Binding Social Identity with Email Address and Automating Email Certificate Issuance

by

Reza Samanfar

A thesis submitted in partial fulfillment of the requirements for the degree of

Master in Computer Science

School of Computer Science

Carleton University

Ottawa, Ontario, Canada

June 2020

Copyright © by Reza Samanfar, 2020

The undersigned recommend to

the Faculty of Graduate Studies and Research

acceptance of the Thesis

Binding Social Identity with Email Address and Automating Email Certificate Issuance

Submitted by **Reza Samanfar**

in partial fulfilment of the requirements for the degree of

Master of Computer Science

Dr. Paul C. van Oorschot

Carleton University

2020

Abstract

End-to-end encrypted secure email remains a largely unsolved problem, from an adoption and deployment viewpoint. In this thesis, we propose a solution that can address two of the main challenges. Our solution involves: 1) Automated issuance and distribution of public-key certificates for use in email applications; 2) a means to make public keys of one user accessible to other users, in a manner that allows a cross-check of their authenticity. We use Keybase, a publicly accessible key database and key trust protocol, to bind a user's social identities to their email addresses. This enables other users to manually verify the social identity of their intended recipient in order to gain trust in their public keys. We also make use of ACME protocol used by an organization called Let's Encrypt for automated certificate issuance.

Acknowledgments

I would like to express my gratitude for my supervisor, Dr. Paul van Oorschot, for his guidance and valuable feedback that made this thesis possible and his support, that kept me going forward throughout my degree. I would also like to thank my family for their unconditional support.

Further acknowledgements are due to my colleagues Hemant Gupta and Christopher Bellman for their illuminating comments and feedback, and to Tshepo Kgengwenyane for their work on ACME Pebble test server and their feedback.

Also, I would like to thank Dr. David Barrera and Dr. David Knox for their valuable and illuminating comments.

Table of Contents

Abstract iii

Acknowledgments iv

Table of Contents v

List of Figures ix

1 Introduction 1

1.1 Introduction . . . 1

1.2 Motivation . . . 2

1.3 Problem Statement . . . 3

1.4 Design Requirements . . . 5

1.5 Contributions . . . 7

1.6 Thesis Overview . . . 8

2 Background and related work 10

2.1 Public Key Infrastructure (PKI) . . . 10

2.1.1 Public key cryptography . . . 10

2.1.2 Digital signatures . . . 11

2.1.3 Certificates and Certificate Authorities (CAs) . . . 11

2.2 Email security . . . 12

2.3 S/MIME and PGP 13
2.4 Keybase 13
2.4.1 Keys on Keybase 14
2.4.2 Accessing device-specific keys and password change 15
2.4.3 Signature chain (sigchain) 17
2.4.4 Is Keybase a trusted server? 24
2.4.5 Keybase's goal 29
2.4.6 Following on Keybase 31
2.4.7 Registering on Keybase 32
2.5 ACME protocol 35
2.5.1 Let's Encrypt 35
2.5.2 ACME 36
2.6 Bitcoin background 38
2.6.1 What is Bitcoin? 39
2.6.2 Block chain 40
2.6.3 Interaction with Bitcoin blockchain for verification on Keybase 40
2.7 Related work 43

3 Threat model and requirements 45
3.1 Keybase threat model 45
3.1.1 DDos attacks against Keybase servers 46
3.1.2 Keybase server compromise 46
3.1.3 Defence mechanisms for defending against server corruption . 47
3.2 ACME Threat Model 47
3.2.1 Authorizations on ACME 49
3.2.2 Denial-of-Service attacks 53
3.2.3 Request Forgery 54

3.2.4 Certificate issuance policy 54
3.2.5 Replay protection 55
3.3 Our Design's threat model 55
3.3.1 Middle-person attacks 56
3.3.2 Server-side threats 56
3.3.3 Client-side threats 58
3.3.4 Impersonation attacks 58
3.3.5 User account compromise 59

