

POLITECNICO DI MILANO
Scuola di Ingegneria Industriale e dell'Informazione
Corso di Laurea Magistrale in Ingegneria Informatica
Dipartimento di Elettronica, Informazione e Bioingegneria



Enabling robust keyservers for OpenPGP:
key reconciliation reengineering for the Peaks keyserver

Relatore: Prof. Alessandro BARENGHI

Tesi di Laurea di:
Andrea GRAZIOSO Matr. 837839

Anno Accademico 2017-2018

*You don't become great by trying to be great.
You become great by wanting to do something,
and then doing it so hard that you become great in the process*
Randall Munroe, xkcd, Marie Curie, 2011

Ringraziamenti

Questa sezione serve per poter ringraziare tutti coloro che hanno reso possibile questo lavoro.

Ringrazio il mio relatore, il professor Alessandro Barenghi, e il mio ex-collega Mattia Pini, i quali hanno iniziato questo enorme lavoro, che ho avuto il piacere (e a volte il dispiacere) di portare avanti.

Ringrazio tutti i professori che ho incontrato, da cui ho imparato non solo la materia ma anche la capacità di apprendere e utilizzare i concetti in modo creativo e interconnesso.

Ringrazio la mia famiglia, mio padre in particolare per avermi spronato anni fa ad intraprendere questo percorso universitario, e per avermi sempre sostenuto.

Ringrazio i miei colleghi e i coinquilini, prima di ogni altro appellativo amici simpaticissimi e compagni di avventure.

Ringrazio tutti i ragazzi del POuL, con cui ho avuto un mare di discussioni, con cui ho riso tantissimo, ma da cui ho imparato veramente un'infinità di cose.

Ringrazio tutti i miei amici del GPC, anche se vi ho “abbandonato”, siete sempre nei miei pensieri.

Un ringraziamento di cuore ad Adelina, che da anni mi sopporta, e mi supporta, seppur a modo suo, continuamente. Grazie per le correzioni (infinite) e per tutti i suggerimenti, probabilmente senza di te questo lavoro non sarebbe nemmeno iniziato.

Grazie a tutti.

Abstract

OpenPGP is a wide known standard used to provide confidentiality and encryption in communications, authenticity in data transfer. Its uses are several, up to the point that today a special infrastructure, a keyserver, is needed to correctly manage 5 million of pgp certificates accumulated during years. Due to design principles, keys are not meant to be removed from keyserver: this lead to an abundance of weak, broken, faulty keys which create further burden, on the now over 20 years old, current state of the art, sks-keyserver, which was not designed to tackle the challenging issue of managing a huge number of keys.

This Project is a follow up on the thesis *Peaks: Adding Proactive Security to OpenPGP Keyserver*: Peaks needed a functional recon daemon to sync with the currently deployed keyserver and be independent from the old codebase.

Now such daemon is completed, and by design is compliant with the other implementations, it can communicate and exchange key material with other servers, but is designed to be extensible and capable of taking full advantage of modern hardware.

This work starts with an introduction on OpenPGP history, its uses and implementation in software. An analysis on the principles and implementation of the current state of the art will follow, focusing on the synchronization between keyserver. Follows a detailed specification on how the reconciliation daemon is implemented in Peaks, recalling the one from sks-keyserver. The last chapter will present some results about the analysis on the running instance of Peaks.

Sommario

OpenPGP é uno standard ampiamente diffuso e utilizzato per garantire sicurezza e confidenzialità nelle comunicazioni, e autenticità nel processo di scambio di dati. Essendo utilizzato in vari ambiti è necessaria un'infrastruttura apposita, cioè un keyserver, capace di accogliere e servire l'ingente quantità di chiavi pubbliche accumulate nel corso degli anni.

Purtroppo, nessuna chiave è mai stata rimossa dai keyserver, provocando uno stagnamento di chiavi ormai deboli, corrotte, dismesse che creano non pochi problemi di gestione del sistema, che, stando allo stato dell'arte, è stato concepito più di 20 anni fa, e non era stato pensato per affrontare certe sfide.

Questo progetto segue la Tesi *Peaks: Adding Proactive Security to OpenPGP Keyserver*: Peaks aveva ancora bisogno di un demone di riconciliazione funzionante, un componente che permettesse la sincronizzare del materiale crittografico.

Questo modulo è ora completo e retrocompatibile con lo stato dell'arte, ma allo stesso tempo estendibile, modulare, documentato, con la capacità di sfruttare al meglio l'hardware moderno di cui oggi disponiamo.

Uno degli ostacoli maggiori durante la sua realizzazione è stata l'assenza di documentazione fatta eccezione per il codice pre-esistente, per cui questo lavoro è un'analisi e reingegnerizzazione dello standard esistente.

Questo lavoro è organizzato in un capitolo introduttivo, riguardante la storia di OpenPGP e della crittografia in generale. Verrà spiegato come funziona lo stato dell'arte, su quali principi e modelli si basa la sincronizzazione e come essa è implementata in Peaks, seguendo quanto fatto nello stato dell'arte. Il capitolo conclusivo chiude la trattazione con i risultati ottenuti tramite analisi sulla nuova infrastruttura in funzione.

Contents

Introduction	1
1 History and State of the Art	7
1.0.1 OpenPGP	7
1.0.2 GPG	12
1.0.3 Keyserver	13
1.1 State of the Art	15
2 Reconciliation Daemon	23
2.1 Analysis of the Recon Protocol	23
2.1.1 High level view	23
2.1.2 Set reconciliation	27
2.1.3 Prefix tree	46
2.1.4 Components	57
3 Experimental Evaluation	59
3.1 Experimental Results	59
3.1.1 Ptree sanity	59
3.1.2 certificate sanity	61
3.1.3 Pubkey security	66
3.1.4 certificate vulnerability	71
3.1.5 Signatures security	76
3.1.6 usage state	79
3.1.7 growing trend	83
Conclusion	85
3.2 Conclusions	85












CONTENTS

A Data types used	87
B Parameters	89
Bibliografia	91

List of Figures

1.1	Example of ASCII-armored PGP Message	8
1.2	Composition of the Packet Tag byte, except the first bit, each value depend of the format: New on top, Old on the bottom	8
1.3	sks-keyserver internals: circles are datasets, full line boxes are processes, dashed line boxes are user inputs, arrows are data exchange between components	16
1.4	Example of partition tree: Internal nodes contains the CheckSum of the set union of Partitions, Leaf node represent Partitions	20
2.1	config validation	24
2.2	single node recon	25
2.3	elements recover	26
2.4	Example of membership file	27
2.5	struct ReconRequestPoly	43
2.6	struct Config	45
2.7	struct Pnode	47
2.8	ptree table	56
3.1	Node number per level	60
3.2	element per node plot in log and linear scale	60
3.3	first levels of the ptree	61
3.4	certificates without user attributes divided by total length	62
3.5	certificates without user attributes divided by total length (log scale)	63
3.6	certificates with user attributes divided by total length	63

LIST OF FIGURES

3.7	certificates with user attributes divided by total length (log scale)	64
3.8	user attributes photographic divided by length	65
3.9	user attributes of other type divided by length	66
3.10	distribution of key creation per year	67
3.11	Signatures partitioning  RSA;  Elgamal;  DSA;  Elliptic Curve;	67
3.12	bit length of RSA public key n parameter	68
3.13	bit length of Elgamal public key p parameter	69
3.14	bit length of DSA public key p and q parameters	70
3.15	EC public keys created during years	71
3.16	Healthy and Unhealthy pubkey number comparison	73
3.17	In depth analysis of the vulnerabilities found per year	73
3.18	Healthy and Unhealthy pubkey number comparison	74
3.19	In depth analysis of the vulnerabilities found per year	74
3.20	Healthy and Unhealthy pubkey number comparison	75
3.21	In depth analysis of the vulnerabilities found per year	75
3.22	Signatures partitioning  signatures  self signatures	76
3.23	Signatures/self-signatures creation per year	77
3.24	Signatures data per year	77
3.25	Signatures used per year	78
3.26	Domain Un-reachability chart  Non-existent Domain  DNS Label error  MX not found  Timeout  Misconfigured Records (SERVFAIL)	81
3.27	top domains per number of mail addresses registered	82
3.28	public keys added per year on top of the pre-existent	83
3.29	Unknown unsafe keys	84

List of Tables

1.1	Packet tags explanation, Old Format is also used for Newer version	9
1.2	Signature Subpackets explanation	10
1.3	User Attribute Subpackets explanation	11
2.1	Message header table	42
3.1	ptree statistics	60
3.2	user attribute popularity	65
3.3	signatures statistics	76
3.4	signature data	79
3.5	statistics from user identities	80
3.6	statistics from unavailable domains	81

LIST OF TABLES

List of Algorithms

2.1.1 client recon main loop	29
2.1.2 REQUEST POLY HANDLER	30
2.1.3 REQUEST FULL HANLDER	31
2.1.4 SOLVE LINEAR INTERPOLATION	33
2.1.5 POPULATE MATRIX	34
2.1.6 GAUSSIAN ELIMINATION	35
2.1.7 BACK SUBSTITUTE	36
2.1.8 server main loop	37
2.1.9 SEND REQUEST	38
2.1.10 FLUSH QUEUE	38
2.1.11 HANDLE REPLY	40
2.1.12 FETCH ELEMENTS	41
2.1.13 NEW CHILD	49
2.1.14 ADD ELEMENT ARRAY	50
2.1.15 INSERT	51
2.1.16 SPLIT	52
2.1.17 NEXT	53
2.1.18 GET CHILD	53
2.1.19 DELETE ELEMENT ARRAY	54
2.1.20 DELETE	55
2.1.21 JOIN	55
2.1.22 GET NODE	56

LIST OF ALGORITHMS

Introduction

With the increasing spreading of computer and communications systems in the 60s, there was an increasing demand for means to protect digital informations and to provide security. Actually, information security was not something new: over the century, complex mechanisms, such as substitution ciphers and polyalphabetic ciphers were developed to conceal messages. These kind of techniques developed until 19th century are now addressed as *classical cipher* to distinguish them from the *modern cipher* which took origin after World War II, with a publication from Claude E. Shannon [Shannon1]. He was inspired during the war to address the problems of cryptography because secrecy systems provide an interesting application of communication theory. In particular, he identified two main goals from cryptography, encryption and authentication which are respectively the practise of protecting a message from beign read by entities other than the receipients, and the act of confirming the identity of the sender.

Shannon attempted to provide a classification of secrecy systems in communications:

1. Concealment systems, to hidden a message using a jargon
2. Privacy systems, in which some kind of hardware needs to be used to recover the original message
3. True secrecy systems in which the message is concealed by proper cryptosystems.

The modern cryptography is based on the last point. In particular, secrecy is achieved using algorithms which employ a key to encrypt and decrypt information. Cryptography is the study of techniques for ensuring communication

which can be both encrypted and authenticated. Generally speaking, cryptography allow to build and analyze protocols to prevent third parties or the public from reading or manipulating messages.

Until the 70s all encryption models in circulation were symmetric key algorithms, in which the same key is used to provide encryption and decryption of the message, so it needed to be available to both sender and receiver. So before establishing a secure communication, the parties would have needed a secure way to exchange those keys, creating a chicken and egg problem. In 1976, a publication from Whitfield Diffie and Martin Hellman changed forever the way cryptosystems are conceived. The article introduced a radically new method of distributing keys, which also stimulated the development of asymmetric cipher or public-key cypher [DiffieH76]. This type of algorithm uses two mathematically related keys, one for encryption and one for decryption, permitting the publication of one of the keys, namely the public one, due to being extremely difficult to determine the other one, the private, simply from the public. So, no secure channel is needed for key exchange, as long as the private key remains in the hands of the owner, the public can be known without compromising security.

By today standards to achieve secure communications we need to ensure the following properties:

- Confidentiality: information can be accessed only by authorized entities.
- Integrity: information can be modified only by authorized entities, and only in the way such entities are entitled to modify it.
- Data Authentication: two parties entering into a communication should identify each other
- Non-Repudiation: information coming from an authenticated entity cannot be denied anymore.

Public key cryptography ensure all these properties: suppose that two parties (let us say Alice and Bob) want to communicate over an insecure channel, first they exchange public keys. Alice encrypts a message using her private key, and then re-encrypt it with Bob's public key. The message is now received by Bob, which is able to decrypt it using his own private key,

achieving confidentiality, and then with Alice's public key, ensuring authentication. Actually since public-key cryptosystems, relying on complicated mathematical computations, are quite slow in performing operations, compared to symmetric key cryptosystems, in many actual implementations a combination of both is used, resulting in a **hybrid cryptosystem**. This system uses the public key cryptosystem to encapsulate not the data itself, but a symmetric session key, which is used to encrypt data. On the receiver end the symmetric session key is recovered with the recipient's private key and used to decrypt the message content.

In a similar way it is possible to achieve authentication and non-repudiation by having Alice generating a **hash signature** for the message and encapsulating it with her own private key, creating a signature which could be verified by Bob using Alice's public key.

The two systems can be combined together to achieve confidentiality, authentication, data-integrity and non-repudiation.

The real problem of this system is how to bind a key with the identity of the owner. Since everyone could generate a new key, the binding between a public key and its owner must be correct, otherwise the algorithm may function correctly but beign completely insecure. Associating a public key with its owner is achieved by creating a binding between an entity and it's key-pair represented by a **digital certificate**, which is managed, used and stored by a set of policies known as **public key infrastructure**. For our purposes we focus on **Pretty Good Privacy (PGP)** created by Phil Zimmermann in 1991[Zimmermann]. PGP encryption uses a hybrid cryptoscheme capable of supporting algorithms. In PGP each public key is bound to an email address, and from its first release, PGP products have included a certificate vetting scheme, a trust model known with the name "**Web of Trust**". According to this scheme a public key could be digitally signed by a third party to check the association between an username or email and the key. The implementation permits several levels of confidence to be included with the signature. PGP allows to cancel (**revoke**) the identity certificates. If a private key is compromised the owner could use a **revocation certificate** to mark the key.

Before starting a secure communication with someone it's necessary to exchange keys, a process which, done manually, easily become both time

consuming and compromisable. To overcome this limitations a **keyserver** is employed.

Keyservers act as central repositories of keys, where everyone could search and securely retrieve a public key, they are *trusted anchor* for our model, which is by definition an authoritative entity for which trust is assumed. Such is the importance of keyservers, so its crucial that the community actively vigilate on the development, behaviour and mantainance of its infrastructure.

To mitigate the risk of anyone surveying of controlling this important structure, keyservers have been organized in large pools of keyservers. Pools provide load balancing, high availability, and robustness to the keyserver network. Using a proven mathematical model, servers's belonging to a pool are capable of synchronizing key material with the lowest amount of complexity, both computationally and bandwidth-wise.

Unfortunately keyservers always suffer from its design principles:

- Once a public key is stored, it's difficult to remove it, because by design keys are not meant to be removed. If a key get compromized or lost became useless key, but it's still saved into the keyserver. This lead to a linear increasing consumption of storage space and inevitably slow down the entire system.
- There's no authentication to publish material on a keyserver, so anyone could store bogus keys.
- There's no check on the key validity, so any key corrupted, or manipulated on purpose, will be stored in keyserver's database.

This work aims to analyze the current status of keyserver infrastructure, by looking at its flaws, its current problems and by a design point of view.

In particular the current most widely adopted keyserver, **sks-keyserver**, is quite old in it's implementation and several bugs has been discovered: a proof of concept has been written to use the keyserver as file storage, as thus abused illegally.

Unfortunately, in recent times, many servers shut down citing attacks like the abovementioned causing network outage, drives filling up, increasing the hosting bill and basically causing a Denial of Service attack.

Furthermore, due to the new European regulation regarding data protection and privacy, affecting all individuals within the EU (**GDPR**), many

keyserver admin decided to shut down the service, in fear that complaints related to the non-removal design choice of key material could be filled against service owners, adding further stress on the keyserver infrastructure.

While suffering from serious bugs, attested also in the official mailing list[sks-keyserver-mail], in which admin complaining of hard to debug errors which affect various parts of the running keyserver, the development is currently stalled, original developer have decided to not contribute anymore[yaaron-quits]. The community seems to be unable to overtake the development of sks-keyserver to provide useful contributions, probably due to the fact that the codebase have as core language OCaml, which by today standard is not as known and used, as a piece of software of this caliber would need to be supported by an active community.

On top of that, the actual code reflects the computer literacy of the 90s, in which many of the today programming concept, such as multithreading, where not implemented in languages and frameworks. Because of that, the code is not hardware efficient, especially nowadays in which cpus are designed to efficiently handle tasks which spans on multiple threads or processes, up to the point that to join the pools of keyserver high hardware requirements are needed, for a relatively simple task.

This work aims to provide a modern approach to keyserver design, with a particular attention on security, easiness develop and sustainability of the codebase and deploy.

Since there's no documentation available on the internal mechanisms, and on the protocols which regulate the behaviour and synchronization between keyserver, this work is mainly an analysis, interpretation and reengineering of the current state of the art, which is by far the most widely known and adopted keyserver among the community and thus provide a reasonably stable starting point to work on.

We aim to fix the current inefficiencies, but also provide a open and documented implementation that the community could use to reinvigorate the keyserver scene and to incentivate the study and analysis of a critical piece of infrastructure.

Chapter 1

History and State of the Art

In this chapter will be discussed what is an OpenPGP certificate, how it is structured, how it could be used to achieve confidentiality and autenticity, the use of GPG, what is a keyserver, its purpose and which operations it performs, and the current state of the art Keyserver.

1.0.1 OpenPGP

When the ideator of the original PGP software became convinced that an open standard for PGP encryption was critical for its success and spread into the cryptographic community, the PGP Inc. proposed to the IETF a standard format for certificates and encrypted messages called OpenPGP detailed in RFC4880 [4], which would became the de facto standard for nearly all the word's encrypted emails.

Currently OpenPGP is under active development, and many clients support the use of OpenPGP compliant software to provide confidentiality, authentication or both. It is mainly used in email communications and Linux package management.

In OpenPGP Messages can be encoded as binary message, or ASCII text, and they are constructed from a sequence of records called *packet*. A packet is a chunk of data containing a packet header, beginning with a tag, used to identify the packet type, and a packet body. Some packet may contains other OpenPGP packets.

Packet Header is composed by:

```

-----BEGIN PGP MESSAGE-----
Version: OpenPrivacy 0.99
yDgBO22WxBHv7O8X7O/jygAEzol56iUKiXmV+X
mpCtmpqQUKiQrFqclFqUDBo
vzSvBSFjNSiVHsuAA==
=njUN
-----END PGP MESSAGE-----

```

Figure 1.1: Example of ASCII-armored PGP Message

Packet Tag: the first octet of the header, used to determine the format of the header and the content of the body part.

Packet Length: the length of the packet.

For compatibility reasons OpenPGP supports two versions format: old and new packet format. The following image shows how the Packet Tag is to be parsed (the leftmost bit is the most significant bit)

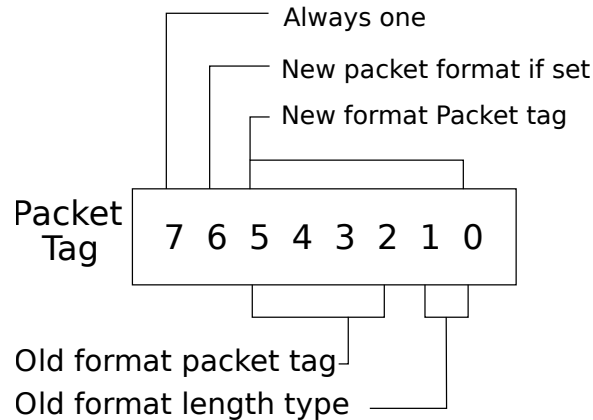


Figure 1.2: Composition of the Packet Tag byte, except the first bit, each value depend of the format: New on top, Old on the bottom

The following table describe what tag could be found, and to which packet corresponds. With the latest release of OpenPGP, several new non backward compatible tags were defined, so all tags beyond the 16 are part of the new specification, while the others could be used both for the old and new format.

