Politecnico di Milano
Scuola di Ingegneria Industriale e dell'Informazione
Corso di Laurea Magistrale in Ingegneria Informatica
Dipartimento di Elettronica, Informazione e Bioingegneria

Peaks: Adding Proactive Security to OpenPGP keyservers

Relatore: Prof. Alessandro BARENGHI

Tesi di Laurea di:
Mattia PINI Matr. 834738

Anno Accademico 2017–2018

Il più grande spreco nel mondo è la differenza tra ciò che siamo e ciò che potremmo diventare. (Ben Herbster)

Ringraziamenti

Desidero ringraziare tutti coloro che mi hanno aiutato nella realizzazione della mia Tesi.
Ringrazio prima di tutto il professor Alessandro Barenghi, Relatore, e il professor Gerardo Pelosi, dai quali è partita l'idea di questo lavoro.

Proseguo con *Google* e *Stack Overflow*, i "santi" di ogni Ingegnere Informatico che si rispetti; ringrazio anche tutto il Politecnico di Milano con i suoi Professori, che mi hanno dato la capacità e le competenze per risolvere ogni tipo di problema presentatosi.

Un ringraziamento particolare va a Fabiana per il suo sostegno durante tutto il mio percorso universitario, che, senza di lei, non sarebbe neanche mai partito. Grazie per gli aggiustamenti nella Tesi, per le *s* mancate, e per quelle di troppo. Grazie per le incazzature e per le non incazzature, per i suggerimenti, e le discussioni che ci sono state.

Vorrei inoltre rigraziare amici e colleghi che mi hanno supportato (e sopportato) in questi anni. Grazie anche alla mia famiglia che ha sostenuto le mie scelte, seppur (a volte) senza condividerle.

Ringrazio tutti per l'aiuto e la comprensione che avete avuto nei miei confronti durante questi anni universitari.
Sperando di non aver dimenticato nessuno (cosa ovviamente impossibile), rinnovo i miei ringraziamenti a tutti.

Grazie Grazie Grazie.

Abstract

An OpenPGP protocol is a standard used to obtain both confidentiality and authenticity in the asynchronous message exchange, especially with emails.
This protocol, now utilized in every part of the world, is born almost 25 years ago, thanks to Phil Zimmermann.

During the years, many problems have been accumulated over the OpenPGP world. The number of the keys has boomed, reaching the value of almost 5 million keys. This enormous number is a difficulty for a server based on a Key-Value storage such as the SKS Keyserver.

I decided to develop this project, creating a new keyserver, to allow more people to manage it, reaching a complete and precise update.

Furthermore, I have to underline the results came to light from the paper *Challenging the Trustworthiness of PGP: Is the Web-of-Trust Tear-Proof?* [2]: in fact, it has been fully demonstrated that a lot of keys are almost obsolete, and many others are still vulnerable.
From this consideration arose the idea to project a keyserver that, not only would handle all the user's requests, but would also analyze every single packet that has intended to form the key.

Overall, this new keyserver should prove that all the keys used in OpenPGP respect specific security data.

The structure of this paper starts with a brief introduction to the cryptography: its history, development and properties. Then, with an excursus, it is described what is OpenPGP, how it is structured, which are the algorithms analyzed and how the SKS keyserver works. In the following chapter, it is detailed the structure of the new keyserver created, while in the final one can be found the graphs representing the results achieved.

Contents

List of Figures

List of Tables

List of Algorithms

Introduction

For thousands of years cryptography has been used to provide a secure way to communicate between parties. The original idea was to share the encrypting and decrypting technique only between the communicating parties to avoid the possible eavesdropping from an unwanted person. This, bind to the absence of strong cryptanalysis method and a real definition of when a cipher is cryptanalytical resistant, was considered a weakness.

With the advent of the modern cryptology, the idea behind the security of the system changed radically. Everyone should have been able to observe and analyze the cryptosystem, because everything about it should has been release to the public, except for the key. This is known as the Kerckhoff's Principle. In the modern cryptology a cryptoscheme is considered secure if the computational complexity to break it is at least equal to the one that we need to solve a computationally hard problem.

The security properties that has to be ensured in the modern cryptology are:

Confidentiality: Confidentiality means that an information is not available to unauthorized user. A user can be any entity that can operate in the system. e.g. a person or a company.

Authenticity: Authenticity is divided in two part: Authentication and Authorization. The authentication property is needed when we want to know who is the counterpart. The authorization instead, is needed to allow or deny specific action to the authenticated user.

Data Integrity: Data integrity refers to the fact that the data can be tamper with. If a cryptosystem can ensure data integrity, the counterpart can verify if the data has been manipulated (for example if part of the

data is missing or if it has been substituted with another content).

Data Authentication: Authentication and data integrity together can provide Data Authentication. In this way we can prevent data forgery (i.e. create false data, conferring its creation to another user).

Non–Repudiation: The non–repudiation property ensures that a statement cannot be retracted.

Symmetric cryptography

In the symmetric cryptography the algorithms use the same key for the encryption of the plaintext (i.e. non-encrypted data) and the decryption of the ciphertext(i.e. encrypted data).

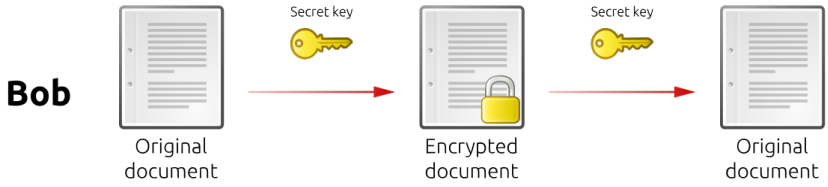


Figure 1: Sketch about how the symmetric cryptography works

When two users (e.g. Alice and Bob) want to exchange message through a symmetric cryptosystem, they have to share a secret key over a secure channel; once done they can encrypt and decrypt the messages with the shared key and send them to each other through an unsecured channel. In this way their messages are protected when travelling over the unsecure channel, because they are encrypted, and any other user cannot read them.

The symmetric cryptography is very computationally efficient and it provides confidentiality and data authentication (Alice is sure that only Bob can read her messages and that the received encrypted message are written only by Bob and vice-versa), but without the non repudiation. On the contrary, it is very difficult to exchange the shared secret over a secure channel because we will need it for every new communication that we want to do. Furthermore a group communication requires that every possible user pair in the group shares its secret over a distinct secure channel, thus also the key management can be a problem; besides, it cannot provide non repudiation.

Asymmetric cryptography

In asymmetric cryptography, algorithms use two keys: a public one, that is shared among all the users, and a private one, that is known only to its owner. When Bob wants to send a secret message to Alice, he can encrypt the message with the public key of Alice and send it to her. Alice can decrypt the message using her private key; in this way every user that can intercept the message is not able to decrypt it because only Bob knows his private key, and the confidentiality is ensured.

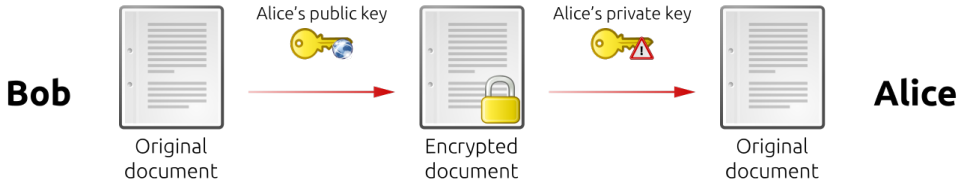


Figure 2: Sketch about how the asymmetric cryptography works

The asymmetric cryptography can also be used to provide authentication and non repudiation. Alice can generate a signature over a message with her private key, and every other user can verify it using Alice's public key. Obviously once the signature is created, the signer can encrypt all the data with the public key of another user to ensure confidentiality, authentication and eventually data-integrity all together.

Usually the signature is not done on a message but on the output of an hash function that has the message as input. An hash function, given an arbitrary length string in input, generates a fixed length output (called digest or hash). This function has to ensure three properties:

First preimage problem: given a digest the recovering of the input message has to be computationally impossible.

Second preimage problem: given a message and its digest, it must be computationally impossible found another message with the same digest.

Collision resistance: it is computationally impossible to found two messages with the same digest.

The asymmetric cryptography is though very slow with respect to the

symmetric one and it need a very long key, to reach the same security margin of the second one.

Hybrid Cryptoschemes

The two different cryptographic primitives are often used together.
For example when Alice wants to start a communication with Bob, she can use a symmetric algorithm to encrypt a message and an asymmetric algorithm to share the symmetric secret with Bob.

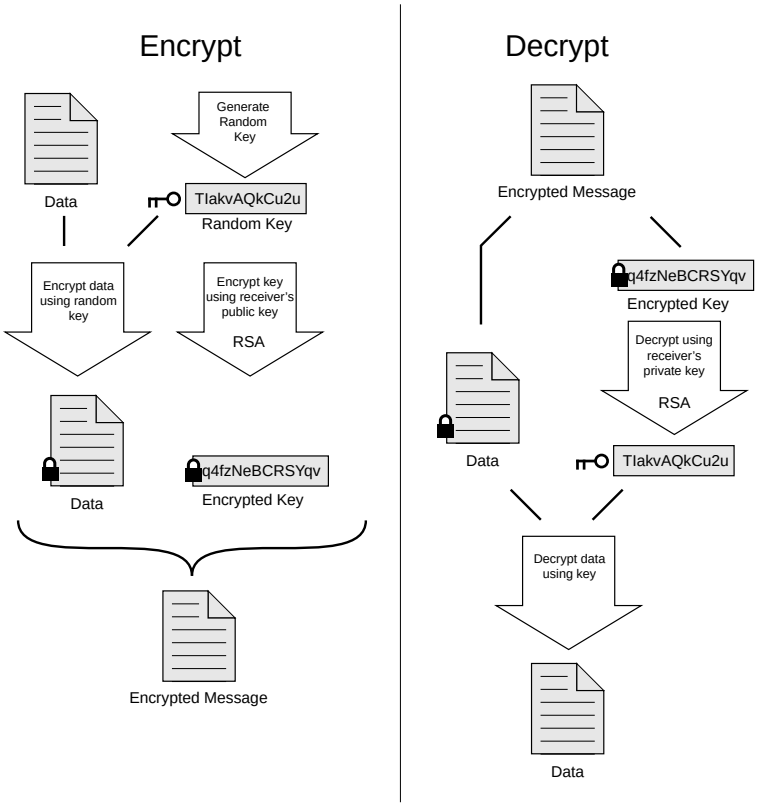


Figure 3: Sketch of how the idea behind an hybrid cryptoscheme

In this way we can share the symmetric secret key over a secure channel and then use it to encrypt and decrypt the messages.

Another problem with the asymmetric cryptography is the truth about the binding between the public key and the user who proclaims to own it. To verify this we can use a *digital certificate*.

Digital Certificate

The digital certificate is a document that certified the binding between a public key and a certain user. It contains a signature done with an asymmetric cryptosystem.

A digital certificate contains information about how the signature has been generated and who is the user, owner of the signed key, plus some other data. It includes the hash algorithm used, the subject name (e.g. can be a person, a company, an organization), the user that has performed the signature (can be an entity called certification authority of another user on the system), eventually it could contain the creation date and the validity time or the expiration date. Obviously it also contain the signature. It is

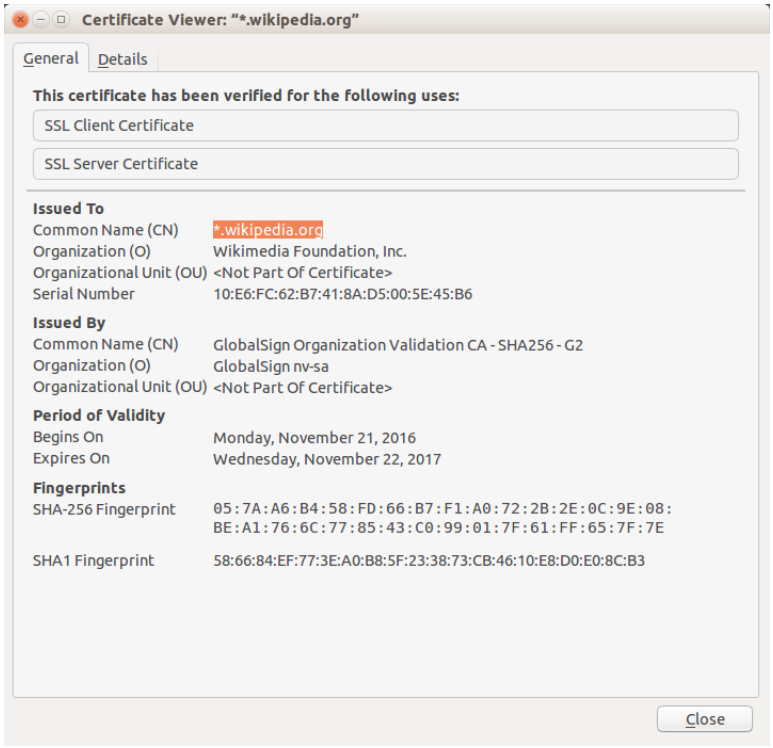


Figure 4: An example of the Wikipedia's digital certificate

distributed as a file with all the information inside.

There are two main infrastructures where the digital certificates are used. The Public Key Infrastructure (PKI) and the Web Of Trust (WoT).

Public Key Infrastructure

In a public key infrastructure the certificates are created by an entity called Certification Authority (CA).

A job for a certification authority is to check (as a matter of fact not always the check is done by it, but it is usually like that) and certificate that a public key belongs to a specific user. The checking can be done meeting the user in person and/or checking its documents.

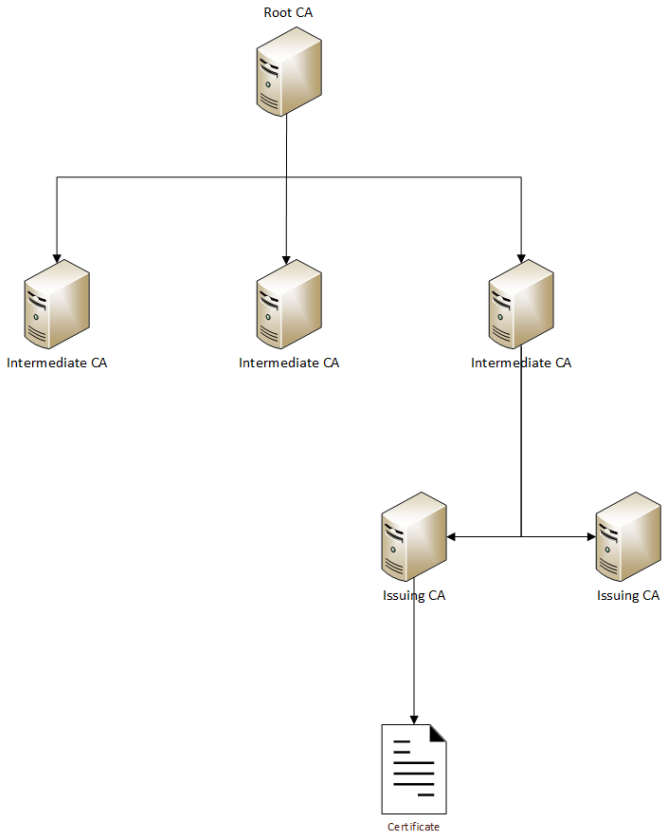


Figure 5: Sketch of the CA hierarchy [9]

The CAs are organized in a tree structure. Every CA can sign (besides the users) the certificate of another CA and its one is signed by a distinct CA. The CA that is not signed by any other CA is called root CA. A root CA certificate is generated by the CA itself. In Figure 5 there is an example of the CA tree structure.

When a user wants to check if a public certificate has to be trusted, he

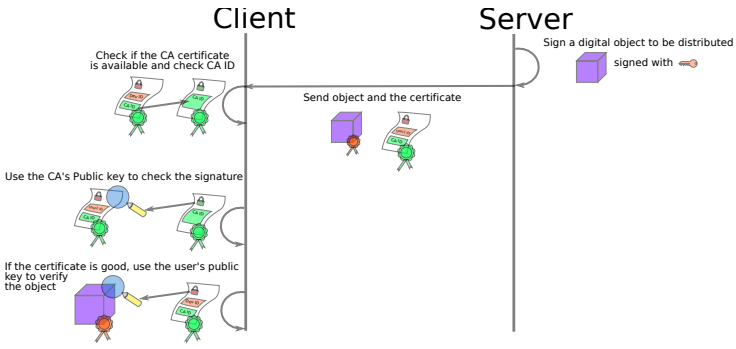


Figure 6: Sketch about the verification of a certificate in a public key infrastructure

checks behind his trusted CA, if there is the one that has performed the signature on the certificate of the interested host. If he finds the CA he can verify the signature, using the CA's public key. If the verification fails the certificate is not considered authentic. This happen also if the CA that has performed the signature, is not among those he trusts.

If a certificate is compromised can be revoked. A list of revoked certificates can be found inside a Certificate Revocation List, an offline list with all the certificates revoked by a specific CA, or in a Online Certificate Status Protocol, an online service that let you know if a certificate has been revoked.

The format of the digital certificates in a public key infrastructure is the X.509, used in many internet protocol (for example in the HTTPS)

Web Of Trust

In the web of trust the role of a CA relies on every user in the system. Each user can sign the key of other known users, confirming their identity. A user can have more than one signature per certificate, in order to increase the trust of the other user on it. Thus if we have to prove the authenticity of a user A we need to verify at least one signature on his certificate: if one of them come from a trusted user (and of course if it is valid), the authenticity is verified, otherwise we have to prove first (in the same way) the authenticity of a user B, who made the signature that we are checking; once proved, B has become a trusted user and we can prove the authenticity of the certificate of A.

The certificates are stored in keyservers so that everyone can have access to them. There is a network of keyservers, all synchronized together.

If a certificate has been compromised, a user can revoke his signature, reducing the level of trust of the other user in that certificate.

A web of trust standard is the OpenPGP, used, for example, in encrypted mail exchange.

Chapter 1

Background and State of the Art

This chapter shows how the OpenPGP certificates are composed and synchronized between the keyservers. It also debate about the asymmetric algorithm used in OpenPGP.

The chapter contains:

- An high level explanation of how OpenPGP is structured, Section 1.1
- A brief history of OpenPGP, Section 1.1.1
- A particular focus about the most important thing, Section 1.1.2
- The asymmetric algorithm used in OpenPGP, Section 1.2
- How the synchronization between the keyservers works, Section 1.3

1.1 OpenPGP

OpenPGP is a security standard that provides confidentiality and authenticity in message and data file exchange. It is used mainly in email exchange (confidentiality and data origin) and linux package manager (data origin only). It is used used to handle asymmetric key, exchanging messages and performing data authentication. An OpenPGP object is represented as a list of packets, each one containing different information.

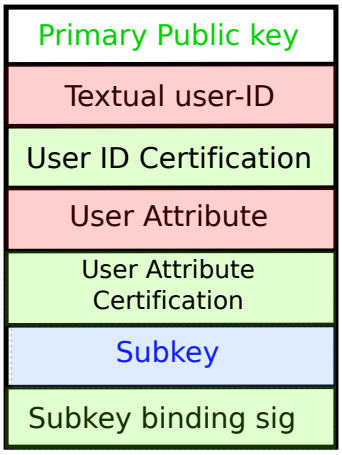


Figure 1.1: Sketch of a list of packets of an OpenPGP version 4 certificate. Those with the green background, are the signatures, the blu one is the subkey; the packets with the red background, finally are the user ID.

In particular an OpenPGP object representing a public key, is called certificate. In Figure1.1 is shown an example of a sequence of packets, representing a certificate in the OpenPGP standard.

The certificates are store in online keyserver, all synchronized together. There is also some offline software that follows this standard (e.g. GNU Privacy Guard[1]) and can be used to create and handle the certificates and to perform encrypting, descrypting and signing.

