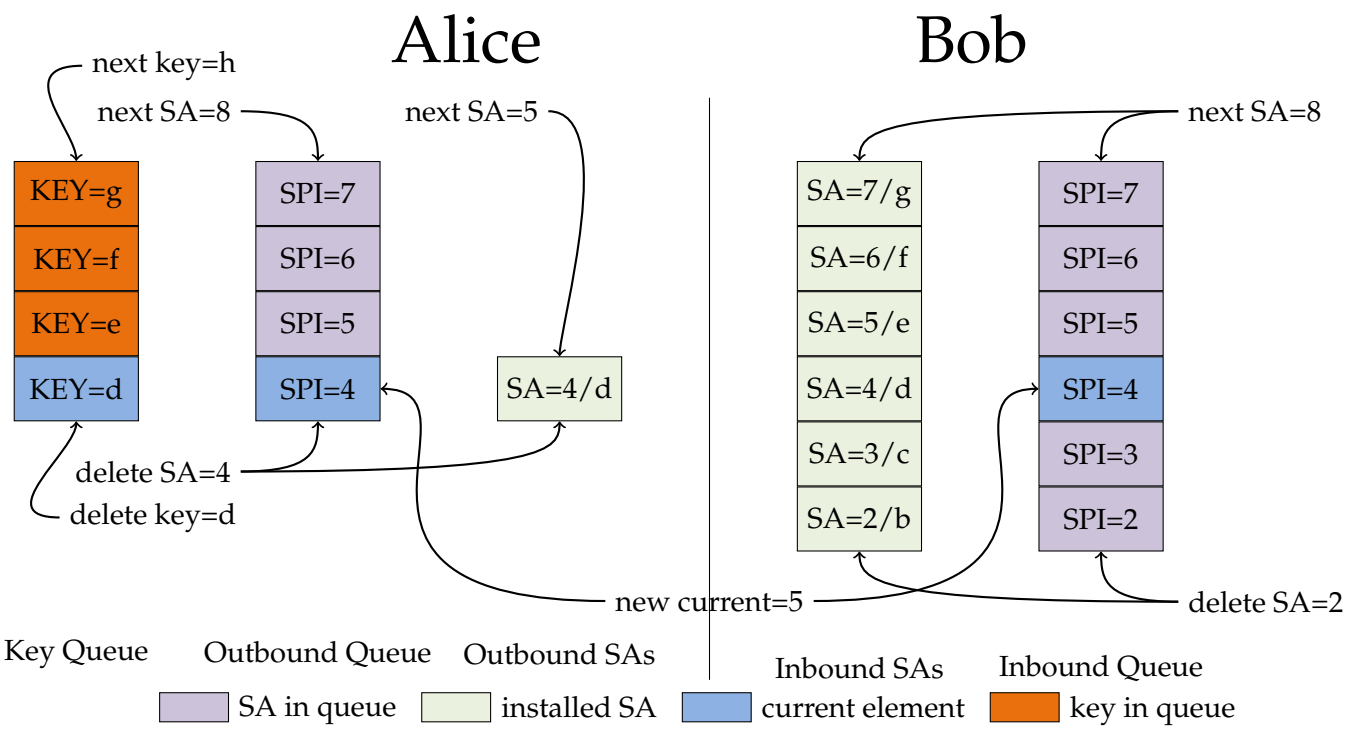


Figure 5.5: Queuing of future and past SAs

- SL_RST_REQ;
- SL_RST_ACK;
- SL_RST_RAK;
- CT_CHG_REQ;
- CT_CHG_ACK;
- CT_CHG_RAK.

While the first one is a dummy type, the types starting with *DT* are explained in section 5.4.2[8], *MA* and *SL* in section 5.4.4 and *CT* below in this section. These messages contain usually the concerning SPI value or a simple counter value. The transmitted data is the reason for the control channel being AH-secured. It is crucial that the originator of a key change message is verified, for a forged message might cause mismatches in the SPI/key assignment. At the same time, the transmitted data does not contain any secret. In IPsec transport mode, which has less overhead compared to the tunnel mode, the AH provides better authentication capabilities for it also verifies (almost) the whole packet including the sender's address (see chapter 3.3.1).

[8]Except for *DT_CHG_FAI* which is reserved for future use.

To ensure high security also in this channel, the SPIs and keys are also changed periodically. The control channel key change mechanism differs in that it resembles a three way handshake like the one proposed by Postel (1981, pp.27-28) instead of a single message[9]. For this purpose, it uses three messages (CT_CHG_REQ, CT_CHG_ACK and CT_CHG_RAK). The first one sent from master to slave induces the process. The subsequent answer notifies the master of the slave's awareness and causes it to perform a key change after the sending of the third message. This message eventually causes the slave to perform a key change.

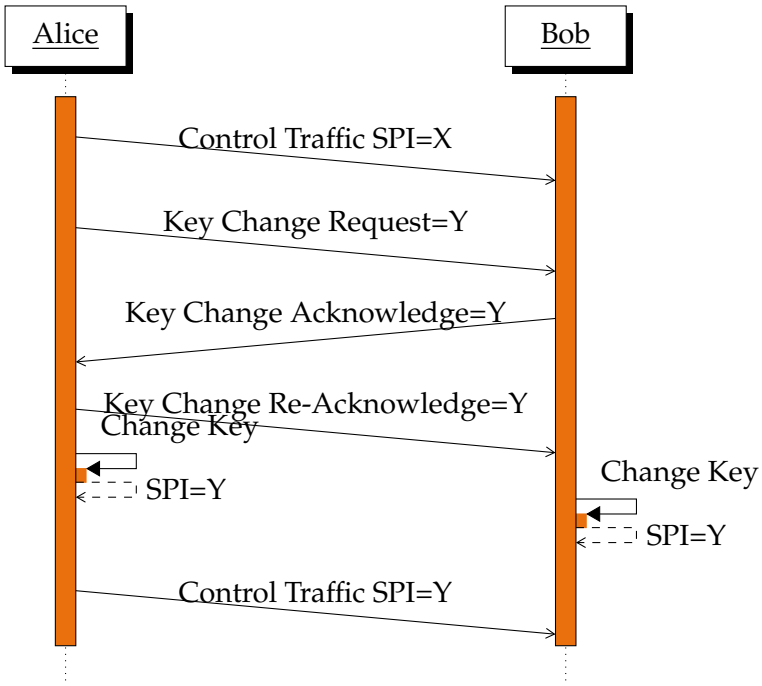


Figure 5.6: Control Key Change Procedure

A failed initiation of this process would cause the repetition of these messages until reaching a timeout. After that, the protocol would then stop and prevent further communication, as the communication is no longer secure because of too much key reuse. Key mismatch on this channel would sever the protocol communication, which means the whole key change mechanism (including the one for the data channel) ceases to function. Also, due to the presumable much lower data rate on the control channel, there is a much lower key change rate needed to maintain the same secrecy level in terms of data bits per key bit. The data rate itself and, dependently, the control channel key rate also depends on the data key change rate (section 5.5.1 contains more information about key rates and lengths). The importance of the control channel and the lower key change rate are the main reasons why it has a more ponderous key change mechanism than the data channel. Additionally, delays on this channel do not directly impact the data communication. Therefore, choosing this method adds to the robustness of the control channel while not having a negative effect on the data channel. Apart from that, the control channel also uses the same queuing technique as the data channel, although, given the much lower key change rate, it uses independently configurable (presumably much smaller) queue sizes.

[9] As stated above, the data channel acknowledgement only serves keepalive purposes and does not influence the actual key change. The key change is therefore a single message process.

5.4.4 Missing Key Correction

Although queue size is important for late packets, it is even more so for the case that key change packets go missing. To make the protocol more robust against lost key change requests, the precalculation of SAs is the first measure. Packets encrypted (or authenticated) with an unnotified new SA still can be correctly deciphered (or verified) by the receiver. Nevertheless, missing key change messages could cause the SAs on both sides to get out of synchronization. This is the reason for the key change messages to contain the SPI. In a lossless environment, a key change message without any further information would be enough to ensure key synchronization, as all information is present on both sides. If, however, key change messages get lost along the way, the SPIs and keys do not match. This is mitigated by the queueing mechanism, but if more messages than stored in the queue get lost (over the complete time the communication channels remain established) communication stops. Therefore, every key change message contains the SPI of the current SA to install. The slave compares each received SPI value with the value it expects to receive (by looking it up in the inbound queue). If the value is not equal, it keeps looking forward[10] in the queue until the correct value is found. The slave then automatically knows the number of missed key changes and installs all of them up to the actually received one into queue and SAD as described in section 5.4.2. Every peer also maintains a queue with missed SPIs in order to cope with already installed late SAs and ignore the corresponding messages. If the received SPI cannot be found in the queue (for it is not sensible to indefinitely calculate future SPIs in hope the received one might match them), the slave triggers a reset process as explained in section 5.4.5. So still a received SPI that exceeds the number of SAs stored in the queue causes the system to reset, but with this mechanism the slave resynchronizes every time it gets an unexpected SA it is still able to recognize. Because of this, not the combined losses of a whole session must exceed the number of queued items to cause a reset, but a single loss period has to.

5.4.5 Key Synchronization Reset

Key synchronization resets actuate both of the two communication peers to initialize all of the components involved in the key synchronization. This means to erase and rebuild all inbound and outbound data channel and control channel SAs by rebuilding their respective queues and the SAD, as well as possibly resynchronize the QKD system. The reset procedure uses the same three-stepped scheme as control channel key changes (see section 5.4.3) do, which means that there is a request, an acknowledgment causing the sender to perform the reset and a re-acknowledgement causing the receiver to perform the reset. Resets can be master or slave resets, depending on the initiator of the reset. Figure 5.7 depicts both possible reset sequences.

There are three occasions on which a reset of the key change system happens:

- The initial reset;
- A master reset caused by failed dead peer detection;

[10] This means in the future SPI direction.

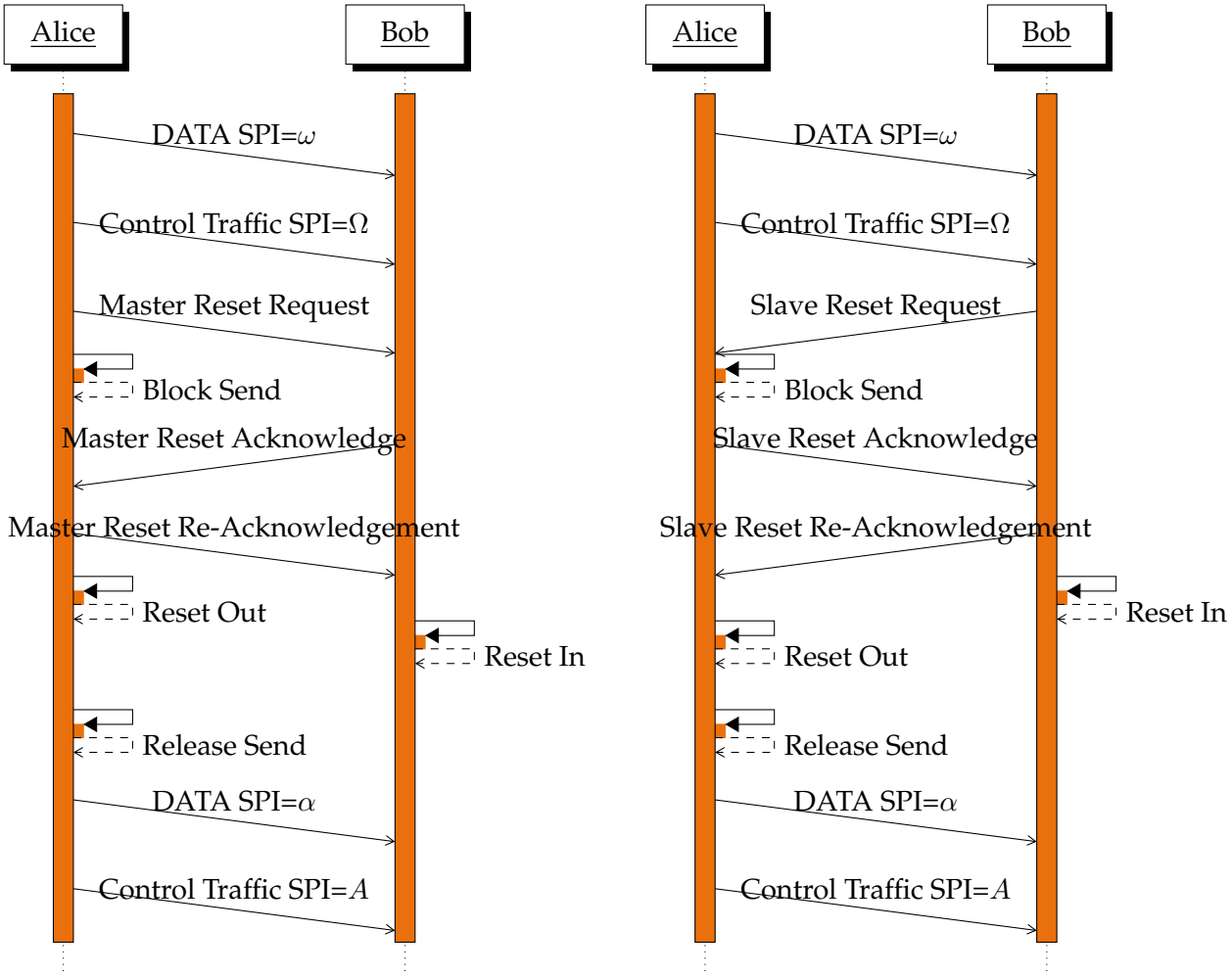


Figure 5.7: Sychronization System Reset Procedures

- A slave reset caused by too many missed SPIs (see section 5.4.4).

The first two cases are master-triggered resets and use the message types MA_RST_REQ, MA_RST_ACK and MA_RST_RAK. The initial reset occurs when the two peers successfully connect both (inbound and outbound) control channels[11]. The second occasion is when the master does not receive an acknowledgement for its key change messages for too long (see section 5.4.2). Every time the master sends a request, it increases a counter. If the counter exceeds a defined maximum value, the master triggers a reset (dead peer detection). The last case is a slave-triggered reset and uses the message types SL_RST_REQ, SL_RST_ACK and SL_RST_RAK. It follows the same scheme as a master reset and is triggered on the condition described at the end of section 5.4.4.

The separation in master and slave resets has the practical reason of assigning a received message to a data and control connection pairing and, thus, determining which of these is to reset (for both directions could be set independently). Due to the conditions leading to

[11]This requires an intially set *seed* control SA for each channel. The *seed* is QKD-derived and set on module startup.

sending a particular reset message, the type of that message automatically determines the corresponding direction. Table 5.1 gives an overview of which messages a peer sends and receives for which directed connection. As with control channel key changes, the system stops

	inbound connection	outbound connection
sending	MA_RST_ACK, SL_RST_REQ, SL_RST_RAK	MA_RST_REQ, MA_RST_RAK, SL_RST_ACK
receiving	MA_RST_REQ, MA_RST_RAK, SL_RST_ACK	MA_RST_ACK, SL_RST_REQ, SL_RST_RAK

Table 5.1: Overview over sent and received reset messages

on failure of a reset process. In that case the outbound communication is stopped by an SP that discards traffic into the data channel, preventing any secret to be sent onto a potentially insecure connection (the system also imposes this policy during a reset process). Applying a block on the sender side suffices due to the slave only acts on the master's behalf and to the unidirectional characteristics of IPsec. This occurs not only if the connection is lost between peers but also if one (or even both) system crashes, given that the SAs have a (time or byte) limit. In this case a alive peer blocks its outbound connection because of the failed dead peer detection, while on a crashed system the SAs time out if not replaced timely. Because the SP is still in place (for the crashed agent could not remove it) but lacks of corresponding SAs once they are timed out, the IPsec system also blocks communication for that connection. If this is undesired, the used SAs should not contain such a limit.

5.4.6 An Alternative Approach

Apart from the described approach, this thesis considered an alternative idea for keeping the keys synchronous, which it calls *monadic key exchange* after the Monadology by German philosopher Gottfried Wilhelm Leibniz. After Leibniz, Monads are principle entities in the universe which do not have (metaphorical) *windows* through which something may get inside or from the inside out. (Leibniz, 1847, p.12) As this key synchronization method functions without interaction between the two nodes the nodes resemble Monads. Basically all required parameters for key changes are already there through QKD, therefore any communication between the peers is out of the scope of the IPsec system. Through QKD, both nodes get the same key at the same time. Given this, if both peers change their keys periodically with the same frequency, the keys stay synchronous. Both peers may realize this synchronization by periodically generating SAs, with the sender switching to the active SPI. Deprecated SAs may time out automatically by a set *hard* expiration timer. This timer is a value of at least one second, due to the architecture of IPsec. As the key exchange rate is higher, there is always a certain amount of expired SAs in an active QKD-secured IPsec connection, which can mitigate problems with network latency and resent packets at the moment of a key change. This method, however, has some major disadvantages. Apart from the need for a precise time synchronization at the start, both peers need to adjust the frequency synchronously in fractions of a second (which is problematic and at least requires expensive hardware). Also, if the key changes take place unconditionally, this method continuously uses QKD-derived key material, even without data traffic using the IPsec connection. The frequency of key changes may adapt to the traffic throughput to achieve the desired amount of data bits per key bit, but

this seems problematic as the sender and receiver may measure different throughput rates due to packet loss. Therefore an adaption algorithm will not work properly without a more sophisticated measuring method, which might use some kind synchronizing communication all the same. These reasons lead to the rejection of this idea in favor for the master/slave approach.

5.5 Security Considerations

This section contains general advice about key change frequencies as well as ciphers and modes to use (section 8.7 gives more specific QKDIPsec configuration advice). It does, however, not consider side attacks on cryptographic systems.

