Universidad Nacional Autónoma de México
Posgrado en Ciencia e Ingeniería de la Computación

A certificate-poisoning-resistant protocol for the synchronization of Web of Trust networks

TESIS

que para optar por el grado de:
Doctor en Ciencia e Ingeniería de la Computación

Presenta:
Gunnar Eyal Wolf Iszaevich

Tutor:
Dr. Jorge Luis Ortega Arjona
Facultad de Ciencias, UNAM

Ciudad Universitaria, Febrero de 2025

Contents

Abstract

A fundamental issue of encrypted network communications is ensuring trust on the identity of the counterpart of a communication. This is commonly achieved by means of a *trust distribution model*.

The decentralized trust distribution model, termed *Web of Trust*, has been found to have several weaknesses. One such weakness –certificate poisoning– has brought the main OpenPGP certificate keyserver network to the limits of sustainability.

The present thesis presents a protocol for key certification and synchronization that avoids the nocive effects of certificate poisoning, while retaining compatibility with as much as possible of the preexisting OpenPGP and HKP toolset.

Chapter 1

Introduction

1.1 Context

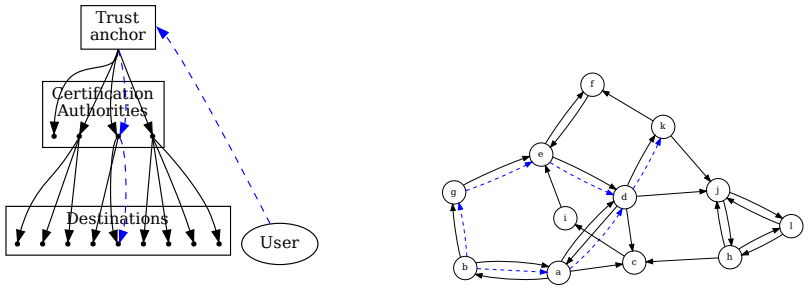
Encrypted communications are nowadays the norm on the Internet. But, contrary to what many believe, encryption goes way beyond wrapping all communication in a cryptographic algorithm to obscure third parties from accessing or tampering with their contents: not only the information must be kept protected from being intercepted by third parties *over the wire*, but every endpoint must ensure the identity of its counterpart to avoid the impersonation by a hostile third party. While in small-scale settings, case-by-case verification could suffice, *trust distribution models* are used for large-scale communication (Jøsang, Gray, and Kinateder 2006).

There are two main models for trust distribution models: a) The centralized model, based on the Public Key Infrastructure Certification Authorities (PKI-CAs, see Figure 1.1(a)), and b) the distributed model, based on the *Web of Trust* (WoT, Figure 1.1(b)) (Abdul-Rahman 1997; Levien 1995; Ryabitsev 2014). Both models rely on a mechanism to distribute (and keep up to date) a set of identities.

Focusing on the distributed model, the most widespread implementation is OpenPGP: there is a network of key servers, using the HKP protocol (Horowitz 1997; Shaw 2003), which maintains synchronization (key additions and modifications) using a *gossip* or *epidemic* protocol (Alon, Barak, and Manber 1987; Demers et al. 1987; Bagdasaryan 2016). This network has been the target of attacks in recent years, jeopardizing the very viability of the distributed model.

1.2 Problem statement

This work targets the threat of *certificate poisoning*: an excessive number of certificates bloating public keys served by the HKP OpenPGP keyserver network. This attack poses a potential *denial-of-service* on users of the WoT network,



(a) Centralized model: PKI-CA. User verifies there is a valid path of trust from a prespecified trust anchor to the destination they want to reach.

(b) Distributed model: WoT. User b builds trust paths towards target k along the existing edges of a graph.

Figure 1.1: Transitive trust models. Black lines denote all trust relationships in the network, and blue dashed lines mean trust paths followed from a user to their target.

which translates into an existential threat to the keyserver network, jeopardizing the viability of the WoT model as a whole.

The main implementations of trust schemes based on the *Web of Trust* model (WoT) are susceptible to attacks that put the entire model at risk, particularly *certificate poisoning* (see Section 2.6.6).; an attacker A creates a large amount of throwaway keys $k_{A1}..k_{AN}$, certifying a victim V'S public key k_V with them; thus, A can make k_V's size (usually below 100KB) to grow to tens of megabytes; next, by uploading k_V to the public keyserver network, A can easily cause a denial of service (DoS) on widely used cryptographic tools such as *GnuPG* whenever it attempts to parse k_V. Given that the HKP protocol used by the keyserver network does not allow for information deletion, V is forced to migrate to new keys. As newly created keys have not yet built relations with other users of the network, and as key certification is a manual process involving individualized identity checking, this attack can effectively lead to the weakening, or even of the complete breakdown, of a WoT network.

1.3 Hypothesis

A protocol is proposed that allows for the synchronization of OpenPGP certificates between keyservers, while preventing the *certificate poisoning* attack. It preserves the main characteristics of the Web of Trust transitive trust distribution schema, including that of being fully decentralized, by performing relatively small modifications to the HKP keyserver interaction model, and allowing for interoperability with the currently deployed client software.

1.4 Approach

This research aims to explore and contribute with a solution to maintain the distributed model's viability, avoiding the need of a centralizing entity. The objective is to propose a protocol that counteracts certificate poisoning, without compromising the main properties of the WoT model — particularly, its fully decentralized, distributed operation.

By requiring all actions affecting a given public key to be signed by its private counterpart, keyservers are no longer vulnerable to poisoned certificates. The recommendation of communicating new certification to the signee and not uploading them to the keyserver network improves from being a *best practice* to being mandated by the interaction model itself.

In this thesis, the described implementation is carried out as a proof of concept by modifying *Hockeypuck*, the currently leading keyserver implementation, attempting to keep the code modifications to a minimum. An experimental verification of this is presented, showing the proposed protocol is effective against the described attack.

1.5 Contributions

We identify as the **main contribution** of this work:

- A network protocol for the exchange of certifications on cryptographic identities that can reliably prevent certificate poisoning attacks against OpenPGP keys in a Web-of-Trust environment, with minimal modification to server software and without requiring modifying existing end-user tools.

This allows us to also enumerate **secondary contributions**:

1. Give the key owner the ability to control which certifications are to be published, granting them reputation control.
2. Allow the WoT decentralized trust model to remain relevant and sustainable for communities based on OpenPGP.

1.6 Structure

This work is structured as follows:

- The upcoming chapter, *Background*, covers the relevant information on encrypted communications, protocols and trust models, presenting a general overview of the field.
- Several attacks on the various trust models have surfaced over the years. Chapter 3, *Related Work* presents instances of how they have been faced. Whenever pertinent, similarities and differences to this work is highlighted.

- Chapter 4, *A proposal for a certificate-poisoning-resistant protocol* details the weaknesses in the WoT model that put its main implementation at risk, as well as the answers put forward by its developers, and reasons as to why it is desirable not to lose the WoT's unique characteristics. The proposed protocol is presented, contrasting it to the preexisting interaction model.

- Chapter 5, *Implementation, validation and results*, presents the technical details for the implementation and its experimental validation. Section 5.1, it reviews how key creations, certifications and attestations are carried out with the *Sequoia* command line tool, and how this translates to packets following the OpenPGP standard. Section 5.2 proceeds to outline the experiment that validates the proposed protocol and presents the results on how its adoption successfully prevents a certificate poisoning attack.

- Finally, Chapter 6, *Conclusions*, reviews the work developed during this thesis, verifies the goals set have been reached, discusses the conclusions of the present work, and sets the lines for future research.

Chapter 2

Background

This chapter presents the basic concepts used throughout the work: it starts by presenting a list of basic terminology to be used throughout in Section 2.1. Section 2.2 presents the basic concepts and implementations of encrypted communications. Section 2.3 explains why, besides encrypting each specific channel, there is a need for establishing the needed trust between communication endpoints, and details some of the main implementations that do so. Section 2.4 introduces trust distribution models, this is, details on said implementations focusing on how they handle trust management. Section 2.5 presents an overview of the two most notable major implementations of the concepts presented in this chapter. Section 2.6 presents the main weaknesses that have appeared over the years in trust distribution models.

2.1 Terminology

This work relies on concepts related to Cryptography. Given the terms used often differ between sub-fields or even between authors, this section briefly presents the definitions for the main concepts used throughout the following pages. The terms are harmonized with those defined in the OpenPGP standard (Callas et al. 2007), following the usage recommended by the Sequoia project (Schaefer 2023), as well as the inspiration provided by the widely circulated "Keysigning Party HOWTO" (Brennen 2008).

Some of the concepts referenced in this list are to be defined and presented at length throughout this chapter. The terms themselves are presented alphabetically; given they are closely interrelated concepts and this section is to remain brief, this list is provided as a set of definitions to come back to rather than a self-contained, ordered glossary, based on the above mentioned references.

Certificate A given user's public key validated (*signed*) by itself and, often, by third parties. *Certifying a key* implies an external actor has validated the link between a *real world* identity and a given key pair.

Cryptographic key Cryptographic communication methods require the parties of a communication to agree on one (for *private key cryptography*) or a set of (for *public key cryptography*) values with which the contents of the communication are scrambled and retrieved. The types of cryptography are presented in the following section.

Key pair A pair of keys for public key cryptography consisting of interrelated public and private (or secret) keys.

Keyring A collection of certificates, often held by communication parties to certify other participants' identities. A keyring can be stored as different packets on a single file or as a collection of files.

Keyserver An online system hosting a database of certificates public key material. Keyservers may be queried by users who wish to acquire the certificate for a communication party they have not had prior contact with.

Keyserver network A set of keyservers which synchronize their sets of key material, so that users of any of the connected keyserver get a homogeneous set of results.

Keystore Repository of certificates from which a given user can deduce trust on a third party when initiating communication. A keyserver can be seen as a public keystore.

Keysigning The action where the owner of an OpenPGP key pair adds their signature to another user's certificate.

OpenPGP The communications standard that encompasses the original PGP program and subsequent implementations, the format of the messages exchanged by said implementations, and an ecosystem of tools used to operate with them. OpenPGP can operate with a number of cryptographic algorithms, and provides a public key cryptosystem.

Pretty Good Privacy (PGP) Privacy software developed by Phil Zimmermann in 1991, which includes public key cryptography, a standard packet and key format, and symmetric encryption as well.

Public Key In public key cryptography, the key of a key pair which can be shared. For practical uses, when transitive trust schemes are used, public keys are often distributed as part of *certificates*; literature often conflates their meaning.

Self-signature The OpenPGP standard specifies a certificate including as a minimum the relevant public key, as well as a certification of said key with its own private key component.

Secret or Private Key In public key cryptography, the key of a key pair which is kept secure. The most common usage in cryptography is the term "Private key", but partly due to its long history, in OpenPGP the term "Secret key" is still prevalent.

Transferable Public Key A certificate that does not include any private (or secret) key material.

Transferable Secret Key A certificate that includes private key material.

Trust Path A route by which trust is extended from one entity to another. In OpenPGP, this is a link of trust between two public keys.

Web of Trust (WoT) A trust model based on the collection (network) of certifications upon keys and resultant trust paths in a user-centered trust model which provide for authentication. Collectively, the trust relationships between a group of keys.

2.2 Encrypted communications

Network communications under the TCP/IP protocol family with which Internet operates are, by default, not secured in any way: the network protocol design is *layered*, and pursues as a design goal being as light and flexible as possible (Braden 1989). As Figure 2.1 shows, communications over TCP/IP appear as a *protocol stack*, where each of the layers communicates with its corresponding layer in the remote host — and none of them deals with securing network communications (Oppliger 1998).

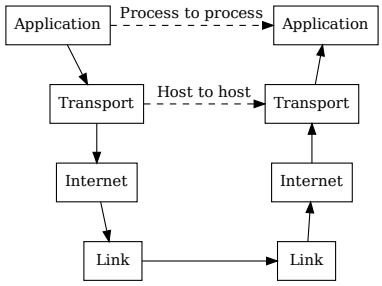


Figure 2.1: Communication layers used in the TCP/IP protocol family

Users needing to ensure in-flight data security —be it for eavesdropping resistance, ensuring message integrity, or certifying message provenance— do so by *tunneling* application-level communications, as it is done by the *Transport Layer Security* protocol (see Subsection 2.5.1; Chen, Miao, and Q. Wang 2007).

On the other hand, users needing to ensure data security in messages that are to be stored need a different mechanism: it is not just a matter of encrypting the data while in transit, but it should be stored strictly in encrypted form only, and decrypted only on demand, as it is done by the implementations of the *OpenPGP* standard (see Subsection 2.5.2, Schwenk 2022, p. 377).

Cryptographic communications require parties to a communications channel to use a key or set of keys to perform the needed transformations on the data to encrypt and decrypt it. Algorithms that work by having all parties to the communication use the same key make up *private key cryptography* (see Subsection 2.2.1), while those in which each of the parties has a different *key pair* belong to *public key cryptography* (see Subsection 2.2.2).

2.2.1 Private key cryptography

Cryptographic algorithms that require all valid users of a communication channel to provide the same key are termed *private key* or *symmetric* cryptography. This means, there is a set of symbols or values that allows for decrypting or participating in a communications channel (W. Stallings 2017, pp. 86–87).

Communications secured by private key cryptography require the key to be kept secret only to the parties involved: any hostile party obtaining the key is able to decrypt messages or pass forged messages as if they are legitimate.

Private key cryptography exists since early in human history. The *Caesar cipher*, a simple mono-alphabetic substitution cipher used over 2000 years ago, uses as its key a small integer k, $0 < k < 26$: the number of characters to shift the whole alphabet. The messages encrypted with it are unintelligible to people not knowing both the algorithm and the key used. Of course, this early algorithm is easy to break by a simple brute-force attack; modern-day algorithms are strong enough to resist any such attack by current computers. One of the strong points in favor of private key algorithms is that they are very efficient, even for large streams of data (Menezes, Van Oorschot, and Vanstone 1996, p. 31).

Private key cryptography is affected by the *key distribution* problem (Merkle 1978): if users can find a channel secure enough to use it to agree on a common key, they could also be able to use it to exchange their full communications. This means, it is not feasible to use only this kind of cryptography as a basis for network-wide communications between arbitrary parties: a key establishment phase needs to take place prior to sustaining any encrypted communications over a secure channel. Furthermore, to ensure private one-to-one communications using private key cryptography, each participant in the network must use a different key for each of the other participants, and the total amount of keys is $\frac{n \times (n-1)}{2}$, as Figure 2.2 shows (W. Stallings 2017, p. 443). As n grows larger, the complexity of securely handling a quadratically-growing number of keys becomes overwhelming. This issue is termed *the n^2 key distribution problem* (Menezes, Van Oorschot, and Vanstone 1996, p. 546).

2.2.2 Public key cryptography

Public key cryptography is introduced by the seminal work of Diffie and Hellman 1976. Quoting from its abstract:

> (...)Widening applications of teleprocessing have given rise to a

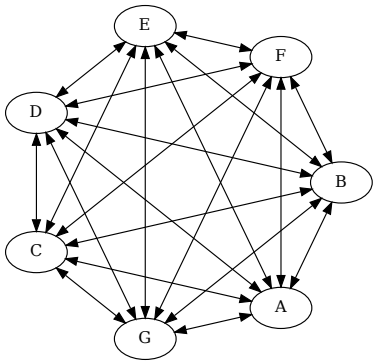


Figure 2.2: When using private key cryptography, each participant in the network must use a different key for each of the other participants, yielding $\frac{n \times (n-1)}{2}$ keys in the system (W. Stallings 2017, p. 443).

> need for new types of cryptographic systems, which minimize the need for secure key distribution channels and supply the equivalent of a written signature.

Diffie and Hellman 1976 is considered a seminal paper as it is the first to propose a cryptosystem under which each participant generates and uses *two* keys: a private one, to be closely guarded in secret, and a public one, that can be widely distributed. The keys are different but *related*: there are two functions implementing invertible transformations, *encrypt* E and *decrypt* D, which given a message and the corresponding key inverts each other.

R. L. Rivest, Shamir, and Adleman 1978 present the first practical public-key encryption and signature scheme (Menezes, Van Oorschot, and Vanstone 1996, p. 2), with an algorithm now referred to as RSA, based on the intractability of factoring large integers. In the cryptosystem brought forward by Rivest, Shamir and Adleman, given a participant A that publishes their public key k_A and guards their private key k'_A, if a cleartext message M and the public key k_A are input to E, it yields the encrypted message C; given the encrypted message C and the private key k'_A to D, it decrypts it, yielding the original message, M. This is,

$$C = E(k_A, M)$$

$$M = D(k'_A, C)$$

E and D are thus known as *trapdoor one-way functions* (W. Stallings 2017, 288–293): they are easy to calculate in one direction, but infeasible to invert without having access to the corresponding key.

Functions D and E can also be used to certify a given message M is emitted by a participant of a network A: if A uses their *private* key k'_A to encrypt a cleartext message M, the result becomes the *signed* message S (Menezes, Van Oorschot, and Vanstone 1996, p. 433; R. L. Rivest, Shamir, and Adleman 1978, p. 121):

$$S = E(k'_A, M)$$
$$M = D(k_A, S)$$

Thus, any other participant with knowledge of A's public key k_A can retrieve the original message, and independently, prove it has been effectively generated by A. This usage is usually called *verifying* a message. For practical reasons, signatures are usually performed over *hashes* of the full message rather than over the message itself (Menezes, Van Oorschot, and Vanstone 1996, p. 429)

2.3 Trust distribution models

Given that the public keys can be, in fact, published, the constraints for agreeing on a secret prior to establishing secure communication between two participants is vastly reduced — but care must still be taken by all participants to ensure they are communicating with the correct destination user, and not with an impersonator. This is known as a *Man-in-the-Middle* or *MitM attack* (Conti, Dragoni, and Lesyk 2016).

In a global scale network like the Internet, it is impossible for any given participant to know all of the public keys for other participants prior to engaging in communications with them. Trust distribution models tackle the problem of how to convey trust to the identity of a given certificate: how to ensure it really belongs to the desired endpoint, and not to an impostor (B. Borcherding and M. Borcherding 1998). To do so, trust distribution models build on the *certification* capability of public key cryptosystem: if user U knows already an introducer I's certificate, and I knows provider P's, the first time U contacts P, P presents its *certified* public key Ck_P so that:

$$Ck_P \leftarrow \{k_P, E(k'_I, H(k_P))\}$$

Ck_P consists of P's public key k_P joined with the encryption, using I's private key k'_I of the result of hashing k_P (Schwenk 2022, 383–385). This way, U has a *trust path* to P, introduced by I.

However, this still presents a problem when considering network-wide deployment: how can U know which introducer I to choose in order to build a trust path to P? How can P know which certificates to serve together with k_P when queried for Ck_P?

2.3.1 Public Key Infrastructure Certificate Authorities (PKI-CAs)

The most commonly used trust distribution model bases its operation on Public Key Infrastructure Certificate Authorities, also termed *trust hierarchies* (B. Borcherding and M. Borcherding 1998). In this model, graphically exemplified in Figure 1.1(a), there is a small group of central, ultimately trusted entries (trust anchors), which certify a set of Certifying Authorities possibly arranged in a hierarchy expressing trust relationships, particularly for delegating certification with specific characteristics.

Trust anchors are set of over a hundred public keys, usually defined by the operating system or Web browser, and are the base for the operation of TLS, covered in Section 2.5.1. This model is based upon *hierarchical, transitive trust*: in order for user A to trust system S's identity, S presents a *certificate chain*, signed by Certification Authority CA, and ultimately signed by TA, one of the defined trust anchors, as presented in the previous section (CA/Browser Forum 2021, 16–18):

$$Ck_S \leftarrow \{k_S, E(k'_{CA}, H(k_S))\};$$
$$Ck_{CA} \leftarrow \{k_{CA}, E(k'_{TA}, h(k_{CA}))\};$$
$$Ck_{TA} \leftarrow \text{Trust anchor}$$

Certificates under this model are usually sent as part of the session establishment protocol, or attached to every relevant message in the case of non-connection-oriented protocols (Rose et al. 2016, p. 13).

A vast majority of the encrypted network traffic nowadays verifies the service's identity by using the this trust model. Trust anchors are usually defined by operating system or web browser providers, not by end users (W. Stallings 2017, p. 457), and although client certificates are possible to obtain, due to their cost and complexity (Aas et al. 2019), its operation is usually limited to end users verifying service providers' identity (Fu et al. 2001). Prospective CAs seeking to be included in the default lists trusted by operating systems and browsers have to comply with the baseline requirements defined by the CA/Browser Forum 2021.

2.3.2 Web of Trust (WoT)

There are, however, several use cases for which PKI-CAs are not a good fit. PKI-CA is ill suited for communications that can be modeled as peer to peer connections, such as e-mail or instant messaging, where trust is measured based on a set of individual nodes representing their social interactions. Models in which universal, system-wide trust anchors cannot be determined, have to be devised in a *decentralized* way: each user of the network acts as their own root of trust, and certifies other arbitrary nodes based on whichever rules are set for said network (usually, based on personally assessing the corresponding user's identity and control of the presented key), as presented in Figure 1.1(b), in

a fashion familiar to many as *social network* diagrams. The introduction of *SDSI — A Simple Distributed Security Infrastructure* starts by highlighting the unsuitability of the PKI-CA model to many common scenarios (Ronald L Rivest and Lampson 1996):

> This paper was motivated by the perception that the existing proposals for a public-key infrastructure (such as X.509-based schemes that require global certificate hierarchies) are both excessively complex and incomplete.

In contrast with the PKI-CAs model, in WoT each of the system's users can validate or *certify* any other arbitrary user's key. Although trust is, again, conveyed transitively, it is no longer hierarchical; the network is a directed graph often presenting cycles and paths have to be sought between any two parties wanting to engage in communication. Paths do not need to be unique; the chain of certificates followed by node b to node k in the aforementioned figure, setting a maximum depth of five *hops*, can be either of:

$$
\begin{aligned}
Ck &\leftarrow \{k_k, E(k'_d, h(k_k))\};\\
Ck_d &\leftarrow \{k_d, E(k'_a, h(k_d))\};\\
Ck_a &\leftarrow \{k_a, E(k'_b, H(k_a))\};\\
Ck_b &\leftarrow \text{Trust path root}
\end{aligned}
\qquad
\begin{aligned}
Ck_k &\leftarrow \{k_k, E(k'_d, h(k_k))\};\\
Ck_d &\leftarrow \{k_d, E(k'_e, h(k_d))\};\\
Ck_e &\leftarrow \{k_e, E(k'_g, H(k_e))\};\\
Ck_g &\leftarrow \{k_g, E(k'_b, H(k_g))\};\\
Ck_b &\leftarrow \text{Trust path root}
\end{aligned}
$$

In contrast with the PKI-CA model, the *trust path root* is the user interested in establishing trust to any other given point in the graph, and not an externally selected third party. Quoting Streib 2003:

> There are other ways to measure how "trusted" a key is, many of them very misleading.
>
> First of all, any measure of trust that does not include your own key, and signatures you've made and received, is bad. Just because I've published something that claims that key X is relatively trusted, does not mean that you should trust it. Your personal web of trust is really the only good measure of trust.

Although this definition might seem excessively *ad-hoc* for wide scale use, it clearly presents the main characteristic of a WoT: it gives a network view *centered on each individual*, and allows them to set their own parameters and boundaries as to whom to extend trust to. The OpenPGP WoT key network has been found to exhibit emergence of small-world caracteristics and it is probably the first such self-organized system (Čapkun, Buttyán, and Hubaux 2002).

While there are other notable instances of WoT networks (Brunner et al. 2020), the focus for this work remains with the most used implementation: OpenPGP.

When certifying a given key, OpenPGP defines four trust levels (Yakubov et al. 2020):

Full (level=4) certificate holder's signature of other users' certificates is fully trusted.

Marginal (level=3) certificate holder's signature can be trusted, but it is better to find other signatures with full trust to confirm the introduction of a certain certificate.

Untrustworthy (level=2) certificate holder's signature should not be trusted and his signature on other users' certificates should be ignored.

Don't know (level=1) there is uncertainty about trustworthiness of certificate holder's signatures of other users' certificates

WoT transitive trust assessment presents a high mathematical complexity which can be mapped in different ways; different users set their trust thresholds at different levels. Jøsang 1999 presents an algebra to quantify trust based on Subjective Logic. The complexity of exploring and determining trust paths does not, however, have to incur in a full network analysis and it is thus not exponential, but can be kept polynomial as only the shortest recommendation paths are searched (B. Borcherding and M. Borcherding 1998).

An important use case of WoT schemes is worth mentioning: given the difficulty of supporting alternative PKI-CA roots of trust under operating-system-provided lists, enterprise- or government-sized organizations can implement a centralized model over a WoT (i.e. by having one key trusted universally by all of its users' generated keys) (Koch 2021a).

In this work, the focus is placed on the global properties of the WoT network as a whole, so no further attention is devoted to signature trust levels or trust quantification.

2.3.3 Trust On First Use (TOFU)

As the two preceding subsections show, transitive trust presents several pitfalls: mainly the ability to select trust agents and comply with the needed steps for obtaining their certificates.

Trust on First Use (*TOFU*), also termed as Leap of Faith (LoF) or Weak Authentication (WA), is a security model where a user, upon a first connection to a yet unauthenticated endpoint, instead of looking for a trusted third party to confirm a new peer's identity, trusts the presented identity to be valid the first time it is encountered, and locally stores it. For any future interactions, though, the communication channel is authenticated with the preexisting, stored identity (Pham and Aura 2011). The model is from the onset known to be weaker than PKI-CAs and WoT, but it is widely used due to it not requiring any additional infrastructure (registration effort, cost, and scalability). Quoting Pham and Aura 2011:

> The security of LoF relies on an important assumption that the attacker is unlikely to be present during the first communication (...)

> In the classical computer security model, LoF is unquestionably insecure. In practice, it has been applied in several context. The most prominent one is SSH. (...) Also, when a user downloads and installs a web browser or an operating system, a list of root certificates (...) is configured on that user's machine. Most users do not bother to verify offline the correctness of these certificates, and thus the LoF mechanism is applied.

Even given all its limitations, TOFU adoption has grown over the years. SSH, mentioned in the above quote, is a very specific case where a systems administrator generates keys in several systems and trusts them because their ability to immediately verify the functionality. Bluetooth establishes cryptographic secure channels between discrete devices with no proper MitM protection because it is a distance-limited protocol, and usually runs in devices with interfaces too limited to perform deeper checking (Sandhya and Devi 2012). The multimedia-oriented Session Initiation Protocol (SIP) needs to blend in with preexisting communications networks where any authentication other than TOFU would be extremely impractical, as well as disrupting to the expected communication experience (Arkko and Nikander 2004). However, since the mid 2010s, TOFU has also grown into a very important niche, where it has become the dominant trust distribution model: instant messaging applications, such as *WhatsApp*, *Telegram* or *Signal*, provide an opportunistic end-to-end secure messaging mode based on TOFU (Herzberg and Leibowitz 2016).

Instant messaging is a very particular case: given its wide adoption, with implementations reaching over a billion users, and the usability hurdles of encryption (Whitten and J Doug Tygar 1999), and the low awareness of its importance (Renaud, Volkamer, and Renkema-Padmos 2014), notifications regarding mismatching keys have to be kept as non-intrusive as possible, but provide a means for off-band verification by tech savvy users (Johansen et al. 2017, pp. 23–48). TOFU has proven to be a good compromise, even though the overall identity assurance of the system is lower than with the other mentioned trust distribution models (Kahn Gillmor 2021, 155–159).

The Autocrypt project, pursuing furthering usability in e-mail encryption in order for more users to be able to adopt it, while building over PGP (which uses the WoT), acknowledges the advantages of TOFU and provides a hybrid trust mechanism. Quoting Krekel, McKelvey, and Lefherz 2018,

> That's why we trust on first use and distribute public keys in the email header. It is hidden, but decentralized, and leaves the users in control of their keys, without them necessarily knowing it. And if they want to do an out-of-band verification with their associates, there will always be user-friendly options, e.g. with a QR code comparison.
>
> Technically, Autocrypt is not much more than a set of some reasonable configuration decisions. But together, the decisions made by Autocrypt can streamline the complex PGP system to be usable for

> encrypted communication between everyone. What encrypted communication needs is simple, measured steps of improvement. That's the only way to bring people together while maintaining the original intent of the architecture.

Thus, TOFU provides a convenient scheme for key distribution, a topic that is presented next, and for minimal involvement cases are as a simple trust conveying mechanism, although for stronger security guarantees it still must be combined with a proper trust distribution model.

2.4 Key distribution

Several schemes are available for determining trust in a given party's identities. However, a problem not yet described is how the certificates themselves for participants of a network are made reachable for said network's participants.

When employing the TOFU model, key distribution is inherently solved: the different TOFU models include the public key as part of their channel establishment protocol (Arkko and Nikander 2004).

For the PKI-CA model, in its most common implementation (TLS), the key distribution mechanism is very similar: the first step of the TLS handshake is the *key share* (Rescorla 2018, pp. 10, 48). The main difference with TOFU-based session establishment is that the *key share* phase is followed by the *certificate chain* exchange phase (Rescorla 2018, p. 64), where the communication endpoints exchange the CA certificates for the keys. This still leaves the issue of communicating trust regarding the trust anchors. As mentioned in Subsection 2.3.1, they are usually defined by the operating system or Web browser in a centralized way, and carry *ultimate trust* for most users' decisions (Hein 2013).

The nature of WoT, however, makes this procedure impossible: upon session establishment, endpoint A does not know which are endpoint B's trusted principals, so it cannot offer a certificate chain. This is not a weakness of the session establishment protocols: in practice, and as Subsection 2.5.2 delineates, the most common settings for WoT are asyncronous; while there is a session establishment, it must be kept minimal as there is no round-trip step — session keys are generated at A, having the knowledge of B's public key (Callas et al. 2007), but not of the path B is connected to the WoT.

Early users of PGP relied on *key distribution parties*, in-person gatherings where a group of users authenticated each other, to later publish their public keys on their Web pages, but this is clearly a non-scalable solution, as it places the huge burden of finding the location for a given key on the party interested in first establishing communications (Rose et al. 2016, p. 73). A natural answer to this issue is the establishment of an *on-line trusted server* to serve as a directory for public keys (Menezes, Van Oorschot, and Vanstone 1996, p. 555).

The first PGP keyserver, developed at MIT in 1994 by Brian A. Lamacchia, with an interface allowing it to be queried over e-mail. By 1997, Mark Horowitz implemented a Web-queriable keyserver, naming it *pksd* (Public KeyServer Daemon, Yamane et al. 2003). The interface for querying keyservers has

been named HKP (OpenPGP HTTP Keyserver Protocol). HKP is an API layer on top of HTTP (Shaw 2003).

Of course, such a server becomes a single point of failure for the whole WoT network: if it is offline, any user attempting to fetch keys or certificates would be effectively blocked from doing so. Even more so, it becomes a single point of *censorship*: if an administrator A for the HKP server wants to block users from securely communicating with a given party B, A can remove B's key from the keyserver (Fiskerstrand 2016b).

2.4.1 The HKP public keyserver network

As several independent keyservers appeared online, the need for synchronization between them becomes clear. The HKP keyserver network uses a *gossip*-based set synchronization protocol, nearly optimal in terms of communication complexity, and tractable in computational complexity (Y. M. Minsky 2002; Y. Minsky and Trachtenberg 2002; Y. Minsky, Trachtenberg, and Zippel 2003). This allows for the establishment of a keyserver network, offering a consistent data set over a global network. Out of the full keyserver network, servers that comply with a set of criteria for inclusion formed the *SKS keyservers pool*[1], from which all participant keyservers are available (Fiskerstrand 2016a, pp. 10–19). The pool also has several *sub-pools* available, grouping servers geographically to allow users to achieve lower latency while usign them. The SKS name is taken from the original software implementing both the keyserver and the synchronization components (Y. Minsky, Klizbe, and Fiskerstrand 2008).

This keyserver pool allows users to upload public keys as well as their certifications to any of the participating keyservers; users can upload information to the keyservers at any point, and this information is synchronized globally (Rose et al. 2016, p. 73).

The design of the keyserver network allows for easy peering and full federation: additional keyservers can be very easily peered with, requiring only one operator to agree to synchronize with them. Operators are invited to peer with several keyservers, to ensure network redundancy, allow for faster propagation times for new packets, and yielding lower synchronization sizes for most synchronization messages (Marshall 2015b).

While SKS is stable and tested software, various attacks –such as the one prompting this work, as well as legal challenges due to privacy-oriented regulations such as the European GDPR– have reduced the network's reliability. By June 2021, the operator for the keyservers pool decided to decomission the `sks-keyservers.net` domain name (Fiskerstrand 2021). The network continues to operate, although in much smaller numbers, as it is discussed in Subection 4.1.1. As this document is being written, the SKS keyserver operators are working to address the several shortcomings leading to this situation (Gallagher 2024a).

[1] `hkp://pool.sks-keyservers.net/`

2.5 Notable implementations

This section covers important implementations of concepts presented earlier in this work.

2.5.1 Transport Layer Security (TLS)

The most widely spread transitive trust scheme is Transport Layer Security, presently standardized at its 1.3 protocol version (Rescorla 2018), and formerly known as Secure Sockets Layer (SSL). This is a protocol family that has been introduced by Netscape Communications in 1996, with the stated goal of establishing secure connection channels between any arbitrary two parties. This, in a time where very few Internet protocols had cryptographic support, and taking several steps to ensure enough efficiency due to the (at its introduction time) high complexity, and thus, computational impact cryptographic operations have for server operations (Freier, Karlton, and Kocher 2011).

From its very conception, SSL, and subsequently TLS, includes trust establishment since the early connection stages, requiring the exchange of X509 certificates at the session establishment stage. In fact, six of the twelve defined alerts for the connection establishment phase are due to failures regarding certificate trust management (i.e. bad, unsupported, revoked or expired certificates). While work has been done on adequating the TLS protocol to be able to use OpenPGP certificates as well as X509 (Mavrogiannopoulos and Kahn Gillmor 2011), its use is explicitly forbidden in the TLS version 1.3 standard. This is, partly, because the lax enforcement the proposal places on WoT certificate verification (Rescorla 2018, Subsection 4.4.2):

> Considerations about the use of the web of trust or identity and certificate verification procedures are outside the scope of this document. These are considered issues to be handled by the application layer protocols.