4 Design and Proof of Concept 60

4.1 ACME Server 62
4.1.1 Protocol for interacting with ACME server 62
4.1.2 Challenges from ACME server 63
4.1.3 ACME Test-Server 65
4.1.4 Required changes for the ACME client 73
4.2 Sending Encrypted Email Between Two Parties 75

5 Security analysis 81

5.1 Middle-person attacks 81
5.2 Server-side attacks 83
5.3 Client-side attacks 84
5.4 Impersonation attacks 85
5.5 Account compromise 86

6 Comparative Analysis and Conclusion 88

6.1 Comparison of secure email solutions 88
6.1.1 Target users 90
6.1.2 Certificate Format 90

6.1.3 Software download requirements 91
6.1.4 Difficulty of certificate acquisition 91
6.1.5 Acquiring correspondents' certificates 92
6.1.6 Ease of gaining trust 93
6.2 Limitations and Future work 94
6.3 Conclusion 97

List of References **98**

List of Figures

1.1 All the entities involved in our proposed design 7

2.1 A sample Merkle tree structure 26

2.2 Process of validating a Merkle tree's root pushed to the Bitcoin blockchain 27

2.3 A sample posted Keybase proof on Facebook 34

2.4 Social media account binding process 35

3.1 Communications Channels Used by ACME 48

3.2 Middle-person Attack on validation channel 51

4.1 Interaction with ACME server 64

4.2 The components of our design. This picture is a copy of Figure 1.1 . 66

4.3 A sample sign up page for our ACME server 67

4.4 A self-signed certificate generated by OpenSSH for the ACME server 73

4.5 Mail exchange between Alice and Bob 78

4.6 Account setting page for Microsoft Outlook 2016 79

6.1 Comparison of alternative secure email solutions 89

Chapter 1

Introduction

1.1 Introduction

Electronic mail has been a primary means of communications between peers and within corporations for many years. As Philip Zimmermann, the creator of PGP protocol, says "It's personal. It's private. And it's no one's business but yours. You may be planning a political campaign, discussing your taxes, or having a secret romance. Or you may be communicating with a political dissident in a repressive country. Whatever it is, you don't want your private electronic mail (email) or confidential documents read by anyone else"[1]. Ensuring privacy, however, requires email to be encrypted due to the fact that sensitive and personal data is otherwise transmitted in cleartext over the internet by emails. Encryption protects the contents of email from being accessed by adversaries and third parties other than their intended recipient.

Most secure email products use public key cryptography in which each user has a pair of public-private keys. Within public key infrastructure, users can share their public keys with other users to enable others to send them encrypted emails. Also, they can use their signing private keys to sign over the emails they themselves send to other users, to provide both data integrity and data origin assurances.

[1] Part of the Original 1991 PGP User's Guide (last updated in 1999)

Today, many email service providers such as Google and Microsoft support encryption for emails that are sent through their servers; however, this is different from what might be required. Encrypting email on transport layer means that email is encrypted from each one server to another along the route, but on the other hand, end-to-end encryption for email means that the email is encrypted on the sender's device and will only be decrypted on the receiver's device (encryption and decryption is done on the end devices only). This does not allow service providers such as Google to read the contents of the email.

There are many secure email products. Most are based on either PGP [46] or S/MIME [39]. Usability is among the challenges preventing users from adopting these products [45], [8], [36]. In this thesis, we propose a design to improve the security of email communications in comparison to existing secure email products. We also argue that our design improves some of the shortcomings of aforementioned products from the usability perspective. This is because our design automates some parts of the certificate issuance and it does not rely on any particular mail client to operate. These advantages could increase the adoption of secure email, and thus improving the security of the email ecosystem as a whole.