1.1.1 Brief history

PGP (namely Pretty Good Privacy) was born in 1991, created by Philip Zimmermann. After some years of development and trouble (Zimmermann was investigated by the US government for "exporting ammunition without a license" because, at that time, cryptosystem with keys larger then 40 bits was considered ammunition, and PGP was used also outside the USA), in 1996 he published the RFC 1991 [1] and founded PGP Inc. In 1997 PGP Inc proposed to create an open standard, and the year later another paper was published, the first one dedicated to the new OpenPGP format: RFC 2440 [7], followed, ten years later, by the RFC 4880 [6] that defines the actual standard, now composed by RSA, Elgamal and DSA, as asymmetric algorithm, and by the International Data Encryption Algorithm (IDEA),

[1] `https://www.gnupg.org/`

TripleDES, CAST5, Blowfish, AES-128, AES-192, AES-256 and Twofish, as symmetric algorithm. During the following years the standard was improved introducing the Camelia Cipher (with the RFC 5581 [24]) and the Elliptic Curve Cryptography (with the RFC 6637 [11]), arriving at the presentation of a draft to update the RFC 4880 [17] by Werner Koch, bringing a new version of OpenPGP (only an idea, not yet implemented by any software) and the Edward Curve Cryptography.

1.1.2 RFC 4880

A full description of the OpenPGP message format can be found in the RFC 4880 [6]. Actually there are two version of the OpenPGP format, version 3 that is deprecated, and version 4 (there is also version 2 but it is equal to version 3, except for the fact that in version 2 the certificate are generated by PGP 2.5 or less, so we will treat version 2 and 3 as they are equals).

A version 3 OpenPGP Public Key is composed by a *public key* packet, an optional *revocation self signature* packet (if the public key is revoked), and a list of *user ID* packets, each one followed by the list of the related *signatures*.

A version 4 instead has a *public key* packet (called *primary key*), followed by two optional Signatures: a *revocation* or a *direct key signature*. Then there is the list of *user IDs* and *signatures*, as version 3, but with an extension: the list can also contains some *user attribute* packets, each one followed by the list of the related signatures. To close the Key there could be a list of *subkeys* and, optionally for each one, a list of related *signatures*.
In version 4 by convention, the primary key is used for the signature while the subkeys are used for the encryption.

An OpenPGP object (e.g. a public key, a secret key, a message) is a block composed by several packets. Each packet is formed by an header, that contains the tag to identify the packet, the length of the packet, and then the real content; the length can be represent in different ways, based on the dimension of the content.

The packets can be divided in three macro categories: *key packets*, *signature packet*, *data packet*. The packets divided by categories are shown in Table 1.1.

Table 1.1: List of OpenPGP packets divided in category

key packets category	signature packets category	data packets category
Public-Key Encrypted Session Key Packet	Signature Packet	Compressed Data Packet
Symmetric-Key Encrypted Session Key Packet	One-Pass Signature Packet	Symmetrically Encrypted Data Packet
Key Material Packet	Key Material Packet	Marker Packet
		Literal Data Packet
		Trust Packet
		User ID Packet
		User Attribute Packet
		Symmetrically Encrypted and Integrity Protected Data Packet
		Modification Detection Code Packet
		Private/Reserved Packet

1.1.3 Key Packets Category

Public-Key Encrypted Session Key Packet A Public-Key Encrypted Session Key Packet containing the symmetric key used to encrypt a message.

Symmetric-Key Encrypted Session Key Packet Same as Public-Key Encrypted Session Key Packet.

Key Material Packet A Key Material packet contains information about the key; there are four types of Key Material Packet:

- Secret Key
- Secret Subkey

- Public Key
- Public Subkey

The Subkey packet is defined only in version 4. In this version the Public-Key is called Primary-Key.
A key is recognized by a fingerprint, a key ID and a version. For a version 3 key, the key ID is the low 64 bits of the public modulus n (in version 3 only the RSA algorithm is defined). On the other hand, the fingerprint is the MD5 hash of the concatenation between the public modulus n and the public exponent e. A fingerprint for a version 4 key, is the octet 0x99, followed by two octet representing the packet length, and the field representing the key, all hashed with the SHA-1 algorithm. The Key ID is the low 64 bits of the fingerprint.

1.1.4 Signature Packets Category

Signature Packet A Signature packet represents a binding between a key packet and a message, a file, or other packet; for example, a user can sign a message to prove that he is the author, or he can sign a User ID packet to confirm that it belongs to a specific person. A signature Packet is composed by various specific subpackets that contains the information about the signature. Based on the employment, a signature has different types. In an OpenPGP key these types are:

***Certification of a User ID and Public-Key packet*:** Three types of signature (*generic*, *persona*, *casual*, *positive*) that certificate that a user ID or a user attribute belongs to a public key. The signer chooses a specific type for the signature, based on the confidence that he has about the binding between the key and the user ID. The *generic* certification indicates the lowest confidence, while the *positive* one, the highest.

***Subkey and Primary-Key Binding Signature*:** Two types of signatures (*Subkey Binding* and *Primary-Key Binding*) that bind the subkey to the primary key and vice-versa. The first one is performed by the primary key, and states that it owns the subkey, and the second one is

a statement done by the subkey, indicating that it is owned by the primary key

***Signature directly on a key*:** A signature computed directly on a key, that binds the inside information to the signed key.

***Key revocation signature*:** A signature that revokes directly a key.

***Subkey revocation signature*:** A signature that revokes a Subkey or Primary-Key Binding Signature.

***Certification revocation signature*:** A signature that revokes a certification over a user ID or a user attribute.

A signature cannot be physically deleted, to make it meaningless a user has to create a new signature that revokes the first one.
When a signature is done by and over the same key is called *self signature*. A self signature is performed by the public key to certificate that a certain user ID belongs to it. Is also used to know the primary (i.e. the most important) User ID packet for an OpenPGP object.
A signature is computed over the hash of the content of the signed packet plus some other information; the signed hash contains also some info situated in the same signature packet. For a version 3 signature, the signed hash is computed over the five octets of the packet body, starting from the signature type.
For a version 4 instead, the fields hashed are the signature version, signature type, public-key algorithm, hash algorithm, hashed subpacket length, and all the hashed subpackets body.
The hashed fields all together form the hashed material.

One-Pass Signature Packet A One-Pass Signature Packet precedes the signed data and allow to verify the signature on the data during their first read.

1.1.5 Data Packets Category

Compressed Data Packet A Compressed Data Packet contains obviously compressed data. It can be found inside an encrypted packet or after a Signature

Symmetrically Encrypted Data Packet A Symmetrically Encrypted Data Packet contains data encrypted with a symmetric key algorithm. Once decrypted it contains a list of other OpenPGP packets.

Marker Packet Dummy Packet. Must be ignored when received. It contains "PGP" raw string in UTF-8.

Literal Data Packet A Literal Data Packet contains data and a field explaining how data is formatted.

Trust Packet A Trust Packet is used inside the local keyrings and should not be exported. It contains information about which key are trustworthy for the user.

User ID Packet A User ID packet contains the name and (optionally) the email address of the owner of the key.

User Attribute Packet A User Attribute packet is similar to the User ID packet, except for the possibility to store more data types other than only text. It is composed by subpackets (such as the signature packet), but until now exists the image subpacket only.

Symmetrically Encrypted and Integrity Protected Data Packet A Symmetrically Encrypted and Integrity Protected Data Packet, is a variant of the Symmetrically Encrypted Data Packet. This, used together with a Modification Detection Code Packet, improve the possibility to detect modification on encrypted data.

Modification Detection Code Packet A Modification Detection Code Packet contains a checksum of the plaintext data. It is used, as specified above, with a Symmetrically Encrypted and Integrity Protected Data Packet to detect possible modification on data.

Private/Reserved Packet A private or reserved packet has no specification in RFC 4880. It should be used only in private environment.

In order to avoid unprintable characters, the entire binary stream is encoded in base64. It is also followed by the 24-bit Cyclic Redundancy Check (a method to compute a checksum for a binary string. A C implementation can be found in [6, Section 6.1, p. 63]) of the stream, encoded in base64 as well. This encoding method is called Radix-64.

```
-----BEGIN PGP PUBLIC KEY BLOCK-----

mQENBFqvsEwBCACRGpvZl6MysyA1W4D0VAq6flJOfFcRly01t5mBqpryPUwBBfDo
Key9nj/FVxQxqwBvGL/oUZwyUSpfHw/961rk4bo6uxC7yQmjS2v5NlwjTIHn97XN
AieZMFbDta0Rs5KwzbJ1V9v7dzp6eOnSNWoUS3JqRi1Cr7hollkNZ4mP75Ij92aG
+fuAgtL1K2GUDvTBJWl2Sc0fEzWDExsI8wcdc1J2n4iDNUh72A6k8qQedBY0mknc
DV/9sw93p9nc4J1rL5fRBv12uqiKikNn+ccPviITTTUYrwSrZ2oKzRR8LoHxG+I8
9nRmklNB86TsQYyBhYzQrIFQc857FxExsrP/ABEBAAG0D3Rlc2lAZmlndXJlLmRv
Y4kBHAQQAQIABgUCWq+wTAAKCRB0Jy1cSDbtVammB/0ciOhzwKnqWqZ04KkwkPWn
6zFKP6pjHHkn/OHpQCjAFy6qrS9F65wo0PNz7E9I3Jb0gwXmAdlNwuzZcqd7UKrn
YxDE0mon1O6ssRtPecvg+6rOSqNUDrm/T7L4bf8V1vH9Ob30G9oUjU9km94Usryo
Pi4NHifPm4CeyQm84tDfAgYTdEod6BfECX2prw5+c0KdruOrKn3mkqzEBANEXUnP
DDkDs+0TePb68To26PkKxhBRuOn1HD1LXFwOlbrM61fwf5b1KozaLdWJEuQ4dQ5G
GUWJl0lAXtoVpkE8wVOtP9lLC0n6AXxIDaPpDAySa8vYPNazOQUa1d0FJbV2lC1d
=g3GG
-----END PGP PUBLIC KEY BLOCK-----
```

Figure 1.2: Example of an ASCII Armored OpenPGP Public key

An OpenPGP object is composed by:

- A line that identifies the object; for a public key object the line is
  - - - - -BEGIN PGP PUBLIC KEY BLOCK- - - - -
- Some optional line that could contain any type of info, e.g. info about the hostname, the version of the server
- a blank line
- The actual Radix-64 data, included the CRC.
- A line similar to the first one that close the object; for a public key object the line is
  - - - - -END PGP PUBLIC KEY BLOCK- - - - -

This representation is called ASCII Armored object.

1.2 Asymmetric algorithms in the OpenPGP standard

1.2.1 RSA Cryptosystem

An RSA public key is composed by a public modulus n, product of two secret prime factors p and q, and a public exponent e. An RSA cryptosystem works with modular arithmetic, and relies on the computational hardness of factoring large integers.

The secret key is composed of two prime numbers, p and q, that are the factors of n ($n = p \cdot q$), the value of the Euler's Totient Function $\varphi(n) = (p-1)(q-1)$ and a secret exponent $d = e^{-1} \bmod \varphi(n)$. The values e and d are thus bound together.

Given an RSA public key $k_{pub} = \langle n, e \rangle$ and the corresponding private $k_{priv} = \langle p, q, \varphi(n), d \rangle$, a message m can be encrypted using the public exponent:

$$c = Enc_{k_{pub}}(m) = m^{e \bmod \varphi(n)} \bmod n$$

and decrypted using the secret exponent:

$$m = Dec_{k_{priv}}(c) = c^{d \bmod \varphi(n)} \bmod n$$

In this way the confidentiality of the message m is ensured because only who has the secret key can decrypt the message.

To prove authenticity instead, a user can sign a message with his secret exponent:

$$s = Sign_{k_{priv}}(m) = m^{d \bmod \varphi(n)} \bmod n$$

and send the sign s together with the message m. Any other user can check the signature, through the public key of the signer:

$$CheckSign_{k_{pub}}(m, s) = m = s^{e \bmod \varphi(n)} \bmod n$$

This equality is true if and only if s has been generated by the user.

In OpenPGP before signing with an RSA key, a message has to be encoded with `EMSA-PKCS1-v1_5` [21, section 9.2]; to perform this encoding we have to:

1. Compute the ASN.1 DER value of the used hash function (the possible values can be found in [21, Appendix A.2.4, p. 62]), concatenated with the hash of the message: we call this result `t`
2. Generate an octet string $S = mLen - tLen - 0\text{x}FF - 0\text{x}FF - 0\text{x}FF$, with $mLen$ desired length of the output string (needs to be at least $tLen + 11$) and $tLen$ the octet length of `t`.

The output string (means the string that will be signed) will be $\texttt{0x00}\|\texttt{0x01}\|S\|\texttt{0x00}\|\texttt{t}$, where $\|$ is the concatenation operator.

1.2.2 Elgamal

Elgamal works in a cyclic subgroup of $\mathbb{Z}_p$, $(\mathbf{G}, \cdot)$ of order q with generator $g \in \mathbf{G}$, where the discrete logarithm problem (i.e. given $a \in \langle g \rangle$, find the smallest positive integer $x \in \mathbb{Z}_q$ such that $g^x = a$) is computationally impossible. This is ensured setting q as a large prime factor of $p - 1$, otherwise, if $p - 1$ has only small prime factors, computing the discrete logarithms is trivial, using the Pohlig–Hellman algorithm [16].

In OpenPGP an Elgamal public key is $k_{pub} \leftarrow \langle p, g, g^x \rangle$ where x, chosen randomly from $\{1, \ldots, q-1\}$, is the secret key.

To encrypt a message m (with $0 \leq m \leq p - 1$) you have to:

1. generate a random number $k \in \{1, \ldots, q-1\}$ and calculate $c_1 \equiv_p g^k$
2. calculate $s \equiv_p (g^x)^k$
3. calculate $c_2 \equiv_p m \cdot s$

The encrypted message is:

$$c = Enc_{k_{pub}}(m) = \langle c_1, c_2 \rangle$$

Given $c = \langle c_1, c_2 \rangle$ the decryption is:

$$m = Dec_{k_{priv}}(c) = c_2 \cdot (c_1^x)^{-1}$$

where $(c_1^x)^{-1}$ is the inverse of c_1^x in the group $\mathbf{G}$. Note also that $c_1^x = s$.

With an Elgamal cryptosystem we can also sign messages. Given m (such that $0 \leq m \leq p-1$) the message to be sign, H the used hash function, and an Elgamal public key, the procedure is the following:

1. generate a random number k such that $1 < k < p-1$ and gcd (k , $p-1$) $= 1$
2. compute $r = g^k \bmod p$
3. compute $s = (H(m) - x \cdot r) \cdot k^{-1} \bmod p-1$ and check $s \neq 0$. If false restart the procedure.

The signature is:

$$S = Sign_{k_{priv}}(m) = \langle r, s \rangle$$

To verify a signature check that $0 < r < p$ and $0 < s < p-1$, and then:

$$CheckSign_{k_{pub}}(m, S) \rightarrow g^{H(m)} = (g^s)^r \cdot r^s$$

The signature is accepted if and only if the equivalence is true.

The Elgamal signature is no more used in OpenPGP due to a vulnerability found in a library used by *GNU Privacy Guard*, that generates the random value k without checking the "coprimality" with p-1.

1.2.3 DSA

The DSA cryptosystem is a US standard for digital signatures [14]. The key idea is similar to Elgamal, and the public key is composed with:

- a prime q of bit length N.
- a prime p such that $q|p-1$ of bit length L.
  The possible values for (L, N) are: $(1024, 160)$, $(2048, 224)$, $(2048, 256)$, and $(3072, 256)$
- a number g such that q is the smallest positive integer that verifies $g^q = 1 \bmod p$.
- $y = g^s$ where $s \in \{1, \ldots, q-1\}$ is the private key.

The signature with DSA algorithm is done by:

1. generating a random number (must be different for any messages we want to sign) $k \in \{2, \ldots, q-1\}$.
2. calculating $r = (g^k \bmod p) \bmod q$ and check $r \neq 0$. If it is equal, reject and restart.
3. calculating $s = k^{-1}(H(m) + x \cdot r) \bmod q$ and check $s \neq 0$. If it is equal, reject and restart.

The signature is (r, s) and can be verified first checking that $0 < r, s < q$ and then computing:

1. $w = s^{-1} \bmod q$
2. $u_1 = H(m) \cdot w \bmod q$
3. $u_2 = r \bmod w \bmod q$
4. $v = (g^{u_1} y^{u_2} \bmod p) \bmod q$

the signature is verified if and only if $v = r$.

1.2.4 ECDSA

The ECDSA cryptosystem reinterprets the DSA cryptosystem employing the additive group of points of an elliptic curve, instead of a multiplicative group in Z_p. A curve is characterized by an equation in a specific field, a curve base point **G** (generator of the curve), its order (an integer n), and a point $Q_a = [d_a]G$, with d_a an integer number that represent the secret key, as specified in the ECDSA standard [14].

In OpenPGP there is no need to generate a curve because pre–generated ones are used. An OpenPGP Public-Key Packet contains the OID (i.e. an object identifier, that allows to recognize an object unequivocally through a string. The list of the OIDs used in OpenPGP for the curves can be found in [17, Section 9.2, p. 72]) of the curve and the point Q_a. The signing procedure is similar to the DSA one, given $e = H(m)$

1. generate a random number $k \in \{1 \ldots, n-1\}$ with $\gcd(k, n) = 1$
2. compute $[k]G = (x_1, x_2)$

3. compute $r = x_1 \bmod n$ and check that is different from 0. If it is equal, reject and restart.
4. compute z that is the L_n leftmost bits of e with $L_n = NumBits(n)$
5. compute $s = k^{-1}(z + r \cdot d_a) \bmod n$ and check that is different from 0. If it is equal, reject and restart.
6. The signature is (r, s)

Having the public key with the used curve and Q_a a valid point on it, the signature can be checked:

1. verifying that $r, s \in [1, n-1]$
2. computing $e = H(m)$ and extract z (as in the signing part)
3. computing $w = s^{-1} \bmod n$
4. computing $u_1 = z \cdot w \bmod n$
5. computing $u_2 = r \cdot w \bmod n$
6. computing $(x, y) = [u_1]G + [u_2]Q_a$, if $(x, y) = \mathbf{O}$ (with $\mathbf{O}$ the identity element) the signature is invalid

The signature is considered valid if and only if

$$r = x_1 \bmod n$$

1.2.5 EdDSA

The EdDSA cryptosystem works in the edward curve theory. In EdDSA we have a curve characterized by an equation, its generator $\mathbf{G}$, a public point A, and 32 bytes of cryptographically secure random data (that should remain secret), from which we will generate the secret key s. EdDSA has been added to OpenPGP with a draft proposal [17] by Werner Koch, the developer of *GNU Privacy Guard*, not yep accepted.

In EdDSA the method to generate the public key changes based on which curve we are using. Thus we will see only the methods regarding the curve *Ed25519*, because is the only one used in OpenPGP.

In order to generate the public key A we have to:

1. compute the SHA-512 of the 32 bytes of random data, and store the lower 32 bytes
2. clear the lower three bits of the first octet and the highest bit of the last octet. Set also the second highest bit of the last octet.
3. interpret the 32 bytes pruned string as a little endian integer, the secret key, s
4. The public key A is the encoding [13, Section 5.1.2, p. 10] of the scalar multiplication $[s]\mathbf{G}$

Given a message M, the curve parameters ($\mathbf{G}$ generator of the curve and L order of the curve), the secret key s, and a public key A (a point on the curve), these are the step to compute the signature:

1. compute h as the SHA-256 hash of the 32 bytes random data, compute s, as we have seen above, and compute $prefix$ as the second half of h
2. compute SHA-512($prefix \| M$) and interpret the 64 bytes result as a little endian integer r
3. compute $R = [r]\mathbf{G}$
4. compute SHA-512($R \| A \| M$) and interpret the 64 bytes result as a little endian integer k
5. compute $S = (r + k \cdot s) \bmod L$

The signature is (R, S), and it is usually found as a 64 bytes string, concatenation of R and S.