It is important to note that, although the most user-visible use case of TLS is browser connection to a Web browser, it is an encryption layer that can carry any TCP-based protocol (Almohri and Evans 2017), or even via IP-over-TLS tunneling, to extend arbitrary networks, serving as the basis for many Virtual Private Network (VPN) products (Hosner 2004).

2.5.2 OpenPGP

The first widespread application to provide e-mail users with strong, public key cryptography is PGP (acronym of *Pretty Good Privacy*), released by Phil Zimmermann in 1991. Zimmermann 1999 cites as his main motivation the erosion of the right to hold private communications in the age of electronic surveillance. The software is presented particularly as an answer to legislative projects mandating communications providers to give government agencies the

ability to eavesdrop in private communications. This worldview continues to shape related its tools and standards.

Given that throughout the 1990s the United States' government regarded strong cryptography as munitions, PGP's distribution is carried out with its source code printed and bound as a book (Zimmermann 1995), so its dissemination would be protected as excercising the right to free speech. Since its first versions, PGP is released under a free software license, leading to its prompt adoption by the various groups that during the 1990s produce the different Linux-based free software distributions and projects. This mainly for its ability to produce cryptographic signatures, and thus, authenticate source code and mailing list messages for secure participation in globally distributed development teams. Adoption of PGP can also be seen in local free software communities, leading to the growth WoT networks, interweaving into a fully global WoT network (Darxus 2002).

Throughout the 1990s, the only implementation for PGP remains the program written by Zimmerman and its newer versions. Following the sale of source code rights for PGP to Network Associates in 1997, and subsequently to Symantec, the development of an official free version ceases. Given the widespread adoption PGP has had, by 1996 a standardization of its message format is adopted (Atkins, William Stallings, and Zimmermann 1996), allowing for third parties' implementations.

Since the Network Associates acquisition, several implementations have been developed. The most commonly used and held as a *de-facto* reference implementation is *GNU Privacy Guard* (GnuPG) (Rose et al. 2016, p. 11). It is used for identity and software authentication in various projects, and it is regarded as the standard for e-mail encryption (Angwin 2015).

While OpenPGP is very commonly used by unaffiliated individuals as well as several technical-oriented organizations, its use has understandably not spread to the corporate area, where S/MIME –based on X.509 certificates and a PKI-CA trust model– are a much more common choice. OpenPGP provides feature parity with S/MIME (Rose et al. 2016, pp. 11–13), but its use is explicitly not recommended in the United States federal government (Rose et al. 2016, p. 74).

The estimated direct and explicit usage of OpenPGP by 2018 is close to 550,000 active users, with a strong correlation of being white males with universitary education in industrially developed countries (Braun and Oostveen 2019). Braun points out that, even though PGP's original intent is to provide *encryption for the masses*, many factors (not the least of which is its usability) have prevented its adoption by others than a tech-savvy, socially privileged subset of the population.

Many more OpenPGP users, though, are *indirect* or *implicit*: several providers incorporate OpenPGP in their secure communications offering, with some caveats that are often frowned upon by the traditional OpenPGP community. One such example is *Proton*, which offers e-mail, VPN, cloud storage and other services, and claims to have over 100 million users (Proton AG 2024).

2.6 Attacks and weaknesses on transitive trust models

Both transitive trust mechanisms are susceptible to different weaknesses or targeted attacks. This section discusses several of such cases.

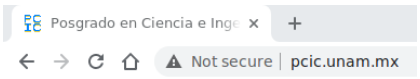
2.6.1 Lack of user understanding of the model

Although most users understand a TLS connection (i.e. using the HTTPS protocol) protects their *data in transit* by establishing a cryptographically secure channel, the protection it provides against a MitM attack (as described in Section 2.3) is much less known. Although there are proposals to include certificate chain of trust information among the user-visible security indicators of Web browsers (Herzberg and Jbara 2008), they have not been widely adopted. It has been shown that user understanding of the identity assurance granted by certificates is often shallow and easy to deceive; quoting Dhamija, J. D. Tygar, and Hearst 2006:

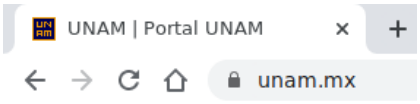
> Attackers can also exploit users' lack of understanding of the verification process for SSL certificates. Most users do not know how to check SSL certificates in the browser or understand the information presented in a certificate. In one spoofing strategy, a rogue site displays a certificate authority's (CA) trust seal that links to a CA webpage.

Furthermore, while users have long been advised to check for the use of *https* or for the padlock indicator signalling a Web connection is encrypted (see Figure 2.3), this practice is directly related to the fact that obtaining a valid certificate takes time and money, and that few attackers would invest in them — which is no longer true. Quoting Aas et al. 2019, "A major barrier to wider HTTPS adoption was that deploying it was complicated, expensive, and error-prone for server operators". However, the beginning of operations of the *Let's Encrypt* Certification Authority in 2015 directly results in TLS certificates becoming available immediately (after a simple and automated domain resolution verification) and at no cost.

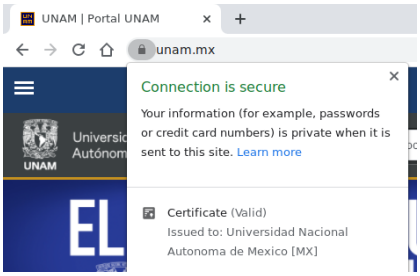
While the overall impact of *Let's Encrypt* is highly positive, in that there are no more barriers to setting up encrypted websites, it undoubtedly leads to the weakening of trust provided by the CA model. Less than a year after beginning operations, certificates requested with malicious intent (including for automated malware delivery, for *command and control* networks to which trojaned systems would connect and wait for instructions on when to launch an attack, or for *typosquatting*, registering domain names visually similar to established brand's, in order to redirect users to them and lure them into revealing their credentials) have been reported (Manousis et al. 2016; Carbone 2016). By 2018, half of all the *phishing* sites (sites built to trick users into submitting their login credentials or other personal data) have valid TLS certificates (Krebs 2018).



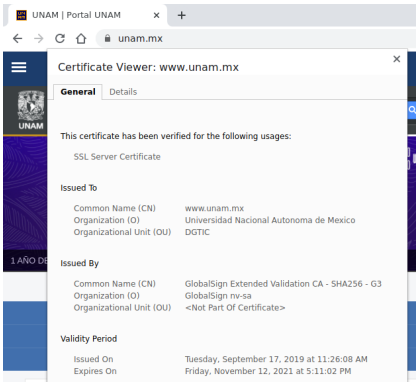
(a) In-browser indication of an insecure *http* connection: a *Not secure* indicator.



(b) In-browser indication of a secure *https* connection, without user intervention: only a padlock.



(c) Clicking on the padlock reveals the *connection is secure* dialog, emphasizing the connection is encrypted (but not explaining how the target site's identity can be trusted).



(d) Only a second click provides information regarding the CA that issued the certificate.

Figure 2.3: Getting to the trust information for the identity of a Web site requires user interaction; this simplifies browsing for the user, but makes the trust model much less visible. Screenshots taken from version 90 of the *Chromium* Web browser.

While the PKI-CA model includes several categories of validation for keys, and *Let's Encrypt* issues only certificates with the lowest validation certification –Domain Validation (DV)– it is shown that, despite visual cues, users are usually unaware of the difference between DV certificates and Extended Validation (EV) certificates (Thompson et al. 2019). Latest browser versions even drop those visual cues (O'Brien 2019; Hofmann 2019), rendering the distinction between DV and EV certificates even harder for end users to recognize (Hunt 2019).

2.6.2 Forgery or theft of CA keys

In the PKI-CA model, CAs are not only required to issue certificates, but if there is any evidence of a certified key being compromised, or rendered insecure due to cryptographic advances, they must also revoke the affected keys and list them both in a well known *Certificate Revocation List* (CRL) and via the *Online Certificate Status Protocol* (OCSP), at least for the duration of the original certificate (CA/Browser Forum 2021, Section 4.9).

CAs are also required to follow a strict physical and procedural security process to ensure the certification key is kept secure at all times (CA/Browser Forum 2021, Chapters 5 and 6). However, throughout the history of the PKI-CA model, and particularly given the use of TLS on the Internet, there are several high profile CA key management incidents.

Verification of certificate revocation The CRLs mechanism for verifying key revocation via CRLs does not scale following the Internet massification. This mechanism specifies a single CRL is to be returned upon client request by each CA, including all of the signed certificates still within their validity range that needed to be revoked (Gibson 2013).

By 2013, several CRLs had become up to 28MB long, with an exponential growth rate. Transfering 28MB only to check whether it is safe to establish a Web session introduces too much of a delay and penalty to network connections. By then, 44% of the revocations stated as their reason to be a key compromise — that is, either the server in question is breached, potentially exposing the certificate to hostile third parties, or errors handling the private key material that could have placed it on a vulnerable location (Gibson 2013).

OCSP, standardized in RFC 2560 and updated in RFC 6960 (Myers et al. 1999; Santesson et al. 2013), reduces the network load by specifying a protocol with which only the required certificate is fetched from the CA. However, given OCSP requests are to be generated at each connection to a new site, this also becomes an unreasonable load for the CA servers. Furthermore, privacy activists point out it discloses user browsing habits by giving detailed trails as to precisely which Web sites they usually visited.

A further problem both with CRLs and OCSP is the client's action in case of certification server's unavailability — what happens if the status of a certificate as revoked or valid cannot be checked? The client can either *fail open* (assume the certificate remains valid, as it is the most usual case, exposing the client to MitM attacks) or *fail closed* (assume the certificate is invalid, assuring protection against MiTM, but probably causing a DoS). None of those approaches is good for all cases (Berkowsky and Hayajneh 2017).

The standardization of TLS version 1.2 (Eastlake 2011) introduces the *status request* extension, with which the server can send to the client a timestamped OCSP response as part of the session establishment, a practice known as *certificate stapling*.

However, although what has been reviewed in this section so far seems to cover the issue of properly handling and revoking keys under the PKI-CA model, the issue of CA key management still has one more troubling angle.

Breach of trust in CA keys Certification Authorities need to be extremely careful on their practices regarding their certification keys. The CA/Browser Forum periodically updates the Baseline Requirements for the Issuance

and Management of Publicly-Trusted Certificates since 2011 (CA/Browser Forum 2021, pp. 9–16), stating:

> This document describes an integrated set of technologies, protocols, identity-proofing, life-cycle management, and auditing requirements that are necessary (but not sufficient) for the issuance and management of Publicly-Trusted Certificates; Certificates that are trusted by virtue of the fact that their corresponding Root Certificate is distributed in widely-available application software.

Nevertheless, various incidents highlight the need for ensuring the CAs actually comply with the practices presented in the baseline requirements. Although CAs are expected to act swiftly and openly, that is very often not the case due to the loss of trust such an admission implies (Berkowsky and Hayajneh 2017).

In August 2011, Iranian users are presented with a rogue certificate for the **.google.com* wildcard domain, signed by the Dutch CA *DigiNotar* (Nightingale 2011). An independent preliminary analysis showed DigiNotar had been breached due to poor security practices, and that, given the weaknesses in the implementation of the PKI-CA ecosystem as used by Web browsers, the depth of the breach required patching the list of trusted CAs in every Internet-connected device (Prins 2011).

There are reports of close to 50 fraudulently issued certificates related to this incident, many of them for high-profile domains. DigiNotar marks the first case for a a CA to be outright removed from browsers' root of trust key directory, highlighting the lack of care this and several other CAs had (Amann et al. 2017).

A similar, although more worrying, case happened few months later. The *Trustwave* CA knowingly produces a *subordinate root certificate* allowing their client to create and sign arbitrary certificates (Espiner 2012). This incident, again, results in manually patching of all CAs across deployed systems.

In 2015, a certificate for GitHub iss found to be issued by the Chinese CA WoSign, without properly checking the site ownership, allowing for potential MitM attacks. Upon further inspection, many more issues come up with this CA. These include the continued issuance of cryptographically weak SHA-1 certificates after the cutoff date where the CA/Browser Forum agreed not to issue them, deliberately *backdating* them so that browsers would still accept them. Thus, issuing many more spurious certificates and refusing to revoke them when they have been found, and even lying about the cause for such certificates, revealed upon the audit of their logs. Even having all of that documented evidence, the time required to drop WoSign from the list of roots of trust for the main browsers has been over 18 months, because of the number of existing certificates that hae

been still under legitimate use (Berkowsky and Hayajneh 2017; Wilson 2016).

In 2015, researchers find that OEM versions of Windows installed in Lenovo computers include a self-signed certificate as part of their roots of trust. The purpose of this spurious certificate is for the *SuperFish* program (also part of the pre-installed operating system image) to intercept browser sessions, performing a MitM attack, and inserting advertisements in the users' browser (Berkowsky and Hayajneh 2017; Lenovo 2015).

Private keys should never be disclosed to a third party, they must be closely guarded by a keypair owner. In 2018, a statement by the Trustico intermediary CA, operating under the DigiCert root certificate, mentions they stored the private keys for some for the certificates they issued (allegedly as a way to protect their users). In an attempt to migrate to the Comodo root, over 23,000 private keys they held are mailed to DigiCert. This led to the keys being immediately revoked, causing all of their customers the need to generate new keypairs and have them re-certified (Varghese 2018).

In 2013, the *Heartbleed* vulnerability is disclosed triggering a massive set of revocations. Several different implementations for the use of TOFU-based schemes for TLS security were proposed (Toth and Vlieg 2013), but none of them managed to get enough user traction to be adopted beyond a small group of propsers.

Naturally, the presented recounts for both issues present several examples of the issues, and are not exhaustive. Their inclusion attempts to highlight the fact that TLS security, based on the PKI-CA centralized trust distribution model, although has been widely adopted for Internet-based communications, is by no means exempt of issues and challenges.

Important contributions in this regard come to light over the past decade; worth highlighting is the move away from the issuance of long-lived certificates (one to four years) that has been the industry standard, to *few days* (precise number not stated in the proposal — Topalovic et al. 2012).

The emergence of the *Let's Encrypt* CA above, together with tools for automatically querying and keeping updated server-side certificates, points in the direction presented by Topalovic et al.; certificates signed by *Let's Encrypt* have three months long validity periods.

2.6.3 The user interface is to blame: *Evil32*

One of the main critiques of the OpenPGP ecosystem is the poor usability of its tooling — so much, it has repeatedly become a staple of bad user interface design (Whitten and J Doug Tygar 1999; Sheng et al. 2006; Woo 2006; Ruoti et al. 2015). A fundamental aspect that many users apply incorrectly throughout the years is key verification after acquiring it from a public key server. This is, after *Alice* looks up *Bob*'s public key from a public keyserver, how to choose

the right key? Even in the straightest case and without any attackers in play, *Bob* might have homonyms on the network (although the mail address cannot be the same), and he might used have several keys in the past, revoked or in disuse for different reasons.

OpenPGP keys are often indexed and referred to using their *fingerprint*, the hexadecimal representation of the 160-bit SHA1 *hash* of the full public key (Callas et al. 2007, p. 71) (see Figure 2.4). Still, this representation is 40 characters long (in the referred figure, the line starting with `4D14` and finishing with `5360`), impractical for person-to-person communication, or even verification.

```
$ gpg --fingerprint gwolf@gwolf.org
pub   ed25519 2019-11-22 [SC]
      4D14 0506 53A4 02D7 3687  049D 2404 C954 6E14 5360
uid           [ unknown] Gunnar Wolf <gwolf@gwolf.org>
uid           [ unknown] Gunnar Wolf <gwolf@debian.org>
uid           [ unknown] Gunnar Eyal Wolf Iszaevich <gwolf@iiec.unam.mx>
sub   ed25519 2019-11-22 [A]
sub   ed25519 2019-11-22 [S]
sub   cv25519 2019-11-22 [E]
```

Figure 2.4: Obtaining the fingerprint of a public OpenPGP key

For many years, it has been customary to refer to key material by just using the *short key ID* (a truncation of just the last 32 bits of the fingerprint, `0x6E145360` for the presented example, Wolf 2016). *Bob* can thus include his short key ID as an epigraph on all of his mails, and *Alice* can refer to any of them to verify she has the right key to communicate with. Although *Long key IDs* (a truncation of the last 64 bits of the fingerprint) are used for applications that need to manage large volumes of cryptographic keys, such as curated keyrings for large projects (Wolf and Gallegos 2017), they are not deemed as human-memorable and human-useful as short key IDs — the long key ID for the above mentioned key would be `0x2404C9546E145360`, clearly more identifiable than the full fingerprint, but far from being convenient or memorable for the end user.

However, 32 bits is a very short address space. Given true randomness and no hostile intentions, collisions would happen with only with $\frac{1}{2^{32}}$ probability. As computing power grows, by the means of calculating many keys, *vanity short IDs* are generated by several OpenPGP users, as shown in Figure 2.5.

```
$ gpg --fingerprint 0x29C0FFEE 0x00000011 0xFFFFFFFE | grep -v uid
pub   rsa4096 2009-05-18 [SCEA]
      9590 8AA6 E4F7 BAA7 8BD6  C148 F1A6 9BE4 29C0 FFEE
sub   rsa4096 2009-05-18 [E]

pub   rsa4096 2014-07-21 [SCEA] [revoked: 2016-08-16]
      CA9D 22A6 25C6 B346 B4EB  04D5 155B 8E51 0000 0011

pub   rsa4096 2009-10-23 [SC]
      5C80 9377 432D 6157 DDF5  2D5D 055C 4B35 FFFF FFFE
sub   rsa4096 2009-10-24 [E]
sub   elg4096 2009-10-24 [E]
```

Figure 2.5: The increase in computing power allowed users to create vanity short key IDs (person-identifying UIDs omitted).

Generating vanity OpenPGP keys by scripting calls to the GnuPG binaries

still requires an overly large amount of work. In 2012, with the announcement of the *Scallion* tool (Klafter and Swanson 2012), GPU computation can easily be used to create vanity OpenPGP keys by means of abusing the *protocol* without the need for brute-forcing the full problem space, increasing instead one of the parameters, and selecting from its output where it matches the desired string. As a means for the demonstration of this tool, the authors make use of Scallion to create keys whose short key IDs collided with the corresponding ones for *all of the keys* in the OpenPGP WoT's *strong set* (keys that had at least one trust relationship to and from the largest set of keys in the WoT), publishing it under the *Evil32* moniker. Not only that, the generated keys mimicking the trust relationships existing in the WoT. For a user checking only short key IDs, *Evil32* keys are discernible only because all of the keys and trust signatures are created on the same date, as shown in Figure 2.6.

```
pub  4096R/C1DB921F 2014-06-16
sig revok  C1DB921F 2016-08-16 __________ __________ [selfsig]
         Hash=60B6A5031028BBDBB8FABEB220B92CCA
         Fingerprint=8B3D DC72 9A27 98BC 9BE9  491D 15FB FDED C1DB 921F

uid Gunnar Eyal Wolf Iszaevich <gwolf@debian.org>
sig  sig3  C1DB921F 2014-08-05 __________ __________ [selfsig]
sig  sig   DD49F17B 2014-08-05 __________ __________ Michael Shuler <mic
sig  sig   DC814B09 2014-08-05 __________ __________ Javier Fernández-Sa
sig  sig   12066207 2014-08-05 __________ __________ Ben Hutchings <ben@
sig  sig   3170EBE9 2014-08-05 __________ __________ Nobuhiro Iwamatsu <
sig  sig   575D0A76 2014-08-05 __________ __________ Martín Ferrari <tin
sig  sig   E4C9123B 2014-08-05 __________ __________ Adrián Pérez de Cas
sig  sig   EAC68A14 2014-08-05 __________ __________ Rafael Martinez Gue
sig  sig   A3623899 2014-08-05 __________ __________ David Bremner <brem
sig  sig   B7CDA2DC 2014-08-05 __________ __________ Max Vozeler <max@nu
sig  sig   C0143D2D 2014-08-05 __________ __________ Christian Perrier <
sig  sig   B6250985 2014-08-05 __________ __________ Andrew Lee (李健秋)
sig  sig   8A1D9A1F 2014-08-05 __________ __________ John Goerzen <jgoer
sig  sig   1C00C790 2014-08-05 __________ __________ Mehdi Dogguy <doggu
sig  sig   556ABA51 2014-08-05 __________ __________ Francisco Manuel Ga
sig  sig   1880283C 2014-08-05 __________ __________ Anibal Monsalve Sal
sig  sig   4A2F162B 2014-08-05 __________ __________ Axel Beckert (ISG D
```

Figure 2.6: Not only a collision between short IDs is proven, but the first UID and the trust relationship between keys are mirrored as well to the *Evil32* set.

The *Evil32* experiment emphasizes on the fact that this is not a vulnerability of OpenPGP, inasmuch keys are not breached and there be no way of confusion if keys are used *correctly*; it is only the short key ID *notation* that urgently needed to be migrated away from. As can be seen in this work's author's case (see Figure 2.7), while both keys match the `0xC1DB21F`, the legitimate key is `0x673A03E4C1DB921F`, and the corresponding *Evil32* key is clearly different — `0xFBFDEDC1DB921F`.

Quoting the authors of the *Evil32* project's Web page, "Aren't you supposed to use the Web of Trust to verify the authenticity of keys? Absolutely! The web of trust is a great mechanism by which to verify keys but it's complicated. As a result, it is often not used" (Klafter and Swanson 2014).

While the *Evil32* experiment is undoubtedly an adversarial attack, it must be emphasized it has not been carried out with malicious intent, but as a way

```
$ gpg --fingerprint 0xC1DB921F
pub   rsa4096 2009-07-09 [SC] [expired: 2020-12-19]
      AB41 C1C6 8AFD 668C A045  EBF8 673A 03E4 C1DB 921F
uid           [ expired] Gunnar Eyal Wolf Iszaevich <gwolf@debian.org>
uid           [ expired] Gunnar Eyal Wolf Iszaevich <gwolf@gwolf.org>
uid           [ expired] Gunnar Eyal Wolf Iszaevich (Instituto de Investigacione
s Económicas UNAM) <gwolf@iiec.unam.mx>

pub   rsa4096 2014-06-16 [SCEA] [revoked: 2016-08-16]
      8B3D DC72 9A27 98BC 9BE9  491D 15FB FDED C1DB 921F
uid           [ revoked] Gunnar Eyal Wolf Iszaevich <gwolf@debian.org>
```

Figure 2.7: Result of querying the keyservers for the `0xC1DB921F` short ID: Two colliding keys with the same first UID. The second key returned belongs to the *Evil32* set.

to prove beyond any doubt the widespread use of short key IDs is no longer secure.

Finally, cryptographic material, derived from random sources, simply cannot be presented in a user friendly way. Kahn Gillmor 2013 claims the use of key IDs should be abandoned altogether, and either replaced by human-meaningful values when presented to users, or with the full 160-bit key fingerprint within scripts and other automated tools. Just as short key IDs are just an arbitrary truncation that greatly diminishes the strong key unicity requirements and allows for exploits derived from wrong identification, long key IDs still present a vulnerable scheme for key identification, but also lose whatever human memorability short key IDs provided.

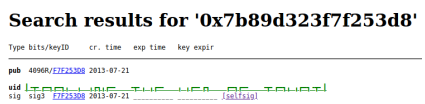
Non-targeted long key ID collisions (this is, collisions not attempted as a preimage attack on a given value, just random values found to collide) have been achieved, and due to being basically a *birthday attack*, can be achieved by the generation of an average of 2^{32} keys (Gil 2013; Skeeto 2019; Wellons 2019). It is still, however, computationally unfeasible to perform attacks similar to the *Evil32* set using 64-bit identifiers.

Nevertheless, *Evil32* draws attention to a simple yet powerful problem: if OpenPGP users *are supposed to* verify the WoT when establishing contact with a new user but in practice do not do so, said users are most likely to fall prey to any impersonation. While formal literature does not yet collect the term, in informal communications this has been named *UID poisoning*: uploading rogue identities to the keyservers with invalid UIDs, corresponding either to nonexisting users, or leading to the equivocation of users attempting to contact legitimate users. The *Evil32* attack is a prime example, but it is far from the only available one; an example of hiding compromised data by attempting equivocation (Lawrence 2018) can be traced on the GnuPG mailing list, where it provoked a very interesting discussion about a way to achieve removal of key information from the keyserver network.

2.6.4 Key UID information for storing arbitrary information

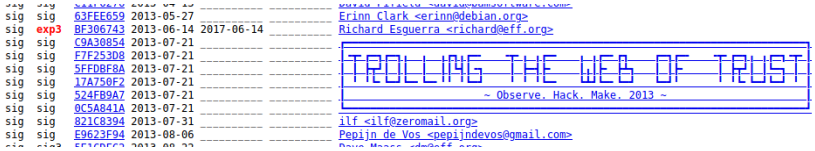
The design of the SKS keyserver network presented in Subsection 2.4.1 is explicitly designed to be resistant to a threat model including national governments' interference or censorship (Hansen 2018). OpenPGP data consists on a set of concatenated data packets following some simple ordering rules (Callas et al. 2007, p. 68). Keyservers rely on a *gossip*-based set synchronization protocol, so the keyserver network accepts new keys and updates on existing keys, but no information can be realistically removed from it — any server still carrying a packet not found at other servers injects it back into the network. The protocol provides no provision for removing unwanted information. This issue has carried legal issues for keyserver operators, being unable to respond to deletion requests based on national privacy laws newer than the implementation (Pramberger 2010). The adoption of the General Data Protection Regulation (GDPR) by the European Union in 2016 makes this issue even more important.

The OpenPGP message format standard specifies keys should have a User ID packet that "consists of UTF-8 text that is intended to represent the name and email address of the key holder. By convention, it includes an RFC 2822 mail name-addr, but there are no restrictions on its content" (Callas et al. 2007, p. 48). This is, there is no enforced format on the contents of User ID strings. An attack on this model is, then, the mass creation and upload of non-meaningful keys, using the WoT as some way of *graffiti pad* (Lee 2014). An example is presented by Lee at the *Observe. Hack. Make. 2013* hacker gathering: by generating one OpenPGP key per line with arbitrary ASCII characters in the field regularly used for the owner's user ID (see Figure 2.8(a)), he signed his own OpenPGP key (Figure 2.8(b)), resulting in the graffiti depicted in Figure 2.8(c)



(a) One of the throwaway identities created for the "Trolling the Web of Trust" attack

(b) Beginning of the listing for the author of the attack: a regular vanity key



(c) Down the signatures list, the *attack* is shown as a multiline graffiti

Figure 2.8: Example of the *Trolling the Web of Trust* attack

The impact of this attack *per sé* is not very high: the creation of some throwaway identities, and the use of the WoT in a way other than intended,

but it highlighted a huge potential for abuse. But in 2018 –again, as a proof of concept, showing it would be impossible to properly take action against a GDPR request for information deletion (Yakamo 2018a)– a program is published that enabled the encoding of arbitrary information as key and certifications material (Yakamo 2018b; Yakamo 2019), allowing the abuse of the SKS network for arbitrary file storage. While this attack has not been widely observed, its effects can be devastating on the keyserver network, as it potentially becomes a distributed, append-only media, with no content removal facilities. If files deemed illegal to be possessed are to be uploaded in this way, this could make many server operators to shut down their servers, with a reasoning similar to Pramberger 2010.

2.6.5 Lack of use of the trust model

An important weakness can be found in the way users have adopted the use of OpenPGP: whether its users *care about* the WoT, or they use OpenPGP implementations only for encrypting communications to out-of-band-verified identities (if at all). This issue does not threaten the validity of the WoT, but does shed a light into its true size.

In November 15, 2018, the HKP keyserver network consisted of 5 217 474 certificates, which are made of public keys and their cross-certifications (Yakubov et al. 2020). The number of signatures each of those certificates, however, present a figure very different to what *Web of Trust* could convey, as seen in Table 2.1: a very large majority (84%) of existing keys exist in isolation and are not linked to any other key. Graphing the amount of keys in relation to the amount of signatures each of them has yields an asymptotic progression, as can be seen in Figure 2.9.

Table 2.1: The distribution of OpenPGP certificates based on the number of signers on 2018.11.15) (Yakubov et al. 2020)

No. of Signers	No. of PGP Keys	Percentage
0	4394932	84.23%
1	424815	8.14%
2	131911	2.53%
3	58944	1.13%
4	37882	0.73%
5	20632	0.40%
6	18302	0.35%
7	12039	0.23%
8	11514	0.22%
9	9150	0.18%
10	7933	0.15%
>10	89420	1.71%

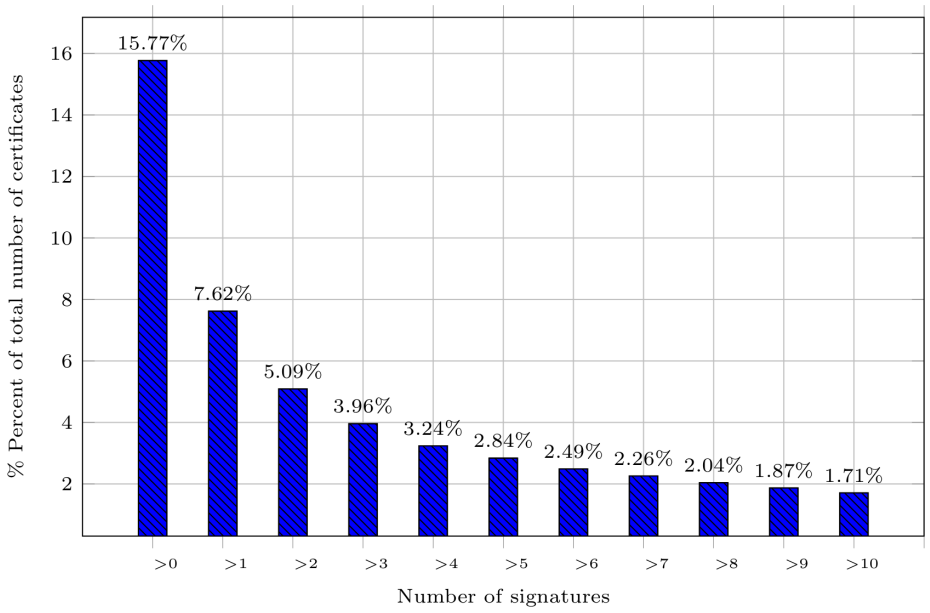


Figure 2.9: Percentage of OpenPGP certificates with third-party signatures greater than the specified amount (on 2018.11.15)) (Yakubov et al. 2020)

Many keys only certify one another, so not all of the 424 815 keys are *reachable* from each other: the graph's strong set is comprised of only 60 000 certificates, this is, around 1.1% of the total number of those in keyserver storage. This is, 99% of users of OpenPGP identities are unlikely to be able to benefit from the WoT.

While discussing this point, it must be pointed out that by far not all of the 5 million keys in the keyserver storage are in active use: the data set still comprises 30 years of history, and it does include many expired, revoked, and even valid-but-forgotten keys.

2.6.6 Certificate poisoning

Certificate poisoning attacks are special cases of what the two above subsections present — paired with a more disruptive, malicious intent.

In July 2019, two keys have been reported to be *poisoned* in the SKS keyserver network, exploiting a weakness in the OpenPGP standard: it defines a packet format in which signatures are simply padded to an existing certificate, which only reach a limit when approaching 150,000 certifications. Client programs that *implement* OpenPGP are not able to handle this amount of data. Two prominent people in the OpenPGP community, Robert J. Hansen and Daniel Kahn Gillmor, have had their certificates suddenly certified by close to 150,000 throwaway identities. This means, attacked certificates, usually not bigger than a couple hundred kilobytes for very well-connected participants, have

grown suddenly to tens of megabytes — a thousandfold increase (Hansen 2019).

When GnuPG (as well as other OpenPGP implementations) attempts to use any poisoned key, it exhibits severe performance degradation, and might lead to data corruption in its keystore (Kahn Gillmor 2019b).

2.7 Summary

This chapter presents the main concepts in the computing field that frame the work this document presents. The main focus of this work is distributed trust in identities asserted over an encrypted communication channel; thus, the first topics presented are private and public key cryptography (Subsections 2.2.1 and 2.2.2). After establishing this, trust distribution models are presented: the more common, centralized *PKI-CA* model (Subsection 2.3.1), followed by the distributed *Web-of-Trust* model (Subsection 2.3.2. A third model, *Trust On First Use*, albeit formally a trust assignment rather than trust distribution model, is presented in Subsection 2.3.3. A short discussion on key distribution follows, particularly centered in the HKP public keyserver network, as it is central to the problem this work targets (Subsection 2.4.1).

In order to describe the attacks and mitigations trust models face over time, it is deemed necessary to present the most common protocols that implement the models presented so far. Thus, Subsection 2.5.1 introduces *Transport Layer Security*, bases its trust on the PKI-CA model, and Subsection 2.5.2 presents *OpenPGP*, the most widely deployed implementation of the WoT model.

The chapter finishes by discussing several inherent weaknesses and explicit attacks, either on the trust models or on their main implementations. First, the important aspect of user understanding is presented in Subsection 2.6.1; transitive trust models deal after all with conveying trust information, and trusting a system's identity is a task that ultimately needs human responsibility, and if left as a purely automated task yields sub-optimal results. Subsection 2.6.2 presents several examples on how highly-trusted root-of-trust keys are leaked or stolen, and how this has led to reshaping the PKI-CA ecosystem. 2.6.3 presents an attack on the WoT model, where the common use of a weak certificate identification method results in the ability of an attacker to easily create large sets of lookalike certificates, granting the ability to fool a careless user. Subsection 2.6.4 presents an attack that, while not very important by itself and that could just be considered a *prank*, made obvious the potential for abuse in the OpenPGP and HKP protocols. Subsection 2.6.6 deals with the central problem this work sets to work on: certificate poisoning, the upload of certificates *bloated* with so many signatures they end up rendering affected keys unusable, and lead to endangering the continuity of the HKP keyserver network.

Chapter 3

Related work

As attacks on the WoT and on the keyserver network have surfaced, defenses or mitigations have also been developed. This chapter presents other ideas to solve the presented issues, highlighting their most important characteristics in the specific problem space this work targets.