Table 1.1: Packet tags explanation, Old Format is also used for Newer version

Tag	Format	Packet Semantics	Subpacket allowed
0	Both	Reserved	N
1	Both	Public-Key encrypted session key	N
2	Both	Signature	Y, see table 1.2
3	Both	Symmetric-Key Encrypted Session Key	N
4	Both	One-Pass Signature	N
5	Both	Secret-Key	N
7	Both	Secret-Subkey	N
8	Both	Compressed Data	N
9	Both	Symmetrically Encrypted Data	N
10	Both	Marker	N
11	Both	Literal Data	N
12	Both	Trust	N
13	Both	User ID	N
14	Both	Public-Subkey	N
17	New	User Attribute	Y, see table 1.3
18	New	Symmetric Encrypted and Integrity Protected	N
19	New	Modification Detection Code	N
60-63	New	Private/Experimental	N

A Public-Key Encrypted Session Key packet holds the session key used to encrypt a message.

A Signature packet describes a binding between some public key and some data. There are a number of possible meanings for a signature, which are indicated in a signature type octet, i.e. 0x20 indicate a Key Revocation Signature, while 0x01 is the Signature of a canonical text document. Follow the list of all Signature Subpackets

Table 1.2: Signature Subpackets explanation

Subpacket Tag	Semantics
0	Reserved
1	Reserved
2	Signature Creation Time
3	Signature Expiration Time
4	Exportable Certification
5	Trust Signature
6	Regular Expression
7	Revocable
8	Reserved
9	Key Expiration Time
10	Placeholder for backward compatibility
11	Preferred Symmetric Algorithms
12	Revocation Key
13	Reserved
14	Reserved
15	Reserved
16	Issuer
17	Reserved
18	Reserved
19	Reserved
20	Notation Data
21	Preferred Hash Algorithms
22	Preferred Compression Algorithms
23	Key Server Preferences
24	Preferred Key Server
25	Primary User ID
26	Policy URI
27	Key Flags
28	Signer's User ID
29	Reason for Revocation
30	Features
31	Signature Target
32	Embedded Signature
100 To 110	Private or experimental

The Symmetric-Key Encrypted Session Key packet holds the symmetric-key encryption of a session key used to encrypt a message.

The One-Pass Signature packet precedes the signed data and contains enough information to allow the receiver to begin calculating any hashes needed to verify the signature.

A Key Material packet contains all the information about a public or private key.

The Compressed Data packet contains compressed data. Typically, this packet is found as the contents of an encrypted packet, or following a Signature or One-Pass Signature packet, and contains a literal data packet.

The Symmetrically Encrypted Data packet contains data encrypted with a symmetric-key algorithm. When it has been decrypted, it contains other packets.

The Marker packet was used by an experimental version of PGP as the Literal packet, but no released version of PGP generated Literal packets with this tag.

A Literal Data packet contains the body of a message; data that is not to be further interpreted.

The Trust packet is used only within keyrings and is not normally exported. Trust packets contain data that record the user's specifications of which key holders are trustworthy introducers, along with other information that implementing software uses for trust information.

A User ID packet consists of UTF-8 text that is intended to represent the name and email address of the key holder.

The User Attribute packet is a variation of the User ID packet. It is capable of storing more types of data than the User ID packet, which is limited to text. The following table list the allowed subpackets:

Table 1.3: User Attribute Subpackets explanation

Subpacket Type	Semantics
1	Image
100-110	Reserved

The Symmetrically Encrypted Integrity Protected Data packet is a vari-

ant of the Symmetrically Encrypted Data packet. It is a new feature created for OpenPGP that addresses the problem of detecting a modification to encrypted data.

The Modification Detection Code packet contains a SHA-1 hash of plaintext data, which is used to detect message modification.

1.0.2 GPG

The Free Software Foundation has developed an OpenPGP compliant software, named **GNU Privacy Guard (GPG)**, available as a free software.

Users of GPG keeps collected keys in a local storage, known as *keyring*, which contains OpenPGP certificates.

Each user generates a collection of public-private key pairs, one for each username or email address which needs a separate identity. One is the *primary key*, the others are *subkeys*. Due to an established convention, the primary key pair is used only for signing purposes, while the subkeys are used to encrypt or sign.

The keyring is accompanied by the *trust-db*, which keeps track of *trust* and *validity* associated with each public key in the keyring. The trust level is the amount of trust the owner of the keyring has given to the public key owner. Validity is the “authenticity” metric of a key.

Trust GPG support the following trust levels:

Unknown automatically assigned to an imported key

Ultimate automatically assigned to the keyring owner’s keys

Full,Marginal,Untrusted which can be freely assigned by the user

Undefined automatically assigned whenever results impossible to determine one of the previous levels

Validity The possible validity levels for a key are: **Full, marginal, untrusted, unknown**.

The correct level to be assigned to a key is determined by GPG using the informations contained in the keyring, namely the trust and validity level of associated keys.

If a key is ultimately trusted, or have a signature verifiable by an ultimately trusted key are considered fully valid.

If a key have a signature verified by a fully valid, fully trusted key, the validity level is set to full.

If the signature of the key is verified by a fully valid but only marginally trusted keys its validity level is set to marginal. If three such signatures are found on the same key, its validity is promoted to full.

As the validity level names may suggest: a key is considered valid only if it is *Fully* trusted.

Revocation Is the process that allow to *revoke* a public key pair, actually nullifying its value, and the value of its signatures.

The revocation is performed by means of a *revocation certificate*, which is generated starting from the private key, and can be distinguished in:

- primary key revocation
- subkey revocation
- signature revocation

The revocation certificate is a special type of self-signature, with signature type 0x20, and an addition data field explaining the reasons behind the revocation.

Since it is a signature, the updated certificate of the user requesting the revocation needs to be propagated to other user to be effective.

1.0.3 Keyserver

A keyserver is a piece of software running on a computer, capable of receiving, storing and serving cryptographic keys to users and programs. Often keyservers are capable of synchronization, that is, the possibility to exchange keys between hosts, with the aim to provide a reliable network in which keys are never lost and always available.

Before keyserver, two parties which would like to exchange messages over a secure channel, needed to locate and exchange keys, which is a time-consuming and insecure if done over an insecure channel. This problem is known as bootstrapping problem. Keyservers act as central repositories of keys, where everyone could search and securely retrieve a public key.

The first keyserver was written, as a thesis, by Marc Horowitz [9], which was designed to handle large numbers of keys, and put the basis for a new generation of keyserver. It established the functional specifications to search, accept requests via email, query via http and replicate.

Follows a number of independently developed keyserver:

onak is a keyserver started to support multiple subkey bindings when at the time, there was no keyserver supporting it, and the owner's key pathfinder [11].

Licensed under GNU Public Licence, the last known update of onak is dated 2016-08-28

CryptNet is developed in C under GPLv2. Its design goals are support to official OpenPGP release, validation of keys packets, support to free RDBMS, validate signature, handle expire and revocation, synchronization with other keyserver [3].

Latest commit to the official repository is dated May 2, 2014

peegeepee is a free to use Public PGP key server, created by a small team of IT professionals. Unfortunately their keyserver implementation appears to be closed-source, so nothing is known about the way the parse/store/manage keys apart from what could be understood from the FAQ section of their site [14].

sks-keyserver is an OpenPGP keyserver whose goal is to provide easy to deploy, decentralized, and highly reliable synchronization, written in Ocaml, with Berkeley DB as database [6]. This is the current state of the art regarding public keyserver, many deployment of sks-keyserver are working and kept in sync with each other in pools of server.

Last commit to the official repository dated 2016-08-07.

hockeypuck is a public keyserver written in Go under AGPL3 licence [10], supports the current OpenPGP standard, is able to synchronize with

the current state of the art, and supports modular storage backend. Hockey puck is currently deployed in pair with a few sks-keyserver implementation to test the robustness of the software.

1.1 State of the Art

Before the current state of the art the adopted way of key server synchronization was based on e-mails exchange between each server in the synchronization network. Whenever a key server receives new informations (either uploaded by a user or received from another key server), it forwards these information to all other key servers in its synchronization list.

Since the old e-mail based protocol was not able to scale very well (especially with the number of servers in the synchronization list), a new protocol was defined for the Synchronizing Key Server (SKS), which is based on an efficient algorithm for set reconciliation, with respect to communication complexity [12], which is now regarded as the current state of the art.

SKS defined the *gossip* functionality, which each client uses to reach out the server in its synchronization list and start the reconciliation process.

From a very basic point of view, the set reconciliation organizes the keys in a n-arity tree, the *partition tree* (**ptree**), which allows to find the differences easily without transmitting a lot of information. Only the modified keys are then exchanged.

The state of the art server implementation consists of two single thread processes.

sks-db, fulfills the normal jobs associated with a public key server, such as answering web requests, allow OpenPGP compliant software to upload and download public keys, and to other keyserver to collect keys after a succesful run of the synchronization protocol. When a new key is received from a client, it get stored into the *Key Database*, its MD5 hashsum is computed and stored into the *PTree Database*. *sks recon* does all the work with respect to reconciling hosts databases. It uses the informations stored into the *PTree Database* to efficiently determine the differences between its own *Key Database* and that of another host.

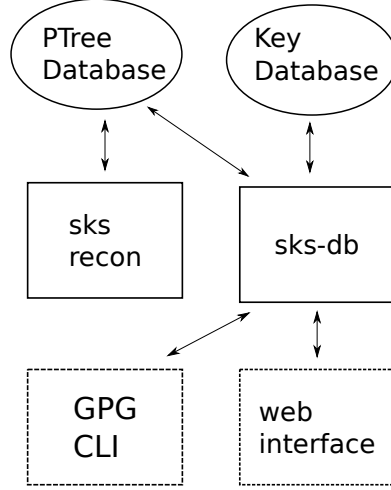


Figure 1.3: sks-keyserver internals: circles are datasets, full line boxes are processes, dashed line boxes are user inputs, arrows are data exchange between components

sks synchronization A group of sks keyserver form a pool capable of **gossip**, meaning the act of starting a synchronization process as client, and **serve**, which is the server part of the recon protocol.

The servers included in the pool responded during the last synchronization, are updated to the required minimum version of the software and actively sync with the rest of the network to update the keys.

The recon process does not synchronize directly the keys, but their MD5 instead. The process involves a **Prefix Tree (ptree)** which holds the hash-sum of all certificate stored by the server in a balanced tree structure. The main concept behind the mathematical model of the synchronization algorithm are explained as a set reconciliation problem [12].

Given two sets S_A and S_B , their differences (as in set difference) $\Delta_A = S_A \setminus S_B$ and $\Delta_B = S_B \setminus S_A$ and the max allowed sizes for the difference sets

$m_A = |\Delta_A|, m_B = |\Delta_B|$ we can define $m = m_A + m_B$ as the maximum number of differences between the two sets.

Considering the trivial case in which $m_A = 1$ and $m_B = 0$, it is possible to reconcile S_A and S_B using a single message with fixed number of bits: the key idea is that the elements of S_A and S_B cancel each other out, leaving only the *parity sum* of the difference, which is the missing element. Host A computes the parity sum of its bitstrings ($parity_A$) and sends it to B . Host B do the same but instead store its parity and receive the one from A , then computes the parity sum of the two and recover the differences.

It is possible to generalize the above mentioned algorithm to the case of multiple differences. Given the set $S = x_1, x_2, \dots, x_n$ we can define its *characteristic polynomial* as $\chi_S(Z) = (Z - x_1)(Z - x_2)(Z - x_3) \dots (Z - x_n)$.

Each element of the set is mapped on an element of the field \mathbb{F}_q where the number of elements of the field (q) should be chosen to be large enough to uniquely label all the elements of both S_A and S_B .

The characteristic polynomial is just a representation of a set, so it contains all the information contained in that set, and its transmission is not cheaper than transmitting the whole set. However, the polynomial allows the canceling operation which was at the core of the parity sum protocol.

In fact, if we consider the ratio between the characteristic polynomials of two sets S_A and S_B

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

We obtain that elements both present in S_A and S_B cancels out, leaving the characteristic polynomials of the differences.

To efficiently compute the ratio of their characteristic polynomials, without actually sending them, the division is performed on the values of the two polynomials at a collection of predetermined evaluation points. The result can be used to interpolate the rational function $\frac{\chi_{\Delta_A}(Z)}{\chi_{\Delta_B}(Z)}$, which solutions (zeros) are exactly the missing elements of the sets.

If an upper bound \bar{m} on m is known, the two hosts can proceed by evaluating the polynomials at the same \bar{m} evaluation points. The ratio of the evaluation is computed at each point, and those values are used to interpolate the rational function. Once reduced and factorized, roots of numerator and

denominator of the rational function will yield the missing elements.

Finding the root for polynomial $f(Z)$ of degree m over a finite field \mathbb{F}_q needs three steps:

First, $f(Z)$ needs to be square-free, by computing the greatest common divisor between the polynomial and its first derivative.

Second, all the irreducible factor of $f(Z)$ needs to be linear: condition verified if $f(Z) = \text{GCD}(f(Z), Z^q - Z)$.

Third, find all the linear factors, i.e using probabilistic techniques

Determining the rational function starting from the values is called *rational interpolation problem*. In general, given d_1 and d_2 degrees of numerator $P(Z) = \sum_i p_i Z^i$ and denominator $Q(Z) = \sum_i q_i Z^i$, of the target rational function and a *support* set V composed by $d_1 + d_2 + 1$ pairs $(k_i, f_i) \in \mathbb{F}_q^2$, there is a unique rational function f such that $f(k_i) = f_i$ for each (k_i, f_i) in V .

Each pair (k_i, f_i) implies a linear constraint of the coefficients of the numerator and denominator of the resulting rational function:

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

Interpolation is achieved by solving the $d_1 + d_2 + 1$ simultaneous linear equations implied by the elements of V .

Since we have a bound \bar{m} on the total degree rather than individual bounds on numerator and denominator, given $\delta = m_A - m_B$ we need to compute bounds on m_A and m_B .

$$m_A \leq \lfloor \frac{(\bar{m} + \delta)}{2} \rfloor = \bar{m}_A$$

$$m_B \leq \lfloor \frac{(\bar{m} - \delta)}{2} \rfloor = \bar{m}_B$$

The computational complexity of this algorithm is dominated by the cost of interpolation using Gaussian elimination to solve the linear system: $O(m^3)$ operations over \mathbb{F}_q

The communication complexity is given by the exchange of \bar{m} evaluations of polynomials, along with the sizes of S_A and S_B : $(b+1)\bar{m} = (\bar{m}+1)(b+1)-1$ bits.

The problem of this protocol is the required bound \bar{m} . In absence of such a bound, a pair of hosts could reconcile their sets by executing the protocol using progressively larger values for \bar{m} , because if the rational interpolation fails for whatever reason, it means that the difference is too high.

To avoid such trial-and-error mechanism we can define two parameters:

- \bar{m} , the *recovery bound*, as the maximum allowed value for m .
- k , the *redundancy factor*, which determines the probability of detecting that the reconciliation is not possible.

Such probability is defined as

$$\varepsilon_k = \left(\frac{|S_A| + |S_B|}{2^b}\right)^k \text{ if } m > \bar{m}$$

The general framework for the algorithm can be abstracted into four primitives [13]:

- $\text{INIT_CS}(\bar{m}, k)$: Returns an initial checksum with recovery bound \bar{m} and redundancy factor k
- $\text{ADD_ELEMENT}(cs, \beta)$: Update checksum cs adding β
- $\text{DEL_ELEMENT}(cs, \beta)$: Update checksum cs deleting β
- $\text{RECOVER}(cs_A, cs_B)$: If $|\Delta_A| + |\Delta_B| \leq \bar{m}$ then missing elements are returned, otherwise fail is returned with probability ε_k

Each hosts have to maintain checksums cs_A and cs_B corresponding to the sets S_A and S_B . To initiate a reconciliation, host A send cs_A to B , which computes the set difference via RECOVER and sends it to A .

As we already seen, the computational complexity is cubic in \bar{m} . To reduce it we can recursively split the checksums into p non intersecting partitions, such that $\bigcup_i P_i = P$ until the algorithm succeeds with a predetermined bound \bar{m} of the number of differences.

Given two sets it is possible to reconcile them by separately reconciling $S_A \cap P_i$ and $S_B \cap P_i$ for each partition and then taking the union of the recovered sets.

This modified version uses an additional parameter: the *partitioning factor* p , which is the number of equally sized partitions created at each level.

Each partition P is classified in *active* if $m_P > \bar{m}$ (in this case the algorithms will recursively descend into sub-partitions of P), *terminal* if it is the root partition or the sub-partition of an active partition and $m_P \leq \bar{m}$, *inactive*, if it is descended from a terminal partition.

To further cut the computation complexity instead of calculating the checksums at each reconciliation it is possible to compute checksums incrementally as elements are inserted and deleted from the set, storing them in a structure called *partition tree*, a p -arity tree in which feature two types on nodes: *internal nodes* and *leaf nodes*.

Considering the partition tree for a generic set S , an internal node corresponding to a partition P contains the checksum csp_P of the elements $S \cap P$, a leaf node corresponding to partition P contains only the set $S \cap P$. In the tree there is an internal node for every partition P such that $|P \cap S| > \bar{m}$, so it is always possible to obtain the checksums necessary for reconciliation.

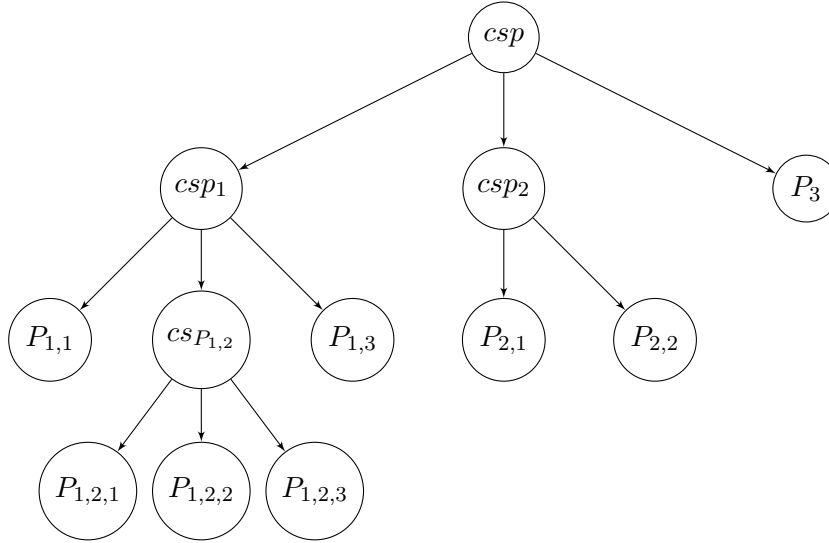


Figure 1.4: Example of partition tree:
Internal nodes contains the CheckSum of the set union of Partitions,
Leaf node represent Partitions

When an element is added to S it is necessary to descend the tree, using `ADD_ELEMENT` to update the checksums for all the internal nodes representing partition to which the element belong, up to the leaf node whose

partition contains the element. When removing an element from S , `DEL_ELEMENT` have to be used to update the same checksums updated during insert.