1.2 Motivation

Email, by default transmitted unencrypted as cleartext, is susceptible to passive network eavesdropping and active network attacks. Encrypted email tools have traditionally faced usability challenges. These challenges have negative effects on adoption of secure email even among communities that desire secure communication (e.g. lawyers) [24]. Today, most emails are not end-to-end encrypted and use of email encryption is typically driven by strong motivations that are present in their environment. For enterprises, secure email products based on S/MIME are being used,

while for non-enterprise users PGP has been popular. PGP's lack of adoption and severe usability issues largely revolve around its key management. S/MIME on the other hand requires trusting an authority to enable trust on issued certificates and is usually used in organizations with a dedicated IT team. Email certificates are public-key certificates which carry, as components, public keys that have corresponding private keys. A user's signing private keys are used for signing email messages, while encryption public keys are used for encrypting symmetric keys which are then in turn used for encrypting the content of email messages. Signature on emails can help the recipients verify that the email is not altered while in transit and encryption is for preventing eavesdropping by third parties.

Being able to automate some processes of issuing S/MIME certificates in a way that it does not require a dedicated IT team, and being able to tie users' public keys that are within their certificate with their social identities, can reduce human errors or mitigate some attacks that may result in exchanging messages with someone other than the user's intended recipient.

1.3 Problem Statement

Existing secure email solutions are mainly based on S/MIME and PGP. Typical product solutions and free implementations built around each of these technologies have their drawbacks. The drawbacks that we focus on in this thesis are directly related to managing and trusting public key certificates.

S/MIME users often work within an organization that has its own internal Certification Authority (CA). This means that users from outside that organization who do not have that CA's public key as a trust anchor within their own client software have no easy, reliable way of acquiring and validating certificates signed by that CA.

On the other hand, PGP users typically prefer to avoid relying on CAs for trust

in public keys of end-users. Instead they use the so-called PGP web-of-trust, which in essence means using manual methods for exchanging and trusting the public keys of other users. As a result, PGP suffers from key distribution and key revocation problems. The key distribution problems often result in PGP users uploading their public keys to various PGP public key servers, or using out of band communications to gain trust in their correspondents' public keys.

The problems that this thesis addresses are within the categories below:

- Users acquiring their own email certificates
- Users renewing their own email certificates
- Users acquiring the email certificates of email correspondents
- Users establishing trust in acquired certificates of email correspondents including upon certificate renewals
- Impersonation threats for email (i.e., abuses related to fraudulent or untrustworthy certificates)

In this design, our goal is to facilitate the email certificate acquisition process and make it a service that is globally available to users with a variety of technical backgrounds. By following the footsteps of Lets Encrypt this service would ideally be free of charge so a wider range of users can adopt this secure email practice. This applies to both acquiring a certificate for the first time and, to the certificate renewal process as required on a periodic basis. By our design, while many actions must be performed at the first instance of certificate issuance (i.e. registration processes and social media bindings), fewer are required to renew a certificate.

Our design proposes to bind users' social media identities to their email certificates. We create a link between a user's email address, their social media identities

and, their public keys. This will require cross-checks by both the CA that issues email certificates during the certification process, and manual cross-checks by end-users at later stages. We believe that this approach will add protection against current impersonation threats in secure email. To achieve this goal we need to have a safe method of email certificate distribution amongst email correspondents, which should be easier than present methods, while reducing the need of out-of-band communications amongst email correspondents.

1.4 Design Requirements

We now present our design requirements.

1. A mechanism to facilitate X.509v3 email certificate acquisition of a user's own certificates, and those of correspondents

2. A cryptographically sound link between social identities and public keys of email correspondents by using identical key pairs to verify social media accounts and prove the ownership of the keys

3. A means for establishing trust in public key certificates of email partners by manual cross-check of social identities by users at the first encounter with a certificate or after renewal of previously trusted certificates

4. A mechanism to make email certificates available to email correspondents

5. Facilitating a certificate renewal process requiring no tremendous user effort

6. Providing protection against email impersonation attacks

7. Protecting against Keybase and ACME server compromise, by relying on cross-checks between design components

Within this design, we present a modification of the ACME protocol to automate some parts of the email certificate issuance. The goal is to enable users to acquire their own email certificates with greater ease, and to bind a user's social identities, public keys and, email address all together. This connection is formed by a series of cryptographic challenges presented in the Keybase and ACME registration processes that with some cross-checks in the registration phase, leads to a sound bond between the three elements mentioned above. Users will be required to manually cross-check the social identity of their email correspondents when presented with new certificates. This will allow confidence in the authenticity of the certificates, i.e., they hold the legitimate public keys of the correspondent that they are trying to communicate with. If performed correctly, these cross-checks provide protection against email certificate impersonation attacks.