5.5.1 Key Rate Requirements and Key Lengths

The strength of any cryptographic solution hinges on key strength, key protection and effectiveness of the cryptographic protocol and algorithms. (Barker, Barker, Burr, Polk, & Smid, 2012, p.5) This means more effective algorithms and more keying material[12] provide a higher level of security (apart from the obvious fact that the key has to remain secret). The IPsec standard determines the first of these two factors by the support of particular cryptographic algorithms while they themselves co-determine the second factor with their supported key lengths. Another part determining key lengths in practice is QKD. The current capabilities of QKD in generating keys determine a practical upper boundary. Current working QKD solutions (such as the one used by the AIT) provide a quantum key rate Q of up to 12,500 key bits per second at close distances, 3,300 key bits at around 25 kilometers and 550 keys bits at around 50 kilometers distance. (Treiber et al., 2009, p.9) Therefore, this solution has to be capable of applying 12,500 bits per second to fully utilize the given QKD key stream at maximum rate. When applied with a key of length k=128 bit[13], this means a key change frequency of approximately 100 keys per second. As these keys must feed both directions, the key rate for one direction is approximately 50 keys per second. This means that the solution must be able to handle one master key change (applying an outbound SA and sending a key change message) and one slave key change (reception of an key change message and applying an inbound SA) every 20 ms (section 8.5 shows the fulfillment of this requirement in this thesis) as a key period P_K. Obviously, the key rates have to be lower when applying longer keys and the quantum rates may not yield the maximum mentioned above and other applications may use key material, so that not the whole material can be used for QKDIPsec. Equation 5.2 shows the composition of the IPsec key period to utilize exactly the available quantum key rate Q.

$$P_K = (\frac{Q}{2k})^{-1} \tag{5.2}$$

[12] Given that the key is truly random, as with this QKD solution.

[13] This is currently the shortest key length recommend in cryptographic literature and therefore requires the highest key change frequency.

A lower boundary of a security rate in terms of data bits per key bits, henceforth called *security rate* - measured in Data Bits per Key Bit (DPK)- for reasons of simplicity, relies on the maximum QKD rate but also depends on the data rate. For instance, a net (data) bandwidth[14] of 100 megabits per second results in a rate of 8,000 DPK, given the practical upper QKD boundary of 12,500 key bits per second. Adjusting the *security rate* above this level, requires to restrict the data rate accordingly. A landmark in this value is 1 DPK. This rate would provide unconditional security when applied with OTP (see section 2.1.2). For the cipher and hash suites included in the IPsec protocol stack, there is no security proof and therefore they are not unconditionally secure. However, applying an IPsec cipher (for instance AES) with an accordingly fast key change and restricted data rate to achieve 1 DPK is the closest match inside standard IPsec, especially when the block size equals the key size.

A sensible upper boundary for the key period (meaning a lower standard in terms of security) is a matter of feasible computational security and the assessment of an acceptable risk. To assess this, this thesis assumes the use of QKDIPsec in a high bandwidth (10 gigabits per second) environment with an AES keying. A recent attack on AES-192/256 uses $2^{69.2}$ computations with 2^{32} chosen plaintexts (Kang, Jeong, Sung, Hong, & Lee, 2013, p.1). Given AES' block size of 128 bits, this equals to $2^{32} * 2^7 = 2^{39}$ of data bits. This attack is not *practically* feasible, as it works only for seven out of 12/14 rounds (and also has unfeasible requirements to data storage on processing power for a cryptanalytic machine). Nevertheless, it serves as a *theoretical* example for modern cryptanalysis and therefore as a role model for determining a minimum security boundary for maximum-security cryptographic environments. The bandwidth of 10 gigabits ($10 * 10^9$) per seconds equals to approximately 9.3 gibibits = $9.3 * 2^{30} \approx 2^{33}$ bits per second. This is a factor of 64 (2^6) away from the attack value above. In order to remain below the data rate, the key must be changed a least every minute. With a bandwidth of 100 megabits ($\approx 2^{26.5}$), this gives rate a more than 10 bit (2^{39} to $2^{26.5}$) security margin. This is a significantly lower key lifetime than the IPsec cryptographic suits standard proposes, which lies at 28,800 seconds or eight hours (Hoffman, 2005). At full 10 gigabit data speed, a key period of one second means a security rate of $\frac{2^{33}}{2^7} = 2^{26}$ DPK with a key length of 128 bits and of $\frac{2^{33}}{2^8} = 2^{25}$ DPK with 256 bits of key length. 100 megabits yield a rate of $\frac{2^{26.5}}{2^7} = 2^{19.5}$ DPK with a key length of 128 or $2^{18.5}$ with 256 bits, respectively.

A more practical impact (although with block sizes ≥ 128 bit not for today's networks) has the *birthday bound* (there are attacks applicable to Cipher Block Chaining (CBC), Output Feedback (OFB) and Counter (CTR) cipher modes when operating close to that bound). Equation 5.3 shows the birthday bound in relation to the cipher block size ω.

$$2^{\frac{\omega}{2}} \tag{5.3}$$

This bound is the reason to refrain from using 64 bit ciphers. (McGrew, 2012, pp.1-3) The boundary lies at 2^{32}, where a speed of 10 gigabit (2^{33}) per second exceeds this factor by two, meaning secure a key period of half a second or less. While this is not a problem for the present solution, for standard IPsec implementations this would be unusual, hence the above

[14] As bandwidth mostly is a total, gross value, the actual data bandwidth is hard to determine. It depends various factors such as packet and window sizes and paddings. In IPsec tunneling mode, this overhead partially counts to the data bits (the original packet headers, ESP trailer and paddings) as it is encrypted, partially it does not.

recommendation. For algorithms with block sizes of 128 bits, which is the highest size standardized in IPsec (see section 3.5.2), the bound is therefore 2^{64}. With 10 gigabit ethernet the key would be valid for over 68 years ($\frac{2^{64}}{2^{33}} = 2^{31}$ seconds). Even with the forseeable development of new standards, critical data rates are not imminent[15].

From a practical point of view, this solution is more secure than standard IPsec even when operating in the same cryptographic modes and with the same key change frequency. The reason is that QKD-derived keying material is truly random in contrast to a classical DH key exchange with values from standard architecture pseudo-random functions. Using different, unrelated keys (such as QKD-derived ones) mitigates related key attacks (N. P. Smart, Rijmen, Warinschi, & Watson, 2013, p.24). For instance, there is an attack on AES-192/256 that uses $2^{99.5}$ computations and data blocks (Biryukov & Khovratovich, 2009, p.1).

The key frequency is also directly proportional to the future secrecy level. The QKD system used by this thesis does not store any keys after their usage (Maurhart, 2006, p.37). Therefore, it implements perfect forward secrecy (Menezes et al., 1996, p.496). As past keys cannot be retrieved, the only possibility to extract the secrets of a stored secured connection is to break them. It is therefore obvious that, the less data has been encrypted with the same key, the less data is disclosed if the key is broken. Higher key change frequencies result in less data per key and, in conclusion, enhance the secrecy for stored encrypted communication session in future.

The key length for a single a block is determined by other factors. According to Barker, Barker, et al. (2012, p.67), key sizes 128b and above are recommended beyond 2031. Quantum computing could significantly lower the lifetime of key in this length. Grover's algorithm for quantum computers would cut the key lengths in half, if an according quantum hardware was available, even as such a device is far from realized (N. Smart, 2012, p.28). Therefore, a key size of 256 bits is recommended to prevent brute force attacks to become feasible in the event of such a device being built. This key size provides *good protection* also against quantum computers[16] (N. Smart, 2012, p.32).

5.5.2 Integrity and Authentication

Authentication ensues by QKD. As both communication partners encrypt their data using quantum distributed keys, their respective counterpart may only decrypt it with the very same key. The two peers authenticate to each other prior to exchanging keys over the quantum channel using the unconditionally secure method of universal hash classes (see section 2.1.3). The QKD-derived key itself is a true random one (see section 4.1) and an adversary has thus to rely on brute force to find it. In this particular case, therefore the knowledge of the quantum keys is regarded as sufficient for authentication purposes.

The authenticity of the key synchronization packets appears from using these keys in ESP and AH modes, respectively. The control channel is fully authenticated with AHs (see section

[15] An example for a *critical* data rate would be Exabit $10^{18} \approx 2^{60}$ bits per second. The birthday bound would then be met in $2^4 = 16$ seconds, needing an according key rate.

[16] Given Shor's algorithm does not apply, which it actually does not, as this is an algorithm for prime factorization and AES does not work on basis of prime factors

5.4.3), while the data itself is encrypted and therefore only using the correct (authentic) keys would extract meaningful data. Using the AH also ensures the integrity of the control channel, while ESP provides integrity and authenticity of the data channel (through an Integrity Check Value (ICV)) if used with an authentication algorithm. Using Counter with CBC-MAC (CCM) or Galois/Counter Mode (GCM) cipher modes (providing both authentication and encryption combined) ensures full integrity and authenticity for the data channel[17](Housley, 2007, p.4). As a tradeoff, ciphers in these modes use 24 (CCM) and 32 (GCM) bits more of keying material as a salt value. The salt does not necessarily have to be QKD-derived, for it is not required to be secret. It has, however, necessarily to be an unpredictable value. (Viega & McGrew, 2005; Housley, 2005, pp.4-7) These services provide only computational security, but have a practical advantage over OTP, which does not provide data integrity by itself but has to be combined with some integrity check algorithm.

N. P. Smart et al. (2013, p.58) advise only to use CTR mode ciphers with an additional authentication algorithm or to use CCM or GCM mode combined ciphers. The National Institute of Standards and Technology (NIST) discourages the use of the *VPN-A* and *VPN-B* (see section 3.5) cipher suites (particularly it does not approve *AES-XCBC-MAC-96*) (Barker, Burr, et al., 2012, p.33). It also requires AES-CBC to be supported but also requires additional integrity protection for CTR (Barker, Burr, et al., 2012, p.34).

QKDIPsec provides the opportunity to choose between a separate authentication and encryption mode or a combined one. The first method uses one of the data keys for the duration of a session as a key for the authentication algorithm. For the second method, this solution uses quantum keys for both key and salt, but a Pseudorandom Number Generator (PRNG)-derived salt value for GCM or CCM mode is also thinkable. For configuration recommendations regarding QKDIPsec please refer to section 8.7.

[17]So far, these modes are only defined for AES to use in IPsec (Internet Assigned Numbers Authority, 2012b).

6 Description of the Used Application Programmers Interface

This chapter starts with an explanation of the reasons that lead to the given choice of the Linux Internet Protocol security (IPsec) interface and describes the same, including the C++ Application Programmer Interface (API) used to interact with the Linux IPsec subsystem. It further describes another protocol of the very same API as it is used to handle network interfaces and routing. The documentation material about the Linux IPsec APIs is very rich in regards of user documentation and how to set up protected connections but very sparse in terms of developer documentation, it is therefore one of the aims of this chapter to close this gap and provide such a documentation. Due to this, the acquired knowledge about the used API is, if not marked otherwise, the result of C++ code reverse-engineering of the Linux *ip xfrm* command of the *iproute2*[1] package, precisely the files *ipxfrm.c*, *xfrm_policy.c* and *xfrm_state.c*, with help of the respective *ip xfrm* and *Netlink* Linux manual pages by Ward (2011) and Kerriks (2012a, 2012b). These efforts where further supported by a *Netlink* tutorial by He (2005) and a small Transform (XFRM) tutorial by Talijanac (2012). The data types descriptions and usage is derived from the */usr/include/linux/xfrm.h* file. The respective sections cite these sources only explicitly where applicable; mostly the understanding of these topics result from a synthesis of above sources without any clear accountability. QKDIPsec uses the very same API for network interface handling and routing as for handling IPsec. This part of this thesis rests on a paper by Horman (2004) and code examples from `stackoverflow.com`[2]. Apart from this, the main source is also reverse engineering of the sources from the *iproute2* package, in this case the files *ipaddress.c* and *iproute.c*.

6.1 Choice of a Linux IPsec Implementation

As this is a highly secure application of IPsec (for it has to maintain the high security path paved by QKD), the IPsec implementation should be chosen as secure as possible. Most of the known IPsec implementations have many vulnerabilities, so the number of known vulnerabilities is a factor on chosing a secure implementation. Two of the most well known implementations, *Openswan*[3] and *strongSwan*[4] show 17 and 12 vulnerabilities in the Common Vulnerabilities and Exposures (CVE) database[5], respectively.

[1] `http://www.linuxfoundation.org/collaborate/workgroups/networking/iproute2`
[2] Most notably `http://stackoverflow.com/questions/14369043`
[3] `https://www.openswan.org/`
[4] `http://www.strongswan.org/`
[5] As of 01-Nov-2013 at `http://cve.mitre.org/cve/cve.html` under the respective search terms

Furthermore, these implementations represent an ancillary middleware between the solution presented in this thesis and the Linux kernel, which possesses, as section states, a native IPsec implementation since version 2.6. Using direct kernel access supports the desire to use as few additional programming libraries as possible (which might not be available) and has the promising outlook to achieve higher performance (which in this case means higher key rates per second and lower consumption of resources) by omitting any middleware. Above all, the presented approach has no need of most of the functions provided by said implementations, as the IPsec implementation itself does not have to be altered, but only the key exchange. Therefore, it is sufficient to control the Security Policy Database (SPD) and Security Association (SA) databases and most prominently provide a key feed for the IPsec SAs in the Linux kernel.

Methods to invoke Linux kernel functions in order to fulfill this task include system calls, *procfs* and *Netlink sockets*. Of these methods, the latter provides the most powerful and (by the use of the well proliferated socket structure) familiar interface. (He, 2005) Additionally, there is a specialized API for the *KAME* IPsec implementation named Protocol Family Key (PF_KEY) which derives from the Berkeley Software Distribution (BSD) routing socket Protocol Family Route (PF_ROUTE) (McDonald et al., 1998, p.3). As section 6.1 states, the *ipsec-tools* library uses this API and the XFRM architecture to access the the Linux IPsec engine. Another toolset, the *iproute* project[6], uses *Netlink* to provide management for a variety of network functions such as network devices, Internet Protocol (IP) addresses, routing, tunneling and *TUN/TAP* interfaces, but also manipulating the SPD and Security Association Database (SAD) for IPsec. (Litvak, 2011) The afore mentioned IPsec implementations also rely on this APIs for key management. For an example, *strongSwan* supports both *Netlink* and PF_KEY as well as the legacy Kernel IPsec (KLIPS) API, using *Netlink* as a default (Lang, 2006).

The Internet Engineering Task Force (IETF) has standardized both PF_KEY and *Netlink*, the former by McDonald et al. (1998) and the latter by Salim, Khosravi, Kleen, and Kuznetsov (2003). The standardized, socket-like interfaces both of these APIs provide make them preferable over system calls and *procfs*. This implementation uses *Netlink* sockets to manipulate the SAD and SPD as this API is found to be more intuitive and conduces the reuse of code due to its ability to administer a broad span of network functions. The present solution uses this additional network functions to manage IP addresses and network routes for the IPsec tunneling, which the PF_KEY API cannot handle. Although there are also 31 known *Netlink*-related CVEs[7] (of which the majority belongs to modules using *Netlink* instead of the libraries itself), using this API does not change the security of the system. *Netlink* is a deeply hooked part of the Linux kernel and therefore the vulnerabilities exist regardless of its usage.

6.2 Netlink

Netlink is a network protocol-like, socket-oriented service to enable communication between kernel and user space processes. Therefore, the *Netlink* data types follow a network packet-like structure. (Kerriks, 2012b). This means that the reference to data structures happens by alignment and length fields rather than reference by pointers, unlike as otherwise usual in

[6] `http://www.linuxfoundation.org/collaborate/workgroups/networking/iproute2`
[7] As of 01-Nov-2013 at `http://cve.mitre.org/cve/cve.html`

programming languages. In other words, the principle is similar to crafting a network packet and sending it over an IP network, only that the network is a node-internal Inter-Process Communication (IPC) system. Therefore, the structure uses a *Netlink* header as lead-in for the data. *Netlink* provides a broad variety of tools to manage different network functions. The API to issue commands with *Netlink* closely resembles the sockets API, which is the de facto standard for Transmission Control Protocol (TCP)/IP applications. (Gilligan, Thomson, Bound, McCann, & Stevens, 2003) Section 6.2.2 provides further details regarding *Netlink* sockets. Similar to standard sockets, *Netlink* uses protocols, which in this case each correlate to a toolset. Currently defined *Netlink* protocols are: (Kerriks, 2012b)

- *NETLINK_ROUTE* manages interfaces, routing and addresses (see section 6.4);
- *NETLINK_XFRM* manages the IPsec subsystem (see section 6.3);
- *NETLINK_W1* manages the 1-wire subsystem;
- *NETLINK_USERSOCK* provides an interface for user-mode socket protocols;
- *NETLINK_FIREWALL* transports Internet Protocol version 4 (IPv4) packets from the *Netfilter* firewall to the user space; Originally intended for module development, it now serves numerous purposes (Horman, 2004, p.11);
- *NETLINK_IP6_FW* provides the *NETLINK_FIREWALL* function for Internet Protocol version 6 (IPv6) packets;
- *NETLINK_INET_DIAG* is used for socket monitoring;
- *NETLINK_NFLOG* is used for firewall logging;
- *NETLINK_SELINUX* provides SELinux notifications;
- *NETLINK_ISCSI* manages Open-iSCSI;
- *NETLINK_AUDIT* is used for auditing;
- *NETLINK_FIB_LOOKUP* provides an interface to the Linux Forwarding Information Database (FIB);
- *NETLINK_CONNECTOR* provides a connector to the Linux Kernel;
- *NETLINK_NETFILTER* manages the *Netfilter* firewall;
- *NETLINK_DNRTMSG* provides access to routing messages from DECnet;
- *NETLINK_KOBJECT_UEVENT* provides monitoring for user space Kernel messages;
- *NETLINK_GENERIC* is a protocol for simplified *Netlink* usage.

The present solution uses the *NETLINK_ROUTE* and *NETLINK_XFRM* protocols to handle network and IPsec subsystems.

6.2.1 Netlink Header

The *Netlink* header defines the length, type, flags, sequence number and *port ID* of a *Netlink* datagram (see figure 6.1 for an overview). This ID represents the sender of the message and therefore is, as it is system-internal, a process number, with a value of zero means the kernel as originator. The length means the total length of the message, including the header. As multipart messages are also possible, the sequence number indicates their order. The type could be one of the following (Kerriks, 2012b)[8]:

- No type (0) - a standard message;
- NLMSG_NOOP (1) - a message to be ignored;
- NLMSG_ERROR (2) - an error message;
- NLMSG_DONE (3) - concluding part of a multipart message;
- NLMSG_OVERRUN (4) - indicates lost data;
- NLMSG_MIN_TYPE (10) - reserved control messages.