This chapter is structured as follows: Section 3.1 references to a full enunciative review of issues and strategies in OpenPGP keystores; Section 3.2 presents works that recognize the problem in OpenPGP's ecosystem in a fundamental level, proposing instead alternative ways for public key distribution abandoning the WoT for trust metrics, and embracing other mechanisms; Section 3.3 refers to works that have searched for patterns or weaknesses on the full set of keys present in the keyservers; Section 3.4 reviews implementations of keyservers that, by changing specific aspects of a HKP keyserver while maintaining compatibility in others, become better equipped to resist or disable vulnerabilities such as those outlined in Section 2.6; finally, Section 3.5 presents articles that assume OpenPGP as a protocol fit for a threat model no longer valid 30 years after its introduction, and present threat models and cryptographic configurations allegedly better suited for present-day communications.

3.1 Abuse-Resistant OpenPGP Keystores

A clear starting point for this chapter's discussion is Kahn Gillmor 2019a. While this cannot be seen as a finalized document, as it is presented as a RFC candidate in the IETF in April 2019 and updated through five revisions, it has lapsed out during the RFC process, so it remained as an expired draft since August 2019. Nevertheless, it is the clearest work presenting a thorough review of the issues keystores face nowadays, as well as different strategies to counter them.

The author begins by introducing the terminology to use, and states the problem as follows (Kahn Gillmor 2019a, p. 5):

> Many public keystores allow anyone to attach arbitrary data (in the form of third-party certifications) to any certificate, bloating that

certificate to the point of being impossible to effectively retrieve. For example, some OpenPGP implementations simply refuse to process certificates larger than a certain size.

This kind of Denial-of-Service attack makes it possible to make someone else's certificate unretrievable from the keystore, preventing certificate discovery. It also makes it possible to swamp a certificate that has been revoked, preventing certificate update, potentially leaving the client of the keystore with the compromised certificate in an unrevoked state locally.

Additionally, even without malice, OpenPGP certificates can potentially grow without bound.

After the problem statement, this document addresses several ways in which it can be mitigated: it begins with simple mitigations that can be implemented upon receiving any new packet, based only on the structure of said packet; progresses towards presents contextual mitigations, this is, packets might be dropped from the keystore as new information is received (i.e. stored information being superseded) or time passes (such as signatures expiring); it then describes first-party-only keystores (allowing users only to use such a keystore to make information on themselves available, not on third parties). It presents the concept of First-party-attested Third-party Certifications (the keystore only accepts third-party certification packets for a given key if they are signed off by the affected key). Further sections address side effects on the OpenPGP tooling and ecosystem of some of the presented ideas as they are currently implemented, and some privacy considerations of publicly-accessible keystores.

It is worth emphasizing that this document is mereley enunciative of different ways the stated problem could be tackled, but does not implement or analyze in detail any of the presented alternatives. It is, however, a very valuable starting point against which other analyzed works can be measured to.

3.2 Key discovery mechanisms

One of the main issues that lead to this work is the need for a robust key discovery and distribution mechanism (described in Section 2.4). Other key discovery and distribution methods devised to counter some of the problematic characteristics of the public keyserver are now presented.

While the main goal of this work is to preserve the *positive* characteristics of the public keyserver scheme, alternative distribution proposals deserve our attention.

Proposals such as *DNS-Based Authentication of Named Entities* (DANE; see Subsection 3.2.1) and *Web Key Directory* (WKD; see Subsection 3.2.2) can solve several of the issues that WoT suffers, although they imply shifting the trust model to the much laxer *Trust On First Use* (TOFU; see Subsection 3.2.3). This strategy is adequate for some use cases, as presented by Koch and Walfield, but

Section 2.3.3 discusses others where the stronger guarantees of a WoT model are still required (Koch 2016; Walfield and Koch 2016).

Here we present some of the main implementations. The main properties and mechanisms are described in Table 3.1

3.2.1 DNS-based Authentication of Named Entities (DANE) for OpenPGP

As OpenPGP lacks a canonical lookup mechanism to securely obtain public keys, the DANE proposal presents a method for publishing public keys as part of the DNS records for the domain where a given mail address is hosted. DANE is standardized for its use on the TLS protocol in RFC 6698 (Hoffman and Schlyter 2012), and for OpenPGP in RFC 7929 (Wouters 2016).

The core logic behind this proposal is that a user A should search for a user B's public key not using the keyserver network (discussed in Subsection 2.4.1), but via a DNS lookup on B's e-mail service provider. That is, if B's mail address is *bob@example.org*, the corresponding public key should be served in a specific record by *example.org*'s domain name server. Figure 3.1 shows an example query on the *gwolf@debian.org* mail address.

```
✓ gwolf@uesebe ~ [1] $ Q=gwolf
✓ gwolf@uesebe ~ [2] $ dig TYPE61 $(echo -n $Q|sha256sum|cut -c -56)._openpgpkey.debian.org

; <<>> DiG 9.16.15-Debian <<>> TYPE61 d65f0b02d8e07704b1c600836ff37ed2a2d789f5b2e2a47b2ddb1a36._
openpgpkey.debian.org
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 44978
;; flags: qr rd ra; QUERY: 1, ANSWER: 1, AUTHORITY: 0, ADDITIONAL: 1

;; OPT PSEUDOSECTION:
; EDNS: version: 0, flags:; udp: 512
;; QUESTION SECTION:
;d65f0b02d8e07704b1c600836ff37ed2a2d789f5b2e2a47b2ddb1a36._openpgpkey.debian.org. IN OPENPGPKEY

;; ANSWER SECTION:
d65f0b02d8e07704b1c600836ff37ed2a2d789f5b2e2a47b2ddb1a36._openpgpkey.debian.org. 27710 IN OPENPG
PKEY mDMEXdg0IBYJKwYBBAHaRw8BAQdAf5wnwSKKCiZg9GO/51fyLJGmdlxP yOPmdgQw2rkRMn2OHkd1bm5hciBXb2xmID
xnd29sZkBkZWJpYW4ub3Jn PoiWBBMWCAA+AhsDBQsJCAcCBhUKCQgLAgQWAgMBAh4BAheAFiEETRQF BlOkAtc2hwSdJATJ
VG4UU2AFAl90wNwFCQdAJzsACgkQJATJVG4UU2BO BQD/SXVJneseUVkHvQ4aa+dUaHSNRCpx74/k5IZEHuQBVIIBAMFCFv6
P hkRmlRAM4Lv042JZC1giFMBvrl6z6f3+vGIMuDMEXdg0thYJKwYBBAHa Rw8BAQdAP8wr/Hhc1RBgPe2pfQA+oxYwpZcpU
Xv6Z9bRvcJMM3aIfgQY FggAJgIbIBYhBEOUBQZTpALXNocEnSQEyVRuFFNgBQJfdMFCBQkDfcAM AAoJECQEyVRuFFNgiZQ
BAP5AEUvBEmua0PX1euzLNsHqCbMqXquDzOCR Vp6/bFGPAQDtvSMNWPMfOI4RsZWrwTUQaOGEy3ZzVy68MoODl4JuArgz B
F3YaCwWCSsGAQQB2kcPAQEHQHSImNxrfmXaaJgbkl4zBPuvYTzrVbIT f1ExXKw23o7MiPUEGBYIACYCGwIWIQRNFAUGU6QC
1zaHBJOkBM1UbhRT YAUCX3TBQwUJA32MlgCBdiAEGRYIABOWIQRgswk91hCOXLlxQu/i9jtD U/RZiQUCXdhoLAAKCRDi9j
tDU/RZiefCAQD7gtQq0T3gPafUePO9Zmks pP2IURgsKA8rSKIEmazNaQD+Ic5hZbC+dd5Hp2vrxSeKkmMi0GavIKcz NO0v
hHwBKQwJECQEyVRuFFNgq/UA/Am3L7DxqOLigZFBUbKpJfWJG4oC bWyOhdBNnubl1AgqAP9TUNO1fRLuY+8V+TUezmsi6sW
bDPBvZlZDQAgc C9CHB7g4BF3YNCASCisGAQQBl1UBBQEBBOA2vNYKCM9LD7hOFnXLtXh1 bOPtxDLsD5ieON3r3VopQgMBC
AeIfgQYFggAJgIbDBYhBEOUBQZTpALX NocEnSQEyVRuFFNgBQJfdMFCBQkDfcCiAAoJECQEyVRuFFNgxtEBAJr9 GTnGF4e
sHedBGvFVZaZb47CSSj4HRZOj3pr7XZowAQCwpIW9m1bTgwYu MvbMWDvcuRcxYi8whCmRX0cV8h5bBw==
```

Figure 3.1: Example DANE query for the `TYPE61` DNS record with the public key for *gwolf@debian.org*.

Wouters 2016 recognizes the size of the DNS records has been criticized as too large. This critique is partly mitigated by striping away from the served signature all other user identities the key might have as well as all the certifications it carries.

While DANE interoperates with unmodified, existing OpenPGP installations and does solve the problem of poisoned certificates, it does so by ignoring and eliminating the WoT altogether (as third party signatures are stripped from

Table 3.1: Key discovery mechanisms comparison (Kahn Gillmor 2016)

Mechanisms → / **Properties ↓**	Key-in-email	WKD	DANE	Keyserver	Mailvelope	Symantec	Keybase	CONIKS
Passive / Active	P	A	A	A	A	A	A	A
Talks to 3rd party	✗	Provider	Provider	✓	✓	✓	✓	Provider
Central Authority	✗	✗	✗	✗	✓	✓	✓	✗
Network protocols	SMTP	HTTPS	DNSec	HKP/HKPS	HKPS/HTTPS	LDAP	HTTPS	HTTPS
User invisibility	✓	✓	✓	✓	✓	✓	✓	✓
Delay	1 round trip	TCP	UDP	TCP	TCP	TCP	TCP	TCP
Ambiguity	✗	✗	✗	✓	✗	✗	✗	✗
Revocation discovery	?	✓	?	✓	✗	✓	?	?
Append-only	✓	✓/✗	✗	✓	✗	✗	?	✓
Consistent-globalview	✗	✗	✗	✗	✗	✗	✗	✓
Cacheability	✗	✓	✓	✓	✗	✓	?	✓

Mechanisms

- Key-in-email sent as an attachment
- WKD HTTPS lookup by email address, described in Subsection 3.2.2
- DANE, DNS-based Authentication of Named Entities, described in Subsection 3.2.1
- Keyserver lookup (SKS pool)
- Mailvelope Keyserver
- Symantec Global Directory
- Keybase
- Google's Key Transparency (CONIKS)

Properties:

Passive / Active Whether the lookup is initiated by the communication party who seeks to discover the key

Talks to 3d party Divulges the e-mail address to a third party

Central Authority Are there centralized authorities for queries?

Network protocols Protocol used for key discovery

User invisibility Can the lookup be hidden from the user?

Delay How long is the delay to retreive the key

Ambiguity Whether it is possile to get more than one key upon lookup

Revocation discovery Can the sender find if a key has been revoked?

Append-only Whether it is a property of the mechanism

Consistent global view Will end-users get the same results regardless of the point they request their key from?

Cacheability Can the same trust properties be achieved over a proxy?

the served certificates). DANE tends heavily towards a centralized model: all of the users for a given mail provider are required to send the keys to the site administrators for them to assemble the needed information in the DNS zone files. While this process can be automated, as is the case in the debian.org example presented in Figure 3.1, it clearly leads to a single point of failure (as well as a single point of potential censorship) for key discovery and distribution. For users of massive mail providers such as *gmail.com* or *hotmail.com*, which sums over half of all mail user globally (Gilbert 2021), it is plainly impossible to get their keys published.

3.2.2 OpenPGP Web Key Directory (WKD)

The above mentioned proposal has the issue of practicity: DNS zone editing to serve DANE records is beyond the abilities of many small service operators; it is a centralized task to be done for each domain, and due to the opacity of the changes made to the DNS zone configuration file, it can potentially lead to service disruption in case of mistakes. Furthermore, DNS lookups are not encrypted, and a passive attacker *sniffing* network traffic can note which OpenPGP keys are requested by which IP addresses, leading to privacy risks.

Koch 2021b proposes a key discovery method using encrypted HTTPS connection. WKD is meant to be installed per domain on a stable, discoverable URL: all keys belonging to users with mail addresses under the `example.org` domain would be found under:

```
https://openpgpkey.example.org/.well-known/
      openpgpkey/example.org/hu/
```

The mail address' local part (the username at `example.org`) is specified as the last component of this URL, after being case-normalized, hashed under the SHA-1 algorithm, and encoded using Z-Base-32; for *someuser@example.org*, the final part of this URL would be:

```
hfh6c7pfzr3uop5ne7qrdwj4uo6hr49p
```

Koch 2021b addresses key discovery, but –as is the case with Wouters 2016– leans towards a TOFU scheme, and serves the keys stripped from certificates altogether.

While Koch 2021b is *mostly centralized*, it includes sections on how users can update their records via the *Web Key Directory Update Protocol* (presented as part of the same work), enabling a *more autonomous* –although not decentralized– operation.

Thus, while WKD addresses the main problem this work sets to (poisoned certificates), it does so by ignoring the properties this work attempts to preserve: fully decentralized operation of the Web of Trust.

Finally, while Koch 2021b is still an IETF Internet Draft and has not yet reached standardization in the form of a numbered RFC document, at least a dozen free software projects have adopted it (Gallagher 2021) — even projects

such as Debian for which the Web of Trust is a much valued asset (Wolf and Gallegos 2017). This is because using WKD allows for easier and better key discoverability, while traditional keyserver-based operation is used for building the WoT.

3.2.3 TOFU for OpenPGP

Although OpenPGP is built around the WoT trust distribution model, Walfield and Koch 2016 present an adequation of *GnuPG*, the leading OpenPGP implementation, extending it to support a TOFU-based trust model *in addition* to the WoT. This proposal builds upon a previous proposal for TOFU for browser-based TLS (PKI-CA) connections (Toth and Vlieg 2013) — not only as a technical implementation, but deeply rooted in the study of human-computer interaction and its relation to security decisions.

The authors state that the WoT's utility remains limited, as it is "generally unreasonable to set someone whom you have never met as a trusted introducer". Thus the WoT viewed as a pairwise authentication cannot be meaningfully extended beyond two hops (this is, to authenticate *friends of friends*). "TOFU clearly offers less theoretical protection than either an X.509-style PKI or the WoT, however, it has the huge advantage that it does not rely on third parties and requires very little user support. For many OpenPGP users (...) TOFU would provide *significantly more protection in practice*".

Walfield and Koch's proposal deals, thus, only with the *key discovery* part of the problem this works sets to address; it is motivated by "the working assumption that most connections are secure", and presents an "asymptotic guarantee" of talking with the desired party, since "the longer the user communicates with the same party, the higher the chances the connection is safe". The authors argue that "anecdotal evidence indicates that MitM attacks and forgeries are rare, but communicating with new people is common"; this is, the identity does not have to be trusted at its first use (as nothing backs the fact that it comes from the right party — only that it does not appear *not to come* from them). Trust in a given key belonging to the right user (a *binding*, as presented in this text) is built over time, over reiterated communication. The attack model assumes an active attacker attempting to perform a MitM attack is less likely to guess the exact moment when the communication key is first downloaded.

This article is grounded not only on validating the security of the trust model, but on studying user interaction patterns. As mentioned in Subsection 2.6.3, GnuPG and other programs implementing OpenPGP are seen as a staple of bad user interface design (Whitten and J Doug Tygar 1999; Sheng et al. 2006; Woo 2006; Ruoti et al. 2015). The authors go on detailing one of the weakest points of the keyserver-based model of operation: given that OpenPGP user IDs are "a free form UTF-8 string, which the OpenPGP specifications recommends to be in the form of an RFC2822 mail name-addr" (Callas et al. 2007). They are easily prone to forgeries: an attacker can create identities that look *very similar* to a given user's. This happens because "an attacker could take advantage of the fact that what humans consider to be equivalent is less strict than what a

computer considers to be equivalent. For instance, many people will indicate that *John Doe* is the same identifier as *John C. Doe*". A further refinement on this attack is to use *homographs* — an attacker can register a mail address containing glyphs in alphabets other than Latin that look very much or exactly like a Latin character, fooling a user into using it. The authors delineate different strategies to help detect and warn a user that a key is potentially forged: the authors favor showing the user communication statistics whenever a key is used, including warnings when the key in question is relatively new, or if it has not been used in a very long time.

The focus of this work is on key discovery. It does not imply the WoT trust model should be let go and replaced with TOFU, but it aims at presenting TOFU as simpler and valid for most use cases. The authors explicitly mention that "for those users for whom these defenses are not sufficient, TOFU is probably also not sufficient and they should instead be directly authenticating their communication partners or using the WoT".

Walfield and Koch do present the issue of not only checking whether a signature is valid or invalid, but of assigning a *trust level* to the known keys. OpenPGP keys can be assigned trust levels, and a transitive trust rating can be assigned i.e. following the algebra for assessing trust proposed by Jøsang 1999. Given the use of a TOFU scheme, the authors propose trust policies based on:

1. Whether the binding is already known and has been used before
2. Detecting conflicts with other bindings (the presence of other keys for the same user ID). The authors point out this does not mean an attack has happened: it is customary for users to *rotate* their keys cross-signing them, or to have multiple keys for different uses.

The paper concludes by stating that, while TOFU has a lower theoretical protection than WoT, given the level of user support it requires, ends up offering better *practical* protection, while keeping a *decentralized* operation much more consistent with OpenPGP's historical development than the centralized PKI-CA model.

3.2.4 Autocrypt

The Autocrypt project (Kahn Gillmor 2021) has been born out of the need to bridge the various e-mail encryption projects and usability: given the awful usability record of mail encryption projects (Whitten and J Doug Tygar 1999; Sheng et al. 2006; Woo 2006; Ruoti et al. 2015), the author realizes the fundamental use modes are wrong and should be rethought from the onset. Autocrypt stems from the realization that it is unacceptable that security tools "are very good, but not used because of their usability". The project attempts to build a cryptographic e-mail solution that "works for people who are not interested in becoming cryptography experts".

The act of adopting cryptographic e-mail tools does not concern a single actor; it must be adopted by a whole social circle. Cryptographic learning work-

shops are doomed to fail, the author says, because a person cannot protect their mail's privacy without the rest of their circle doing the same. Autocrypt aims at becoming a standard that multiple Mail User Agents (MUAs, mail clients) can implement, adding extended headers to the mails for encryption to happen automatically when enough information has been exchanged (*opportunistic encryption*), and building upon the principles presented about the TOFU trust model (see Subsection 3.2.3). Thus, Autocrypt addresses several shortcomings leading to attacks on diferent parts of the OpenPGP ecosystem.

Autocrypt solves key discovery by including the sender's public key in-band as a header in all e-mail messages. It does not aim at protecting communications against an active attacker that sends a forged first communication in order for the recipient to fall for a hostile key, but it does protect against passive adversaries (eavesdropping).

E-mail headers are *leaky* in nature: mails encrypted following the OpenPGP standards leak enough metadata for a close user profile to be built. Encrypted messages carry as clear text recipient lists (the *To:*, *Cc:*, *Reply-To:* headers), a short clear-text summary (*Subject:*), even threading indication (*References:*, *In-reply-to:*) and others. Autocrypt moves those headers (except for the unavoidable part, the explicit *To:* header for each given recipient) to the encrypted part of the mail, but specifies that the MUA should present them as if they are sent traditionally.

The author closes by emphasizing the user experience is as important as the cryptography behind it: as it is the case with TOFU for OpenPGP, Autocrypt proponents know it is less secure with the traditional OpenPGP way, but it leads to many more people adopting encrypted mails as a standard way of communicating.

3.2.5 ClaimChain

Autocrypt serves already as the basis for other projects. Addressing some weak aspects of the Autocrypt proposal, Kulynych et al. 2018 aims at assuring the integrity and authenticity of a identity-key binding. It does so by adding information on the keys of *neighbor keys* to build a WoT, supported by an authenticated blockchain-like ledger to ensure, in a decentralized way, that all relevant identities are presented upon key exchange, but in a privacy preserving way: social connections cannot be inferred for each user.

Key certifications in this proposal are termed *claims*, and each user builds a chain-of-blocks structure (based on unique-resolution key-value Merkle tree) to be sent with each message including their claims. The main contribution of ClaimChain is that claims are encrypted and leak no information that could lead to create a graph of social links between users, strengthening privacy in comparison to the keyserver-provided WoT scheme. Claims are linked to *capabilities*, determining which actors (using their private keys) can verify which claims, allowing the controlling user to present the social graph subset needed to validate a fraction of the WoT to specific people.

ClaimChain is designed to perform in-band key distribution, this is, the

public key and its related claims are to be attached (i.e. as headers) to encrypted messages. This naturally increases the size of each exchanged message, but allows for a truly decentralized operation (no servers other than the mail exchanges are required for the scheme to be effective). The authors show that, after a simulated network with 10 000 mails exchanged, the required bandwidth per message is under 30KB, the total *self-storage* (local storage of each user's own ClaimChain blocks) is under 50KB, and the *gossip storage* (space used by information received from other users) is under 2MB.

The authors propose this system to be used with *opportunistic encryption*, showing through simulation its use over the publicly available Enron mail database, in which over 66% of the exchanged mails are encrypted after 10 000 messages.

The article presents a comparison table with other key distribution systems, reproduced here as Table 3.2.

Table 3.2: Comparison of key distribution systems from an end-user perspective. *Social graph visibility*: who learns the user's social graph; *Active attack detection*: whether active attacks by malicious providers, users, and network adversaries can be detected; *Total key availability*: guarantee that recipients' current encryption keys are always available to senders; n: number of users; s: number of sent messages; r: number of received messages; b: maximum total number of contacts of any user after s sent and r received messages (Kulynych et al. 2018).

	In-band PGP	**SKS keyserv.**	**CONIKS**	**Keybase**	**Namecoin**	**ClaimChain**
Social graph visibility	E-mail provider	Public	Provider	Public	Public	Authorized readers
Active attack detection	✗	✗	✓‡	✓‡	✓‡	✓
Total key availability	✗	✓	✓	✓	✓	✗
Sending bandwidth, $O(\cdot)$	$s \cdot b$	$s \cdot b^2$	$s \cdot b \cdot \log(n)^\dagger$	$s \cdot b \cdot (b + \log(n))^\dagger$	$s \cdot b \cdot (b + \log(n))^\dagger$	$s \cdot b^2 \cdot \log(b)$
Receiving bandwidth, $O(\cdot)$	$r \cdot b$	$r \cdot b$	$r \cdot \log(n)^\dagger$	$r \cdot (b + \log(n))^\dagger$	$r \cdot (b + \log(n))^\dagger$	$r \cdot b^2 \cdot \log(b)$
Local storage, $O(\cdot)$	b^2	b^2	$b + \log(n)$	b^2	b^2	$r \cdot b^2 \cdot \log(b)$

† Without costs for auditing the transparency log / verifying blockchain history

‡ Requires global consensus on system's state

Finally, while the argument for ClaimChain is built with OpenPGP as the target implementation, the semantics for the exchanged chains are completely different from what is covered by the OpenPGP standard. So its adoption would require a compatibility layer.

3.3 Analysis on the full keyserver data set

The append-only set of keys served and maintained by the OpenPGP keyservers consists, as of this writing, of more than six million keys, each of them including all of the certificates on them, and totalling over 20 gigabytes of raw data and

representing 30 years of OpenPGP history. This data set has sparked several research papers. This section presents most relevant papers for this work.

3.3.1 Search for key weaknesses

The work by Böck 2015 starts from the fact that "keyservers operate on an add only basis", and take on searching in the full data set for well known cryptographic weaknesses and similar vulnerabilities.

Böck's work is succinct, and presents only two issues searched throughout the keyserver database: a vulnerability in the DSA and ECDSA algorithms upon duplicate k values, where the random factor k, assumed to be globally unique, can be used to aid an attacker into calculating the private key, and an attack on the RSA algorithm where, given distinct keys that share the N modulus, a batch GCD algorithm can be used –again– to attack many keys in parallel.

While attacking cryptographic algorithms is outside the scope of this work, Böck's paper is relevant because some of the problems it reports in the keyserver data. Quoting from the paper, the author describes how key certificates (signatures) can be searched massively for the DSA vulnerability (Böck 2015):

> Therefore searching for duplicate r values in DSA signatures seems like a worthwile idea. The key server data contains different kinds of signatures that can all be tested. (...) Having all the values in a database allows us searching for duplicate values via MySQL:
>
> (...)
>
> This query will give around 350 results. However, when trying to calculate the private keys, it turns out most of these results aren't real signatures. They are merely copies of other signatures with data errors in them. This is generally something that needs to be considered when working with the key server data: the key servers don't check the correctness of the data submitted.

A similar finding is reported when attempting the RSA attack: most of the results the tool reports are faulty keys.

Finding the amount of invalid key data in the keyserver network is the most relevant issue for this work: current keyservers store the arbitrary information they receive, without validating it at all, increasing the likelihood of somebody uploading codified illegal content as described in Subsection 2.6.4.

3.3.2 Threat models to jeopardize the WoT functionality

Barenghi et al. 2015 presents "a practical threat model, aiming at invalidating the public key authentication mechanism provided by OpenPGP, on the basis of a broken keypair either directly or indirectly authenticated by a trustworthy user", as well as how this threat model applies to the real-world data in the keyserver network.

The authors start by presenting structural and statistical features of the WoT and its *strong set* (the "maximally connected subgraph where there is at least one key path between every node pair"). When this text has been written, the total amount of certificates in the keyserver network is been above 3.5 million, but the strong set has been composed of only close to 60000 certificates. The authors detail the strong set as having a maximum graph diameter of 27 *hops*, with 38.7% of all distances smaller or equal to 5. Given a positive integer r and an arbitrary node of the WoT n, they define the *r-certified set of n* as the "set of nodes reachable from n via a valid certifier-certified chain of length shorter or equal to r".

Given the total WoT can be pruned by specifying maximum values for r, they present the following threat model: an attacker E exploits cryptographic weaknesses in a preexisting key T in the strong set, to fool a user A into trusting a fake identity for their desired communications target B by injecting certificates from T.

Two threat models are presented: compromising a fully trusted certificate (this is, when A already trusts T with a `full` trust level) and compromising a key verified (certified) by a fully trusted certificate (A does not yet trust T, but trusts I with a `full` trust level, and I trusts T with this same trust level). In both cases, E is able to inject a certificate from T to their desired fake identity B', impersonating B.

Now, subverting cryptographic weaknesses in preexisting keys is not an easy feat. In the section titled *State of Health of the OpenPGP Global Keyring*, the authors analyzed the data set for known weaknesses:

Outdated RSA key sizes As hardware becomes faster, key sizes that were once secure and recommended for long-term use are now easy to break. OpenPGP supports 512- and 768-bit RSA keys, and the authors point out the practical feasibility of finding the prime factors for 768-bit keys (Kleinjung et al. 2010). We document large projects phasing out 1024-bit keys because of these same concerns (Wolf and Gallegos 2017). There are over 11,000 certificates for keys up to 768 bits in the keyservers, and only 10% of them have been revoked.

Prime RSA modulus If the chosen modulus N of a RSA key is prime, $\phi(N)$ can be trivially computed as $\phi(N) = N - 1$. Only one such key was found in the keyservers, and while it is not part of the strong set, it is signed by one of its members.

Common primes in RSA keys Given two key moduli $n = pq$ and $n' = p'q'$, if one of their factors is the same ($p = p'$), it is relatively efficient to compute both of them. Given the amount of weak keys found by Heninger et al. 2012, the authors expected many keys to be affected by this issue, but only two were found; the authors attribute this to the good PRNG implementations used in the OpenPGP implementations. The scan for this vulnerability did discover 253 keys with nonprime factors, and the affected keys are not valid.

ElGamal with small random integer l The ElGamal cryptosystem has been mostly superseded due to it having similar strength and a significant larger output size than DSA. A issue was found with GPG's implementation of ElGamal (Nguyen 2004): l should be an unpredictable random integer of the same bit-size as p; usual values for 1024-bit keys are $p \geq 2^{1024}$ and $q \geq 2^{160}$. For efficiency reasons, GPG generated DSA keys with l in the bitsize range of q instead, which allows to recover the private key of affected ElGamal keys with minimal effort. Over 1200 unrevoked ElGamal keys are part of the WoT.

MD5-based signatures The MD5 hashing algorithm was long used as a hashing algorithm, but it was proven to be vulnerable to prefix-based collision attacks (Stevens et al. 2009). Despite it being explicitly discouraged in the standard defining OpenPGP (Callas et al. 2007), over 115,000 older certificates can still be compromised (and, thus, used for attacks) using this mechanism, and almost 4% of them are reachable from the strong set.

3.4 Improvements over the HKP keyserver network

The keyserver model, as well as the *Gossip* protocol that enables it to be federated, have been *organically* created and adopted. That is, an implementation is put forward, and as it solves an existing problem *well enough*, it is adopted.

This is not to say they were adopted with no thought or analysis, but possibly new ideas prompt answers better suited to the weaknesses ail the ecosystem. This section explores several such ideas.

3.4.1 BlockPGP: a Blockchain-based Framework for PGP Key Servers

Yakubov et al. 2020 identify the problem of the propagation time a revocation certificate takes to be propagated across the HKP keyserver network as a relevant isssue. Given the likeness of some of the SKS keyserver network's properties to a blockchain, mostly the fact that both are, effectively, a distributed, append-only transaction ledger, adopting a smart-contracts-capable blockchain (particularly, a private implementation of Ethereum) as the backing store for the OpenPGP WoT.

The authors refer to the *log-based PKI* approach presented to address the same problem in the PKI-CA model: the use of highly-available public log servers to monitor and publish certificates, aiding in any CA misbehavior, being the most widely deployed one Google's *Certificate Transparency* (CT) (Laurie, Langley, and Kasper 2013). Although public logs do help counter issues such as those outlined in Subsection 2.6.2, risks such as MitM attacks are still present in log-based PKIs, and their support for revocation and error handling can still be furthered (Matsumoto, Szalachowski, and Perrig 2015).

Yakubov et al. 2020 lists the following as the main issues in order to trust the current PGP key servers infrastructure:

PGP key servers should become more trustworthy The OpenPGP HKP protocol description itself notes that key servers are not to be taken as authoritative and trusted, and all certificates should be checked by interested users. It is trivial –as Subsection 2.6.3 shows– to upload fake key pairs, and it is up to each user to validate whether the trust path to any given key is to be trusted.

Man-in-the-middle risk The authors claim that, given communication between clients and the keyserver network is often done over encrypted (TLS) channels, any breach such as the examples presented in Subsection 2.6.2 would lead OpenPGP data integrity to depend on the PKI-CA model.

Key server synchronization delays A revocation certificate published to a given OpenPGP keyserver can take 15 to 30 hours to reach the full keyserver network, opening an important vulnerability window.

Some key servers are not active Many of the servers in the keyserver network are not kept as up to date as they should, often presenting delays of several days to synchronize with their peers. Said delays increase the risk of downloading compromised or revoked certificates as valid.

The authors point out that, due to the nature of the key synchronization protocol, the *gossip* exchange is based on the full key storage and not on the loading timestamp. So it is impossible to remove certificates, as explained in Subsection 2.6.4. Similar semantics could be reached by using a blockchain-based implementation — focusing particularly on decentralization, persistence and auditability. They present an adequation of the *Hockeypuck* OpenPGP keyserver program using an Ethereum-based blockchain (using an independent blockchain, not linked to the homonym cryptocoin) to synchronize, taking advantage of smart contracts for key validation, and using Proof-of-Authority consensus algorithm. This leads to an implementation much lighter in computational terms than the blockchains used for crypto currencies.

By switching the synchronization protocol to a blockchain implementation, they achieved consensus formation in a matter of two minutes (eight blocks, with block formation time at 15 seconds, as implemented in the Ethereum blockchain) in contrast with up to 30 hours. Inactive key servers can be easily detected and brought up to speed by checking the latest block number they carry.

The authors suggest using a blockchain based on Ethereum, but on a separate instance (not sharing the node set the eponymous cryptocoin uses). The reason for this is threefold: a) using Proof-of-Authority consensus (PoA) consensus, where transactions can be done by any authorized nodes, instead of Ethereum's miner-based Proof-of-Work (PoW) consensus; b) avoiding the need to download Ethereum's public blockchain for a keyserver to start operating, which as of 2020 amounted to 125GB, where the proposed blockchain needs only

to hold keyserver-related data; c) taking part of the public Ethereum blockchain requires paying fees for loading data to the blockchain and pulicly distributing it. OpenPGP operation should continue to be free of such fees, reling only on the infrastructure the operator community is willing to donate.

The Ethereum network requires transactions to be linked to a user account. The authors implement this by using the *Comments:* field in the OpenPGP specification. They present it as a further opportunity to thwart attacks: the keyserver history would no longer be an unauthenticated, inalterable, append-only log, but being user account identification possible, it allows for searching, scrutinizing, and filtering out all actions by a given malicious actor. The authors ponder the existence of an *administrator* account that, while it is antithetical to the decentralized philosophy of the OpenPGP network, can allow for alteration of all nodes. They mention this account is active in the proof-of-concept implementation, which can be disabled in a future production version.

The authors' implementation, based on a modified *Hockeypuck* keyserver, implies additional restrictions in contrast with SKS operation. Most important, an *introducer* that signs a certificate cannot upload it to a key server without the certificate holder's acceptance. This change in behavior by itself limits the ability of an attacker to poison certificates.

3.4.2 The PEAKS keyserver

The thesis by Pini 2018 presents a implementation of a new keyserver, *Peaks*, designed to be interoperable with the SKS keyserver network. Pini's work is tutored by Alessandro Barenghi, and starts from the work described in Subsection 3.3.2.

Pini's proposed keyserver differs from other implementations in that it key material is not stored as a *binary blob*, but is analyzed according to the OpenPGP standard. Its constituent parts are individually stored in a MySQL relational database system, to "handle better the amount of data using a relational database, be better updated and maintained, and verify that certificates have no security issues" (Pini 2018, p. 36).

This work presents not only a full, clearly explained listing of classes and exceptions for PEAKS throughout its Chapter 2, but a thorough recap of the *Gossip* protocol, detailing in Section 1.3 important parts of its implementations mathematically, as pseudocode, and as protocol diagrams.