If the number of elements in a leaf node is raised above \bar{m} it must became an internal node, and if the number of elements in a leaf node is less then $\bar{m} + 1$, it needs to be converted to an internal node.

The expected depth of the partition tree at S is bounded by $\left\lceil \log_p \left\lfloor \frac{|S|}{\bar{m}} \right\rfloor \right\rceil +$
4

Chapter 2

Reconciliation Daemon

2.1 Analysis of the Recon Protocol

What follow is an analysis and subsequent reengineering of the synchronization protocol. After the discussion of the mathematical model of the previous chapter we now focus on more technical aspect, including the actual implementation in the state of the art. In this chapter will be presented a top-bottom approach on recon protocol: recon mechanism with message format explanation, how the keys are actually exchanged between peers, internals of the *ptree*. Will finish the chapter a brief explanation of how keys are imported into Peaks: certificate parsing, generating hashes, and problems found.

2.1.1 High level view

First, an high level overview accompanied with sequence diagrams will explain how messages are exchanged between peers, and then will follow a detained explanation on how messages are composed and managed.

As a premise, because the network of keyservers is configured as peer-to-peer, but the interaction between hosts is implemented as client-server, every hosts act alternatively as client and server, following a certain schedule (i.e. hosts start the client recon every 60 seconds, but during the wait time can accept incoming connection as server, blocking the client part until done)

When a client and a server wants to start the reconciliation process they first of all proceed exchanging the own config.

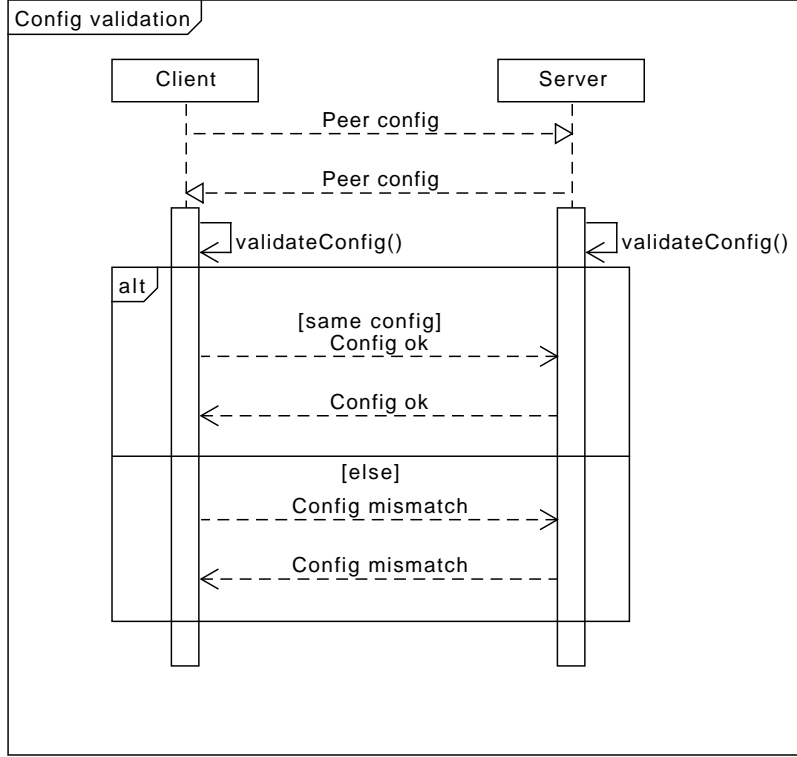


Figure 2.1: config validation

The client, upon connecting to the server sends its own config, and receives the server's one. If the configs have the same values of $mbar(\bar{m})$, $bitquantum$, $version$ and hosts are not already in a synchronization process with another peer, it is possible to start reconciliation, else the recon process will fail, and the peers exchange a message with the reason for the failure.

The reconciliation proceed by trying to apply the linear interpolation algorithm with partitioning, previously defined in its mathematical model, for which will be outlined an actual implementation.

The partitions are computed before starting the recon, by collecting hash-sums of the stored certificates and creating a tree known as **Prefix Tree** (**ptree**), from which the root is selected as first partition, then if the reconciliation algorithm fails (i.e., the number of certificates which are not shared between the server exceeds the value of \bar{m} in the configurations), child nodes are retrieved from the tree and used as active partition, and so on recursively

until a leaf node is reached.

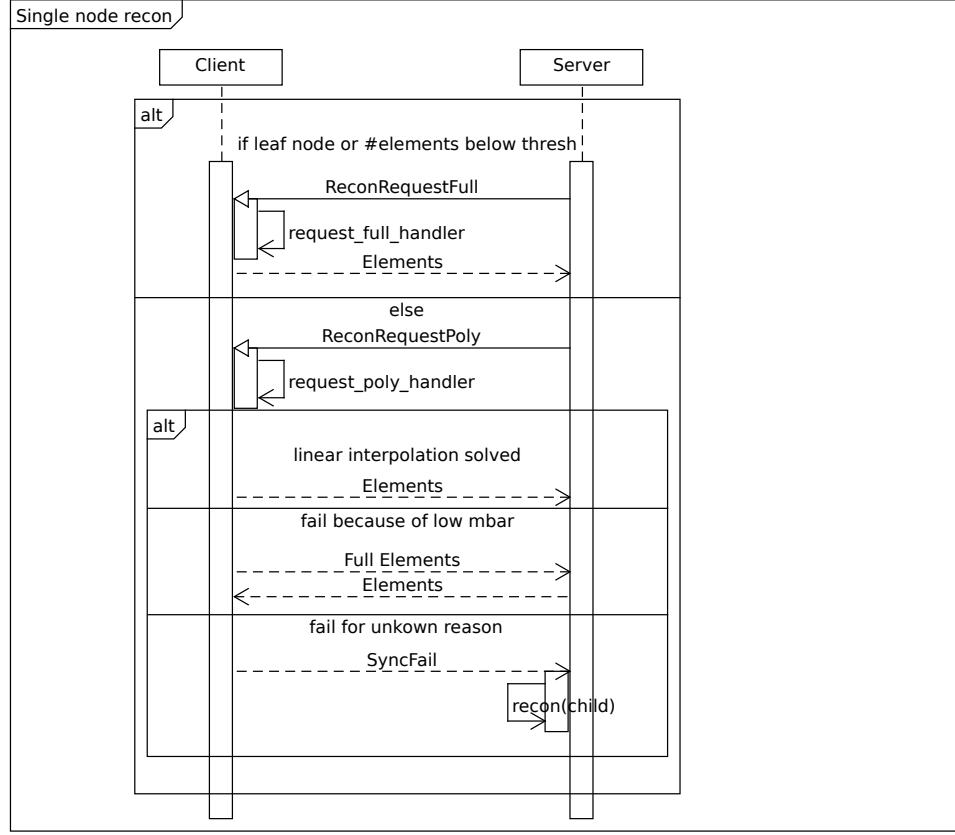


Figure 2.2: single node recon

The server starts by fetching a node from the ptree (starting with the root node) and sends a *ReconRequestFull* if the node is a leaf node or if the number of elements in its child nodes is less than the *split threshold*, a *ReconRequestPoly* otherwise. The client replies recovering the corresponding node from its own ptree and comparing the elements of its node with the elements received from the requests and missing parts (the set difference) to the server.

If the node does not satisfy the two requirements a *ReconRequestPoly* is sent, which is the starting point to perform the linear interpolation, because the request embed the evaluation of the characteristic polynomial in the pre-determined interpolation points. The result of the SOLVE operation determines what may follow:

- If it is solved correctly an *Elements* request is sent to the server con-

taining the set of elements the client needs and the client stores the set of elements the server needs

- If it fails because m is too low, a *FullElements* is sent containing information of the node for which the SOLVE operation failed, the server will proceed to perform a basic set difference, stores the client needs and sends its needs to the client
- If it fails for another reason (i.e. finding the solution is mathematically unfeasible) the client sends a *SyncFail* to the server, that will proceed to apply recursively the algorithm to the children until the full set of differences is found.

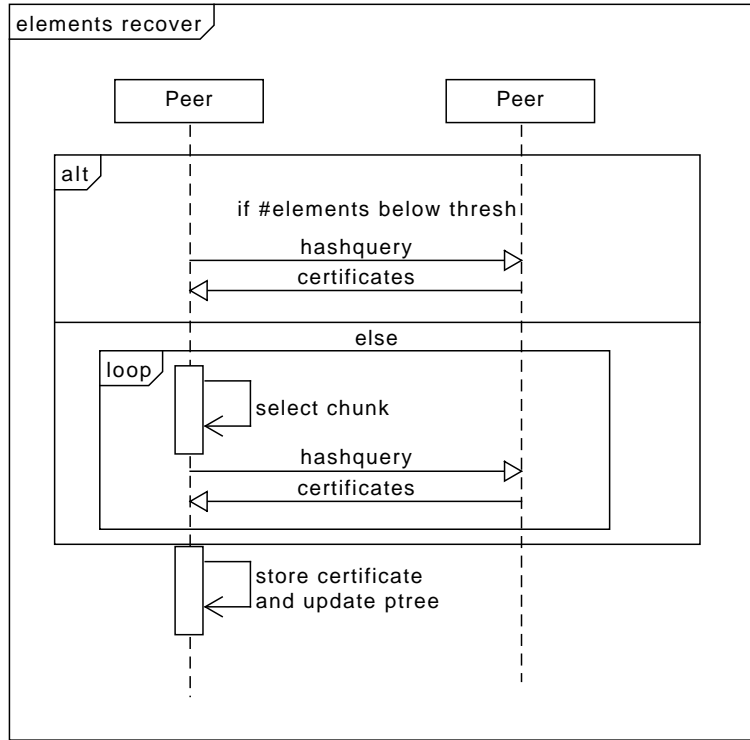


Figure 2.3: elements recover

Once the reconciliation process ends client and server know the hashes of the elements to recover.

The steps for recovering the found elements are the same for both peers, If the number of elements to recover is below the *request chunk size* threshold

a request is sent to the other peer containing all the hashes, otherwise the elements are divided in chunks, and multiple requests need to be performed.

Upon receiving a hash request a peer reply with the full certificate which corresponds to the given hash, so that the requesting peer is able to store the certificate in its own database and update the ptree with the new information.

2.1.2 Set reconciliation

This section will be a depth analysis of the reconciliation protocol and algorithms. Some specific data structure will be explained in place, while the pieces which needs to be treated separately, such as the message exchanged between peers and the structures used will be outlined in specific sections at the end.

Before even starting recon, the system administrators needs to agree to recon by placing each other address/port combination into a specific file, called *membership*, otherwise the server will ignore the connection.

The *membership* file contains a line, composed by hostname, recon port and an optional comment beginning with a “#”, for every host to recon with.

```
yourserver.example.net 11370 # Your full name <emailaddress for admin purposes> 0xPreferrefPGPkey
keyserver.gingerbear.net 11370 # John P. Clizbe <John@Gingerbear.net> 0xD6569825
sks.keyservers.net 11370 # John P. Clizbe <John@Gingerbear.net> 0xD6569825
keyserver.rainydayz.org 11370 # Andy Ruddock <andy.ruddock@rainydayz.org> 0xEEC3AFB3
keyserver.computer42.org 11370 # H.-Dirk Schmitt <dirk@computer42.org> 0x6A017B17
```

Figure 2.4: Example of membership file

When a client wants to *gossip* with a server, it selects a random line from the file, from which it retrieves the address and port of the remote server to establish a connection.

Then client and server exchanges:

- \bar{m} , necessary to determine the compatibility of thresholds for the linear interpolation
- *bitquantum*, necessary to determine the compatibility of ptree structure
- *version*, necessary to determine the software compability
- *http port*, necessary to know on which port host serve certificates

- *other*, an optional series key-value strings, actually unused.

Each host sends its own config and receives the other, so that both could check the equivalence of the settings.

If the settings are the same, each host sends as confirmation message a string containing the text “passed”.

If even one of the settings has a mismatch it generates an error message with a brief explanation for the rejection, so it will send two strings, one with the text “failed”, and the other one with the reported error, for instance “mismatched bitquantum”, or “different version”.

Failures during this phase can occur also because the host is already syncing with another peer: the recon protocol cannot handle more than two peers at once, furthermore database and ptree needs to be modified in an atomic manner so the new reconciliation attempts are blocked.

After the config has been checked successfully the client and server follows different paths: the server coordinates the recon by sending messages to the client and managing the larger part of the bookkeeping required by the algorithm.

Both will keep in a local array the hash of the certificates which needs to be collected from the other peer.

client recon

The client keeps a queue of messages and loops forever responding to server’s requests. It could receive a:

- **ReconRequestPoly**, containing the data necessary to execute the linear interpolation algorithm. The client fetches from the ptree the data corresponding to the node which prefix is included into the request and tries to recover the elements. If the node is not present, the tree is searched for the nearest parent, towards the root, which is selected instead.
 - If succeeds an *Elements* message is created containing the recovered local needed elements in the message queue and stores the remote needed elements.

Algorithm 2.1.1: client recon main loop

Data: Message: incoming message from the network
local needs: vector of ZZp elements which collect all elements
needed by client

Result: missing certificates are recovered

Init local needs;
repeat
 read message;
 switch *message type* **do**
 case *ReconRequestPoly*
 | local needs \leftarrow REQUEST POLY HANDLER(Message);
 case *ReconRequestFull*
 | local needs \leftarrow REQUEST FULL HANDLER(Message);
 case *Elements*
 | local needs \leftarrow (Message \rightarrow elements);
 case *Done*
 | /* end of the loop */
 case *Flush*
 | SEND PENDING MESSAGES;
 otherwise
 | /* end of the loop with error condition */
 | ERROR ;
 until *Done received or maximum recover threshold exceeded*;
 FETCH ELEMENTS(local needs) ;

- If fails because \bar{m} is too low and the analyzed node is a leaf or the children are empty a *FullElements* is stored in the message queue
- If it fails for another reason it stores a *SyncFail* message in the queue

Algorithm 2.1.2: REQUEST POLY HANDLER

Input: Request: ReconRequestPoly Message
Output: local needs: vector of ZZp
Data: remoteSamples: vector of ZZp
 prefix: bitstring
 localNode: pnode recovered from the ptree using the prefix in the request
 localSamples: vector of ZZp
 remote needs: set of ZZp elements needed by the remote peer

remoteSamples \leftarrow get samples from incoming request;
 prefix \leftarrow get prefix from incoming request;
 localNode \leftarrow NODE(prefix);
 localSamples \leftarrow get samples from localNode;
if remote needs, local needs \leftarrow SOLVE (localSamples, remoteSamples)
then
 | generate Elements request with remote needs;
 | **return** local needs;
else
 | **if** linear interpolation raise error "low mbar" and
 | target node is leaf or have a number of elements less than split
 | threshold **then**
 | | generate FullElement request with target node information;
 | **else**
 | | generate SyncFail request;

- **ReconRequestFull**, containing the data used to perform a classic set difference. The client fetches the node corresponding to the one in the request, and performs a symmetric difference between its own node elements and the elements received in the request. An *Elements* message is created containing the recovered local needed elements in the message queue and stores the remote needed elements.
- **Elements**, containing the response of a *FullElements* request. The elements contained are stored, since they are assumed to be missing.

Algorithm 2.1.3: REQUEST FULL HANLDER

Input: Request: ReconRequestFull Message
Output: local needs: vector of ZZp elements
Data: node: pnode
 remote needed elements: vector of ZZp elements
 node \leftarrow search node with the prefix in the request;
 local elements \leftarrow GET ELEMENTS(node);
 remote elments \leftarrow get elements from Request;
 local needs, remote needs \leftarrow SYMMETRIC DIFFERENCE(local elements,
 remote elements);
 generate Elements request with remote needed elements;
return local needs;

- **Done**, signal the end of the recon protocol. No more messages are sent/received: from this point both peers will proceed to fetch the stored elements.
- **Flush**, signal received when the server needs to receive the pending responses from the client, which will send all stored messages at once.

Each received message marks a *step* in the communication between peers. If at the end of a step, not started by a **Done** message, the client detects that the maximum recover threshold, as in the maximum amount of recoverable certificates in one round of recon, has been reached, it goes directly to certificate recovering. If a message different from the above mentioned is received, the recon protocol stops with an error, and the server is notified with an **Error** message.

linear interpolation algorithms

The linear interpolation algorithm is the crucial part of the whole recon protocol, since it is used to effectively reduce the time complexity of the process. It is started when the server sends a *ReconRequestPoly*, which the client can use to perform the interpolation using the samples provided in the request (*remote samples*) and its own samples (*local samples*).

The algorithm operate of a Matrix, which identify the linear equations system described in the previous chapter, which needs to be solved in order to find the missing elements.

First *local samples* and *remote samples* are used to populate the matrix. Then a series of operation are performed on the matrix.

The *gaussian elimination* is a sequence of elementary row operations to modify the matrix until the lower left-hand corner of the matrix is filled with zeros, as much as possible. In particular three operations are performed: Swapping two rows, Multiplying a row by a scalar, Adding a multiple of one row to another. By means of a correct gaussian elimination, a matrix can always be transformed into an upper triangular matrix, or *row-echelon form*.

From the *row-echelon form* the *back substitution* is applied to solve the system represented by the Matrix, In an upper triangular matrix, the substitution need to be performed starting from the last row, and then substituting backward in rows, toward the first, solving all equations.

The resulting values are the coefficient of the interpolated rational function.

To recover the missing elements the greatest common divisor of numerator and denominator is calculated and simplified from both factors, which are now minimal.

Finding the roots of numerator and denominator (if any) will yield the respectively missing elements of the sets. The problem of actually finding the roots of the polynomials has been discussed in chapter 1. In Peaks the primitive of the NTL C++ Library have been used to avoid reimplementing of all the mathematical operations.

server recon

The server starts the actual reconciliation by sending the first message of the protocol. To manage the communications it keeps two queues, the *request queue* (which actually is a deque since it is needed to append on both ends) and the *bottom queue*. An entry for the *request queue* is composed by a node reference and a bitstring.

The *request queue* is used to enqueue all the pending requests. A *request entry*, an entry for the *request queue*, is composed by the reference to a **pnode**, and its **bitstring**. When a request has to be processed, it is popped from the queue, send to the handler, which generate a message for the client, depending of the pnode encapsulated in the request. To the processed request is attached a *Bottom State* and it is moved into the *bottom queue*.