To simplify our design, we suggest that our email certificates be made publicly available by incorporating them into Keybase's file system. This however is not a design requirement; valid certificates acquired from other sources can also be trusted based on the cross-checks within our design. By our design, certificate renewal is also semi-automated in the sense that many of the initial actions needed to acquire a certificate for the first time are eliminated. This could convince secure email users to continue using this practice if the renewal process does not require a tremendous effort.

While within this design we introduce two main components (Keybase and ACME), we believe that a server compromise in one of the servers would have much smaller impact on the user compared to a secure email user that relies solely on a single trusted server. This is due to the cross-checks between Keybase and ACME in the certificate issuance process. Use of X.509 v3 certificates and importing them into mail clients is not one of our design requirements, although we note that many current email clients already support, by various means, importing and using these

certificates.

1.5 Contributions

Our main contribution is proposing a design that can bind users' social identities, their public keys, and their email addresses all together enabling other users to verify the sender of the email through social media and the authenticity of the received email. Binding a user's social identity with their public keys is done by Keybase, and automating parts of issuing email certificates (binding the keys to an email address stated within the certificate) is done by modification of the ACME protocol discussed in Section 4.1.3 [4]. Figure 1.1 shows all the involved parties in our proposed design along with all the required interactions that are shown by arrows between entities.