The full database scheme comprises 11 tables (shown in Figure 3.2), in which the key and certificate structure is "exploded" and allows for an easier analysis, such as the statistical analysis presented in Pini 2018, Chapter 3. It is worth noting that all of the information received by the *Gossip* protocol are stored as raw data in the *certificate* column of the `gpg_keyserver` table, as otherwise, future *Gossip* runs would pull them back again; its `is_unpacked` column denotes whether it was successful unpacking and analyzing the information.

While information received by *Gossip* exchanges must be accepted, PEAKS does perform sanity validations of the key data submitted directly to it, rejecting non-RFC4880-conforming data.

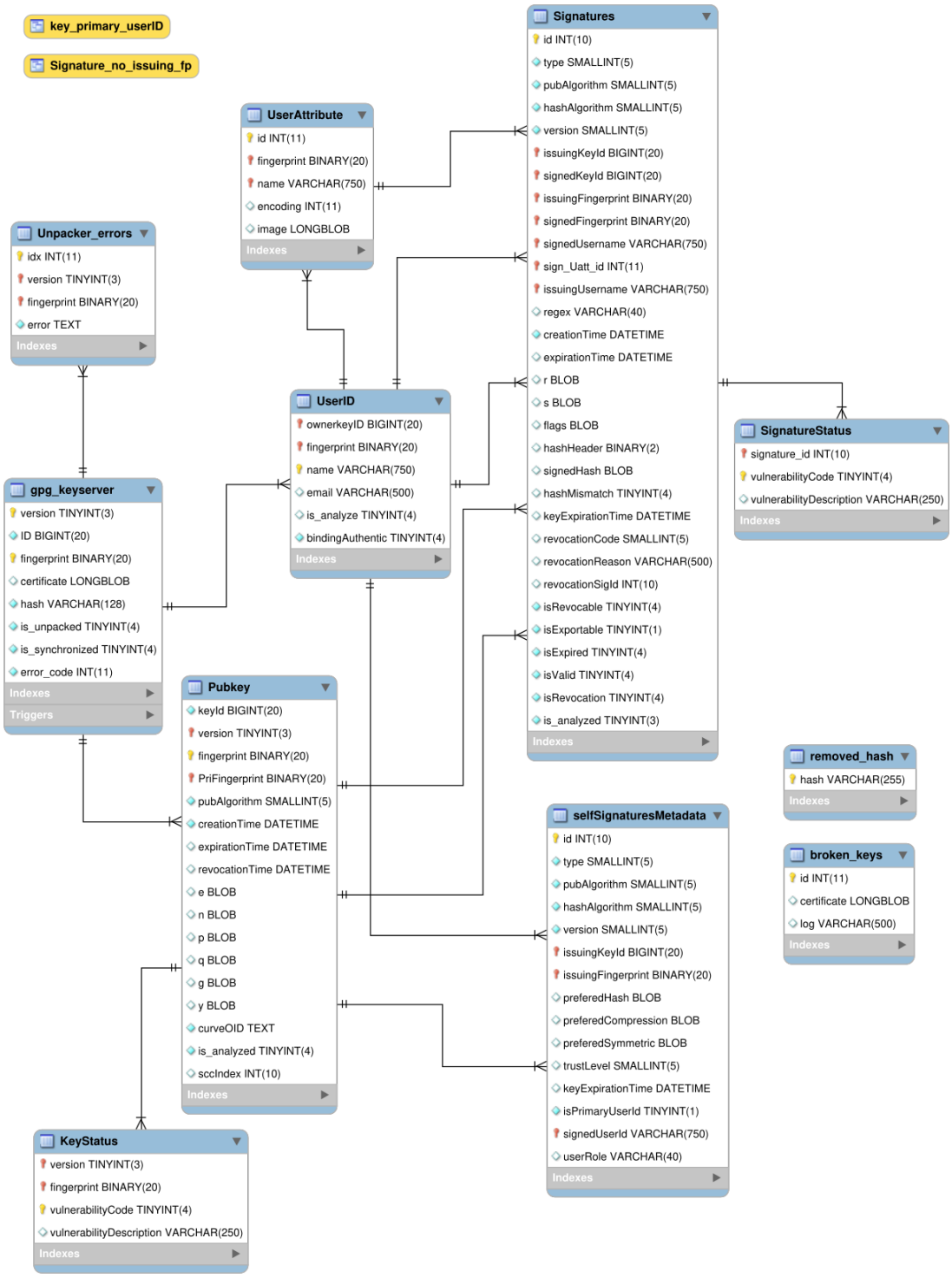


Figure 3.2: Database diagram for the PEAKS keyserver (Pini 2018)

The analyzer component of PEAKS includes a parameterized list of security checks, clearly inspired by Barenghi et al. 2015, including the various items mentioned in Subsection 3.3.2.

While by empirical means it does not seem PEAKS systems are online and participating in the SKS keyserver network, this work carries a strong contribution in it being used for a transversal analysis of the full key set of the keyserver network. Finally, 22 figures and 3 tables are presented, providing an overview of the health of the keys present throughout time (Pini 2018, p. 61–78). Particularly, Figure 3.10 presents the amount of vulnerable RSA keys due to different factors, and as shown in Figure 3.3. The *Evil32* key set in 2014 (discussed in Subsection 2.6.3) has been created optimizing for speed in keypair creation, incurring in *CommonFactor* and *OutdatedKeySize* vulnerabilities.

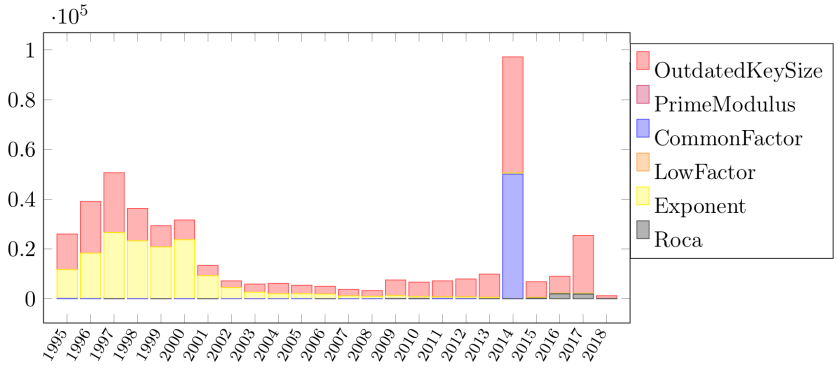


Figure 3.3: Vulnerable RSA keys, noting their vulnerability cause, created over the years (Pini 2018).

3.4.3 Keyserver synchronization without prior context

Rucker 2017 presents an implementation of a synchronizing keyserver using a *gossip* protocol lighter than the protocol used by the interoperating SKS and *Hockeypuck* keyservers (Y. M. Minsky 2002), and using Invertible Bloom Filters (Eppstein et al. 2011). The main contribution for this work lies in that this Invertible Bloom Filter estimates small differences, meaning that servers can more frequently poll each other to check for updates without requesting a large amount of data, and placing an emphasis on minimizing computation overhead.

The proposed keyserver supports an API compatible with HKP, extending it with the `/ibf` and `/strata` points for the Invertible Bloom Filters and the set-difference estimator functionality.

In contrast with the PEAKS keyserver surveyed in 3.4.2, for simplicity sake (and stating that clients must perform their validation anyway), Rucker explicitly does not attempt to perform key data validation other than verifying data consists of valid packets (which limits the abilty to upload random data), and keys are required to be at least PGPv4.

Said claims are backed with benchmarks for the keyserver. Key searches, which take close to a tenth of a second to be answered "have a very poor response time (...) due to an unoptimized index that requires every search to scan the list of all keys". Several strategies for improving this are suggested. Other than that, set synchronizations can be executed in the order of tens of thousands of requests per second.

A synchronization frequency between keyservers of once per minute is suggested based on this study. If the keyservers have no differences, the minimum data exchange is of just 4KB. The synchronization overhead for the bloom filter increases linearly in the number of keys to be synchronized, and the increase on the set difference estimator overhead is roughly logarithmic.

3.4.4 `keys.openpgp.org` and Hagrid, Keeper of Keys

Lee 2019 presents a non-formal paper in which he presents the keyserver situation as beyond a point of no return. He starts the article by stating, "The SKS keyserver network is dying. This has been a long time coming", stating certificate poisoning as the *nail in SKS's coffin*.

Lee is the author of the first widely-known implementation that used OpenPGP signatures to store arbitrary data creating a graffiti of sorts (as detailed in Subsection 2.6.4), and claiming the SKS developers to have neglected patches he offered for critical vulnerabilities for over five years.

The author points out that "three prominent software projects in the OpenPGP ecosystem (Sequoia-PGP, OpenKeychain, and Enigmail) collectively run the Hagrid server `keys.openpgp.org`".

Hagrid is a reimplementation of a keyserver as part of the *Sequoia* OpenPGP project (Breitmoser et al. 2022), with several fundamental differences regarding other keyservers. Quoting from Lee 2019:

> - It's a central service that isn't federated and doesn't sync with other keyservers. They plan to decentralize it at some point, but the federated Hagrid servers will never be badly-run hobby projects like SKS keyservers sometimes are.
> - Anyone can upload public keys, but Hagrid strips all user IDs, signatures, and everything else from them. Only the cryptographic key material, like the numbers required to encrypt a message to the public key, or a revocation certificate, is freely distributed.
> - If a user wants their user ID (their name and email address) to be published, the user needs to prove ownership over the email address using double opt-in validation – Hagrid will send a verification email, and the user must click a link in the email.
> - Because signatures are stripped, Hagrid cannot be used to facilitate the web of trust. If you sign someone's key and want others to know that you signed it, you'll need to find some other way to distribute that signed public key to those people.

The Hagrid keyserver's website explains its main strategies (Sequoia Project 2022). Besides being non-federated and able to act on key material without involving *Gossip* synchronization, key information is *minified* and limited in scope: it does not distribute third party signatures (transitive trust certificates), quoting certificate poisoning ("attaching so many megabytes of bloat to a key that it becomes practically unusable"). The *Sequoia* developers recognize there is value in distributing a WoT. They present ideas such as attaching signatures to the *signer* key instead of doing so to the signed key, but wrap up by stating that "the keys.openpgp.org service is meant for key distribution and discovery, not as a *de facto* certification authority".

This means that, while the Hagrid keyserver *does* counter the certificate poisoning attacks, it is achieved partly by disabling decentralized operation, and partly by disabling transitive trust handling. And although the author mentions that they "plan to decentralize it at some point", the implementation currently available is designed and meant to disable this operation mode deemed fundamental for this work.

Górny 2019 points out that Hagrid-based keyservers also fail to implement UID revocations, detailing some ways where e-mail accounts that get compromised are not able to be removed from the server, presenting a new venue for MitM attacks. Górny also highlights use cases where UIDs that do not have an associated e-mail address, or where a given e-mail address is shared between various users cannot be represented.

It is worth mentioning that, even though this section is based in published communications, as of January 2021 (commit `39c0e12`), Hagrid accepts third party signatures including the management of *key attestations* (the core component of this work, to be described in Section 4.2) as part of the certificates it serves. The authors still maintain that Sequoia is not meant to work as a federated service as a part of the SKS keyserver network, but as a centrally-managed service.

3.4.5 Efficient and private certificate updates

Among the recommended *best practices* for OpenPGP is frequently updating the local keyring, this is, the locally available set of keys with which said user interacts, thus importing certificate updates and revocations. Mueller 2020 addresses this operation, targeting two main issues:

Efficiency Whenever a user requests the update of their local certificates database, they have to download all of the keys the user has registered, as there is no way of knowing whether a key is updated, other than downloading its latest version and fully importing it

Privacy Whenever a key update happens, said user reveals their full contacts network to the keyserver.

The author presents an experimental study of the size and frequency of OpenPGP certificate updates, finding that over a one year period, each cer-

tificate has a probability of 0.062% of having been updated — of course, the distribution of key activity is far from homogeneous, and regular OpenPGP users find their contacts in particular update their keys with a much higher probability.

The article points out that, in order to preserve privacy, the server could send key prefixes only, achieving k-anonymity (this is, allowing the client to fetch certificates in a way it divulges no more than the k neighboring keys), with a recommendation of $k = 5$. This value comes from the theoretical observation that the size of the anonymity set (k) is determined by the total number of certificates (C) and the length of the prefix (in bits, l):

$$k = \frac{|C|}{2^l}$$

k change over time, as the list of keys in the server's set grows. As per the data analyzed in the article (sampled in June 2020), neighboring keys with sizes ≥ 5 could be picked by requesting keys by prefix. 17-bit prefixes form 2^{17} sets of between 13 and 66 keys each (averaging 36), which is the most adequate to guarantee the sought k-anonymity with $k = 5$.

It should be noted that, while this approach addresses anonymity issues, it goes counter to the other stated goal of this thesis: efficiency. The amount of keys a user has to download to perform a local keyring update grows by an order of magnitude.

As for efficiency, the author immediately dismisses the use of HTTP's caching mechanism (via either the *ETags* or the *If-Modified-Since* headers) as they would be susceptible to tracking with great precision.

The author proposes that the server maintains a list of certificates that have changed since the last update, with granularity of days. Combining both approaches, clients can –based on the size of their local certificates database– query the server for updates on given prefixes, giving the server some bits instead of the full key signature as discussed above, resulting in the exchange depicted in Figure 3.4.

This still exposes a trivially enumerable list of certificates that have received updates since a given time period. The author suggests presenting the results of this prefix search as a a Bloom filter, as it obfuscates the elements it contains, and it presents the attractive ability to trivially merge two filters.

3.5 Moving away from OpenPGP

Several proposals start by asserting that, given PGP's age and the fact it has been built to be used in a network so different from current-day Internet, this obsoletes the threat model that OpenPGP sets to solve. Hence, the only way to offer secure person-to-person communication –at least for specific use cases– is to leave OpenPGP behind and switch to a newer encrypted communications system altogether.

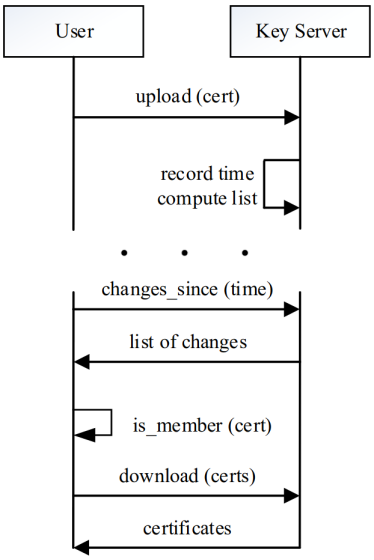


Figure 3.4: List-based certificate retrieval protocol, as proposed by Mueller 2020

3.5.1 Off-the-Record Communication

Already by 2004, Borisov, Goldberg, and Brewer 2004 note that one of the main weaknesses in the OpenPGP model is structural: the privacy expectations for OpenPGP communications rely on either the exchanged messages being kept forever secure, or the private keys for both parties for a given communication to never be "leaked", by carelessness or as a result of an attack. Their work has, from the very onset, a very provocative title (*Off-the-Record Communication, or, Why Not To Use PGP*). So it is naturally expected it would address perceived shortcomings in OpenPGP's threat model.

The authors present a communications protocol presented under the *Off-the-Record* (OTR) name, mainly targeting Instant Messenger (IM) applications. The security expectations set by the authors mimic those found in casual social communications: OTR's communication model follows more that of two parties informally talking in person, rather than *epistolary* communications. OTR does provide assurance to every communications party A that they are communicating with the right, off-band authenticated party B, and encrypts all of the communication between A and B. Even if an eavesdropper E records all of their communication –and even if E gets hold of A or B's master encryption key material– they are not able to retrieve the communications' contents. Of course, neither do A or B — just like in a regular, non-eavesdropped informal talk, words are gone once spoken.

OTR builds on *perfect forward secrecy* by using very short-lived encryption keys, generated after each message exchange (or at most every minute, for very low-powered devices) using the Diffie-Hellman key agreement protocol

on random values. Contrary to most cryptographic uses, OTR does not offer non-repudiation. Much to the contrary, it goes out of the way to *achieve* repudiability: while the Diffie-Hellman exchanges are protected by digital signatures and individual exchanged messages are signed by message authentication codes (MAC), after-the-fact attackers are not able to prove A is the author of m_A even having $\text{MAC}(A, m_A)$. This because, after the ephemeral session keys are rotated, the protocol mandates A and B to reveal their past MAC keys. If an attacker E gets a copy of the session, E does not get the necessary keys to forge the logged messages. In fact voiding any proof of the messages being originated from any given party.

All in all, Borisov, Goldberg, and Brewer 2004 present a very –at the time– novel way of bringing secure encryption to a communication media *very different* from what OpenPGP sets to do. Throughout the text, they reason about an interactive protocol where a steady stream of messages is encrypted using OTR to provide the security guarantees here described. Their Section 6 addresses how to apply OTR's principles to high-latency communications such as e-mail. Given there is no communication initialization phase and assuming both parties of an exchange to be online is often unfeasible, they suggest the use of ring signatures. Although repudiability guarantees would be lower than with the low-latency proposed protocol. The authors finish by acknowledging OTR is not as well suited for e-mail due to its high latency, but reiterates that user' communications *are still more private than if they had used PGP or S/MIME* (Borisov, Goldberg, and Brewer 2004).

3.5.2 Assessing OpenPGP itself: hints of deeper issues

Halpin 2020 presents a series of weaknesses and possible attacks based on weaknesses on OpenPGP stemming, not just from implementation details, but on its design decisions and architecture. It starts by detailing several such weaknesses, following a OpenPGP's history. The weaknesses are presented according to their origin:

1. *Design issues in OpenPGP*: Design choices that have been sensible when PGP is invented and first standardized, between 1991 and 1997, but do not make sense in terms of modern cryptographic protocol design.

    (a) *Message compression*: When OpenPGP has been devised, it is considered good practice to compress a message before encrypting it, and it is so specified in the standard, as scan be seen in Figure 3.5. Years later, it has been shown that compression can lead to attacks on encrypted data as information about plaintext entropy can be gained from the information leaked by the compressed length (Kelsey 2002).

    (b) *Sign-then-encrypt*: OpenPGP specifies a message should be signed first, and the result be encrypted, as Figure 3.5 shows. As a result, there is no knowledge of the intended recipient at signature creation time, and an attacker can take an encrypted message, decrypt it,

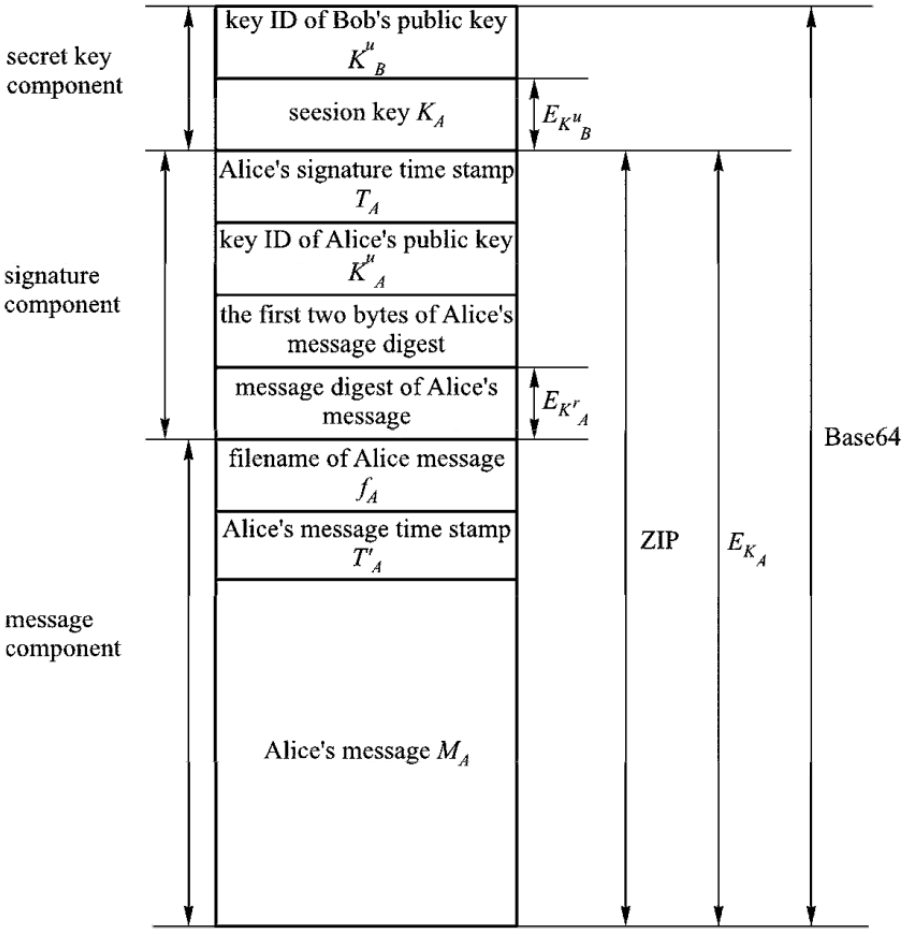


Figure 3.5: General format of a OpenPGP message sent by *Alice* to *Bob* (J. Wang and Kissel 2015, p. 193)

and encrypt to another arbitrary key. The original signature remains valid, allowing for surreptitious forwarding attacks.

(c) *Unauthenticated headers*: OpenPGP is often used for e-mail-based communications. When a message is encrypted and sent, the encryption does not include the mail headers (mainly used for message delivery, but also to convey different metadata on the message). There are several implementations leading to exposing less headers to cleartext exposition or tampering, such as Autocrypt (described in Subsection 3.2.4), but few mail clients have adopted them.

2. *Attacks on PGP*: The following are attacks on specific pieces of OpenPGP infrastructure, not necessarily inherent to the whole PGP ecosystem.

   (a) *Protocol attacks*: Attacks on the cryptographic results derived from the behaviors specified in the OpenPGP standard are well known. Attacks derived from deliberately sending corrupted ciphertext, expecting the user to return quoting the portion they are able to read (thus acting unwittingly as a *decryption oracle*), lead to chosen-ciphertext attacks that have been shown to be able to break the encryption scheme (Jallad, Katz, and Schneier 2002).

   The authors note other protocol attacks, but note they are successfully mitigated in current implementations.

   (b) *Client attacks*: OpenPGP messages are often processed and displayed to the user via e-mail clients. Upon PGP's first release, e-mail consisted exclusively of static text. Over the years, messages are usually be sent as HTML content, which can include styling information via CSS and active content via JavaScript. Given that OpenPGP can be applied to some (and not necessarily all) of the MIME components of a mail, it has been shown that an attacker can capture a valid, encrypted OpenPGP message, attach it to a clear-text HTML part, and once the message has been decrypted and shown to the user, exfiltrate the encrypted component via JavaScript to a network resource controlled by the attacker (Poddebniak et al. 2018).

   The authors mention other examples of attacks on composition of multipart MIME messages where OpenPGP signs some (but not all) of the MIME parts. Many e-mail clients present unclear messages where the user might be confused as to which parts of the content are PGP-signed and which are sent as clear-text.

   (c) *Key management attacks*: Finally, this category presents attacks on the way key material is relayed between valid OpenPGP users. The author identifies three critical flaws in the design of keyservers: a) No requirement for key IDs and user IDs to be unique, which leads to attacks such as *Evil32* (presented in Subsection 2.6.3); b) No authentication used by keyservers, so anybody can update a key claiming to be anyone, effectively opening a window for impersonation, as well

as for certificate poisoning; c) Key verification is in theory done via a WoT, with trust values to be interpreted by each user, effectively forming a "small world" network. Given that user key material in keyservers is public, this social graph of trust reveals the personal data of every user's WoT is public, which goes against any reasonable modern-day privacy expectations.

The author devotes a section to covering Autocrypt (reviewed in Subsection 3.2.4), as it acknowledges most of OpenPGP's issues "are not cryptographic attacks on PGP itself, but primarily due to OpenPGP's interaction with a wider and mostly under-specified infrastructure" (Halpin 2020). This is again done citing the historical context of the attempts to update RFC4880 by the IETF community. While Halpin acknowledges value in the improvements brought forward on OpenPGP brought forward by the Autocrypt project, he points out several underlying weaknesses are still present, particularly lack of authentication and header spoofing. Specific mention is also made of the *TOFU* operation (see Subsections 2.3.3 and 3.2.3) for opportunistic encryption used by Autocrypt: "The danger is that users believe their messages are confidential and even authenticated, when they are indeed not. Although the alternatives like Signal are centralized, they are likely more secure and in line with modern user expectations. Opportunistic encryption does simply not provide the certainty needed by many users about their communication" (Halpin 2020).

The author wraps up by reevaluating whether OpenPGP can be fixed, or if it should be deprecated in favor of newer encryption mechanisms. Most of the problems identified in OpenPGP come from its lack of authenticated encryption. There are IETF working groups working in per-message authenticated encryption with associated data (AEAD).

The author mentions that "decentralized public key infrastructure is an open research problem, both in terms of cryptography and usability. The usability of keys is *the* root usability problem of PGP, and without a breakthrough, new standards will not fix this unless keys are managed on behalf of the user — which goes against the values of decentralization and user control" (Halpin 2020).

3.6 Summary

Throughout this chapter we have presented a review of other works that target different weaknesses and vulnerabilities present in OpenPGP keystores and keyservers.

The IETF draft presented in Section 3.1, presented few weeks before certificate poisoning comes to light, outlines several possible mitigations and redesign ideas that could help implementing abuse-resistant keystores. One of the ideas enunciated by this document is First-party-attested Third-party Certifications, which has been chosen as the core of the present thesis.

Section 3.2 presents five different proposals for key discovery constituting alternatives to public keyservers. The five alternatives discussed through the sec-

tion –DANE, WKD, TOFU for OpenPGP, Autocrypt and ClaimChain– present different ways of relaying public key information to parties interested in initiating communications. They all, however, do so by reducing the role played by a public decentralized WoT. The presented proposals are all viable and valid; particularly the first three are deployed and standardized. However, the present work is carried out aiming at preserve the full decentralization and the value of social cross-certification that constitutes the WoT, as explained in Section 2.3.

Section 3.3 reviews two works that study the full set of keys. Subsection 3.3.1 goes over the full public keyserver set of keys, reporting on vulnerable key algorithms used, weak parameters employed for seeding the keys, finding invalid data stored as part of key material, or similar issues. Subsection 3.3.2 searches for broken key material in the keyserver database in the hopes of finding the probability for either weak keys in the strong set that are easy to compromise based on a list of known weaknesses.

Section 3.4 presents five texts that propose replacing the HKP keyserver network in order to ensure its better operation and not have it fall for the weaknesses of its current model. Subsection 3.4.1 presents a blockchain implementation aimed at preserving all decentralization attributes of HKP keyservers, but ends up introducing an *administrator* account for the blockchain that introduces centralization. Subsection 3.4.2 presents an alternative keyserver, allegedly compatible with the keyserver network, but that validates all of its key material, storing and re-assembling it in a database that allows for better and fuller analysis of several facts. The work presented in Subsection 3.4.3 puts forward an alternative synchronization method, using Invertible Bloom Filters, which carry less overhead and would allow a higher synchronization frequency between servers, reducing the window of opportunity an attacker can have before a fact reaches the whole network (i.e. a certificate revocation) is published by a user. Subsection 3.4.4 presents Sequoia's *Hagrid*, a validating keyserver, while promoting the virtues of a centrally administered service consisting only by verified, valid data. Subsection 3.4.5 presents the case for privacy in certificate updates, by proposing that certificate updates carried out by client software request a *set of key prefixes* instead of individual keys, to ensure personal social data is not yielded to keyserver operators to track.

Finally, Section 3.5 reviews two works that propose completely migrating way from OpenPGP to other private communication protocols better suited for today's social and technical requirements, presenting how the privacy requirements around which OpenPGP was designed do not hold for many ways of communication, and why many of OpenPGP tools' usability shortcomings are its direct consequence.

Chapter 4

A proposal for a certificate-poisoning-resistant protocol

This chapter presents the proposed *First-Party Attested Third-Party Certification* (1PA3PC) protocol to address the issues faced by the non-centralized HKP keyserver network.

Section 4.1 refines the problematic presented in Section 1.2, having reviewed the needed concepts for better understanding it, and presenting results of empirically probing the keyserver network's health over several years, and formalizes in Subsection 4.1.2 why the certificate poisoning attack can take place: the protocol currently in place does not require acknowledgement for appending a signature packet to an existing certificate.

Section 4.2 presents the proposed protocol, giving an interaction overview that shows how a signee requires to publicly acknowledge (this is, to *attest*) any packets appended to their certificate. Section 4.3 details the steps, following an algorithmic notation, to be carried out in order to implement the 1PA3PC protocol.

4.1 Assessment of the magnitude of the problem

As discussed in Sections 2.3 and 2.4 , the use of a large-scale network of transitive trust based on a decentralized WoT scheme requires means for distributing both public keys and peer certification over said keys. In order for the key distribution scheme to be decentralized as well, it is desirable for said scheme to spread over a keyserver network (as opposed to WKD or DANE, presented in Subsections 3.2.1 and 3.2.2).

A working keyserver network has existed for close to 20 years, primarily based on the SKS software. However, both social and technical issues have

led it to become unreliable; as Subsection 2.6.4 explains, many operators have stopped offering keyservers due to the impossibility to remove information once it has been uploaded, particularly as it makes impossible to comply with the European General Data Protection Regulation (GDPR) takedown requests.

Certificate poisoning attacks (see Section 2.6.6) enable any attacker to make any chosen certificate unusable by appending to it too many certifications (by bogus, throwaway identities). Given that, because of the *Gossip* keyserver synchronization protocol, once information regarding any given certificate is uploaded to the keyserver network it is impossible to remove it, said attacks constitute a severe threat.

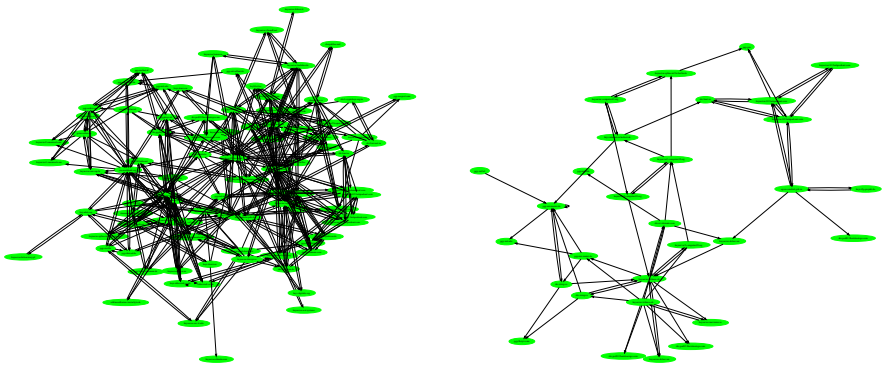
This work holds that there is still value in a fully decentralized Web-of-Trust keyserver network. We proceed to present arguments as and a proposal as how to keep such a service useful and reliable given the described attacks.

4.1.1 A previous study on the SKS keyserver network status

This subsection presents partial results a previous, unpublished study, started in early 2019, aiming to understand the evolution of the network's complexity, that shows how the SKS keyserver network is breaking apart. This clearly illustrated how the current situation endangers the future of the SKS network, a vital component to the decentralized OpenPGP WoT network. The original scope of this study has been to discover and work on the connectivity between keyservers aiming to provide a further analysis it using social network analysis.

While this study's continued data gathering and presenting has been kept as stable as possible over the three years this Subsection explains, there have been important changes in the network topology, in the servers taken as root for discovery, and in the software keyservers use. Some of the gathering and interpreting routines are impacted by said changes; the present data, nevertheless, still supports this narrative.

While the social network analysis do not ultimately lead to meaningful results, the graphs it has produced in early stages make obvious the problem that is now described: if the SKS keyserver network, as an example of a decentralized WoT network, is facing an existential threat. Figure 4.1(a) shows that in early 2019 there has been a lively server community, with strong inner connectivity and a good degree of redundancy. Three years later, figure 4.1(b) presents a striking comparison: the network has severely shrunk, and there is only marginal connectivity between server clusters. The network could very easily partition and stop syncing. Fortunately, due to the nature of the *gossip* protocol, even though edges are directed, information *does* flow bidirectionally (the direction of edges show only which server *declares* to initiate the connections). It should be also mentioned that the two presented figures are *snapshots*: they present the reality found at the moment a particular scan is attempted. However, scans are performed four times a day, and while the topology they describe does vary, the overall traits remain mostly stable when reviewing several snapshots close in time.



(a) Network snapshot as of 2019.01.23 (b) Network snapshot as of 2022.03.02

Figure 4.1: SKS keyserver network connectivity. Nodes represent servers, directed edges represent the declared *Gossip* network; direction represents which server initiates the connection.

The effect of the 2019 attacks in Subsections 2.6.3, 2.6.4 and 2.6.6 leads to a sharp drop in the number of keyservers as part of this network: as Figure 4.2 shows, in early 2019, between 300 and 600 successful connections are often sustained during each run, but between June and August, this number falls down to between 80 and 200, and seemingly reaching an equilibrium during 2020 below 100. Said observations over time can also be appreciated in Figure 4.3: the older measurements can be seen to the right, averaging almost 400 successful connections, with a ± 150 variance. Newer measurements, much more abundant, peak around 90.

Additionaly, the original authors of the SKS program operated for several decades the `sks-keyservers.net` domain, which worked as a set of *keyserver pools*: keyservers that fulfilled given conditions are grouped under DNS aliases (eg. `na.pool.sks-keyservers.net` for servers hosted in North America, `eu.pool.sks-keyservers.net` for Europe; servers operating under the TCP port 80 (HTTP) are grouped under `p80.pool.sks-keyservers.net`, as well as many others). This allows server operators to know their service would be *federated* into a network, and users specifying any of those pools to their OpenPGP implementation would distribute their network load among the whole keyserver network; users are left without a single, reliable address from which to obtain OpenPGP keys and certificates (Tange 2021). But not only that: as Figure 4.2 clearly shows with a second sharp drop, the `sks-keyservers.net` domain name is retired in June 2021; many keyserver operators retire their servers, as the viability of a decentralized network is once more endangered. Figure 4.3 presents a histogram of the successful connections over three years, showing a narrow but clear peak with less than 100 successful connections, and a wider, lower peak with 350-500 connections.

