Welcome to the 2000s: Enabling casual two-party key signing

Tobias Mueller
tobiasmue@cryptobitch.de

1 Introduction

Email is one of the oldest and most popular applications on the Internet. The technology dates back to 1971 and was not designed with today's attackers in mind [31]. Over the last forty years, attackers have gained many capabilities, such as eavesdropping or manipulating data on the network.

The protocols involved in Email have not significantly changed since their inception. RFC 5322 [23] is probably the most prominent standard related to Email and its roots date back to RFC 733 from 1977 [24]. Despite encryption being deployed from client to server and from server to server, Emails are still vulnerable to attacks because client-to-client (or end-to-end) encryption has not been wildly adopted. Without client-to-client encryption, Emails can be read and modified by a malicious operator, putting their confidentiality and integrity at stake.

It was in the early nineties that PGP sought to bring a solution to that problem by implementing public key cryptography and a decentralised Public Key Infrastructure (PKI). However, the decentralised PKI, the Web of Trust (WoT), is depending on people participating in it by signing other peoples' keys. So far, the act of signing a key required multiple error-prone steps and external entities such as a key-server on the Internet.

The 2000s brought us WiFi, link-local networks, mobile handheld devices, and built-in cameras. However, we have yet to leverage these technologies in order to solve problems related to signing keys in the WoT.

This paper addresses some of those problems by implementing a new protocol which minimises the dependencies on external entities and automates as many steps as possible. To summarise, the contributions of this paper are:

- analysis of existing key signing protocols as well as tools to assist the key signing process with respect to the barriers they pose to wide adoption,

- a new protocol targeted at small groups of people or even individuals for signing their keys, and
- an implementation of the new protocol.

The remainder of this paper is structured as follows. Section 2 describes the technologies related to problem of key signing. Section 3 describes the problem users of the WoT face when trying to sign other peoples' keys. A new approach for signing a key is presented in section 4. Section 5 presents an implementation of the new protocol which removes dependencies on key-servers and manual fingerprint validation. Related work is discussed in section 6. This paper concludes in section 7 with a summary of the findings of this papers.

2 Background

This section describes key concepts related to OpenPGP, key signing, and to the solution proposed in the following section.

2.1 OpenPGP

The first software to make asymmetric key cryptography popular was PGP in 1991 [18]. In 1997, the OpenPGP alliance was formed "to define [a] standard that had formerly been a proprietary product since 1991. [...] OpenPGP has become the standard for nearly all of the world's encrypted Email" [21].That standard is nowadays known as RFC 4880 [27] and defines a message format for encrypted or signed messages, keys, and certificates. Because it only defines the format of the messages, it is agnostic of the transport protocol. However, Email can be assumed to be the most popular application making use of OpenPGP messages.

Besides the original PGP, a Free Software [29] implementation named GnuPG exists.

2.2 Web of Trust

As OpenPGP applications need to obtain key material, i.e. public keys, in order to encrypt messages, some form of a Public Key Infrastructure (PKI) is needed. Among the problems a PKI tries to solve is the distribution of the correct key for an entity you want to communicate with.

Keys in the OpenPGP PKI can be signed by anyone. It is thus referred to as being decentralised [1]. It was aptly named Web of Trust (WoT). The semantic of a signature is commonly understood to be an attestation of the "correctness" of the key. "Correct" usually means that the information that is on a key, such as the name and Email address, does indeed belong to the person who has access to the private key.

A revocation in the WoT is a special signature on a key. Popular OpenPGP implementations ask the user to create such a revocation signature when creating a new key and to store that signature in a secure place. This signature can embed more metadata indicating a reason for the revocation.

This PKI model is vastly different from the popular X.509 PKI [7] which is very centralised and suffers from a range of structural problems [6, 13]. The WoT can be seen as a superset of the X.509 PKI as the mesh-style trust paths in the WoT can model a centralised PKI but not vice versa. Although a standard exists that allows X.509 to be used in a mesh-style network [5], it only supports full meshes and not partial meshes as the WoT does.

2.3 Key-Server

The keys in the WoT are managed in a network of key-servers. A key-server, in the OpenPGP terminology, is a database with OpenPGP keys. The list of protocols an OpenPGP implementation needs to use in order to interact with a key-server is not well defined. In practise, Lightweight Directory Access Protocol (LDAP) and Hypertext Transfer Protocol (HTTP) are used and the vast majority of the deployed key-servers use HTTP Keyserver Protocol (HKP) [26] as their transport protocol[1]. This protocol is based on HTTP and does not provide any guarantees for the confidentiality or integrity of the transferred data.

A popular argument is that a key-server serves public keys which, as their name suggest, are public and thus the communication with a key-server does not need any protection. However, without confidentiality, every request to a key-server exposes who the user is attempting to communicate with. Similarly, without ensuring integrity, the user is left vulnerable to receiving a manipulated key. For example, an attacker may decide to strip off OpenPGP packets of revocation signatures. The user will thus use a potentially corrupted key which the attacker controls. Even worse, the version of GnuPG that the most recent Ubuntu 14.04 ships (1.4.16) is vulnerable to a trivial man in the middle attack. The user can specify the fingerprint when it asks GnuPG to retrieve that key from a key-server which is recommended by a rather popular list of "best practices" when using GnuPG[2]:

> Do not blindly trust keys from key servers.
>
> Anyone can upload keys to key servers and there is no reason that you should trust that any key you download actually belongs to the individual listed in the key. You should therefore verify with the individual owner the full key fingerprint of their key. You should do this verification in real life or over the phone. Once you have verified the key fingerprint that you need, you may download the key from the key server pool:
>
> ```
> gpg --recv-key fingerprint
> ```

However, GnuPG accepts any key the server returns, even if its fingerprint does not match the one the user specified. In order to solve these problems, Transport Layer Security (TLS)

[1]cf. `https://sks-keyservers.net/status/`

[2]`https://help.riseup.net/en/security/message-security/openpgp/best-practices`

support was added to GnuPG in 2009 [15]. This, ironically enough, uses HTTPS which uses X.509 certificates. Unfortunately, only one third of the available key-servers support TLS[3].

Besides these implementation problems of OpenPGP clients, key-servers also suffer from several structural problems [12]. Given the problems associated with key-servers and the poor adoption of potential solutions, avoiding key-servers is a desirable goal.

3 Keysigning

This section identifies and describes common requirements of protocols for signing keys in the WoT. A protocol will be described and it will be shown how it addresses the identified requirements.