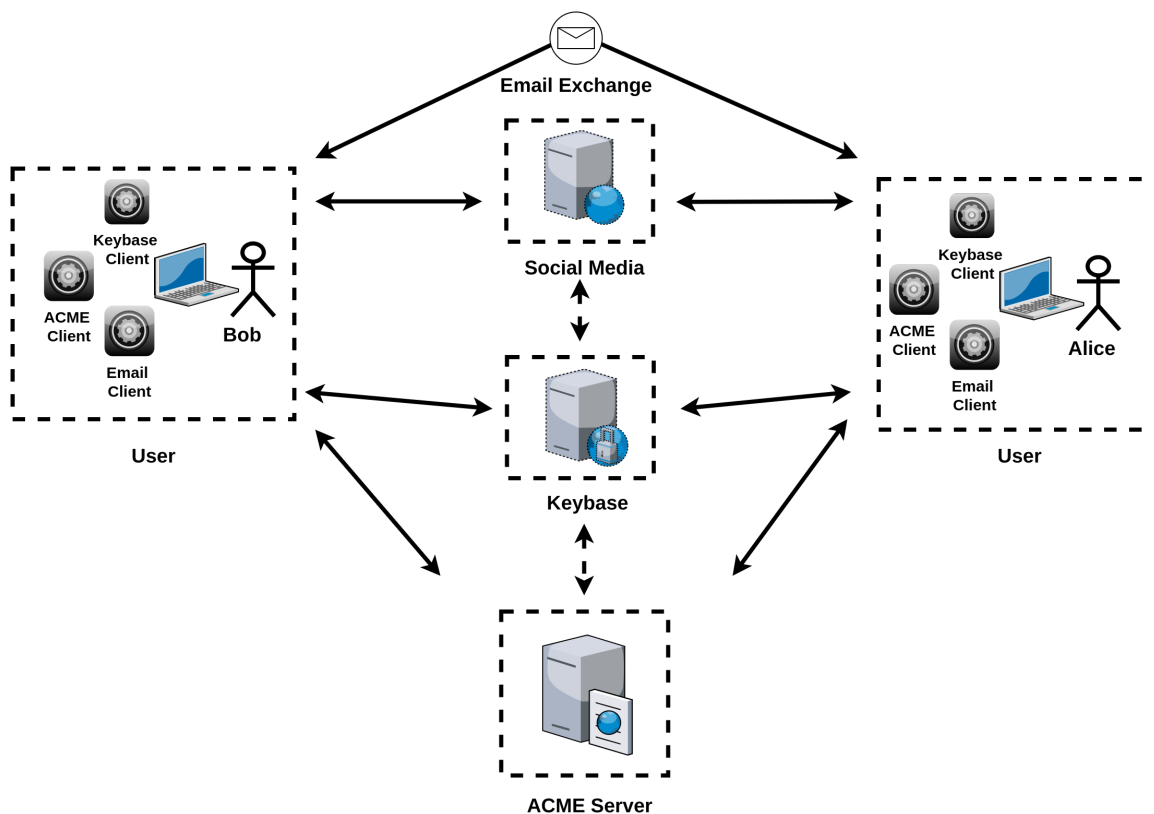

Figure 1.1: All the entities involved in our proposed design

For two users to exchange encrypted and signed emails between each other based on this design they each need to have an account on a social media platform that is supported by Keybase. They are required to register an account and prove ownership of their social media accounts on Keybase over various steps discussed in Section 2.4.7. Then, they would each register an account with the ACME server and proceed to complete the actions required to receive two email certificates (one for encryption and one for signing). The steps for acquiring an email certificate are discussed in Section 4.1. In order to interact with Keybase and ACME, users will need to install Keybase and ACME clients on their systems. At the end, in order to store other user's certificates and to send and receive signed and encrypted emails they would need to have email clients with S/MIME support on their system.

We also present a threat model for our design, and provide arguments supporting the claim that our proposed solution improves email security over existing secure email products. We show that binding a user's social identity with their email address can help mitigate some of the threats which involve a human error or a targeted attack. We argue that integrating this binding while reducing interactions required to obtain and renew a certificate, improves security and makes certificate issuance more accessible for users of widely varying technical backgrounds.

1.6 Thesis Overview

The remainder of this thesis is structured as follows. Chapter 2 explains background concepts of email security followed by introductions to Keybase in Section 2.4, ACME protocol in Section 2.5, and Bitcoin blockchain in Section 2.6 to help readers follow the verification process introduced in Keybase section. Chapter 3 discusses threat models for Keybase, ACME, and our design as a whole. Chapter 4 presents our proposed design for binding users' public keys, email addresses, and social identities. Details of

the changes required to present a proof of concept is also discussed. Chapter 5 returns to the threat model discussed in Chapter 3 and discusses how our proposed design addresses each of the threats. Chapter 6 discusses a comparison between currently available alternate secure email solutions, concludes with a discussion of the benefits and limitations of our design, and future work.

Chapter 2

Background and related work

In this chapter, we go over fundamental concepts that are related to email security, followed by an introduction to Keybase and the ACME protocol. In Section 2.7, related work and the differences between our work and similar related works are emphasized.

The fundamental concepts include an overview of Public Key Infrastructure and digital certificate, email security, Pretty Good Privacy (PGP), and Secure/Multipurpose Internet Mail Extensions (S/MIME). Keybase is discussed and its key security and sigchain structure is explained. This is followed by an overview of the ACME protocol. Section 2.6 provides an introduction to Bitcoin block chain and how to interact with it to perform verification is included.

2.1 Public Key Infrastructure (PKI)

In this section we discuss Public Key cryptography and related topics.

2.1.1 Public key cryptography

Public key cryptography, or asymmetric cryptography, is a form of cryptography in which there is a pair of keys for encryption and decryption. The key pair includes

a public key and a private key. As the name suggests, the public key can be shared publicly with other users. They can encrypt data using that key and the user who holds the corresponding private key (hopefully the only user who must keep it secret), can decrypt the data [44].