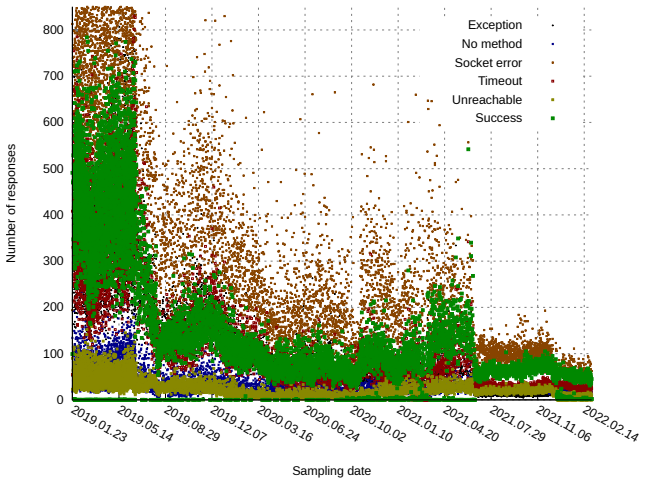


Figure 4.2: Livelihood analysis of the SKS keyserver network, showing the reply status for each of the member servers, measured between January 2019 and March 2022.

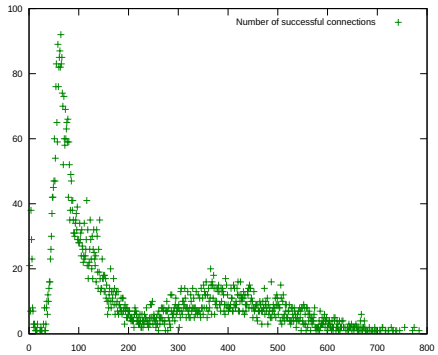


Figure 4.3: Histogram of the number of successful HTTP connections to SKS keyservers from the livelihood analysis of the SKS keyserver network, measured between January 2019 and March 2020.

4.1.2 An interpretation of the HKP key exchange protocol

This work holds that the very lax HKP key exchange protocol can be identified as the culprit for the above situation.

A sample key exchange and validation is presented in Figure 4.4. Do note that the first step (identity validation) is presented as a required step. Key certifications are usually done pairwise (this means, it is usually expected for *Alice* to certify *Bob*'s key and for *Bob* to certify *Alice*'s key), but each individual carries out the validation separately; both parties should first validate their respective identities,[1] and exchange their *key fingerprints*: a single, constant-length *hash* of their public key, as the third line in Figure 2.7 (a string following the form `AB41 C1C6 8AFD 668C A045 EBF8 673A 03E4 C1DB 921F`, this is, a 160-bit hash, presented for readability as ten 16-bit groups, encoded as their hexadecimal values). But this should be stressed: this exchange is done *off-band*, so the keyserver network cannot enforce it to have been carried out.

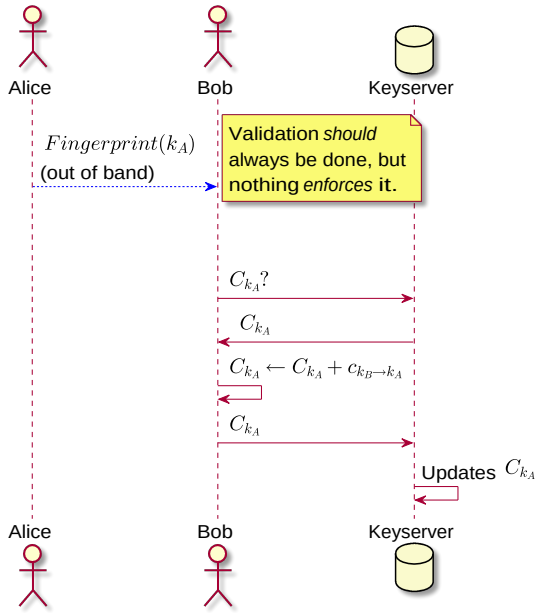


Figure 4.4: Sequence diagram presenting how Bob currently certifies Alice's key.

Bob's following interactions can all be carried out exclusively with the keyservers: he asks the keyserver for the certificate C_{k_A}, matching *Alice's* fingerprint and including her public key, k_A. The server returns this certificate if it is already known (or returns an error otherwise), which *Bob* proceeds to certify[2] ($C(B, k_A)$) and send back to the keyserver. Finally, the keyserver appends

[1] Each person can have different policies and requirements to consider another person's identity as *trusted*.

[2] *Certifying* is often referred to as *signing*, and *certifications* are often called *signatures*;

$C(B, k_A)$ to k_A, following which all requests for k_A includes $C(B, k_A)$. Naturally, due to the *Gossip* protocol, this is propagated throughout the keyserver network.

Any third party *Carol* can then request *Alice*'s certificate, C_{k_A} from the keyserver network, build a trust path to it, and initiate encrypted communications with *Alice*. All this, regardless of whether they have been already in contact with *Alice* or *Bob*, as shown in Figure 4.5.

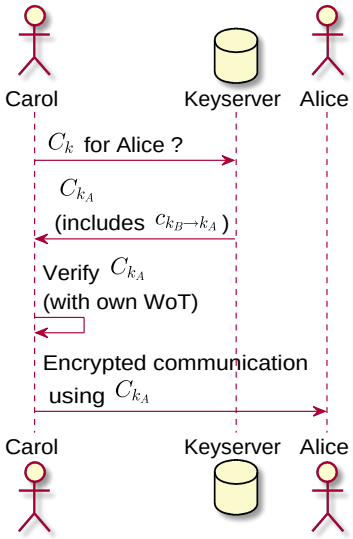


Figure 4.5: Sequence diagram presenting how Carol can find and use Alice's key and validate a WoT trust path for it using a public keyserver, and start a trusted, encrypted communications channel afterwards.

Naturally, the above described communication is vulnerable to certificate poisoning: given that *Alice* has no real control over C_{k_A} (she only controls the corresponding private key), a hostile *Mallory* can easily create n throw-away identities $\{k_{M_1}, k_{M_2}, ...k_{M_n}\}$ and the respective certifications on k_A, this is, $\{C(M_1, A), C(M_2, A), ...C(M_n, A)\}$. With no validation in place, the keyserver network blindly accepts all those certifications over k_A into C_{k_A}.

4.2 First-party attested third party certification protocol

The essence of the 1PA3PC protocol this work presents is to require that any packets added to a certificate belonging to any participant of the keyserver network is publicly accpeted (*attested*) by its owner before being added to the

while said terms clearly avoid the cacophony of so many terms including the *certif–* component, throughout this work we strive for better precision by using the more formal alternatives.

database. By requiring, in contrast with the preexisting protocol described in Subsection 4.1.2, the active participation of the signee for any signature packets to be appended to their certificate, certificate poisoning is effectively erradicated from the threat model.

A side effect of this work is that all *social relation* indications that can be inferred by looking at the WoT connections as a social graph actually represent the real social connections (this is, *Bob* cannot sign a famous user's identity without said user accepting his signature).

In contrast with having all interaction for *Alice's* key certification by *Bob* happening between *Bob* and the *Keyserver*, as presented in Figure 4.4, both actors now interact with each other, and only *Alice* is able to place her certification's modifications to the *keyserver*, as shown in Figure 4.6.

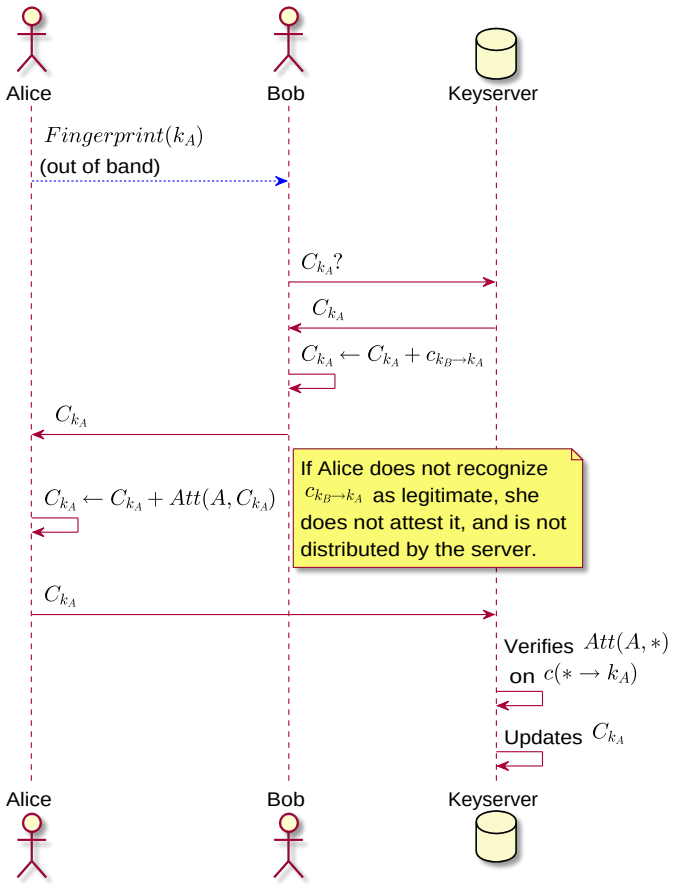


Figure 4.6: Sequence diagram for the proposed first-party-attested key certification

Alice and *Bob* still require to perform off-band identity validation, and the

Algorithm 1: *Bob* intends to sign *Alice*'s certificate

Input: *Bob* has verified Alice as the owner of k_A out-of-band, has k_A's fingerprint f_A.
Output: *Alice*'s certificate C_{k_A} including *Bob*'s signature, $c_{k_B \to k_A}$.
Result: *Alice* receives $c_{k_B \to k_A}$ to act upon.

```
1 begin
2 |  C_{k_A} ← search(keyserver, Alice);
3 |  if found(C_{k_A}) and fingerprint(C_{k_A}) == f_A then
4 |  |  C_{k_A} ← sign(C_{k_A}, k_B);
5 |  |  send(Alice, C_{k_A});
6 |  end
7 end
```

first step of the protocol is –as it is in the current implementation, outlined in Subsection 4.1.2– for *Bob* to retrieve C_{k_A} from the *keyserver*. After locally certifying C_{k_A} to obtain $C(B, k_A)$. However, *Bob* sends the modified certification to *Alice*. *Alice* can then decide whether the certificate should be published or not. If she decides to proceed, creates an attestation $Att(A, C(B, k_A))$.

It is then *Alice* who sends her certified public key to the *keyserver* (and, thus, to the keyserver network). The receiving *keyserver* validates that an incoming packet includes a modification to C_{k_A}, so verifies whether this carries A's attestation and, if so, adds it to the information it knows about C_{k_A}.

4.3 Protocol walk-through

As figure 4.6 shows, our proposed protocol involves the same three actors: two public key owners (*Alice* and *Bob*), who respectively own (control both the public and the private components) C_{k_A} and C_{k_B}, and an automated keyserver.

After meeting in person and exchanging identity information so that *Bob* trusts Alice is the true owner of C_{k_A}, she gives him the key's *fingerprint*. *Bob* wants to publicly certify *Alice*'s certificate, so he follows Algorithm 1:

Bob requests the keyserver for the certificate corresponding to *Alice*'s identity based on its fingerprint and, after verifying the received information is correct and can be asserted to be C_{k_A}, creates a signature packet $sign(C, k_B)$ and appends it to his local copy of C_{k_A}. *Bob* then sends C_{k_A}, with his certification appended, to *Alice*.

Upon receiving C from *Bob*, *Alice* proceeds with Algorithm 2: *Alice* traverses the list of packets in C_{k_A}. When she finds a packet c that contains *Bob*'s certification of her key ($k_B \to k_A$), given she remembers having recently met and exchanged key fingerprints with *Bob*, acknowledges it as a legitimate certification. Of course, if *Alice* does not recognize c to be a legitimate packet, she discards it.

She then appends c to her certificate C_{k_A}, creates an attestation $Att(k_A, C_{k_A})$

Algorithm 2: *Alice*'s actions to acknowledge *Bob*'s signature by attesting it.

Input: Certificate C_{k_A} received from *Bob*.
Output: Updated certificate C_{k_A}.
Result: *Alice* sends the keyserver $c_{k_B \to k_A}$ and the matching attestation $Att(k_A, C_{k_A})$.

```
1  begin
2      foreach packet c in C_{k_A} do
3          if c = k_B → k_A and A.acknowledge(c) then
4              C_{k_A} ← c ;
5              C_{k_A} ← Att(k_A, C_{k_A});
6              send(keyserver, C_{k_A});
7          else
8              discard(c_{k_A});
9          end
10     end
11 end
```

acknowledging the addition of this signature to her certificate, and appends this attestation to her certificate as well. Having done that, *Alice* uploads her updated certificate C_{k_A} to the keyserver.

This triggers the keyserver to start Algorithm 3: the keyserver receives a set of RFC4880-compliant packets, P. P happens to include data augmenting *Alice*'s certificate, C_{k_A}. Do note that the keyserver does not validate who is the originator for: it might have been submitted by an individual user such as *Alice*, or it could have been imported after synchronizing with a different keyserver. Algorithm 3 is executed nonetheless.

For the purposes of this work, we focus on the case where the received data includes packets to be appended to C_{k_A}. The keyserver builds an index of attestations Att_P present in P and starts processing all packets in P. If a given packet c is a certification on k_A, it searches whether Att_P includes a matching attestation $Att(k_A, c)$. If a matching attestation is found, the keyserver adds both c and $Att(k_A, c)$ to its database, otherwise, it discards c.

At this point, the proposed 1PA3PC protocol effectively counters the flaws in the current scheme that make the certificate poisoning possible (refer back to Figure 4.4): by making any packet addition to certificate C_{k_A} depend on having been approved by its owner *Alice* by the means of an attestation $Att(k_A, c)$, a legitimate user B (*Bob*) is able to certify C_{k_A}. However, given each certification needs *Alice* to attest it, a malicious actor *Mallory* is no longer able to attack C_{k_A} with certificate poisoning. At most, she can create an army of fake identities to saturate *Alice*'s mailbox with unwanted attestation requests — but such an attack constitutes only a temporary annoyance (*Alice* is able to purge her mailbox of said spam), and does not carry any longer lasting implications.

Algorithm 3: The keyserver receives *Alice*'s certificate, which might carry new packets in it

Input: Keyserver has received cryptographic material including any packets to be appended to C_{k_A}

Result: Keyserver's database is updated, appending to *Alice*'s certificate C_{k_A} the new signature $c_{k_B \to k_A}$ and its attestation by *Alice*, $Att(k_A, C_{k_A})$.

```
1  begin
2      Att_P ← attestations_index(P)
3      foreach packet c in P do
4          if c(∗ → k_A) then
5              if Att_P includes Att(k_A, c) then
6                  k_A ← c;
7                  k_A ← Att(k_A, c);
8                  add_to_database(k_A);
9              else
10                 discard(c);
11             end
12         end
13     end
14 end
```

4.4 Summary

After having built up the knowledge, both regarding and in the formal challenges being faced by the decentralized keyserver network, presented in the earlier chapters, this chapter starts by giving a more focused view on the problem stated in Section 1.2.

Subsection 4.1 presents a study done throughout three years, which documents how the size of the keyserver network is effectively shrinking.

Subsection 4.1.2 presents the currently implemented protocol for performing key certifications: the reason why certificate flooding is possible is that no action needs to be taken by a given person to acknowledge a certification is legitimate.

Section 4.2 presents the core of this work: the protocol with which a keyserver ensures a key certification for a given user is accepted by the receiving user.

Section 4.3 details how the protocol is to be implemented for each of the involved actors, walking through the specific algorithms for each of them.

Chapter 5

Implementation, validation and results

This chapter starts by outlining the integration of this proposed work into the existing OpenPGP services and tools ecosystem, while following the IETF standard defining OpenPGP, RFC4880,[1] and discusses the tests carried out that prove the proposed protocol does serve as a counter measure against the certificate poisoning attack.

5.1 Implementing the protocol

This research proposes a protocol to counter the ill effects of certificate poisoning. In this section, we present an implementation carried out with modifications as small as possible to currently deployed software, as a means for validating our protocol's effectivity. While *GnuPG*, the *de-facto* reference OpenPGP implementation, does not yet implement attestation (as it is shown in Subsection 5.1.2), it is supported by *Sequoia,*[2] a modern, Rust-based reimplementation that is gaining traction in the OpenPGP user space.

The version of *Sequoia* used for the code snippets shown in this chapter is quite different from usual OpenPGP tools, as it operates in a *stateless* way: it does not have a user directory (such as *GnuPG*'s `.gnupg`) in which all of the user's cryptographic material is stored. So the user identity to be used for each operation always has to be specified as an explicit command-line parameter. The current Sequoia versions do offer both a stateful implementation, `sq`, and a stateless implementation based on the Stateless OpenPGP Command Line

[1] This work is developed based on the published RFC 4880 standard (Callas et al. 2007), and following the standardization process for its update. In August 2024, RFC 9580 is released (Wouters et al. 2024). We decided that this work should continue to refer to RFC 4880 as it is still the document that all of the implementations studied and used already adhere to. We do not believe any changes in the new standard to counter what this work discusses or proposes.

[2] https://sequoia-pgp.org/

Interface (Gillmor 2024), `sqop`.

5.1.1 Certificate attestation under *Sequoia*

Before detailing the implementation done as a proof for our protocol, we consider relevant to detail how the needed functionality is achieved. In order to illustrate how certificate attestation can be carried out with *Sequoia*'s command-line tool, `sq`, this subsection follows up the explanation with users *Alice* and *Bob*, both creating their keys. *Bob* signs *Alice*'s certificate, and *Alice* attests that signature as valid. Notice that here an `sq` tool is used, available to the general public as of April 2023. The aim is just to illustrate the general workflow *using currently available tooling.*

Key creation (both parties) *Alice* and *Bob* create their cryptographic key material. Each of them would do this separately, on their own computer. In both cases, the full fingerprint is shown, as it has to be communicated to the other interested party (for the examples shown, this output is filtered to show only the relevant line of output). *Alice* operates from the host called `wonderland`, and *Bob* from `sponge`:

```
wonderland$ sq key generate -u 'Alice <alice@example.org>' --export
     alice.pgp
wonderland$ sq inspect alice.pgp | grep Fingerprint
    Fingerprint: 5D0C4ABC3AA08597421E65BEA348D974414C729A
```

The first generated file, `alice.pgp`, contains the full cryptographic material (both the private and the public key). `alice.pgp.rev` is the revocation certificate (which is not further used in this case). *Bob* does the equivalent actions on his system:

```
sponge$ sq key generate -u 'Bob <bob@example.net>' --export bob.pgp
sponge$ sq inspect bob.pgp|grep Fingerprint
    Fingerprint: 43E737F786979A81CC41A6A75742563F7281572B
```

The files generated by both parties are not yet good for uploading to a keyserver: as it is shown later in Subsection 5.1.2, they include both the private and the public components.

Extracting and uploading the certificate The generated key should be always carefully guarded, as it contains the private key, which is secret material. In order to upload the certificates to a keyserver, *Sequoia* requires the user to extract it from the full key. After doing this, both users send their certificates to the keyserver:

```
wonderland$ sq key extract-cert --output alice_cert.pgp alice.pgp
wonderland$ sq keyserver send alice_cert.pgp
```

```
sponge$ sq key extract-cert --output bob_cert.pgp bob.pgp
sponge$ sq keyserver send bob_cert.pgp
```

Obtaining, signing and sending *Alice*'s certificate After having met in person to exchange their fingerprints, *Bob* fetches *Alice*'s certificate from the keyserver and signs it. Notice that *Bob* needs to specify which UID or UIDs from *Alice*'s certificate to sign:

```
sponge$ sq keyserver get alice 5D0C4ABC3AA08597421E65BEA348D974414C729A
     --output alice_cert.pgp
sponge$ sq certify bob.pgp alice_cert.pgp 'Alice <alice@example.org>'
     --output alice_cert_signed.pgp
```

The next step is to communicate *Alice*'s signed certificate back to her; *Bob* uses the Unix `mutt` mail management program to do so, due to the ease to be driven from the command line.

```
sponge$ echo 'Signed certificate is attached to this mail.' | \
    sq sign --cleartext-signature --signer-key bob.pgp | \
    mutt 'Alice <alice@example.org>' -s 'Your OpenPGP signed certificate'
         -a alice_cert_signed.pgp
```

The above pipeline builds a mail for which the subject is the string `Your OpenPGP signed certificate`, which includes the attached file `alice_cert_signed.pgp`, and with the text specified in the first line cleartext-signed as the mail body.

***Alice* verifies, attests, and uploads the signed certifcate** Once *Bob* has sent the mail to *Alice*, given she already knows *Bob*'s identity, she can verify the mail actually comes from the expected party. The following exchange is somewhat simplified, as the workflow for mail handling can be quite cumbersome. From her mail client, she feeds the received message body to `sq verify` and, if the verification satisfies her, saves the attachment as `my_signed_cert.pgp`:

```
[wonderland mail client] | sq verify --signer_cert bob_cert.pgp
Good signature from 5742563F7281572B
Hi! Please find your signed certificate attached to this mail.
1 good signature
[wonderland mail client] attachment_save_as my_signed_cert.pgp
```

Having verified the mail really comes from *Bob* (by checking the fingerprint, `5742563F7281572B`; this is usually automatically done by the mail client), she merges the signature into her main certificate, attesting it:

```
wonderland$ sq keyring merge alice.pgp alice_cert_signed.pgp --output
     alice_key_with_cert.pgp
wonderland$ sq key attest-certifications alice_key_with_cert.pgp -o
     alice_attested.pgp
wonderland$ sq key extract-cert --output alice_attested_cert.pgp
     alice_attested.pgp
```

Finally, *Alice* can now send her attested certificate, including *Bob*'s certification, to the keyservers:

```
wonderland$ sq keyserver send alice_attested.pgp
```

At that point, a keyserver operating under the proposed protocol should accept *Alice*'s attested certificate, including *Bob*'s certification.

5.1.2 The attestation as OpenPGP packet-level data

OpenPGP keys stored as files are opaque and not meant for end-users to inspect. `sq` provides two commands, `inspect` and `packet`, to provide users respectively with simple and detailed information about the information. This subsection presents the corresponding information for the salient points presented in Subsection 5.1.1. Given the amount of output yielded both by the `inspect` and `packet` subcommands is too large, comparisons between different runs is made using the Unix `diff` tool, capturing the output of two program invocations. This is included to clarify the effect of the specific actions analyzed.

The first and second steps mentioned previously are key creation and, from there, public key extraction. The differences between the full key and the uploadable certificate are, as expected, only the removal of the `Secret key` packets:

```
$ diff -u <(sq inspect alice.pgp) <(sq inspect alice_cert.pgp)
--- /dev/fd/63  2023-04-26 12:37:17.018675195 -0600
+++ /dev/fd/62  2023-04-26 12:37:17.018675195 -0600
@@ -1,9 +1,8 @@
-alice.pgp: Transferable Secret Key.
+alice_cert.pgp: OpenPGP Certificate.

     Fingerprint: 5D0C4ABC3AA08597421E65BEA348D974414C729A
 Public-key algo: EdDSA
 Public-key size: 256 bits
-     Secret key: Unencrypted
   Creation time: 2023-04-26 18:25:26 UTC
 Expiration time: 2026-04-26 11:51:47 UTC (creation time + P1095DT62781S)
       Key flags: certification
@@ -11,7 +10,6 @@
          Subkey: DE923B01881B31292AFCA6E08517F86ECBD84BCB
 Public-key algo: EdDSA
 Public-key size: 256 bits
-     Secret key: Unencrypted
   Creation time: 2023-04-26 18:25:26 UTC
 Expiration time: 2026-04-26 11:51:47 UTC (creation time + P1095DT62781S)
       Key flags: signing
@@ -19,7 +17,6 @@
          Subkey: C95BA9C4669D295FF5BC634763E668F03D50B993
 Public-key algo: EdDSA
 Public-key size: 256 bits
-     Secret key: Unencrypted
   Creation time: 2023-04-26 18:25:26 UTC
 Expiration time: 2026-04-26 11:51:47 UTC (creation time + P1095DT62781S)
       Key flags: authentication
@@ -27,7 +24,6 @@
          Subkey: E17B9CF1F4404363BE6D14ACB42B4346746850B4
 Public-key algo: ECDH
 Public-key size: 256 bits
-     Secret key: Unencrypted
   Creation time: 2023-04-26 18:25:26 UTC
 Expiration time: 2026-04-26 11:51:47 UTC (creation time + P1095DT62781S)
       Key flags: transport encryption, data-at-rest encryption
```

Bob's certification on *Alice*'s key consists in a signature packet appended to *Alice*'s generated certificate:

```
$ diff -u <(sq packet dump alice_cert.pgp) <(sq packet dump alice_cert_signed.pgp)
--- /dev/fd/63  2023-04-26 12:38:04.074703065 -0600
+++ /dev/fd/62  2023-04-26 12:38:04.074703065 -0600
@@ -50,6 +50,22 @@
     Digest prefix: 4BC9
     Level: 0 (signature over data)

+Signature Packet, new CTB, 195 bytes
+    Version: 4
+    Type: GenericCertification
+    Pk algo: EdDSA
+    Hash algo: SHA512
+    Hashed area:
+      Signature creation time: 2023-04-26 18:26:53 UTC (critical)
+      Signature expiration time: P1826DT18235S (2028-04-25 23:30:48 UTC)
    (critical)
+      Issuer: 5742563F7281572B
+      Notation: salt@notations.sequoia-pgp.org
+        00000000  76 ba 5b 1b 44 d7 e1 6c  dc fd 6f a2 e7 10 fe d2
+        00000010  f6 b3 9a 5c e8 44 68 ae  a3 38 c4 dd 5d 88 19 4a
+      Issuer Fingerprint: 43E737F786979A81CC41A6A75742563F7281572B
+    Digest prefix: EB9E
+    Level: 0 (signature over data)
+
 Public-Subkey Packet, new CTB, 51 bytes
     Version: 4
     Creation time: 2023-04-26 18:25:26 UTC
```

And the difference between the certificate signed by *Bob* and it being attested by *Alice* also consists only of a signature packet of type `AttestationKey`:

```
$  diff -u <(sq packet dump alice_cert_signed.pgp) <(sq packet dump
    alice_attested_cert.pgp)
--- /dev/fd/63  2023-04-26 12:38:33.466720474 -0600
+++ /dev/fd/62  2023-04-26 12:38:33.466720474 -0600
@@ -50,3 +50,17 @@
     Digest prefix: 4BC9
     Level: 0 (signature over data)

+Signature Packet, new CTB, 255 bytes
+    Version: 4
+    Type: AttestationKey
+    Pk algo: EdDSA
+    Hash algo: SHA512
+    Hashed area:
+      Signature creation time: 2023-04-26 18:26:21 UTC (critical)
+      Issuer: A348D974414C729A
+      Notation: salt@notations.sequoia-pgp.org
+        00000000  92 84 f8 12 b4 4a f0 f0  70 5b fe 55 78 13 e1 98
+        00000010  a2 b8 dc ac 22 71 a9 6c  01 fe 00 94 41 a6 f7 28
+      Issuer Fingerprint: 5D0C4ABC3AA08597421E65BEA348D974414C729A
+      Attested Certifications:
+        1DBDA99BBD686E0C87FD6350B04559124C3F99A3BF13C9D6EC9031AF72 \
+        75F48AA226B6F8A871C0763A9A88F22E7EDA4D08287EF043BCBE59CAA3 \
+        1037640282F5
+ (critical)
+    Digest prefix: 3413
+    Level: 0 (signature over data)
+
 Signature Packet, new CTB, 195 bytes
     Version: 4
     Type: GenericCertification
```

Notice that, while the attestation is also a signature packet, `sq` does recognize it carries *Attested Certifications*. This is an extension to RFC 4880; given

the *GnuPG* key does not yet support this extension, if the same two certificates are compared by its equivalent to `sq packet dump` (`gpg --list-packets`), the report shows basically a self-signature[3] with extra data following a notation defined at `salt@notations.sequoia-pgp.org` in the *hashed subpacket 20*:

```
$ diff -u <(gpg --list-packets alice_cert_signed.pgp)  <(gpg --list-packets
    alice_attested_cert.pgp)
(...)
-# off=507 ctb=c2 tag=2 hlen=3 plen=195 new-ctb
+# off=507 ctb=c2 tag=2 hlen=3 plen=255 new-ctb
+:signature packet: algo 22, keyid A348D974414C729A
+       version 4, created 1682533581, md5len 0, sigclass 0x16
+       digest algo 10, begin of digest 34 13
+       critical hashed subpkt 2 len 4 (sig created 2023-04-26)
+       hashed subpkt 16 len 8 (issuer key ID A348D974414C729A)
+       hashed subpkt 20 len 70 (notation: salt@notations.sequoia-pgp.org=[not
    human readable])
+       hashed subpkt 33 len 21 (issuer fpr v4
    5D0C4ABC3AA08597421E65BEA348D974414C729A)
+       critical hashed subpkt 37 len 64 (?)
+       data: [256 bits]
+       data: [254 bits]
```

In order to validate an attestation for the keyserver to accept or reject it upon submission or reconciliation, we can focus on the relevant packet (which can be isolated as *Sequoia* identifies its type as `AttestationKey`) in the `alice_attested_cert.pgp` file obtained above:

```
Signature Packet, new CTB, 3 header bytes + 255 bytes
    Version: 4
    Type: AttestationKey
    Pk algo: EdDSA
    Hash algo: SHA512
    Hashed area:
      Signature creation time: 2023-04-26 18:26:21 UTC (critical)
      Issuer: A348D974414C729A
      Notation: salt@notations.sequoia-pgp.org
        00000000  92 84 f8 12 b4 4a f0 f0  70 5b fe 55 78 13 e1 98
        00000010  a2 b8 dc ac 22 71 a9 6c  01 fe 00 94 41 a6 f7 28
      Issuer Fingerprint: 5D0C4ABC3AA08597421E65BEA348D974414C729A
      Attested Certifications:
        1DBDA99BBD686E0C87FD6350B04559124C3F99A3BF13C9D6EC9031AF72 \
        75F48AA226B6F8A871C0763A9A88F22E7EDA4D08287EF043BCBE59CAA3 \
        1037640282F5
 (critical)
    Digest prefix: 3413
    Level: 0 (signature over data)
```

This packet can be easily identified as created by *Alice*, as its `Issuer` matches the lower 64 bits of the `Fingerprint`, which is the same as her public key value.

During the development of this work, a yet missing piece as of the version of *Sequoia* when this work is carried out (at vesion 0.27.0) is a way to ask *Sequoia* for which signatures on a given certificate have been attested by its owner.

[3]The OpenPGP standard recommends certificates to always be self-signed and be distributed including the self-signature so that it is a complete entity even when the secret key is removed from the transferable secret key (Callas et al. 2007, p. 11.2)

5.1.3 Code modifications for implementing the 1PA3PC protocol

The previous section presents how attestations are represented inside each certificate. Some minor code modifications are necessary in order to implement the 1PA3PC protocol here. Notice that the full implementation is shown at Appendix A.

As explained in Section 1.4, a guiding criteria is to keep the code modification for implementing this protocol at a minimum. This section describes the changes made to the *Sequoia* command line tool `sq` (see Appendix Section A.1) and to the *Hockeypuck* keyserver software (refer to Appendix Section A.2).

Due to some versioning conflicts between the *Hockeypuck* and *Sequoia* versions used, it is deemed convenient to run them in separate virtual servers, so *glue code* in the form of two scripts (see Appendix Section A.3) is written.

Finally, the installation, configuration, and measurement of the test system is driven by a custom-developed script, presented in Appendix Section A.4.

5.2 Validation

In order to verify the proposed protocol is effective against the certificate poisoning attack in a hostile environment, we set up the following experiment:

With a network of five keyservers based on the *Hockeypuck* software (Marshall 2015a). 500 OpenPGP keys are generated using the *Sequoia* client (Azul et al. 2021), and a WoT is laid out on it, with a signature between each of two keys at random being made with $p = 0.01$. We do not believe specific WoT properties are relevant for the experiment, so it was not attempted to replicate the internal structure of the real OpenPGP WoT. The reason for creating the signatures is to generate the expected variance in the number of packets that constitute each of the keys, thus driving their size to realistic values. The 500 keys are uploaded randomly to each of the five keyservers. Figure 5.1 shows the upload and synchronization progress to the five keyservers until they converge on the same keyset. As Figures 5.1, 5.2 and 5.3 show, key generation and upload happens since the beginning of the experiment and approximately until the end of its first minute ($t = 60$).

The keyservers connect with each other for synchronization several times per minute, and this can be seen with the spread of keys through them, starting approximately at $t = 15$ and until the number of keys per server stabilizes at 500, in Figure 5.1, at $t = 215$. Signature packets continue to be appended to the keys, but their number remains stable with 500 keys from this point on: all of the additional information received consists on signature packets added to existing keys.

After the "valid" signatures are generated, and as the "legitimate" WoT gets populated and the signatures are still spreading through the keyserver network, an attack is simulated: starting around $t = 200$, 1 000 further keys are created (but not uploaded to the keyserver network). Five victim keys are selected and

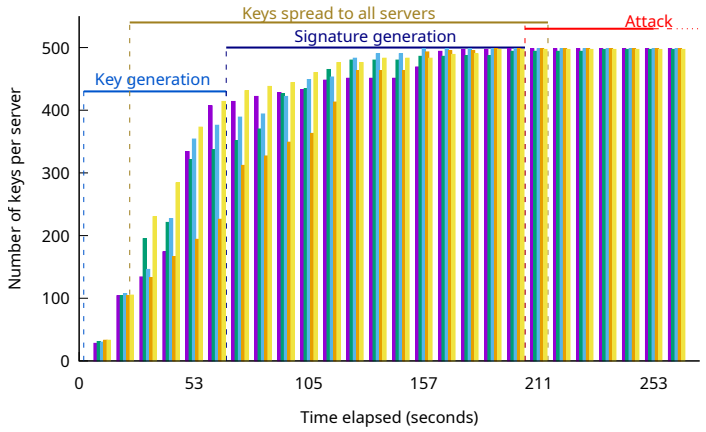


Figure 5.1: Number of keys present in the keyserver network over the lifetime of the attack simulation.