Given a message M, a 64 bytes string representing the signature, the curve parameters ($\mathbf{G}$ generator of the curve and L order of the curve), and a public key A (a point on the curve), the correctness of the signature can be checked decoding the first half of it as a point R, the second one as an integer S, and decoding (see Section 1.2.7) the public key A as a point A'. If one of the decoding fails, the signature is incorrect; otherwise, if the decoding proceeds successfully, we have to compute SHA-256($R||A||M$) and interpret the 64 bytes result as a little endian integer k.
The final check to do is:

$$CheckSign_{k_{pub}}(m, Sign) \rightarrow [S]\mathbf{G} = R + [k]A'$$

1.2.6 ECDH

The Diffie-Hellman algorithm is an anonymous key agreement protocol, between online parties. It is odd found it in a standard, used mainly for email communication, and thus an asynchronous message exchange.

As the other curve based algorithms used in OpenPGP, also ECDH works with a pre-computed curve, in particular with the Curve25519.

The idea behind a Diffie–Hellman cryptosystem is to exchange a symmetric key over an insecure channel. To do this, a user A can compute his own public key, generating a 32 random bytes (we will call them a) and give them to the `X25519` function [18, Section 5, p. 10] together with a 32 bytes string formed by one byte representing the number 9 (u-coordinate of **G** for the curve Curve25519), followed by 31 bytes representing the number 0.
The result of the `X25519` function is the public key K_A.

Two users (A and B) that want to communicate, have to generate and exchange their public key. Then A can compute the shared secret $K = \texttt{X25519}(\texttt{a}, \texttt{K}_\texttt{B})$ with $K_B = \texttt{X25519}(\texttt{b}, \texttt{9})$ and B can compute the same shared secret $K = \texttt{X25519}(\texttt{b}, \texttt{K}_\texttt{A})$, with $K_A = \texttt{X25519}(\texttt{a}, \texttt{9})$, 9 is, as above, a 32 bytes string formed by one byte representing the number 9, followed by 31 bytes representing the number 0. A and B can now exchange secret message, using a symmetric key, derived with a key derivation function, using K, K_A, K_B [17, Section 13.4, p. 88].

1.2.7 Encoding and Decoding Curve Point

A point on a curve is represented with two coordinates(x, y).
Sometimes to improve the efficiency, a point can be encoded, in order to transmit only one coordinate and the sign of the other one, so it can be recovered from the equation of the curve.

In OpenPGP a flag before a point indicates the way the point is transmitted. There are two possible flags: 0x04 and 0x40

0x04**:** This flag is found only on a point used with ECDSA or ECDH. It indicates that the point is not compressed and the first half of the string that follows is the x coordinate and the other half is the y one.

0x40**:** This flag indicates that the point is compressed, and contains the x coordinate with the sign of the y one.

For ECDSA or ECDH an octet string can be decoded following this steps:

1. The first octet of the string indicates the sign for y. If it is equal to 02_{16}, the sign must be set to 0; if it is equal to 03_{16} the sign must be set to 1, otherwise the decoding fails.

2. The rest of the string is the big endian representation of the coordinate x. Verify that $x \in [0, p-1]$ with p parameter of the curve, if not the decoding fails.

3. Compute $\alpha = x^3 + a \cdot x + b \bmod p$ with a and b parameters of the curve.

4. Compute the square root of α, $\beta = \sqrt{\alpha}$

5. If $sign = \beta \bmod 2$, β is the y-coordinate, otherwise $p - \beta$ is.

For EdDSA instead, the way to retrieve the two coordinates is the following:

1. Verify that the number of octet of the string is 32, if not raise an error.

2. Interpret the string as a little endian representation of an integer number. The 255_{th} bit of this number represent the least significant bit (i.e. the sign) of the x-coordinate. The y-coordinate is found clearing this bit.

3. From the curve equation we know $x^2 = \frac{(y^2-1)}{(d \cdot y^2+1)(mod p)}$, where d and p are parameters of the curve

4. Then we define $u = y^2 - 1$ and $v = d \cdot y^2 + 1$, thus $x^2 = \frac{u}{v}$.

5. To compute the square root of $\frac{u}{v}$ we should first define the candidate root $x = \frac{u}{v}^{\frac{p+3}{8}} = u \cdot v^3 (u \cdot v^7)^{\frac{p-5}{8}} \bmod p$

6. There are three cases:

   - $v \cdot x^2 = u \bmod p$: x is a square root
   - $v \cdot x^2 = -u \bmod p$: $x = x \cdot 2^{\frac{p-1}{4}}$ is a square root
   - otherwise: decoding fails

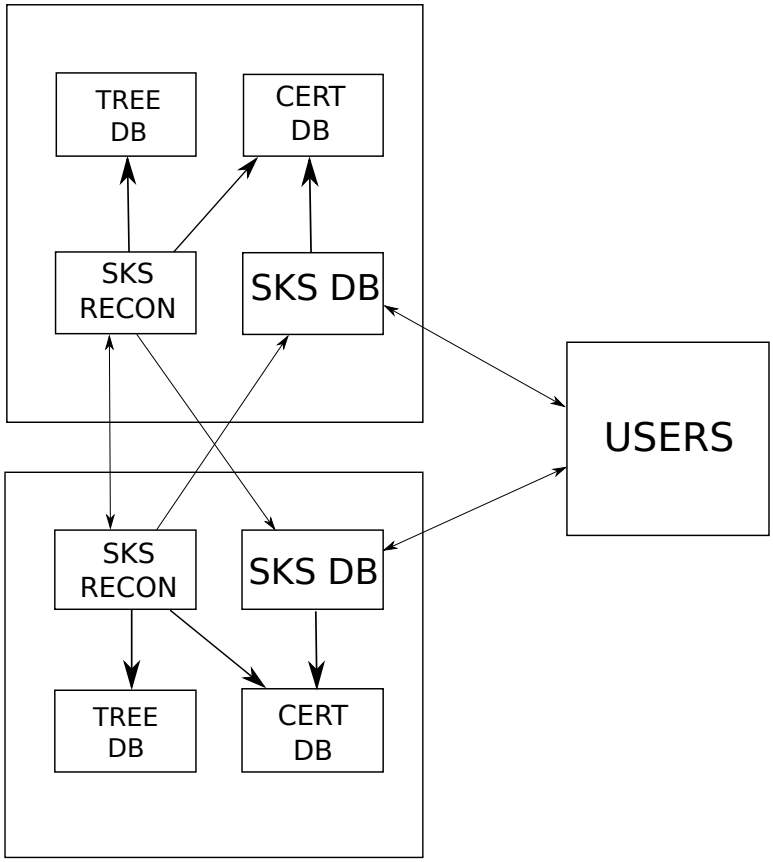
7. Finally we can use the sign computed at 2, if $x = 0$ and sign is 1 the decoding fails; otherwise we check $sign = x \bmod 2$, if true the x-coordinate is the one computed at 6, otherwise $x = p - x$

In Section 6 of [13] can be found a python implementation of all the algorithms regarding the EdDSA cryptosystem.

1.3 SKS keyserver

The certificates are distributed through keyservers, maintained by volunteers, on which works SKS [15]. An SKS keyserver is synchronized with the other in order to guarantee that all the certificates is available in all the keyservers.

Figure 1.3: How two SKS keyserver works



The SKS keyserver is divided in two main processes, *sks db* and *sks recon*,

that communicate with two Berkeley DB; for a sketch see Figure 1.3. The first one is a daemon that deals with the normal process that typifies a keyserver. It answers to all the web requests, handling the upload of certificate by the users, and providing the stored keys. The other one, deals with the reconciliation between keyservers, and the synchronization between the tree and the database of the certificates. The *sks recon* doesn't synchronize directly the certificates, but their MD5 hash. Finished this synchronization, it fetches the certificates querying the *sks db* with the recovered MD5 values.

1.3.1 Synchronization algorithm

In this section will be shown the main idea behind the synchronization algorithm.

The synchronization algorithm is designed to ensure the best communication performance with the minimum bandwidth consumption. The main idea is based on the set reconciliation problem [20]. Given two sets S_A and S_B, their differences $\Delta_A = S_A \setminus S_B$ and $\Delta_B = S_B \setminus S_A$, and the dimension of the differences m_A and m_B, we can define $m = m_A + m_B$ as the maximum number of differences between the two sets.

Each set is represented with a checksum that can be used to determine the differences between the sets. Given a set $S = \{x_1, x_2, \ldots, x_n\}$, where x_i is an element of the set, we can define the characteristic polynomial (i.e. the checksum) as

$$\chi_S(Z) = (Z - x_1)(Z - x_2)\ldots(Z - x_n)$$

As we can see the zeros of the characteristic polynomial, are the elements of the set.

Each element of the set is mapped into a field $\mathbb{F}_q$ with $q \geq 2^b$ and b length of the bitstring representing the mapped elements.

Considering the ratio between the characteristic polynomial of the two sets:

$$\frac{\chi_{S_A}(Z)}{\chi_{S_B}(Z)} = \frac{\chi_{S_A \bigcap S_B}(Z) \cdot \chi_{\Delta_A}(Z)}{\chi_{S_A \bigcap S_B}(Z) \cdot \chi_{\Delta_B}(Z)} = \frac{\chi_{\Delta_A}(Z)}{\chi_{\Delta_B}(Z)}$$

we can use directly the ratio $\frac{\chi_{\Delta_A}(Z)}{\chi_{\Delta_B}(Z)}$ to recover the differences. The idea to efficiently compute the ratio (i.e. to avoid the computation of the polynomial division), is to divide the results of the evaluation of the polynomials $\chi_{S_A}(Z)$

and $\chi_{S_B}(Z)$ on a set of points in $\mathbb{F}_q$, and use the result of the division, to interpolate the required rational function.

Given d_1 and d_2 degrees of the numerator $P(Z) = \sum_i p_i Z^i$ and denominator $Q(Z) = \sum_i q_i Z^i$ of the rational function, and a set V of $d_1 + d_2 + 1$ pairs $(k_i, f_i) \in \mathbb{F}^2$, there is a unique rational function such that $f(k_i) = f_i$ for each (k_i, f_i). Each pair (k_i, f_i) implies thus a linear constraint of the coefficient of numerator and denominator:

$$k_i^{d_1} + p_{d_1-1}k_i^{d_1-1} + \ldots + p_0 = f_i \cdot (k_i^{d_2} + q_{d_2-1}k^{d_2-1} + \ldots + q_0)$$

The interpolation is accomplished by solving a system with all the equations implied by the elements of V.

Before starting the reconciliation, the two hosts have to agree on a set of evaluation points, on which the ratio $\frac{\chi_{S_A}(Z)}{\chi_{S_B}(Z)}$ will be computed.

Once computed the interpolation, we have the coefficient and we can recover the missing elements computing the zeros of the numerator and denominator.

Summing up, to perform the reconciliation, an host A send its checksum to an host B, that computes the two differences (Δ_A, Δ_B) and sends Δ_B to A.
The reconciliation fails, if the maximum number of differences m is too high. To avoid this we can define:

- $\bar{m}$ as the maximum allow value for m. Above this value the reconciliation fails.
- k the probability of detecting that the reconciliation is not possible, as $(\frac{|S_A|+|S_B|}{2^b})^k$ if $m > \bar{m}$

The computational complexity of the function that, given the checksum cs_A, cs_B of the two sets S_A, S_B, compute their differences, is $O(b{\cdot}\bar{m}^3{+}b{\cdot}\bar{m}{\cdot}k)$. As you can see is cubic in $\bar{m}$; to avoid this we can choose a fixed value for $\bar{m}$ and split the set in p (called *partitioning factor*) as equal as possible subsets. If the reconciliation algorithm seen above fails due to $m > \bar{m}$, we have to execute it on all the partition of the original set. If it fails also on a partition we can split that partition in p subpartitions and continue recursively until the reconciliation succeeds in all partitions.

To avoid the continuous computing of the checksum, for each host that wants to reconcile we can further improve the efficiency, representing the split in partitions as a tree.

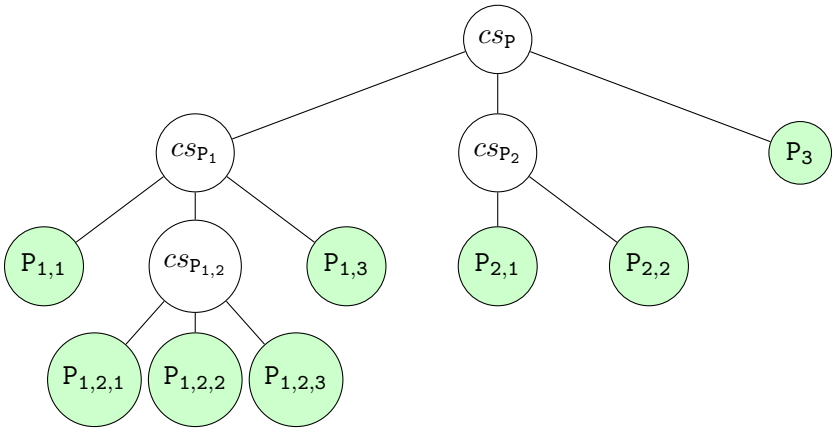


Figure 1.4: Sketch of the partition tree. The green nodes represent the leafs

In Figure 1.4 there is an example of the partition tree. The leafs of the tree contain the elements of the partition; the nodes instead, the checksum of the set composed by the union of all the elements of each leaf found, descending from that node (e.g $\mathtt{cs}_{\mathtt{P}_1}$ contains the checksum of the set $S = \mathtt{P}_{1,1} \cup \mathtt{P}_{1,2,1} \cup \mathtt{P}_{1,2,2} \cup \mathtt{P}_{1,2,3} \cup \mathtt{P}_{1,3}$), and the number of the elements of this set. The union of the sets in the leafs composes the complete set $\mathtt{P}$. In this way we can avoid the computation of the checksum for each invocation of the reconciliation function, but we need to edit it accordingly every time an element is added or removed to and from the tree. We also have to be sure that any node in the tree, has a unique index, equals in the tree of each other hosts, so that we can put the elements in the same partition on every host, to avoid duplicates.

1.3.2 Synchronization of SKS keyserver

In an SKS keyserver the tree is composed as follows:

- every node contains the checksum computed as is shown in Section 1.3.1, and the number of the elements of the set on which the checksum is calculated.
- every leaf represents a partition of the set composed by the MD5 di-

gests of the binary representation of the certificates. To avoid that the same key, represented with a different packet order, generates two different hash, this one is calculated on the binary representation of the packets, ordered by their tag and content.

The MD5 digests are mapped in a field $\mathbf{F}_q$ with
$q = 305128895516023225051275203525794373 39$. The *partitioning factor* p is equal to $2^{bitquantum}$, where *bitquantum* is a value that can be manually set. The default value for *bitquantum* is 2.
Before starting the reconciliation, two SKS hosts, exchange some of their settings to ensure that they are equal. The settings in question are:

- filters: each server has some filters that shows how the server handles the incoming certificates. The default value is: `yminsky.dedup; yminsky.merge`. The first value indicates that a server Delete and then UPload a new key: this is needed because the keyserver works with a key-value storage, thus the merged certificates are not updated but deleted and then re-inserted. The second one means that the server performs the merge between two certificates represented by the same primary key, to combine the different packets.

- bitquantum: the value of the *bitquantum* variable, to ensure that the two hosts use the same tree structure.

- mbar: the value of $\bar{m}$. The default value is 5.

If these values are not equal, the servers don't start the reconciliation.
The reconciliation algorithm recovers only the missing hashes and store them to a file. After that, another function deals with the recover of the certificates.

Every node is recognized by an index that has to be calculated in the same way in all the hosts. Given a node, the algorithm 1.3.1 is the one used to compute the indexes of its child nodes in an SKS keyserver. The key idea is to incrementally compute the indexes, starting by the one of the parent node and extend its length by *bitquantum* bits, following a certain pattern. The algorithm takes in input the extended index of the original node (not yet updated), called `bs`, the bit from which the update of `bs` should starts, and the final length of the new index.

Algorithm 1.3.1: Algorithm used in sks to compute the indexes for the children of a node

input : bs: at the first call is the bistring representing the index of the starting node (i.e. the one for which we need the index of the children) with the length extended by bitquantum, at the other calls is the updated index
bit: actual considered bit, starts from the latest bit (the most significant) of the bistring representing the index of the starting node + 1
len: len + bitquantum– Final length of each new index

output : Set of index for all the children of the starting node
parameter: `set` (string, bit): function explained in the text
`unset` (string, bit): function explained in the text

```
1  Function child_keys(bs, bit, len):
2      if bit ≥ len then
3          return {bs}
4      else
5          set(bs, bit)
6          keys1 = child_keys(bs, bit + 1, len)
7          unset(bs, bit)
8          keys2 = child_keys(bs, bit + 1, len)
9          return keys1 ∪ keys2
10 End Function
```

The algorithm computes the indexes of all the child nodes, filling the new bits of it, using the set and unset function. Finally, it returns a set with all the indexes of the child nodes.

The set and unset functions, given a bitstring (`ba`) and the position in bit that has to be updated in input, return the modified bitstring. They calculate first the `byte_pos` with the remaining `bit_pos` of the interested position, and then the ASCII code of the char in the previously computed `byte_pos` position of the string `ba`. Then in the set function the new character is the one corresponding to the ASCII code found calculating the bitwise `or` between 1 shifted by `7 - bit_pos` position, and the ASCII code of the old character. The unset function instead, computes the bitwise `and` instead of the `or`, between the bitwise negation of 1 shifted by `7- bit_pos`, and the ASCII code of the old character. Once computed the new character, the two

functions return the ba string with the new character in place of the old one.

Algorithm 1.3.2: Algorithm used to compute the index starting from an MD5 hash of the certificate (interpreted as a number), to know where to store it

input : depth Depth of the actual node
hash Data to be inserted
output : index
constant: bitquantum (defined during the settings of the tree)
rmask (i) = 0xFF $\ll$ (8 - i)
lmask (i) = 0xFF $\gg$ (8 - i)
chartoint (c): return the ASCII code of the character c
land: bitwise AND operator
lor: bitwise OR operator

```
1  Function string_index(depth, hash):
2      lowbit ← depth · bitquantum
3      highbit ← lowbit · bitquantum- 1
4      lowbyte ← lowbit/ 8
5      lowbit ← lowbit mod 8
6      highbyte ← highbit/ 8
7      highbit ← highbit mod 8
8      if lowbyte = highbyte then
9          byte ← chartoint(hash[lowbyte])
10         return (byte ≫ (7 - highbit)) land (lmask(highbit- lowbit + 1))
11     else
12         byte1 ← chartoint(hash[lowbyte])
13         byte2 ← chartoint(hash[highbyte])
14         key1 ← (byte1 land (lmask(8 - lowbit))) ≪ (highbit + 1)
15         key2 ← (byte2 land (rmask(highbit + 1))) ≫ (7 - highbit)
16         return key1 lor key2
17 End Function
```

The hashes of the key has to be put in the same node in all the SKS keyserver. Given an hash, to know where to store it, we can start from the root of the tree and use the algorithm 1.3.2 to know the index of the children in which the subtree, where the hash has to be put, starts. Once found the children the algorithm should be re-used for the same reason, recursively, until a leaf is detected.
Having the depth of the node that we are considering, the algorithm com-

putes the index of the next child node needed. In the first 2 lines, it computes the variables `lowbit` and `highbit`, and then splits them in bytes and remaining bit (lines 4–7); now if the `highbyte` and the `lowbyte` values are the same, the algorithm fetches the ASCII code of the `lowbyte` character of the `hash` string, and returns a logical shift to the right by `7 - highbyte` positions of that code, in a bitwise `and` with the left mask of `highbit - lowbit - 1` (lines 9–10); otherwise it fetches the two characters in position `lowbyte` and `highbyte`, and computes:

$\texttt{key}_1$ as the logical left shift of the bitwise `and` of the ASCII code of the first character with the left mask of `8 - lowbit`, by `highbit + 1` positions (lines 12,14).