The present solution actively sends only standard types (0), while it may receive and process error messages (2) from the kernel. Also, from the various flags which might occur. This approach uses two of them:

- NLM_F_REQUEST (1) - mandatory for all requests;
- NLM_F_ACK (4) - the addressed kernel module should send an answer.

While the first flag is self-explanatory, the second indicates an unconditional answer. Without that flag, kernel modules ordinarily send only answer in case of an error (the error messages of type 2 stated above, with an appropriate error code). This flag changes that behavior, so that the kernel module should send an error message (type 2), but with zero as an error code. (Kerriks, 2012b) The reason this work uses acknowledge flags is that the function for receiving messages does apparently not time out in a reasonable timespan[9] and waits until a message arrives (even if none ever does). The invocation of this receiving method is however necessary to detect errors on sent requests (for instance a duplicate SAD or SPD entry). The only way to resolve this situation is to provoke answers from the IPsec system to a successful request as well. For an exception to this program behavior see section 7.2.

6.2.2 Netlink Socket

In order for a *Netlink* packet to traverse to the kernel space, it needs a socket (which the C++ function *socket()* provides). The address for *Netlink* packets is *AF_NETLINK* and the protocol

[8] The man page omits the types NLMSG_OVERRUN and NMSG_MIN_TYPE but they are defined in */usr/include/linux/xfrm.h*

[9] If it does at all - while testing the functions, it never actually did.

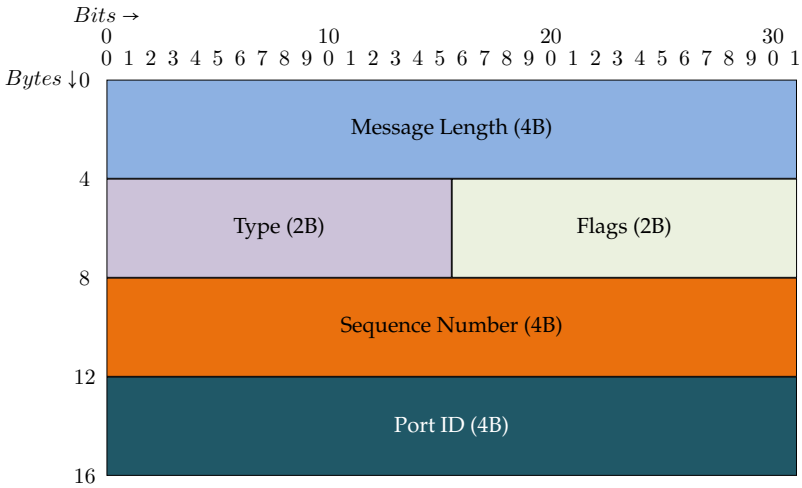


Figure 6.1: Netlink Header Structure

a *Netlink* protocol. The next step is to bind the socket onto a address with the function *socket()*. This address (of type *AF_NETLINK*) is usually the process number and should be the same as the one used in the *process ID* in the *Netlink* header. Closely resembling the standard sockets API (see section 6.2) the socket function calls are quite similar, as listing 6.1 displays. While the first entry (Gilligan et al., 2003, p.9) opens a standard TCP socket[10], the second one (Salim et al., 2003, p.29) opens a *Netlink* socket.

```
socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
socket(AF_NETLINK, SOCK_RAW, NETLINK_ROUTE);
```

Listing 6.1: Comparison between standard and Netlink socket calls

The function *sendmsg* then sends the message given the socket and a message structure. This structure contains a *name*, which is in this case the *Netlink* address of the receiver (mostly zero, for the kernel as receiver) and a pointer to a Input-Output Vector (IOV) structure including their respective lengths. The latter structure eventually contains the *Netlink* message in form of a pointer to its beginning, called *base*. Additionally it also holds a length which is the complete size of the message (*Netlink* header and payload - where payload refers to to a *Netlink* protocol like the ones described in the next sections). (Wright & Stevens, 1995, pp.481-483) The message in the given length lies in the memory in the above stated alignment. Figure 6.2 illustrates the relations between the *Netlink* and Socket API structures in a style based on Unified Modeling Language (UML). Although UML is not designed and well-suited to depict non-object oriented concepts, it seems adequate to give an overview of the *Netlink* socket API, also including functions (especially the *socket()* function, whose returned socket number is also a parameter for *bind()* and *sendmsg()*) as node types. In this case UML *has-a* relations mean parameter inputs and *uses* relations mean that they take the return value of the *used* function as an input.

[10] *SOCK_STREAM* is the standard for connection oriented protocols like TCP. Also, the original source uses zero a protocol number, which is a dummy protocol for TCP. The present listing uses *IPPROTO_TCP* instead in order to provide a better comparability with the *Netlink* socket call.

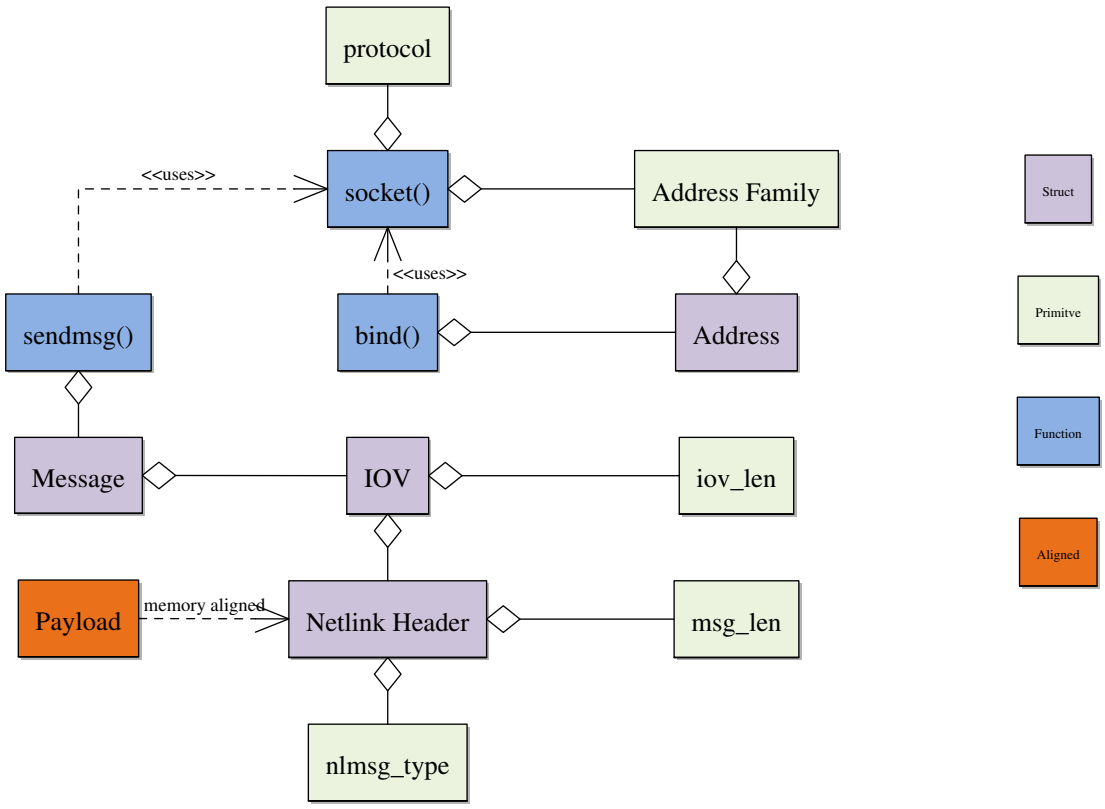


Figure 6.2: C++ *Netlink* pseudo UML diagram

6.3 Transform

The Transform (XFRM) protocol (*NETLINK_XFRM*) is a *Netlink* protocol used in the Linux *ip* command to manage the Linux IPsec subsystem. More precisely, the command *ip xfrm policy* handles SPD entries and *ip xfrm state* SAD entries. To add an SPD entry similar to the example in section 6.1, *ip* uses the sequence

```
ip xfrm policy add src 143.224.68.25 dst 143.224.68.23 proto esp↩
    dir out ptype main priority 2147483648
tmpl proto esp mode transport
```

Listing 6.2: Example configuration for ip xfrm

for the SPD entry. To send XFRM messages to the kernel, the protocol type of the socket is *NETLINK_XFRM* (Kerriks, 2012b). Each XFRM message consists of the *Netlink* header and an XFRM specific part as a payload. While there are other possible forms to use these messages, the following sections explain the structures of the SPD and SAD messages used by this work and concentrate on the XFRM part. For *Netlink* details and mechanisms to send these packets please refer to section 6.2.

6.3.1 Handling the Security Policy Database

This approach uses two *Netlink* XFRM message types to add and remove entries from the SPD:

- New policy (XFRM_MSG_NEWPOLICY);
- Delete policy (XFRM_MSG_DELPOLICY).

The new policy message contains the *Netlink* header, a user policy (xfrm_userpolicy_info) and one or more user templates (xfrm_user_tmpl) aligned in Type-Length-Value (TLV) format with the help of the rtattr data type. Figure 6.3 displays the structure of a *Netlink* XFRM new policy message.

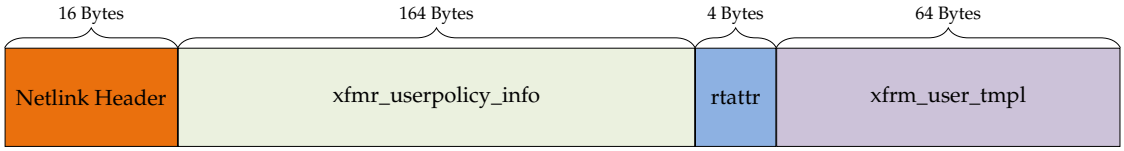


Figure 6.3: Structure of a New Security Policy Message

The user policy is a 164 byte-structure which would be too much to display in its entirety at this point. Noteworthy parts of this structure are the substructures of *lifetime config*, which consists of byte, packet and time limits for the Security Policy (SP), the value for the direction (inbound or outbound), and the *selector* (*XFRM_SELECTOR*). The *selector* comprises the source and destination IP addresses, IP prefixes, port numbers and port masks as well as the address family, in most cases IP version 4 (AF_INET) or 6 (AF_INET6). Additionally, the policy contains an action which could be block (*XFRM_POLICY_BLOCK*) or allow (*XFRM_POLICY_ALLOW*). As tests showed, this constitutes a complete SP except for the *user template*. Therefore, a message provided with this parameters would result in a valid SPD entry who directs the kernel to send the packet with no user template, effectively performing the IPsec BYPASS option if the action is allow or DISCARD if the action is deny (see section 3.1.1). Consequently, the *ip xfrm policy* command simply shows an SP with no user template, while *setkey -PD* shows *none* as user template if the action of such a message is allow while the two commands show the notices of *action block* and *discard* if the action is block, respectively.

The user template, if present, occurs after the user policy. It contains the information about the IPsec protocols (Authentication Header (AH) and Encapsulating security Payload (ESP)) and modes (transport or tunnel), including *outer* IP address for tunnel mode connections and an optional Security Parameters Index (SPI) for the SP. As an SP may contain multiple templates, their alignment shows classic TLV encoding, with an additional structure, the *rtattr*, announcing type and length of the template. This 4 byte-structure contains the length (*rta_len*) first and the type (*rta_type*) as a second attribute. In case of a user template the type is *XFRMA_TEMPL*. Figure 6.4 shows the structure of TLV-aligned user templates.

A user template itself has a size of 64 bytes. Messages with according user templates (with protocol and mode set) included create SPD entries that exhibit the PROTECT action. The

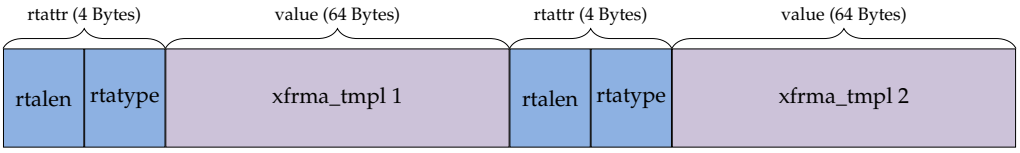


Figure 6.4: Structure of User Templates in TLV Format

command *ip xfrm policy* therefore shows an entry with template and without *action block* notification while *setkey -PD* shows *ipsec* as template. The listing 6.3 shows the output of *setkey -PD* for all three types in comparison.

```
10.0.11.0/24[any] 10.0.11.33[any] 255
        in none
        created: Nov 24 16:21:21 2013  lastused:
        lifetime: 0(s) validtime: 0(s)
        spid=8 seq=0 pid=4134
        refcnt=1

10.0.11.0/24[any] 10.0.11.33[any] 255
        in discard
        created: Nov 25 01:31:28 2013  lastused:
        lifetime: 0(s) validtime: 0(s)
        spid=256 seq=1 pid=14503
        refcnt=1

10.0.11.0/24[any] 10.0.11.33[any] 255
        in ipsec
        esp/transport//require
        created: Nov 24 16:24:15 2013  lastused:
        lifetime: 0(s) validtime: 0(s)
        spid=8 seq=0 pid=4244
        refcnt=1
```

Listing 6.3: Comparison of ALLOW, BLOCK and PROTECT SPD Entries

The message to delete SPD entries is similar, only that the message after the *Netlink* message header has the type *xfrm_userpolicy_id* and basically consists only of the direction, selector and index and contains no templates, although there are more identifying options possible. It is therefore a smaller, 64 bytes big, equivalent of a new policy message with only the parts that identify one SPD entry, as only the identification of the respective SP is necessary to delete it. Figure 6.5 gives a picture of this message.

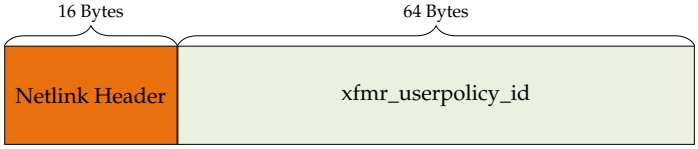


Figure 6.5: Structure of a Delete Security Policy Message

6.3.2 Handling the Security Association Database

Analogous to the SPD in section 6.3.1, this approach uses two messages handle the entries of the SAD:

- New SA (XFRM_MSG_NEWSA);
- Delete SA (XRFM_MSG_DELSA).

The new SA message contains, apart from the 16-bytes *Netlink* header, a structure of the type xfrm_usersa_info and the cryptographic algorithms with the respective keys in TLV format. The 220-bytes *usersa* info contains the same selector structure as a new policy message does, although it does not contain the main selector. Instead the processed address values reside in an id structure (*xfrm_id*) immediately following the selector and containing the destination address, SPI and protocol values, and an additional source address field following the id structure. Other values in the new SA request contain replay parameters, the address family (mostly *AF_INET* or *AF_INET6*, the IPsec mode and lifetime configuration similar to a new policy request. It also contains fields for the current lifetime values. Figure 6.3 displays the structure of a *Netlink* XFRM new policy message.

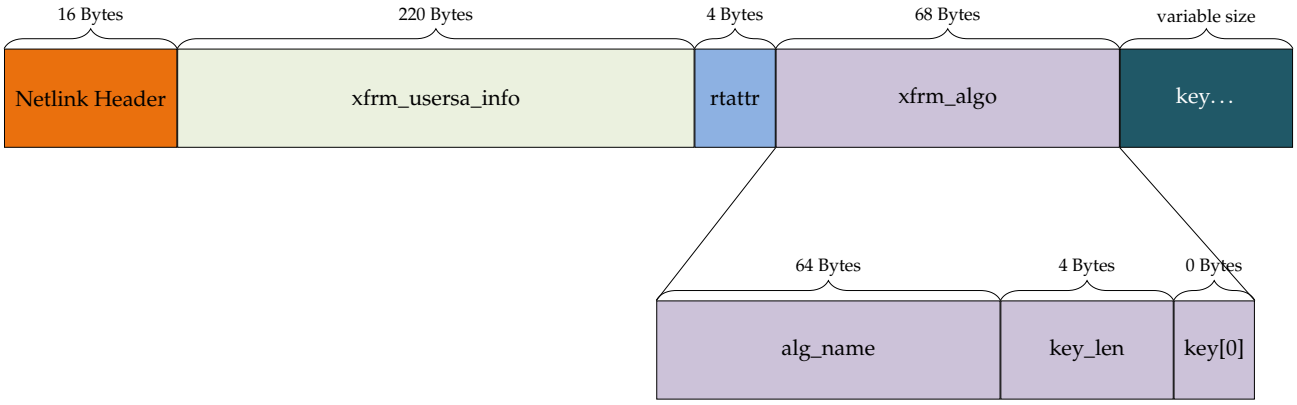


Figure 6.6: Structure of a New Security Association Message

The *rtattr* structure has the same TLV-aligned, 4 byte, length and type structure as in user policy messages. The value part, however, consists of a cryptographic algorithm set instead of a user template. This set (type *xfrm_algo*) has a size 68 bytes, of which 64 are reserved for the algorithm name (for instance *aes* or *twofish*) and 4 bytes of key length. Additionally, it contains a character field of length zero for the key. The key itself is supposed to follow immediately, featuring the same length specified in the *rtattr* length field. Therefore, the zero-sized character field pointer automatically points to the key. If an ESP SA should protect its traffic with both protection and authentication it has use of two algorithms, therefore the according message contains two consecutive TLV structures. Alternatively it may contain a combined algorithm, thus containing again only one of these structures, but with an additional 4 byte Integrity Check Value (ICV) field, placed between the key length and the key pointer.

Like the delete policy message, the delete SA message assembles only the parts of its creating counterpart, which identifies a database entry and is therefore necessary to delete it.

These components are the destination address and its family, the protocol and the SPI, constituting the 24 bytes long type *xfrm_usersa_id*. Unlike delete policy, it needs an additional TLV-aligned part (again with a 4-bytes *rtattr* struct leading in) to delete entries based the source address. In this case, the value after the lead in is the 16 bytes long address structure (*xfrm_address_t* - section 7.6 discusses the address structure in more detail). Figure 6.7 shows the composition of the complete message.

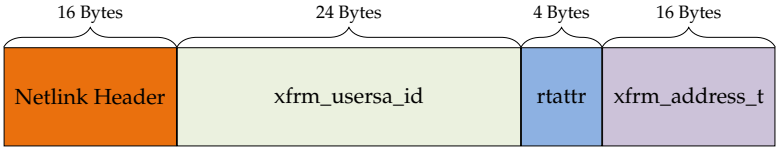


Figure 6.7: Structure of a Delete Security Association Message

6.4 Route

The route protocol (*NETLINK_ROUTE*) is a *Netlink* protocol that handles the Linux routing subsystem including methods to manipulate network interfaces. The present solution uses the *Netlink route* API to manage IP interfaces and routes in order to create a virtual link for the IPsec tunnel mode. It is more intuitive than XFRM, as the add and delete messages of the route protocol are structured identically, much in contrast to XFRM.