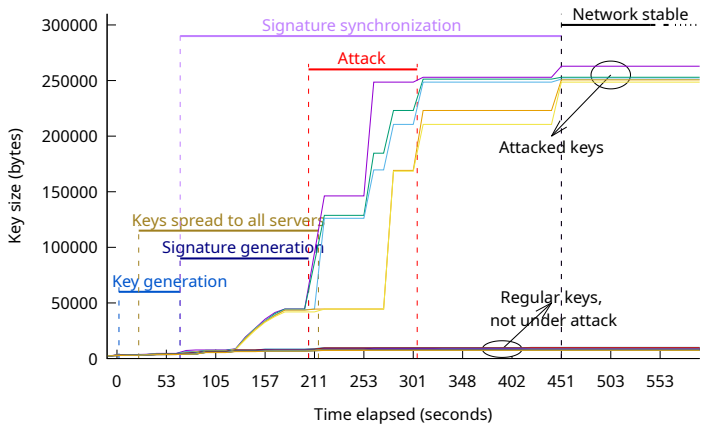


Figure 5.2: Size of the 20 largest certificates, as seen by one of the keyservers in the network, during the lifetime of the attack simulation, using the traditional protocol.

signed with the 1 000 attacker keys, with the signatures uploaded randomly to each of the keyservers. The attack is quite efficient time-wise: it takes only close to 100 seconds to perform. As a result, as evidenced in Figure 5.2, by $t = 300$, the five attacked keys are *bloated* to almost 300KB, while the rest of the keys remain under to 5KB in size. The keyserver synchronization protocol is left running so the information spreads to the whole network, and by $t = 450$, information flow over the keyserver network stabilizes.

In contrast, using our proposed 1PA3PC protocol, the same experiment is carried out. It becomes clear that the attack does not succeed, as Figure 5.3 shows: the 20 largest keys in the keyring remain all within the same range and, while they continue to successfully receive certifications, an ill-intentioned *Mallory* is no longer enable to disrupt *Bob* or threaten his certificate's connectivity in the WoT.

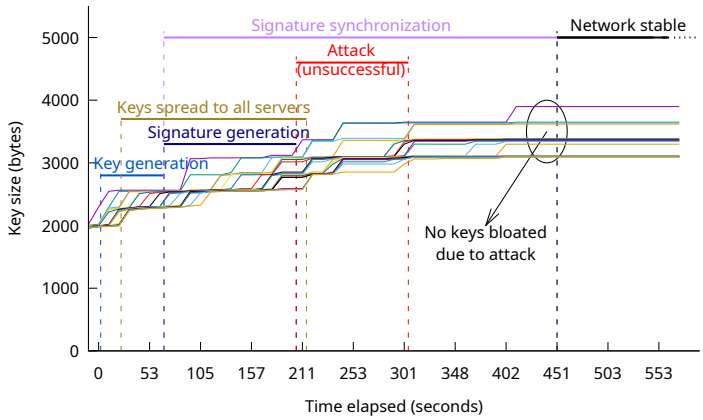


Figure 5.3: Size of the 20 largest certificates, as seen by one of the keyservers in the network, during the lifetime of the attack simulation, using our 1PA3PC-enforcing protocol.

Summing up the reported times, the experiment consists of the following approximate intervals shown in Table 5.1.

Table 5.1: Stages of the network during the experiment

Time (seconds)	**Stage**
0 – 60	Key generation
15 – 215	Keys spread to all servers
60 – 200	Signature generation
200 – 300	Attack
60 – 450	Signature synchronization
450 –	Network is stable

As for the results, while the attack led to five keys being disproportionately

larger than the rest of the keys present in the keyserver network in Figure 5.2, showing evidence of a successful attack, Figure 5.3 shows all keys within the same range, as expected from the random signature distribution explained earlier in this section. It is worth keeping in mind that the attack simulated in this experiment consisted of 1 000 throwaway, hostile keys only. Observed certificate poisoning attacks have been close to a hundred times bigger.

5.3 Summary

After delineating our proposal in Chapter 4, this chapter presents the implementation and experimental validation carried out supporting this thesis.

Section 5.1 presents, step by step, how users can use the *Sequoia* OpenPGP client to carry out each of the steps needed by our proposed protocol: create key pairs, extract the certificate, interact with keyservers, sign third party certificates, verify them and attest their validity. Subsection 5.1.2 presents the packet-level differences between the relevant stages of the interaction. Subsection 5.1.3 point to each of the relevant sections of Appendix A, where the details of the server code modification and deployment are fully detailed.

Section 5.2 begins by presenting the details of the experiment carried out to prove the efficacy of the 1PA3PC protocol against the certificate poisoning attack and shows the observed behavior, clearly showing the attack is successful before our protocol is introduced, but is completely avoided once the proposed modifications are active.

Chapter 6

Conclusions

This final chapter looks back on the research work in Section 6.1, validating in Section 6.2 the goals set by the hypothesis are met, identifying in Section 6.3 the contributions brought forward. Section 6.4 aims at a general evaluation of the contribution it does within the field by comparing with the related works presented throughout Chapter 3. Finally, Section 6.5 explores future directions where this work can be continued.

6.1 Summary of the research work

This research thesis introduces a protocol for countering the OpenPGP certificate poisoning attack in the setting of a decentralized keyserver network based on a *Gossip* synchronization protocol. The central points of this work are also published in the *Journal of Internet Services and Applications* (Wolf and Ortega-Arjona 2024).

The characteristics of OpenPGP, the reasoning behind a decentralized model, and the details of the *Gossip* centralization protocol are introduced in Section 1.1, and detailed through Chapter 2. The certificate poisoning attack is first presented in Section 1.2. The reasoning behind its emergence is presented throughout the several smaller attacks and weaknesses presented throughout Section 2.6, and particularly in Subsection 2.6.6.

Chapter 3 presents different mechanisms that, directly or indirectly, help reduce the impact of this attack. Section 3.1 presents a document laying out several different possible paths for tackling abuse in OpenPGP keystores. Section 3.2 presents several works that identify the problem as stemming from the decentralized, append-only nature of the keyserver network, and propose to replace it with other key discovery and distribution techniques. Section 3.3 presents several transversal studies aimed at finding the prevailing weaknesses in the main keyserver database, comprising over 30 years of public cryptographic material. Section 3.4 details several proposals attempting to improve the OpenPGP ecosystem's security without abandoning the keyserver model, and keeping the

centrality of the Web of Trust as a distributed transitive trust model. Section 3.5 presents works by different authors holding that OpenPGP's shortcomings are too many, and proposing other communication alternatives not relying in its technology.

Chapter 4 presents the "First-party-attested Third-party certifications" protocol (1PA3PC) that is the core of this work. In order to do this, Section 4.1 presents evidence on how the keyserver network is in the brink of a breakdown since the certificate poisoning attacks are public, and presents the formalization on how this is partly due to the absolute lack of verification in the key material submitted to keyservers. Section 4.2 presents the proposed protocol from a high level overview, and Section 4.3 presents a detailed walk-through, presenting the algorithms each of the communication parties follows to exchange certifications following 1PA3PC.

Having the protocol enunciated, it is necessary to prove its effectiveness for countering the certificate poisoning attack. Chapter 5 details the experiment performed to validate 1PA3PC works as expected. Section 5.1 explains the necessary first steps in creating, validating and certifying identities using the *Sequoia* client software for a minimal sample scenario between communication parties *Alice* (A) and *Bob* (B). Subsection 5.1.3 outlines the code modifications performed in order to implement our protocol in the *Hockeypuck* keyserver software. Having the modifications ready, Section 5.2 details the experiment with which this work is backed, and presents the observed results in a keyserver network.

6.2 Hypotesis restatement

As stated in Section 1.3, the hypothesis for this work is:

> A protocol is proposed that allows for the synchronization of OpenPGP certificates between keyservers, while preventing the *certificate poisoning* attack. It preserves the main characteristics of the Web of Trust transitive trust distribution schema, including that of being fully decentralized, by performing relatively small modifications to the HKP keyserver interaction model, and allowing for interoperability with the currently deployed client software.

As presented in Chapter 5, the proposed protocol is proven to counter the ill effects of *certificate poisoning*. It does not change semantics nor most operations related to the WoT transitive trust distribution schema, and ensures decentralized operation can still function. As mentioned in Subsection 5.1.3, the required code modifications are really minimal, totaling less than 500 lines of code, and retaining client compatibility with the existing keyserver network.

It is worth pointing out that the hypothesis mentions *interoperability with the currently deployed client software*. The relevance for insisting in maintaining compatibility is to avoid the need for a large-scale software migration. Projects such as *GnuPG*, *Thunderbird*, *Mailvelope* or *ProtonMail* boast about millions

of installations. Proposing a change requiring an incompatible update for all users is impossible.

In contrast, there are currently close to a hundred keyservers (Gallagher 2024b). It is not beyond reason to coordinate a protocol update with the current keyserver operators.

As for the validation of the proposed 1PA3PC protocol, the experimental validation presented in Section 5.2 shows the proposed protocol is effective in resisting *certificate poisoning* attacks, while still maintaining all of the decentralization characteristics of the original protocol.

6.3 Contributions restatement

In Section 1.5 it is stated that:

> The **main contribution** of this thesis is:
>
> - A network protocol for the exchange of certifications on cryptographic identities that can reliably prevent certificate poisoning attacks against OpenPGP keys in a Web-of-Trust environment, with minimal modification to server software and without requiring modifying existing end-user tools.
>
> This allows us to also enumerate **secondary contributions**:
>
> 1. Give the key owner the ability to control which certifications are to be published, granting them reputation control.
> 2. Allow the WoT decentralized trust model to remain relevant and sustainable for communities based on OpenPGP.

The main contribution is clearly the reason why this work started: not only it is seen as the target to achieve, but represents in a way going back to the baseline situation that existed before the appearance of the *certificate poisoning* attacks.

Secondary contribution 1 provides a very valuable addition to the WoT model: as a direct effect of 1PA3PC, WoT users are now able to accept or repeal certifications appended to their identities. While this is not part of OpenPGP's original design goals, it does provide the important ability for a user *not to be associated* with any other random user that certifies their key. Given that WoT graphs can be used to map the social environment of a person, we consider it of great importance to be able to reject given associations.

As for secondary contribution 2, our work also achieves this goal. As we near the final stages of this project, the *Sequoia* suite has incorporated the proposed functionality.[1] We expect other implementations to gradually follow suit.

[1] The `list_attestations` function has been proposed in https://gitlab.com/sequoia-pgp/sequoia-sq/-/issues/94 and accepted as of August 13, 2024.

An important discussion to have in the wider OpenPGP community before proposing incorporation of our work in the *Hockeypuck* keyserver is whether it could lead to a situation termed as *death by kindness*: despite being over 30 years old, the OpenPGP WoT has never had massive adoption. The currently active set of users[2] is estimated to be among the tens of thousands. As discussed in Subsection 2.6.5, less than 15% of keyserver users ever register signatures to their keys, and we can expect that a majority of OpenPGP users do not ever upload their keys to said network.

The obtained result is partly due to the usability barrier of the relevant toolset. If further complications are to be added to get signatures, it might lead to lowering usage, reducing even further the WoT's significance (hence the term *death by kindness*). Unlike client tools, 1PA3PC adoption requires all of the keyservers participating in a network to run with our proposed protocol (as otherwise the delta between servers would grow too large and end up breaking the *Gossip*-based synchronization).

6.4 Comparison

Chapter 3 reviewed several works related in some way to this proposal. Section 3.1 refers, as a starting point, to Kahn Gillmor 2019a, which was drafted shortly after the certificate poisoning attack surfaced. This draft outlines several possible implementations to protect a keystore from different kinds of attack, and the credit for introducing the 1PA3PC concept remains Kahn Gillmor's. Our work formalizes, implements and validates this idea.

The works covered in Section 3.2 propose means for key discovery other than the use of public keyservers. Their answers cover thoroughly the attack our work targets, as well as others, but in the case of DANE (Hoffman and Schlyter 2012; Wouters 2016) and WKD (Koch 2021b), do so by abandoning all of the WoT's properties, as third party signatures are stripped from the served keys. This is mostly true also for TOFU for OpenPGP (Walfield and Koch 2016) and Autocrypt (Kahn Gillmor 2021): while an interested user can fetch keys from a keyserver to build a trust path to any given stripped certificate received i.e. attached to an e-mail on a first contact (as Autocrypt does), the interaction model itself *shifts trust* to the low attack probability of a first contact. Hence, while TOFU for OpenPGP and Autocrypt do not solve certificate poisoning, they *route around it* by discouraging WoT usage. ClaimChain (Kulynych et al. 2018) keeps the WoT as a core foundation for identity assurance, but given the implementation is shifted to a blockchain implementation, with certifications recorded in a distributed ledger, while targeting integration in the OpenPGP ecosystem, it does require a full change of client software, as no widely deployed OpenPGP client software incorporates its functionality.

[2] As mentioned in Section 6.2 there are millions of users of OpenPGP, but estimations on the *Web of Trust* usage, based mostly upon observation of the rate of changes in the keyservers' data, is much lower.

As for the ideas presented in Section 3.4, while contrasting with those presented above, they all aim at fixing different problems in the keyserver network *without discarding* either the keyserver network itself or the WoT model. The BlockPGP keyserver (Yakubov et al. 2020) is prompted by the realization that the keyserver network's viability is hampered by several issues that diminish the trust a user can have on any of the participating servers, mainly due to synchronization delays and the inability to reject fake or invalid key IDs. The proposal is to replace the *Gossip* protocol by a blockchain implementation, in principle not too different from ClaimChain (Kulynych et al. 2018). Yakubov' model includes the requisite for a certificate holder to validate any signatures done to their key, as our 1PA3PC proposal does. Yakubov implements this idea, though, using a *Proof-of-Authority* consensus model in order to allow for an administrator to remove unwanted modifications from the database; this goes against the lack of centralization that is warranted by the WoT model and that is preserved by our proposal.

Pini 2018 implements a new keyserver software that, besides storing certificate data as unstructured data, analyzes and "explodes" each OpenPGP packet of a received certificate into a series of tables in a relational database system, which much better permits analyzing the keyserver data looking for problematic patterns. As the keyserver implemented by Pini implements the *Gossip* protocol, the information is also stored as OpenPGP packets. The analysis done by Pini allows for the keyserver to reject non-RFC-conforming data, and it would be trivial to set thresholds to discard data packets constituting certificate poisoning.

Rucker 2017 proposes a new keyserver software that includes a novel synchronization protocol instead of *Gossip*, based on Invertible Bloom Filters to estimate small differences in large datasets such as the databases held by keyservers. It explicitly targets efficiency in key synchronization, not allowing for further validation, and limits validation on received packets to ensuring they are properly formatted OpenPGP packets, version 4 or higher. As such, then, this work proposes a different synchronization strategy, but does not address the issue of controlling certificate poisoning.

Lee 2019 describes the *Hagrid* keyserver, written by the *Sequoia* project, and used for the operation of the `keys.openpgp.org` keyserver. *Hagrid* is defined as a *verifying OpenPGP key server* (Breitmoser et al. 2022): it does not accept random OpenPGP data, but only valid certificates, and before redistributing any of them, it requires the certificate holder to confirm, using a link sent by e-mail, they control the relevant identity. This has the direct implication of not supporting synchronization at all (Sequoia Project 2022). `keys.openpgp.org` also specifies it should be used "for key distribution and discovery, not as a *de facto* certification authority", so it does not distribute third party signatures. Our work aims at keeping the WoT viable and able to operate over a decentralized network.

Mueller 2020's proposal focuses on facilitating OpenPGP users to update their local keystores, so as to be able to trust the information therein to be fresh (mainly relevant when dealing with certificate revocations). It proposes

for an approach that does not leak to the keyservers the social composition of each participant, by downloading neighboring certificates with a shared key prefix. Mueller proposes that each keyserver should publish a list of certificates structured as a Bloom filter presenting those that have received updates in a given period, with granularity of days, to offset for the increased network bandwidth requirement. This requires changes far deeper both to the keyserver and client software than what we are proposing, and was deemed unnecessary given the fast synchronization results shown in Section 5.2.

Section 3.5 presents works that highlight the inadequacies of the OpenPGP protocol to cope with the evolution of interpersonal communications. The *Off-the-Record* (OTR) protocol (Borisov, Goldberg, and Brewer 2004) introduces a radically different set of cryptographic guarantees to the communications it protects, aiming at providing a channel that models person-to-person informal chat rather than epistolary communication.

Halpin 2020 starts from the usability issue that plagues the OpenPGP ecosystem, and concludes that said issues stem from a standard that is too complex and makes several wrong design decisions, opening up the protocol to several of the weaknesses, among them, those reviewed in Section 2.6. Halpin's work concludes with a series of recommendations done to the OpenPGP standards working group, but points out that "one option is simply to abandon fixing PGP and start from a cleaner design that natively supports modern cryptography and a per-message AEAD construction". Further, it connects with our work by concluding:

> Technically, decentralized public key infrastructure is an open research problem, both in terms of cryptography and usability. The usability of keys as identifiers is the root usability problem of PGP, and without a breakthrough, new standards will not fix this unless keys are managed on behalf of the user – which goes against the values of decentralization and user control.
>
> Simply deprecating OpenPGP without further work misses the real reason why there are still many supporters of OpenPGP for encryption in the developer community: OpenPGP is viewed as open and decentralized as opposed to current fragmented secure messaging applications.

6.5 Future work

The WoT landscape has not stood still since work on this project began. In July 2024, RFC 9580 has been published (Wouters et al. 2024). This document, termed the *crypto refresh* during its multiyear development process, drives the compatibility that OpenPGP implementations follow, and surely deserves analyzing and harmonizing both with the problematic and solutions presented so far.

Presenting a security-relevant protocol backed by experimental evaluation can only prove the protocol works in the situation as foreseen. As future work, a valuable addition to the presented work can be implementing this protocol (or possibly the whole of HKP) for formal validation in a validation framework such as the Tamarin theorem prover (Basin et al. 2022). This kind of proof would bring much tighter security guarantees compared to what has been achieved by this work.

The work we present assumes a network of well-behaved keyservers. As mentioned in Section 6.3, a number of keyservers running the traditional protocol peering with a network of 1PA3PC-based servers could continue to work and sync, but an ever-growing delta would increasingly add *dead weight* to the communications, until potentially a breaking point is reached. Finding said breaking point, and finding strategies to avoid their occurrence (including the modification of the *Gossip* protocol, as several authors mentioned in Section 3.4 have suggested) can lead to interesting research. Naturally, the potential for *death by kindness* (also mentioned in 6.3) can be heightened by adding further steps to key upload and exchange, so it must be necessarily weighed in for any potential proposal.

The UID poisoning problem, introduced in Subsection 2.6.3, can also be tackled with a protocol similar to the one presented in this work. This would probably require adding keyserver validation information inside the certificates, and requiring an intermediate step where the keyserver *knows about* a key but does not yet *trust* in it (only keeps it in a *staging area* for a given time, until its UIDs are validated). This could lead to a federated alternative to centralized services such as `keys.openpgp.org`, presented in Subsection 3.4.4.

References

Aas, Josh et al. (2019). "Let's Encrypt: An Automated Certificate Authority to Encrypt the Entire Web". In: *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*. CCS '19. London, United Kingdom: Association for Computing Machinery, pp. 2473–2487. ISBN: 9781450367479. DOI: 10.1145/3319535.3363192. URL: https://doi.org/10.1145/3319535.3363192 (cit. on pp. 15, 23).

Abdul-Rahman, Alfarez (1997). "The Pgp Trust Model". In: *EDI-Forum: the Journal of Electronic Commerce*. Vol. 10. 3, pp. 27–31. URL: https://ldlus.org/college/WOT/The_PGP_Trust_Model.pdf (cit. on p. 5).

Almohri, Hussain and David Evans (2017). *TLS Outside The Web*. URL: https://tlseminar.github.io/tls-outside-the-web/ (cit. on p. 21).

Alon, Noga, Amnon Barak, and Udi Manber (1987). "On Disseminating Information Reliably Without Broadcasting". In: *Proceedings — IEEE International Conference on Distributed Computing Systems*, pp. 74–81 (cit. on p. 5).

Amann, Johanna et al. (2017). "Mission Accomplished? HTTPS Security after DigiNotar". In: *Proceedings of IMC'17*, pp. 325–340. URL: https://dl.acm.org/doi/abs/10.1145/3131365.3131401 (cit. on p. 26).

Angwin, Julia (Feb. 2015). *The World's Email Encryption Software Relies on One Guy, Who is Going Broke*. URL: https://www.propublica.org/article/the-worlds-email-encryption-software-relies-on-one-guy-who-is-going-broke (cit. on p. 22).

Arkko, Jari and Pekka Nikander (2004). "Weak Authentication: How to Authenticate Unknown Principals without Trusted Parties". In: *Security Protocols*. Ed. by Bruce Christianson et al. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 5–19. ISBN: 978-3-540-39871-4. URL: http://www.pnr.iki.fi/publications/cam2002b.pdf (cit. on pp. 18–19).

Atkins, Derek, William Stallings, and Philip Zimmermann (1996). "PGP Message Exchange Formats". In: 1991. URL: https://tools.ietf.org/html/rfc1991 (cit. on p. 22).

Azul et al. (Mar. 2021). *User manual: sq - A command-line frontend for Sequoia, an implementation of OpenPGP*. URL: https://docs.rs/crate/sequoia-sq/0.26.0 (cit. on p. 76).

Bagdasaryan, Eugene (2016). *Gossip protocols*. URL: http://www.cs.cornell.edu/courses/cs6410/2016fa/slides/19-p2p-gossip.pdf (cit. on p. 5).

Barenghi, Alessandro et al. (2015). "Challenging the Trustworthiness of PGP: Is the Web-of-Trust Tear-Proof?" In: *Computer Security – ESORICS 2015*. Ed. by Günther Pernul, Peter Y A Ryan, and Edgar Weippl. Cham: Springer International Publishing, pp. 429–446. ISBN: 978-3-319-24174-6. URL: https://link.springer.com/chapter/10.1007%5C%2F978-3-319-24174-6_22 (cit. on pp. 44, 50).

Basin, David et al. (2022). "Tamarin: verification of large-scale, real-world, cryptographic protocols". In: *IEEE Security & Privacy* 20.3, pp. 24–32. URL: https://ieeexplore.ieee.org/abstract/document/9768326/ (cit. on p. 86).

Berkowsky, Jake A and Thaier Hayajneh (2017). "Security issues with certificate authorities". In: *2017 IEEE 8th Annual Ubiquitous Computing, Electronics and Mobile Communication Conference (UEMCON)*. IEEE, pp. 449–455. URL: https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=8249081 (cit. on pp. 25–27).

Böck, Hanno (2015). "A look at the PGP ecosystem through the key server data". In: *IACR Cryptol. ePrint Arch.* 2015, p. 262. URL: http://eprint.iacr.org/2015/262 (cit. on p. 44).

Borcherding, Birgit and Malte Borcherding (1998). "Efficient and trustworthy key distribution in webs of trust". In: *Computers & Security* 17.5, pp. 447–454. URL: https://dl.acm.org/doi/abs/10.1016/S0167-4048(98)00001-7 (cit. on pp. 14–15, 17).

Borisov, Nikita, Ian Goldberg, and Eric Brewer (2004). "Off-the-record communication, or, why not to use PGP". In: *Proceedings of the 2004 ACM workshop on Privacy in the electronic society*. Association for Computing Machinery, pp. 77–84. URL: https://doi.org/10.1145/1029179.1029200 (cit. on pp. 54–55, 85).

Braden, Robert (1989). "Requirements for Internet Hosts – Communication Layers". In: *Internet Engineering Task Force (IETF)* 1122. URL: https://www.rfc-editor.org/info/rfc1122 (cit. on p. 11).

Braun, Sven and Anne-Marie Oostveen (2019). "Encryption for the masses? An analysis of PGP key usage". In: *Mediatization Studies* 2, pp. 69–84. DOI: 10.17951/ms.2018.2.69-84. URL: https://reshistorica.journals.umcs.pl/ms/article/view/6947 (cit. on p. 22).

Breitmoser, Vincent et al. (2022). *Hagrid: a verifying keyserver*. URL: https://gitlab.com/keys.openpgp.org/hagrid/ (visited on 03/28/2022) (cit. on pp. 51, 84).

Brennen, V.Alex (2008). *The Keysigning Party Howto*. URL: https://www.cryptnet.net/fdp/crypto/keysigning_party/en/keysigning_party.html (cit. on p. 9).

Brunner, Clemens et al. (2020). "DID and VC:Untangling Decentralized Identifiers and Verifiable Credentials for the Web of Trust". In: *2020 the 3rd International Conference on Blockchain Technology and Applications*. DOI:

10.1145/3446983.3446992. URL: http://dl.acm.org/citation.cfm?id=3446992 (cit. on p. 16).

CA/Browser Forum (Apr. 2021). *Baseline Requirements for the Issuance and Management of Publicly-Trusted Certificates*. Tech. rep. 1.7.4. CA/Browser Forum. 99 pp. URL: https://cabforum.org/baseline-requirements/ (cit. on pp. 15, 24–26).

Callas, Jon et al. (2007). *OpenPGP Message Format*. RFC 4880. URL: https://www.rfc-editor.org/info/rfc4880 (cit. on pp. 9, 19, 28, 31, 40, 46, 70, 75).

Čapkun, Srdjan, Levente Buttyán, and Jean-Pierre Hubaux (2002). "Small Worlds in Security Systems: An Analysis of the PGP Certificate Graph". In: *Proceedings of the 2002 Workshop on New Security Paradigms*. NSPW '02. Virginia Beach, Virginia: Association for Computing Machinery, pp. 28–35. ISBN: 158113598X. DOI: 10.1145/844102.844108. URL: https://doi.org/10.1145/844102.844108 (cit. on p. 16).

Carbone, Richard (2016). *The Age of Encryption*. SANS Institute Reading Room. URL: https://www.sans.org/reading-room/whitepapers/vpns/age-encryption-37397 (cit. on p. 23).

Chen, Jianhua, Fang Miao, and Quanhai Wang (2007). "SSL/TLS-based Secure Tunnel Gateway System Design and Implementation". In: *2007 International Workshop on Anti-Counterfeiting, Security and Identification (ASID)*, pp. 258–261. DOI: 10.1109/IWASID.2007.373739 (cit. on p. 11).

Conti, Mauro, Nicola Dragoni, and Viktor Lesyk (2016). "A Survey of Man In The Middle Attacks". In: *IEEE Communnications Surveys & Tutorials* 18.3, pp. 2027–2051. DOI: 10.1109/COMST.2016.2548426. URL: https://doi.org/10.1109/COMST.2016.2548426 (cit. on p. 14).

Darxus (2002). *sig2dot GPG/PGP Keyring Graph Generator*. URL: http://www.chaosreigns.com/code/sig2dot/ (cit. on p. 22).

Demers, Alan et al. (1987). "Epidemic algorithms for replicated database maintenance". In: *Proceedings of the sixth annual ACM Symposium on Principles of distributed computing*, pp. 1–12 (cit. on p. 5).

Dhamija, Rachna, J. D. Tygar, and Marti Hearst (2006). "Why Phishing Works". In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. CHI '06. Montréal, Québec, Canada: Association for Computing Machinery, pp. 581–590. ISBN: 1595933727. DOI: 10.1145/1124772.1124861. URL: https://doi.org/10.1145/1124772.1124861 (cit. on p. 23).

Diffie, Whitfield and Martin Hellman (1976). "New directions in cryptography". In: *IEEE transactions on Information Theory* 22.6, pp. 644–654 (cit. on pp. 12–13).

Eastlake, Donald E. (Jan. 2011). *Transport Layer Security (TLS) Extensions: Extension Definitions*. RFC 6066. DOI: 10.17487/RFC6066. URL: https://rfc-editor.org/rfc/rfc6066.txt (cit. on p. 25).

Eppstein, David et al. (2011). "What's the Difference? Efficient Set Reconciliation without Prior Context". In: *Proceedings of the ACM SIGCOMM 2011 Conference*. SIGCOMM '11. Toronto, Ontario, Canada: Association for

Computing Machinery, pp. 218–229. ISBN: 9781450307970. DOI: 10.1145/2018436.2018462. URL: https://doi.org/10.1145/2018436.2018462 (cit. on p. 50).

Espiner, Tom (Feb. 2012). *Trustwave sold root certificate for surveillance*. URL: https://www.zdnet.com/article/trustwave-sold-root-certificate-for-surveillance/ (cit. on p. 26).

Fiskerstrand, Kristian (2016a). "An update on the sks-keyservers.net services". In: *OpenPGP.conf*. URL: https://sks-keyservers.net/files/2016-09_OpenPGP-Conf-sks-keyservers.pdf (cit. on p. 20).

— (2016b). *OpenPGP Certificates can not be deleted from keyservers*. URL: https://blog.sumptuouscapital.com/2016/03/openpgp-certificates-can-not-be-deleted-from-keyservers/ (cit. on p. 20).

— (2021). *SKS Keyservers*. URL: https://web.archive.org/web/20220119094712/https://www.sks-keyservers.net/ (cit. on p. 20).

Freier, Alan, Philip Karlton, and Paul Kocher (2011). "The secure sockets layer (SSL) protocol version 3.0". In: 6101. URL: https://www.rfc-editor.org/info/rfc6101 (cit. on p. 21).

Fu, Kevin et al. (2001). "Dos and Don'ts of Client Authentication on the Web". In: *10th USENIX Security Symposium*. USENIX. URL: https://www.usenix.org/legacy/events/sec01/fu/fu.pdf (cit. on p. 15).

Gallagher, Andrew (2021). *What is a Web Key Directory?* GnuPG. URL: https://wiki.gnupg.org/WKD (cit. on p. 39).

— (2024a). *SKS peer mesh graphs*. URL: https://spider.pgpkeys.eu/graphs/ (visited on 08/13/2024) (cit. on p. 20).

— (2024b). *The State of the Keyservers in 2024*. URL: https://blog.pgpkeys.eu/state-keyservers-2024.html (cit. on p. 82).

Gibson, Steve (2013). *Security Certificate Revocation Awareness: The case for "OCSP Must-Staple"*. URL: https://www.grc.com/revocation/ocsp-must-staple.htm (cit. on p. 25).

Gil, David Leon (2013). *OpenPGPv4 long keyid collision test cases?* URL: https://mailarchive.ietf.org/arch/msg/openpgp/Al8DzxTH2KT7vtFAgZ1q17Nub_g/ (cit. on p. 30).

Gilbert, Nestor (2021). Finances Online. URL: https://financesonline.com/number-of-email-users/ (visited on 08/15/2021) (cit. on p. 39).

Gillmor, Daniel Kahn (Mar. 2024). *Stateless OpenPGP Command Line Interface*. Internet-Draft draft-dkg-openpgp-stateless-cli-10. Work in Progress. Internet Engineering Task Force. 69 pp. URL: https://datatracker.ietf.org/doc/draft-dkg-openpgp-stateless-cli/10/ (cit. on p. 71).

Górny, Michał (2019). *SKS poisoning, keys.openpgp.org / Hagrid and other non-solutions*. URL: https://blogs.gentoo.org/mgorny/2019/07/04/sks-poisoning-keys-openpgp-org-hagrid-and-other-non-solutions/ (cit. on p. 52).

Halpin, Harry (2020). "SoK: Why Johnny Can't Fix PGP Standardization". In: *Proceedings of the 15th International Conference on Availability, Reliability and Security*. ARES '20. Virtual Event, Ireland: Association for Computing Machinery. ISBN: 9781450388337. DOI: 10.1145/3407023.3407083. URL: https://doi.org/10.1145/3407023.3407083 (cit. on pp. 55, 58, 85).

Hansen, Robert J. (2018). *Remove public key from keyserver*. URL: https://lists.gnupg.org/pipermail/gnupg-users/2018-January/059734.html (visited on 05/02/2022) (cit. on p. 31).

— (2019). *SKS Keyserver Network Under Attack*. URL: https://gist.github.com/rjhansen/67ab921ffb4084c865b3618d6955275f (cit. on p. 34).

Hein, Blaine (2013). *PKI Trust Models: Whom do you trust?* SANS Institute Reading Room. URL: https://www.sans.org/reading-room/whitepapers/vpns/pki-trust-models-trust-36112 (cit. on p. 19).

Heninger, Nadia et al. (2012). "Mining your Ps and Qs: Detection of widespread weak keys in network devices". In: *21st {USENIX} Security Symposium ({USENIX} Security 12)*, pp. 205–220. URL: https://www.usenix.org/system/files/conference/usenixsecurity12/sec12-final228.pdf (cit. on p. 45).

Herzberg, Amir and Ahmad Jbara (Oct. 2008). "Security and Identification Indicators for Browsers against Spoofing and Phishing Attacks". In: *ACM Trans. Internet Technol.* 8.4. ISSN: 1533-5399. DOI: 10.1145/1391949.1391950. URL: https://doi.org/10.1145/1391949.1391950 (cit. on p. 23).

Herzberg, Amir and Hemi Leibowitz (2016). "Can Johnny finally encrypt? Evaluating E2E-encryption in popular IM applications". In: *Proceedings of the 6th Workshop on Socio-Technical Aspects in Security and Trust*, pp. 17–28. URL: https://dl.acm.org/doi/pdf/10.1145/3046055.3046059 (cit. on p. 18).

Hoffman, Paul E. and Jakob Schlyter (Aug. 2012). *The DNS-Based Authentication of Named Entities (DANE) Transport Layer Security (TLS) Protocol: TLSA*. RFC 6698. DOI: 10.17487/RFC6698. URL: https://rfc-editor.org/rfc/rfc6698.txt (cit. on pp. 37, 83).

Hofmann, Johann (Aug. 2019). *Intent to Ship: Move Extended Validation Information out of the URL bar*. URL: https://groups.google.com/g/firefox-dev/c/6wAg_PpnlY4 (cit. on p. 24).