Extending the description already given in section 2.2, a popular Web site with instructions to hold a key signing party describes the act of key signing as “digitally signing a public key packet and a user id packet that is attached to the key in that public key packet. Key signing is done to verify that a given user id and public key really do belong to the entity that appears to own the key. In more basic terms it is done to verify that the representation of identity in the user id packet is valid. Usually, this means that the name on the PGP key matches the name on the identification the person is presenting to you when asking that you sign their key” [2].

3.1 Requirements

When looking at some protocol descriptions for signing OpenPGP keys [17, 2, 33, 3] or for holding “key signing parties”[14, 11, 8, 19], the following common requirements can be identified.

Validate UIDs. With OpenPGP, a key can have multiple User IDs (UIDs). Technically, a UID is an arbitrary string, but OpenPGP clients ask the user for a name and an Email address when creating a key.

It is commonly understood that a signature on a UID signifies the correctness of the information shown. Hence, it is expected that, before creating a signature, the information is validated. Usually, the name and the Email address are checked. The name is either known or some form of ID is used as a proof. The Email address can be validated by sending an encrypted message under the public key of the UID to the address in question. The Email can only be read if the person does indeed have access to the private key and thus rightfully has the Email address in a UID of that key.

Obtain authentic key. In order to produce a signature on a key, the key itself needs to be retrieved. In the ideal case, the signee provides a file, held in a trusted environment, of which both parties assert that it contains the relevant key. Other options range from using a pendrive to transfer the key, finding it stored on a networked file system, or

[3]as of 2014-08-20, cf. `https://sks-keyservers.net/status/`

asking a key-server. The ideal case is commonly approximated by securely obtaining the fingerprint of the key, e.g. via the audible channel at an in-person meeting with the proclaimed key owner. This fingerprint is then used to retrieve a key from an untrusted medium. Note that the fingerprint does not represent the ideal case because of the attacks mentioned in section 2.3. Also note that other key signing protocols attempt to solve this problem by providing a collection of the keys of all participants of a "key signing party". Such a protocol, however, incurs setup costs which makes it unattractive for small groups of people to sign their keys.

Publish signature. After having signed a key (or rather: UIDs on a key), the signature needs to be published in order to strengthen the WoT. It is also possible to not publish the signature or to produce a 'local' signature which the OpenPGP client will not export. It is good practise to send the signature, and the signature only, to the address mentioned in the signed UID.

With these requirements the owner of the key to be signed needs to be present and ideally in-person as it is hardly possible for someone else to convincingly attest the UIDs or to provide an authentic key.

3.2 Contemporary protocol for key signing

When two or more people meet, they might want to sign each others' keys. Various documents on the Web describe how to do this best [17, 2, 33, 3]. All assume that the people do not know each other and thus require a proof of the person's identity.

All protocols have the following steps in common: When you meet someone you want to sign keys with, you obtain a hard copy of the fingerprint and the UIDs of the key to be signed. You verify the names in the UID, e.g. with the help of documents you trust. You create a signature for the UIDs you are convinced carry the correct name and send the signature in an encrypted Email to the addresses of the UIDs.

These protocols fulfil the requirements mentioned in section 3.1: **Validate:** The person convinced you that the names on the UIDs are indeed correct, e.g. because they showed documents you trust. The Email addresses are verified by sending the produced signature as an encrypted Email message to the address in question. Only if the person does indeed have access to the private key, the signature can be decrypted and published. **Obtain:** The person convinced you that the fingerprint does indeed belong to the public key they claim to have the private part of, e.g. with a piece of paper with the printed fingerprint. **Publish:** The signee receives the signature in an encrypted message, thereby proving to having access to the private key. The signee may then opt for sending the signature to the key-server network.

Typically, the best practises require you to "carry slips of paper or maybe business cards with your name (as it appears on your photo identification and OpenPGP key), along with the associated key fingerprint" [20], because it "is to check that the copy of the key you have obtained either from a keyserver or Email is actually the key this person uses and you haven't somehow gotten a forged key."[3].

The best practises, however, do not consider an attacker who is able to modify a key in transit, e.g. strip packets revocation packets off the key. As it was already discussed earlier (i.e. in section 2.3), that attack is trivially implementable. The presented protocols do thus not sufficiently address the problem of obtaining an authentic copy of the key to be signed.

4 Casual Key Signing Protocol

As the presented protocols do not address the problem of a small group obtaining authentic copies of their keys to be signed, a new protocol is now presented.

The already presented protocols try to make a big key signing party (KSP) more efficient. The cost of the initial setup are prohibitive for small groups or individuals to sign their keys. These costs are associated with the secure transfer of the key and the validation of the fingerprint of the key holder.

In order to allow casual key signing, even between only two individuals meeting each other, the process of signing the other person's key must be as automated as possible and have as little requirements as possible. A dependency on key-servers must be avoided as well as the need of exchanging physical material such as a business card with a fingerprint printed on it.

The new protocol will rely on already established trust as well as the availability of a secure channel for exchanging a few bytes of secret material.

The secret material is the confirmation of the authenticity of the public key. In the protocols described above, this is obtained by confirmation of the person claiming to hold the key. This confirmation is usually written down on a sheet of paper and indicated as a simple check-mark. Advanced users use a pen to produce a signature indicating that they did indeed see that other person and that they did confirmed the correctness of the fingerprint of the public key. This piece of paper is usually carried home and the fingerprints are then transcribed into the computer.

These error-prone steps can be avoided if the authenticity of a key can be asserted in an automated manner, right at the venue were people intend to sign their keys. If two machines are able to connect to each other, e.g. being in the same WiFi network with the ability to open a TCP connection, it is possible to transfer the key. However, an attacker might still be able to manipulate data if no further attention is paid. As such, a secure channel is desirable for the transfer of the key. In order to match the security of today's key signing protocols, only the fingerprint needs to be transferred securely. It can be assumed that, when people meet to sign their keys, they can establish a secure physical channel either by talking to each other or by showing something to the other person. This fact can be leveraged to either transfer the fingerprint via voice or a visual representation on screen. The signing party can either type the fingerprint into their machine or, if a camera is built-in or otherwise available, scan the visual representation. For automation purposes, the visual channel is to be preferred as it is much easier to generate a machine readable visual representation of data than to generate machine readable audible data.

However, the secure channel is of low bandwidth as hearing and typing are manual tasks which do not scale well and the amount of data that can be transferred visually, in a machine readable manner, is limited. In order to transmit data visually QR codes [32], another technology of the 2000s, can be used. The amount of data a QR code can embed is limited to around 25kB [25]. While that is enough to fit the key material of an OpenPGP key, is it not enough to reliably transfer the whole key with UIDs, signatures, and other relevant data. However, it is enough to transfer the fingerprint or a key to a Hash-based Message Authentication Code (HMAC) [16].