$\texttt{key}_2$ as the logical right shift of the bitwise `and` of the ASCII code of the second character with the right mask of `highbit + 1` by `7 - highbit` positions (lines 13,15).

then returns the bitwise `or` between $\texttt{key}_1$ and $\texttt{key}_2$

The algorithm starts to reconcile from the root of the tree and if needed, it continues on its children. When the reconciliation on a node (or a leaf) ends, the recovered hash values are not inserted in the tree but stored in a file. Afterwards another function deals with the recovering of their respective certificates.

Client requests:	Server response:
• `ReconRqst_Full`	• `SyncFail`
• `ReconRqst_Poly`	• `Elements`
• `Elements`	• `FullElements`
• `Done`	• `Error`
• `Flush`	

Figure 1.5: Type of requests that an SKS client or server can perform

Given a node of a tree, a client, can perform two type of requests:

ReconRqst_Full: containing the number of the elements in it and the elements. It is performed if the considered node is a leaf or if the size

of the set on which the checksum is computed is less than the upper bound $\bar{m}$. This request contains all the elements of the leaf or the elements of the set on which the checksum of the node is computed.

ReconRqst_Poly: performed if the previous condition is not verified. This request contains the index of the node, the number of elements of the considered node (i.e. the ones on which the checksum is computed), and the checksum.

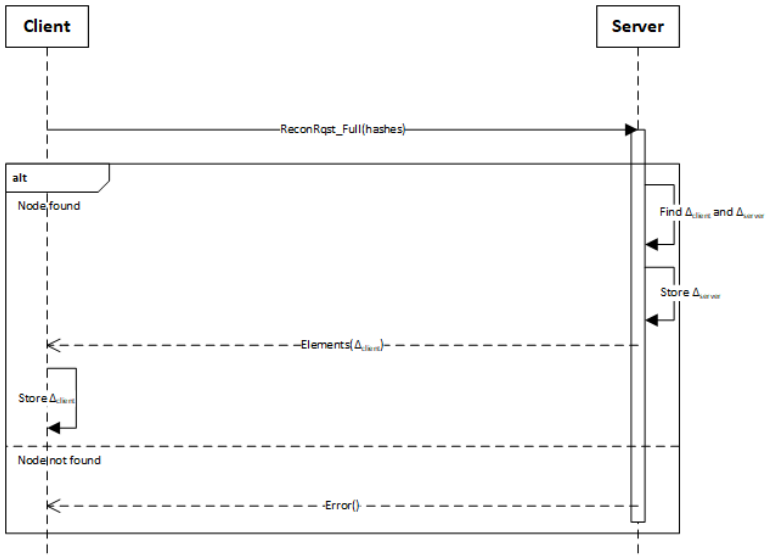


Figure 1.6: Flow of the requests starting from a ReconRqst_Full

A server that receives a `ReconRqst_Full` request, computes the two differences $(\Delta_{client}, \Delta_{server})$, where Δ_{client} is the set of hashes missing from the client and Δ_{server} is the set of hashes missing from the server, directly on the set (i.e. as a differece between two sets, without using the interpolation). The second one is stored, while the first one is sent back to the client. At this point also the client can store the differences, and the reconciliation of that node of the tree is finished (and thus also on its children).
In Figure 1.6 is shown the sequence diagram for the `ReconRqst_Full` request.

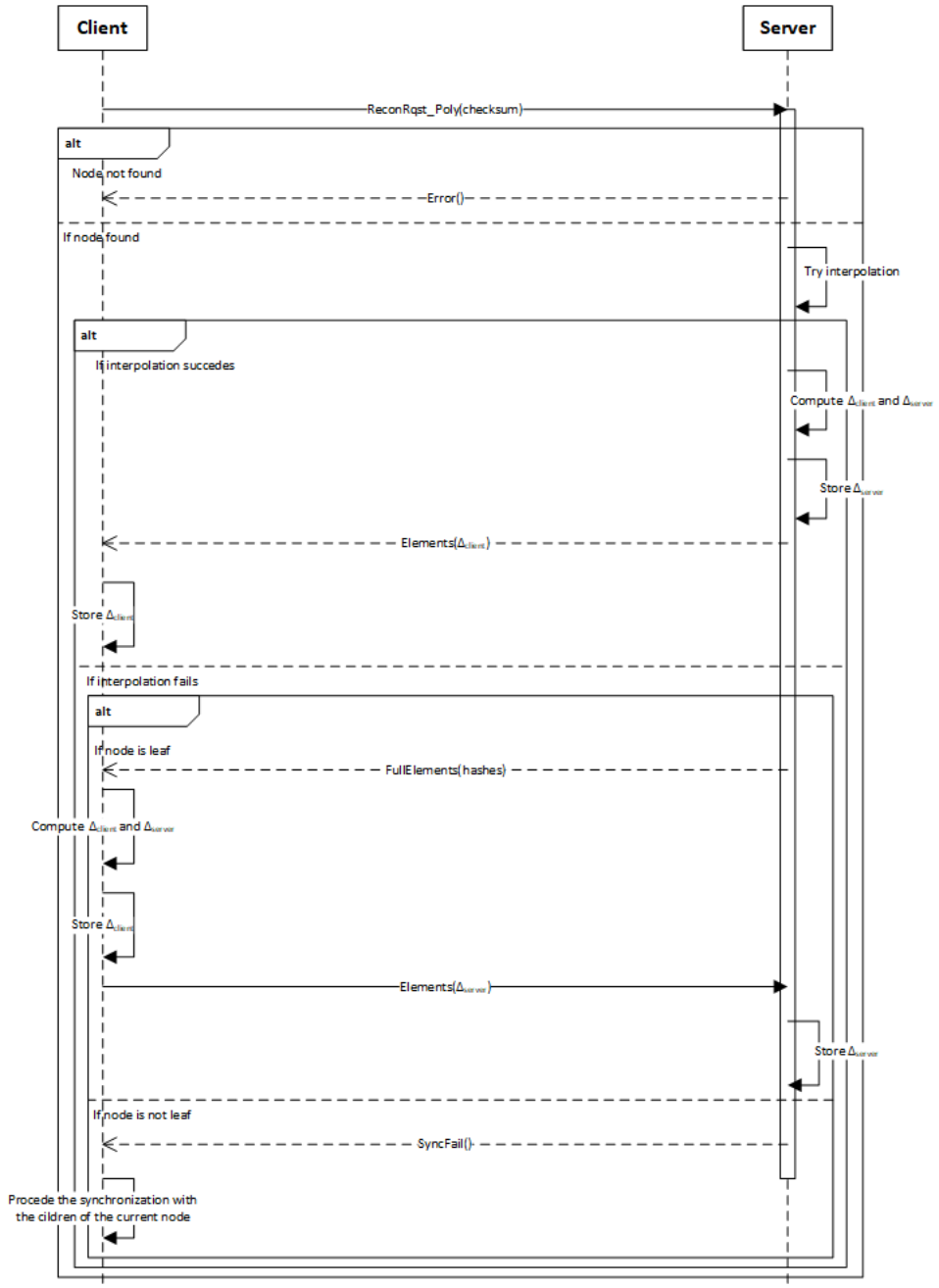


Figure 1.7: Flow of the requests starting from a `ReconRqst_Poly`

If a server receives a `ReconRqst_Poly` instead, computes the differences using the interpolation function; if it gives a result, the server stores the Δ_{server} differences and sends back the Δ_{client} one, otherwise (if the interpolation is impossible) answer with:

- a `FullElements` reply, if the considered node is a leaf or if the size of the set on which the checksum is computed is less than the upper bound $\bar{m}$. This request contains all the elements of the leaf or the elements of the set on which the checksum of the node is computed.
- a `SyncFail` request, if the previous condition is not verified.

If the client receives a `FullElements` response, it sends back all the server missing elements, otherwise it continues the synchronization with the children of the considered node. In Figure 1.7 you can find a sequence diagram for a `ReconRqst_Poly` request.

The server doesn't close the connection until it receives a `Done` request by the client, but it is suddenly close if an `Error` response is generated by the server. The `Error` response is not sent immediately but the server wait for a `Flush` request from the client.

A `Flush` request is sent by the client if it doesn't receive an answer by the server and the number of unanswered sent requests is over a certain predefined limit. If a server receives a `flush`, it sends the `Error` responses accumulated, and continues to fetch the other requests.

When the reconciliation algorithm ends, a function handle the recover of the key from the server. It sends a `POST` request to the other *sks db* daemon, composed by the number of the recovered hashes and, for each of them, both the size of the hash and the actual hash. The server answers with the number of the returned keys, followed by, for each of these, both the size and the certificate.

When a key is added to the server, the tree is not automatically updated: there is a daemon that handle the synchronization between the certification database and the tree. If a key is merged, the hash could change: to deal with this modification, the old hash should be removed from the tree, and the new one inserted.

Chapter 2

PEAKS – A reenginered OpenPGP Keyserver

In this chapter will be shown how the new implemented keyserver works, showing how the database is structured, how the implementations of the various software are done and the choice that has been made.

2.1 PEAKS

The basic idea was to create an SKS keyserver that could handle better the amount of data using a relational database, be better updated and maintained, and verifies that the certificates have not security issues.

Table 2.1: List of the peaks software, divided into daemons and single launch applications

Daemon	Application
PKS	*unpacker*
reconciliation daemon	*analyzer*
	dump import

The certificates are stored and unpacked in a MySQL database. The software composing the study are shown in Table 2.1

2.1.1 Database

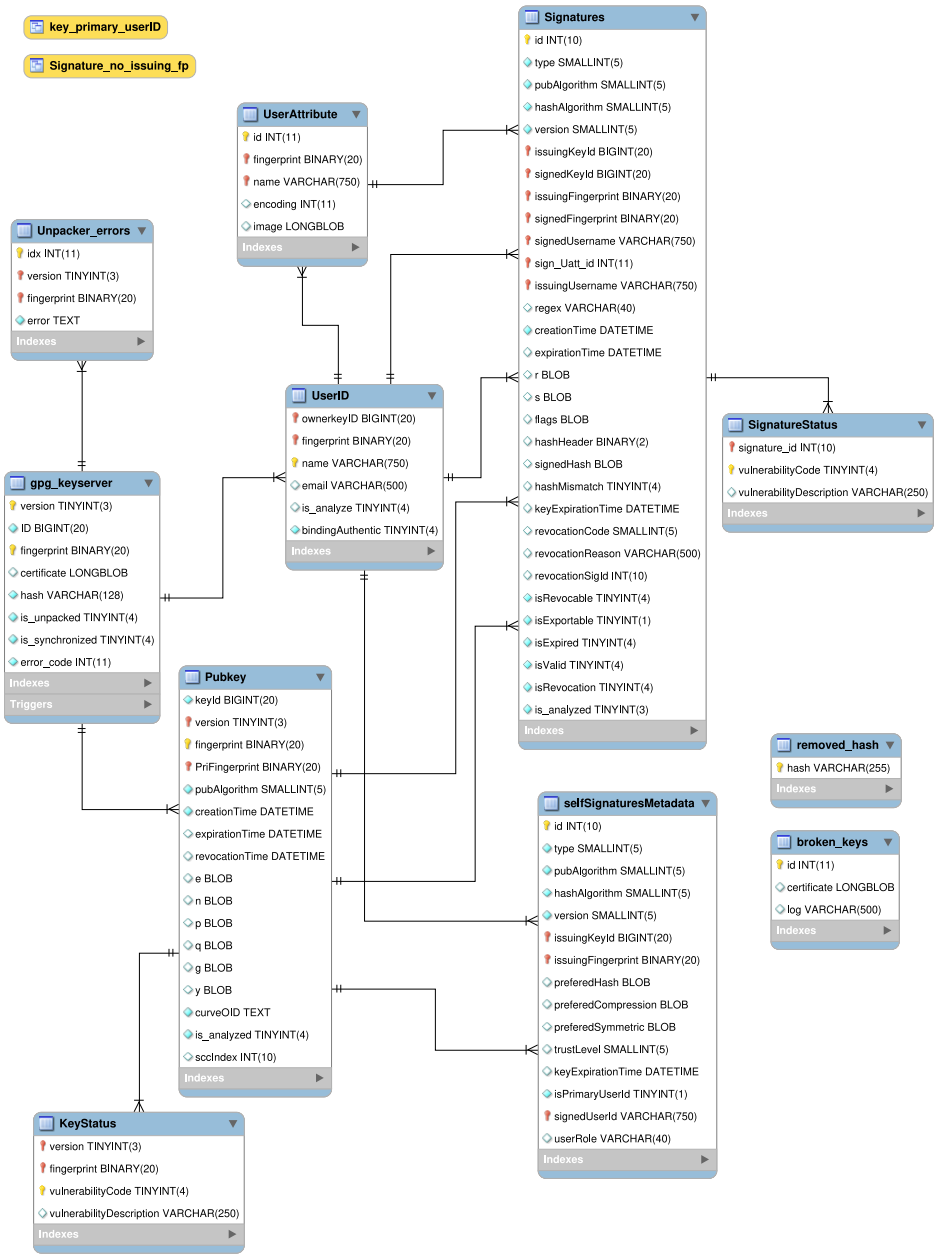


Figure 2.1: ER schema of the used database

The ER schema of the database is in Figure 2.1. The tables used are:

`gpg_keyserver:` contains all the certificates. Each certificate is identified by the version and the fingerprint of the primary key. Other information in the table are:

- The key ID. It could used to query the certificate by the users.
- The hash used for the synchronization between keyservers.
- The flag `is_unpacked`. It can be:
  - -1: if the unpacking of the certificate has generated an irreparable error
  - 0: if the certificate has not been unpacked yet.
  - 1: if the unpacking ends without errors.
  - 2: if not all the packets of the certificate has been exported. The reason can be found in the `Unpacker_Errors` table.
- The flag `is_synchronized`. It is used by the recon daemon to maintain the tree updated.
- The `error_code` related to the parsing of the certificate. The list of the error code can be found in Table 2.2.

`Pubkey:` contains information about keys and subkeys. The field in the table are:

- `Key ID`, `version` and `fingerprint`: used to recognize the key.
- `priFingerprint`: NULL if the key is primary, otherwise contain the fingerprint of the primary key of the certificate.
- `pubAlgorithm`: the code representing the asymmetric algorithm of the key.
- `creationTime`, `expirationTime`, `revocationTime`: Three fields that contain the creation, the expiration (if any) and the revocation (if any) time of the key. The first field is the only one contained in a *public key* packet, the other two are taken from the *signature* table.
- The values of the public key (`n`, `e`, `p`, `q`, `g`, `y`, `curveOID`). If a value is not a part of the public key its field if NULL.

- `is_analyzed`: this field is 1 if the key has been analyzed, 0 otherwise.

`Signatures:` contains information about the signatures. The field in the table are:

- `id`: an auto–increment values that identifies each signature in the table.
- `type`: the type of the signature represented with an integer value.
- `pubAlgorithm`: the code representing the asymmetric algorithm used to perform the signature.
- `hashAlgorithm`: the code representing the hash algorithm used to compute the hash.
- `version`: the version of the signature.
- `issuingKeyId`, `issuingFingerprint`, `issuingUsername`: the values of key ID, fingerprint, and user ID of the key that has issued the signature.
- `signedKeyId`, `signedFingerprint`, `signedUsername`: the values of key ID, fingerprint and user ID of the key on which the signature is computed.
- `sign_Uatt_id` the ID of the user attribute that represents a field in the `UserAttribute` table. NULL if the signature is not computer over a user attribute.
- `regex`, `flags`: value of the *regex* and `flags` subpackets (if any).
- `creationTime`, `expirationTime`: Three fields that contain the creation and the expiration (if any) time of the signatures.
- `r`, `s`: the values of the signature, for the RSA algorithm `r` is empty.
- `hashHeader`: the value of the header of the hash.
- `signedHash`: the value of the hash that has been signed. This value is not in the *signature* packet but is computed by the *unpacker*.
- `hashMismatch` this value is equal to 1 if the hash header doesn't match with the one of the hash computed following the rules in the RFC 4880, 0 otherwise.

- `keyExpirationTime`: this value is found only on a self signature. Indicates the expiration time of the key on which the signature is performed.
- `revocationCode`, `revocationReason`, `revocationSigID`: the first two indicate the code and the reason of the revocation, the third one is related to the `id` of the signature that has compute the revocation.
- `isRevocable`: 1 if the signature can be revoked, 0 otherwise.
- `isExportable`: 1 if the signature can be exported from a local keyring, 0 otherwise.
- `isExpired`: 1 if the signature is expired, 0 otherwise.
- `isValid`: 1 if `isExpired` is 1 or there is a revocation on that signature, 0 otherwise.
- `isRevocation`: 1 if the type of the signature is a revocation one, 0 otherwise.
- `is_analyzed`: 1 is the signature has been already analyzed, 0 otherwise.

`User ID:` contains information about the user ID. The field of the table are:

- `ownerkeyID`, `fingerprint`: used to identifies the certificate on which the *user ID* packet belongs.
- `name`, `email`: the first one is the content of the packet, the second one only the email (if any).
- `is_analyzed`: flag equals to 1 if the email has been analyzed, 0 otherwise.
- `bindingAuthentic`: the response of the analysis done over the email (not performed by this study).

`User Attribute:` contains information about the user attribute. The field of the table are:

- `id`: an auto–increment value that represent the user attribute.
- `fingerprint`, `name`: used to identifies the bind between the user ID and the user attribute.
- `encoding`: the encoding of the image, until now only the jpeg encoding (tag 1) is defined.
- `image`: the binary string of the image.

`SelfSignatureMetadata:` Contains the *self signatures* performed by a key over a user ID in the same certificate. The columns are different from the Signature, because this table is used to fetch the most important user ID packet among the ones in a certificate. The signatures in this table are placed also in the `Signatures` table. The field of the table are:

- `id`: an auto–increment value used to identifies each self signature.
- `type`, `pubAlgorithm`, `hashAlgorithmIndex`, `version`: same as the `Signatures` table.
- `issuingKey`, `issuingFingerprint`: the key ID and fingerprint of the issuing key (in this table it coincide with the signed key).
- `preferredHash`, `preferredCompression`, `preferredSymmetric`: three fields found in a subpacket of a signature, that give some info about the preferences of the owner of the signed key.
- `trustLevel`: level of trust of the signatures as specified in 1.1.2.
- `keyExpiration`: same as the `Signatures` table.
- `isPrimaryUserId`: a flag that indicates if the signed user ID is the most important in the certificate.
- `signedUserId`: the user ID on which the signature is computed.
- `userRole`: a string that explain the role of the user.

`Unpacker_errors:` Contains all the errors occurred during the unpacking of a key. The list of the errors can be found in 2.2.

`KeyStatus, SignatureStatus:` Contains the vulnerabilities found during the analysis of keys and signatures. The list of vulnerabilities can be found in 2.4 and 2.5.

`removed_hash:` Contains the hashes that are been removed from the gpg_keyserver table. This table is used by the *recon daemon*, to maintain updated the tree.

`broken_keys:` Contains all the certificate that are impossible to read and/or decode.