Horowitz, Mark (1997). "PGP public key server". MA thesis. MIT. URL: https://www.mit.edu/afs/net.mit.edu/project/pks/thesis/paper/thesis.html (cit. on p. 5).

Hosner, Charlie (2004). *OpenVPN and the SSL VPN Revolution*. SANS Institute Reading Room. URL: https://www.sans.org/reading-room/whitepapers/vpns/openvpn-ssl-vpn-revolution-1459 (cit. on p. 21).

Hunt, Troy (Aug. 2019). *Extended Validation Certificates are (Really, Really) Dead*. URL: https://www.troyhunt.com/extended-validation-certificates-are-really-really-dead/ (cit. on p. 24).

Jallad, Kahil, Jonathan Katz, and Bruce Schneier (2002). "Implementation of chosen-ciphertext attacks against PGP and GnuPG". In: *International Conference on Information Security*. Springer, pp. 90–101. URL: http://www.cs.umd.edu/~jkatz/papers/pgp-attack.pdf (cit. on p. 57).

Johansen, Christian et al. (2017). *Comparing implementations of secure messaging protocols*. Tech. rep. ISBN 97-82-7368-440-0. Universitetet i Oslo. URL: https://www.duo.uio.no/handle/10852/60949 (cit. on p. 18).

Jøsang, Audun (1999). "An Algebra for Assessing Trust in Certification Chains". In: *Proc. Network and Distributed Systems Security Symposium, 1999 (NDSS'99)*. The Internet Society. URL: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.60.4065 (cit. on pp. 17, 41).

Jøsang, Audun, Elizabeth Gray, and Michael Kinateder (2006). "Simplification and analysis of transitive trust networks". In: *Web Intelligence and Agent Systems: An International Journal* 4.2, pp. 139–161. URL: http://www.sce.carleton.ca/faculty/esfandiari/agents/papers/josang.pdf (cit. on p. 5).

Kahn Gillmor, Daniel (2013). *OpenPGP Key IDs are not useful*. URL: https://web.archive.org/web/20200926000016/https://debian-administration.org/users/dkg/weblog/105 (cit. on p. 30).

— (2016). *Key Discovery Comparison*. The 3rd OpenPGP Email Summit. URL: https://wiki.gnupg.org/OpenPGPEmailSummit201607/KeyDiscoveryComparison (cit. on p. 38).

— (Aug. 2019a). *Abuse-Resistant OpenPGP Keystores*. Internet-Draft draft-dkg-openpgp-abuse-resistant-keystore-04. (Expired draft). Internet Engineering Task Force. 58 pp. URL: https://datatracker.ietf.org/doc/html/draft-dkg-openpgp-abuse-resistant-keystore-04 (cit. on pp. 35, 83).

— (2019b). *OpenPGP Certificate Flooding*. URL: https://dkg.fifthhorseman.net/blog/openpgp-certificate-flooding.html (cit. on p. 34).

— (2021). "Autocrypt: Repensar el cifrado del correo electrónico". In: *Mecanismos de privacidad y anonimato en redes: Una visión transdisciplinaria*. Ed. by Gunnar Wolf. Universidad Nacional Autónoma de México, pp. 153–160. ISBN: 978-607-30-4808-8. URL: https://priv-anon.unam.mx/libro (cit. on pp. 18, 41, 83).

Kelsey, John (2002). "Compression and information leakage of plaintext". In: *International Workshop on Fast Software Encryption*. Springer, pp. 263–276 (cit. on p. 55).

Klafter, Richard and Eric Swanson (2012). *Scallion*. URL: https://github.com/lachesis/scallion (cit. on p. 29).

— (2014). *Evil 32: Check Your GPG Fingerprints*. URL: https://evil32.com (cit. on p. 29).

Kleinjung, Thorsten et al. (2010). "Factorization of a 768-bit RSA modulus". In: *Annual Cryptology Conference*. Springer, pp. 333–350. URL: https://

link.springer.com/content/pdf/10.1007/978-3-642-14623-7_18.pdf (cit. on p. 45).

Koch, Werner (2016). *A Simple Solution to Key Discovery*. Paper presented at OpenPGP.conf. URL: https://www.openpgp-conf.org/program.html#werner (cit. on p. 37).

— (2021a). *Koch: A New Future for GnuPG*. URL: https://lwn.net/Articles/880248/ (cit. on p. 17).

— (May 2021b). *OpenPGP Web Key Directory*. Internet-Draft draft-koch-openpgp-webkey-service-12. Work in Progress. Internet Engineering Task Force. 18 pp. URL: https://datatracker.ietf.org/doc/html/draft-koch-openpgp-webkey-service-12 (cit. on pp. 39, 83).

Krebs, Brian (Nov. 2018). *Half of all Phishing Sites Now Have the Padlock*. URL: https://krebsonsecurity.com/2018/11/half-of-all-phishing-sites-now-have-the-padlock/ (cit. on p. 23).

Krekel, Holger, Karissa McKelvey, and Emil Lefherz (July 2018). "How to Fix Email: Making Communication Encrypted and Decentralized with Autocrypt". In: *XRDS: Crossroads, The ACM Magazine for Students* 24.4, pp. 37–39. ISSN: 1528-4972. URL: https://doi.org/10.1145/3220565 (cit. on p. 18).

Kulynych, Bogdan et al. (2018). "ClaimChain: Improving the Security and Privacy of In-Band Key Distribution for Messaging". In: *Proceedings of the 2018 Workshop on Privacy in the Electronic Society*. WPES'18. Toronto, Canada: Association for Computing Machinery, pp. 86–103. ISBN: 9781450359894. DOI: 10.1145/3267323.3268947. URL: https://doi.org/10.1145/3267323.3268947 (cit. on pp. 42–43, 83–84).

Laurie, Ben, Adam Langley, and Emilia Kasper (June 2013). *Certificate Transparency*. RFC 6962. DOI: 10.17487/RFC6962. URL: https://rfc-editor.org/rfc/rfc6962.txt (cit. on p. 46).

Lawrence, Jason (2018). *Hide UID From Public Key Server By Poison Your Key?* URL: https://lists.gnupg.org/pipermail/gnupg-users/2018-January/059718.html (visited on 05/02/2022) (cit. on p. 30).

Lee, Micah F. (2014). *Trolling the Web of Trust*. URL: https://github.com/micahflee/trollwot (cit. on p. 31).

— (2019). *The Death of SKS PGP Keyservers, and How First Look Media is Handling It*. URL: https://code.firstlook.media/the-death-of-sks-pgp-keyservers-and-how-first-look-media-is-handling-it (cit. on pp. 51, 84).

Lenovo (2015). *SuperFish Vulnerability*. URL: https://support.lenovo.com/mx/en/product_security/superfish (cit. on p. 27).

Levien, Raphael (1995). "Attack resistant trust metrics". PhD thesis. University of California at Berkeley. URL: https://www.levien.com/thesis/thesis.pdf (cit. on p. 5).

Manousis, Antonis et al. (2016). *Shedding Light on the Adoption of Let's Encrypt*. arXiv: 1611.00469 [cs.CR]. URL: https://arxiv.org/pdf/1611.00469.pdf (cit. on p. 23).

Marshall, Casey (2015a). *Hockeypuck*. URL: https://hockeypuck.io/ (visited on 09/26/2022) (cit. on p. 76).
— (2015b). *Hockeypuck — Populating the keyserver*. URL: https://hockeypuck.io/populating.html (visited on 09/23/2024) (cit. on p. 20).
Matsumoto, Stephanos, Pawel Szalachowski, and Adrian Perrig (2015). "Deployment Challenges in Log-Based PKI Enhancements". In: *Proceedings of the Eighth European Workshop on System Security*. EuroSec '15. Bordeaux, France: Association for Computing Machinery. ISBN: 9781450334792. DOI: 10.1145/2751323.2751324. URL: https://doi.org/10.1145/2751323.2751324 (cit. on p. 46).
Mavrogiannopoulos, Nikos and Daniel Kahn Gillmor (2011). "Using OpenPGP Keys for Transport Layer Security (TLS) Authentication". In: *Internet Engineering Task Force (IETF)* 6091. URL: https://www.rfc-editor.org/info/rfc6091 (cit. on p. 21).
Menezes, Alfred J, Paul C Van Oorschot, and Scott A Vanstone (1996). *Handbook of applied cryptography*. CRC press. URL: http://cacr.uwaterloo.ca/hac/ (cit. on pp. 12–14, 19).
Merkle, Ralph C (1978). "Secure communications over insecure channels". In: *Communications of the ACM* 21.4, pp. 294–299 (cit. on p. 12).
Minsky, Yaron, John Klizbe, and Kristian Fiskerstrand (2008). *sks-keyserver*. URL: https://github.com/SKS-Keyserver/sks-keyserver (cit. on p. 20).
Minsky, Yaron and Ari Trachtenberg (2002). "Practical Set Reconciliation". In: *40th Annual Allerton Conference on Communication, Control and Computing*. Vol. 248. URL: https://git.gnunet.org/bibliography.git/plain/docs/practical.pdf (cit. on p. 20).
Minsky, Yaron, Ari Trachtenberg, and Richard Zippel (2003). "Set reconciliation with nearly optimal communication complexity". In: *IEEE Transactions on Information Theory* 49.9, pp. 2213–2218. URL: http://ipsit.bu.edu/documents/ieee-it3-web.pdf (cit. on p. 20).
Minsky, Yaron Moshe (2002). "Spreading rumors cheaply, quickly, and reliably". PhD thesis. Cornell University. URL: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.20.5620&rep=rep1&type=pdf (cit. on pp. 20, 50).
Mueller, Tobias (2020). "Let's Refresh! Efficient and Private OpenPGP Certificate Updates". In: *2020 International Conference on Software, Telecommunications and Computer Networks (SoftCOM)*, pp. 1–6. DOI: 10.23919/SoftCOM50211.2020.9238161. URL: https://ieeexplore.ieee.org/document/9238161 (cit. on pp. 52, 54, 84).
Myers, Michael et al. (1999). "X.509 Internet Public Key Infrastructure Online Certificate Status Protocol - OCSP". In: URL: https://tools.ietf.org/html/rf2560 (cit. on p. 25).
Nguyen, Phong Q (2004). "Can we trust cryptographic software? Cryptographic flaws in GNU Privacy Guard v1. 2.3". In: *International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, pp. 555–570.

URL: https://link.springer.com/content/pdf/10.1007/978-3-540-24676-3_33.pdf (cit. on p. 46).

Nightingale, Jonathan (2011). *Fraudulent *.google.com Certificate*. URL: https://blog.mozilla.org/security/2011/08/29/fraudulent-google-com-certificate/ (cit. on p. 26).

O'Brien, Devon (Aug. 2019). *Upcoming Change to Chrome's Identity Indicators*. URL: https://groups.google.com/a/chromium.org/g/security-dev/c/h1bTcoTpfeI/m/jUTk1z7VAAAJ (cit. on p. 24).

Oppliger, Rolf (1998). "Security at the Internet layer". In: *Computer* 31.9, pp. 43–47. URL: https://ieeexplore.ieee.org/abstract/document/708449 (cit. on p. 11).

Pham, Viet and Tuomas Aura (2011). "Security analysis of leap-of-faith protocols". In: *International Conference on Security and Privacy in Communication Systems*. Springer, pp. 337–355. URL: https://doi.org/10.1007/978-3-642-31909-9_19 (cit. on p. 17).

Pini, Mattia (2018). "PEAKS: adding proactive security to OpenPGP keyservers". MA thesis. URL: http://hdl.handle.net/10589/140111 (cit. on pp. 48–50, 84).

Poddebniak, Damian et al. (2018). "Efail: Breaking S/MIME and OpenPGP email encryption using exfiltration channels". In: *27th {USENIX} Security Symposium ({USENIX} Security 18)*, pp. 549–566. URL: https://www.usenix.org/conference/usenixsecurity18/presentation/poddebniak (cit. on p. 57).

Pramberger, Peter (2010). *Keyserver.pramberger.at terminating*. URL: https://lists.nongnu.org/archive/html/sks-devel/2010-09/msg00009.html (cit. on pp. 31–32).

Prins, J. R. (Sept. 2011). *digiNotar Certificate Authority breach "Operation Black Tulip"*. Tech. rep. 1.0. Fox-IT. 13 pp. URL: https://cryptome.wikileaks.org/0005/diginotar-insec.pdf (cit. on p. 26).

Proton AG (2024). *Proton: Privacy by default*. URL: https://proton.me (visited on 08/16/2024) (cit. on p. 22).

Renaud, Karen, Melanie Volkamer, and Arne Renkema-Padmos (2014). "Why Doesn't Jane Protect Her Privacy?" In: *Privacy Enhancing Technologies PETS 2014*. Ed. by Emiliano De Cristofaro and Steven J. Murdoch. Vol. 8555. Lecture Notes in Computer Science. Springer International Publishing, pp. 244–262. ISBN: 978-3-319-08506-7. URL: https://doi.org/10.1007/978-3-319-08506-7_13 (cit. on p. 18).

Rescorla, Eric (2018). "The Transport Layer Security (TLS) Protocol, Version 1.3". In: *Internet Engineering Task Force (IETF)* 8446. URL: https://www.rfc-editor.org/info/rfc8446 (cit. on pp. 19, 21).

Rivest, R. L., A. Shamir, and L. Adleman (Feb. 1978). "A method for obtaining digital signatures and public-key cryptosystems". In: *Communications of the ACM* 21.2, pp. 120–126. ISSN: 0001-0782. DOI: 10.1145/359340.359342. URL: https://doi.org/10.1145/359340.359342 (cit. on pp. 13–14).

Rivest, Ronald L and Butler Lampson (1996). "SDSI-a simple distributed security infrastructure". In: URL: http://www.schuba.com/christoph/pub/courses/it251-ss2001/papers/sdsi.pdf (cit. on p. 16).

Rose, Scott et al. (2016). *Trustworthy email*. Tech. rep. NIST-SP 800-177. National Institute of Standards and Technology. URL: https://doi.org/10.6028/NIST.SP.800-177r1 (cit. on pp. 15, 19–20, 22).

Rucker, Alexander (2017). *An Efficient PGP Keyserver without Prior Context*. URL: https://www.scs.stanford.edu/17au-cs244b/labs/projects/rucker.pdf (cit. on pp. 50, 84).

Ruoti, Scott et al. (2015). "Why Johnny still, still can't encrypt: Evaluating the usability of a modern PGP client". In: *arXiv preprint arXiv:1510.08555*. URL: https://arxiv.org/abs/1510.08555 (cit. on pp. 27, 40–41).

Ryabitsev, Konstantin (Feb. 2014). *PGP Web of Trust: Core Concepts Behind Trusted Communication*. linux.com. URL: https://www.linux.com/training-tutorials/pgp-web-trust-core-concepts-behind-trusted-communication/ (cit. on p. 5).

Sandhya, S and KA Sumithra Devi (2012). "Analysis of Bluetooth threats and v4. 0 security features". In: *2012 International Conference on Computing, Communication and Applications*. IEEE, pp. 1–4 (cit. on p. 18).

Santesson, Stefan et al. (June 2013). *X.509 Internet Public Key Infrastructure Online Certificate Status Protocol - OCSP*. RFC 6960. DOI: 10.17487/RFC6960. URL: https://rfc-editor.org/rfc/rfc6960.txt (cit. on p. 25).

Schaefer, Heiko (2023). *OpenPGP for application developers*. URL: https://openpgp.dev/book (cit. on p. 9).

Schwenk, Jörg (2022). *Guide to internet Cryptography*. Springer. URL: https://link.springer.com/book/10.1007/978-3-031-19439-9 (cit. on pp. 11, 14).

Sequoia Project (2022). *keys.openpgp.org "FAQ" web page*. URL: https://keys.openpgp.org/about/faq (visited on 05/12/2022) (cit. on pp. 52, 84).

Shaw, David (Mar. 2003). *The OpenPGP HTTP Keyserver Protocol (HKP)*. Internet-Draft draft-shaw-openpgp-hkp-00. Work in Progress. Internet Engineering Task Force. 8 pp. URL: https://datatracker.ietf.org/doc/html/draft-shaw-openpgp-hkp-00 (cit. on pp. 5, 20).

Sheng, Steve et al. (2006). "Why Johnny still can't encrypt: evaluating the usability of email encryption software". In: *Symposium On Usable Privacy and Security*. ACM, pp. 3–4. URL: https://cups.cs.cmu.edu/soups/2006/posters/sheng-poster_abstract.pdf (cit. on pp. 27, 40–41).

Skeeto (2019). *PGP long key ID collision*. URL: https://www.reddit.com/r/crypto/comments/cglgy9/pgp_long_key_id_collision/ (cit. on p. 30).

Stallings, W. (2017). *Cryptography and Network Security: Principles and Practice*. Prentice Hall. ISBN: 978-1292158587 (cit. on pp. 12–13, 15).

Stevens, Marc et al. (2009). "Short chosen-prefix collisions for MD5 and the creation of a rogue CA certificate". In: *Annual International Cryptology Con-*

ference. Springer, pp. 55–69. URL: https://link.springer.com/content/pdf/10.1007/978-3-642-03356-8_4.pdf (cit. on p. 46).
Streib, Drew (2003). *keyanalyze — Analysis of a large OpenPGP ring*. URL: https://web.archive.org/web/20090923190128/http://dtype.org/keyanalyze/ (cit. on p. 16).
Tange, Ole (2021). *sks-keyservers gone. What to use instead?* URL: https://unix.stackexchange.com/questions/656205/sks-keyservers-gone-what-to-use-instead (cit. on p. 62).
Thompson, Christopher et al. (2019). "The web's identity crisis: understanding the effectiveness of website identity indicators". In: *28th USENIX Security Symposium (USENIX Security 19)*, pp. 1715–1732. URL: https://www.usenix.org/system/files/sec19-thompson.pdf (cit. on p. 24).
Topalovic, Emin et al. (2012). "Towards short-lived certificates". In: *Web 2.0 Security and Privacy*, pp. 1–9. URL: https://cseweb.ucsd.edu/~dstefan/cse127-winter19/papers/topalovic:towards.pdf (cit. on p. 27).
Toth, Gabor X and Tjebbe Vlieg (2013). *Public Key Pinning for TLS Using a Trust on First Use Model*. Tech. rep. Technical report, University of Amsterdam. URL: https://www.os3.nl/_media/2012-2013/courses/rp1/p56_report.pdf (cit. on pp. 27, 40).
Varghese, Sam (2018). *Trustico revokes SSL certificates due to stored private keys*. ITWire. URL: https://www.itwire.com/security/trustico-revokes-ssl-certificates-due-to-stored-private-keys.html (cit. on p. 27).
Walfield, Neal H. and Werner Koch (2016). "TOFU for OpenPGP". In: *EuroSec'16: Proceedings of the 9th European Workshop on System Security*, pp. 1–6. URL: https://doi.org/10.1145/2905760.2905761 (cit. on pp. 37, 40, 83).
Wang, Jie and Zachary A Kissel (2015). *Introduction to network security: theory and practice*. John Wiley & Sons (cit. on p. 56).
Wellons, Christopher (2019). *PGP Key Poisoner*. URL: https://github.com/skeeto/pgp-poisoner (cit. on p. 30).
Whitten, Alma and J Doug Tygar (1999). "Why Johnny Can't Encrypt: A Usability Evaluation of PGP 5.0." In: *USENIX Security Symposium*. Vol. 348, pp. 169–184. URL: https://www.usenix.org/legacy/events/sec99/full_papers/whitten/whitten.ps (cit. on pp. 18, 27, 40–41).
Wilson, Kathleen (2016). *Distrusting New WoSign and StartCom Certificates*. Mozilla Security Blog. URL: https://blog.mozilla.org/security/2016/10/24/distrusting-new-wosign-and-startcom-certificates/ (cit. on p. 27).
Wolf, Gunnar (2016). *Stop it with those short PGP key IDs!* URL: https://gwolf.org/2016/06/stop-it-with-those-short-pgp-key-ids.html (cit. on p. 28).

Wolf, Gunnar and Gina Gallegos (2017). "Strengthening a curated web of trust in a geographically distributed project". In: *Cryptologia* 41.5, pp. 459–475. DOI: 10.1080/01611194.2016.1238421 (cit. on pp. 28, 40, 45).

Wolf, Gunnar and Jorge Luis Ortega-Arjona (May 2024). "A Protocol for Solving Certificate Poisoning for the OpenPGP Keyserver Network". In: *Journal of Internet Services and Applications* 15.1, pp. 46–58. DOI: 10.5753/jisa.2024.3810. URL: https://journals-sol.sbc.org.br/index.php/jisa/article/view/3810 (cit. on p. 80).

Woo, Wing Keong (2006). "How to exchange email securely with Johnny who still can't encrypt". PhD thesis. University of British Columbia. URL: https://open.library.ubc.ca/cIRcle/collections/ubctheses/831/items/1.0064951 (cit. on pp. 27, 40–41).

Wouters, Paul (2016). "DNS-Based Authentication of Named Entities (DANE) Bindings for OpenPGP". In: *Internet Engineering Task Force (IETF)* 7929. URL: https://www.rfc-editor.org/info/rfc7929 (cit. on pp. 37, 39, 83).

Wouters, Paul et al. (2024). *OpenPGP*. RFC 9580. URL: https://www.rfc-editor.org/info/rfc9580 (cit. on pp. 70, 85).

Yakamo, K (2018a). *Are PGP key-servers Breaking the Law under the GDPR?* URL: https://medium.com/@mdrahony/are-pgp-key-servers-breaking-the-law-under-the-gdpr-a81ddd709d3e (cit. on p. 32).

— (2018b). *Are SKS keyservers safe? Do we need them?* URL: https://medium.com/@mdrahony/are-sks-keyservers-safe-do-we-need-them-7056b495101c (cit. on p. 32).

— (2019). *Using PGP keyservers for decentralised file storage*. URL: https://github.com/yakamok/keyserver-fs (cit. on p. 32).

Yakubov, Alexander et al. (2020). "BlockPGP: A Blockchain-based Framework for PGP Key Servers". In: *International Journal of Networking and Computing* 10.1, pp. 1–24. DOI: 10.15803/ijnc.10.1_1 (cit. on pp. 16, 32–33, 46–47, 84).

Yamane, Shinji et al. (2003). "Rethinking OpenPGP PKI and OpenPGP Public Keyserver". In: *CoRR* cs.CY/0308015. URL: http://arxiv.org/abs/cs/0308015 (cit. on p. 19).

Zimmermann, Philip (1995). *PGP Source Code and Internals*. Cambridge, Mass: MIT Press. ISBN: 0262240394 (cit. on p. 22).

— (1999). *Why I Wrote PGP*. URL: https://www.philzimmermann.com/EN/essays/WhyIWrotePGP.html (cit. on p. 21).

Appendix A

Source code modifications for the implementation of a 1PA3PC-enabled keyserver

This appendix includes:

- Modifications made to the *Sequoia* command-line utility, `sq`, in Section A.1
- Modifications made to the *Hockeypuck* keyserver software, in Section A.2
- Glue code for interfacing *Hockeypuck* with *Sequoia* whenever it needs to validate new (incoming) certificates), in Section A.3
- Provisioning tool used to set up and run the experiment, in Section A.4

All of the code presented in this Appendix can also be downloaded from the `https://1pa3pc.gwolf.org/` Web address.

A.1 Modifications to *Sequoia*

While the command-line `sq` tool does provide a way to create attestations by means of the `attest-certifications` subcommand, as of the moment this work is carried out, it does not yet provide the functionality to *verify* the attestation list. We identify the need to modify the `sq` command and create a `list-attestations` command.

In order to work with the same `sq` version deployed in the test systems, and following the instructions in git tag `b89c17`, this repository is *grafted* upon the preceding `sequoia` repository as follows:

```
$ git remote add sequoia https://gitlab.com/sequoia-pgp/sequoia
$ git fetch sequoia
```

```
$ git replace --graft b89c172 82eb0d7
$ git fetch sequoia
$ git checkout sq/v0.27.0
$ git switch -c list_attestation
```

The following files are identified as relevant for implementing the required function:

sq/src/sq.rs Entry points for the `sq` binary. Dispatches subcommands to each of the `SqSubcommands::*` modules.

sq/src/sq_cli.rs Implements the subcommands for the `sq key` namespace.

sq/src/commands/key.rs Implements the actual logic dispatched when the `key` subcommands are called.

openpgp/src/cert/amalgamation.rs This file defines the module named `ValidUserIDAmalgamation`. One of the functions defined by it is `attestated_certifications()`, which returns the userid's attested third-party certifications.

The full modifications for implementing the `list_attestations()` functionality, with its main code in the `sq/src/commands/key.rs` file, but including modifications also to `sq/src/sq-usage.rs` (documenting the changes in the user-queriable help produced by the program) and to `sq/src/sq_cli.rs` (the logic to actually call `list_attestations()` when so requested by the user) follows, in a `diff` format:

```
diff --git a/sq/src/commands/key.rs b/sq/src/commands/key.rs
index b132adf4..0dd1a910 100644
--- a/sq/src/commands/key.rs
+++ b/sq/src/commands/key.rs
@@ -8,7 +8,7 @@ use crate::openpgp::Result;
 use crate::openpgp::armor::{Writer, Kind};
 use crate::openpgp::cert::prelude::*;
 use crate::openpgp::packet::prelude::*;
-use crate::openpgp::packet::signature::subpacket::SubpacketTag;
+use crate::openpgp::packet::signature::subpacket::{SubpacketTag, SubpacketValue};
 use crate::openpgp::parse::Parse;
 use crate::openpgp::policy::{Policy, HashAlgoSecurity};
 use crate::openpgp::serialize::Serialize;
@@ -33,6 +33,7 @@ use crate::sq_cli::KeyUseridStripCommand;
 use crate::sq_cli::KeyExtractCertCommand;
 use crate::sq_cli::KeyAdoptCommand;
 use crate::sq_cli::KeyAttestCertificationsCommand;
+use crate::sq_cli::KeyListAttestationsCommand;
 use crate::sq_cli::KeySubcommands::*;

 pub fn dispatch(config: Config, command: KeyCommand) -> Result<()> {
@@ -43,6 +44,7 @@ pub fn dispatch(config: Config, command: KeyCommand) ->
     Result<()> {
         ExtractCert(c) => extract_cert(config, c)?,
         Adopt(c) => adopt(config, c)?,
         AttestCertifications(c) => attest_certifications(config, c)?,
+        ListAttestations(c) => list_attestations(config, c)?,
     }
     Ok(())
 }
```

```
@@ -714,6 +716,53 @@ fn adopt(config: Config, command: KeyAdoptCommand) ->
     Result<()> {
     Ok(())
 }

+fn list_attestations(config: Config,  command: KeyListAttestationsCommand)
+                     -> Result<()> {
+    let input = open_or_stdin(command.key.as_deref())?;
+    let key = Cert::from_reader(input)?;
+    let fpr = key.fingerprint();
+    let creation_time = std::time::SystemTime::now();
+    // From the packets that make up the key, pick only those with
+    // valid crypto, and which mke sense at this particular time
+    let valid_cert = key.with_policy(&config.policy, creation_time)?;
+
+    let mut num_attested = 0;
+    let mut att_certs: Vec<String> = Vec::new();
+    for uid in valid_cert.userids() {
+        for att in uid.attested_certifications() {
+            // Attested cetifications make sense currently only in V4
+            // signatures.
+            if let Signature::V4(v4) = att {
+                let hashed = v4.hashed_area();
+                for subpacket in hashed {
+                    // We want to report the fingerprint of the
+                    // certificate issuer that's being attested
+                    match subpacket.value() {
+                        SubpacketValue::IssuerFingerprint(fp) => {
+                            num_attested += 1;
+                            att_certs.push(fp.to_string());
+                        }
+                        _ => {}
+                    }
+                }
+            }
+        }
+    }
+
+    if num_attested == 0 {
+        println!("{} certifications attested for key {}.", num_attested, fpr);
+    } else if num_attested == 1 {
+        println!("{} certification attested for key {}:", num_attested, fpr);
+    } else {
+        println!("{} certifications attested for key {}:", num_attested, fpr);
+    }
+    for att in att_certs {
+        println!("  {}", att);
+    }
+
+    Ok(())
+}
+
 fn attest_certifications(config: Config, command: KeyAttestCertificationsCommand)
                          -> Result<()> {
     // Attest to all certifications?
diff --git a/sq/src/sq-usage.rs b/sq/src/sq-usage.rs
index 186cd0f7..5053da7f 100644
--- a/sq/src/sq-usage.rs
+++ b/sq/src/sq-usage.rs
@@ -393,6 +393,8 @@
 //!             Converts a key to a cert
 //!     attest-certifications
 //!             Attests to third-party certifications
+//!     list-attestations
+//!             List certification attestations
 //!     adopt
 //!             Binds keys from one certificate to another
 //!     help
```

```
@@ -767,6 +769,30 @@
 //! $ sq key attest-certifications --none juliet.pgp
 //! ```
 //!
+//! ### Subcommand key list-attestations
+//!
+//! ```text
+//!
+//! Present a list of attestations present in the certification.
+//!
+//! USAGE:
+//!     sq key list-attestations [KEY]
+//!
+//! ARGS:
+//!     <KEY>
+//!             Lists attestations on KEY
+//!
+//! OPTIONS:
+//!     -h, --help
+//!             Print help information
+//!
+//!
+//! EXAMPLES:
+//!
+//! # Shows all the attestations present on the key
+//! $ sq key list-attestations juliet.pgp
+//! ```
+//!
 //! ### Subcommand key adopt
 //!
 //! ```text
diff --git a/sq/src/sq_cli.rs b/sq/src/sq_cli.rs
index f477e9d9..1a46ff43 100644
--- a/sq/src/sq_cli.rs
+++ b/sq/src/sq_cli.rs
@@ -1741,6 +1741,7 @@ pub enum KeySubcommands {
     ExtractCert(KeyExtractCertCommand),
     Adopt(KeyAdoptCommand),
     AttestCertifications(KeyAttestCertificationsCommand),
+    ListAttestations(KeyListAttestationsCommand),
 }

 #[derive(Debug, Args)]
@@ -2267,6 +2268,31 @@ pub struct KeyAttestCertificationsCommand {

 }

+#[derive(Debug, Args)]
+#[clap(
+    name = "list-attestations",
+    display_order = 210,
+    about = "List certification attestations",
+    long_about =
+"
+Present a list of attestations present in the certification.
+",
+    after_help =
+"
+EXAMPLES:
+
+# Shows all the attestations present on the key
+$ sq key list-attestations juliet.pgp
+",
+)]
+pub struct KeyListAttestationsCommand {
+    #[clap(
+        value_name = "KEY",
+        help = "Lists attestations on KEY"
```

```
+    )]
+    pub key: Option<String>,
+}
+
 #[derive(Parser, Debug)]
 #[clap(
     name = "wkd",
```

A.2 Modfications to *Hockeypuck*

The keyserver software itself, *Hockeypuck*, has to be modified so that it validates each of the received packets upon a certificate insertion or update. Given this validation is done no matter how the update or insertion are received (this is, either directly via the HKP interface or as part of a *Gossip* update), the modification made in the `storage/storage.go` file, which defines the API needed to implement the HKP service's storage backend. The validations for either key insertions or updates are carried out in the `UpsertKey()` function, that receives as its two parameters a `Storage` and a `openpgp.PrimaryKey` objects, and returns a `Keychange` object and a possible `error` (or `nil`). The needed modifications are:

```
Index: hockeypuck/packaging/src/gopkg.in/hockeypuck/hkp.v1/storage/storage.go
===================================================================
--- hockeypuck.orig/packaging/src/gopkg.in/hockeypuck/hkp.v1/storage/storage.go
+++ hockeypuck/packaging/src/gopkg.in/hockeypuck/hkp.v1/storage/storage.go
@@ -23,6 +23,11 @@ import (
        "io"
        "time"

+       "io/ioutil"
+       "os"
+       "os/exec"
+       "strings"
+
        "gopkg.in/errgo.v1"

        "gopkg.in/hockeypuck/openpgp.v1"
@@ -184,12 +189,73 @@ func firstMatch(results []*openpgp.Prima

 func UpsertKey(storage Storage, pubkey *openpgp.PrimaryKey) (kc KeyChange, err
      error) {
        var lastKey *openpgp.PrimaryKey
+       var errMsg = ""
+
        lastKeys, err := storage.FetchKeys([]string{pubkey.RFingerprint})
        if err == nil {
                // match primary fingerprint -- someone might have reused a
                     subkey somewhere
                lastKey, err = firstMatch(lastKeys, pubkey.RFingerprint)
        }
+
+       // -*- hockeypuck_pcic patch starts -*-
+
+       // We will be checking the received key, whether or not it is
+       // found in the DB. For that reason, to avoid clobbering err,
+       // we store its result in new_key_not_found (to be used a
+       // couple of lines below)
+       new_key_not_found := false
+       if IsNotFound(err) {
+               new_key_not_found = true
```

```
+       }
+
+       // Temporary file to be used for communication with Sequoia
+       // for attestation validation
+       f, err := ioutil.TempFile("", "hkp-*")
+       if err != nil {
+               return nil, errgo.Newf("Could not create temporary file")
+       }
+
+       defer f.Close()
+       defer os.Remove(f.Name())
+
+       // Clearly bad style: open a predictably-named log file to
+       // record our modifications. In production code, this has to
+       // be eviscerated!
+       log_f, err := os.OpenFile("/tmp/modif.log",
    os.O_APPEND|os.O_WRONLY|os.O_CREATE, 0600)
+       if err != nil {
+               return nil, errgo.Newf("Could not append to modification log")
+       }
+       defer log_f.Close()
+
+       // Write the ASCII-armored key certificate to check to the temporary file.
+       openpgp.WriteArmoredPackets(f, []*openpgp.PrimaryKey{pubkey})
+
+       out, err := exec.Command("/usr/local/bin/filter_certs", "--file",
    f.Name() ).Output()
+       out_str := strings.TrimSpace(string(out))
+       if err != nil {
+               errMsg = strings.Join([]string{errMsg,
+                       fmt.Sprintf("Error verifying with an external command:
    %q\n", err)},
+                       "\n")
+               log_f.WriteString(errMsg)
+               return KeyNotChanged{}, errgo.New(errMsg)
+       }
+
+       // The filter returns an ASCII-armored key certificate as
+       // STDOUT. Parse it back into a openpgp.PrimaryKey.
+       new_kr, err := openpgp.ReadArmorKeys( strings.NewReader(out_str) )
+       if err != nil {
+               return KeyNotChanged{}, nil
+       }
+
+       pubkeys := new_kr.MustParse()
+       if len(pubkeys) != 1 {
+               return KeyNotChanged{}, errgo.Newf("Unexpected answer:
    verification answer should contain a single key (has %d)", len(pubkeys))
+       }
+       pubkey = pubkeys[0]
+
+       // -*- hockeypuck_pcic patch ends -*-
+
+       if new_key_not_found {
+               _, err = storage.Insert([]*openpgp.PrimaryKey{pubkey})
+               if err != nil {
+                       return nil, errgo.Mask(err)
```

As explained in Subsection 5.1.3, due to versioning differences, the decision is made to use a *Sequoia* attestation listing service in a different virtual server than the *Hockeypuck* keyservers. Thus, the `UpsertKey()` function calls a *glue code* for this communication; said code is presented next.