Algorithm 2.1.4: SOLVE LINEAR INTERPOLATION

Input: remote samples: vector of \mathbb{ZZp} of remote interpolation values
 local samples: vector of \mathbb{ZZp} of local interpolation values
 interpolation points: vector of \mathbb{ZZp} representing points used for interpolation
 M: matrix composed of \mathbb{ZZp} numbers
 Numerator, Denominator: polynomials over \mathbb{ZZp}

Output: client needs
 server needs

Data:
 for $i \leftarrow 0$ to server samples size do
 | values[i] \leftarrow rsamples[i]/lsamples[i];
 diff \leftarrow |serversamplessize – clientsamplessize|;
 if diff > samples from server size - 1 then
 | /* too much difference to interpolate */
 | ABORT;
 M \leftarrow POPULATE MATRIX(VALUEs);
 GAUSSIAN ELIMINATION(M);
 BACK SUBSTITUTE(M);
 /* Recovering characteristic polynomials from matrix */
 Init $poly_a$ and $poly_b$;
 for $i \leftarrow 0$ to m_a do
 | set $poly_a$ coefficient to $M(i, \bar{m})$;
 for $i \leftarrow 0$ to m_b do
 | set $poly_b$ coefficient to $M(i + m_a, \bar{m})$;
 set both polynomials lead coefficient to 1;
 $poly_g \leftarrow GCD(poly_a, poly_b)$;
 /* recovering numerator and denominator of the rational function */
 Numerator $\leftarrow poly_a / poly_g$;
 Denominator $\leftarrow poly_b / poly_g$;
 if Numerator and Denominator are monic, and have a number of distinct roots equal to their degree then
 | remote missing elements \leftarrow FIND ROOTS(Numerator);
 | local missing elements \leftarrow FIND ROOTS(Denominator);
 return remote missing elements, local missing elements

Algorithm 2.1.5: POPULATE MATRIX

Input: Values: vector of \mathbb{ZZp} containing the ratio of the interpolation values

Output: M: matrix $\bar{m} \times \bar{m} + 1$ populated with linear equation system data

```

/* populate matrix                                     */
p  $\leftarrow$  Values[0];
 $\bar{m} \leftarrow$  Values size;
if ( $\bar{m} + diff$ ) is even then  $\bar{m} = \bar{m} - 1$ ;
 $m_a \leftarrow (\bar{m} + diff)/2$ ;
 $m_b \leftarrow (\bar{m} - diff)/2$ ;
define M Matrix ( $\bar{m} \times \bar{m} + 1$ );
for  $i \leftarrow 0$  to  $\bar{m}$  do
     $acc \leftarrow 1$ ;
     $k_i \leftarrow points[i]$ ;
     $f_i \leftarrow values[i]$ ;
    for  $j \leftarrow 0$  to  $m_a$  do
         $M(i, j) \leftarrow acc$ ;
         $acc \leftarrow acc \cdot k_i$ ;
     $k_i m_a \leftarrow acc$ ;
     $sum \leftarrow -f_i$ ;
    for  $j \leftarrow m_a$  to  $\bar{m}$  do
         $M(i, j) \leftarrow acc$ ;
         $sum \leftarrow sum \cdot k_i$ ;
     $f_i k_i m_b \leftarrow -sum$ ;
     $M(i, \bar{m}) \leftarrow f_i k_i m_b - k_i m_a$ ;
return M;

```

Algorithm 2.1.6: GAUSSIAN ELIMINATION

Input: Matrix: matrix of $\mathbb{Z}\mathbb{Z}\mathbb{p}$ on which perform the algorithm
Result: Matrix will be in row-echelon form

```

 $h \leftarrow 0;$ 
/* Initialization of the pivot row */
 $k \leftarrow 0;$ 
/* Initialization of the pivot column */
while  $h < \text{Matrix rows and } k < \text{Matrix columns}$  do
    /* Find the k-th pivot */
     $i_{max} \leftarrow \text{argmax}(i = h \text{ to } m, \text{abs}(A[i, k]));$ 
    if  $M(i_{max}, k) = 0$  then
        /* No pivot in this column, pass to the next */
         $k \leftarrow k + 1;$ 
    else
        swap rows( $h, i_{max}$ );
        /* Do for all rows below pivot */
        for  $i \leftarrow h + 1$  to  $m$  do
             $f \leftarrow A[i, k]/A[h, k];$ 
            /* Fill with zeros the lower part of pivot column */
            /*
             $A[i, k] \leftarrow 0;$ 
            /* Do for all remaining elements in current row */
            for  $j \leftarrow k + 1$  to  $n$  do
                 $A[i, j] \leftarrow A[i, j] - A[h, j] \cdot f;$ 
            /* Increase pivot row and column */
             $h \leftarrow h + 1;$ 
             $k \leftarrow k + 1;$ 

```

Algorithm 2.1.7: BACK SUBSTITUTE

Input: Matrix: matrix of $\mathbb{Z}\mathbb{Z}\mathbb{p}$ already in row-echelon form
Result: linear system of equations represented by Matrix get solved.
The Matrix, modified in place, will be an identity sub-matrix, plus the column with the resulting coefficients

```

for  $j \leftarrow \text{Matrix Rows} - 1$  to 1 do
     $last \leftarrow \text{Matrix Rows} - 1$ ;
    for  $j2 \leftarrow j - 1$  to 0 do
         $scmult \leftarrow M(j2, j)$  ;
        for  $i \leftarrow last$  to  $\text{Matrix Columns} - 1$  do
             $sval \leftarrow M(j, i)$ ;
            if  $sval \neq 0$  then
                 $v \leftarrow M(j2, i)$ ;
                 $v \leftarrow v - (sval \cdot scmult)$ ;
                 $M(j2, i) \leftarrow v$ ;
         $M(j2, j) \leftarrow 0$ ;
```

The *Bottom queue* is used to manage all the current ongoing message exchange. An entry (*Bottom entry*) is composed by a *request entry*, to which a state flag is attached. This flag can assume two values: 0 (Bottom), the starting value, or 1 (FlushEnded), which means that the messages for this requests have been already processed and can be discarded.

Once the server completes the configuration check, it generates the first request entry, which holds information on the root node of the ptree, and pushes the request in the *request queue*.

The server now will enter its main loop until a *Done* message is generated. It will check if the *bottom queue* is empty, i.e when the recon process has just started, so it will pop one from the *request queue*, and analyze its content:

- If the node contained in the request is a leaf or its children are empty a *ReconRequestFull* is generated with the key and the elements contained in the node
- Else a *ReconRequestPoly* is generated with the key, the samples and the size of the node.

Whatever message is generated it is stored in the message queue, and a *bottom request* is added to the corresponding queue.

Algorithm 2.1.8: server main loop**Input:****Output:**

Data: local needs: vector of ZZp elements needed by the server
 request queue: deque of generated request starting from the current state of the recon protocol
 bottom queue: queue of requests currently in progress
 msg: Message read asynchronously from the network
 request: request taken from the request queue ready to be processed (the output will be a message for the other peer)

Result: server recover missing certificates

Init local needs;

insert initial request into request queue;

repeat **if** *bottom queue is empty* **then**
 request \leftarrow pop from the request queue;
 SEND REQUEST(request);
 else bottom \leftarrow get the top from bottom queue; **switch** *bottom* \rightarrow *state* **do** **case** *FlushEnded*
 pop item from the bottom queue;
 toggle flush off ;
 case *Bottom* **if** *there is an incoming message* **then**
 msg \leftarrow readMessage;
 pop from the bottom queue;
 HANDLE REPLY(msg, bottom \rightarrow request);
 else
 if *bottomQueueSize > max outgoing recon requests*
 or *request queue is empty* **then**
 if *flush on* **then**
 pop from the bottom queue;
 if *there is an incoming message* **then**
 msg \leftarrow read message;
 local needs \leftarrow HANDLE REPLY(MSG,
 BOTTOM.REQUEST);
 else

FLUSH QUEUE;

else
 request \leftarrow pop from the request queue;
 SEND REQUEST(request);
 until *Done has been generated*;

send done;

FETCH ELEMENTS(local needs);

Algorithm 2.1.9: SEND REQUEST

Input: request: a request entry taken from the request queue
Output:
Data: message: Message for the client generated from the request
bottom request: bottom entry, input request get a flag and it is moved into the bottom queue
elements: vector of ZZp elements of the pnode which is referenced in the request
svalues: vector of ZZp elements, which represent the checksum of the pnode's partition
num elements: int number of all the elements under the pnode
Result: a message is ready to be send
if *node in request is leaf or have less than split threshold elements*
then
 elements \leftarrow GET ELEMENTS(request \rightarrow pnode);
 message \leftarrow ReconRequestFull(request \rightarrow key, elements);
else
 num elements \leftarrow GET NUM ELEMENTS(request \rightarrow pnode);
 svalues \leftarrow GET SVALUES(request \rightarrow pnode);
 message \leftarrow ReconRequestPoly(request \rightarrow key, num elements, svalues);
add message to message queue;
bottom request \leftarrow request + state Bottom;
add bottom request to bottom queue;

Algorithm 2.1.10: FLUSH QUEUE

Input:
Output:
Data: message queue: vector of Messages which store the pending message for the client
void request: a void request is added with state Flush just to signal that the flushing of the messages has been done
Flushing state: get changed to True
add Flush message to message queue;
send all queued messages ;
void request \leftarrow bottom request with state FlushEnded;
PUSH(bottom queue, void request);
toggle Flush on;

If it is not empty a request is gathered from the top of the *bottom_queue*, depending on the flag value:

- If flag is *FlushEnded* then a request is taken from the bottom of the *bottom_queue* and the flush flag is set to False
- If the flag is *Bottom* then
 - if there is an incoming message, it is read, an item is popped from the *bottom_queue*, and the message, along the request is handled
 - else
 - * if the current size of the bottom queue is more that the maximum outstanding recon request threshold or the *request_queue* is empty, then if flush flag is active an item is popped from the *bottom_queue*, a message is read from the network if any, and it is handled alongside the request, else the message queue gets flushed.
 - * else an item is popped from the *request_queue* and is sent to processing.

fetching certificates

Once client and server main loops end with a *Done* message, they both collect an array with the elements missing from their own sets. Since this array contains the hashes of the certificates to recover it is necessary to query the other peer to actually recover the key material. Normally, the number of keys which could be recovered in a single recon session is limited by the *max_element_recover* threshold (which by default is 15000). However this number is still huge if a peer would attempt to recover all the elements at once, because a single key, depending on the encryption algorithm used, on the presence of user attribute can grow in the order of ten to hundred of Kilobytes. So in the fetching phase, the elements are requested in chunks of fixed size, i.e. 100 at once.

If the elements to recover are less than the fixed size, then they are requested all at once.

The request for the missing keys is a POST request to the db part of the keyserver, so the port that is included into the config file at the start of

Algorithm 2.1.11: HANDLE REPLY

Input: Message: Message to be handled

Output: local needs: vector of ZZp which represent elements needed by the server

Data: Message header: first byte of the Message, will tell which type of Message was received.

childVector: vector holding the children of the pnode in the message

req: request entry generated from the above mentioned child nodes

local elements: vector of ZZp elements recovered from the pnode embed into the request

remote elements: vector of ZZp elements embed into the request

switch *Message header* **do**

case *SyncFail Message*

 childVector \leftarrow get target node children;

 req \leftarrow generate request from first child PUSH(request queue, req);

foreach *child in childVector but the first* **do**

 req \leftarrow generate request;

 PREPEND(request queue, req);

case *Elements Message*

return (Message \rightarrow elements) ;

case *FullElements Message*

 remote elements \leftarrow (Message \rightarrow elements);

 local elements \leftarrow GET ELEMENTS(Message \rightarrow pnode);

 local needs, remote needs \leftarrow SYMMETRIC DIFFERENCE(local elements, remote elements);

return (local needs);

otherwise

 /* Unexpected message has arrived, aborting */
 ABORT;

Algorithm 2.1.12: FETCH ELEMENTS

Input: elements: vector of ZZp to recover

Result: recovered elements are stored in DB
recovered hashes are stored in ptree for next recon
peer has completed the synchronization and is ready to
accept new recon attempts or gossip

Data:

if *elements size* < *request chunk size* **then**

 | keys \leftarrow REQUEST_CHUNK(elements);

else

 | split elements into n slices;

 | Init empty vector keys;

for $i \leftarrow 0$ **to** n **do**

 | elements \leftarrow select slice i ;

 | keys \leftarrow keys + REQUEST_CHUNK(elements);

store recovered keys into database;

store recovered hashes into ptree;

the recon protocol. The target URL and associated parameters have to be queried for each needed certificates chunk. The response body will contains the corresponding certificates serialized as string.

- Request type: POST
- Request url: “http://<peer hostname or ip address>:<peer port>/pks/hashquery”
- Request header:
 - Content-Type: sks/hashquery
- Request Post Fields: the only data which needs to be sent is a string composed by a sequence of: 16^1 encoded as **int**, bytes representing a ZZp, for every element.

The response is composed by:

- an **int** which represents the number of keys received
- a series of **string**, each encoding a certificate

¹16 is given by the length of the bytes encoding the finite field used by SKS - 1

Notably, the request, and relative response are performed in clear, without any form of authentication or data validation.

message specifications

Now will follow a detailed message specification explaining how the messages exchanged between peers are structured, transmitted and received.

message receiving and sending

When reading (sending) any of the messages from (to) the another peer via network connection, the first data to be read (written) is the length of the whole message, encoded as **int**, to which follows the full message.

The first byte of the message, is the message header, which, read as a **short unsigned int**, identifies the rest of the message.

Table 2.1: Message header table

int	Message
0	ReconRequestPoly
1	ReconRequestFull
2	Elements
3	FullElements
4	SyncFail
5	Done
6	Flush
7	Error
8	DBRequest
9	DBReply
10	Config

message data types

All the data contained in the described messages needs to be marshaled when sent, and unmarshaled when received. Each data type has its own method of being serialized, which is explained as follows:

short int composed by a single byte, so it does not require a special notation.

int has to be written in big-endian notation, following the convention in the documentation of Internet Protocols which is to express numbers in decimal and to picture data in "big-endian" order [15] [5], so what needs to be done is literally swap the byte order. This has to be done for every integer sent/received.

string intended as bytes array, it is necessary to first send it's size as **int**, and then send the bytes representing the string.

bitstring first send the size in bits as **int**, then send the **string** which correspond to the byte representation of the bitstring, using the appropriate method.

ZZp First send the number of bytes which represent the number as **int**, then a fixed number of bytes has to be sent, equal to the bytes encoding the finite field used by sks-keyserver, which is 17. So, if a number in byte notation does occupy less than 17 bytes, it is simply padded with null bytes (0x00) until the requested length has been reached.

ZZp array First send the dimension of the array as **int** and then send all numbers using the appropriate method for **ZZp**.

message types

ReconRequestPoly Message sent when the conditions to use the linear interpolation algorithm holds

Figure 2.5: struct ReconRequestPoly

```

bitstring prefix
int size
vector(ZZ) samples

```

It contains:

- *prefix*: the **bitstring** which uniquely identifies a node into the prefix tree, for example “000011”
- *size*: number of elements under the node
- *samples*: vector of \bar{m} size composed by **ZZp** values used in the interpolation to recover missing elements

ReconRequestFull Message sent when the conditions to use the linear interpolation algorithm do not holds.

It contains:

- *prefix*: contains the **bitstring** which uniquely identifies a node into the prefix tree, for example “000011”
- *elements*: is the set of **ZZp** numbers stored under the given prefix node

Elements Message sent when a specific set of elements needs to be exchanged, for instance when the reconciliation ends and a certain part of the recovered elements have to be communicated to the other peer

It contains:

- *elements*: set of **ZZp** numbers recovered from the in progress operation

FullElements Message sent whenever knowledge of the full state of a node is needed, for example when the current node under exam is a leaf

It contains:

- *elements*: set of **ZZp** numbers contained in a specific node

SyncFail Used to signal the failure of a specific step of the protocol, for instance when the interpolation fails. This is an expected and recoverable fail.

The message is empty.

Done Used to signal that the reconciliation protocol has ended without errors.

The message is empty.

Flush Used by the server when the client is expected to actually send queued messages.

The message is empty.

Error Used to signal a non recoverable error of the synchronization protocol which needs to be stopped.

It contains:

- *text*: reason for the unexpected failure

DBRequest Unused

It contains:

- *text*: string (unkown uses)

DBReply Unused

It contains:

- *text*: string (unkown uses)

Config holds the configuration parameters exchanged at the beginning of the recon protocol

```
string version
int http_port
int bitquantum
int mbar
string filters
hashmap<string, string> other
```

Figure 2.6: struct Config

- *version*: relative to sks-keyserver. The current version is “1.1.6”, having a string different from this will result in a reject from the other peer.
- *bitquantum*: determines the arity of the prefix tree. The current value is 2.

- *mbar*: determines the number of points used in the linear interpolation algorithm.
- *filters*: the filters applied to certificates. The current and only defined filters are “yminsky.dedup, yminsky.merge”
- *other*: is a, currently unused, key-value structure (string to string) which could be used to exchange more data before the actual recon.

Unnamed messages

Sometimes stand-alone messages are sent, which do not need the message header short int type, or the length written ahead of message. In Peaks, some of this messages have a convenience structure, even if they are marshaled in a different way. This messages are *Config_ok* and *Config_error*, used to acknowledge the config in the first step of reconciliation. The data is just send out of marshaling because it is supposed to be received in that particular moment of the protocol.

In peaks this messages are:

Config ok

It is composed by a single string containing the text “passed”

Config mismatch

It contains two strings, one with the text “failed”, and the other with the reason for the rejection, for instance “mismatched bitquantum”, or “different version”, which typically ends in the other peer log file.

2.1.3 Prefix tree

The **Prefix Tree** (**ptree**) is the n-arity tree used to store MD5 hashsum of certificates as prefixes, to achieve a computationally faster recon process. Its health state is crucial to the keyserver, because a malformed or unbalanced tree could worsen the performance of the elements lookup and slow down the whole process, and is fundamental that the informations stored in the tree are consistent with the one in the database.

The tree supports two kind of nodes: *internal* nodes and *leaf* nodes. Each node in a partition tree corresponds to a different partition *P* of the

stored hashsums set S . When a bitstring is added to the set, the ptree must be updated accordingly. It is necessary to descend the tree, adding an element to the checksums of all internal nodes on the path towards the leaf node, where the element is added. If the number of elements stored in a leaf node overcomes the maximum threshold, it must be converted into an internal node, and children created for it, to inherit the elements of the node. Similarly, if the number of elements in a leaf node is brought below the minimum threshold, it must be converted to an internal node, and its children joined.

As noted during Chapter 1 the ptree supports the following operations:

- adding elements with complexity $\Theta(\bar{m}(\log |S| - \log \bar{m}))$
- deleting elements with the same complexity $\Theta(\bar{m}(\log |S| - \log \bar{m}))$
- search nodes

Actually, the delete operation is never used: even if an element is deleted, the synchronization protocol would reinsert the element in the exact same place as before, as if the deletion never occurred.

Before starting the recon process a ptree must be build from the certificate database. For each certificate an MD5 hash is calculated and stored into the tree.

In peaks the *ptree* data structure only contains a reference to the root node.

```
string node_key
ZZp vector node_svalues
bool is_leaf
int num_elements
ZZp vector node_elements
```

Figure 2.7: struct Pnode

A **Prefix Node (pnode)**, is a node of the ptree. It contains:

- node key: the identifier of the node in the database. It is a bitstring composed by the bitstring of the parent node concatenated with the

child index in binary, i.e. “001101”. This value is embedded into the *ReconRequestPoly* and *ReconRequestFull*, such that the remote peer is able to recover the same node or a parent node from its own ptree.

- node svalues: the checksums of the node, which is updated when an element is inserted/deleted. This value is embedded into a *ReconRequestPoly*, the request which start the linear interpolation algorithm.
- is leaf: a flag used to determine if the node is a leaf node. Only leaf nodes actively store the elements to sync.
- num elements: number of elements under the current node. If the node is a leaf the parameters represents the length of the element array stored. If the node is an intermediate one, this value holds the sum of the num elements value of its children
- node elements: vector holding the stored elements of a leaf node, the hashsums in ZZp format. It is empty for intermediate nodes

The ptree is created empty and is immediately populated with a root node, which is a *pnode* with an empty key.

NEW CHILD is the operation in which new nodes are created. It employ the parent node reference, and the index of the new child. To compose the key of the child, at the parent node key is appended the index in binary. A vector of ZZp is created with values setted to 1, to initialize the svalues of the new node. When the tree is empty it generate a root node with no parent node and empty key.