The message to add and delete addresses consists of a structure called *ifaddrmsg* following the mandatory *Netlink* header. This structure contains the following items (Kerriks, 2012c):

- The address family of the address to add or remove (mostly IPv4 or IPv6);
- The prefix length of the address;
- The scope of the address (for instance global or link local);
- The index of the link to which it is to be attached to or removed from;
- Additional flags.

The address to add or delete from the interface in question follows in TLV format. Conveniently, it is possible to reuse the address structure from XFRM, this keeps the structure consistent with the other parts of the solution and versatile enough to use both IPv4 and IPv6 addressing. Figure 6.8 depicts the structure of this type of message. The messages to add and delete addresses to and from interfaces are completely identical except for the *Netlink* message type. This means that adding and deleting an address differs only by setting the message type in the *Netlink* header to *RTM_NEWADDR* to add or *RTM_DELADDR* to delete an address. (Kerriks, 2012c)

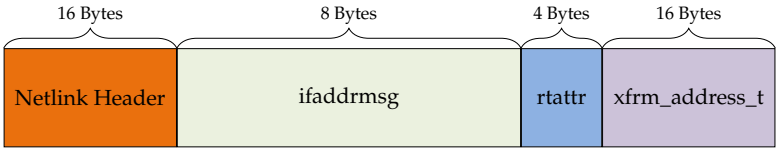


Figure 6.8: Structure of an Address Message

Analogously, the message to add or delete a route contains a structure named *RtMsg* that contains the following items (Kerriks, 2012c):

- The address family of the route (mostly IPv4 or IPv6)
- The source and destination prefix lengths
- The type of service;
- The routing table ID (as Linux uses Routing tables for different purposes)
- The route origin (Internet Control Message Protocol (ICMP) redirect, the kernel, boot, the administrator or unspecified)
- The scope of the route (for instance global or link local)
- The route type (for instance unicast)
- Additional flags

Likewise to the address messages, route messages contain an address, this time the destination address in TLV format. Regarding routes, the add and delete messages differ not only by the type (*RTM_NEWROUTE* opposed to *RTM_DELROUTE*), but also by the fact that the add route contains an additional parameter (in TLV) which specifies the gateway address, as figure 6.9 depicts. This is naturally not necessary for delete messages. (Kerriks, 2012c)

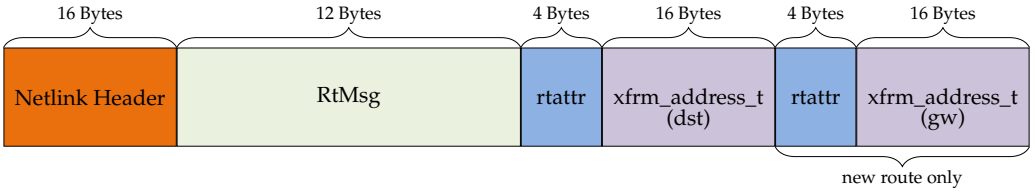


Figure 6.9: Structure of a Route Message

7 Solution Implementation

This chapter describes the software design and implementation of QKDIPsec. It describes the class design of the approach in detail, also explaining the crafted classes for Internet Protocol security (IPsec) and interface management. A description of the synchronization management system, which handles the synchronization of the IPsec keys with the peer host concludes the part about the System components. The Netlink Application Programmer Interface (API) parts of this implementation are based on the *iproute2* package as chapter 6 describes. As stated in the introduction, the solution has to simulate Quantum Key Distribution (QKD) behavior until the interface function at the side of the Austrian Institute of Technology (AIT) QKD software has been finished. This is the purpose of the key manager. It also serves as an adapter class to the QKD system when the respective function is finished. The solution uses its own namespace *qkd::IPsec* in accordance with the QKD software that uses *qkd* and subspaces.

In order to describe the design more visually, this chapter contains some Unified Modeling Language (UML) class diagrams. These diagrams deviate from the standard in that they do not contain return types for functions and function parameters, but each function has zero, one or more dots between its related parentheses to indicate the quantity of its parameters. The thesis uses this convention in order to save space, otherwise the classes could not be shown in that level of detail. As the central class, *ConnectionManager* contains a large number of class members which exhibit C++ standard data types, the according class diagram also truncates them to only showing the quantity per data type (nevertheless, it does display class members of own data types in full). As a single diagram displaying all the components would be far too extensive and complicated, this explanation splits the classes into three main parts and some extra utility and configuration classes. To increase clarity, the displayed classes have different colors, corresponding to the respective part of the solution. Blue represents the connection manager (responsible for key synchronization and the overall control) and the QKD adapter parts, purple the Netlink message parts, green the IPsec and network manger parts with their factories and orange represents utility and configuration classes. If some classes do not exhibit class members, they are not the focus of the respective diagram (and thesis section) and appear in detail in one of the other diagrams. Also the solution uses only default destructors (except the connection manager), therefore destructors are also omitted in the class diagram.

7.1 The Connection Manager

The connection manager (class *ConnectionManager*) has two main purposes:

- Serving as main controller for the whole QKDIPsec system (and thus, being an entry point for module calls);
- manage the key synchronization between the peers and execute the protocol explained in section 5.4.

To fulfill this task, the connection manager has four different IPsec managers and one network manager (see section 7.2). The IPsec manager handles the Security Policys (SPs) and Security Associations (SAs) for each direction of the data and control channel (see section 5.3), while the network manager handles Internet Protocol (IP) addresses and routes. It also has four key managers serving key feeds for each IPsec manager (see section 7.5). The class takes an IPsec (*IPsecConfig*), key (*KeyConfig* - which means in this case QKD) and key synchronization (*KeySyncConfig*) configuration, of which it distributes the first two to the according managers, while the last serves as the connection managers own configuration. This configuration holds the queue sizes and key change rates (in milliseconds for the data and in seconds for the control channel), as well as the timeout value after which the manager gives up on a failed system reset attempt and ceases to work. In the latter case it suspends all functions but is ready to immediately reconnect to the peer (in the meanwhile it blocks the data connection - see section 5.4.5). The routine for the key synchronization splits (apart from IPsec connection initialization routines, setting initial keys and SPs) in a master (sender, started with *start_controlChannelSender(int port)*) and a slave (listener, started with *start_controlChannelListener(int port)*) part, both running in dedicated threats that may be started and terminated via the connection manager[1]. The listener wait for a peer connection (via *socket*[2] *bind()* and *accept()*) and then goes through a loop that receives key synchronization messages (containing a type and value) and reacts accordingly by installing SAs via the IPsec managers and sending appropriate answers to the peer. The sender also possesses a main loop which it enters after a *socket connect()*. Within this loop, it basically sends the key change and reset messages for both channels according to the set timer. The timer for the data channel sets the clock rate for the loop (realized with a *sleep()* function), for it is supposed to be the smaller of the two rates. The key synchronization configuration validates this at its creation time. Based on the key change rate, the routine calculates a multiplier for full seconds and uses this as a base for the control channel key change and timeout values. Therefore the sender of one peer connects to the listener of the other and vice versa. To send answers, the listener co-uses the sending connection of the sender but is solely responsible for evaluate answers. Both routines exhibit a mechanism to wait for each other on the same machine in order to ensure that the control channel in both direction is open. Otherwise one peer would not be able to send and acknowledge key changes, causing both peers to try to initiate a reset process due to the lack of control communication and ultimately (if the problem persisted) to

[1] Technically they terminate themselves, for they are detached threads. The connection manager has a *stop* function for both threads that sets one of three variables; one to end each of the threads and a third to indicate a fatal error, ending both threads. The threads read those variables and terminate themselves if they are set accordingly.

[2] The socket uses the option *SO_REUSEADDR* in order to prevent timeout issues in case of an unexpected connection termination.

end their sender and listener routines. The key change rate and the IPsec SA lifetimes may also be changed during operation. The calling entity might measure the data rate on the data throughput and adjust the key period accordingly to achieve a desired *security rate* (see section 5.5.1). In future, the connection manager may choose the key change rate accordingly by itself, for instance by reading sent data values from an interface and changing the key if it reaches a ceratin threshold. Figure 7.1 shows the connection manager in context of other classes. As other IPsec APIs than *Netlink* are thinkable, the connection manager accepts an abstract IPsec manager *factory*, which might be filled with different concrete implementations (see section 7.2). This resembles in some aspects, though not fully implements, a *strategy pattern* (Freeman, Robson, Bates, & Sierra, 2004, p.22).

7.2 Kernel and Netlink IPSec Manager

The kernel IPsec manager (class *KernelIPsecManager*) is responsible for adding and deleting entries into and from the operating system's IPsec databases, the Security Policy Database (SPD) and the Security Association Database (SAD). It therefore provides a function to add and delete an SA and an SP (where there are different adding methods for transport and tunnel mode SPs. It also provides a separate method for updating the key. In order to be open to the usage of other APIs, the kernel IPsec handling consists of a virtual *KernelIPsecManager* class and concrete implementors. The kernel API used in this solution is Netlink (while other possibilities include Protocol Family Key (PF_KEY) - see section 6.1), therefore the actual implementing subclass is *NetlinkIPsecManager*. Its second parent class *NetlinkManager* provides methods to send Netlink packets (and receive the respective answers) using the functions from section 6.2.2. The manager therefore serves as a mediator between the *Netlink* system and the connection manager. Doing so, it instantiates the message classes from section 7.3 and sends them to the *Netlink* system. The *Netlink* IPsec manager also possesses an add and delete SA message as a class member to speed up the key change process (whereas it instantiates the other *Netlink* messages when needed). These to messages contain all necessary information to install (and delete, respectively) an SA for the IPsec connection its manager controls. On key change, those messages only gain new Security Parameters Index (SPI) an key values and are then immediately sent to install an new SA and delete the old one, effectively changing SPI and key. The *Netlink* IPsec manager also possesses a specialized method to send the necessary *Netlink* messages more swiftly by refraining from provoking an answer, sending them without the *NLM_F_ACK* flag (see section 6.2.1). To further speed up the process, all methods taking part in the key update process are *inline* functions. The network manager also uses the same scheme with an abstract kernel class (*KernelNetworkManager*) and a concrete implementation (*NetlinkNetworkManger*, which has also *NetlinkManager* as a second base class). These classes are simpler than the IPsec manager classes and provide methods to add and delete network addresses and routes. As the connection manager possesses multiple managers (four IPsec, one network) it uses a *factory pattern* (Freeman et al., 2004, p.134) to instantiate them. The abstract factory *KernelManagerFactory* and (in this *Netlink* implementation) the concrete *NetlinkManagerFactory* provide all necessary managers used by the connection manager. The connection manager has the kernel manager factory as a constructor parameter and uses it to subsequently instantiate its respective manager classes in the required amount.

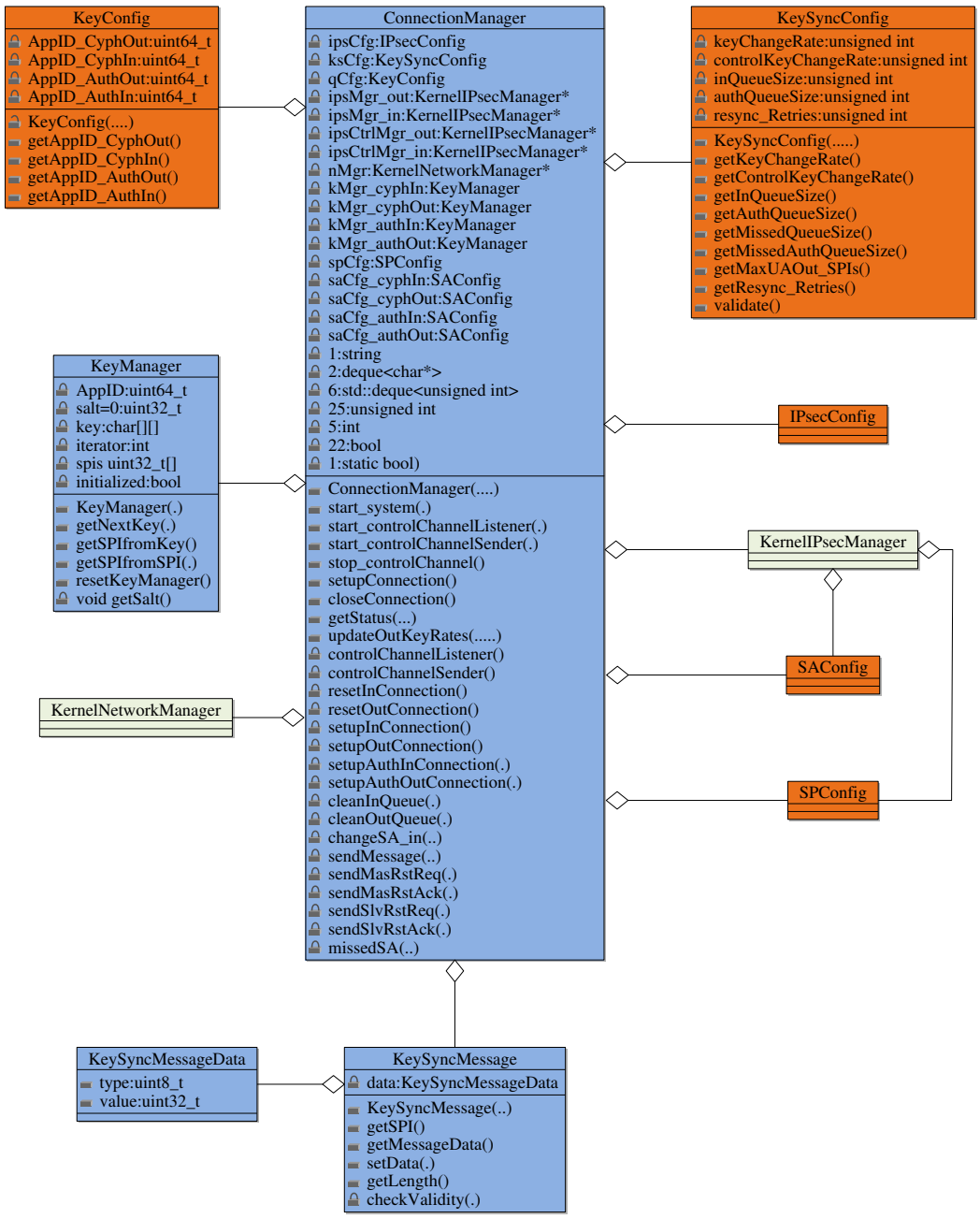


Figure 7.1: UML Diagram of the Connection and Key Manager Environment

An additional Class, *IPsecConfig*, encapsulates the IPsec configuration. This class consists of the source and destination IP address, as well as the tunnel source and destination addresses, if working in IPsec tunnel mode. There are two constructors for the class. The first constructor only accepts source and destination addresses and sets a boolean *tunnel_mode* variable to false, while the second additionally accepts the tunnel addresses as parameters and sets the boolean variable to true. This configuration class therefore determines also the IPsec mode. Trying to read tunnel addresses from a configuration in transport mode results in an *exception*. Section 7.3 describes specialized classes for the other configuration items of these databases.

The IPsec mode must match in both databases in order for the IPsec connection to function properly. The IP addresses used in the SA correspond to the source and destination addresses in the SP if the configuration uses IPsec transport mode. In tunnel mode, on the other hand, these addresses correspond to the tunnel addresses of the SP. (Bausewein, 2012)[3] Therefore, the SA addresses must be set differently, depending on the IPsec mode, nevertheless the class stores it mode afer instantiation. It also contains other necessary information like ciphers and lifetimes. Figure 7.2 provides an overview of these classes in a UML class diagram.

7.3 Netlink Messages

The *NetlinkMessage* class serves as an abstract (or virtual) base class. It abstracts the functions of its subclasses used and contains the common Netlink header as member variable, while its subclasses hold the corresponding Transform (XFRM) (or *route* header and payload (including its lead-in). The alignment-based structure, as section 6.2 describes it, and the resulting lack of pointers dictate a strict lineup of the data contained in the class in order for *Netlink* to read it properly. The only reason why a supertype-subtype relation between the class holding the Netlink header and the classes holding the variable payloads is possible in this case is the memory segmentation of current computers. Because of this segmentation, class code and its data objects reside in different memory segments. (von der Linden, 1994, pp.142-143) Therefore the data structures of the superclass lie immediately before the ones of the subclass without any member function code in between. This leaves the packet alignment structure intact as long as the classes do not have any additional member variables that lie in between and therefore would interrupt the packet-like structure in the memory. Also, some system internal realignment procedures for optimization could jeopardize this structure. With the current alignment, the subclass may process the packet structure as a whole unit, provided it has sufficient access of the header structure, which means it has to be *protected* or *public* or the superclass provides appropriate *getter* and *setter* methods for all necessary attributes of the header. QKDIPsec uses the latter approach. The *getter* methods are public to access to provide the message size and a pointer to the beginning of the message for the Input-Output Vector (IOV) structure (see section 6.2.2). There are also protected *setter* methods to set the size and message type for the subclass, as the superclass is unaware of the type and therefore (for these message structures have different sizes) also the size.

The XFRM subclasses are *NetlinkNewSAMessage, NetlinkNewSPMessage, NetlinkDeleteSAMessage* and *NetlinkDeleteSPMessage*. Each class contains a message structure resembling the one sections 6.3.1 and 6.3.2 describe. The message types with a Type-Length-Value (TLV)-aligned structure contain that structure in a nested, data-only *payload* class, which in each case holds the *rtattr* and the respective payload itself. This aligns the payload data sequentially after the respective XFRM data in the same manner as the parent/child relationship aligns the XFRM data immediately after the Netlink header and therefore results in valid compound Netlink XFRM message. All of these messages take their configuration directly as constructor parameters, most prominently the source and destination IP addresses, while new SP messages possess an additional constructor with the possibility to add tunnel source

[3] This behavior was found out with hints from this source and comprehensive testing under Linux. There was no evidence found in the corresponding RFC by Kent and Seo (2005). Therefore this might only represent the Linux implementation of IPsec.