The database has also two views (`key_primary_userID` and `Signature_no_issuing_fp`) and one trigger. The first view contains a projection of the `SelfSignatureMetadata` table in order to have the primary User ID for each primary key.
The other one is used by the unpacker daemon, to update the fingerprint of the issuing in the `Signatures` table (not always the issuing fingerprints are in the Signature packets, so we have to fetch it using the issuing key id). The trigger instead is used to fill the `removed_hash` table when a key is updated in the `gpg_keyserver` table; it saves the old hash in the `removed_hash` table and stores the new one in `gpg_keyserver`.

2.1.2 PKS
The new *sks db* daemon

The PKS represent the *sks db* daemon in an SKS keyserver. As is shown in Section 1.3 the *sks db* daemon, handles the upload of the new certificates, performing the merge (if needed), and it provides them to the users and the other servers.

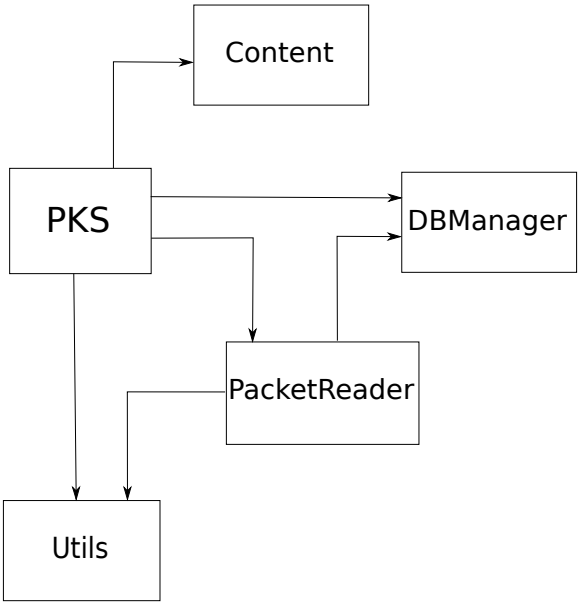


Figure 2.2: Sketch of the classes of the PKS daemon

The classes that compose the PKS are shown in Figure 2.2

PKS

The main class is the `PKS`, it uses the `content` and the `utils` class to provide organized contents in the web pages; it is also appointed to handle the requests of the certificates coming from the *sks recon* daemon with a list of the MD5 hashes. The useful URLs are:

- `host/pks`: that respresents the homepage. From here is possible search and upload the certificate.
- `host/lookup?search=SEARCHSTRING&op=index`: the search page. The `SEARCHSTRING` is the name of the searched user.
- `host/pks/lookup?op=vindex&search=ID`: the page that shows the content of the certificate. The `ID` string can be the key ID or the fingerprint of the certificate requested.
- `host/pks/lookup?op=get&search=ID`: the certificate page. It shows the certificate encoded with the armored method.

content

The `content` contains all the HTTP objects represented as structs including the dynamic content of the web pages (e.g. the `homepage`, the `submit_form`, the list of found key).

utils

The `utils` class contains the functions to elaborate and encode the HTML code, before displaying the page to the users. It also includes some structs that are used to handle the certificate during the elaboration of the code.

PacketReader

The `PacketReader` is the part that manages the parsing of the the certificate, uploaded in an armored encoding, and deal with the possible generated errors. The error handling is explained in Section 2.1.2. During the parsing it fetches the fingerprint and the version from the *primary key*, and asks to the `DBManager` if there already is a certificate with the same `version` and `fingerprint` to (eventually) perform the merge, that is examined in Section 2.1.2. After that it uses two structs, situated in the `utils` class, to virtually represent the tables `gpg_keyserver` and `UserID` of the database, filling them with all the information needed, and pass them to the `DBManager` for the upload in the database. The information inserted can be found in detail in Section 2.1.2

DBManager

The `DBManager` class manages the communication with the MySQL database for both the `packetReader` and the `PKS` classes. It contains all the performed queries and the structs used to represent the tables of the database.

Error handling

The possible errors are divided in two part. The key error category, that represents the error caused by a non–meaningful key (i.e. if a key doesn't follow the rules in the RFC 4880, or if a packet doesn't represent a public key)

and the parsing error category, that represents an error during the creation of the object key. The key errors are shown in Table 2.2:

Table 2.2: Possible errors generated during the verification of the meaningfulness of a key, excepto for the last one that is generated during the merge function

Name	Code	Prevent the upload
NotExistingVersion	1	YES
BadKey	2	YES
NotEnoughPackets	3	MAY
FirstPacketWrong	4	MAY
SignAfterPrimary	5	NO
AtLeastOneUID	6	NO
WrongSignature	7	NO
NoSubkeyFound	8	NO
Ver3Subkey	9	NO
NoSubkeyBinding	10	NO
NotAllPacketsAnalyzed	11	NO
NotAPublicKey	12	YES
DifferentKeys	14	NO

`NotExistingVersion:` caused when the certificate has a version different than 2, 3 or 4. The version of a certificate is the one of its primary key.

`BadKey:` caused when the armored representation of the binary stream doesn't represent an OpenPGP certificate.

`NotEnoughPackets:` caused when there is only one packet in a certificate (a certificate should have at least two packets: a *primary key* and a *user ID*). If the only packet found is a *primary key* packet, the certificate is uploaded, otherwise it is rejected and inserted in the `broken_keys` table.

`FirstPacketWrong:` caused when the first packet of a certificate is not a *primary key* packet. The daemon upload the certificate, only if a *primary key* is found; otherwise it is rejected and inserted in the `broken_keys` table.

`SignAfterPrimary:` caused where there is a Signature Packet after the first one that is not a revocation or a direct signature.

`AtLeastOneUID:` caused when no User ID packets are found.

`WrongSignature:` caused when a signature after a User ID (or a user attribute) is not a *certification* or a *certification revocation* signature.

`NoSubkeyFound:` caused when (in version 4) no subkeys is found.

`Ver3Subkey:` caused when (in version 3) there is a subkey.

`NoSubkeyBinding:` caused when a signature after a subkey is not a *subkey binding*, *primary key binding*, *key revocation* or a *subkey revocation*.

`NotAllPacketsAnalyzed:` caused when, during the parsing, not all the packets have been read.

`NotAPublicKey:` caused when the armored representation of the binary stream doesn't represent doesn't represent a certificate.

`DifferentKeys:` caused when you try to merge two certificates that doesn't have the same primary key.

Each error is bound with an error code. In table 2.2 there is a list of the error codes. In case of a `NonExistingVersion`, `badKey` or `NotAPublicKey` error, the key is rejected and uploaded in the `broken_keys` table, with the description of the error that causes the problem. In the other case the error is deal with, and the key is uploaded in the `gpg_keyserver` table with the corresponding error code. In case of a `DifferentKeys` error, the merge is avoided and the upload continues.

The parsing errors are instead shown in Table 2.3:

`PubkeyAlgorithmNotValid:` caused when the algorithm code found in a public key is not valid.

`PubkeyVersionNotValid:` caused when a version found in a public key is not valid.

Table 2.3: Errors generated during the parsing of the binary stream of a certificate

Name	Code	Prevent the upload
PubkeyAlgorithmNotValid	32	YES
PubkeyVersionNotValid	33	YES
LengthLEQZero	34	YES
SignaturePKANotValid	35	YES
SignatureHashNotValid	36	YES
SignatureVersionNotValid	37	YES
SignatureLengthWrong	38	YES

`LengthLEQZero:` caused when it is found a packet with a length (specified in the header) equal to zero.

`SignaturePKANotValid:` caused when the algorithm code found in a signature is not valid.

`SignatureHashNotValid:` caused when the hash found in a signature is not valid.

`SignatureVersionNotValid:` caused when a version found in a signature is not valid.

`SignatureLengthWrong:` caused when the second octet of the signature is different than 5. That octet represent the length of the hashed material and, following the rules of the RFC 4880, has to be 5.

All the parsing errors break the creation of the object key and then the armored key is automatically stored in the `broken_keys` table, with the reason that caused the error.

Merge

Two certificates can be merged if and only if their primary key packets correspond. The idea for the merging algorithm is the one used in the *sks db*. Basically all the packets of a key are stored in a structure:

- The first element of the structure is the *primary key* that represents the certificate.
- The second element is a set of signatures made on the *primary key*.
- The third element is a set of tuples ⟨user ID, list of signatures⟩. Each tuple represents the user ID packets with its signatures. The user attribute packets are treated as user ID ones and inserted in this set.
- The last element is a set of tuple ⟨public subkey, list of signatures⟩ that represents all the subkeys with theirs signatures.

Except for the *primary key*, all this items are merged each one with the corresponding element of the other certificate, avoiding repetitions. The *primary key* is the same and thus there is no need to merge it.
Once the new structure is ready, the new packet list is reconstructed, and the merging is completed.

The merge function in the PKS server is slightly modified to adapt it to the needs. The second item of the structure is a set of tuples ⟨primary key, list of signatures⟩, but is handled as in the *sks db* (i.e. the first item of each tuple is not considered). There are, instead, three new items:

- a list of pair ⟨user ID, user attribute⟩ representing the connection between user ID and user attribute.
- a list of user ID, to ensure that all the user ID packets have been inserted in the merged key.
- a list of packets that should not belong to a key (e.g. a message packet, a marker packet). These packets are inserted at the end of the packet list of the merged key.

The procedure for the merging is the same of the SKS keyserver. The first two element of the new items list are used during the combination of the ⟨user ID, list of signatures⟩ tuples, to avoid lost of information (e.g. to maintain the binding between a *user ID* and a *user attribute*); and the third one is used to collect all the inappropriate packets in order to handle all the inappropriate packets.

Insertion

One the certificate has been parsed, the PKS daemon insert it in the `gpg_keyserver` table. The fields of the table are filled with:

- `version`: the version of the *primary key*.
- `keyID`: the key ID of the *primary key*.
- `fingerprint`: the fingerprint of the *primary key*.
- `certificate`: the binary stream of the certificate.
- `hash`: used in the synchronization, computed as we have seen in 1.3.2.
- `is_unpacked`: flag set to 0.
- `is_synchronized`: set to 0.
- `error_code`: the code of the generated error (if any).

After the certificate, also the information about the user ID packets are inserted:

- `ownerkeyID` and `fingerprint`: the same of the *primary key*.
- `name`: the content of the packet.
- `email`: the email extracted from the `name` (if any).
- `is_analyzed`: set to 0.
- `bindingAuthentic`: set to 0.

2.1.3 Unpacker

The unpacked daemon deals with the unpacking of the key. It extracts the packets of the certificate having the `is_unpacked` field set to 0 in the `gpg_keyserver` table. The daemon uses the same structure seen in the merge function. In case of error it follows the same idea of PKS: if the structure can be recovered, it does, otherwise it generates an error and continues with the next key. In the second case the flag `is_unpacked` in the `gpg_keyserver` table is set to -1.

Once the structure is created, it is parsed to verify that there are no packets out of place (for example a signature that should be after a user ID, found after a subkey). If an improper packet is found, it is simply removed and the alteration is registered in the `Unpacker_errors` table. The altered certificates are marked `is_unpacked` field within the `gpg_keyserver` table.

Having a structure that seems correct, the daemon continues with the unpacking and stores all the info about the packets in the database. During this phase, is also computed the hash of the packets that has been signed, so that the Analyzer can verify them.
In order to speed up the checking of a signature, in a signature packet there is a field that contains the first two bytes of the signed hash. This value has to be compared with the first two bytes of the computed hash and if it is different the signature is automatically invalid.
The hash is computed following the procedure in the RFC 4880. We call any hash calculated following this rules `right calculated hash`. During the develop of the *analyzer* has been found many `right calculated hash` that doesn't have a right hash header. This hashes is divided in two categories, the *key hashes* and the *user attribute hashes*. The first one refers to the hashes computed for a signature over a key or a subkey packet.
Examine in depth this problem, it turned out that, some hashes were computed inverting the two keys (i.e. the issuing and the signed ones), before the hash computation, thus the `right calculated hash`'s header doesn't correspond with the one in the signature packet. To avoid this problem, and verify the signature with the same hash used to made it, the hash, when the headers compared doesn't result equal, is computed inverting the keys. If not even in this way the comparing result correct, the stored hash is the `right calculated hash`.
The second category is referred to hashes computed for a signature over a user attribute packet. the hash for signing a *user attribute* is computed over the content of the packet. The content for a user attribute packet is a set of the content of the subpackets (until today only one is defined) each one preceded by its header, where is specified the length of the content. This length could be represented in three ways:

- one bytes if the length of the subpacket is equal or less than 191 bytes. In this case the only byte represent directly the length

- two bytes if the length of the subpacket is include in $[192, 8383]$. In this case the length of the content is the hex value of $(((\texttt{bytesize}(\texttt{content}) \gg 8) + 192) \ll 8) + (\texttt{bytesize}(\texttt{content})\&\texttt{0xFF}) - 192$
- five bytes if the other situations are not verified. In this cases the length is `0xFF` followed by four bytes representing the hex value of the length

Has been found out that not always the length representation follows the right rule, thus we have the same problem as above, the hash header in the signature packet doesn't correspond with the one of the computed hash. The mitigation strategy used is the same used above, in case of a non corresponding hash, this is computed three times, ones for each possible length representation, until a corresponding header is found. If it is not found, the `right calculated hash` is stored in the database.
With this implementation we lost a small number of a `right calculated hash` with a wrong hash header in the signature packet. This because there is a small number of signatures packets having an hash header computed in one way and the values (r, s) of the signature computed over the hash calculated in another way.
This "strange" choice is justified because in a small testing environment has been found more *wrong calculated hashes* (i.e. the one computed without following the rules written in the RFC 4880) with a right header that bring to a successfully verified signature, than *right calculated hashes* (i.e. the ones computed following the rules of the RFC 4880) with a wrong header that bring to a successfully verified signature.
The field `hashMismatch` in the table, is then set to 1 if the inserted hash doesn't correspond with the `right calculated hash`.

After the unpacking the daemon completes the info about the signatures, writing in the database the fingerprint of the key that has performed the signature, and sets the expired and valid flags. The first one is set if the signature is expired and the second one if the first one is set or if there is a revocation on that signature.

2.1.4 Analyzer

The analyzer daemon performs verifications about the security of keys and signatures. Each vulnerability found is inserted respectively in the `KeyStatus` and `SignatureStatus` tables. For the key analysis the daemon takes directly the values of the parameters of the public key from the `Pubkey` table and perform various checks based on which is the used algorithm.

Table 2.4: List of all the possible key vulnerabilities

Vulnerability Name	Vulnerability Code	Prevent the Signature Check	Algorithm Checked
OutdatedKeySize	1	NO	RSA/Elgamal/DSA
PrimeModulus	2	NO	RSA
CommonFactor	3	NO	RSA
LowExponent	4	NO	RSA
LowFactor	5	NO	RSA
Roca	6	NO	RSA
pNotPrime	7	YES	Elgamal/DSA
qNotPrime	8	YES	DSA
gLessEq1	9	YES	Elgamal/DSA
gSubgroup	10	NO	Elgamal/DSA
p&q	11	YES	DSA
CurveWrong	12	YES	ECDSA/EdDSA
PointNotOnCurve	13	YES	ECDSA/EdDSA

The key vulnerabilities are placed in Table 2.4

RSA

For RSA cryptosystem the vulnerabilities are:

`OutdatedKeySize:` The length of n should be bigger than 2048 bits, otherwise a bruteforce attack can be performed.

`PrimeModulus:` The modulus (n) shouldn't be prime. If it is prime is trivial to compute the $\varphi(n)$ parameter, and then recover the secret key d.

`CommonFactor:` The modulus should not have common factor with another key. Given two modulus $n = p \cdot q$ and $n' = p' \cdot q'$ with $p = p'$, is possible

to recover the common factor p, computing the greatest common divisor, through the Extended Euclidian Algorithm [16]. Once found p, computing q and q' is a trivial division. This vulnerability is not checked if the `LowFactor` one is found.

`LowExponent:` Given the same message m encrypted 3 times with a public exponent $e = 3$, and 3 public modulus (n_1, n_2, n_3), we can compose a system

$$\begin{cases} c_1 = m^3 mod n_1 \\ c_2 = m^3 mod n_2 \\ c_3 = m^3 mod n_3 \end{cases}$$

and solve it using the Chinese Reminder Theorem [16], find $\mathbf{X} \equiv m^3 mod(n_1 \cdot n_2 \cdot n_3)$. Now we can trivially compute m as $m = \sqrt[3]{\mathbf{X}}$, because $m^3 < n_1 \cdot n_2 \cdot n_3$, and thus the modular has no effect. This method can be extended on any possible e, having a sufficiently large number of different public modulus. The key is considered vulnerable if $e > 17$.

`LowFactor:` The modulus should not have lower factor. Given a modulus $n = p \cdot q$ with p a sufficiently small prime, is trival compute q. The first 3000 primes have been checked checked as factor of n.

`Roca:` The key should not be vulnerable to ROCA [22]. A vulnerability found in a software library used by Infineon Technologies AG, to generate the RSA modulus.

Elgamal

For Elgamal cryptosystem the vulnerabilities are:

`OutdatedKeySize:` The length of p should be at least 2048 bits.

`pNotPrime` following the rules of the Elgamal cryptosystem, p should be prime.

`gLessEq1:` g should be greater than 1, otherwise the group $\mathbf{G} = \langle g \rangle$ (i.e. the one generated by g) is the identity group. In this case, given two signatures (r_1, s_1) and (r_2, s_2), $r_1 = r_2$ because $g^k = 1 \forall k$, is possible to find the secret key x.

`gSubgroup:` g should be in the right subgroup (the condition checked is $g^p = 1 \bmod p$).

DSA

For DSA cryptosystem are verified the same vulnerabilities of Elgamal, plus:

`OutdatedKeySize:` The length of p and q are checked following the NIST directive [14]. The tuples $\langle p, q \rangle$ checked are:

- $\langle 1024, 160 \rangle$
- $\langle 2048, 224 \rangle$
- $\langle 2048, 256 \rangle$
- $\langle 3072, 256 \rangle$

`qNotPrime:` in addition to the one of p, also the primality of q is verified, for the same problem.

`gSubgroup` To verify the correctness of the subgroup we check that $g^q = 1 \bmod p$ and that g doesn't belong to the subgroup of dimension different than q (i.e. that $g^n! = 1 \bmod p$ with $n! = q$ the order of the oder subgroup of $\mathbb{Z}_p$).

`p&q` $p - 1$ should be a multiple of q.

Curve algorithms

For ECDSA, EdDSA the possible vulnerabilities are:

`CurveWrong:` The curve used is not compatible with the algorithm (e.g. Curve25519 used with a ECDSA algorithm)

`PointNotOnCurve:` The point should be on the curve. This vulnerability happens also if the point A is encoded and the decoding fails (i.e. if the decoding of a point fails, the point is not on the curve).

Signature

For each signature, the daemon takes the parameters (r, s), the hash computed by the *unpacker*, and the values of the issuing public key, to verify its vulnerabilities.

Table 2.5: List of all the possible signature vulnerabilities

Vulnerability Name	Vulnerability Code
`MD5Used`	21
`WrongAlgorithm`	22
`RepeatedR`	23
`WrongCheck`	24
`UnusablePublicKey`	25
`NotExportable`	26

The list of signature vulnerabilities can be found in 2.5.

`MD5Used:` A signature should not use the MD5 algorithm to compute the hash to be signed. The algorithm doesn't resist to the collision problem.

`WrongAlgorithm:` The algorithm that should be used to perform the signature are: RSA (with the `EMSA-PKCS1-v1_5`), DSA, ECDSA and EdDSA. Elgamal should not be used, as we have seen in 1.2.2. ECDH is a key agreement and thus is not used to perform a signature.