Elements inserted in ptree are MD5 hashes of certificates, so hexadecimal values, but to perform the insert operation the corresponding bitstring and numerical value are used, and ultimately the numerical value is stored in the ptree database. When an element is inserted in the tree:

- the element is used to generate a *marray*, the array of ZZp numbers generated starting from a set of pre-calculated interpolation points (0, 1, -1, 2, -2).
- the element and marray are used in the INSERT operation, which starts from the root node.

Algorithm 2.1.13: NEW CHILD

Input: parent node: pnode parent of the new child (usually is a reference to the calling node)
child index: integer index of the current node, along with the prefix of the parent node will generate the child node key

Output: child node: new pnode

Data: child key: bitstring key of the child node, initialized from the parent node key, then the index of the node is appended
svalues: vector of ZZp, initialized to 1

child node \leftarrow generate new leaf empty Pnode **if** *parent node not null*
then

child key \leftarrow get node key from parent;
append to child key bitquantum bits equal to the child index in binary;
set node key(child node, child key);
/* initialize svalues */
svalues \leftarrow init vector of size $\bar{m} + 1$;
for $i \leftarrow 0$ **to** \bar{m} **do**
| svalues[i] \leftarrow 1;
set node svalues(svalues);
set node leaf flag to True;

return *child node*;

- If the root node is a leaf the new element is added
- Otherwise the element is passed to one of the child nodes which perform the same check until a free node is found
- In both cases the checksum of every node in the path to the leaf node is updated using the content of the *marray*

After the addition of the element to a node, if the number of elements hosted is greater than the split threshold, the node is subject to the SPLIT operation. The SPLIT creates $2^{bitquantum}$ children, which share the prefix of the parent node, and the elements stored in the parent node are equally divided among them.

Algorithm 2.1.14: ADD ELEMENT ARRAY

Input: element: element for which the marray is generated
Output: marray: vector of ZZp used to update the partitions
checksum
Data: points: array of ZZp representing the interpolation points
Init marray with $\bar{m} + 1$ cells;
for $i \leftarrow 0$ **to** $\bar{m} + 1$ **do**
| marray[i] \leftarrow points[i] - element;
return (marray);

The *marray* is a vector generated from the new element to insert, or delete, used to update the checksums (*svalues*) of all nodes in the path from the root, to the node in which the new element will be placed, or deleted.

The insert operation performed on the tree is recursive. The insert start from the root node, if it is not a leaf node, the insert proceed to the next child, selecting the index on the basis of the bitstring representing the element to insert, with the NEXT operation. If the current node is a leaf and its elements are less than the split threshold the element can be inserted, so the *svalues* (representing the partition) of the node are updated with the content of *marray*, so to match the new partition. Instead, if the number of elements overcame the threshold, the SPLIT operation is called on the node, creating children, and dividing the elements of the node between them. The number of elements, stored separately in the node will keep the count of elements under the node always updated, so that the number of elements of a given partition is known.

Algorithm 2.1.15: INSERT

Input: element: ZZp element to insert

marray: vector of ZZp needed to update the node checksum
for the linear interpolation

Output:

Result: elements is inserted, and eventually, node is splitted

Data: current node: starting from the root, hold the pnode in which
the algorithm is trying to insert the element

depth: integer which hold the current depth reached in the
tree while searching an available pnode

depth \leftarrow 0;

while *element is not placed* **do**

 UPDATE SVALUES(current node, marray);

 add 1 to current node elements number;

if *current node is leaf* **then**

 /* only leaf node store elements */

if *size of elements stored in current node > split threshold*

then SPLIT(current node, depth) **else**

 add element into current node elements array;

return ;

else

 child index \leftarrow NEXT(current node, element, depth);

 current node \leftarrow GET CHILD(current node, child index);

 depth \leftarrow depth + 1;

Algorithm 2.1.16: SPLIT

Input: node: pnode on which perform the split
depth: current depth of the tree

Output:

Result: children node are created and successfully initialized

Data: child num: number of child to create
idx: integer index of the child node
children array: temporary vector of pnodes
child: temporary pnode holding the new child

child num $\leftarrow 2^{bitquantum}$;

init children array of size child num;

for $i \leftarrow 0$ **to** child num **do**

 child \leftarrow NEW CHILD(node, i);
 add child to children array;

foreach element in node elements **do**

 /* dividing parent node elements into children */
 ;
 idx \leftarrow NEXT(node, element, depth);
 child \leftarrow children array[idx];
 marray \leftarrow ADD ELEMENT ARRAY(element);
 INSERT(child, element, marray);

set node elements empty;

set node leaf flag to false;

From a SPLIT will be created always the same number of children equal to $2^{bitquantum}$. Every child key is composed by the parent node bitstring, to which is appended the bitstring representing the child index.

Algorithm 2.1.17: NEXT

Input: parent node: node which call the operation
element: element for which the insert is begin carried
target depth: current depth reached by the insert
Output: key: index of the child in which attempt insert
Data: current depth: integer which keeps track of the current depth in the tree
element bits: bitstring representing element to insert
bits: temporary bitstring holding bitquantum extracted bits

```

current depth  $\leftarrow$  0;
element bits  $\leftarrow$  TO BITSTRING(element);
next node  $\leftarrow$  GET ROOT();
while current depth  $\leq$  target depth do
    /* extraction is performed from the least to the most
       significant bit */
    bits  $\leftarrow$  extract next bitquantum bits from element bits;
    next node  $\leftarrow$  GET CHILD(next node, bits);
    current depth  $\leftarrow$  current depth + 1;
return next node;

```

The NEXT algorithm returns the index of the child to select, so GET CHILD is needed to actually select the correct child. It is basically a tree traversal, but since the ptree is a $2^{bitquantum}$ -arity tree, *bitquantum* bits needs to be fetched from the bitstring to know what direction has to be taken.

Algorithm 2.1.18: GET CHILD

Input: parent node: pnode of the node requesting child
index: index of the child to retrieve
Output: child node with corresponding index
Data:
child key \leftarrow parent node key;
bitstring index \leftarrow TO BITSTRING(index);
child key \leftarrow APPEND(child key, bitstring index);
child node \leftarrow search ptree DB for child key;
return *child node*;

Similarly to the addition it is possible to delete elements from the ptree, and the operations performed are symmetric: a marray to revert the checksum values will be generated and the changes will be propagated down the branch in which the element resides, eventually causing a join of the hosting node if, after the deletion, its number of elements falls below the join threshold.

Algorithm 2.1.19: DELETE ELEMENT ARRAY

Input: element: element for which generate the deletion marray
Output: marray: array of ZZp used to update the partitions
checksum
Data: points: array of ZZp representing the interpolation points
Init marray with $\bar{m} + 1$ cells;
for $i \leftarrow 0$ **to** $\bar{m} + 1$ **do**
 /* inv is the inverse operation applied on ZZp numbers
 */
 marray[i] \leftarrow inv(points[i] - element);
return (marray);

The JOIN is the opposite of the SPLIT, nodes with a number of elements less than the join threshold are joined with the children into a single node, which returns to a leaf.

When, during recon, client and server exchange messages, they do not know the exact ptree structure of each other, but they send requests which assume the existence of a specific node, i.e. *ReconRequestPoly* contains information gathered from the local tree. Such request has to be matched with the information stored on the receiver side, so the primitive GET NODE is used to search the node starting from its known prefix. If the node is not found, a search begins, recursively trying to fetch the parent node, up to the root.

In peaks, for durability, the ptree is stored in a MySQL database. A single table is enough to store the needed information since what needs to be stored are the attributes of a single node: key, elements, svalues, number of elements, leaf flag.

Algorithm 2.1.20: DELETE

Input: current node: pnode on which perform the delete of the element
element: ZZp element to delete
marray: vector of ZZp used to revert the checksum of the partition

Output:

Result: element get deleted from node elements, and eventually node is joined

Data: depth: int used to known the current depth of the operation
child index: index of the new child, calculated when it's necessary to select a new node

depth \leftarrow 0;

while *element is not removed* **do**

UPDATE SVALUES(current node);
add 1 to current node elements number;
if *current node is leaf* **then**

/* only leaf node store elements */
delete element from the node;
return;

else

if *size of elements stored in current node < join threshold*
then JOIN(depth)

else

child index \leftarrow NEXT(current node, element, depth);
current node \leftarrow GET CHILD(current node, child index);
depth \leftarrow depth + 1;

Algorithm 2.1.21: JOIN

Input: node: pnode on which perform join

Output:

Result: node and child nodes are joined

children vector \leftarrow recover child nodes;

foreach *child in children vector* **do**

add child elements to parent node;
delete child;

set node to leaf;

Algorithm 2.1.22: GET NODE

Input: node key: key of the requested node
Output: node: pnode with the nearest prefix to the requested one
found \leftarrow False;
repeat
 node \leftarrow try to get node with node key equal to key from the DB;
 if *node is found* **then**
 found \leftarrow True;
 else
 /* remove the last 2 bits from the bitstring */
 resize node key to $size(nodekey) - 2$;
until *not found and size(node key) > 0*;
return node

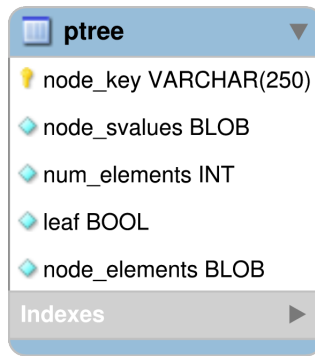


Figure 2.8: ptree table

- node key can be stored as a string, directly storing the sequence of 1 and 0 composing the bitstring. This would be the “primary key”, since a node is searched by key, and to a key is associated exactly one node.
- node elements is a string, a marshaled view of the vector object, since this value can grow quite a bit a field capable to store an arbitrarily large number of character is needed
- node svalues is same as node elements, but its size is more constrained
- number of elements is an integer
- leaf flag is a boolean

2.1.4 Components

The *ptree* stores the hash of the certificates, such that the recon protocol is able to sync hashes first, and then receives the missing certificates. Those hashes are calculated separately from the recon daemon and the *ptree* by a component capable of reading, parsing and storing OpenPGP certificates. In Peaks this component is the **Dump import**. It is responsible for reading the certificates from PGP files, one at a time, parsing and storing it into the database, for initialization of the database and, in general, when importing new certificates. In the process the hashsum of the certificate, used in recon is created: it is the MD5 hash of the concatenation of each packet, sorted by packet tag, and at equal tag by content.

The *Dump import* uses an openPGP parsing library in C++ [7], which has been modified to account the issues encountered when dealing with certificates, including:

- Wrong user attribute subpacket header: the standard imposes that the length of the user attribute subpacket would be encoded with 1, 2 or 5 bytes depending on its length. We found out that in some cases the length would be encoded using 5 bytes, i.e. leaving the middle one to 0.
- Wrong signature algorithm: the algorithms that could be used to generate signatures are known, but unfortunately keys with unknown algorithms have been found, and the only possible way to parse them was to follow the state of the art approach to be compliant, which is read the packet until end of string
- Wrong packet size: the library was modified to read more bytes than the one effectively advertised by the packet header, and to output the wrong size, for compliance.
- Wrong packet format: modifications were needed to tackle packets in which were present null packets, or fewer packets instead of the expected number. The solution applied was to force reading the packet content, regardless of the standard parameter sequence.
- Error signaling for Unknown signature version: removed, because some

certificate have in the Signature packet a non-standard signature version, which, as a side problem, have no way to be parsed, so the solution applied was to save all the packets without trying to inspect further

- Wrong user attribute packet encoding: some implementation were found using the user attribute packet for more than the designed purpose (storing images), so the content of the packet has to be analyzed even if not compliant with the standard

The *recon daemon* uses the NTL library [16] to manage the integers with modulus, used for the linear interpolation. NTL is a high-performance, portable C++ library providing data structures and algorithms for arbitrary length integers; for vectors, matrices, and polynomials over the integers and over finite fields; and for arbitrary precision floating point arithmetic. In particular NTL has been used for its arbitrary length integer arithmetic and polynomial arithmetic over the integers and finite fields including basic arithmetic, polynomial factorization, irreducibility testing, computation of minimal polynomials.

Chapter 3

Experimental Evaluation

3.1 Experimental Results

In this chapter there will be a collection of statistics, metrics, and gathered data from the analysis of the current snapshot of the keyserver infrastructure.

Will be analyzed the current status of: Ptree sanity, Certificates sanity, Security state, Liveness state.

3.1.1 Ptree sanity

The ptree is basically a tree in which the arity is decided by the **bitquantum** parameter. The structure of the tree has to be for two servers to synchronize properly. A tree is a widely used structure in computing because they allow faster insert, search and retrieval compared to linked lists as long as the tree is balanced.

If the tree is unbalanced (up to the point that become totally flat) part of its effectiveness is lost.

This analysis try to point out the current state of the ptree, by looking at its height, balance, leaf distribution and elements per leaf node so that we know if it is effective as it is designed to be.

Dimensions of tree and nodes The ptree is conventionally created with an arity of 4. It currently holds 349473 nodes and have an height (depth) of 10.

Figure 3.1: Node number per level

level	nodes
0	1
1	4
2	16
3	64
4	256
5	1024
6	4096
7	16384
8	65536
9	262092

Table 3.1: ptree statistics

Nodes	349473
Leaf nodes	262105
Maximum number of element per leaf node	51
Minimum number of elements per leaf node	4
Mean number of elements per leaf node	20
Variance of the number of elements per leaf node	20.68

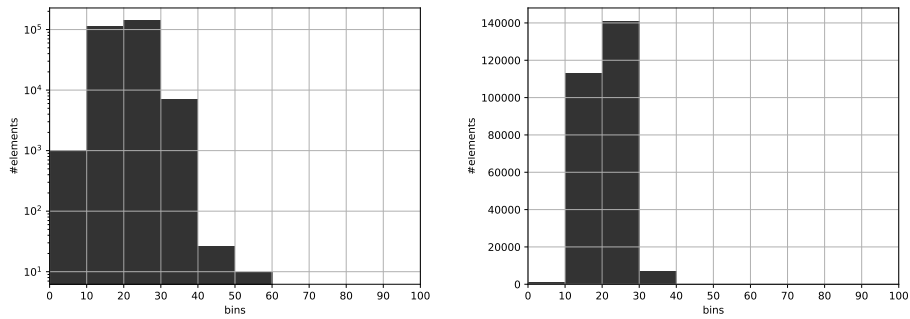


Figure 3.2: element per node plot in log and linear scale

Balance of the tree A tree is balanced only if for every intermediate node of the tree the following properties hold:

- The left and right subtree heights differ by at most one.
- The left subtree is balanced
- The right subtree is balanced

Some tree structure implementation offer auto-balancing properties, but this is not the case for the ptree implementation, which has been designed with the idea that hashsum of certificates would be equally distributed, since the output string of an hash function can be considered “casual”. So with a clever implementation that would divide hashsums between nodes based on the corresponding numerical value, the tree should be always balanced.

Actually the longest branch in the tree has depth 9 and the shortest has depth 9, so the tree is well balanced.

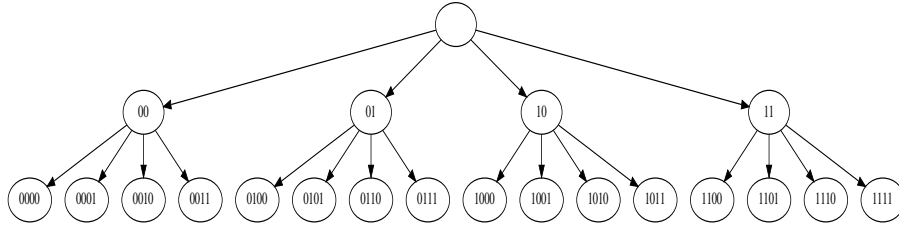


Figure 3.3: first levels of the ptree

partial representation of the tree

3.1.2 certificate sanity

We analyzed the length of the certificates and in particular of user attribute, to understand what type of attributes users are including in their certificate, and to scan for uncommon user attributes dimensions which would cause issue to the keyserver network. Several system administrator have point out issues during the recon process, which may also be caused by large keys being exchanged [17].

The following chart represent the length of the certificates in our database, divided in bins. Up to 10 Kb bins are small (1Kb), then gets progressively larger.

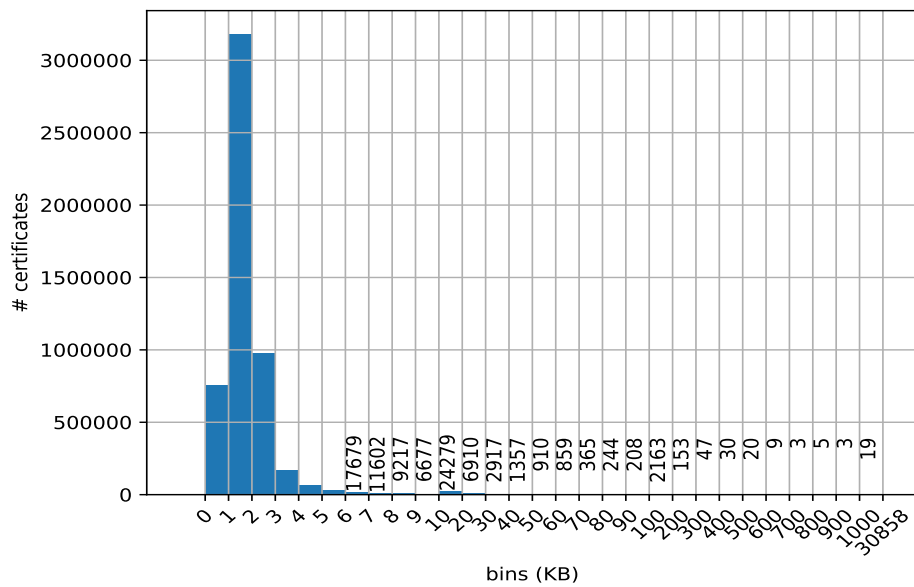


Figure 3.4: certificates without user attributes divided by total length

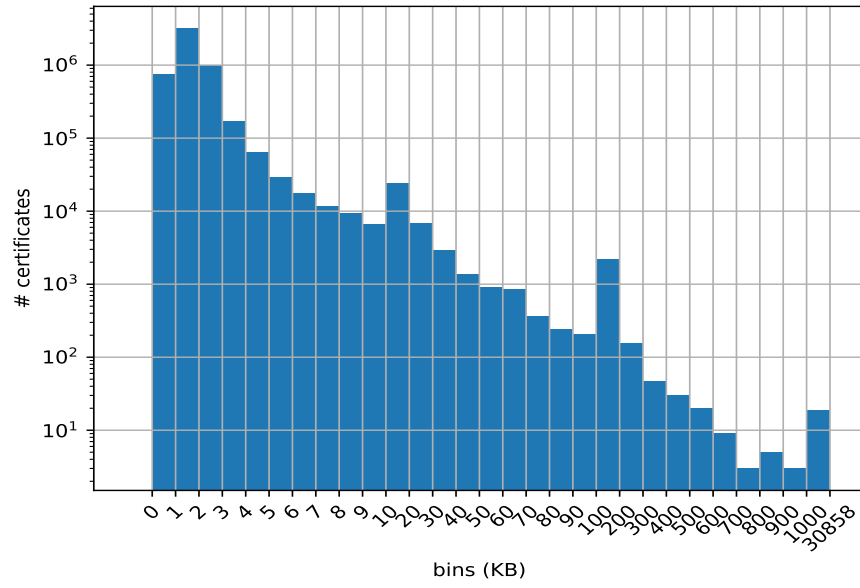


Figure 3.5: certificates without user attributes divided by total length (log scale)