A.3 Glue code for *Hockeypuck* to query *Sequoia*

Two Perl programs are developed to be used as *glue code*, this is, to aid a *Hockeypuck* server request a validation service from *Sequoia* installed in a different system. As mentioned in Subsection 5.1.3, this is done due to versioning differences, and should not be part of a production setting. Even more, given these programs *pipe* network traffic to a system binary, care must be taken never to expose this to an open, potentially hostile network. It is necessary, though, for faithfully reproducing the experimental setup we carry out.

The `tcp_to_sq.pl` program opens a listening TCP/IP socket in port 3333 of the server hosting *Sequoia* and writes whatever is received from any incoming connections on said socket to a temporary file, until a line indicating the end of the OpenPGP information is received (with the `END PGP` marker).

Subsequently, *Sequoia* is called with the `sq key list-attestations` command on the given filename, and its output (the list of attested signatures as part of a given certificate) are sent back to the originating client:

```
#!/usr/bin/perl
use strict;
use Capture::Tiny qw(capture);
use IO::Socket qw(AF_INET SOCK_STREAM);
use File::Temp;

my @cmd = qw(/usr/local/bin/sq key list-attestations);
$|=1;

my $server = IO::Socket->new(
    Domain =>  AF_INET,
    Type => SOCK_STREAM,
    Proto => 'tcp',
    LocalHost => '0.0.0.0',
    LocalPort => 3333,
    ReusePort => 1,
    Listen => 5,
) || die "Can't open socket: $IO::Socket::errstr";

while (1) {
    my ($conn, $buf, $keyfile, $filename, $out, $err, $exited);
    $conn = $server->accept();

    $keyfile = File::Temp->new();
    $filename = $keyfile->filename;

    while ($buf = <$conn>) {
        print $keyfile $buf;
        last if $buf =~ /END PGP/
    }
    $keyfile->flush;

    ($out, $err, $exited) = capture { system @cmd, $filename };
    if ($exited) {
        warn "Error running «@cmd $filename»: $exited";
        warn $err;
    }
    print $conn $out;
}

$server->close();
```

A second *glue* program is deployed on each of the *Hockeypuck* server sys-

tems: `filter_certs`. This program, called from our modifications to *Hockeypuck* (see Appendix Section A.2), calls *Sequoia* (the version available in the *Hockeypuck* systems, not affected by the versioning conflict mentioned above) to split a certificate into its corresponding individual packets. Any non-OpenPGP-conforming data is dropped. Signature packets are verified to be attested (by sending them to the `tcp_to_sq.pl` program presented above). A certificate including only the attested certifications is then rebuilt and returned to the keyserver.

```
#!/usr/bin/perl -w
use Capture::Tiny qw(capture);
use File::Copy;
use File::Glob ':bsd_glob';
use File::Temp qw(tempdir);
use Getopt::Long;
use IO::Socket qw(AF_INET SOCK_STREAM SHUT_RD);
use strict;

my ($stdout, $stderr, $exited);
my (%conf, $dir, $cert_file, @pkts, $valid_cert);

# Defaults
%conf = (debug => 0,
         attest_port => 3333,
         attest_ip => '10.0.3.250');
GetOptions(\%conf, 'debug', 'file=s', 'attest_port=i', 'attest_ip=s');

$dir = tempdir(CLEANUP => 1);
$cert_file = join('/', $dir, 'cert.pgp');
move($conf{file}, $cert_file);
chdir($dir);

# Certificates always have to be split, checked and joined, as they
# will always bear a signature (even if it's a self-sig).
($stdout, $stderr, $exited) = capture {system qw(sq packet split), $cert_file};
die "Could not split packets from $cert_file" unless $exited == 0;

@pkts = valid_pkts_from_cert();
$valid_cert = join_pkts(@pkts);
print $valid_cert;
exit 0;

# Filter the packets in the certificate, returning only the (indexes
# of) those we consider valid.
sub valid_pkts_from_cert {
    # "sq packet dump" tells us what is the nature of each of the
    # packets. We then assemble the validated certificate with:
    # - All Public-Key or Public-Subkeys packets
    # - All Signature packets where the Issuer Fingerprint is the same of any
    #   of the cert's Public-Key's Fingerprint.
    # - All Signature packets that are part of the list of attestations
    # - Packets that are _not_ signature packets are also passed without
    #   further validation.
    my ($cert_data, $pkt_idx, @pkts, $fpr, @attestations);

    ($stdout, $stderr, $exited) = capture {system qw(sq packet dump), $cert_file};
    die "Could not dump packets in $cert_file" unless $exited == 0;
    $cert_data = $stdout;
    @attestations = parse_attest_list();
    debug("Parsed attestations:", @attestations);

    $pkt_idx = 0;
    for my $pkt (split(/\n(?=\S)/, $cert_data)) {
        $pkt =~ /^(.+) Packet/;
        my $pkt_type = $1;
```

```
            debug("Packet $pkt_idx: $pkt_type");
            # Find this key's fingerprint: if @pkts is empty, this is the
            # first packet we review, and its fingerprint is the master
            # key fingerprint.
            if (scalar @pkts == 0 and $pkt_type eq 'Public-Key' and
                $pkt =~ /\n\s+Fingerprint: ([\dABCDEF]{40})\n/) {
                $fpr = $1;
                debug('Fingerprint:', $fpr);
            }

            if ($pkt_type eq 'Signature') {
                if ($pkt =~ /\n\s+Issuer Fingerprint: ([\dABCDEF]{40})\n/) {
                    my $cert_fpr = $1;
                    if (defined($fpr) and $cert_fpr eq $fpr) {
                        # Self-sig
                        push(@pkts, $pkt_idx);
                    } elsif (scalar(@attestations) > 0 and
                             grep {$_ eq $cert_fpr} @attestations) {
                        debug("Packet $pkt_idx properly attested ☺");
                        # Attested signature
                        push(@pkts, $pkt_idx);
                    } else {
                        # Discard this packet
                        debug("Discarding non-attested $pkt_type packet $pkt_idx");
                    }
                }
            } else {
                push(@pkts, $pkt_idx);
            }
            $pkt_idx += 1;
        }
        debug(scalar(@pkts), " valid packets: ", join(', ', @pkts));
        return @pkts;
    }

    # Once we know which packets we are to keep from the certificate we
    # are validating, join it into a new certificate.
    #
    # Returns the resulting ASCII-armored certificate.
    sub join_pkts {
        my (@pkts, @cmd);
        @pkts = @_;
        @cmd = ('sq', 'packet', 'join');
        for my $pkt (@pkts) {
            my $file = bsd_glob("${cert_file}-${pkt}--*");
            debug("File for $pkt: $file");
            push(@cmd, $file);
        }
        ($stdout, $stderr, $exited) = capture { system(@cmd) };
        die 'Could not join packets ', join(', ', @pkts) unless $exited == 0;
        return $stdout;
    }

    # Returns the list of attested certifications this cert has.
    sub parse_attest_list {
        my (@lines, @atts, $att_hlp, $att_res, $num, $fh, $cert);

        # To send the certificate to be checked to the server, we read it
        # from the file
        open ($fh, '<', $cert_file);
        $cert = join("\n", <$fh>);
        close($fh);

        # Create the client socket to the predefined server
        $att_hlp = IO::Socket->new(
            Domain => AF_INET,
            Type => SOCK_STREAM,
            proto => 'tcp',
```

```
            PeerPort => $conf{attest_port},
            PeerHost => $conf{attest_ip}
            );
        if (! $att_hlp) {
            warn "Could not connect to attestation helper: #IO::Socket::errstr";
            return undef;
        }

        # Send the certificate ($cert) and receive the results ($att_res)
        $att_hlp->send($cert);
        $att_hlp->shutdown(SHUT_RD);
        $att_hlp->recv($att_res, 1024);
        $att_hlp->close;

        # Split the results, looking for the attested key signatures
        @lines = split(/\n/, $att_res);
        if (scalar(@lines) >= 1 and $lines[0] =~ /^(\d+) certification/) {
            $num = $1;
        } else {$num = 0;}

        # No attestations found.
        return if (!defined($num) or $num eq 0);
        for my $lin (@lines) {
            $lin =~ /^\s+([\dABCDEF]{40})/ and push @atts, $1;
        }

        return @atts;
    }

    sub debug {
        return unless $conf{debug};
        print STDERR @_,"\n";
    }
```

A.4 Provisioning tool to set up and run the experiment

Finally, the following code in Ruby has been written as `build_img.rb`. This program requires root privileges to be run. It installs, configures, and runs the needed containers for the experiment. This script grew organically, and in retrospect, it would be clearer and much more maintainable to have it as a *Ansible* playbook or using a similar infrastructure.

This program is built using the `OptionParser` library, allowing to specify from the command line which steps to include or skip, and to set up different values (i.e. number of keyservers, amount of keys to generate or to poison, etc). The main phases it implements are:

1. Initializes and performs basic OS configuration in the specified number of servers as `lxc` containers.

2. Installs and initializes either the "regular" or the modified *Hockeypuck* software in each of the keyservers, configuring a random peering arrangement between them. Each of them initializes a local *PostgreSQL* database.

3. Sets up monitoring for the amount and size of keys for each of the keyservers.

4. Creates, cross-signs, and attests and uploads the specified number of PGP keys following specified configurable parameters.

5. Runs the attack and waits for information flow between keyservers to stabilize

```
#!/usr/bin/ruby
require 'fileutils'
require 'faker'
require 'logger'
require 'net/http'
require 'optparse'
require './labstats'

raise RuntimeError, 'This script has to be run with root permissions' unless
     Process.uid == 0

@conf = {
  :do_configure_peering => true,
  :do_create_database => true,
  :do_create_pgp_keys => true,
  :do_initial_setup => true,
  :do_install_hockeypuck => true,
  :do_install_attest_ck => true,
  :do_pgp_poison => true,
  :do_sign_pgp_keys => true,
  :guest_name => 'hkp%02d',
  :guest_dir => '/var/lib/lxc/%s',
  :hkp_pkg_base => 'hockeypuck-pcic_1.0-1_amd64.deb',
  :hkp_pkg_modif => 'hockeypuck-pcic_1.0-2_amd64.deb',
  :install_pkgs => %w(eatmydata screen ssmtp postgresql sudo git bzr mercurial
                      golang sq openssh-server libcapture-tiny-perl),
  :install_pkgs_attest => %w(eatmydata screen ssmtp netcat-openbsd git cargo
                             pkg-config libssl3 libssl-dev clang llvm
                             nettle-dev openssh-server libcapture-tiny-perl),
  :ip_gateway => '10.0.3.1',
  :ip_network => '10.0.3.%d/24',
  :log_prio => :debug,
  :log_to => 'build_img.log',
  #:log_to => STDOUT,
  :modified_hkp => true,
  :num_hosts => 5,
  :peer_prob => 0.3,
  # How likely is it that a signature will be attested? (in
  # relation with the total signatures created at
  # pgp_sign_prob)
  :pgp_attested_sign_prob => 0.3,
  # How likely is that a given key will sign any other key
  :pgp_sign_prob => 0.01,
  :pgp_keydir_filename => 'pgp_keydir.txt',
  :pgp_keys_dir => 'generated_pgp_keys',
  # Total keys generated will be pgp_keys_per_host ✖ num_hosts
  :pgp_keys_per_host => 100,
  :pgp_keys_to_poison => 5,
  :pgp_attacker_keys => 1000,
  :use_eatmydata => true
}

OptionParser.new do |opts|
  opts.banner = 'Usage: build_img.rb [options]'
  opts.on('-h', '--help', 'Prints this help'){puts opts;exit 0}
  opts.on('-n NUM', '--num_hosts', 'Number of HKP servers to build') { |n|
    @conf[:num_hosts] = n.to_i
  }
  opts.on('-A', '--no-attest-ck',
          'Skip attestation checker setup') {
```

```
    @conf[:do_initial_attest_ck] = false
  }
  opts.on('-C', '--no-config-peering',
          'Skip the creation of a peering network') {
    @conf[:do_configure_peering] = false
  }
  opts.on('-D', '--no-create-db',
          'Skip PostgreSQL DB initialization') {
    @conf[:do_create_database] = false
  }
  opts.on('-H', '--no-install-hkp',
          'Skip downloading and installing Hockeypuck') {
    @conf[:do_install_hockeypuck] = false
  }
  opts.on('-I', '--no-initial',
          'Skip initial container creation') {
    @conf[:do_initial_setup] = false
  }
  opts.on('-M', '--no-modified-prot',
          'Install Hockeypuck without the protocol modification') {
    @conf[:modified_hkp] = false
  }
  opts.on('-P', '--no-create-pgp',
          'Skip the creation of OpenPGP keys') {
    @conf[:do_create_pgp_keys] = false
  }
  opts.on('-S', '--no-sign-keys',
          'Skip the signature of OpenPGP keys') {
    @conf[:do_sign_pgp_keys] = false
  }
  opts.on('-X', '--no-poison-pgp',
          'Skip the poisoning of OpenPGP keys') {
    @conf[:do_pgp_poison] = false
  }
end.parse!


def build_guest(guest_num)
  initial_setup(guest_num) if @conf[:do_initial_setup]
  create_database(guest_num) if @conf[:do_create_database]
  install_hockeypuck(guest_num) if @conf[:do_install_hockeypuck]
  configure_peering(guest_num) if @conf[:do_configure_peering]
end

def initial_setup(guest_num)
  guest = guest_name_for(guest_num)
  # Create the LXC base install
  @log.info("Installing base system packages for #{guest}")
  cmd = %W(lxc-create -t debian -n #{guest} -- -r bullseye
           --packages=#{@conf[:install_pkgs].join(',')} )
  cmd.unshift('eatmydata') if @conf[:use_eatmydata]
  system(*cmd)

  # For reliability and predictable network addressing: set up the
  # network configuration in the lxc configuration, comment it from
  # the system
  @log.info("Configuring network for #{guest}")
  File.open( File.join(@conf[:guest_dir] % guest, 'config'),
             'a' ) do |conf|
    conf.puts "lxc.net.0.ipv4.address = #{ip_and_netmask_for(guest_num)}"
    conf.puts "lxc.net.0.ipv4.gateway = #{@conf[:ip_gateway]}"
  end
  net_conf = File.open(filename_in_guest(guest, '/etc/network/interfaces'), 'r'
                       ).readlines.map {|lin| lin="# #{lin}" if lin =~ /eth0/}
  File.open(filename_in_guest(guest, '/etc/network/interfaces'), 'w') do |net|
    net.puts(net_conf)
  end
```

```
  # Copy the "glue script" to interface between Hockeypuck and Sequoia
  File.copy_stream('filter_certs',
                   filename_in_guest(guest, '/usr/local/bin/filter_certs')
                  )
  File.chmod(0755, filename_in_guest(guest, '/usr/local/bin/filter_certs'))

  @log.info("Starting up container #{guest}")
  # Wait for things to settle (is eatmydata a good idea or not?) and
  # start the guest system up
  sleep(2)
  system('lxc-start', '-n', guest)
end

def attest_ck_setup()
  guest = 'attest';
  @log.info('Building attestation checker')
  cmd = %W(lxc-create -t debian -n #{guest} -- -r bookworm
           --packages=#{@conf[:install_pkgs_attest].join(',')})
  cmd.unshift('eatmydata') if @conf[:use_eatmydata]
  system(*cmd)

  # 'sq' is compiled from the modified version of Sequoia's repository
  File.copy_stream('sq', filename_in_guest(guest, '/usr/local/bin/sq'))
  File.chmod(0755, filename_in_guest(guest, '/usr/local/bin/sq'))

  @log.info("Configuring attestation checker's network")
  File.open( File.join(@conf[:guest_dir] % guest, 'config'),
             'a' ) do |conf|
    # 240 → 10.0.3.250
    conf.puts "lxc.net.0.ipv4.address = #{ip_and_netmask_for(240)}"
    conf.puts "lxc.net.0.ipv4.gateway = #{@conf[:ip_gateway]}"
  end
  net_conf = File.open(filename_in_guest(guest, '/etc/network/interfaces'), 'r'
                       ).readlines.map {|lin| lin="# #{lin}" if lin =~ /eth0/}
  File.open(filename_in_guest(guest, '/etc/network/interfaces'), 'w') do |net|
    net.puts(net_conf)
  end

  # "Glue script" to check with Sequoia the keys sent by the Sequoia servers
  File.copy_stream('tcp_to_sq.pl',
                   filename_in_guest(guest, '/usr/local/bin/tcp_to_sq'))
  File.chmod(0755, filename_in_guest(guest, '/usr/local/bin/tcp_to_sq'))
  File.copy_stream('tcp_to_sq.service',
                   filename_in_guest(guest,
                        '/etc/systemd/system/tcp_to_sq.service'))

  @log.info("Starting up attestation container")
  # Start the guest system up
  system('lxc-start', '-n', guest)

  adduser = guest_running(guest,
                          %w(adduser --disabled-password --disabled-login
                             --gecos SequoiaHelper sq))
  Process.wait(adduser.pid)
  guest_running(guest, %w(systemctl enable tcp_to_sq))
  guest_running(guest, %w(systemctl start tcp_to_sq))
end

def create_database(guest_num)
  guest = guest_name_for(guest_num)
  @log.info("Creating database in #{guest}")
  # Wait until PostgreSQL is running, and create a HKP user and
  # database. Have to lxc-attach as "real" files in /run are not
  # exposed through the cgroups
  wait_for_psql = guest_running(guest,
      ['perl', '-e', 'until (-e "/var/run/postgresql/.s.PGSQL.5432") {print ".";
           sleep 1}'])
  Process.wait(wait_for_psql.pid)
```

```
  wait_for_psql.close

  psql = guest_running(guest,
                       %w(sudo -u postgres psql template1))
  psql.puts 'CREATE USER hkp PASSWORD \'hkp-pcic-test\';'
  psql.puts 'CREATE DATABASE hkp OWNER hkp;'
  psql.close_write
  @log.debug '--- PostgreSQL results: ' + psql.read
  psql.close
end

def install_hockeypuck(guest_num)
  guest = guest_name_for(guest_num)
  @log.info("Installing Hockeypuck in #{guest}")

  # Copy and install the corresponding Hockeypuck package into the
  # container
  hkp_pkg = @conf[:modified_hkp] ?
              @conf[:hkp_pkg_modif] :
              @conf[:hkp_pkg_base]
  FileUtils.cp(hkp_pkg, filename_in_guest(guest, 'root'))
  dpkg = guest_running(guest,
                       [%W(dpkg -i /root/#{hkp_pkg})])
  Process.wait(dpkg.pid)
  @log.debug '--- dpkg results: ' + dpkg.read
  dpkg.close

  # Fix the configuration to use the database we just created
  fix_conf = guest_running(guest,
                           ['perl', '-p', '-i', '-e',
                            ('s/^# dsn/dsn/; s/h0ck3y/hkp-pcic-test/; ' +
                             's/INFO/DEBUG/'),
                            '/etc/hockeypuck-pcic/hockeypuck.conf'])
  Process.wait(fix_conf.pid)
  fix_conf.close

  # If our modifications are in place, the binaries have to be
  # rebuilt.
  if File.exists?(filename_in_guest(guest, '/usr/sbin/hockeypuck-rebuild'))
    rebuild = guest_running(guest, '/usr/sbin/hockeypuck-rebuild')
    Process.wait(rebuild.pid)
    rebuild.close
  end
end

def configure_peering(guest_num)
  guest = guest_name_for(guest_num)
  @log.info("Configuring peering between servers for #{guest}")
  conf = File.open(filename_in_guest(guest,
                   '/etc/hockeypuck-pcic/hockeypuck.conf'), 'a')

  @peers[guest_num].each_with_index do |peer, idx|
    # No vamos a hablar con nosotros mismos
    next if guest_num == idx
    # Si en el volado salió que con este no, pos no
    next unless peer

    peer_addr = ip_addr_for(idx)
    conf.puts ""
    conf.puts '[hockeypuck.conflux.recon.partner.%s]' %
              peer_addr.gsub(/\./,'-')
    conf.puts "httpAddr=\"#{peer_addr}:11371\""
    conf.puts "reconAddr=\"#{peer_addr}:11370\""
  end

  # Listos para iniciar el servicio de Hockeypuck! Monitoreamos hasta
  # que inicia correctamente (¡qué feo!)
  @log.info('About to start hockeypuck-pcic at %s' % guest)
```

```
    success = false
    times = 0
    while ! success
      sleep(1)
      times += 1
      start_hkp = guest_running(guest, ['systemctl', 'restart', 'hockeypuck-pcic'])
      pid, status = Process.wait2(start_hkp.pid)
      start_hkp.close

      success = true if status.exitstatus == 0
      @log.info('%s startup @%d: %d → %d (%s)' % [guest, times, pid, status,
                                                  (success ? '✔' : '✘')])
    end
  end

  def create_pgp_keys(guest_num)
    @log.debug '--- Generating %d OpenPGP keys for guest %d' %
               [@conf[:pgp_keys_per_host], guest_num]

    @conf[:pgp_keys_per_host].times do |i|
      person = Faker::Name.name()
      mail = Faker::Internet.email(name: person)
      userid = '%s <%s>' % [person, mail]
      c_time = Faker::Time.backward(days: 1095).strftime('%Y-%m-%dT%H:%M:%S')
      exp_time = Faker::Time.forward(days: 1095).strftime('%Y-%m-%dT%H:%M:%S')

      Dir.mkdir(@conf[:pgp_keys_dir]) unless Dir.exists?(@conf[:pgp_keys_dir])
      keyfile = File.join(@conf[:pgp_keys_dir],
                          '%02d_%04d.%s.pgp' % [guest_num, i,
                                                person.split(/\s/).join('')]
                         ).gsub(/\.+/, '.')
      certfile = keyfile + '.cert'

      # Generación de la llave usando Sequoia (desde el anfitrión)
      system('sq', 'key', 'generate',
             '--userid', userid,
             '--creation-time', c_time,
             '--expires', exp_time,
             '--export', keyfile)
      system('sq', 'key', 'extract-cert', keyfile, '-o', certfile)

      # Registramos la llave y archivo en el archivo de
      # control/bitácora, para su posterior consumo a la hora de firmar
      # (omitiendo el nombre del directorio
      File.write(@conf[:pgp_keydir_filename],
                 "#{File.basename(keyfile)}::#{userid}\n",
                 mode: 'a+')

      # Enviamos la llave por HKP al servidor
      res = hkp_send_key(ip_addr_for(guest_num), File.read(certfile))
      print ' [%d-%d %s]' % [guest_num, i,
                             (res.nil? ? '✔' : '✘ ')]
    end

    # Damos una "patada" a Hockeypuck para que inicie la sincronización
    Process.wait(guest_running(guest_name_for(guest_num),
                               ['systemctl', 'restart', 'hockeypuck-pcic']
                              ).pid)
  end

  def gen_pgp_signatures()
    keyfiles = {}
    tot_sign = {:signatures => 0, :attestations => 0}
    File.read(@conf[:pgp_keydir_filename]).split(/\n/).each do |lin|
      lin =~ /^(.+)::(.+)$/ or
        @log.warn "Unexpected line parsing #{@conf[:pgp_keydir_filename]}: #{lin}"
      keyfiles[$1] = $2
    end
```

```
  keyfiles.keys.each do |key1|
    keyfiles.keys.each do |key2|
      # Sólo firmar con probabilidad ≥ pgp_sign_prob
      next unless rand <= @conf[:pgp_sign_prob]
      if tot_sign[:signatures] % 50 == 0
        print "\n%04d " % tot_sign[:signatures]
      end
      tot_sign[:signatures] += 1
      print '👍'
      signed = '%s-signs-%s.cert' % [key1, key2]

      uid = keyfiles[key2]
      @log.debug 'New signature %s → %s (%s)' % [key1, key2, uid]
      system('sq', 'certify',
             pgp_file_for(key1),
             pgp_file_for('%s.cert' % key2),
             uid,
             '--output', pgp_file_for(signed))

      # Atestiguar la firma si la probabilidad así lo marca
      if rand <= @conf[:pgp_attested_sign_prob]
        @log.debug '%s attests %s→%s signature' % [key2, key1, key2]
        print '☑'
        tot_sign[:attestations] += 1
        # Atestiguar y guardar en el mismo archivo (signed)
        signed_full_key = '%s.pgp' % signed
        system('sq', 'keyring', 'merge',
               pgp_file_for(key2),
               pgp_file_for(signed),
               '--output', pgp_file_for(signed_full_key))
        system('sq', 'key', 'attest-certifications',
               pgp_file_for(signed_full_key),
               '-o',
               pgp_file_for('%s.att' % signed_full_key))
        # Reescribimos pgp_file_for(signed), hay que especificar --force
        system('sq', '--force', 'key', 'extract-cert',
               '--output', pgp_file_for(signed),
               pgp_file_for('%s.att' % signed_full_key))
      end

      to_server = ip_addr_for( (0..@conf[:num_hosts]-1).to_a.sample )
      @log.debug 'Sending %s→%s key to server %s' % [key1, key2, to_server]
      hkp_send_key(to_server, File.read(pgp_file_for(signed)))
    end
  end
  return tot_sign
end

def pgp_poisoning(num)
  # Creamos 1000 llaves para montar el ataque. Por lo visto, Hockeypuck
  # ya se protege contra  pgp-poisoner.
  attack_dir = 'attack_keys'
  attack_pids = []
  Dir.mkdir(attack_dir) unless Dir.exists?(attack_dir)
  @log.debug('Creating %d poisoning keys' % @conf[:pgp_attacker_keys])
  print "\n🏭"
  @conf[:pgp_attacker_keys].times do |i|
    print "🏭" if i % 50 == 0
    system('sq', '--force', 'key', 'generate',
           '--userid', 'Poisoning attacker key #%d' % i,
           '--export', File.join(attack_dir, i.to_s))
  end

  # Elegimos las víctimas aleatorias que nos indicaron para atacar
  File.read(@conf[:pgp_keydir_filename]).split(/\n/).sample(num).
    each_with_index do |v, idx|
    pid = fork()
```

```
      if ! pid.nil?
        # Proceso padre: toma nota del hijo y lanza el siguiente ataque
        attack_pids << pid
        next
      end

      (victim, uid) = v.split('::')
      victim_dir = File.join(attack_dir, victim)
      Dir.mkdir(victim_dir) unless Dir.exists?(victim_dir)
      print " #{idx}☣..."
      @log.debug('Attack %d: Poisoning key «%s»' % [idx, victim])
      # Generamos cada firma como un archivo independiente porque hkp
      # limita por tamaño máximo de POST
      @conf[:pgp_attacker_keys].times do |i|
        print '☠'  if i%50 == 0
        to_server = ip_addr_for( (0..@conf[:num_hosts]-1).to_a.sample )
        IO.popen(['sq', 'certify', File.join(attack_dir, i.to_s),
                  pgp_file_for(victim + '.cert'), uid]) do |sign|
          signed = sign.read
          hkp_send_key(to_server, signed)
        end
      end

      # El ataque se realiza desde un subproceso. Finalizamos el proceso
      # al terminar el ataque.
      exit(0)
    end

    @log.debug('%d attacks launched.' % num)
    attack_pids.each { |pid| Process.waitpid(pid) }
    @log.info('%d attacks finished.' % num)
  end

  def pgp_file_for(keyfile)
    return File.join(@conf[:pgp_keys_dir], keyfile)
  end

  def hkp_send_key(server, key_data)
    begin
      post_url = 'http://%s:11371/pks/add' % server
      res = Net::HTTP.post_form(URI(post_url), 'keytext' => key_data)
      return nil if res.code.to_i == 200
    rescue Errno::ECONNREFUSED
      @log.warn('Failed to send key to server %s: connection refused' % server)
    rescue
      @log.warn('Error sending key to server %s: %s (%s)' %
                [server, res.code.to_i, res.message])
    end
  end

  def make_peer_list()
    peers=[]
    num = @conf[:num_hosts]

    # Crear un arreglo anidado vacío (más "bonito" trabajar con arreglos
    # cuadrados)
    num.times do |i|
      peers[i]=[]
      num.times do |j|
        peers[i][j] = nil
      end
    end

    # Llenar con la probabilidad especificada
    num.times do |i|
      num.times do |j|
        next if i==j
        r=rand()
```

```
      if r > (1 - @conf[:peer_prob])
        peers[i][j] = 1
        peers[j][i] = 1
      end
    end
  end
  return peers
end

def peering_matrix()
  num=0
  peers = []
  peers << 'Peering matrix for the test network:'
  peers << '=================================='
  peers << ''
  peers << "   " + @conf[:num_hosts].times.map{|i| i.to_s}.join('')
  peers << @peers.map do |lin|
    h = "#{num} "
    num+=1
    h + lin.map{|host| host ? 'X' : ' '}.join('')
  end

  return peers.join("\n")
end

def guest_name_for(guest_num)
  return @conf[:guest_name] % guest_num
end

def filename_in_guest(guest, target)
  return File.join(@conf[:guest_dir] % guest, 'rootfs', target)
end

def ip_and_netmask_for(guest_num)
  return @conf[:ip_network] % (guest_num + 10)
end

def ip_addr_for(guest_num)
  return ip_and_netmask_for(guest_num).gsub(/\/\d+/, '')
end

def guest_running(guest, cmd)
  lxc_cmd = %W(lxc-attach -n #{guest} --)
  lxc_cmd << cmd
  lxc_cmd.flatten!
  return IO.popen(lxc_cmd, 'r+')
end

@log = Logger.new(@conf[:log_to])
@log.level = @conf[:log_prio]
@log.info('--- Starting lab instantiation')
started_at = Time.now

# Creamos la lista de cómo se van a comunicar entre sí los servidores,
# para implementarla durante la instalación en cada archivo de
# configuración.
@peers = make_peer_list()

# Vaciamos el archivo de directorio de llaves. Lo manejamos como
# archivo externo para comunicarlo entre procesos (y sí, fuchi, sin
# mutex para protegerlo... Crucemos los deditos)
File.write(@conf[:pgp_keydir_filename], '')

# Lanzamos un proceso independiente para construir a cada uno de los
# servidores de la red, y uno adicional para el verificador de
# testificación
builders = []
(@conf[:num_hosts]+1).times do |guest_num|
```

```
  @log.info("--> #{guest_num} / #{@conf[:num_hosts]}")
  pid = Process.fork
  if pid
    builders << pid
  else
    if guest_num < @conf[:num_hosts]
      build_guest(guest_num)
      @log.info '*** Guest number %d finished building (%4.2f seconds)' %
                [guest_num, (Time.now - started_at)]
    else
      attest_ck_setup if @conf[:do_install_attest_ck]
      @log.info '*** Attestation checker finished building (%4.2f seconds)' %
                [guest_num, (Time.now - started_at)]
    end
    exit(0)
  end
end

# Esperamos a que terminen de construirse todos los servidores
builders.each {|pid| Process.wait(pid)}
@log.info '*** Laboratory built! (%4.2f seconds)' % (Time.now - started_at)
@log.info '    %d containers created (%s -- %s)' %
          [ @conf[:num_hosts],
            guest_name_for(0),
            guest_name_for(@conf[:num_hosts]-1)]

# Ya que el servidor de BD está corriendo, iniciamos los monitores de
# crecimiento de número de llaves y el histograma total de llaves
NumKeysMonitor.new(@conf[:guest_name], @conf[:num_hosts])
KeySizeHistogram.new(@conf[:guest_name], @conf[:num_hosts])

# Registramos la matriz de conexiones en cada uno de los servidores
# (para referencia más fácil al depurar)
if @conf[:do_configure_peering]
  peers = peering_matrix()
  @conf[:num_hosts].times do |i|
    peer_file = filename_in_guest( guest_name_for(i), '/root/peers.txt' )
    File.open(peer_file, 'w') {|f| f.puts peers}
  end
  @log.debug "Peering information:\n#{peers}"
end

# Allow for servers to start up before sending requests
sleep(5)
if @conf[:do_create_pgp_keys]
  pgp_creators = []
  @conf[:num_hosts].times do |guest_num|
    pid = Process.fork
    if pid
      pgp_creators << pid
    else
      create_pgp_keys(guest_num)
      exit(0)
    end
  end
  pgp_creators.each{|pid| Process.wait(pid)}
end

# Creamos las firmas de forma aleatoria entre las llaves generadas,
# siguiendo los parámetros :pgp_* de @conf
if @conf[:do_sign_pgp_keys]
  if ! @conf[:do_create_pgp_keys]
    @log.warn('Not creating key signatures: keys not created, cannot' +
              'assume they exist.')
  end
  tot_sign = gen_pgp_signatures()
end
```

```
if @conf[:do_pgp_poison]
  @log.info('Poisoning and uploading %d generated keys' %
            @conf[:pgp_keys_to_poison])
  pgp_poisoning(@conf[:pgp_keys_to_poison])

  @log.info('All is done. Keep the program running to log synchronization.')
  print "\n\nTime elapsed since end of attack: "
  Thread.new do
    i=0
    print '    '
    while true
      i+=1
      print "\b\b\b\b\b%05d" % i
      sleep 1
    end
  end

  finished = false
  begin
    sleep(36000) # 10hr
    @log.info('Finished!.')
    finished = true
  ensure
    @log.info('Long sleep interrupted') unless finished
  end
end
exit 0
```