`RepeatedR:` for Elgamal, DSA and ECDSA, if the same user generates two signatures, having the same r value, means that the user has used the same value k for the computation of each signature. Thus we can write this system:

$$\begin{cases} s_1 = k^{-1}(H(m_1) + x \cdot r) \bmod q \\ s_2 = k^{-1}(H(m_2) + x \cdot r) \bmod q \end{cases}$$

where s_1, s_2, r are the parameter of the two signatures, $H(m_i)$ is the hash of the message m_i, and x is the secret key. Is trivial solve the system, and extract x.

`WrongCheck:` This vulnerability identifies the invalidity of a signature. The check is done using the formulas seen in 1.2. If the check doesn't came true the signature is invalid.

`UnusablePublicKey:` the signature cannot be checked due to a vulnerability found on the issuing key. The list of problematic vulnerabilities can be found in Table 2.4

`NotExportable:` A signature having the `NotExportable` flag set, should remain in the local keyring and thus should not appear in a keyserver.

Any possible vulnerability has a code and a description which are both inserted in the database. When a step in the analyzer generates an error, this is stored with a code that is 100 plus the code of the vulnerability whose verification has generated the error (for example if an error has generated during the check of the `OutdatedKeySize` vulnerability, which has code = 1, the error has code = 101). The error code is stored in the same table of the other vulnerabilities, to avoid the creation of another table. All the errors can be recognized, having a `vulnerabilityCode` greater than 100.

2.1.5 Reconciliation daemon

The reconciliation daemon is the original one, seen in 1.3, slightly modified to adapt it to the used database.

The tree is still stored in a Berkeley DB, but all the certificates are inserted and taken to and from the MySQL database. As we have seen above, inside the database we have:

- The hash of the certificate in the `gpg_keyserver` table.
- The flag `is_synchronized`, in the same table, that indicates if a key hash has been added to the tree or not.

- The table `removed_hash` instead contains the hash removed from the `gpg_keyserver` table (i.e. the ones that belong to the certificates that have been merged, before the joining).

In the reconciliation daemon there is a function that deals with the insertion of the new hash in the tree. It recovers the hash which has the `is_synchronized` flag equal to 0 (i.e. those just added), adds them to the tree (following the right pattern) and sets the flag to 1. The same function handles also the removed hash. When a key is updated, the hash changes and the old one is inserted in the `removed_hash` table. This function recovers all the removed hash and deletes them from the tree. The hash is removed also from the MySQL table as soon as it is deleted from the tree.

For initializing a new keyserver, the reconciliation daemon contains also a function that takes all the hashes from the database and inserts them in a new tree. The flag `is_synchronized` is not verified and thus not updated.

2.1.6 Dump Import

The dump import handles the import of a key dump. It is useful when we want to initialize a new database. The software is a union between the *PKS server* and the *Unpacker daemon*. It reads the certificates, following the rules seen in 2.1.2 to handle the error, and unpacks them, following the procedure seen in 2.1.3. The certificates are read them from a folder. Obviously the flag `is_unpacked` is set consequently, as we have seen in 2.1.3. The `is_synchronized` flag is set to 1, so to import the hash in a tree, has to be used the specific function in the reconciliation daemon.

2.1.7 Libraries

The libraries used in these implementation are:

- CppCMS – High Performance C++ Web Framework [3]: a framework designed for developing both Web Sites and Web Services. It is used in the PKS daemon.

- A C++ Implementation of RFC 4880 [5]: it is used in all the software but the Reconciliation daemon, to parse and create the keys. It has been modified to adapt it to the needs of this study.

- FastGCD [10]: efficient algorithm used to speed up the computation of the greatest common divisor between the RSA modulus in the *analyzer daemon*.
- NTL – A Library for doing Number Theory [25]: used to implement the computation of the formulas of the signature checking in the *analyzer*.
- GMP – The GNU Multiple Precision Arithmetic Library [23]: not used directly. It is used in all the previous library for support in mathematical operation

Chapter 3

Experimental Results

In this chapter will be shown all the problems and the results found during of the analysis. They are divided into key dimension categories, key revoked and created, key expired and created, signatures expired, and vulnerabilities found. At the end the most used domains in the emails and the are shown.

3.1 Problems found

During this study have came out a lot of problems. First of all there are a lot of packets which content has been slightly modified (not on purpose), creating broken packet. Given that in OpenPGP two packets are considered equal, if and only if their content is the same, in case of broken packets in a certificate, these are seen different (w.r.t. the original one) and thus maintained in the packets list during a merge; for this reason there are a lot of certificates that contain duplicate packets.

Moreover another critical issue is the wrong handling of the packets list by some keyrings, that have incorrectly changed their position. In fact could be found certificates with a *certification signature* (i.e. the one that should be situated after a user ID), after a subkey, or worse a *message signature* after a primary-key. In this case (i.e. if the moved packet is a *signature*), the check on the signature does not result true, because the checked hash is computed over the wrong packets.

The signature packet has also a two octet string (called header), representing the first two octets of the signed hash, used to boost the checking of the signature (if the two octets don't coincide, the signature should be directly discarded). This property should be valid for all the signatures, but has been found out that is not; in fact some signatures result correct also if the header doesn't correspond.
In this study have been also found some problem related to the computation of the hash; as is shown in Section 2.1.3 the hashes computed without following the RFC 4880 have been tracked in the database.

Another problem related to the *user attribute* packet is again connected to the wrong implementation of the RFC 4880. The *Image Attribute Subpacket* (i.e. the one that contains the image) has, before the image, an *image header* formed with:

- two bytes containing the length of the image, encoded as a little-endian number.
- one byte for the image header version (only version 1 is defined, which represents a 16 bytes image header)
- one byte representing the format of the image (only the value 1 si defined, that represent a JPEG image)
- 12 reserved bytes that must be set to 0

These last 12 bytes not always are set to 0. To avoid possible lost of information (if a single bit changes, the signature doesn't result correct) the 12 bytes have been left as they were found in the certificates. Being marked as *reserved* and equal to 0, these bytes must not be used due to possible future update of the RFC 4880.

In addition to these, there are also a lot of security issues.
Many keys have been generated with bad values (e.g. with an RSA modulus prime, or with a p value for DSA not prime) and so they could be compromised; some other, still valid and used, have values with a length that cannot guarantee a good security margin.
During the years have been found vulnerabilities over the implementation of offline keyrings and software used to handle the certificates. Also these ones can be used to to break a public key.

Another security issue is related to the user ID: they could have an email address that is no more existing, or worse that has a free domain, and thus it can be bought, to recreate the same email address, impersonating another user. The list of all the security issues checked can be found in Section 2.1.4. All this problems are summarised in the following charts.

3.2 Charts

During the writing of this study have been found many keys and signatures having a strange creation year, such as 1970 or 2106. Unless otherwise specified, in the next charts are shown only the keys and signatures created/expired between 1995 and 2018. Moreover the data about 2018 are incomplete and regard only the first three months of the year.

3.2.1 Dimension of the key created over the years

In this section is shown how the dimension of the parameters of the public keys have changed over the years of their creation. The total number of keys is 9810963.

RSA

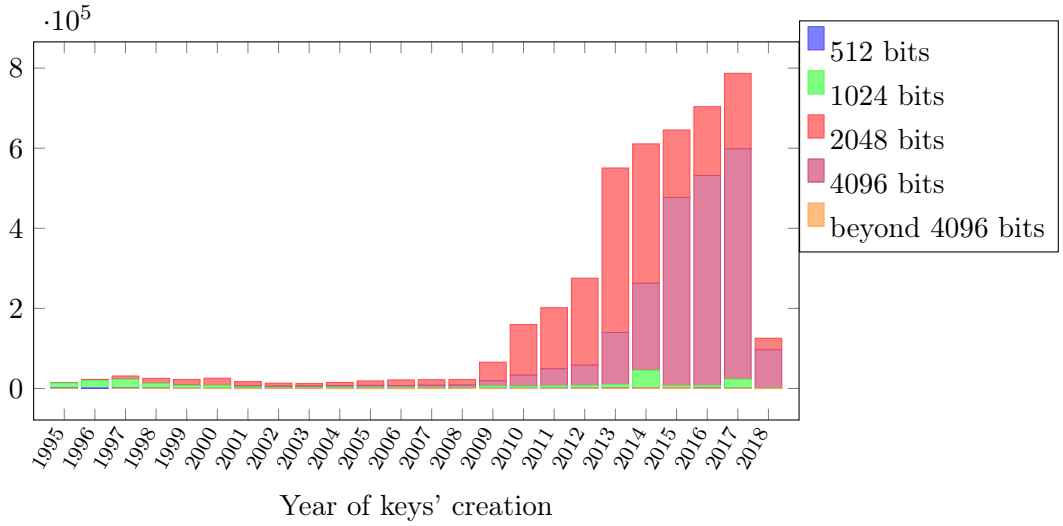


Figure 3.1: Bit length of RSA modulo

This chart represents the length of the public modulus n of the RSA public keys. Below the 2048 bits size the modulus is not more secure. Until 2020 the modulus can be considered secure from a length of 2048 bits onwards. It is estimated that a modulus with a length greater than 3072 bits can be considered secure until 2030.
The two most utilized dimensions are 2048 and 4096 bits. The RSA keys are the $45,11\%$ of the total keys.

Elgamal

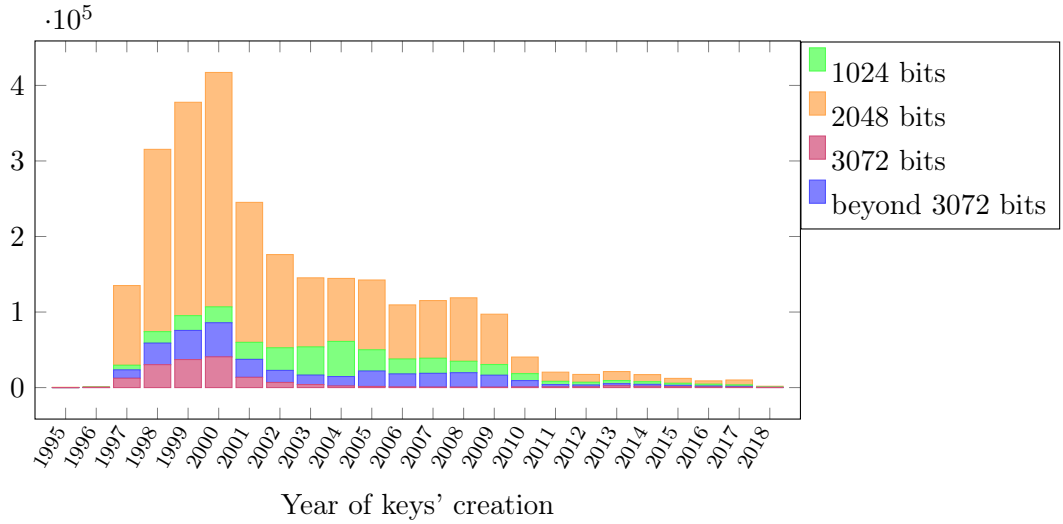


Figure 3.2: Bit length of Elgamal p value

This chart represents the length of the parameters p of the Elgamal public keys. A 1024 bits length for p is no more secure and should not be used. Until 2020/2030 a length of 2048 bits can still be considered secure, but after 2030 should be used p with a length of at least 3072.
The key lengths employed are quite good, but the number of Elgamal keys created is decreasing, probably due to the vulnerability found in GPG (see Section 1.2.2). The Elgamal keys are the $27,44\%$ of the total keys.

DSA

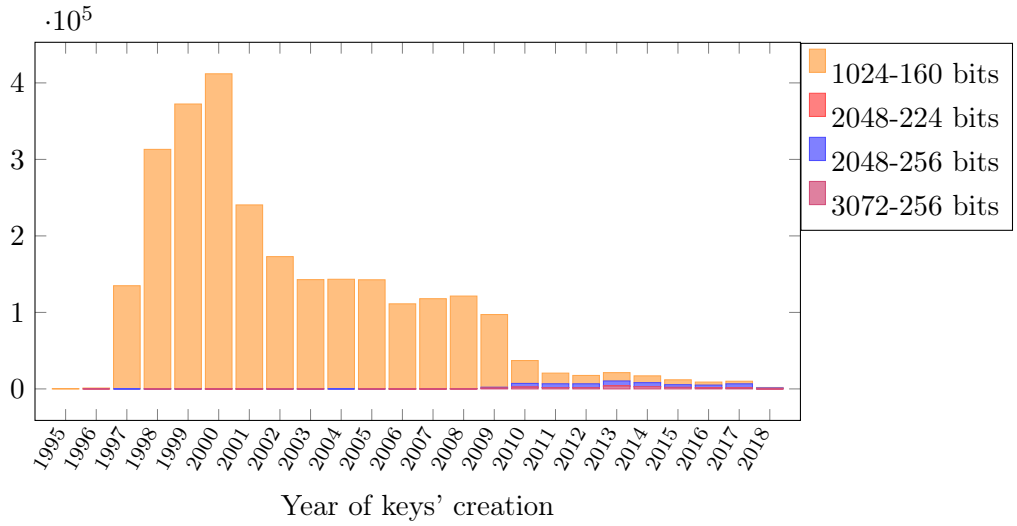


Figure 3.3: Bit length of DSA p and q values

This graph represents the length of the parameters p and q of the DSA keys. The length of p and q is fixed, and is shown in the legend. The security level is the same as Elgamal, with a length of 1024 bits no more secure and a recommended size of at least 2048 bits. This graph is similar to the Elgamal one, because, when you want to use a discrete logarithm algorithm, GPG creates a certificate with both DSA and Elgamal keys, the first one for signing and the other one for encrypting. Around 2013 GnuPG changes its default value for the algorithm of the new key (that was DSA for the signing and Elgamal for the encryption) to RSA for both operations. Thus the number of new DSA keys has decreased together with the number of the Elgamal ones, in favor of RSA keys. The number of the DSA keys are $27,36\%$ of the total.

Elliptic Curve

The elliptic curve algorithms in OpenPGP use some fixed curves, thus there is no need for plotting and check their size. The number of the keys that use these algorithms is 0.09% of the total keys.

3.2.2 Key revoked over the years

These charts show how much the revocation signature is used.

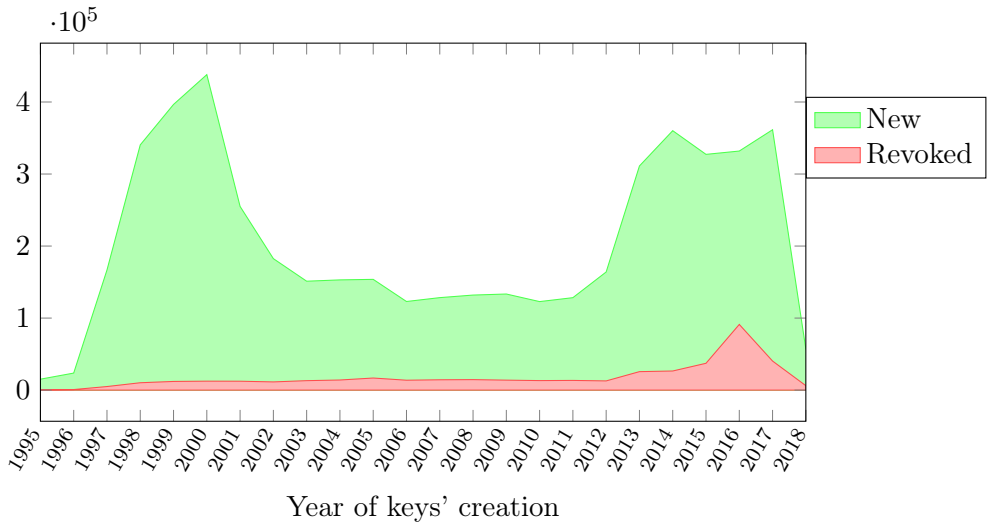


Figure 3.4: Comparison between revoked and created primary keys

This graph represents the number of the new primary key created each year, compared with the revocation signatures performed on primary keys in the same year. The total number of primary keys is 4957134, while the revocation is performed only over the 8.7% of them.

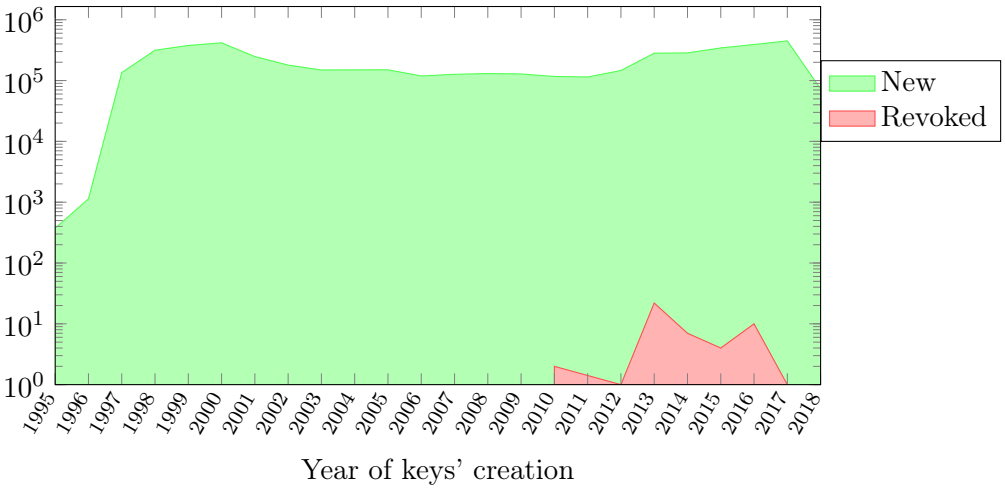


Figure 3.5: Comparison between revoked and created subkeys. The y-axes uses a logarithmic scale

This graph represents the number of the new subkey created each year, compared with the revocation signatures performed on subkeys in the same year. The y-axes uses a logarithmic scale, because the number of revoked subkeys is small and thus it would not be visible. The revoked subkeys are few, probably because a certificate is considered invalid when its primary key is revoked, and thus all the subkeys of that certificate are invalid too, without performing any revocation signature. In fact the number of subkeys is 4832256 and only 47 of them are revoked.

3.2.3 Key expired over the years

These charts show of how much the expiration property of a `key` packet is used.

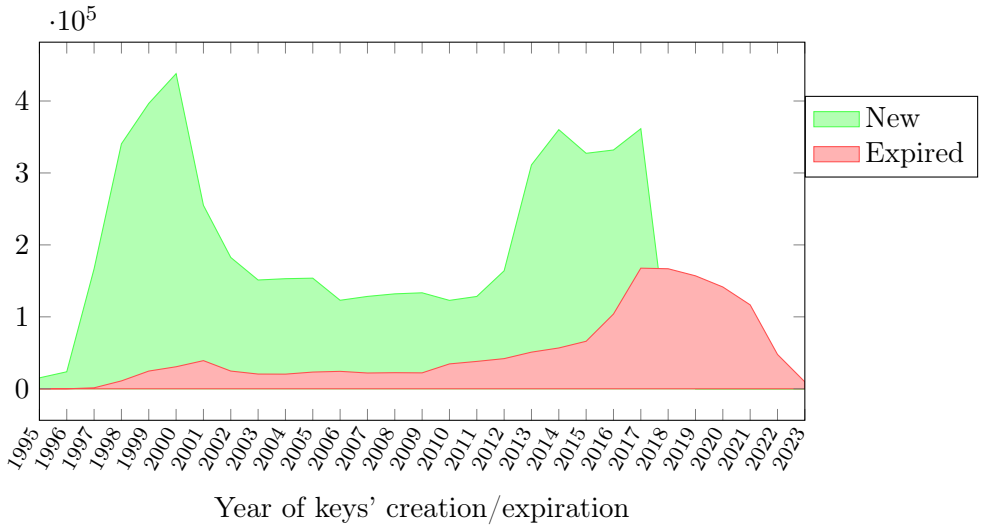


Figure 3.6: Comparison between expired and created primary keys

This graph represents the number of the new primary key created each year, compared with the ones expired in the same year. The expired keys line is slightly shifted to the right, proof that the expiration property of the keys is used, but, unfortunately, not too much, having only the 30% of expired primary keys over a total of 4957134.