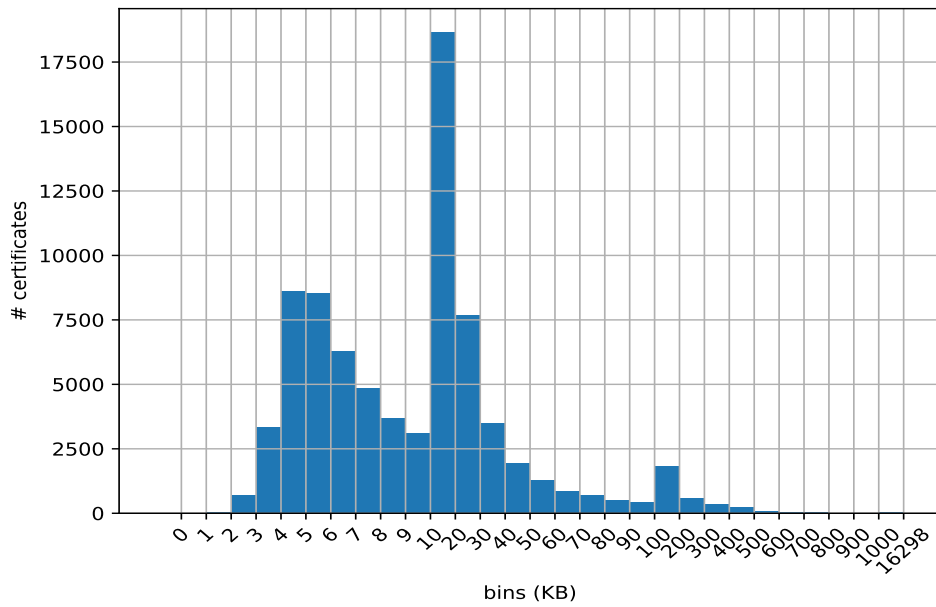


Figure 3.6: certificates with user attributes divided by total length

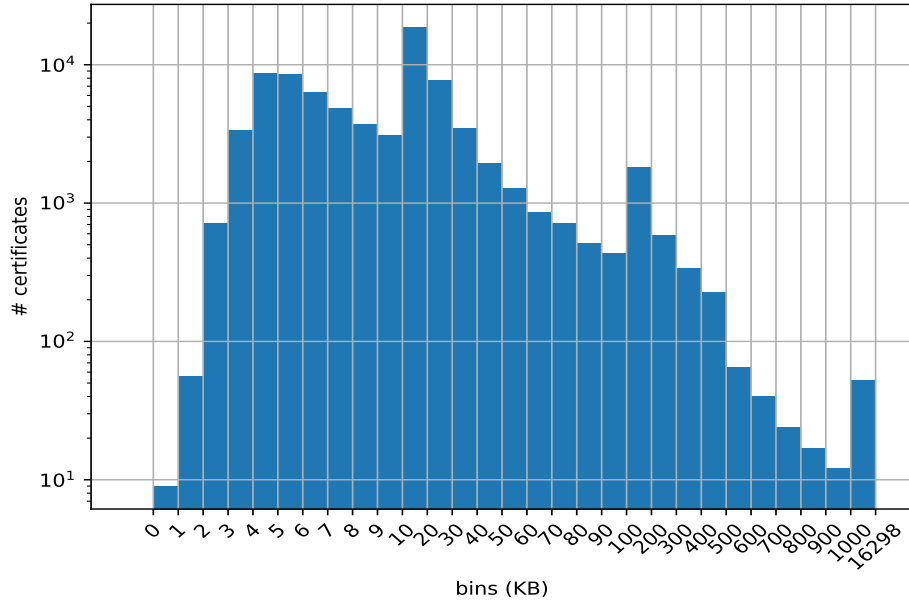


Figure 3.7: certificates with user attributes divided by total length (log scale)

Numbers are placed on the smaller part to highlight that, even if in a lower percentage, there are certificate of every possible size. It is easy to see that the concentration of certificates is higher in the lower bins, most certificates weight less than 3KB and this is a good news.

There few big certificates, mostly because of the inclusion of images as user attribute. Some of this certificates could reach the dimension of some Megabytes, luckily they are rare.

In particular about image user attributes, we need to wonder if it is really necessary for a keyserver to include such information of a public key. After all, images attached to a key cannot be validated, so even if it could be possible to know for sure that the subject of the picture is a real person, it would be impossible to prove the binding between the picture and the person in the picture and the identity.

Now a close up of the size of user attributes

Table 3.2: user attribute popularity

Total Certificates	5349825
Certificates with user attributes (photo)	82308
Certificates with other user attributes	1134

As per RFC4880, image user attribute, identified by subpacket tag “1” is the only subpacket allowed, but we found some user attribute identified by the tag “0”, which will be represented as “Other user attributes”. Furthermore we will not inspect the actual content of an image subpacket the validity of the data as image.

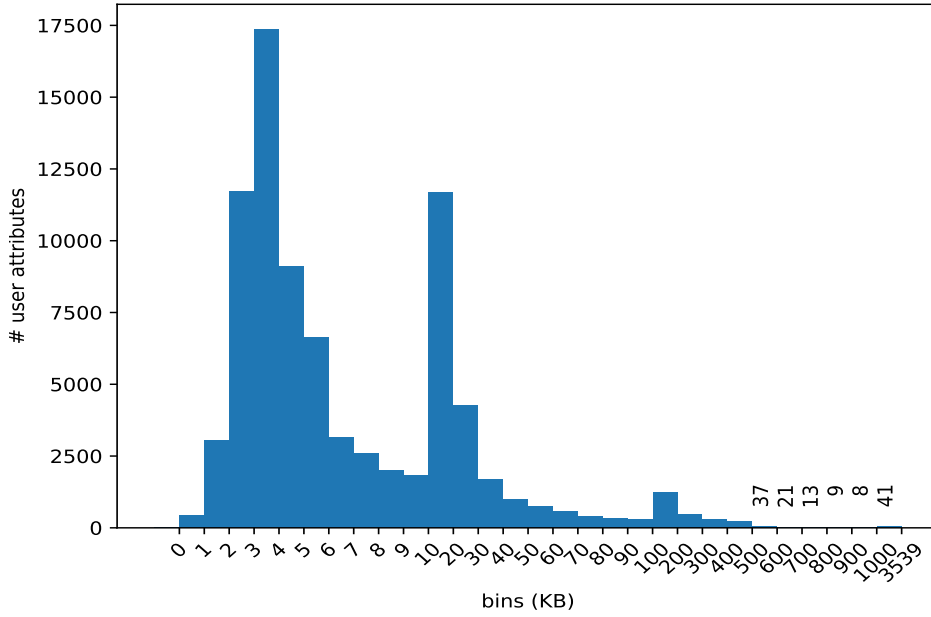


Figure 3.8: user attributes photographic divided by length

It is confirming what seen before: the majority of user attributes lengths are grouped into smaller bins. Only a few user attributes have image which surpass in size the Megabyte.

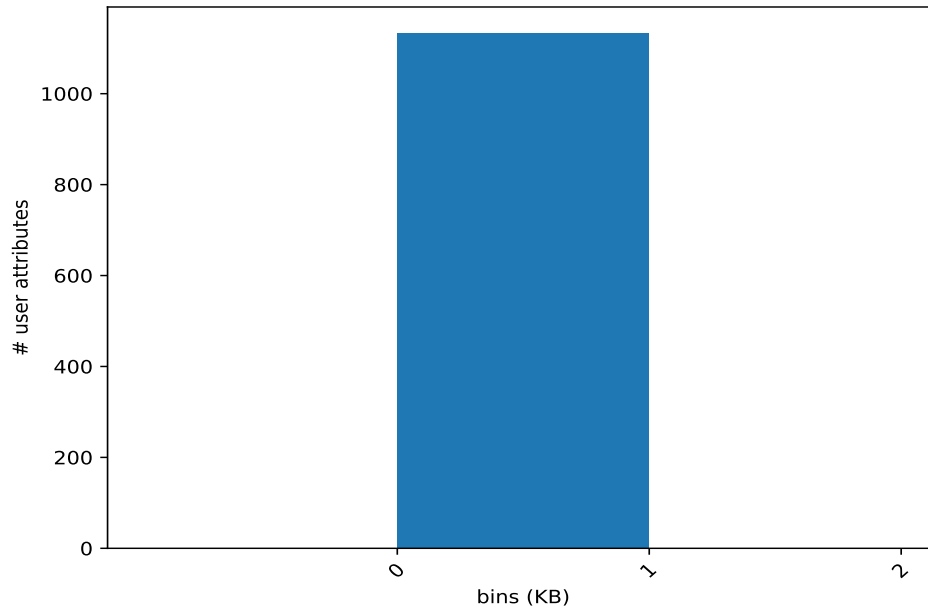


Figure 3.9: user attributes of other type divided by length

Fortunately “unknown” user attributes are limited in size to 1KB

3.1.3 Pubkey security

In this section we analyze the public keys loaded in peaks database. The grand total of public keys is 10446761, divided in:

- RSA public keys: 5049129
- Elgamal public keys: 2695549
- DSA public keys: 2687985
- Elliptic Curve public keys: 14098

The following chart will provide a 20 year long view of the evolution of the algorithms used for the generation of public keys.

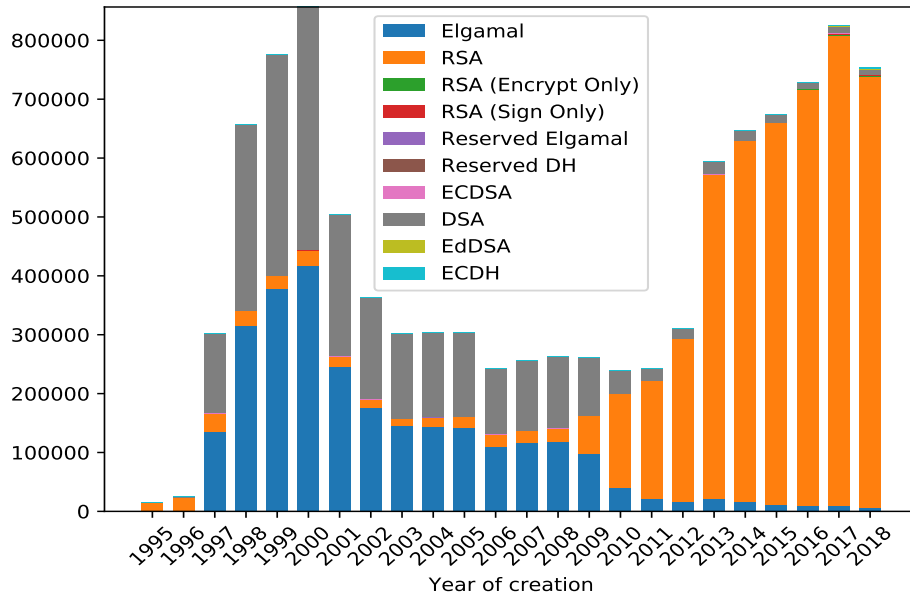


Figure 3.10: distribution of key creation per year

DSA and Elgamal were more prevalent before 2003-04 when RSA keys started taking place quickly, maybe due to RSA being adopted as the default choice by GPG, due to being, at the time, the way to overcome old DSA 1024-bit limit, and 160-bit signature limit, and by SSH key generator utility [8]. While Elliptic Curve being considered the future of public key algorithms is still considerable in its infancy because its usages are rare: less than 1% of keys are generated by EC.

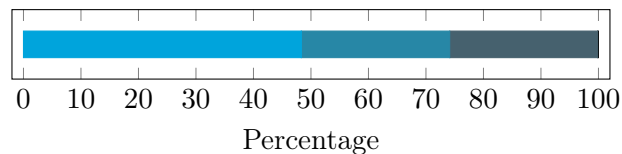


Figure 3.11: Signatures partitioning

- RSA;
- Elgamal;
- DSA;
- Elliptic Curve;

Each algorithms has its own section in which more in depth view are analyzed and discussed.

RSA

RSA public keys are the majority of the set. They account for almost 50% of public keys.

As of today standards keys created with a bits size of less than 2048 are not considered secure [2]. Until 1980s, key of 512 bits were recommended as sufficient security standard. Since many embedded devices did not provide enough processing power to have an efficient generation of larger keys even 330 bits keys were used in the early 1980s. Academic successes in breaking those keys, led to a change in recommended key size, which would increase to 1024 during the 1990s.

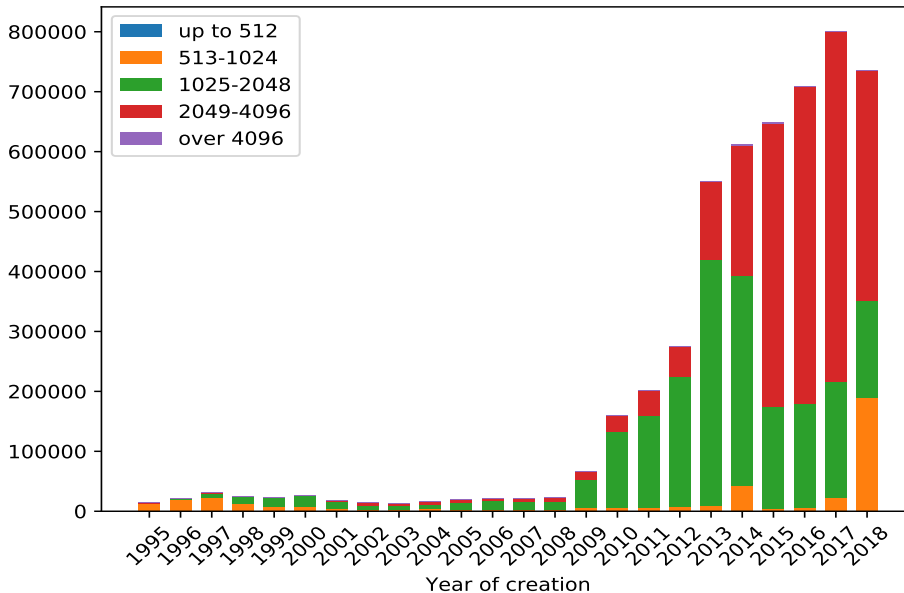


Figure 3.12: bit length of RSA public key n parameter

The chart represent the numbers of keys with the corresponding n parameter found. It is easy to see that since 2013 there was a shift to adopt more robust key sizes, and it is a positive news. However during the very last year is possible to see also a spike in the creation of weak keys.

Elgamal

Elgamal public keys are almost the 26% of the set.

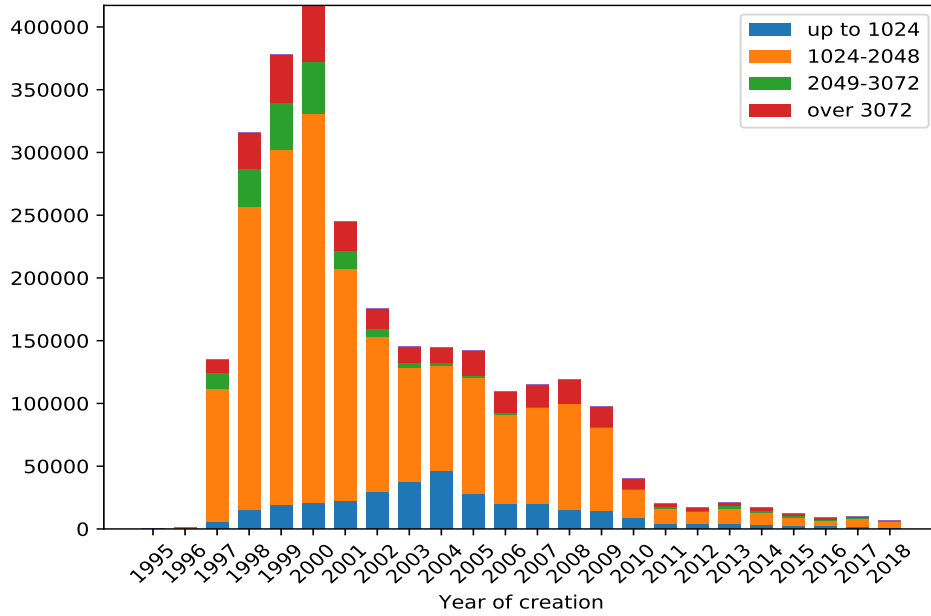


Figure 3.13: bit length of Elgamal public key p parameter

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 chart show that the trend in using Elgamal is declining.

DSA

DSA public keys are almost the 25% of the set.

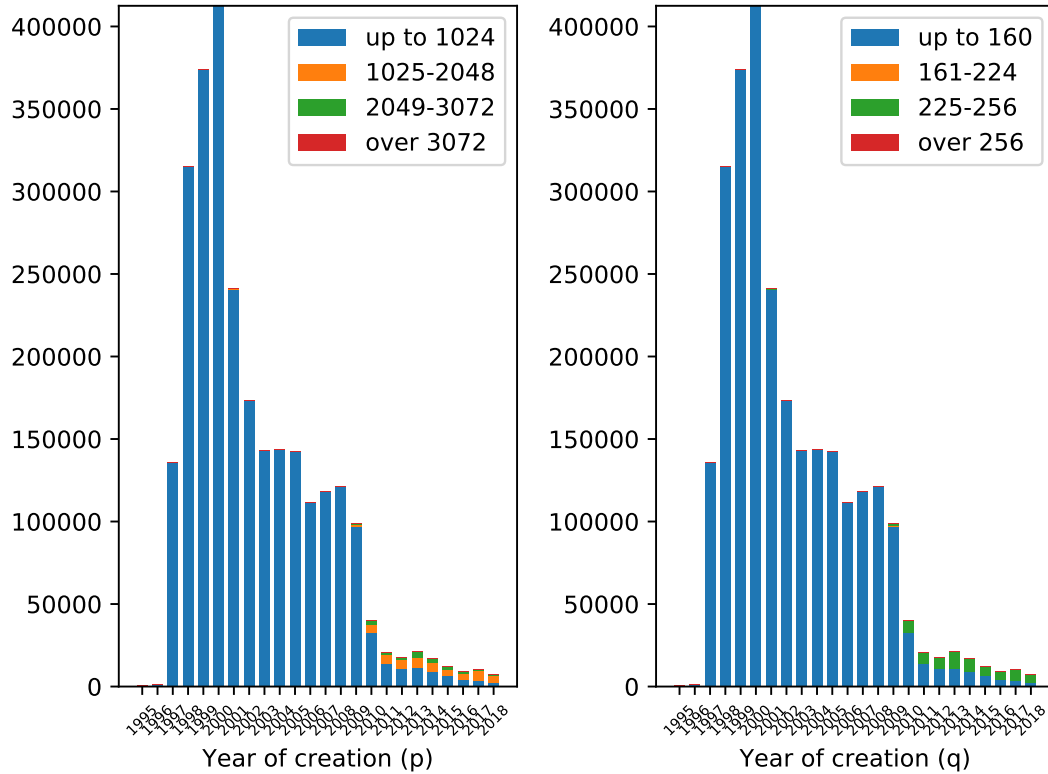


Figure 3.14: bit length of DSA public key p and q parameters

This graph is similar to the Elgamal one for the same reason. In the chart is presented a view of the two parameters of DSA, q and p which present the same trend. p is considered no longer secure with a length of 1024 bits or less, 2048 bits are needed to achieve a recommended level of security.

Elliptic curve

Elliptic Curve keys are the least used, only 0.13% of the set.