2.1.2 Digital signatures

Digital signatures may be attached to the messages sent and they can be verified with the public key of the user signing the message. Based on the assumption that only the signing user has the private signing key and the assumption that the corresponding public/private key pair actually belong to that specific user, other users can have confidence that the message originated from that user (data origin authentication). Due to the fact that the signature of the message depends on the message itself (user signs the hash of the message with his private signing key), users that verify the signature have some assurance that the message has not been tampered with (data integrity). Another property of digital signatures is non-repudiation. Due to the data origin evidence, it is hard for a party to deny signing a message when he has done so using his private signing key that corresponds to his authenticated public key [44].

2.1.3 Certificates and Certificate Authorities (CAs)

Users who are encrypting data with the public key of their intended recipient have to make sure that the public key actually belongs to the person they are sending the data to. In this scenario, digital certificates can be helpful.

Digital certificates are data structures that bind an identity to a public key. The essential fields in digital certificates are subject name, the public key that is known to be associated with that subject name, and the signature of the third party that has issued the certificate (the signature is over all the fields within the certificate). That

third party is called a Certificate Authority (CA), and to trust the signature of the CA over a certificate means having confidence that the CA has verified the legitimate association of the identity named on the "subject name" field with the public key indicated in the certificate. This requires the CA to perform a verification process prior to issuing a certificate for an entity. There are various methods of verification in which CAs pose challenges to entities requesting a certificate. These challenges are to provide evidence of ownership of the private key that is claimed to be owned by the entity and also evidence that the requesting party has a legitimate claim to the asserted identity or subject name. To trust a certificate, relying parties should verify the signature of the CA over the certificate. This requires having certified knowledge of the mentioned CA's public key. In the web applications, the public keys of these CAs are hard-coded into well-known web browsers (as self-signed certificates provided by the browser vendors). These CA's are also called trust anchors [44].

2.2 Email security

Secure email aims to provide three security properties: confidentiality, integrity and data origin authentication. Email should be encrypted so that only the sender and the intended recipient possess the means to access the contents. Use of authenticated encryption or digital signatures allows detection if the contents are altered. Within the public key infrastructure, each user has a pair of public-private encryption keys and public-private signing keys. The public part of the key can be advertised to other users, but the private part should only remain known to owner of the key pair. One user being able to digitally sign data enables other users to verify the origin by checking the signing party's signature with the signing party's public key. Assuming that each party knows the public key of the other one and exchanged emails are encrypted and signed, we can achieve confidentiality and integrity. In other words,

the receiver of an email can have confidence that the email was encrypted such that no adversary could have had access to the clear text while it was being transmitted, and by checking the signature on the email, they can detect if the contents of email were altered. The digital signature also provides data origin authentication, i.e., confidence in who the originator of the email was with the same assumption of certified public keys.

2.3 S/MIME and PGP

S/MIME[1] [39] and PGP[2] [46] are two systems that enable secure email. Both allow encryption and decryption of data, plus digital signatures and signature verification for data origin authentication. They allow users to sign, verify signatures, encrypt, and decrypt data transmitted over the internet.

PGP users directly exchange public keys with each other, and each user must choose which keys to trust and the duration of that trust is up to them. With S/MIME, parties rely on mutually trusted third parties (Certification Authorities). The CAs issue public key certificates as explained in Section 2.1.3. S/MIME will be our main focus.

2.4 Keybase

Keybase is a publicly-auditable directory of keys and *identity proofs* [24] [17]. The main function of Keybase is to store their users' public keys in a manner that allows other users to trust the keys. Keybase also provides cloud storage for users, and users can upload and store any type of file on their storage in the Keybase file system

[1]Secure/Multipurpose Internet Mail Extensions
[2]Pretty Good Privacy

(KBFS)[3]. In mid 2017, Keybase introduced Keybase Teams. This allows a group of users under a single name to share a private folder for storing their files and have their own chat channels. This functionality is similar to programs such as Slack[4]. Keybase also offers client software that can be installed on users' devices, matching the functionalities of the website interface. In addition, it provides secure end-to-end encrypted chat and easy access to the KBFS.