The protocol is described as follows:

- A: Announce availability of a key on the network
- A: Generate a machine readable code of a Message Authentication Code (MAC) to protect the integrity of the key
- B: Scan the code and obtain the MAC
- B: Discover the key available on the network and download it
- B: Check integrity of the key using the MAC obtained
- B: Sign the UIDs of the keys
- B: Send the signature as encrypted Email to the addresses to the respective UID
- A: Decrypt messages received and publish signature on key-servers or within the local network

This scheme is secure if the people intending to sign their keys can assert that neither the audible nor the visual can be corrupted. An attacker, in order to successfully corrupt either channel, needs to influence both parties such that they are neither seeing what is displayed on the screen nor hearing what is said by the other person.

The protocol fulfils the requirements identified in section 3.1:

Validate: Regular verification can be done, e.g. with trusted documents. If, however, the people signing their keys have already established a trust relationship, verification of the names on the UIDs does not need to be as strictly handled as with strangers. The Email addresses encoded in UIDs are validated using the existing best practise of sending the signature in an encrypted Email.

Obtain: Only the correct key will be processed because the key obtained is checked with the help of the MAC which has been transferred securely. Note that the desired property of the secure data channel is integrity, not confidentiality. The visual channel is assumed to be secure. Note that an attacker might control the generation of the bar-code and thus be able to manipulate the key in transit. However, such an attacker is likely able to control much more than the bar-code, e.g. the OpenPGP installation. An attacker controlling OpenPGP installation is much more severe and will not be protected against by any key signing protocol.

Publish: Following best practises, the person whose key is being signed is able to publish the signature once they are able to decrypt Email messages.

This protocol enables people to sign their keys in an ad-hoc fashion leveraging technologies we have had for more than a decade. The key idea is to establish a secure channel for transferring the key to be signed. It obsoletes fingerprints printed on paper slips and vulnerable communication with a key-server. It also helps to make signing key a more instantaneous process which might encourage people to participate in the WoT more.

Contrasting already existing protocols, the new approach accepts the fact that users have arrived in the 2000s and carry at least one portable computing device with a camera, instead of urging users to not bring computers to KSPs.

5 Implementation

This section describes an implementation of a novel approach for signing keys in the WoT.

The protocol mentioned in section 4 was implemented as Free Software using Python, GNOME libraries, and GTK+ for a GUI. Currently, the program can perform two tasks. One is the server side which publishes the availability of the key on the network, generates a machine readable code, and makes the key available for download. The other is the client side which scans a bar-code, obtains keys from the local network, signs keys, and emails the encrypted signatures.

After having selected a key, the server encodes the key's fingerprint as a QR code and displays it on the screen. An HTTP server is then spawned and the availability of the key is announced using Avahi[4]. The HTTP server is implemented using Python's extensive standard library.

When the client is started, it asks the user to either type in a fingerprint or to scan a QR code. Both channels, i.e. the manually typed fingerprint or the visual channel for scanning the bar-code, are assumed to be secure. The QR code scanner was implemented using GStreamer's ZBar bindings[5]. This was manually wrapped into a GTK+ widget to be shown in a GUI.

After having either typed in a fingerprint or scanned a bar-code, the client iterates over the available key signing services on the local network and tries to download the key. When a key was successfully downloaded, its authenticity is checked with the fingerprint obtained. Note that several ways exist to achieve the desired integrity property. The fingerprint was chosen because it matches current key signing protocols. In the future, an HMAC will be chosen to obtain integrity over the full data transferred.

Once the key has been obtained, its UIDs are signed. Currently, the GnuPG wrapper created by the Monkeysign project (cf. section 6.4) is used. The signatures are exported as files and Emails are sent to the addresses in the UIDs.

Several ways to send Emails exist. The traditional UNIX is to rely on `sendmail` to work. However, with more and more deployments on non-server machines in the 2000s, `sendmail` is not necessarily installed let alone configured. As a consequence, opening a Simple Mail

[4]`http://avahi.org/`

[5]`http://gstreamer.freedesktop.org/data/doc/gstreamer/head/gst-plugins-bad-plugins/html/gst-plugins-bad-plugins-zbar.html`

Transfer Protocol (SMTP) connection with a mail server is an option. That would, however, require the user to configure the Email settings at least twice: Once for the regular Email client and once for the key signing solution. In order to mitigate that, one could obtain the SMTP configuration from the Email client used. However, no unified way for Email clients to store their configuration exists, i.e. one would need to find out what Email client the user uses and have specific routines for obtaining the settings. However, in order to be extensible, the hard dependency on SMTP was not considered as being useful for this implementation. The current implementation lets the desktop used open the user's Email client and preload it with all information necessary, i.e. the recipient's address, the encrypted signature, and an introductory text. That was done using `xdg-email`[6]. Letting the user's Email client pop up also has the advantage that the user confirms the act of sending a signed key to the key holder.

6 Related Work

This section describes other work which attempts to make the process of signing other peoples' keys in the WoT easier. While approaches exist to remodel the existing WoT (e.g. [10] or [9]), the approach followed in this paper does not alter the existing WoT in any way. This section does thus only discuss tools related to the currently deployed and WoT. Note that this also excludes other (proprietary) PKIs such as those from contemporary mobile messaging applications.

6.1 CABot

The CABot "is a set of scripts that help[s] managing some parts of a PGP keysigning process. It sends encrypted challenges to UIDs of an OpenPGP key, analyses the replies and assists the key owner in signing them" [22]. It requires the user to modify the server receiving their Emails. This is a demanding requirement which many users are unlikely to be able to do.

6.2 CAFF

CAFF is "a Perl script that helps you to automate the whole bunch of manual steps you have to do for every single key you want to sign after a key signing party" [28].

It is designed to help the traditional protocols for signing keys. As such, it expects fingerprints of the keys to sign. Hence, some form of a key signing party needs to be conducted in order to obtain the fingerprint of the OpenPGP key a person claims to have the private key to and to verify the person's identity matching the UIDs of the key. Afterwards, every fingerprint the user wants to sign is given to CAFF. Once the user asserts that they do indeed want to sign the UIDs on the key CAFF tries to find the key in the users' keyring or looks it up on a key-server (cf. 2.3). CAFF then signs the key it found, creates an Email message explaining what this is, attaches the freshly created signature, encrypts the Email message under the key in question,

[6] `http://portland.freedesktop.org/xdg-utils-1.0/xdg-email.html`

and sends the Email to the address of the just signed UID. Note that CAFF does not add the signed key to the user’s normal OpenPGP keyring. Instead, if the key holder does indeed have both control over the Email address in question and access to the private key, the key holder will receive the message, decrypt it, and then import the signature into their own keyring. The key holder may then publish the new signature to the public key-server network.