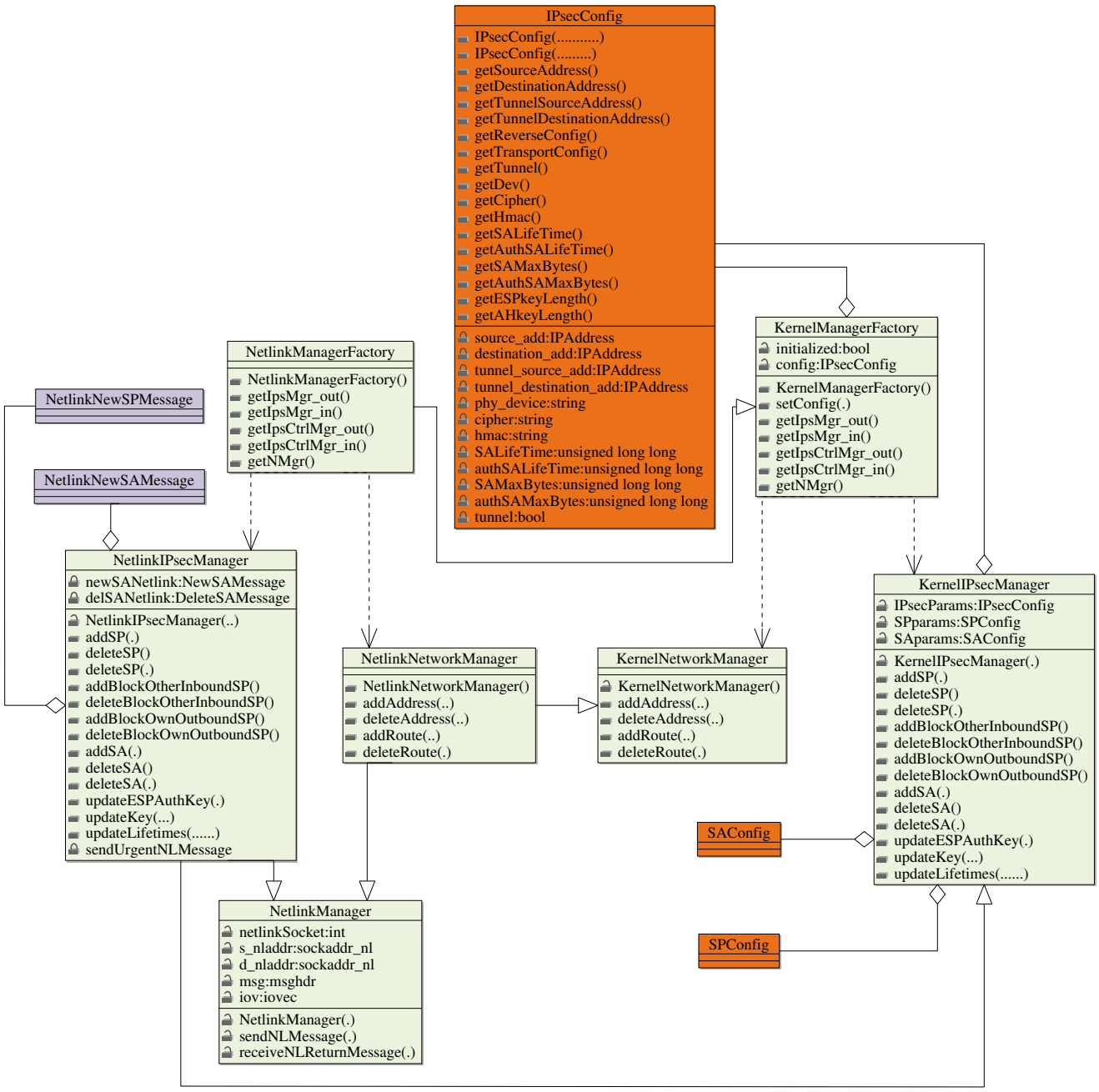


Figure 7.2: UML Diagram of the IPsec Kernel Manager Factory Environment

and destination address. This constructor automatically fabricates a new message for an IPsec tunnel mode SP. For SAs, there is no such distinction as they do not contain the tunneling addresses, instead the constructor accepts an additional boolean value that determines whether the SA is in tunnel mode.

As the key length inside the structure is unknown a priori, the message reserves enough memory for the allowed maximum and sets the message length according to the actually needed key size in situ, effectively truncating the message. Apart from that, the requirement to im-

pose both authentication and encryption on the data channel adds extra complexity to the structure. The structures for single (encryption or authentication) and combined algorithms differ slightly in that combined ones possess an additional field for the Integrity Check Value (ICV) length before the key. This problem is solved by using a C++ *union*. In case of single algorithms, the SA needs two payloads, one for encryption and one for the authentication algorithm. Because of the unknown key length, the alignment of the second structure cannot happen statically. Therefore a buffer for a second payload follows directly after the first one. After the construction of the first payload, the second structure is constructed and then copied to immediately follow the first one. By this procedure, an SA message is flexible enough to hold one single (for the control channel) or two single or a combined (for the data channel) cryptographic algorithm including its variably sized key.

An own class (*SPConfig*) delivers the remaining parameters to create or delete an SP (direction, source and destination ports, port masks, IPsec protocol and the SPI) to the constructors of the SP-processing messages, as the *NetlinkManager* itself may use this construct to keep this parameters. For convenience *SPConfig* returns an all-one port mask if the mask is not set, while a port is. This means that setting a port without mask yields a single-port restriction. If this had been implemented differently, setting a port without mask would have had no effect, as a default (all-zero) port mask does not restrict the port range at all. The SP configuration also holds a parameter that determines the action of the SP, for QKDIPsec uses two occasions to *discard* (see section 3.1.1) traffic via an SP. Firstly to block traffic to the QKDIPsec, that does not come from the QKDIPsec peer (or *Bob*) and secondly to impose outbound blocks for the data channel traffic if no QKD key is available and during an ongoing or after a failed reset process (see section 5.4.5). With the class *SAConfig*, SA messages also have their respective configuration class containing the Key, Cipher, SPI, Soft_byte_limit, Hard_byte_limit, Soft_packet_limit and Hard_packet_limit as encapsulated parameters.

For *Netlink route* messages, the approach is more straightforward, for their respective *Netlink* messages exhibit a more uniform structure than the ones of XFRM. Therefore, it was possible to abstract some structures into the mediator classes *NetlinkRouteMessage* for routes and *NetlinkAddressMessage* for addresses to be applied to a network interface. These messages hold all the data necessary to fulfill their task, including a nested payload class that holds the (route destination or interface) IP address to be processed. The subclasses for adding and deleting the respective items (*NetlinkNewRouteMessage, NetlinkDeleteRouteMessage, NetlinkNewAddressMessage* and *NetlinkDeleteAddressMessage*) basically only set the message type within their constructor. An exception is *NetlinkNewRouteMessage* which needs a second payload for the gateway address and the message length and scope parameter must also be set newly in the *route* messages. This second payload can be set statically, much in contrast to *NetlinkNewSAMessage*. Figure 7.3 shows an UML class diagram of this part of the implementation.

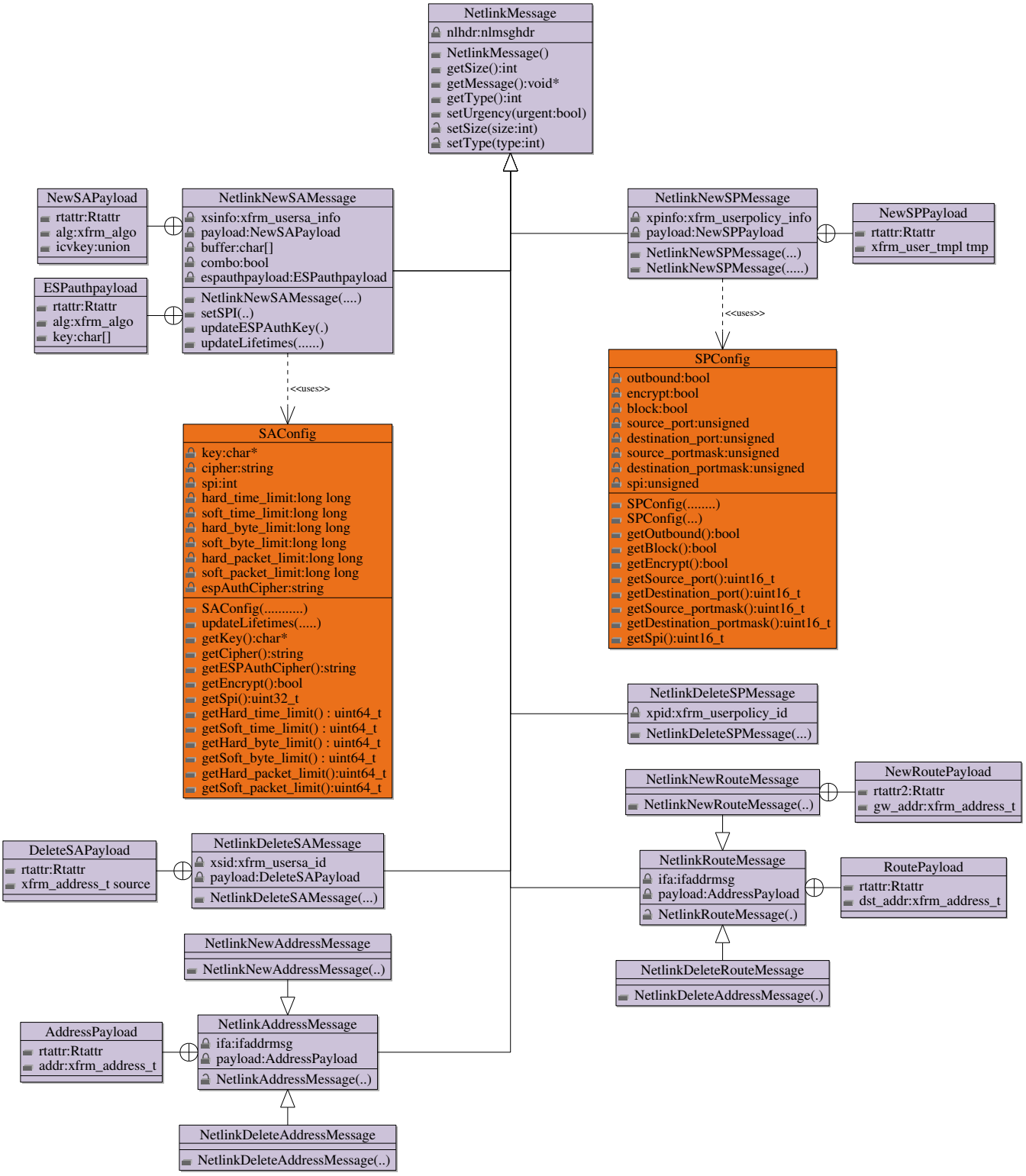


Figure 7.3: UML Diagram of Implemented Netlink XFRM Message Classes

7.4 Utility Classes

The given class setup also features a class for IP addresses (*IPAddress*), which, constituting from an address string in dotted decimal (for Internet Protocol version 4 (IPv4) addresses) or coloned hexadecimal (for Internet Protocol version 6 (IPv6) addresses) notation, provides the address for the Netlink API in the needed form (see section 7.6). To achieve this, the class uses *boost::asio::ip::address* class to convert the string into a number and specify whether the address is a IPv4 or IPv6 address. It then converts the address in to the *big-endian* format and fills a *xfrm_address_t* structure. Appropriate *getter* functions make the address and family accessible to other classes. The class *Utils* provides functions to convert numbers between the network (*big endian*) and x86 (*little endian*) formats, as well as a function to deliver a Linux error message to a given error code. There is also a (yet experimental) function to expand a key through duplication to save key material in non-critical operations. An own class *Exception*, derived from *std::exception* enriches the latter with an *integer* error code and defines message types for different QKDIPsec errors (see table 7.1). The defining *header* file, contains also an own exception class for each error. The advantage in defining an exception type and no just specialized exceptions alone, is that if the class *Exception* is caught, the type may be determined in a simple *switch* statement. The class *NLReturnException* returns additionally the error code supplied by *Netlink* and its error message with the help of the *Utils* class.

Error Type	Error Code	Exception
GENERIC_ERROR	0	Exception
NLSEND_ERROR	100	NLSendException
NLRETURN_ERROR	101	NLReturnException
IP_ADDRESS_ERROR	200	IPAddressException
MGR_FAC_ERROR	201	ManagerFactoryException
KEY_CONFIG_ERROR	203	KeyConfigException
IPSEC_ERROR	300	IPsecManagerException
IPSEC_KEY_ERROR	301	IPsecKeyException
IPSEC_KEYSYNC_ERROR	302	IPsecKeySyncException
CONNECTION_ERROR	400	ConnectionError
SOCKET_ERROR	401	SocketError
BIND_ERROR	402	BindError
ACCEPT_ERROR	403	AcceptError
SERVER_NOT_FOUND_ERROR	404	ServerNotFoundError
CONNECT_ERROR	405	ConnectError
QKD_ERROR	500	reserved for QKD engine errors
QKD_NOKEY_ERROR	501	QKDKeyException

Table 7.1: QKDIPsec Exception Types and Error Codes

The class *Cipher* provides all necessary information (key length, Initialization Vector (IV) length and capabilities - encrypt, authenticate or both) to a cryptographic algorithm identified by name in the constructor. It further provides the desired *Netlink* name (for instance *Netlink* expects *hmac(sha1)* as a name value for Secure Hash Algorithm (SHA)1). The class distinguishes between different key length versions of the same algorithm through a name supplement in the constructor. As *Netlink* does not make that distinction by name, the class

returns the same *Netlink* name but a different key length for a different algorithm version. For instance, the class might get *aes-128* or *aes-256* and input but returns in both cases the algorithm name *aes*, but a key length of 128 or 256, respectively. As these classes are used by the majority of the other classes, figure 7.4 pictures these classes without context for clarity reasons.

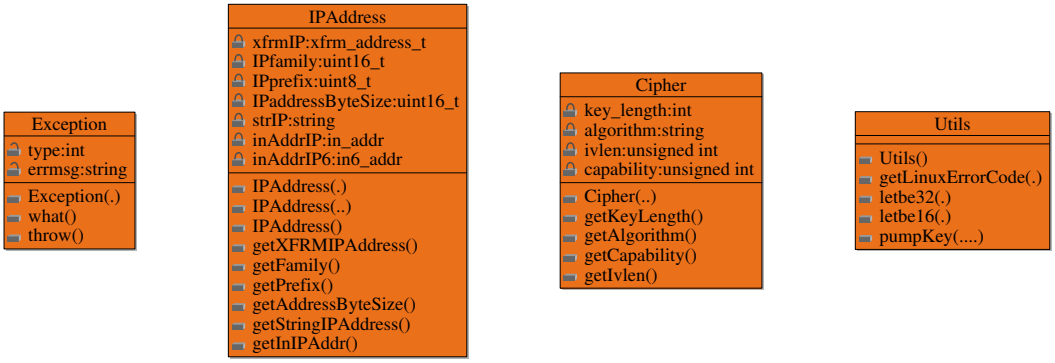


Figure 7.4: UML Diagram of Implemented Netlink XFRM Message Classes

In addition to the utility classes, there is a header file named *NLconstants* that aggregates all necessary *defines* used by QKDIPsec.

7.5 Key Manager and Integration into the AIT QKD Software

The relation between the AIT QKD software and QKDIPsec is twofold:

- The QKD software calls and controls the module;
- The module calls the QKD engine to acquire keys.

The first direction of the relation happens via the connection manager that provides instantiation and control mechanisms for any external user. The opposite direction is the responsibility of the key manager (class *KeyManager*).

As stated before, this solution momentarily simulates the delivery of QKD keys with help of a ring buffer filled with keys. Prerequisite for the full integration of QKDIPsec into the AIT QKD system is the opportunity to acquire keys per application identifier from the latter. This functionality is on the schedule of AIT QKD development, although it is has not been finished yet. For this reason the actual integration is out of scope of this thesis. The integration is, however, prepared in the key manager and its configuration class. The latter will be the handoff point for QKD engine, which ultimately delivers the QKD keys. This handoff will happen via the parameter `qkd::q3p::engine_instance Q3PEngine` that points to the actual engine. The QKD software will set this parameter in the key configuration class, itself a constructor parameter of the connection manager. The connection manager will then pass

this pointer to the *Q3PEngine* to the key manager on construction. The key manager, then with direct access to the QKD engine will call the following function[4]:

```
acquire_keys(qkd::key::key_ring & cKeys, uint64_t nAppId,↩
   uint64_t> nBytes, std::chrono::milliseconds nTimeout);
```

Listing 7.1: Q3P key acquisition Signature

Out of the *key_ring* it will then get the actual keys and return it as a *char**. The key length is a parameter in *getNextKey* function of the key manager and will then (converted from bits to bytes) be passed to the *acquire_keys* function. If the function will not return a key, the key manager will throw a *QKDKeyException*. The solution is (currently and in future) programmed to repeat the key fechting process until an actual key is derived (which means that no exception occurs). Similarily, the inital SPI and the salt for the subsequent SPIs will be derived from the QKD engine. The hash-derived calculation of further SPIs will remain unchanged. In summary, the key manager simulates key generation as long as the key acquisition based on application identifier does not yet work on side of the AIT QKD software. When this is done, the key manager will serve as a mediator between the QKD software and QKDIPsec (or more precisely, the connection manager), implementing a *strategy pattern* (Freeman et al., 2004, p.243). All changes for integration have to be made solely in the classes *KeyManager* and *KeyConfig*.

7.6 IPv6 Considerations

Although this solution was designed to ultimately support IPv6 in future, this is not fully implemented yet. The class *IPAddress* is IPv6 aware and returns the correct address family (*AF_INET* or *AF_INET6*) to be added injected into the kernel. Also, the whole IPsec and interface addressing parts fully support IPv6. Yet lack of support for the new IP are the routing parts and the *socket* connections, which will be in the focus of future work.

[4]This is the yet pending prerequisite function for the integration.

8 Testing and Configuration

This chapter focuses on the practical capabillities of QKDIPsec. It documents the testing process of this solution and further gives advice on how to configure it for field-use.