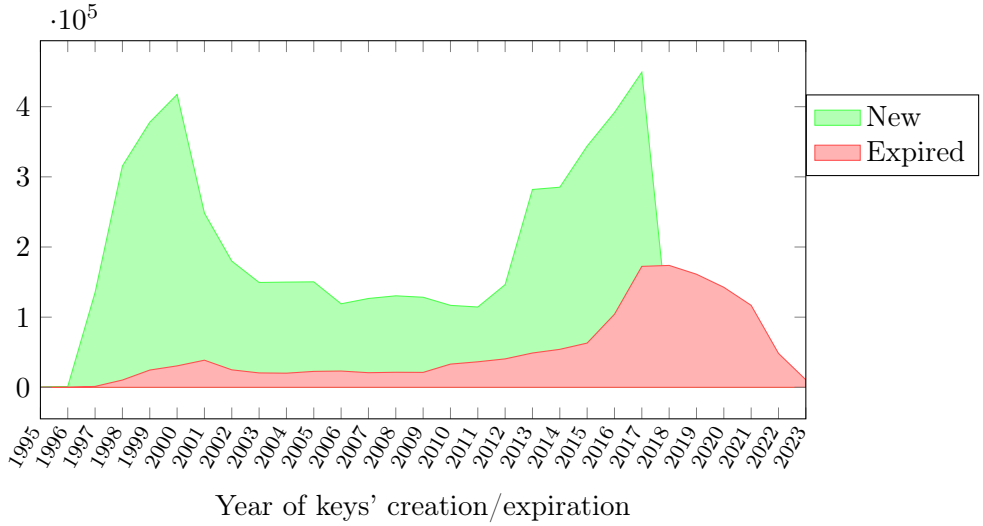


Figure 3.7: Comparison between expired and created subkeys

This graph represents the number of the new subkey created each year, compared with the ones expired in the same year. This chart is quite similar to the one above, because many software used to manage the keys, create the certificates directly with the primary key, a user ID and a subkey; thus the expiration time of the two keys coincides.

3.2.4 Signatures expired over the years

This graph points out how much the expiration property of a signature packet is used among the users, showing how many signatures expire and born over the years. The expiration property for a signature is not used as much, only the $11,3\%$ of the signatures has an expiration time set.

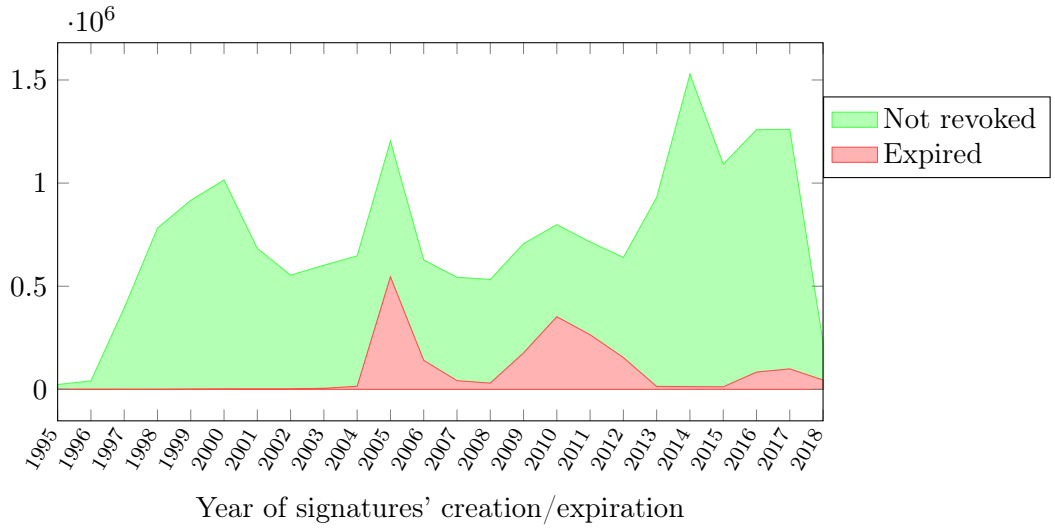


Figure 3.8: Comparison between expired and non-revoked signatures

3.2.5 Hash mismatches over the years

This graph shows the number of hash that are not computed following the RFC 4880.

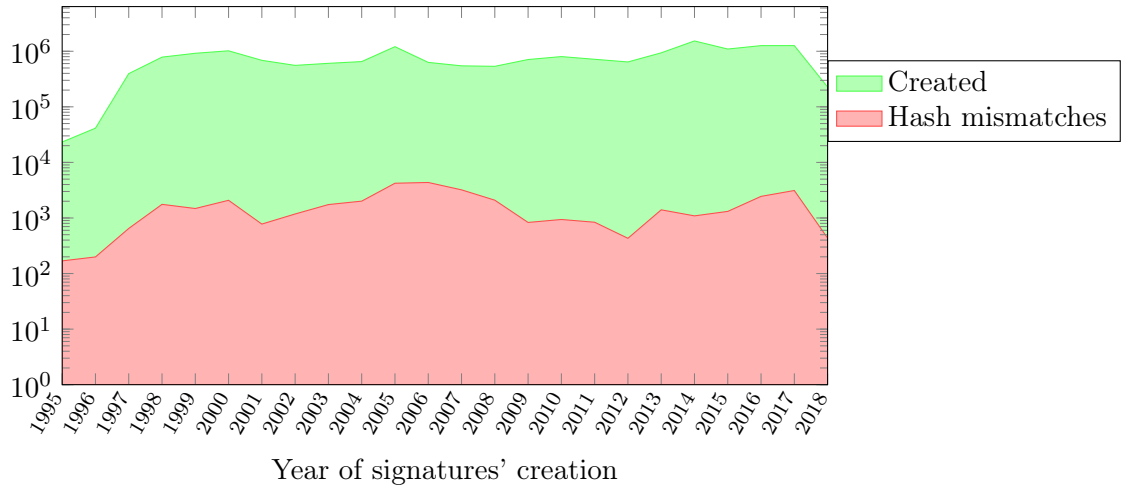


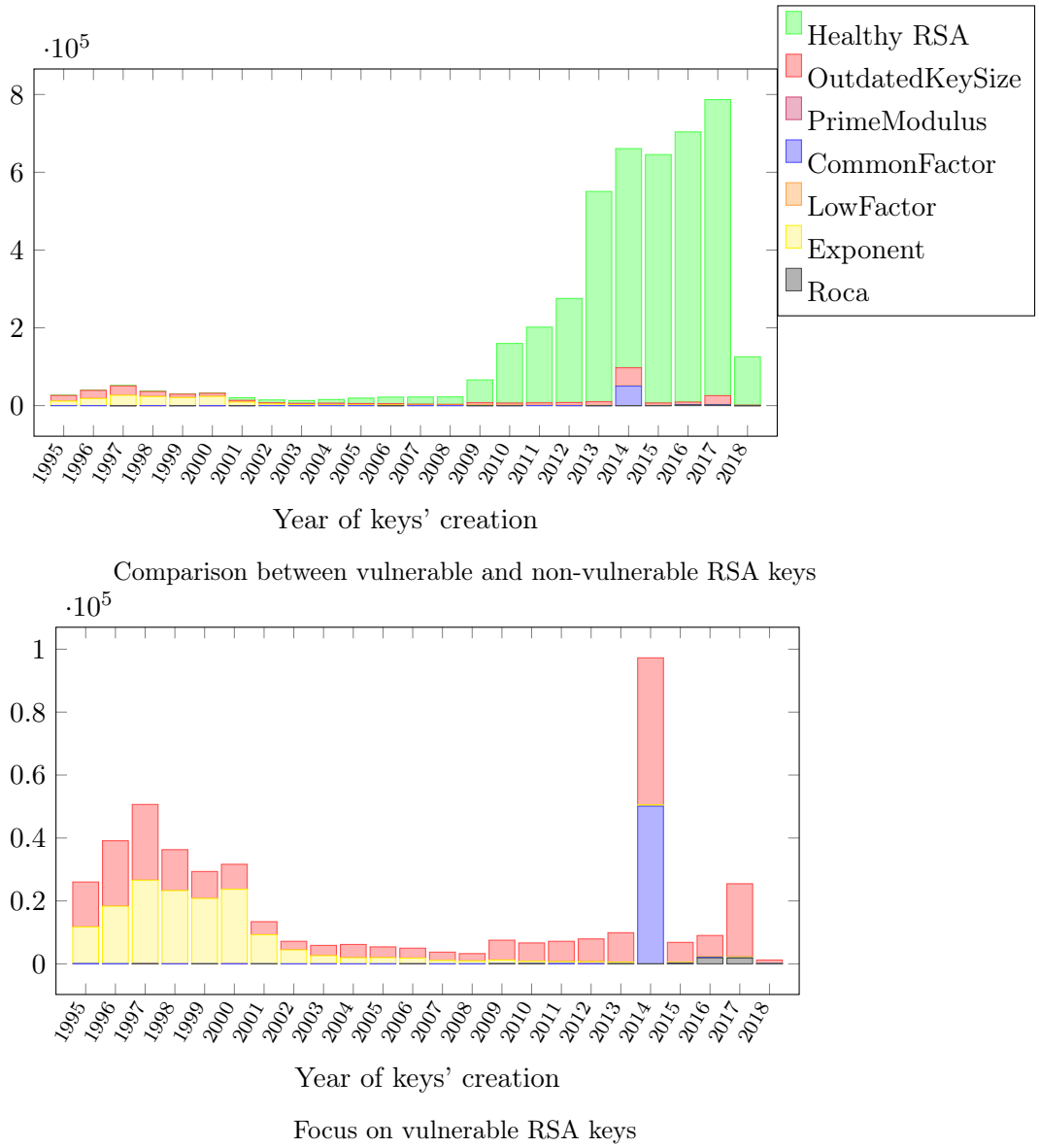
Figure 3.9: Comparison between right and wrong computed hashes. The y-axes uses a logarithmic scale

There are not many signatures with the hash wrongly computed, thus the y-axes uses a logarithmic scale. Luckily they correspond only to the 0.2% of the total.

3.2.6 Vulnerable keys

The following graphs show how many keys are vulnerable over their creation year. The vulnerabilities tested are listed in Table 2.4. A key can result vulnerable to more than just one vulnerability.

RSA



Comparison between vulnerable and non-vulnerable RSA keys

Focus on vulnerable RSA keys

Figure 3.10: Vulnerabilities found on RSA keys

The first graph represents the number of vulnerability of RSA, compared with the keys without any vulnerability (among those tested). The second one is a focus over the vulnerabilities only. The number of healthy RSA keys grow up with the decreasing of the `OutdatedKeySize` and the `Exponent` vulnerabilities, that diminish over the years. The other vulnerabilities are more or less stable a part for the `CommonFactor`, that has a peak in 2014.

Elgamal

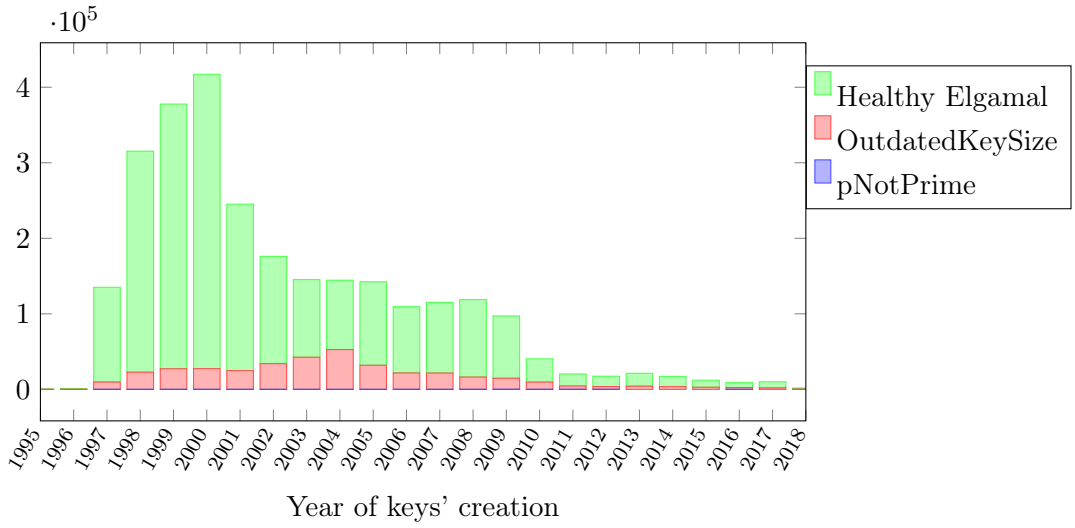


Figure 3.11: Comparison between vulnerable and non-vulnerable Elgamal keys

This graph represents the number of vulnerabilities of Elgamal, compared with the healthy Elgamal keys. Also with this algorithm the main issue is the dimension of p, considered too small. No Elgamal keys were found to be vulnerable to `gLessEq1` and `gSubgroup`, while a few number is vulnerable to `pNotPrime`.

DSA

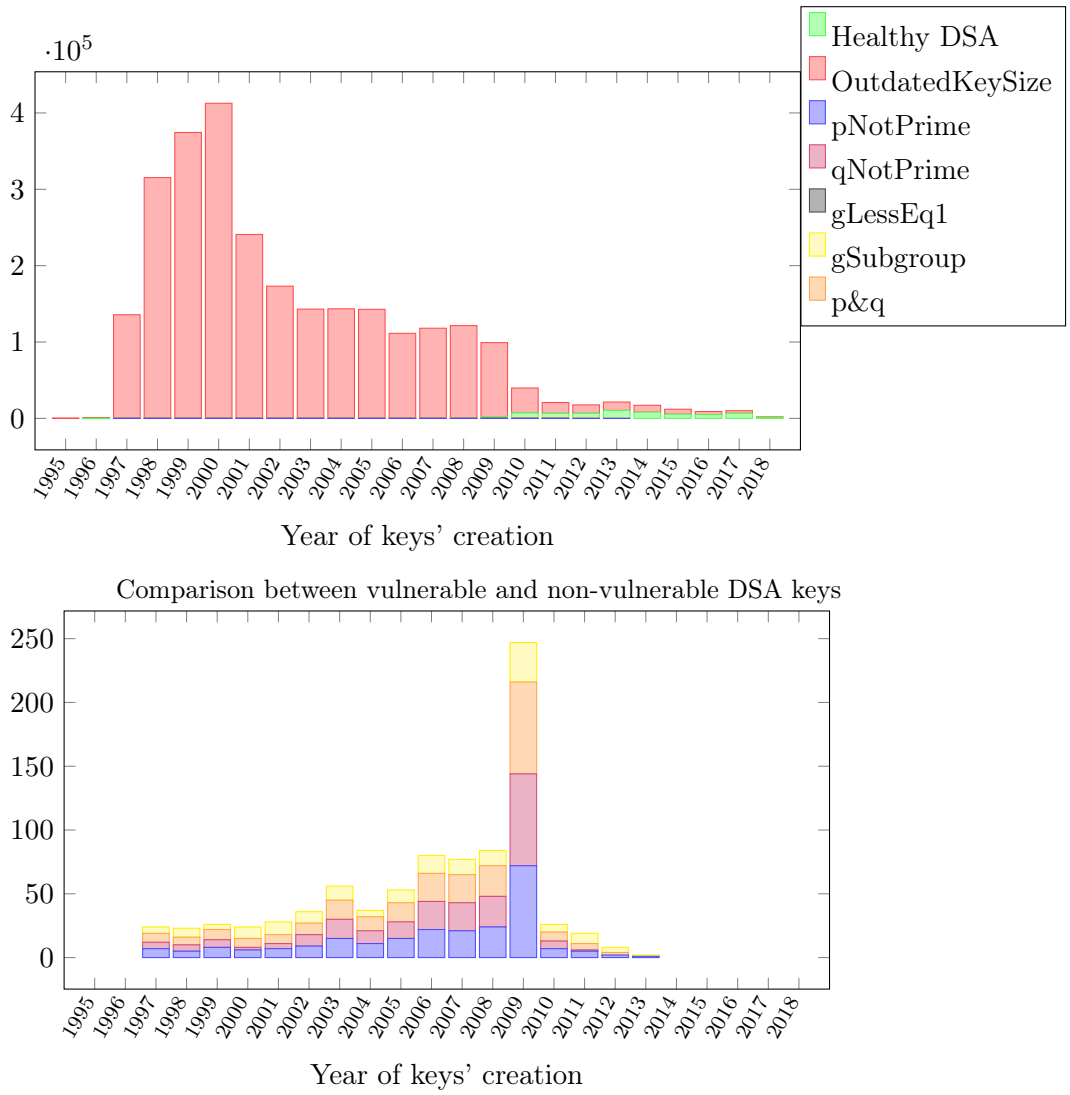


Figure 3.12: Vulnerabilities found on DSA keys

The first graph represents the number of vulnerabilities of DSA, compared with the number of healthy DSA. Unfortunately most of the DSA keys are considered vulnerable due to the too small dimension of the length of p and q (i.e. 1024 bits) that was the only one largely supported. The number of not-vulnerable keys grow with the decrease of the *low-size* keys. The other vulnerabilities found are not so much, but in the second graph

there is a comparison between the vulnerabilities found over DSA without the `OutdatedKeySize` one.

Curve Algorithms

For ECDSA and EdDSA no vulnerabilities (among the ones checked) have been found. Unfortunately these algorithms are used only in the 0.06% of the keys. They should be used more, especially EdDSA, because they are safer and faster than the *non-curve* algorithms.

Result summary

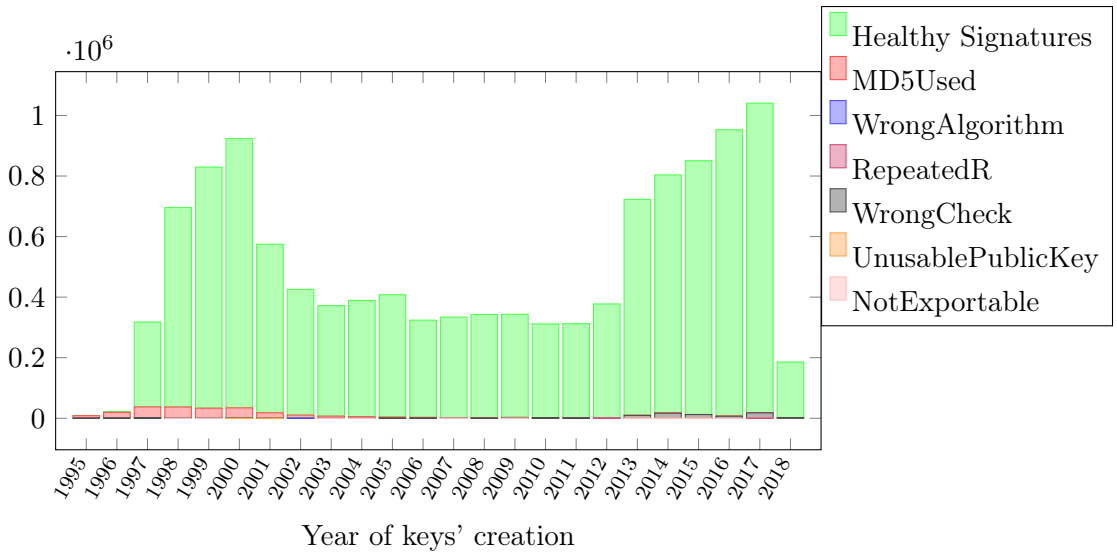
	RSA	**Elgamal**	**DSA**	**ECDSA EdDSA**
Total keys	4425447	2692453	2684052	5555
Healthy keys	4089804	2311869	60355	5518
OutdatedKeySize	248024 (118099)	380420 (380302)	2623684 (2623425)	NONE
PrimeModulus	1 (0)	NONE	NONE	NONE
CommonFactor	51304 (13547)	NONE	NONE	NONE
LowExponent	4 (1)	NONE	NONE	NONE
LowFactor	162542 (70071)	NONE	NONE	NONE
Roca	3964 (3962)	NONE	NONE	NONE
pNotPrime	NONE	261 (143)	238 (0)	NONE
qNotPrime	NONE	NONE	217 (0)	NONE
gLessEq1	NONE	0 (0)	0 (0)	NONE
gSubgroup	NONE	0 (0)	158 (2)	NONE
p&q	NONE	NONE	240	NONE
CurveWrong	NONE	NONE	NONE	NONE
PointNotOn-Curve	NONE	NONE	NONE	1 (1)
Error during analysis	9	25	114	36
Vulnerable keys without errors	335634	380563	2623695	1
Vulnerable keys including errors	335643	380584	2623697	37

Table 3.1: Results of the analysis over the keys

In Table 3.1 there are the results of the analysis performed on all the public keys (i.e. also the ones with a strange creation time). A key can be vulnerable to one or more vulnerabilities. In the table the number in the parenthesis indicates the keys having that vulnerability only. The `Vulnerable keys` lines instead, avoid duplicates, counting a key only one time. Moreover the number of vulnerable keys is stated two times, one including and one excluding the errors raised during the analysis.