While CAFF makes it much easier to sign other people’s key properly, it is designed for helping to sign a large amount of keys in a traditional key signing party. It does not address the weaknesses of the general key signing protocols regarding individual key signing. In particular, it does not address the problem of obtaining an authentic key. It also requires the user to have a Mail Transfer Agent (MTA) running, which many users are not able to do (cf. section 5).

6.3 PIUS

“PGP Individual UID Signer (PIUS) helps attendees of PGP keysigning parties. It is the main utility and allows you to quickly and easily sign each UID on a set of PGP keys. It is designed to take the pain out of the sign-all-the-keys part of PGP Keysigning Party while adding security to the process”[4].

This tool is much like CAFF and can be considered a reimplementation in Python. It also does not address the issue of obtaining an authenticated key and relies on the user providing the keys to sign in a keyring.

6.4 Monkeysign

Monkeysign “is a tool to overhaul the OpenPGP keysigning experience and bring it closer to something that most primates can understand”[30]. It is based on the ideas of CAFF and extends those by enabling users to print out and scan two dimensional bar-codes of their keys. This helps to avoid “reciting tedious strings of hexadecimal characters” [30]. Monkeysign also eases sending the encrypted Emails with the signed key as it can talk directly to an SMTP server without the need of a local MTA.

The work presented in this paper is inspired by the Monkeysign approach and extends it by allowing the secure transfer of the key. Contrasting Monkeysign, the presented solution does not need to connect to a key-server and does not need the user to print, remember, and bring a piece of paper home. Also, sending Emails has further been eased as the user sees their regular Email client preloaded with a ready-to-send Email.

7 Conclusion

This section summarises the findings of this paper and gives an outlook for future work.

Section 2 showed that key-servers are an essential part of the OpenPGP PKI. However, communication with key-servers can not satisfyingly done securely, because either the client does not perform necessary plausibility checks of the received data or because the server does not support a secure transport protocol. The adoption of more secure key-server transport protocols has been low and trivial attacks on the communication with key-servers with severe impact are still possible. Reducing the need of having to communicate with key-servers is desirable.

Section 3 identified common requirements of existing key signing protocols. Those protocols aim at making very well attended key-signing events more convenient. Those events come with high setup costs. These costs are associated with the requirements a secure signing protocol has and are prohibitive for small and very small groups to hold such a key-signing party. One of the identified requirements is being able to obtain an authentic copy of the public key before signing it. This requirement is commonly addressed by providing a fingerprint of the key and by relying on key-servers to return the appropriate key. However, as was discussed earlier, decreasing the need of key-servers is worthwhile.

A new protocol was introduced in section 4. That protocol is mainly designed for people who have already established a trust relationship, but is also suited for the general case. It tried to make they process of signing a key as simple as possible by making several assumptions. The presented protocol solves the problem of obtaining an authenticated copy of a person's key by establishing a secure channel to exchange the key to be signed or a MAC thereof.

An implementation of the protocol was described in section 5. A laptop's webcam and a QR code are used to establish a secure channel for exchanging a key's MAC. Alternatively, the audible channel can be used to read out and to transcribe the MAC.

As presented in section 6, existing solutions do not sufficiently solve the problem of signing a person's key. As they follow traditional key signing protocols, the problem of obtaining an authenticated copy is left to the user. Other problems with existing tools include the necessity to provide configuration for sending emails or no appealing user interface.

The presented solution assumes that users have a computing device and a network connection at their disposal. It is also assumed that the private key is available and able to produce signatures for other keys. As it is possible to create sub-keys which are not able to produce signatures for other keys, this will pose a problem with users who do not carry their main key on their portable computer.

The optional availability of a camera greatly helps to reduce the complexity when establishing an authenticated channel. Currently, it is only the fingerprint that is encoded as a QR code to authenticate the key sent over the insecure channel. While that matches the security of current key signing protocols, it does not protect against certain attacks. As discussed in section 2.3, the fingerprint does not represent a secure MAC for an entire OpenPGP key. The fingerprint is generated by calculating a SHA1 hash over the creation time and other cryptographically used material of the key [27, p. 12.2.].Other information, such as UIDs or revocation certificates, do not alter the fingerprint. Hence, an attacker can manipulate the key in transit and without changing the fingerprint. However, this matches the security level of the current best practises of key-signing parties. Unfortunately, it is not always possible to encode the whole key and it's UIDs, because the QR code only allows around 25kB (cf. section 4). It is, however, possible to attach images (in fact, arbitrary data) to a key so that it is easy for a key to be larger than 25kB. As the solution presented in this paper leverages a secure (but low bandwidth) channel, it is certainly possible to exchange a proper cryptographic key which will be used to establish a high bandwidth secure channel. This, and other improvements, are planned and the development can be followed on the Web[7].

Currently, the presented implementation works on a local network, assuming that people can establish a secure channel when meeting in person. It might, however, be possible to use the presented scheme on the Internet, instead of on a local network. However, due the sheer size of the Internet and the scarcity of IPv4 addresses many people are in NATted networks which do not automatically allow a direct connection coming in from the Internet. It is also questionable whether the required secure channel can be established.

The presented solution was designed with Ellison's Law[8] in mind, which states that "the user-base for strong cryptography declines by half with every additional keystroke or mouseclick required to make it work". It aims to be an easy enough tool for people to use in order to increase participation in the WoT and strengthen it.

References

[1] Alfarez Abdul-Rahman and Stephen Hailes. 'A Distributed Trust Model'. In: *Proceedings of the 1997 Workshop on New Security Paradigms*. NSPW '97. New York, NY, USA: ACM, 1997, pp. 48–60. ISBN: 0-89791-986-6. DOI: 10.1145/283699.283739. URL: http://doi.acm.org/10.1145/283699.283739 (visited on 19/08/2014).

[2] V. Alex Brennen. *The Keysigning Party HOWTO*. 24th Jan. 2008. URL: http://www.cryptnet.net/fdp/crypto/keysigning_party/en/keysigning_party.html (visited on 20/08/2014).

[3] Phil Dibowitz. *PGP Key Signing*. 25th Apr. 2007. URL: http://www.phildev.net/pgp/gpgsigning.html (visited on 21/08/2014).