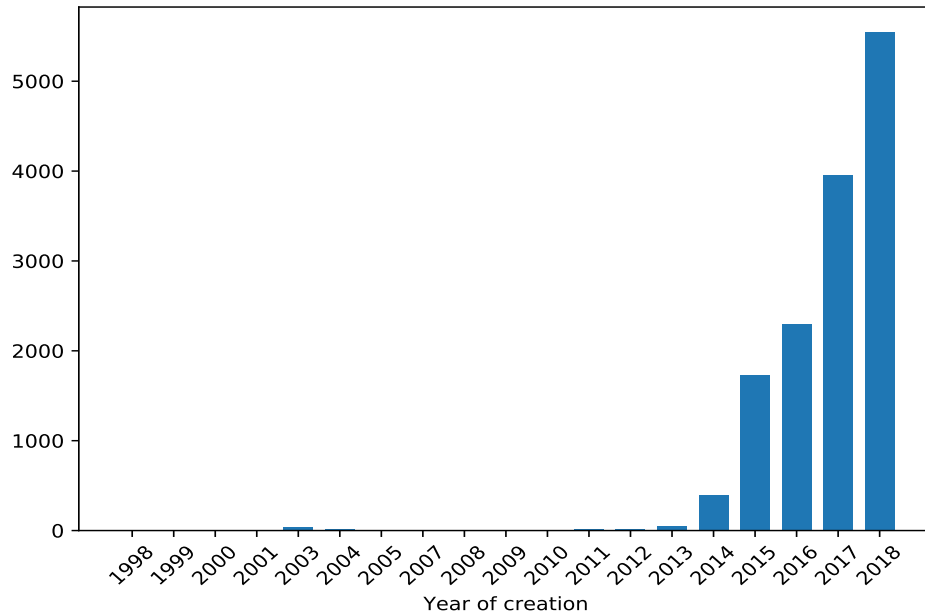


Figure 3.15: EC public keys created during years

The elliptic curve algorithms in OpenPGP use some fixed curves, thus there is no need for plotting and check their size.

3.1.4 certificate vulnerability

Peaks is capable of scanning in a background job the whole set of public keys loaded to detect the common known vulnerabilities. This is an improvement over the state of the art, which do not provide any feedback over inserted key material.

The searched vulnerabilities are:

- RSA Vulnerabilities:
 - Small key size:
 - Modulus is prime
 - Modulus has common factor: the key has a common factor with another key, which greatly simplify the recovery of the other factor

of both keys.

- Modulus has low prime factor
- Exponent too small
- Roca vulnerability found
- Missing parameters: the key miss the core parameter to be evaluated, probably caused by a broken key.

- Elgamal:

- P is not prime
- Q is not prime
- G equals to 1
- G wrong subgroup
- $p - 1$ is not a multiple of q
- Missing parameters

- DSA Vulnerabilities are the same of Elgamal

- Elliptic Curve Vulnerabilities:

- algorithm doesn't work on this curve
- pubkey doesn't belong to the curve
- Missing parameters

Now will be presented the results from the analysis.

Each algorithm results will be in its own section. The first chart of each section will be a comparison between the numbers of healthy and unhealthy keys, and then, an in depth view of the unhealthy keys will be provided.

RSA

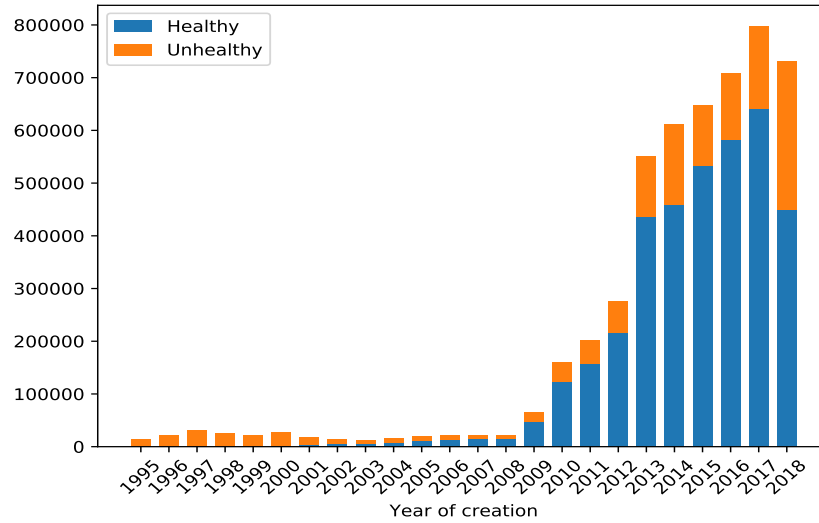


Figure 3.16: Healthy and Unhealthy pubkey number comparison

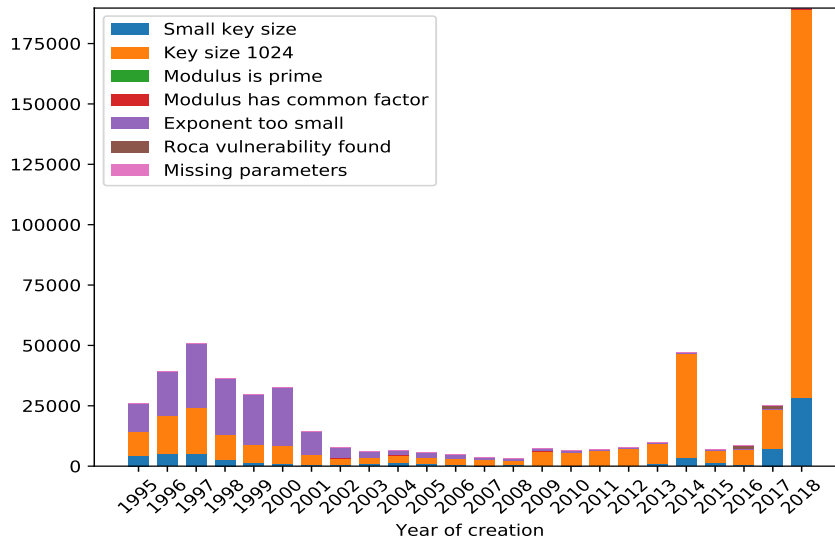


Figure 3.17: In depth analysis of the vulnerabilities found per year

The number of vulnerable RSA keys increased during years, along with the popularity of the algorithm. Even in the current year ten of thousands of weak RSA keys are created and uploaded to a keyserver.

Elgamal

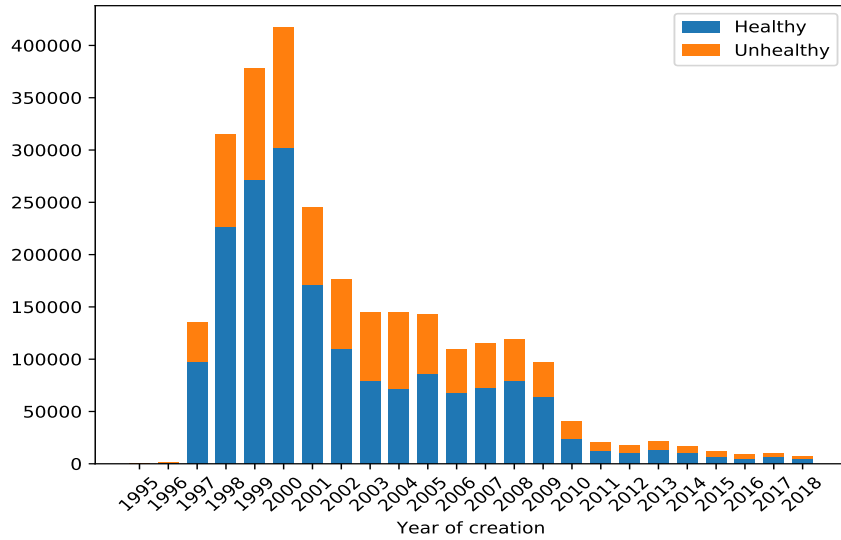


Figure 3.18: Healthy and Unhealthy pubkey number comparison

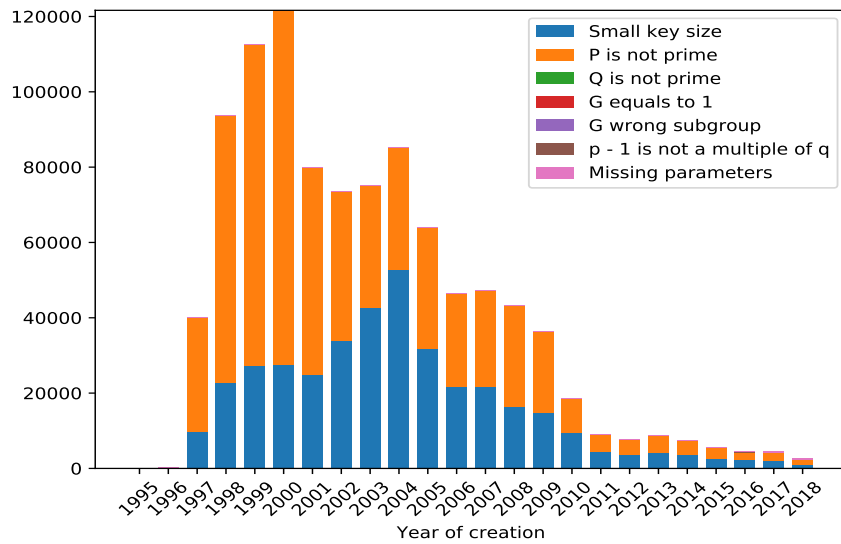


Figure 3.19: In depth analysis of the vulnerabilities found per year

With Elgamal algorithm the main issues found by the analyzer are the dimension of the p (<2048) parameter, which get detected also as not prime.

DSA

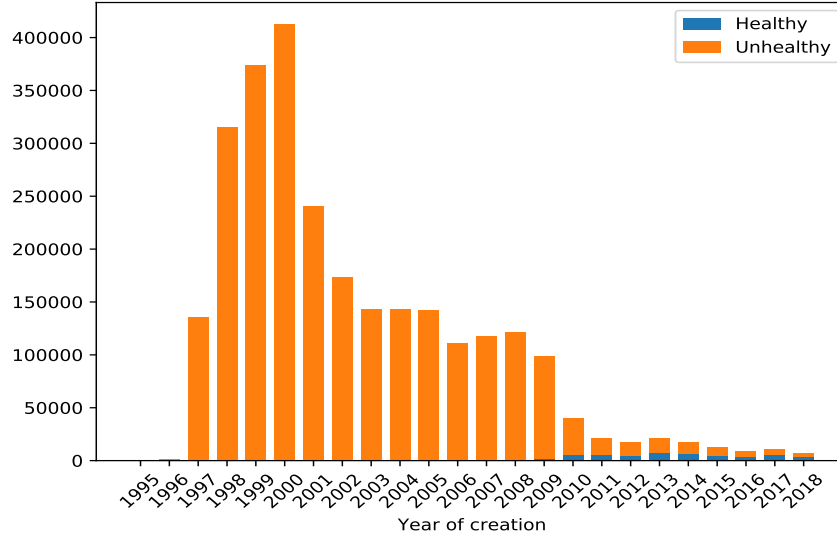


Figure 3.20: Healthy and Unhealthy pubkey number comparison

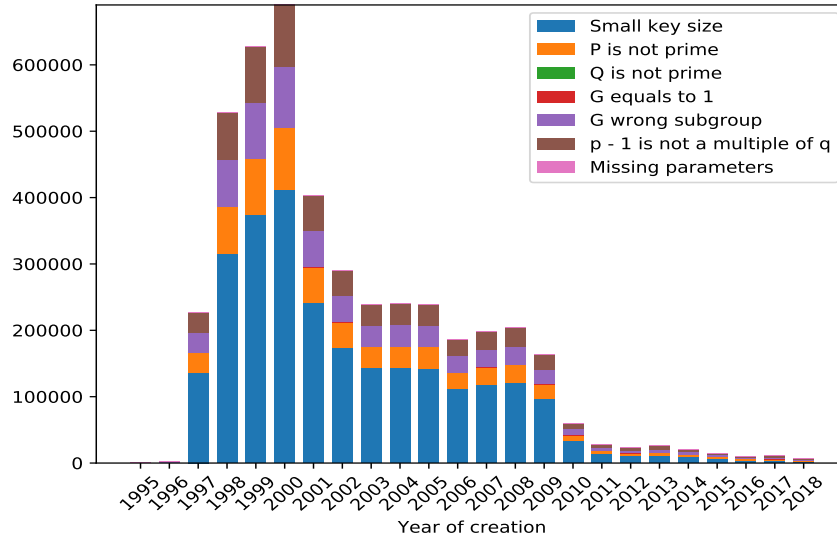


Figure 3.21: In depth analysis of the vulnerabilities found per year

DSA keys are considered vulnerable due to the too small dimension of the length of p (<2048) and q (<224) parameters.

3.1.5 Signatures security

Now the same analysis will be outlined also for signatures data.

Table 3.3: signatures statistics

total signatures	20201531
signature	11605785
signatures valid	9483504
signatures expired	2122281
signatures revocation	22822
self signatures	8595746
self signatures expired	1355
self signatures revocations	435207
self signatures valid	8594391

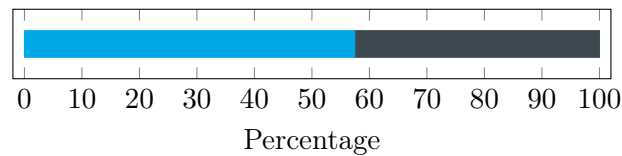


Figure 3.22: Signatures partitioning

■ signatures
■ self signatures

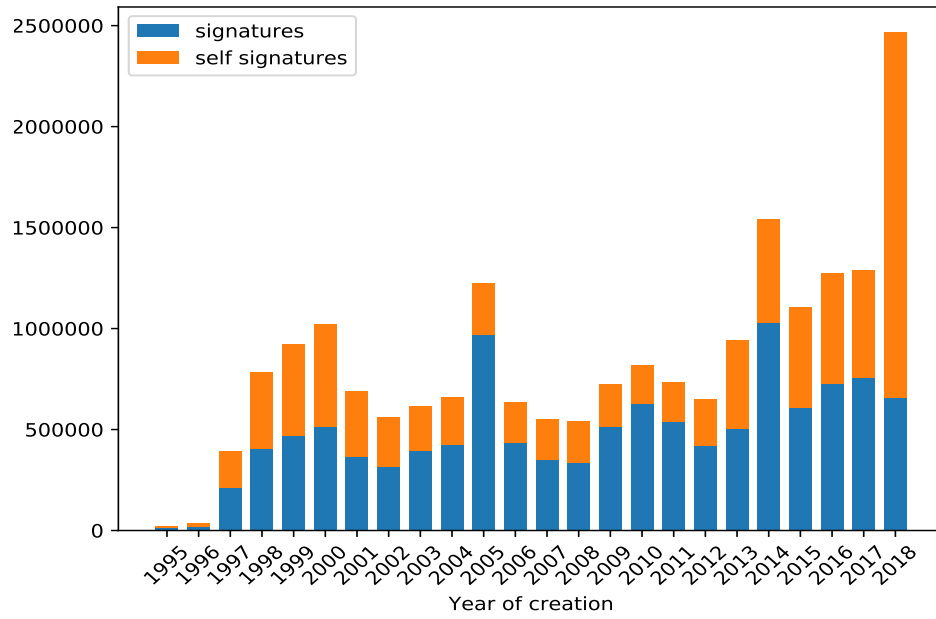


Figure 3.23: Signatures/self-signatures creation per year

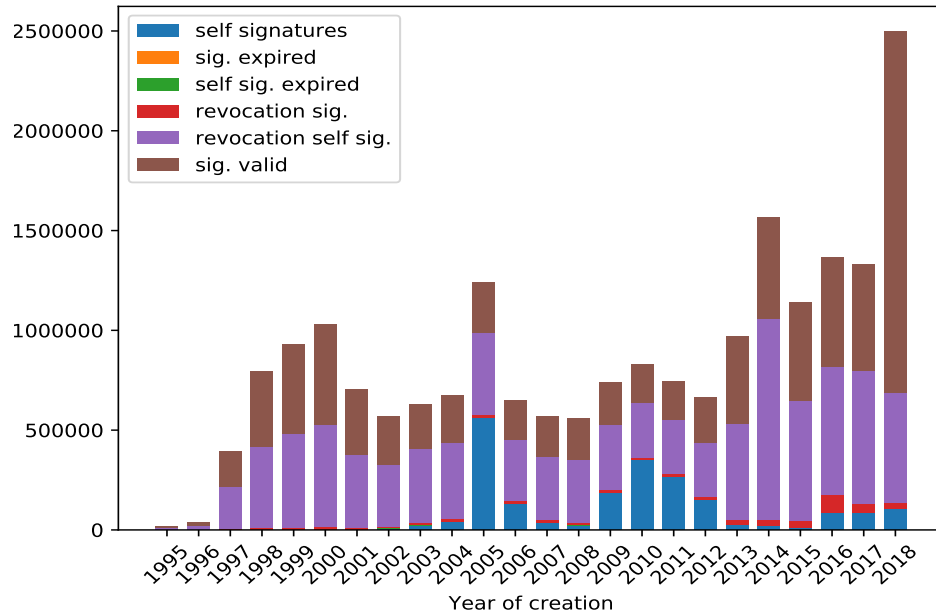


Figure 3.24: Signatures data per year

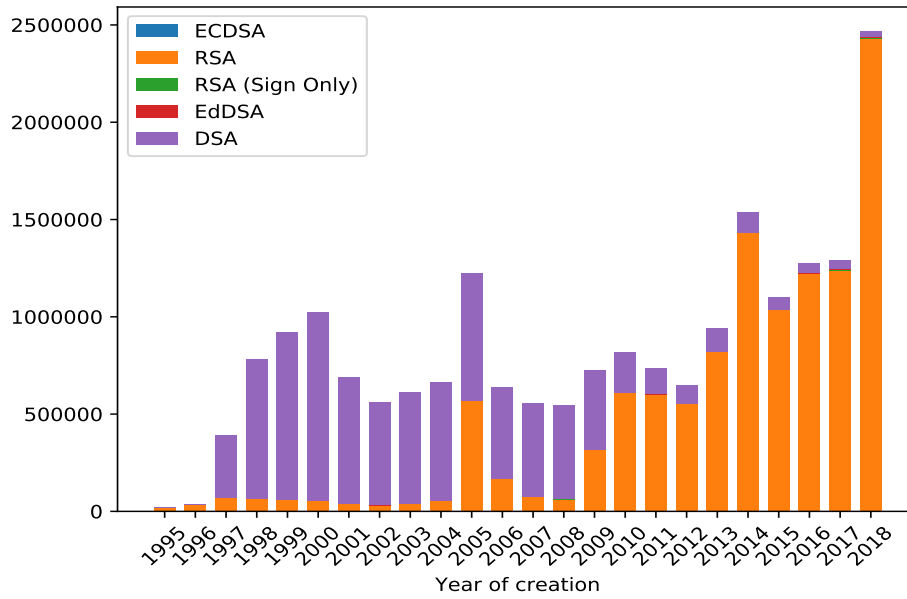


Figure 3.25: Signatures used per year

Table 3.4: signature data

	ECDSA	RSA	RSA (Sign Only)	EddSA	DSA
1995	0	17,536	0	0	758
1996	0	35,452	0	0	2,388
1997	1	71,536	8	0	$3.2 \cdot 10^5$
1998	0	65,375	7	0	$7.18 \cdot 10^5$
1999	0	60,337	1	0	$8.59 \cdot 10^5$
2000	1	53,546	0	6	$9.67 \cdot 10^5$
2001	1	39,625	0	0	$6.5 \cdot 10^5$
2002	0	32,462	1	1	$5.27 \cdot 10^5$
2003	0	41,449	0	95	$5.72 \cdot 10^5$
2004	1	53,531	0	76	$6.07 \cdot 10^5$
2005	0	$5.71 \cdot 10^5$	20	1	$6.54 \cdot 10^5$
2006	0	$1.67 \cdot 10^5$	67	0	$4.7 \cdot 10^5$
2007	0	74,537	77	55	$4.8 \cdot 10^5$
2008	5	63,476	154	0	$4.8 \cdot 10^5$
2009	0	$3.16 \cdot 10^5$	282	3	$4.08 \cdot 10^5$
2010	2	$6.09 \cdot 10^5$	226	0	$2.09 \cdot 10^5$
2011	19	$6.01 \cdot 10^5$	72	5	$1.33 \cdot 10^5$
2012	26	$5.53 \cdot 10^5$	26	0	97,444
2013	165	$8.21 \cdot 10^5$	14	0	$1.19 \cdot 10^5$
2014	317	$1.43 \cdot 10^6$	400	297	$1.05 \cdot 10^5$
2015	2,016	$1.03 \cdot 10^6$	729	844	66,030
2016	1,822	$1.22 \cdot 10^6$	1,078	2,043	48,463
2017	2,310	$1.23 \cdot 10^6$	4,223	4,521	43,587
2018	2,802	$2.43 \cdot 10^6$	3,473	7,223	28,949

3.1.6 usage state

From the certificates loaded in peaks database, we were able to collect a lot of user identities inputs. Of such inputs, we found out that not all address where is the “standard form” (Name Surname (Optional comment) <email address>), so we need to filter the data to find what kind of data was stored.