8.1 Test Hardware

Development and testing took place on two Personal Computer (PC) systems running on GNU/Linux. The first test and development system (henceforth called *Alice*) was a notebook with a 2 GHz Intel Pentium M processor, 1 GB RAM running on Fedora 18, kernel 3.11.7. The second machine (henceforth called *Bob*) was a PC with an Core 2 Duo E6300 processor (dual core operating at 1.866 GHz), 2 GB RAM and Fedora 18, kernel 3.11.9. The Linux versions were not updated during the software development and testing process to provide a stable testing environment. Both systems were more than six years old (although equipped with a more recent operating system) and thus could be seen as legacy systems. Tests were conducted in a Local Area Network (LAN) environment as well as via a mobile internet connection and an Transport Layer Security (TLS) Virtual Private Network (VPN). The latter part (together with the legacy status of the testing systems) demonstrates the robustness and performance of the solution by running it under suboptimal, significantly less than state of the art conditions.

8.2 Early Proof of Concept Testing

A crucial part of this work was to determine whether the proposed system could change the Internet Protocol security (IPsec) key with the required frequency (see section 5.5.1). Therefore, prior to any software implementation tasks, a proof of concept testing conducted on *Alice* determined which key rates a *rapid rekeying* system could achieve. The test consisted of a *setkey* configuration, which sequentially added and removed an IPsec Security Association (SA) 10,000 times (except for the last time, where it remained in the database) , plus a different, stable SA to get a beginning time stamp. After the completion of the *setkey* execution the comparison of the two timestamps resulted in a timespan which the execution took to complete, in this particular case 88 seconds. As the chosen algorithm was *Blowfish* with a key length of 488 bytes (the longest possible for encryption within the IPsec standard - see section 3.5.2), this result equals to a key rate of approximately 55,500 key bits per second.

These tests were later repeated again using a *setkey* script, but also with the present implementation and a script conducting the same test using the *ip xfrm* command, this time both

on *Alice* and *Bob*. Due to the timespan in between the three test series, the used Linux kernel versions range from 3.9 to 3.11[1] on both machines. Table 8.1 shows the results of these subsequently conducted tests.

Machine	duration in seconds / keys per second		
	setkey	ip xfrm	QKDIPsec (present solution)
Alice	104/96.15	103/97.09	105/95.24
Bob	86/116.28	83/120.48	84/119.05

Table 8.1: Rapid Rekeying Proof of Concept Test Results

The first test - on *Bob*- yielded key rates of approximately 53,000 key bits per second, while the notebook achieved around 10,000 less. The differences between the three methods for rekeying are not considered as being significant. These test included in each case a creation and a deletion of an IPsec SA. Although there are update messages, it is not possible to update key, for the kernel does not process new keying material for existing SAs. (Egerer, 2013) Further tests showed, however, that deleting messages is apparently more time-consuming than creating ones. While on *Alice* it took 105 seconds with this method, it took only 19 seconds to complete when only establishing SAs, yielding a rate of 526 keys per second, but this method has some disadvantages (see section 5.4.2 for considerations regarding this method).

8.3 Test Network Setup

The environment to test the actual solution consisted of two different network configurations. The first one was a fully switched LAN between *Alice* and *Bob* with an additional sniffer notebook (henceforth called *Eve*) on *Bob*'s side. The sniffer notebook was attached to a network port on the same switch mirroring all traffic from and to *Bob* (see figure 8.1).

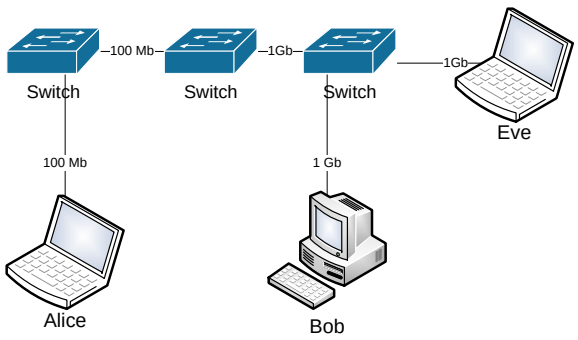


Figure 8.1: Local Area Network (LAN) Test Setup

[1] As stated before, the kernel versions were frozen on 3.11 during software development. However, during the prior proof of concept phase they were not.

Eve was running a *Wireshark* sniffer[2] on *Windows 7* to verify the network traffic between *Alice* and *Bob*. The second setup consisted of an environment to test the Wide Area Network (WAN) capabilities of the solution. In this setup, *Alice* established an TLS-secured VPN tunnel via a mobile Internet connection. From this point the connection proceeded via a switched LAN to *Bob*, again with *Eve* at his side. Figure 8.2 depicts this configuration.

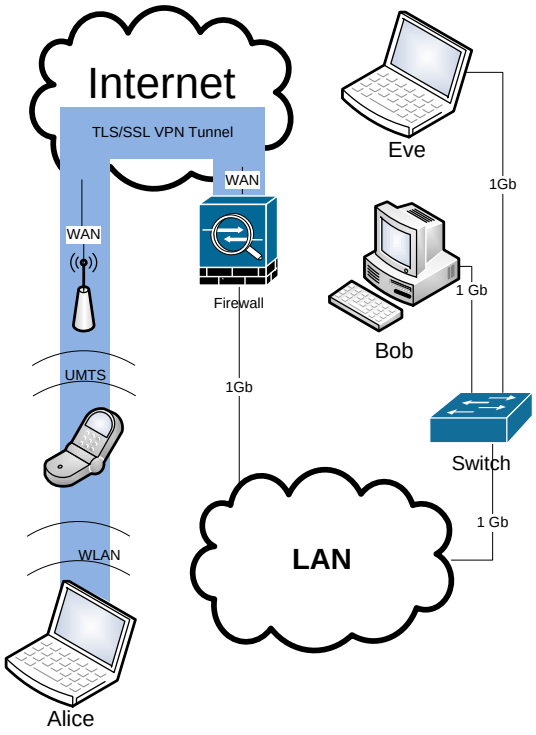


Figure 8.2: Wide Area Network (WAN) Test Setup

8.4 Performance Tests

Performance tests were conducted in the LAN and WAN environment. In both cases the tests consisted in the copying of a file via SSH File Transfer Protocol (SFTP) from *Alice* to *Bob* and vice versa, while running *ping* simultaneously. Both filecopies and ping were started from *Alice*. Due to bandwidth and time constraints, the testing file for WAN was significantly smaller (1.813.904 bytes) than the LAN file (69.533.696 bytes). The tests were repeated multiple times with different QKDIPsec settings, as well as with *standard* IPsec (encrypted with no key changes) and completely without IPsec. In summary, the unencrypted copies were expectedly the fastest from *Alice* to *Bob* in LAN environment, while QKDIPsec took nearly double time (the key frequency had only insignificant impact) but standard IPsec was the slowest (including a statistical outlier). In the WAN environment, the file copies all finished within a one second margin. The difference was thus not significant. Table 8.2 shows these test results

[2] `http://www.wireshark.org/`

in seconds, each try within one setting separated by a slash and the result of simultaneous ping in percent. The Advanced Encryption Standard (AES)-Cipher Block Chaining (CBC) and *Blowfish* tests were conducted with encryption only. All tests used a controll channel key period of three seconds with Secure Hash Algorithm (SHA)-256-Message Authentication Code (MAC) as authentication method, as this is the solution's default.

Setting	LAN			WAN		
	A→B	B→A	Ping	A→B	B→A	Ping
unencrypted	6/6/7/6	7/9/7/8	100%	10/10/10/10	9/7/6/7	99%
AES-256 CCM						
standard IPsec	14/14/16/15	17/18/26/18	100%	11/11/11/11	11/5/6/5	99%
50 ms	8/10/8/9	14/16/16/16	100%	14/10/11/13	6/5/5/5	95%
25 ms	10/9/8/8	14/15/17/16	100%	10/11/10/10	6/7/6/7	94%
20 ms	9/9/9/9	11/16/17/12	100%	12/11/13/10	9/5/6/6	98%
AES-256 CBC						
20 ms	9/7/7	11/13/17	100%	10/11/11	9/7/8	100%
Blowfish-448						
20 ms	14/9/7	15/13/14	99%			

Table 8.2: Performance Test Results

These results show that the solution is able to perform with various key change frequencies, both in LAN and WAN. During these tests, *Eve* recorded the actual network traffic to verify the key changes. This is possible by keeping track of the IPsec Security Parameters Index (SPI) changes of the packets. Table 8.3 shows a random sample of key change periods in milliseconds during the above mentioned LAN 20ms AES-256-CCM test. The first big column shows Encapsulating security Payload (ESP) packets from *Alice* to *Bob* while the second shows the opposite direction. As the recorded data contains one file copy from *Alice* to *Bob* (in the first half of the record) and one vice versa (in the second half), one randomly chosen sample of five consecutive key changes for each direction and from each half is chosen. This is because in each half one of the peers sends significantly less data than the other, as it only acknowledges the received packets from the file copy. The total average of these values is 0.020495 ms, which is approximately 2.5% above 20 ms per key change. This may be explained by the send and receive overhead for processing the key change messages, for the period is determines only the sleeping duration of a sender thread (see section 7.1).

The results for WAN cannot be determined by the same method, beause on the one hand there is not enough traffic to get exact time readings and on the other hand network latency adds a degree of vagueness to them. Therefore, using the same random choosing method of taking one sample for each half and each traffic direction, for each sample 20 key change periods where summed up and averaged. The total average over these samples is approximately 0.2475, which is significantly higher (approximately 19%) than in the LAN setting. This may be caused by latency in this environment.

	A→B		B→A	
	1st	2nd	1st	2nd
LAN	0.0220	0.0216	0.0208	0.0203
	0.0187	0.0204	0.0197	0.0235
	0.0145	0.0216	0.0203	0.0176
	0.0195	0.0243	0.0204	0.0197
	0.0225	0.0180	0.0207	0.0238
Ø	0.0194	0.0212	0.0204	0.0210
WAN $\sum 20$	0.5201	0.4899	0.4302	0.5397
Ø	0.0260	0.0245	0.0215	0.0270

Table 8.3: Network Sniffing Results

8.5 Endurance Tests

One test regarding the endurance of the solution was one that showed the capability of maintaining the connection over a longer period of time. The test was conducted with an earlier development version of QKDIPsec and ran in LAN environment over around 16 hours. It consisted of a running ping test on a 50 ms *Blowfish* configuration without control channel key changes. Of 56179 pings returned 56164 resulting in a return rate of approximately 99.97%. This test was also conducted in WAN environment, but (due to both tests ran overnight) an automated network connection reset after around eight hours prevented meaningful results.

To test the loss of single packets, *if* clauses to prevent the sending of particular (three consecutive) SPIs where temporarily added to the code. The testing code was subsequently run on *Alice*, simultaneously with a ping to test the reachability of *Bob*. While these SPIs where successfully installed on *Alice*, on *Bob* they were not because of the lacking notification. Due to the precalculation mechanism (see section 5.4.2) no ping was lost. The next sent SPI caused *Bob* to resynchronize the connection by post-installing the missed SPI. Two change cycles later, again by an *if* clause in *Alice*' code, the missing SPIs were sent to simulate excessive late key change packets, only to be dropped for they already had been installed. Both behaviors therefore worked according to the scheme in section 5.4.4.

An additional test withheld (by the same method as the tests above) more key change notifications than the queue size. More precicely, the queue size was set to 50 and *Alice* suspended key change messages for 50 change cycles after 200 sent regular ones. Expectedly, the connection reseted (by a master reset) and the connection started from the beginning, resulting in a reset approximately every 12.5 seconds. Despite permanent connection resets in the direction from *Alice* to *Bob*, simultaneous ping tests on two test series both yielded only marginal losses (99.74% and 99.36% success rate).Furthermore, it was possible to unimpededly copy the reference file in both directions.

The last test was actively severing the network connection. Pulling the plug on one side resulted in a connection loss that was only recoverable by executing the connection setup routine. This normally does not occur automatically in QKDIPsec but can be induced by the calling function (ordinarily the Austrian Institute of Technology (AIT) Quantum Key Distri-

bution (QKD) software). The cause for this behavior is that a shut down (or connectionless) interface loses its additional Internet Protocol (IP) addresses and therefore the tunnel address for the data channel. This problem might be circumvented by implementing an own virtual interace in the future. When servering the connection along the path (thus leaving the peer interfaces intact) the solution automatically recovered (loosing only traffic during the servered phase) when reconnected timely or entered the reset procedures (reset trial and function suspension on time) on disruption spanning over more than the timeout period, according to protocol.

8.6 Test Summary

The results show sufficient performance of the solution with a 256 bit cipher, given the goals specified in section 5.5.1. The shown stable key period of less than 25 ms (=40 keys per second) yields a a key injection rate of greater than 10240 key bits for each direction totaling in 20480 key bits opposed to 12500 as miminum required. A 128 bits key would therefore yield 10240 key bits per second, $\approx 19\%$ less than the goal of 12500 (which is logical as the goal with a 128 bit key would exactly be a key period of 20 ms). This second result, however, is less significant, for the use of a 256 bit key is recommended (see section 5.5). Furthermore, section 8.2 suggests that the hardware performance on *Alice* might not be enough to meet this requirements (95.24 key changes per second with no other running programs impacting the performance opposed to 100 key changes per minute as a requirement). For these reasons the results of these test are deemed sufficient. The endurance test all showed sufficient results, meaning that the solution behaved according to the protocol in section 5.4. Only an interface shutdown yielded in a state, unrecoverable by the solution itself. This condition may be prevented by a future planned virtual interface.

8.7 Configuration Reference

The setup of QKDIPsec occurs through instantiation of the class *ConnectionManager* by the according configuration classes (*IPsecConfig, KeyConfig, KeySyncConfig*[3]). The key manager configuration contains the application identifiers (and in future the QKD engine instance), that have to be provided by the calling instance. The key synchronization configuration contains the data channel (default: 50 ms) and control channel key change (default: 3s) frequencies, the data (default: 400) and control (default: 7) inbound queue sizes (the respective outbound sizes are automatically the half of the inbound ones) and the reset timeout (default: 10s). Without the presence of any performance problems, changes to this default should only be made on the data channel key frequency to set it according to the desired key period or Data Bits per Key Bit (DPK) rate, respectively.

The IPsec configuration contains the necessary IPsec connection paramaters (physical and tunnel IP address and the network interface) and the IPsec security parameters (ciphers and SA lifetimes). The default ciphers in QKDIPsec are AES-256 in Counter with CBC-MAC

[3] Also, an other *KernelManagerFactory* than the default *Netlink*

(CCM) mode for the data channel and SHA-256-MAC as these are deemed secure regarding to N. Smart (2012, p.89). These settings are also the recommended ones. A set SA time limit *must* correspond to the key change rates and queue sizes (with queue size times key change rate as lower boundary), while a byte limit *must* correspond to used key length (of the cypher or authentication algorithm) times the maximum data bits per key bits rate (see section 5.5.1 for this value). For the reason described in section 5.4.5 (and also to prevent SAs to remain in the system on the event of an unclean shutdown) it is recommended to set SA timeouts (although the default is no timeouts).

In order for QKDIPsec to run properly the peer's firewalls (and additional ones within the path) have to be adjusted properly[4] to allow ESP traffic and the traffic according to the chosen ports for the control channel (set in the *start_controlChannelListener()* and *start_controlChannelSender()* methods within the connection manager). To build the solution, the following flags where in the development machines' C++ settings:

- Compiler: *-O0 -g3 -pedantic -Wall -Wextra -Werror -c -fmessage-length=0 -std=gnu++0x -pthread*
- Linker: *-lboost_system -pthread -lcrypto*

[4]For the solution tests, the firewalls on *Alice* and *Bob* have been deactivated with the command *systemctl stop firewalld.service*

9 Conclusion

This thesis shows a possible solution to add Internet Protocol security (IPsec) functionality to the AIT QKD software and offers an approach to use IPsec with unconditionally secure derived keying material through Quantum Key Distribution (QKD). To achieve this, it contains a key *rapid rekeying* synchronization protocol that exploits the unique situation of using QKD; to be able to encrypt data without the need to take care of a dedicated key exchange but merely synchronizing quantum keys. Therefore this solution is able to realize IPsec without the use of Internet Key Exchange (IKE). The protocol was subsequently implemented, using the Linux C++ Application Programmer Interface (API) *Netlink*, which proved to be suitable for this task. The resulting software solution, QKDIPsec, was comprehensively and successfully tested with key periods as short as 20 ms in Local Area Network (LAN) and Wide Area Network (WAN) environments.

The presented research also showed that this cryptographic period is more than sufficient for the use of IPsec with today's QKD systems and therefore provides a practical maximum key rate for IPsec with quantum keys. It also showed that a cryptographic period of one minute is practically sufficient for highly secure applications today and in forseeable future and discussed sensible boundaries in terms of Data Bits per Key Bit (DPK) based on the *birthday bound* and resemblance to One-Time Pad (OTP). It further recommends a key length of 256 bits and the use of the Advanced Encryption Standard (AES)-Counter with CBC-MAC (CCM) cipher for encryption.[1]

Due to pending requirements to the Austrian Institute of Technology (AIT) QKD software, QKDIPsec has yet to simulate the integration of QKD keys into IPsec. Nevertheless, this thesis shows the possibility of such an enterprise and paved the road for its implementations by providing complete IPsec functionality and an adequate interface to QKD.

9.1 Outlook

Logical further work based on this thesis is the full integration into the AIT QKD software and subsequent tests with actual QKD keying, proving the assumptions of this thesis under real conditions. Ultimately, the goal of integration will be a to turn QKDIPsec into a market-ready solution. Therefore, meeting customer's demands, a further improvement will include the use of virtual interfaces as tunnel endpoints. To simplify the enforcement of security goals, the solution may support the reading of (virtual) interface counters in future in order to automatically adjust the key change frequency and Security Associations (SAs) time and byte limits to achieve the desired DPK rate (see section 5.3). To provide better support for

[1] See section 5.5

embedded devices the solution might support asymmetric queue structures to save memory. On the other hand, it might implement an alternative approach for updating keys without the deletion of SAs (see section 5.4.2). To prevent the threat of possible disruption of the *Netlink* message data structure under certain circumstances, the memory alignment of the *Netlink* message classes might be overhauled. Lastly, full support the advancing Internet Protocol version 6 (IPv6), is intended to be implemented.