[4] Phil Dibowitz. *Pius*. 2011. URL: http://www.phildev.net/pius/ (visited on 28/08/2014).

[5] Yuriy Dzambasow et al. *Internet X.509 Public Key Infrastructure: Certification Path Building*. Sept. 2005. URL: http://tools.ietf.org/html/rfc4158 (visited on 19/08/2014).

[6] Carl Ellison and Bruce Schneier. 'Ten Risks of PKI: What You're not Being Told about Public Key Infrastructure'. In: *Computer Security Journal* 16.1 (2000), pp. 1–7. URL: https://www.schneier.com/paper-pki-ft.txt (visited on 19/08/2014).

[7] Stephen Farrell and Carlisle Adams. *Internet X.509 Public Key Infrastructure Certificate Management Protocols*. Mar. 1999. URL: http://tools.ietf.org/html/rfc2510 (visited on 19/08/2014).

[8] FOSDEM. *Keysigning*. Jan. 2014. URL: https://archive.fosdem.org/2014/keysigning/ (visited on 25/08/2014).

[7] https://www.github.com/muelli/geysigning

[8] http://web.archive.org/web/20070307134136/http://www.cultdeadcow.com/panel2001/hacktivism_panel.php

[9] Guibing Guo, Jie Zhang and Julita Vassileva. ‘Improving PGP Web of Trust Through the Expansion of Trusted Neighborhood’. In: *Proceedings of the 2011 IEEE/WIC/ACM International Conferences on Web Intelligence and Intelligent Agent Technology - Volume 01*. WI-IAT ’11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 489–494. ISBN: 978-0-7695-4513-4. DOI: `10.1109/WI-IAT.2011.134`. URL: `http://dx.doi.org/10.1109/WI-IAT.2011.134` (visited on 19/08/2014).

[10] Rolf Haenni and Jacek Jonczy. ‘A New Approach to PGP’s Web of Trust’. In: *23rd Chaos Communication Congress*. 23C3. Berlin, Germany, 30th Dec. 2006, pp. 61–66. DOI: `10.1.1.106.7979`. URL: `http://events.ccc.de/congress/2006/Fahrplan/events/1607.en.html`.

[11] Kevin Herron. *Keysigning Party Guide*. Jan. 2011. URL: `http://bit.ly/14RXVcU` (visited on 25/08/2014).

[12] Tillman Holst. ‘Possible Threats to PGP Key Servers’. Bachelorthesis. Hamburg: Hamburg University, 27th Aug. 2006.

[13] Ralph Holz et al. ‘The SSL Landscape: A Thorough Analysis of the x.509 PKI Using Active and Passive Measurements’. In: *Proceedings of the 2011 ACM SIGCOMM Conference on Internet Measurement Conference*. IMC ’11. New York, NY, USA: ACM, 2011, pp. 427–444. ISBN: 978-1-4503-1013-0. DOI: `10.1145/2068816.2068856`. URL: `http://doi.acm.org/10.1145/2068816.2068856` (visited on 19/08/2014).

[14] *Key signing party*. In: *Wikipedia, the free encyclopedia*. Page Version ID: 606399491. 2nd Aug. 2014. URL: `http://en.wikipedia.org/w/index.php?title=Key_signing_party&oldid=606399491` (visited on 25/08/2014).

[15] Werner Koch. *GnuPG 1.4.10 released*. E-mail. 2nd Sept. 2009. URL: `http://lists.gnupg.org/pipermail/gnupg-announce/2009q3/000291.html` (visited on 19/08/2014).

[16] Hugo Krawczyk, Ran Canetti and Mihir Bellare. *HMAC: Keyed-Hashing for Message Authentication*. Feb. 1997. URL: `http://tools.ietf.org/html/rfc2104` (visited on 20/08/2014).

[17] Daniel Manrique. *KeySigningParty - Ubuntu Wiki*. 7th Nov. 2011. URL: `https://wiki.ubuntu.com/KeySigningParty` (visited on 25/08/2014).

[18] Richard A Mollin. *An introduction to cryptography*. Boca Raton: Chapman & Hall/CRC, 2007. ISBN: 9781420011241 1420011243 1584886188 9781584886181.

[19] David Nalley. *PgpKeySigning*. 31st Mar. 2014. URL: `http://wiki.apache.org/apachecon/PgpKeySigning` (visited on 25/08/2014).

[20] Ben Nemec. *OpenPGP Web of Trust - OpenStack*. 13th Oct. 2013. URL: `https://wiki.openstack.org/wiki/OpenPGP_Web_of_Trust` (visited on 21/08/2014).

[21] OpenPGP Alliance. *About OpenPGP*. URL: `http://openpgp.org/about_openpgp/` (visited on 20/08/2014).

[22] Peter Palfrader. *CA-Bot*. 27th Oct. 2007. URL: `http://cabot.alioth.debian.org/` (visited on 28/08/2014).

[23] Resnick Peter W. *Internet Message Format*. Oct. 2008. URL: `http://tools.ietf.org/html/rfc5322` (visited on 19/08/2014).

[24] K. T. Pogran et al. *Standard for the format of ARPA network text messages*. 21st Nov. 1977. URL: http://tools.ietf.org/html/rfc733 (visited on 19/08/2014).

[25] *QR code*. In: *Wikipedia, the free encyclopedia*. Page Version ID: 621994275. 20th Aug. 2014. URL: http://en.wikipedia.org/w/index.php?title=QR_code&oldid=621994275 (visited on 20/08/2014).

[26] David Shaw. *The OpenPGP HTTP Keyserver Protocol (HKP)*. Mar. 2003. URL: http://tools.ietf.org/html/draft-shaw-openpgp-hkp-00 (visited on 19/08/2014).

[27] David Shaw et al. *OpenPGP Message Format*. Nov. 2007. URL: http://tools.ietf.org/html/rfc4880 (visited on 19/08/2014).

[28] Jorge Soares. *caff - Debian Wiki*. 18th Feb. 2014. URL: https://wiki.debian.org/caff (visited on 28/08/2014).

[29] Richard Matthew Stallman. *What is free software?* 5th Aug. 2014. URL: http://www.gnu.org/philosophy/free-sw.html (visited on 20/08/2014).