3.2.7 Vulnerable signatures

The following graphs show how many signatures are vulnerable or not valid over their creation year. The vulnerabilities tested are listed in Table 2.5



Comparison between vulnerable and non-vulnerable signatures

Vulnerabilities found on signatures

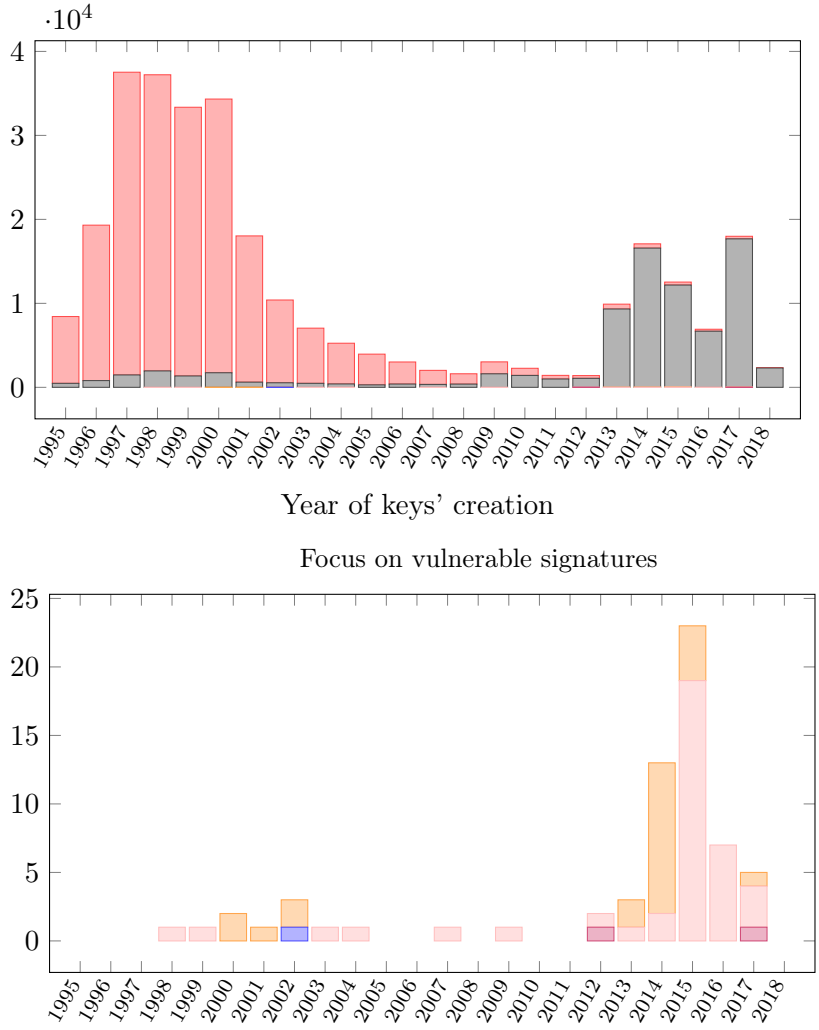


Figure 3.13: Vulnerabilities found on signatures (cont.)

The most widespread vulnerability is the one that regards the usage of the MD5 algorithm for the hashing part. This is decreasing but is still used in some signatures created in the last few years. The results of the check of the signatures are stable around one thousand of wrongly checked signatures per year, and slightly grow with the increasing of the number of the signatures.

Table 3.2: Results of the analysis over the signatures

Total signatures (analyzed)	11873679
Healthy signatures	11578519
MD5Used	219731 (212909)
WrongAlgorithm	1 (1)
RepeatedR	2 (2)
WrongCheck	81680 (74834)
UnusablePublicKey	24 (21)
NotExportable	40 (17)
Error during analysis	532
Vulnerable signatures without errors	294631
Vulnerable signatures including errors	295160

In Table 3.2 there are the results of the analysis performed on all the signatures (i.e. also the ones with a strange creation time). A signature can be vulnerable to one or more vulnerabilities, thus the idea is the same as the key vulnerabilities in Table 3.1: the vulnerability lines contains both the number of signatures having that vulnerability and, in parenthesis, the number ok signatures having only that vulnerability. The `Vulnerable signatures` lines instead, count a signature only one time. Moreover the number of all the vulnerable signatures is stated two times, one including and one excluding the errors raised during the analysis.
Contrary to the analysis of the keys, not all the signatures have been analyzed due to the lack of some issuing keys in the database.

3.2.8 Email domain used

This graph shows the usage of the email domains in the OpenPGP world, divided in five-years periods, based on the creation of the corresponding `user ID`. All the domains having a usage less than the 10% of the maximum occurrence of each graph, have been included in the `other` field

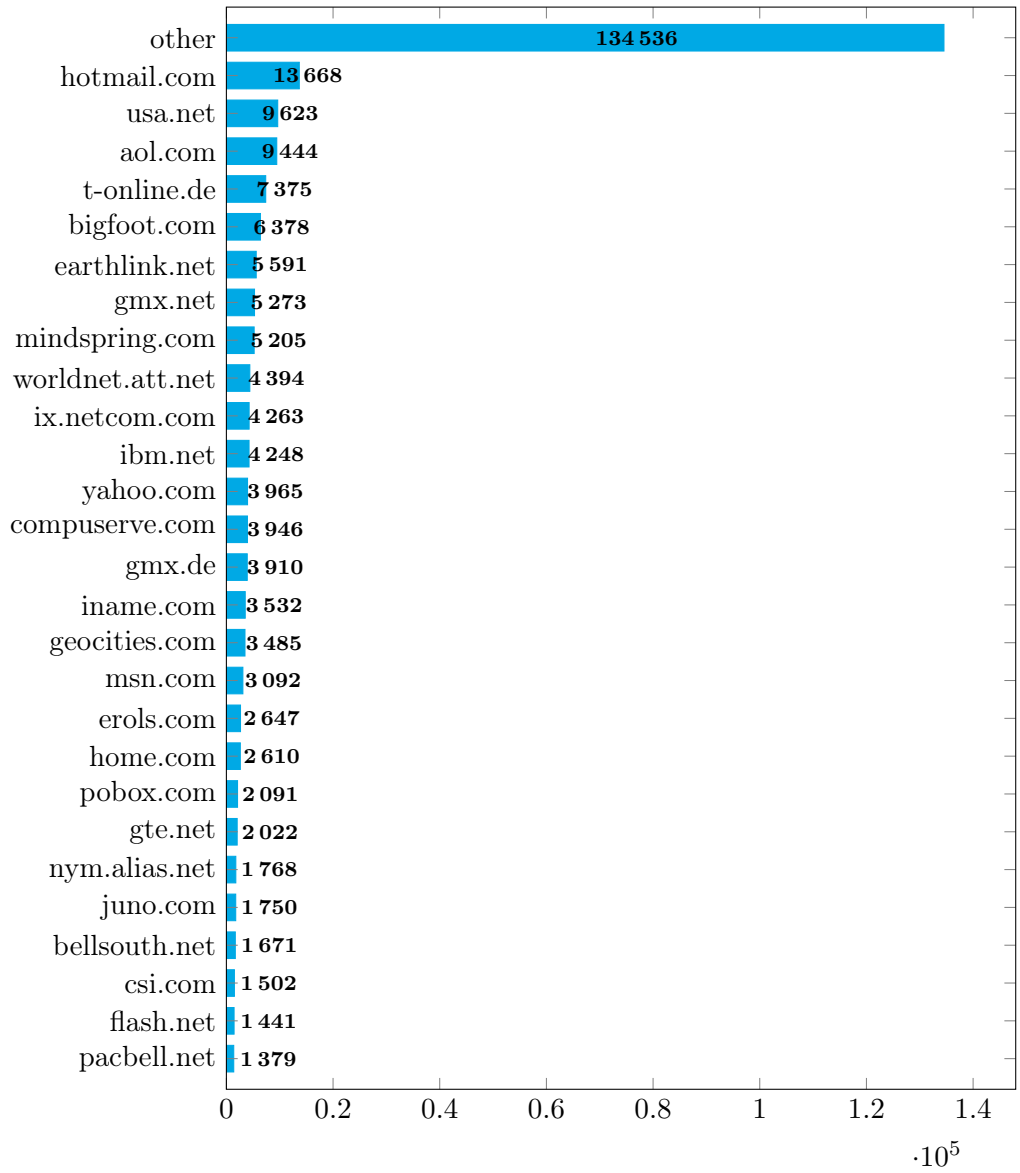
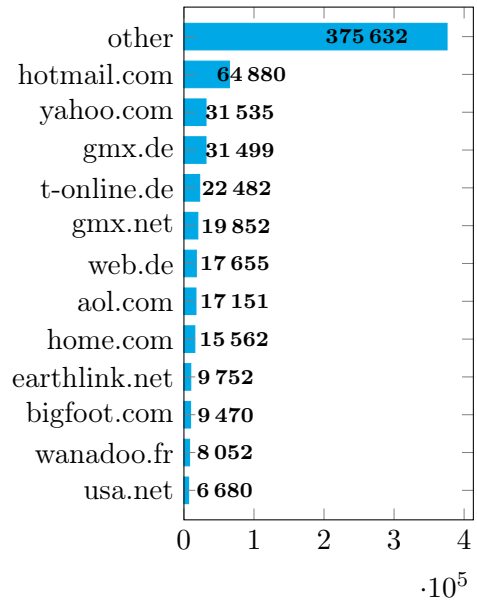
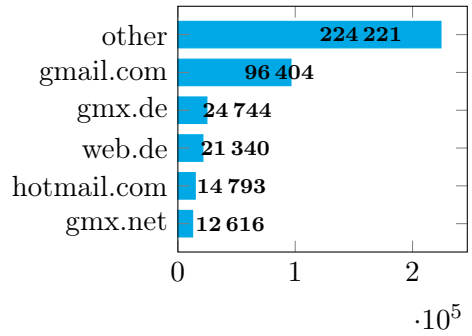


Figure 3.14: List of domain with their usage between 1994 and 1998

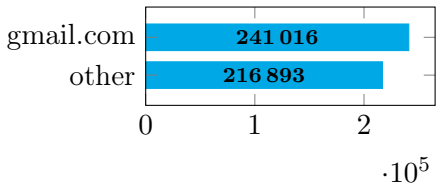
In the first five years the variety of domain usage was very high, just think that the number of domain with an occurrence less than 5 is $\approx$ 120000, 9 times the most utilized domain, *hotmail.com*; it is the leader of the domains, with 13668 occurrences, and has maintained the leadership also in the following five years.



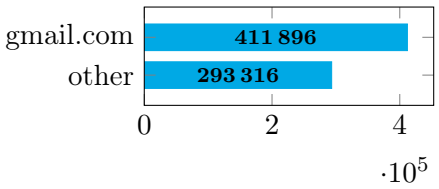
(a) List of domain with their usage between 1999 and 2003



(b) List of domain with their usage between 2004 and 2008



(c) List of domain with their usage between 2009 and 2013



(d) List of domain with their usage between 2014 and 2018

Figure 3.15

Before 2003, the number of domain with less than 5 occurrences was 5 times the most utilized domain (*hotmail.com* again), versus the 9 times of the previous years.

Between 2004 and 2008 *Gmail* was in beta but it has immediately reach the top of the ranking, thanks also to the less usage of *hotmail*. In this years also the variety of the used domain has decreased, the number of little used domain reached $\approx$ 220000, 2.3 times of *gmail.com*. That, from 2009, boomed, overtaking the throne, having a usage bigger than the sum of the occurrence of all the other domains.

Table 3.3: Composition of the field `other`

	Less used domain	**Total domain**	**Ratio**
1994 – 1998	118325	134536	87.95%
1999 – 2003	338191	375632	90.04%
2004 – 2008	207974	224221	92.75%
2009 – 2013	202793	216893	93.50%
2014 – 2018	273282	293316	93.17%

In Table 3.3 is shown how the `other` bars are composed. They have, more or less, always the same ratio between the number of few utilized domains (i.e. the ones with less than 5 occurrences) and the total.

Conclusion

The users in the OpenPGP world should be careful about the importance of the security behind keys and signatures, that should have always an expiration date, and they must be revoked when compromised. Every time a key is fetched from a keyserver, it should be verified if it could be compromised or not; in the first case, obviously, the key must not be used, and the data encrypted and signed with that key should not be trusted.

There should be also a transition to the elliptic curve algorithms, that use smaller keys (w.r.t. the non-curve algorithms) ensuring the same security margin. In this way the encryption, decryption and signing operations are speeded up.

Future developments

Some further developments that can be done starting from this study are:

- The analysis of the emails
- The reordering of the packets
- To repair (where possible) the inconsistency with RFC 4880
- The extending of the support over OpenPGP version 5

Email Analysis Check that the emails and their domains exist and thus that cannot be created a new email like to the one non existing, for impersonating other users.

Packets Reordering Examine the packets of the keys, to find out which are not in the correct position and, eventually move them. This operation should be done modifying the merging function in both the synchronization and the PKS daemon, to avoid that during a merge, the wrong positioned packets are restored.

Meaningfulness check Try to restore the meaningfulness in the certificate whose packets doesn't comply with the RFC 4880. This operation is thorny because first of all the broken packets should be found (not always is clear which packet is broken and which not), then they should be removed without breaking the synchronization (i.e. we have to filter the certificates received from the other keyserver during the synchronization and serve the original one).

Version 5 In the latest draft of the RFC 4880 [17], there is an idea about a new version of OpenPGP. When (and if) this new version will be officially released, it should be included in this server in order to support the new standard.

The OpenPGP world is in continuous evolving, and the progress should be followed, thus this study have to be kept active and updated.

Bibliography

[1] Derek Atkins, William Stallings, and Philip Zimmermann. Pgp message exchange formats. RFC 1991, RFC Editor, August 1996. http://www.rfc-editor.org/rfc/rfc1991.txt.

[2] Alessandro Barenghi, Alessandro Di Federico, Gerardo Pelosi, and Stefano Sanfilippo. Challenging the trustworthiness of pgp: Is the web-of-trust tear-proof? Technical report, Department of Electronics, Information and Bioengineering âĂŞ DEIB, Politecnico di Milano, Milano, Italy, 2015.

[3] Artyom Beilis. Cppcms – high performance c++ web framework.

[4] Daniel R. L. Brown. Sec 1: Elliptic curve cryptography. Technical Report 2.0, Standards for Efficient Cryptography Group, May 2009. https://www.secg.org/sec1-v2.pdf.

[5] calccrypto. Openpgp c++. https://github.com/calccrypto/OpenPGP.

[6] J. Callas, L. Donnerhacke, H. Finney, D. Shaw, and R. Thayer. Openpgp message format. RFC 4880, RFC Editor, November 2007. http://www.rfc-editor.org/rfc/rfc4880.txt.

[7] Jon Callas, Lutz Donnerhacke, Hal Finney, and Rodney Thayer. Openpgp message format. RFC 2440, RFC Editor, November 1998. http://www.rfc-editor.org/rfc/rfc2440.txt.

[8] CheapSSLSecurity. Understanding the role of certificate authority in pki. https://cheapsslsecurity.com/blog/

`understanding-the-role-of-certificate-authorities-in-pki`. [Online; accessed 18-March-2018].

[9] diaryFolio. Certification authority and applying certificate to your domain. [Online; accessed 18-March-2018].

[10] Nadia Heninger and J. Alex Halderman. Fastgcd. `https://github.com/sagi/fastgcd`.

[11] A. Jivsov. Elliptic curve cryptography (ecc) in openpgp. RFC 6637, RFC Editor, June 2012. `http://www.rfc-editor.org/rfc/rfc6637.txt`.

[12] Andrey Jivsov. Compact representation of an elliptic curve point. Internet-Draft draft-jivsov-ecc-compact-05, IETF Secretariat, March 2014. `http://www.ietf.org/internet-drafts/draft-jivsov-ecc-compact-05.txt`.

[13] S. Josefsson and I. Liusvaara. Edwards-curve digital signature algorithm (eddsa). RFC 8032, RFC Editor, January 2017. `https://www.rfc-editor.org/rfc/rfc8032.txt`.

[14] Cameron F. Kerry, Acting Secretary, and Charles Romine Director. Fips pub 186-4 federal information processing standards publication digital signature standard (dss), 2013. `https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.186-4.pdf`.

[15] SKS Keyserver. Sks keyserver. `https://bitbucket.org/skskeyserver/sks-keyserver/overview`.

[16] Donald E. Knuth. *The Art of Computer Programming, Volume 2 (3rd Ed.): Seminumerical Algorithms*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1997.

[17] Werner Koch. Openpgp message format. Internet-Draft draft-ietf-openpgp-rfc4880bis-03, IETF Secretariat, December 2017. `http://www.ietf.org/internet-drafts/draft-ietf-openpgp-rfc4880bis-03.txt`.

[18] A. Langley, M. Hamburg, and S. Turner. Elliptic curves for security. RFC 7748, RFC Editor, January 2016. `https://www.rfc-editor.org/rfc/rfc7748.txt`.

[19] M. Lochter and J. Merkle. Elliptic curve cryptography (ecc) brainpool standard curves and curve generation. RFC 5639, RFC Editor, March 2010. `https://www.rfc-editor.org/rfc/rfc5639.txt`.

[20] Y. Minsky, A. Trachtenberg, and R. Zippel. Set reconciliation with nearly optimal communication complexity. *IEEE Transactions on Information Theory*, April 2004.

[21] K. Moriarty, B. Kaliski, J. Jonsson, and A. Rusch. Pkcs #1: Rsa cryptography specifications version 2.2. RFC 8017, RFC Editor, November 2016. `https://www.rfc-editor.org/rfc/rfc8017.txt`.

[22] Matus Nemec, Marek Sys, Petr Svenda, Dusan Klinec, and Vashek Matyas. The Return of Coppersmith's Attack: Practical Factorization of Widely Used RSA Moduli. In *24th ACM Conference on Computer and Communications Security (CCS'2017)*, pages 1631–1648. ACM, 2017.

[23] GNU Project. The gnu multiple precision arithmetic library. `https://gmplib.org/`. Arithmetic without limitations.

[24] D. Shaw. The camellia cipher in openpgp. RFC 5581, RFC Editor, June 2009.

[25] Victor Shoup. Ntl - a library for doing number theory. `http://www.shoup.net/ntl/`.