Even if a lot of public keys are actually left without an email address, we were able to collect a lot of domains used, with the purpose of test the availability, and actually we found a large number of unreachable hosts.

As demonstration of the keyserver network usage and in general of OpenPGP software appliances we collected also evidences of public keys used to sign PPAs.

A Personal Package Archive (PPA) is a special software repository for uploading source packages to be built and published as an APT repository by Launchpad, a web platform, developed and maintained by Canonical Ltd, that allows users to develop and maintain software, particularly open-source software. A developer which want to build open source software is required to create a key pair, used to sign created packages, so users could import the public key from a public keyserver and verify the authenticity of the software.

Unfortunately we noted also an alarming trend in user inputs: there were a bunch of url, which we do not scan, but for sure some were magnet link to torrent files. This is the practical demonstration that keyserver needs to be proactively secured against malicious user inputs, which even if not damaging the keyserver infrastructure directly, could damage the image of the keyserver's role in the OpenPGP ecosystem, and expose the network to several misuses.

Table 3.5: statistics from user identities

Total user identities found	7447201
Mail addresses	6033190
Domains (from mail addresses)	1145792
Domains reachable	744037
PPA references	19084
Url	3661
Magnet link	1732

All domains found have been tested for availability of an MX record, that specifies a mail server responsible for accepting email messages on behalf of a recipient's domain. The query have been all double checked for higher reliability: the domain is checked multiple times, until two consecutive equal

response values are found. However, being bulk networking tests, some error, in particular regarding the error reported, is expected.

Table 3.6: statistics from unavailable domains

Unavailable Domains	401755
Non existent Domains	285762
DNS Label error (Empty Label/Label too long)	283
No MX in Answer	97511
DNS Misconfigured (Servfail in DNS query)	11113

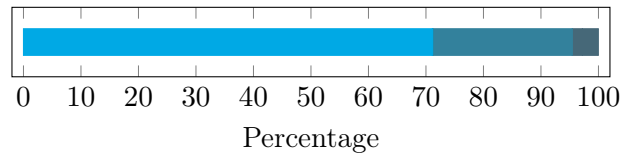


Figure 3.26: Domain Un-reachability chart

- Non-existent Domain
- DNS Label error
- MX not found
- Timeout
- Misconfigured Records (SERVFAIL)

A huge portion of the domains once used are not reachable anymore.

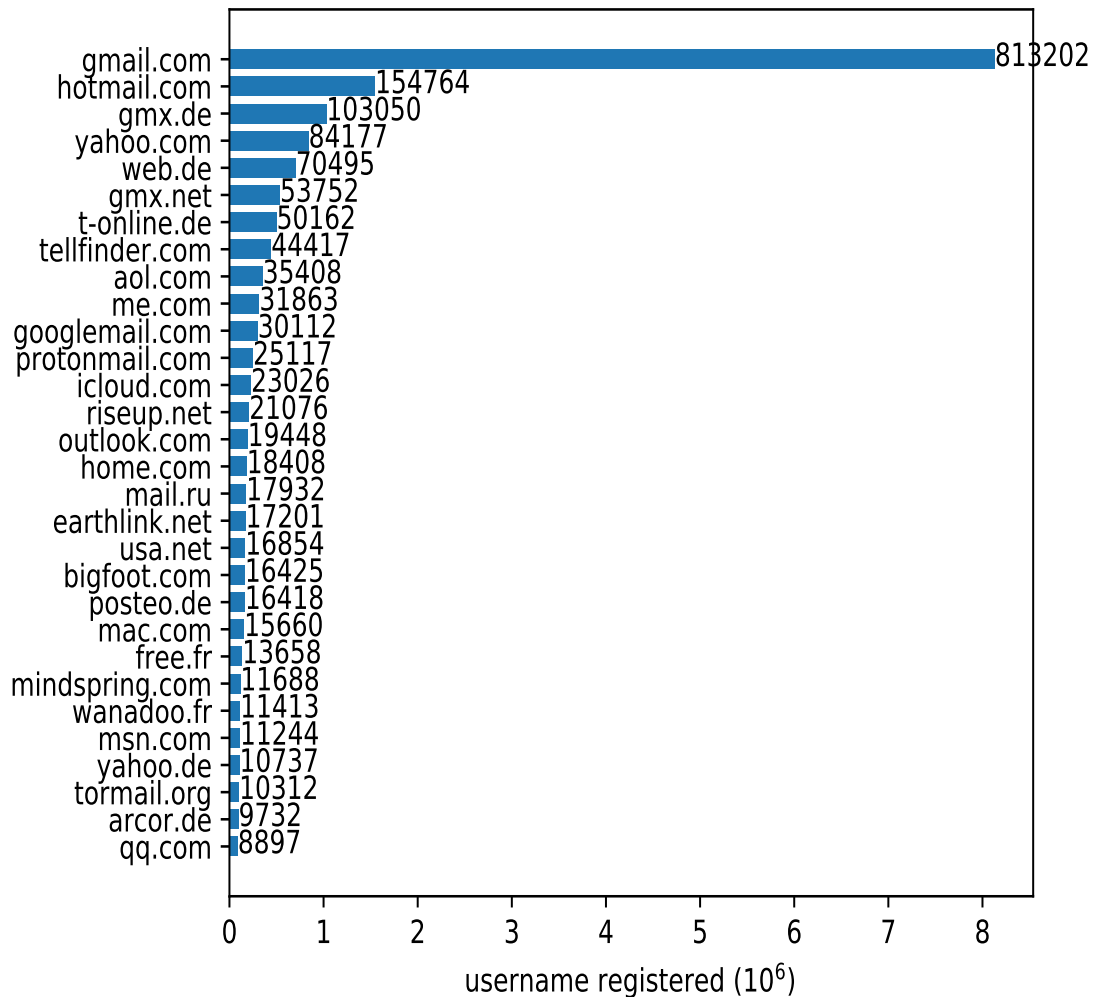


Figure 3.27: top domains per number of mail addresses registered

The results are clear: Gmail addresses are the overwhelming majority, followed by Hotmail. Since its release between 2004 and 2008 Gmail gained a huge popularity, with a diversity of used domains in continuous decrease, maybe also due to the increased difficulty of running a self-hosted mail server [1]

3.1.7 growing trend

In the last years, there is a growing interest in public key cryptosystem, confirmed by the number of keys introduced into the keyserver network, which increases as the year passes.

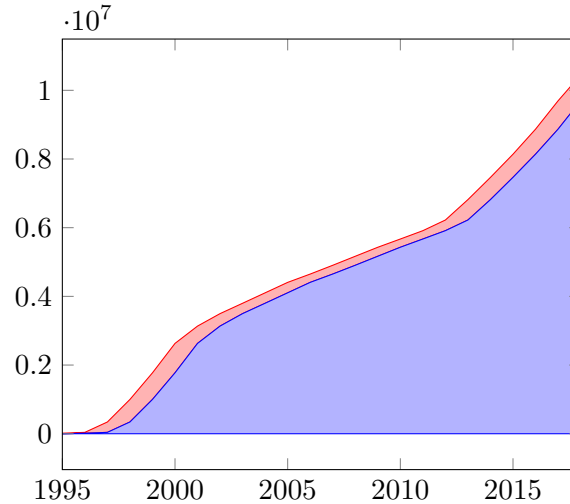


Figure 3.28: public keys added per year on top of the pre-existent

Those big number are creating issues such as network interruptions, high load on the machine and high disk space usage, affecting pool stability and service availability, even convincing some system administrator to shut down their keyserver.

Since the trend is clear, in the next years to come keyserver will need to be very robust pieces of software, capable of handling a large number of keys and eventually filter out noise coming from tests, abuses, broken implementations, etc. Currently the servers pool based on the state of the art are trying to tackle the issue using cluster nodes and having higher hardware threshold for inclusion in pool. In my honest opinion the solution is not to “throw more CPUs/RAM at the problem”, but to build clever, engineered and optimized software, capable of handling the load even on low-to-mid level machines, especially because keyserver are build by the community for the community, so everyone should be able to contribute.

In particular during the last year, there was a spike of unsafe RSA keys (1024 bit) uploaded to the keyserver network. The following chart focus on this public keys since they appear to follow a common pattern: similar UID,

using a lot of UID (> 50) per public key, same algorithm used (RSA), same n factor size (1024 bit).

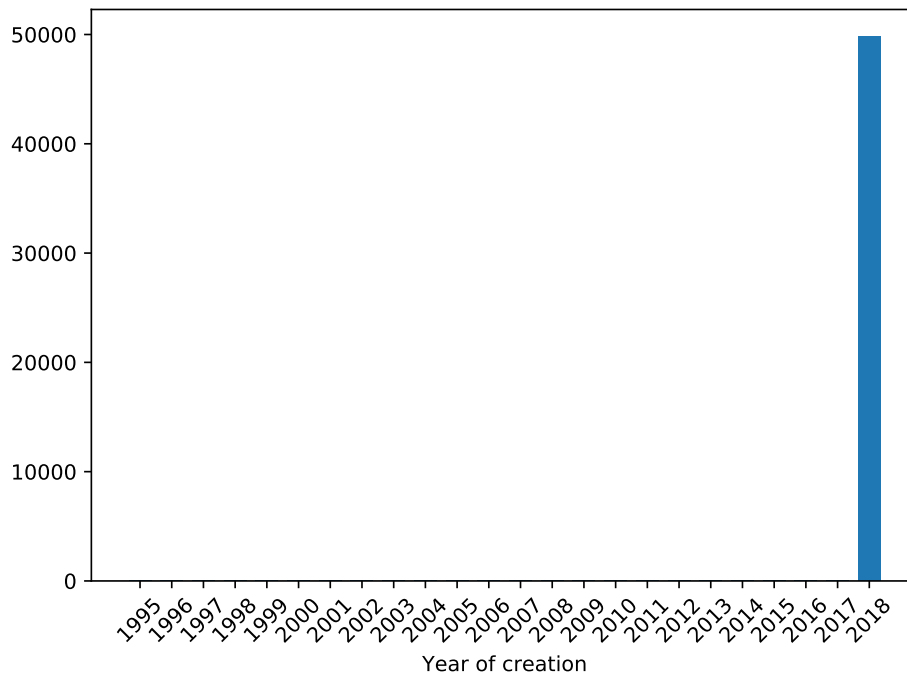


Figure 3.29: Unknown unsafe keys

Given the nature of the keyserver pool, it is impossible to know from where those keys came from, so it is necessary to have a filter mechanism which at least limit the rate of keys uploaded by a single entity, i.e. to avoid the unsupervised creation and uploading of public keys by an automatic program

Conclusion

3.2 Conclusions

The OpenPGP community needs to be more proactive regarding the software on which it rely daily: the current state of the art suffers from a stagnation in the development cycle, and this is not tolerable, since it could be the starting point for arising security issues, becoming the single point of failure of a solid cryptosystem.

Future works Peaks is not yet complete. Since it is based on a modern language, have a well-documented basis, and uses extensible and documented libraries it should be possible to extend its capabilities:

- Implementing key removal: it should be possible now to implement a correctly working key removal functionality
- Display warnings for insecure keys: on the web page displaying informations about searched key it should be possible to display accurate informations, gathered from the analyzer, regarding the security issues that a key may suffers
- Use a more robust hashing algorithm for the recon protocol: MD5 it's now considered broken, so more robust algorithms should be implemented to avoid attacks to the keyserver infrastructure and the whole WoT.
- Filter incoming keys: Users should not be allowed to upload unfiltered key material. Big user attributes, illegal usernames, broken keys should be blocked by the keyserver before importing them, reporting an error message to the user.

Conclusion

- Compliance with new standards: Whenever new version of OpenPGP should be release, this software needs to be updated to reflect those changes
- More documentation: There are still modules left to be documented. For any developer new to the keyserver world the learning curve should not be too harsh

The OpenPGP community is like a flowing river: it never stops, and so should be development of software needed to support and maintaining alive such community

Appendix A

Data types used

During all the chapter 2 several data types has been cited and used in several ways: definition of structures, algorithms, textual explanation of data flow, etc. This Appendix will illustrate which types were taken into account and how they where included as part of the specifications.

Int *Signed integer* of 4 bytes, capable of representing 4 294 967 296 unique values.

Short int Small integer of just 1 byte, capable of representing up to 256 different values.

Bool Boolean value: can assume *True* of *False*.

ZZp *Integers modulo p*, arbitrarily big. The modulus p may be any positive integer, not necessarily prime. [16] In Peaks the value of p is fixed to “530512889551602322505127520352579437339”

String Variables of this type are able to store sequences of characters, such as words or sentences. Each character composing the string is expected to use just 1 byte.

Bitstring Store a sequence of “1” and “0”. The *bitstring* should not to be confused with the *bitset*: is the bitstring new bits are appended to the sequence at the right (in bit terms they are inserted as the least significant

bits). The internal representation of the bitstring is a byte array, and an integer, from which the correct bit sequence can be recovered.

Set Collection of unique elements. Referred to ZZp numbers special are must be taken when using the set provided by the standard libraries, because usually set are stored as binary tree which, in order to provide fast access to elements, sort them by some order, but with modular integers there is no defined concept of ordering.

Vector Container to store multiple occurrences of a particular data type. Elements are appended at the end or at the front, and the accesses to the data is usually sequential from the first to the last element.

Queue Designed to operate in a FIFO context (first-in first-out), where elements are inserted into one end of the container and extracted from the other.

Deque Double-ended queues are sequence containers with dynamic sizes that can be expanded or contracted on both ends (either its front or its back).

Appendix B

Parameters

During description of the implementation, especially in algorithms, but also in explanations, several parameters were used. Now they will be explained in the following appendix, specifying for each, if it is derived, or decided arbitrarily, and its current value, if any.

mbar Integer, is a parameter which influences the ptree and all the synchronization process.

Default 5

bitquantum Integer, influences how many children a node in the prefix tree has.

Default 2

ptree thresh mult Integer, a coefficient to be multiplied with other parameters to obtain a specific threshold.

Default is 10

num samples Integer, number of samples used in the linear interpolation algorithm.

Calculated as $mbar + 1$

points Vector of ZZp numbers, representing interpolation points.

Dimension of the vector is *num_samples*.

Default values are extracted from the sequence [0, 1, -1, 2, -2, ...]

Parameters

split threshold Integer, threshold at which a node's partition is too big and needs to be splitted into $2^{\text{bitquantum}}$ child nodes

Calculated as $\text{ptreethreshmult} \cdot \text{mbar}$

join threshold Integer, threshold at which the children of a node are no longer required and they are joined with the parent

Calculated as $\frac{\text{splitthreshold}}{2}$

P SKS ZZp, finite field used by sks-keyserver.

Default is 530512889551602322505127520352579437339

P SKS bytes Integer, number of bytes which composes the PSKS bytes representation.

Default is 17

hashquery len Integer, fixed size for sending hashquery request to peer.

Calculated as P SKS Bytes -1

version String, version of the software, referred to sks-keyserver versions.

Default is "1.1.6", which is the current version of sks-keyserver.

recon port Integer, port used for the recon protocol.

Default 11370

http port Integer, port used to serve certificates.

Default 11371

filters String, filters used for certificate merging.

Default value "yminsky.dedup, yminsky.merge"

gossip interval Integer, interval between gossip attemps.

Default value 60

max read length Integer, maximum bytes length allowed of a single message.

Default $1 \ll 24$

max recover size Integer, maximum number of elements which could be recovered from a single reconing session.

Default 15000

max request queue len Integer, max length for the recon request queue, used by the server.

Default value is 60000

request chunk size Integer, maximum chunk of certificates that could be requested by the peer in a single http request.

Default 100

Bibliography

- [1] Mitchell Anicas. <https://www.digitalocean.com/community/tutorials/why-you-may-not-want-to-run-your-own-mail-server>, 2014. Online; checked 28 November 2018.
- [2] Elaine Barker and Quynh Dang. <http://dx.doi.org/10.6028/NIST.SP.800-57pt3r1>, 2015. Online; checked 28 November 2018.
- [3] V. Alex Brennen. Cryptnet. <http://freshmeat.sourceforge.net/projects/cks/>. Online; checked 8 November 2018.
- [4] Jon Callas, Lutz Donnerhacke, Hal Finney, David Shaw, and Rodney Thayer. Openpgp message format. *RFC*, 4880:1–90, 2007.
- [5] Danny Cohen. On holy wars and a plea for peace. *IEEE Computer*, 14(10):48–54, 1981.
- [6] Yaron Minsky et al. sks-keyserver. . Online; checked 8 November 2018.
- [7] Jason Lee. <https://github.com/calccrypto/OpenPGP>, 2013. Online; checked 28 November 2018.
- [8] GPG mailing list. <https://lists.gnupg.org/pipermail/gnupg-announce/2009q3/000291.html>, 2009. Online; checked 28 November 2018.
- [9] Horowitz Mark. A PGP Public Key Server. <http://www.mit.edu/afs/net.mit.edu/project/pks/thesis/paper/thesis.html>, 1996. Online; checked 8 November 2018.
- [10] Casey Marshall. hockey puck. <https://hockeypuck.github.io/>. Online; checked 13 November 2018.

BIBLIOGRAPHY

- [11] Jonathan McDowell. onak. <http://www.earth.li/projectpurple/progs/onak.html>. Online; checked 8 November 2018.
- [12] Yaron Minsky, Ari Trachtenberg, and Richard Zippel. Set reconciliation with nearly optimal communication complexity. <http://ipsit.bu.edu/documents/ieee-it3-web.pdf>, 2004. Online; checked 8 November 2018.
- [13] Yaron Minsky and Ari Trachtenberg. Practical set reconciliation. <http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.456.7200&rep=rep1&type=pdf>, 2002. Online; checked 9 November 2018.
- [14] peegeepee. peegeepee. <https://peegeepee.com/>. Online; checked 8 November 2018.
- [15] Joyce K. Reynolds and Jon Postel. Assigned numbers. *RFC*, 1700:1–230, 1994.
- [16] Victor Shoup. <https://www.shoup.net/ntl/doc/tour.html>, 2018. Online; checked 28 November 2018.
- [17] Yegor Timoshenko. <https://bitbucket.org/skskeyserver/sks-keyserver/issues/60/denial-of-service-via-large-uid-packets>, 2018. Online; checked 28 November 2018.