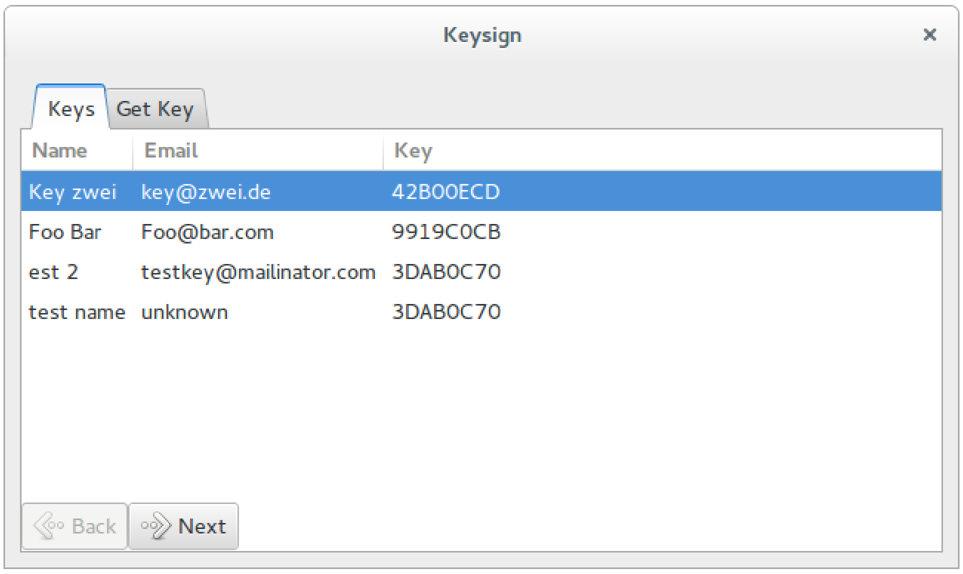
[30] *The Monkeysphere Project*. 16th Dec. 2013. URL: http://web.monkeysphere.info/ (visited on 27/08/2014).

[31] Ray Tomlinson. *The First Email*. The First Network Email. URL: http://openmap.bbn.com/~tomlinso/ray/firstemailframe.html (visited on 18/08/2014).

[32] ‘United States Patent: 5726435 - Optically readable two-dimensional code and method and apparatus using the same’. 5726435. Masahiro Hara et al. 10th Mar. 1998. URL: http://patft.uspto.gov/netacgi/nph-Parser?Sect2=PTO1&Sect2=HITOFF&p=1&u=/netahtml/PTO/search-bool.html&r=1&f=G&l=50&d=PALL&RefSrch=yes&Query=PN/5726435 (visited on 20/08/2014).

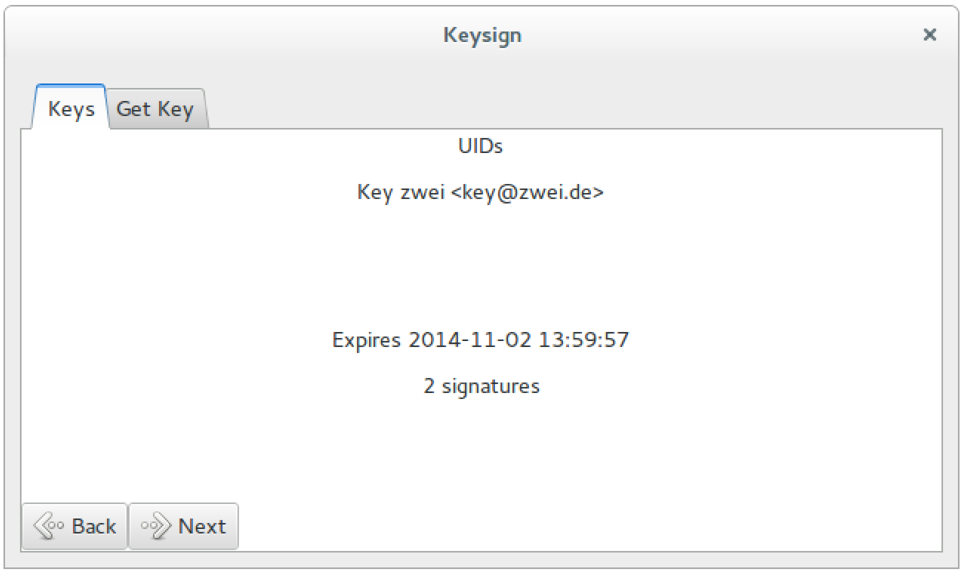
[33] Paul Wise. *Keysigning*. 10th May 2014. URL: https://www.debian.org/events/keysigning (visited on 25/08/2014).

A Appendix



Name	Email	Key
Key zwei	key@zwei.de	42B00ECD
Foo Bar	Foo@bar.com	9919C0CB
est 2	testkey@mailinator.com	3DAB0C70
test name	unknown	3DAB0C70