Glossary

ε-ASU$_2$ ε-almost strongly universal$_2$. 7, 8, 25

0MQ Zero Message Queue. 53

AES Advanced Encryption Standard. 46, 48, 54, 70–72, 98, 100, 102

AH Authentication Header. 34–40, 43, 44, 48, 58, 63, 64, 71, 72, 79, 109, 111

AIT Austrian Institute of Technology. 1, 2, 4, 30, 31, 50–52, 55, 56, 58–60, 69, 84, 93, 94, 99, 102, 109

API Application Programmer Interface. 50, 54, 73–75, 77, 82, 84, 86, 92, 102

AQUA Advancement in Quantum Architecture. 56

ARC Austrian Research Centers. 55

B92 *Bennett 1992*. 21

BB84 *Bennett/Brassard 1984*. 18–24, 30, 31, 50, 109

BBM92 *Bennett/Brassard/Mermin 1992*. 20, 50

BICM Binary Interleaved Code Modulation. 29

bit Binary Digit. 1

BSC Binary Symmetric Channel. 9, 28

BSD Berkeley Software Distribution. 50, 54, 74

CAST *Carlisle Adams/Stafford Tavares*. 46

CBC Cipher Block Chaining. 46–48, 70, 72, 98, 109, 113

CCM Counter with CBC-MAC. 47, 72, 98, 100, 102

CTR Counter. 47, 70, 72, 109, 113

CVE Common Vulnerabilities and Exposures. 73, 74

DBUS Desktop Bus. 30, 31

DES Data Encryption Standard. 46, 48

DF Don't Fragment. 33

DH *Diffie/Hellman*. 5, 43–45, 55, 56, 71

DPK Data Bits per Key Bit. 70, 100, 102

DSCP Differentiated Services Codepoint. 33, 34

E91 *Ekert 1991*. 19–22, 109

ECB Electronic Code Book. 45

ECN Explicit Congestion Notification. 40

EH Extension Header. 35, 36, 39, 40

EPR Einstein-Podolsky-Rosen. 16, 30

ESP Encapsulating security Payload. 34, 35, 37–40, 43–48, 58, 70–72, 79, 81, 98, 101, 109, 111

FIB Forwarding Information Database. 75

FIFO First In First Out. 63

GCC GNU Compiler Collection. 50, 52

GCM Galois/Counter Mode. 47, 49, 72

GMAC Galois Message Authentication Code. 49

GNU GNU's Not Unix. 52, 95

HMAC Hashed Message Authentication Code. 37, 38, 48, 49, 113

IANA Internet Assigned Numbers Authority. 36, 44, 45

ICMP Internet Control Message Protocol. 83

ICV Integrity Check Value. 36–38, 72, 81, 90

IDEA International Data Encryption Algorithm. 46

IEC International Electrotechnical Commission. 52

IETF Internet Engineering Task Force. 45, 49, 74

IKE Internet Key Exchange. 3, 41–45, 50, 54–56, 59, 60, 102

IKEv2 Internet Key Exchange version 2. 41–44, 109, 111

IOV Input-Output Vector. 77, 88

IP Internet Protocol. 32–40, 55, 57, 58, 60, 74, 75, 79, 82, 85, 87, 88, 90, 92, 94, 100, 109

ipad Inner Padding. 49

IPC Inter-Process Communication. 75

IPsec Internet Protocol security. 1–3, 32–36, 39–46, 48, 50–52, 54–64, 68–76, 78, 79, 81, 82, 84–90, 92–102, 109, 111, 113

IPv4 Internet Protocol version 4. 75, 82, 83, 92

IPv6 Internet Protocol version 6. 35, 36, 39, 40, 75, 82, 83, 92, 94, 103

ISAKMP Internet Security Association and Key Management Protocol. 41, 54, 55

ISO International Organization for Standardization. 52

IV Initialization Vector. 37, 38, 46, 92

KLIPS Kernel IPsec. 74

LAN Local Area Network. 95–99, 102, 110

LDPC Low-Density Partiy Check. 9, 10, 29–31

LIFO Last In First Out. 61

MAC Message Authentication Code. 36, 44, 47, 48, 98, 101

MD Message Digest. 48, 49

MLC/MSD Multilevel Coding / Multistage Decoding. 29, 30

MTU Maximum Transmission Unit. 34

NIST National Institute of Standards and Technology. 37, 44, 72

NSA National Security Agency. 44

NVC Nitrogen Vacancy-Center. 17

OFB Output Feedback. 70

opad Outer Padding. 49

OTP One-Time Pad. 1, 4–6, 54, 55, 59, 70, 72, 102

PC Personal Computer. 95

PF_KEY Protocol Family Key. 74, 86

PF_ROUTE Protocol Family Route. 74

PPP Point-to-Point Protocol. 31

PRNG Pseudorandom Number Generator. 72

Q3P Quantum Point-to-Point Protocol. 31, 59

QBER Quantum Bit Error Rate. 26

QKD Quantum Key Distribution. 1–4, 6, 8, 10, 12, 15–18, 23–25, 29–31, 50–52, 54–61, 63, 66–73, 84, 85, 88, 90, 92–97, 99–102, 109, 111

qubit Quantum Bit. 11, 12, 15, 18–26, 30

RC Rivest Cipher. 45, 46, 48

RFC Request for Comments. 37, 44

RIPEMD RACE Integrity Primitives Evaluation Message Digest. 48

RSA *Rivest/Shamir/Adlemann*. 5

SA Security Association. 32–34, 36, 41–43, 45, 55, 56, 58, 60–64, 66–69, 74, 81, 85, 86, 88–90, 95, 96, 100–103, 109

SAD Security Association Database. 32, 34, 35, 50, 51, 55, 57, 58, 63, 66, 74, 76, 78, 81, 86

SARG *Scarani/Acín/Ribordy/Gisin*. 23, 24

SEQKEIP Secure Quantum Key Exchange Internet Protocol. 55

SFTP SSH File Transfer Protocol. 97

SHA Secure Hash Algorithm. 48, 49, 60, 92, 98, 101

SP Security Policy. 56, 68, 79, 80, 85, 86, 88–90

SPD Security Policy Database. 32, 33, 50, 51, 57, 58, 74, 76, 78–81, 86

SPI Security Parameters Index. 34, 36–38, 41, 42, 51, 55, 60–68, 79, 81, 82, 86, 90, 94, 98, 99, 113

SSP Six-State Protocol. 22

SU_2 strongly universal$_2$. 7

TCP Transmission Control Protocol. 58, 59, 75, 77

TFC Traffic Flow Confidentiality. 37, 38

TLS Transport Layer Security. 95, 97

TLV Type-Length-Value. 79, 81–83, 88

TTL Time-to-Live. 35, 40

UDP User Datagram Protocol. 41

UML Unified Modeling Language. 77, 84, 88, 90

VPN Virtual Private Network. 44, 55, 95, 97

WAN Wide Area Network. 97–99, 102, 110

XCBC Extended Cipher Block Chaining. 48

XFRM Transform. 50, 73, 74, 78, 79, 81, 82, 88, 90

XOR Exclusive Or. 5, 9, 46–49, 55

List of Figures

List of Tables

List of Equations

References

Adams, C. (1997). *RFC2144: The CAST-128 Encryption Algorithm.* Internet Engineering Task Force.

Austrian Institute of Technology. (2013). *Einstein-Podolski-Rosen Entangled Photon Pair System EPR SYS-405.* Retrieved from http://www.ait.ac.at/research-services/research-services-safety-security/optical-quantum-technology/einstein-podolski-rosen-entangled-photon-pair-system-epr-sys-405/?L=1 (retrieved at August 26, 2013)

Baldwin, R., & Rivest, R. (1997). *RFC2040: The RC5, RC5-CBC, RC5-CBC-Pad, and RC5-CTS Algorithms.* Internet Engineering Task Force.

Barker, E., & Barker, W. (2012). *Recommendation for the Triple Data Encryption Standard (TDES) (Revision 1 - NIST Special Publication 800-67).* National Institute of Standards and Technology. Retrieved from http://csrc.nist.gov/publications/nistpubs/800-67-Rev1/SP-800-67-Rev1.pdf (retrieved at March 12, 2014)

Barker, E., Barker, W., Burr, W., Polk, W., & Smid, M. (2012). *Recommendation for Key Management – Part 1: General (Revision 3 - NIST Special Publication 800-57).* National Institute of Standards and Technology. Retrieved from http://csrc.nist.gov/publications/nistpubs/800-57/sp800-57_part1_rev3_general.pdf (retrieved at March 12, 2014)

Barker, E., Burr, W., Jones, A., Polk, T., Rose, S., Smid, M., & Dang, Q. (2012). *Recommendation for Key Management – Part 3: Application-Specific Key Management Guidance(Revision 3 - NIST Special Publication 800-57).* National Institute of Standards and Technology. Retrieved from http://csrc.nist.gov/publications/nistpubs/800-57/sp800-57_PART3_key-management_Dec2009.pdf (retrieved at March 12, 2014)

Bausewein, H. (2012). *Debian Wiki - IPsec.* Retrieved from https://wiki.debian.org/IPsec (retrieved at December 9, 2013)

Bell, J. S. (1964). On the Einstein Podolsky Rosen paradox. *Physics*, *1*(3), 195-200.

Bennett, C., Brassard, G., Crepeau, C., & Maurer, U. (1994). Generalized privacy amplification. In *Information theory, 1994. proceedings., 1994 ieee international symposium on* (p. 350-).

Bennett, C., Brassard, G., Ekert, A., Fuchs, C., & Preskill, J. (2004). Summary of the Theory Component of Quantum Key Distribution and Quantum Cryptography. In T. Heinrichs (Ed.), *A Quantum Information Science and Technology Roadmap Part 2: Quantum Cryptography.* Advanced Research and Development Activity. Retrieved from http://qist.lanl.gov/pdfs/whole_roadmap.pdf (retrieved July 24, 2013)

Bennett, C. H. (1992, May 25). Quantum Cryptography Using Any Two Nonorthogonal States. *Physical Review Letters*, *68*, 3121-3124.

Bennett, C. H., & Brassard, G. (1984). Quantum Cryptography: Public Key Distribution and Coin Tossing. In *Proceedings of the IEEE International Conference on Computers, Systems*

and Signal Processing (p. 175-179). Bangalore: Intitute of Electrical and Electronical Engineers.

Bennett, C. H., Brassard, G., Jozsa, R., Mayers, D., Peres, A., Schumacher, B., & Wootters, W. K. (1994). Reduction of Quantum Entropy by Reversible Extraction of Classical Information. *Journal of Modern Optics*, *41*(12), 2307-2314.

Bennett, C. H., Brassard, G., & Mermin, N. D. (1992, February). Quantum cryptography without bell's theorem. *Physical Review Letters*, *68*, 557-559.

Berzanskis, A., Hakkarainen, H., Lee, K., & Hussain, M. R. (2009). *Secret Signaling System* (Patent No. US 7602919).

Bethune, D., & Elliott, C. (2004). Section 6.1: Weak Laser Pulses over Fiber. In T. Heinrichs (Ed.), *A Quantum Information Science and Technology Roadmap Part 2: Quantum Cryptography.* Advanced Research and Development Activity. Retrieved from `http://qist.lanl.gov/pdfs/whole_roadmap.pdf` (retrieved July 24, 2013)

Biham, E., Huttner, B., & Mor, T. (1996, October). Quantum Cryptographic Network based on Quantum Memories. *Physical Review A*, *54*, 2651-2658. Retrieved from `http://arXiv.org/abs/quant-ph/9604021v1` (preprint retrieved at August 18, 2013)

Biryukov, A., & Khovratovich, D. (2009). Related-key cryptanalysis of the full aes-192 and aes-256. In *Advances in cryptology–asiacrypt 2009* (pp. 1–18). Springer.

Black, J., & Rogaway, P. (2000). Cbc macs for arbitrary-length messages: The three-key constructions. In M. Bellare (Ed.), *Advances in cryptology — crypto 2000* (Vol. 1880, p. 197-215). Berlin Heidelberg: Springer.

Bloch, M., Thangaraj, A., McLaughlin, S. W., & Merolla, J.-M. (2006). Quantum Cryptography: Public Key Distribution and Coin Tossing. In *Proceedings of the IEEE Information Theory Workshop.* Punta del Este, Uruguay: Intitute of Electrical and Electronical Engineers. Retrieved from `http://arxiv.org/abs/cs/0509041v1` (preprint retrieved at August 22, 2013)

Braden, R. (1989). *RFC1122: Requirements for Internet Hosts - Communication Layers.* Internet Engineering Task Force.

Brandner, S. (1997). *RFC2119: Key words for use in RFCs to Indicate Requirement Levels.* Internet Engineering Task Force.

Brassard, G., & Salvail, L. (1994). Secret-key reconciliation by public discussion. In T. Helleseth (Ed.), *Advances in cryptology — eurocrypt '93* (Vol. 765, pp. 410–423). Berlin Heidelberg: Springer.

Bruß, D. (1998, Oct). Optimal Eavesdropping in Quantum Cryptography with Six States. *Physical Review Letters*, *81*, 3018-3021.

BSD. (2004). *setkey (8) - Linux manual page.* (command *man ip xfrm* on GNU/Linux; retrieved at September 25, 2013)

Cachin, C., & Maurer, U. (1997). Unconditional security against memory-bounded adversaries. In J. Kaliski B. S. (Ed.), *Advances in Cryptology — CRYPTO '97* (Vol. 1294, p. 292-306). Berlin, Heidelberg: Springer.

Carter, L., & Wegman, M. N. (1979). Universal Classes of Hash Functions. *J. Comput. Syst. Sci.*, *18*(2), 143-154.

Clauser, J. F., Horne, M. A., Shimony, A., & Holt, R. A. (1969, October). Proposed Experiment to Test Local Hidden-Variable Theories. *Physical Review Letters*, *23*, 880-884.

Deering, S., & Hinden, R. (1998). *RFC2460: Internet Protocol, Version 6 (IPv6) Specification.* Internet Engineering Task Force.

Dieks, D. (1982, November 22). Communication by epr devices. *Physics Letters A*, *92*(6),

271-272.

Diffie, W., & Hellman, M. E. (1976). New directions in cryptography. *IEEE Transactions on Information Theory*, *22*(6), 644-654.

Dirac, P. A. M. (1939, 7). A new notation for quantum mechanics. *Mathematical Proceedings of the Cambridge Philosophical Society*, *35*, 416-418.

Dworkin, M. (2001). *Recommendation for Block Cipher Modes of Operation (NIST Special Publication 800-38A).* National Institute of Standards and Technology. Retrieved from `http://csrc.nist.gov/publications/nistpubs/800-38a/sp800-38a.pdf` (retrieved at August 31, 2013)

Dworkin, M. (2004). *The CCM Mode for Authentication and Confidentiality (NIST Special Publication 800-38C).* National Institute of Standards and Technology. Retrieved from `http://csrc.nist.gov/publications/nistpubs/800-38C/SP800-38C_updated-July20_2007.pdf` (retrieved at March 15, 2014)

Dworkin, M. (2007). *Galois/Counter Mode (GCM) and GMAC (NIST Special Publication 800-38D).* National Institute of Standards and Technology. Retrieved from `http://csrc.nist.gov/publications/nistpubs/800-38D/SP-800-38D.pdf` (retrieved at March 15, 2014)

Eastlake, D., & Jones, P. (2001). *RFC3174: US Secure Hash Algorithm 1 (SHA1).* Internet Engineering Task Force.

Egerer, T. (2013). *How to best conduct a cipher key update?* Retrieved from `https://lists.strongswan.org/pipermail/dev/2013-November/000941.html` (Electronic mailing list message, retrieved at December 1, 2013)

Einstein, A., Podolsky, B., & Rosen, N. (1935, May). Can Quantum-Mechanical Description of Physical Reality Be Considered Complete? *Physical Review*, *47*, 777–780.

Ekert, A. K. (1991, August 5). Quantum Cryptography Based on Bell's Theorem. *Physical Review Letters*, *67*, 661-663.

Elliott, C., Pearson, D., & Troxel, G. (2003). Quantum cryptography in practice. In *Proceedings of the 2003 conference on applications, technologies, architectures, and protocols for computer communications* (pp. 227–238).

Enzer, D. G., Hadley, P. G., Hughes, R. J., Peterson, C. G., & Kwiat, P. G. (2002). Entangled-photon six-state quantum cryptography. *New Journal of Physics*, *4*(1), 45.

Fehr, S. (2010). Quantum Cryptography. *Foundations of Physics*, *40*(5), 494-531.

Frankel, S., & Herbert, H. (2003). *RFC3566: The AES-XCBC-MAC-96 Algorithm and Its Use With IPsec.* Internet Engineering Task Force.

Freeman, E., Robson, E., Bates, B., & Sierra, K. (2004). *Head First Design Patterns*. Sebastopol: O'Reilly.

Gallager, R. (1962). Low-Density Parity-Check Codes. *IRE Transactions on Information Theory*, *8*(1), 21-28.

Gilligan, R., Thomson, S., Bound, J., McCann, J., & Stevens, W. (2003). *RFC3493: Basic Socket Interface Extensions for IPv6.* Internet Engineering Task Force.

Gisin, N., Ribordy, G., Tittel, W., & Zbinden, H. (2002, Mar). Quantum cryptography. *Review of Modern Physics*, *74*, 145-195.

Glenn, R., & Kent, S. (1998). *RFC2410: The NULL Encryption Algorithm and Its Use With IPsec.* Internet Engineering Task Force.

Harkins, D., & Carrel, D. (1998). *RFC2409: The Internet Key Exchange (IKE.* Internet Engineering Task Force.

He, K. K. (2005, February). Why and How to Use Netlink Socket. *Linux Journal*(130), 21-28. Retrieved from `http://www.linuxjournal.com/article/7356` (retrieved at

October 11, 2013)

Heisenberg, W. (1927). Über den anschaulichen Inhalt der quantentheoretischen Kinematik und Mechanik. *Zeitschrift für Physik*, *43*(3-4), 172-198.

Hoffman, P. (2005). *RFC4308: Cryptographic Suites for IPsec.* Internet Engineering Task Force.

Hoffman, P. (2007). *RFC4894: Use of Hash Algorithms in Internet Key Exchange (IKE) and IPsec.* Internet Engineering Task Force.

Horman, N. (2004). *Understanding And Programming With Netlink Sockets.* Retrieved from `http://people.redhat.com/nhorman/papers/netlink.pdf` (retrieved at October 7, 2013)

Housley, R. (2005). *RFC4309: Using Advanced Encryption Standard (AES) CCM Mode with IPsec Encapsulating Security Payload (ESP).* Internet Engineering Task Force.