Figure 1: Select a key to publish on the network



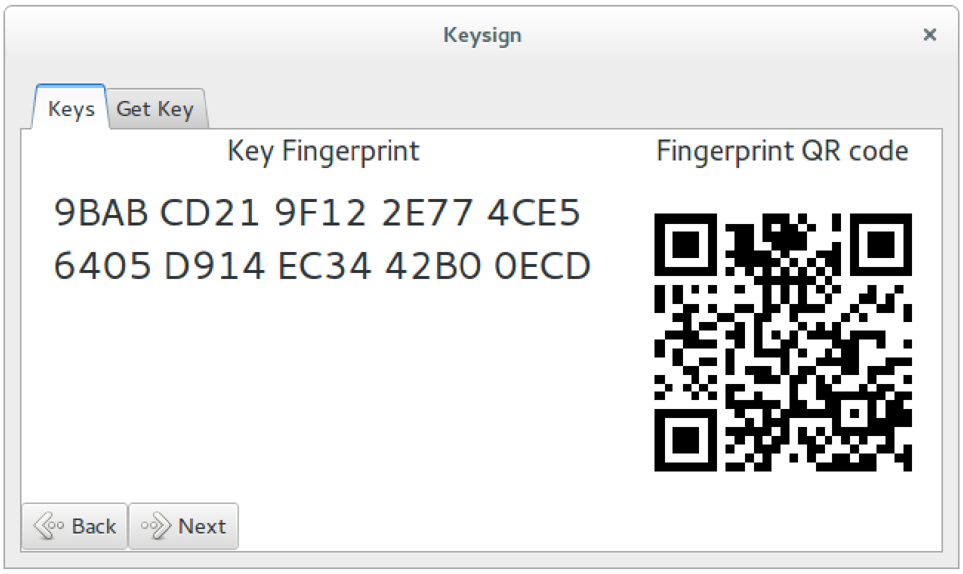
UIDs

Key zwei <key@zwei.de>

Expires 2014-11-02 13:59:57

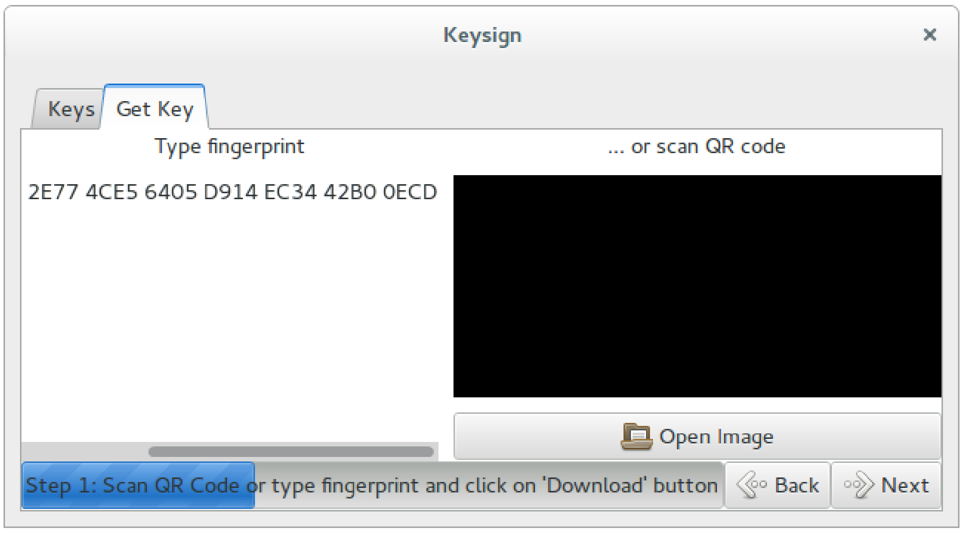
2 signatures

Figure 2: Key details and publish confirmation



9BAB CD21 9F12 2E77 4CE5
6405 D914 EC34 42B0 0ECD

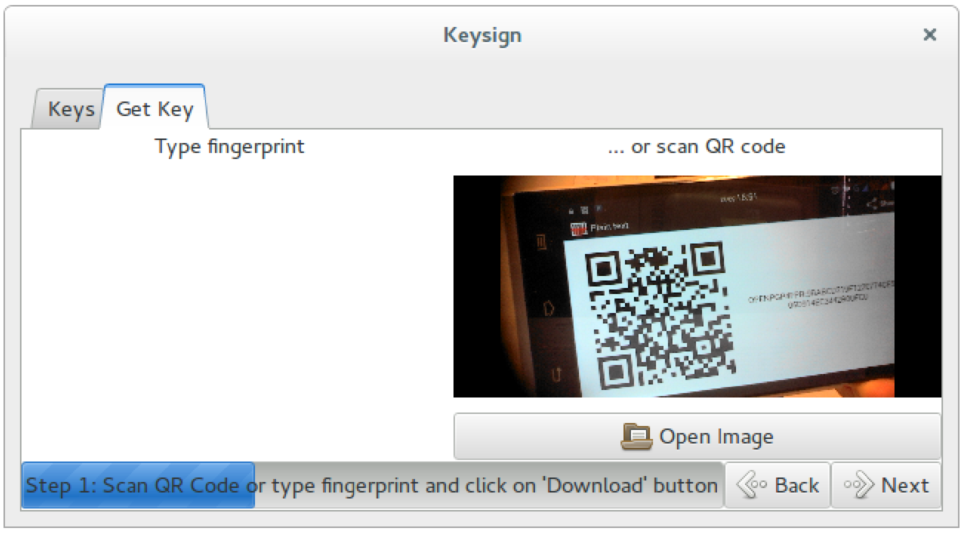
Figure 3: Fingerprint and QR Code representation of the selected key



2E77 4CE5 6405 D914 EC34 42B0 0ECD

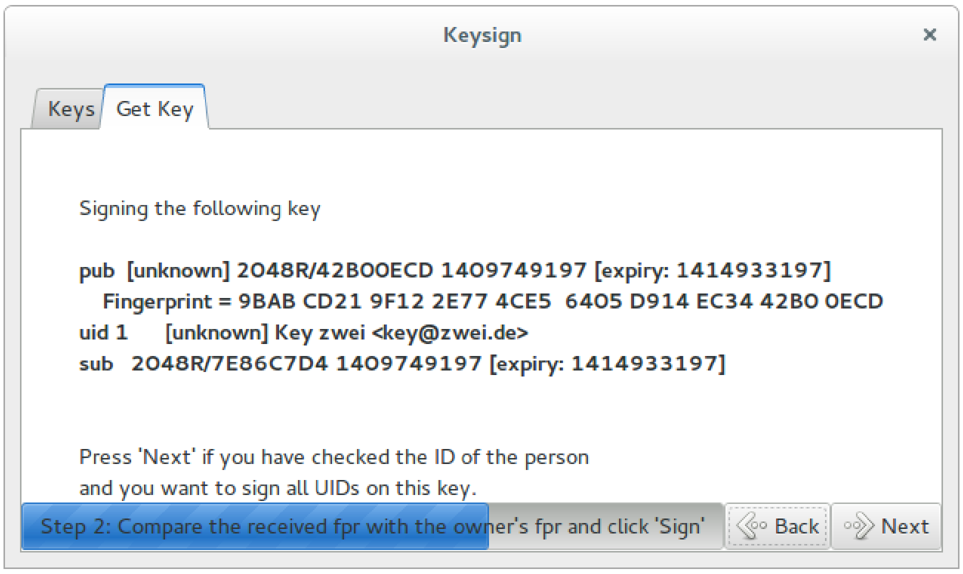
Step 1: Scan QR Code or type fingerprint and click on 'Download' button

Figure 4: Enter fingerprint or scan barcode of key to be signed



Step 1: Scan QR Code or type fingerprint and click on 'Download' button

Figure 5: Scanning of QR Code representation of the key's fingerprint



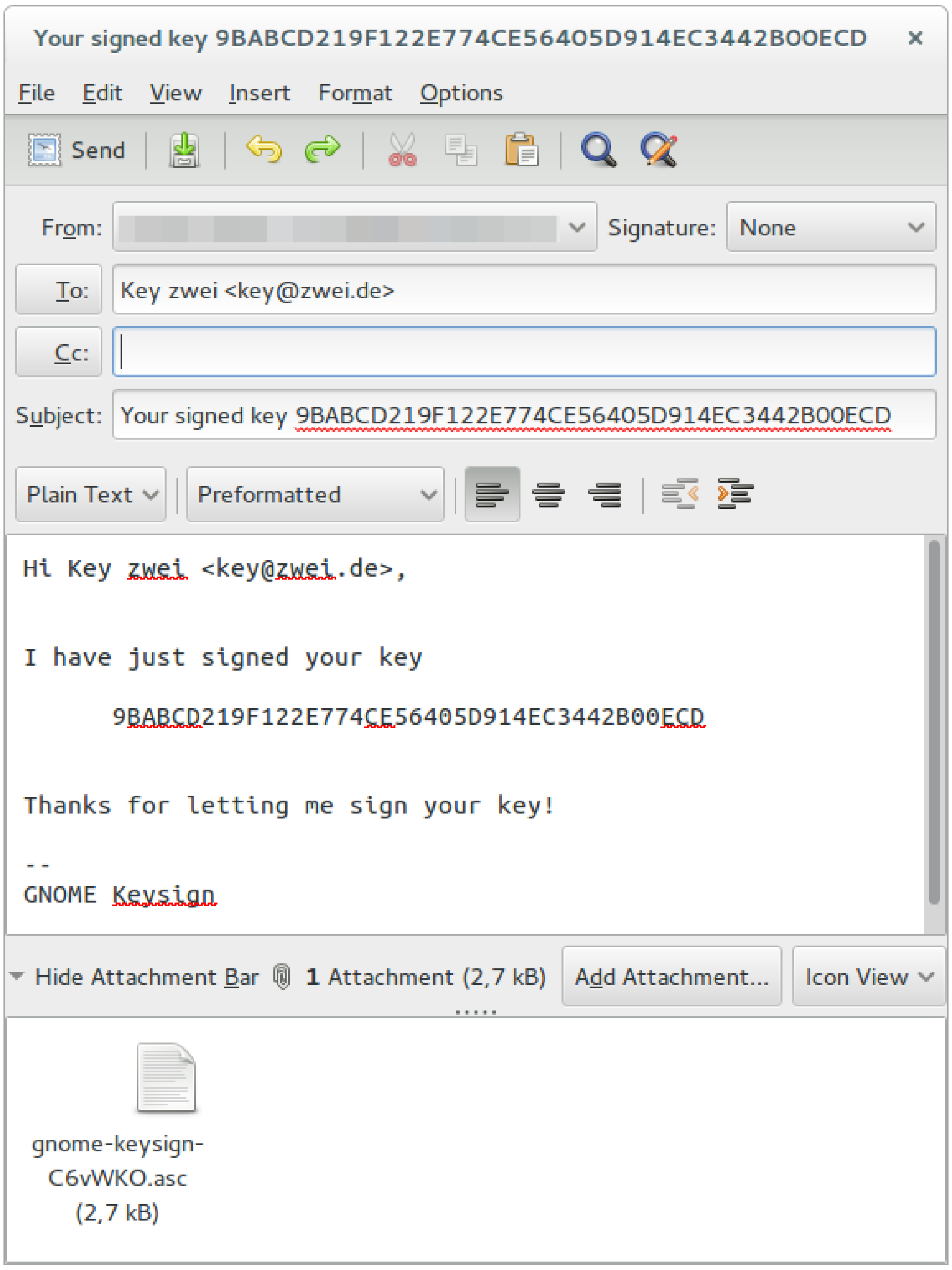
Signing the following key

```
pub  [unknown] 2048R/42B00ECD 1409749197 [expiry: 1414933197]
   Fingerprint = 9BAB CD21 9F12 2E77 4CE5  6405 D914 EC34 42B0 0ECD
uid 1     [unknown] Key zwei <key@zwei.de>
sub   2048R/7E86C7D4 1409749197 [expiry: 1414933197]
```

Press 'Next' if you have checked the ID of the person
and you want to sign all UIDs on this key.

Step 2: Compare the received fpr with the owner's fpr and click 'Sign'

Figure 6: Confirm signing


Your signed key 9BABCD219F122E774CE56405D914EC3442B00ECD
File Edit View Insert Format Options
Send

From: [redacted] Signature: None

To: Key zwei <key@zwei.de>

Cc:

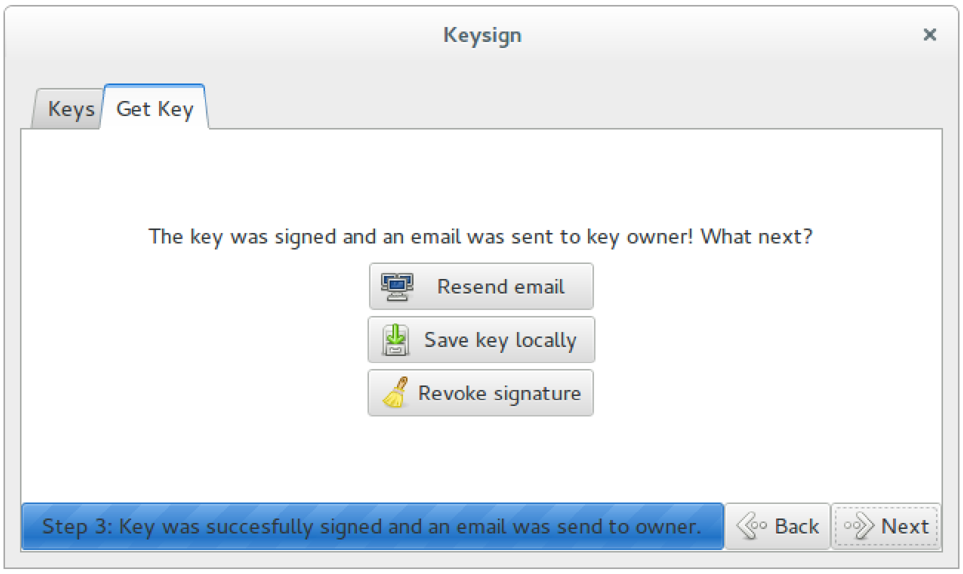
Subject: Your signed key 9BABCD219F122E774CE56405D914EC3442B00ECD

```
Hi Key zwei <key@zwei.de>,

I have just signed your key

      9BABCD219F122E774CE56405D914EC3442B00ECD

Thanks for letting me sign your key!

--
GNOME Keysign
```

Hide Attachment Bar
1 Attachment (2,7 kB)
Add Attachment...
Icon View

gnome-keysign-C6vWKO.asc (2,7 kB)

Figure 7: Automatic opening of composer



The key was signed and an email was sent to key owner! What next?

Step 3: Key was succesfully signed and an email was send to owner.

Figure 8: Keys signed