Housley, R. (2007). *RFC5084: Using AES-CCM and AES-GCM Authenticated Encryption in the Cryptographic Message Syntax (CMS).* Internet Engineering Task Force.

Hughes, R. (2004). Report of the Quantum Cryptography Technology Experts Panel. In T. Heinrichs (Ed.), *A Quantum Information Science and Technology Roadmap Part 2: Quantum Cryptography.* Advanced Research and Development Activity. Retrieved from `http://qist.lanl.gov/pdfs/whole_roadmap.pdf` (retrieved July 24, 2013)

Inamori, H. (2002). Security of practical time-reversed epr quantum key distribution. *Algorithmica*, *34*(4), 340-365.

Internet Assigned Numbers Authority. (2011). *Cryptographic Suites for IKEv1, IKEv2, and IPsec.* Retrieved from `http://www.iana.org/assignments/crypto-suites/crypto-suites.txt` (retrieved September 1, 2013)

Internet Assigned Numbers Authority. (2012a). *IPSEC AH Transform Identifiers.* Retrieved from `http://www.iana.org/assignments/isakmp-registry/isakmp-registry.xhtml#isakmp-registry-7` (retrieved at March 13, 2014)

Internet Assigned Numbers Authority. (2012b). *IPSEC ESP Transform Identifiers.* Retrieved from `http://www.iana.org/assignments/isakmp-registry/isakmp-registry.xhtml#isakmp-registry-9` (retrieved at March 13, 2014)

Kanda, M., Miyazawa, K., & Esaki, H. (2004). Usagi ipv6 ipsec development for linux. In *Applications and the internet workshops, 2004. saint 2004 workshops. 2004 international symposium on* (p. 159-163). Intitute of Electrical and Electronical Engineers.

Kang, J., Jeong, K., Sung, J., Hong, S., & Lee, K. (2013). Collision Attacks on AES-192/256, Crypton-192/256, mCrypton-96/128, and Anubis. *Journal of Applied Mathematics*, *2013*, 713673.

Kaufman, C. (2005). *RFC4306: The Internet Key Exchange (IKEv2).* Internet Engineering Task Force.

Kaufman, C., Hoffman, P., Nir, Y., & Eronen, P. (2010). *RFC5996: The Internet Key Exchange (IKEv2.* Internet Engineering Task Force.

Kent, S. (2005a). *RFC4302: IP Authentication Header.* Internet Engineering Task Force.

Kent, S. (2005b). *RFC4303: IP Encapsulating Security Payload.* Internet Engineering Task Force.

Kent, S., & Atkinson, R. (1998). *RFC2401: Security Architecture for the Internet Protocol.* Internet Engineering Task Force.

Kent, S., & Seo, K. (2005). *RFC4301: Security Architecture for the Internet Protocol.* Internet Engineering Task Force.

Keromytis, A., & Provos, N. (2000). *RFC2857: The Use of HMAC-RIPEMD-160-96 within ESP and AH.* Internet Engineering Task Force.

Kerriks, M. (Ed.). (2012a). *netlink (3) - Linux manual page.* (command *man 3 netlink*' on GNU/Linux; retrieved at October 30, 2013)

Kerriks, M. (Ed.). (2012b). *netlink (7) - Linux manual page.* (command *man 7 netlink* on GNU/Linux; retrieved at October 30, 2013)

Kerriks, M. (Ed.). (2012c). *rtnetlink (7) - Linux manual page.* (command 'man 7 rtnetlink' on GNU/Linux; retrieved at March 05, 2014)

Keyl, M. (2002, October). Fundamentals of quantum information theory. *Physics Reports*, *369*, 431-548.

Kollmitzer, C., & Pivk, M. (2010). *Applied Quantum Cryptography* (Vol. 797). Berlin, Heidelberg: Springer.

Krawczyk, H., Bellare, M., & Canetti, R. (1997). *RFC2104: HMAC: Keyed-Hashing for Message Authentication.* Internet Engineering Task Force.

Kwiat, P., & Rarity, J. (2004). Section 6.4: Entangled Photon Pairs. In T. Heinrichs (Ed.), *A Quantum Information Science and Technology Roadmap Part 2: Quantum Cryptography.* Advanced Research and Development Activity. Retrieved from `http://qist.lanl.gov/pdfs/whole_roadmap.pdf` (retrieved July 24, 2013)

Lai, X., & Massey, J. L. (1991). A proposal for a new block encryption standard. In I. B. Damgård (Ed.), *Advances in cryptology — eurocrypt '90* (Vol. 473, pp. 389–404). Berlin Heidelberg: Springer.

Lang, J.-P. (2006). *strongSwan UserDocumentation: strongSwan plugins.* Retrieved from `http://wiki.strongswan.org/projects/strongswan/wiki/PluginList` (retrieved at November 14, 2013)

Law, L., & Solinas, J. (2011). *RFC6379: Suite B Cryptographic Suites for IPsec.* Internet Engineering Task Force.

Lee, H., Yoon, J., Lee, S., & Lee, J. (2005). *RFC4196: The SEED Cipher Algorithm and Its Use with IPsec.* Internet Engineering Task Force.

Leibniz, G. W., Freiherr zu. (1847). *Monadologie* (R. Zimmermann, Ed.). Vienna: Braumueller und Seidel.

Litvak, M. (2011). *ip (8) - Linux manual page.* (command *man 8 ip* on GNU/Linux; retrieved at November 1, 2013)

Lo, H.-K., Chau, H., & Ardehali, M. (2005). Efficient Quantum Key Distribution Scheme and a Proof of Its Unconditional Security. *Journal of Cryptology*, *18*(2), 133-165.

Lodewyck, J., Bloch, M., García-Patrón, R., Fossier, S., Karpov, E., Diamanti, E., ... Grangier, P. (2007, October). Quantum key distribution over 25 km with an all-fiber continuous-variable system. *Physical Review A*, *76*, 042305. Retrieved from `http://arXiv.org/abs/0706.4255v2` (preprint retrieved at August 22, 2013)

Lütkenhaus, N. (1999, May). Estimates for practical quantum cryptography. *Phys. Rev. A*, *59*, 3301-3319. Retrieved from `http://arXiv.org/abs/quant-ph/9806008v2` (preprint retrieved at August 18, 2013)

MagiQ Technologies. (2007). *MAGIQ QPN 8505 Security Gateway.* Retrieved from `http://www.magiqtech.com/MagiQ/Products_files/8505_Data_Sheet.pdf` (retrieved at December 2, 2013)

Manral, V. (2007). *RFC4835: Cryptographic Algorithm Implementation Requirements for Encapsulating Security Payload (ESP) and Authentication Header (AH).* Internet Engineering Task Force.

Matsui, M., Nakajima, J., & Moriai, S. (2004). *RFC3713: A Description of the Camellia Encryption Algorithm.* Internet Engineering Task Force.

Maurhart, O. (2006). *Q3P - A Proposal* [SECOQC deliverable]. (Unpublished Technical Report)

Maurhart, O. (2010). QKD networks based on Q3P. In C. Kollmitzer & M. Pivk (Eds.), *Applied Quantum Cryptography* (Vol. 797, p. 151-171). Berlin, Heidelberg: Springer.

Maurhart, O. (2013, May 10). *The AIT QKD Software Handbook.*

McDonald, J., Metz, C., & Phan, B. (1998). *RFC2367: PF_KEY Key Management API, Version 2.* Internet Engineering Task Force.

McGrew, D. A. (2012). Impossible plaintext cryptanalysis and probable-plaintext collision attacks of 64-bit block cipher modes. *IACR Cryptology ePrint Archive*, *2012*, 623.

Menezes, A. J., van Oorschot, P. C., & Vanstone, S. A. (1996). *Handbook of Applied Cryptography*. Boca Raton: CRC Press.

Nagayama, S. (2010). *Modifying IKE for IPsec with Quantum Key Distribution.* Retrieved from `http://aqua.sfc.wide.ad.jp/publications/nagayama_b-thesis.pdf` (retrieved at March 12, 2014)

Nagayama, S., & Van Meter, R. (2009). *Internet-Draft: IKE for IPsec with QKD.* Internet Engineering Task Force. (draft-nagayama-ipsecme-ipsec-with-qkd-00, expired work)

Naik, D. S., Peterson, C. G., White, A. G., Berglund, A. J., & Kwiat, P. G. (2000, May). Entangled State Quantum Cryptography: Eavesdropping on the Ekert Protocol. *Physical Review Letters*, *84*, 4733-4736. Retrieved from `http://arXiv.org/abs/quant-ph/9912105v1` (preprint retrieved at August 18, 2013)

Nam, S. W. (2004). Section 6.3: Single-Photon Light Sources. In T. Heinrichs (Ed.), *A Quantum Information Science and Technology Roadmap Part 2: Quantum Cryptography.* Advanced Research and Development Activity. Retrieved from `http://qist.lanl.gov/pdfs/whole_roadmap.pdf` (retrieved July 24, 2013)

National Institute of Standards and Technology. (1999). *Data Encryption Standard (AES) - Federal Information Processing Standards Publication 46-3.* Author. Retrieved from `http://csrc.nist.gov/publications/fips/fips46-3/fips46-3.pdf` (retrieved at March 13, 2014)

National Institute of Standards and Technology. (2001). *Advanced Encryption Standard (AES) - Federal Information Processing Standards Publication 197.* Author. Retrieved from `http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf` (retrieved at March 13, 2014)

National Institute of Standards and Technology. (2002). *Secure Hash Standard (SHA) - Federal Information Processing Standards Publication 180-2.* Author. Retrieved from `http://csrc.nist.gov/publications/fips/fips180-2/fips180-2withchangenotice.pdf` (retrieved at March 15, 2014)

Neppach, A., Pfaffel-Janser, C., Wimberger, I., Lorünser, T., Meyenburg, M., Szekely, A., & Wolkerstorfer, J. (2008). Key management of quantum generated keys in ipsec. In *Proceedings of seccrypt 2008* (p. 177-183). INSTICC Press.

Nielsen, M. A., & Chuang, I. L. (2000). *Quantum Computation and Quantum Information*. Cambridge: Cambridge University Press.

Pieprzyk, J., Hardjono, T., & Seberry, J. (2003). *Fundamentals of computer security*. Berlin, Heidelberg: Springer.

Piper, D. (1998). *RFC2407: The Internet IP Security Domain of Interpretation for ISAKMP.* Internet Engineering Task Force.

Pivk, M. (2010a). Preliminaries. In C. Kollmitzer & M. Pivk (Eds.), *Applied Quantum Cryptography* (Vol. 797, p. 3-21). Berlin, Heidelberg: Springer.

Pivk, M. (2010b). Quantum Key Distribution. In C. Kollmitzer & M. Pivk (Eds.), *Applied Quantum Cryptography* (Vol. 797, p. 23-47). Berlin, Heidelberg: Springer.

Poppe, A., Fedrizzi, A., Ursin, R., Böhm, H., Lörunser, T., Maurhardt, O., ... Zeilinger, A. (2004). Practical quantum key distribution with polarization entangled photons. *Optics Express*, *12*(16), 3865–3871.

Postel, J. (1981). *RFC793: Transmission Control Protocol.* Information Sciences Institute University of Southern California.

Ralph, T. C. (1999, December). Continuous Variable Quantum Cryptography. *Physical Review A*, *61*, 010303. Retrieved from `http://arXiv.org/abs/quant-ph/9907073v1` (preprint retrieved at August 18, 2013)

Rescorla, E. (1999). *RFC2631: Diffie-Hellman Key Agreement Method.* Internet Engineering Task Force.

Rivest, R. (1992). *RFC1321: The MD5 Message-Digest Algorithm.* Internet Engineering Task Force.

Robertson, H. P. (1929). The uncertainty principle. *Physical Review*, *34*, 163-164.

Roy, V. (2004). *Benchmarks for Native IPsec in the 2.6 Kernel.* Retrieved from `http://www.linuxjournal.com/article/7840` (retrieved at September 1, 2013)

Salim, J., Khosravi, H., Kleen, A., & Kuznetsov, A. (2003). *RFC3549: Linux Netlink as an IP Services Protocol.* Internet Engineering Task Force.

Scarani, V., Acín, A., Ribordy, G., & Gisin, N. (2004, Feb). Quantum Cryptography Protocols Robust against Photon Number Splitting Attacks for Weak Laser Pulse Implementations. *Physical Review Letters*, *92*, 057901. Retrieved from `http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.48.1150&rep=rep1&type=pdf` (preprint retrieved at 08/16/2013)

Schartner, P., & Kollmitzer, C. (2010). Quantum-Cryptographic Networks from a Prototype to the Citizen. In C. Kollmitzer & M. Pivk (Eds.), *Applied Quantum Cryptography* (Vol. 797, p. 173-184). Berlin, Heidelberg: Springer.

Schneier, B. (1994). Description of a new variable-length key, 64-bit block cipher (blowfish). In R. Anderson (Ed.), *Fast software encryption* (Vol. 809, pp. 191–204). Springer Berlin Heidelberg. Retrieved from `http://dx.doi.org/10.1007/3-540-58108-1_24` doi: 10.1007/3-540-58108-1_24

Schrödinger, E. (1930). Zum Heisenbergschen Unschärfeprinzip. *Sitzungsberichte der Preussischen Akademie der Wissenschaften, physikalisch-mathematische Klasse*, 296–303.

Sfaxi, M., Ghernaouti-Hélie, S., Ribordy, G., & Gay, O. (2005). Using quantum key distribution within ipsec to secure man communications. In *Proceedings of metropolitan area networks (man2005).*

Shannon, C. E. (1948, July, October). A Mathematical Theory of Communication. *The Bell System Technical Journal*, *27*, 379-423, 623–656. (Reprint)

Shannon, C. E. (1949, October). Communication Theory of Secrecy Systems. *The Bell System Technical Journal*, *28*, 656–715.

Smart, N. (Ed.). (2012). *ECRYPT II Yearly Report on Algorithms and Keysizes (2011-2012).* European Network of Excellence in Cryptology II. Retrieved from `http://www.ecrypt.eu.org/documents/D.SPA.20.pdf` (retrieved at March 17, 2014)

Smart, N. P., Rijmen, V., Warinschi, B., & Watson, G. (2013). *Algorithms, Key Sizes and*

Parameters Report - 2013 recommendations. European Network and Information Security Agency. Retrieved from `http://aqua.sfc.wide.ad.jp/publications/nagayama_b-thesis.pdf` (retrieved at March 12, 2014)

Stallman, R., et al. (2013). *Using the GNU Compiler Collection - For gcc version 4.8.1.* GNU Press. Retrieved from `http://gcc.gnu.org/onlinedocs/gcc-4.8.1/gcc.pdf` (retrieved September 1, 2013)

Stallman, R. M., McGrath, R., & Smith, P. D. (2010). *GNU Make - A Program for Directing Recompilation - Version 3.82.* GNU Press. Retrieved from `http://www.gnu.org/software/make/manual/make.pdf` (retrieved September 1, 2013)

Stinson, D. (1992). Universal hashing and authentication codes. In J. Feigenbaum (Ed.), *Advances in cryptology — crypto '91* (Vol. 576, p. 74-85). Berlin, Heidelberg: Springer.

Stucki, D., Legré, M., Buntschu, F., Clausen, B., Felber, N., Gisin, N., ... others (2011). Long-term performance of the swissquantum quantum key distribution network in a field environment. *New Journal of Physics*, *13*(12), 123001.

Swiss Quantum. (2009). *QKD enhanced IPsec Encryptor.* Retrieved from `http://swissquantum.idquantique.com/?QKD-enhanced-IPsec-Encryptor` (retrieved at March 11, 2014)

Talijanac, M. (2012). *XFRM Programming.* Retrieved from `http://www.croz.net/eng/xfrm-programming/` (retrieved at October 20, 2013)

Tittel, W., Brendel, J., Zbinden, H., & Gisin, N. (2000, May). Quantum Cryptography Using Entangled Photons in Energy-Time Bell States. *Physical Review Letters*, *84*, 4737-4740.

Treiber, A., Poppe, A., Hentschel, M., Ferrini, D., Lorünser, T., Querasser, E., ... Zeilinger, A. (2009, April). A fully automated entaglement-based quantum cryptography system for telecom fiber networks. *New Journal of Physics*(11), 045013.

Ursin, R., Tiefenbacher, F., Schmitt-Manderbach, T., Weier, H., Scheidl, T., Lindenthal, M., ... Zeilinger, A. (2007). Entanglement-based quantum communication over 144 km. *Nature Physics*, *3*, 481-486.

Vernam, G. S. (1919). *Secret Signaling System* (Patent No. US 1310719).

Vernam, G. S. (1926). Cipher Printing Telegraph Systems For Secret Wire and Radio Telegraphic Communications. *Transactions of the American Institute of Electrical Engineers*, *XLV*, 295-301. (Reprint B-198)

Viega, J., & McGrew, D. (2005). *RFC4106: The Use of Galois/Counter Mode (GCM) in IPsec Encapsulating Security Payload (ESP).* Internet Engineering Task Force.

von der Linden, P. (1994). *Expert C Programming: Deep C Secrets*. Upper Saddle River: Prentice Hall.

Ward, D. (2011). *ip xfrm (8) - Linux manual page.* (command *man 8 ip xfrm* on GNU/Linux; retrieved at October 30, 2013)

Wegman, M. N., & Carter, J. L. (1981). New Hash Functions and Their Use in Authentication and Set Equality. *Journal of Computer and System Sciences*, *22*(3), 265-279.

Wolf, S. (1999). Unconditional Security in Cryptography. In I. Damgård (Ed.), *Lectures on Data Security* (Vol. 1561, p. 217-250). Berlin, Heidelberg: Springer.

Wootters, W. K. (1998, August 15). Quantum entanglement as a quantifiable resource. *Philosophical Transcriptions of the Royal Society A*, *356*, 1717-1731.

Wootters, W. K., & Zurek, W. H. (1982, October 28). A single quantum cannot be cloned. *Nature*, *299*(5886), 802-803.

Wright, G. R., & Stevens, W. R. (1995). *TCP/IP Illustrated, Volume 2 - The Implementation*. Indianapolis: Addison-Wesley.

Zbinden, H., Bechmann-Pasquinucci, H., Gisin, N., & Ribordy, G. (1998). Quantum cryptography. *Applied Physics B*, *67*(6), 743-748.

Listings