Keybase enables users to cryptographically bind their public key to their social media accounts. The steps of registration and account binding are discussed in Section 2.4.7. Every user has their public keys on their Keybase profile (usually the main PGP key that is created in the sign up process but, users can add more keys if they wish.), a list of devices that are associated with their account, and a list of social media accounts that are verifiable using Keybase as being associated with the account.

2.4.1 Keys on Keybase

When Alice joins Keybase, she can either generate a key pair with the Keybase client or import an existing key pair of her own. The public and private keys are denoted K_A and k_A, respectively. The first such pair that Alice registers is called Alice's *eldest key pair*. It is the first among many keys that will be called *siblings* (*sibkeys* and *subkeys*) as detailed in section 2.4.6.

Each time that Alice uses a different device that she intends to be used to interact with Keybase, there are two per-device key pairs generated by the Keybase client installed on Alice's device: a signing key pair called "sibkey" and an encryption key pair called "subkey". The keys are produced based on *elliptic curve cryptography* (ECC) in which operations are over sets of defined points on an elliptic curve. This offers computational efficiency over *integer factorization cryptography* [44]. "Curve25519" is an

[3]https://keybase.io/blog/kbfs
[4]https://slack.com/

elliptic curve that offers 123 bits of security and is used in elliptic curve Diffie-Hellman key agreement. Ed25519 is Edwards-curve Digital Signature Algorithm (EdDSA) using Curve25519. The process for generating keys on Alices's i^{th} device is [16, 37]:

1. An Ed25519 signature key pair ($S_A{}^i$, $s_A{}^i$) is randomly generated by Keybase client consisting of a 32-byte public key and a 64-byte private key. These are the new sibkeys.

2. A Curve25519 encryption key pair ($N_A{}^i$, $n_A{}^i$) is generated that are both 32-byte strings. These are the new subkeys.

3. $S_A{}^i$ is signed with a valid sibling key (or the eldest key on Alice's first device) and the new key can sign the delegated key as well. The sibkey signature alongside the public key itself then is pushed to Alice's *signature chain* as a statement (explained in Section 2.4.3).

4. $N_A{}^i$ is signed with $s_A{}^i$ and this subkey signature, alongside the public key itself, is pushed to Alice's signature chain as a statement.

5. The private keys $s_A{}^i$ and $n_A{}^i$ are stored locally on the device i.

2.4.2 Accessing device-specific keys and password change

Each of the user's *device-specific* keys are stored in a password protected manner on its associated local devices [16]. The user would need to provide his Keybase password in order to decrypt his locally stored device-specific private keys. However, if the user decides to change his password, there should be a way to access the encrypted and locally stored keys with the new password on all the devices (even the devices that were offline during the password change). Keybase uses a server-aided protocol to ensure that the password change will be reflected on all of the user's devices [16].

In this protocol, Keybase uses a server-side *mask* for decrypting device-specific keys. This mask is updated after a password change. This updated mask will enable the user's other devices to decrypt their locally stored keys with the new password instead of the old password. This mask is provided by the Keybase server in the decryption process and user's device keys are stored locally at all times and are not exposed to the Keybase server even in the encrypted form [16].

Bob, a Keybase user with the password P_B, makes use of this protocol to encrypt and decrypt his device-specific keys as shown in the following steps [16]:

1. Keybase client on Bob's device *d*, generates a new random secret key ${k^d}_B$ (symmetric), and encrypts device-specific keys (stored locally on device *d*) with it. The client computes:

$$s^{d}{}_{B} = k^{d}{}_{B} \oplus c_{B} \tag{2.1}$$

   Here c_B= Scrypt (P_B) is a symmetric key derived using the password-based key derivation function *Scrypt* designed to thwart brute force attacks with memory complexity [32, 33].

2. The client then sends ${s^d}_B$ to the server, which stores it in a data structure associated with device *d*.

For encryption and decryption on device *d* for Bob, the steps below will be performed [16]:

1. Bob needs to authenticate himself to the server and, Bob's client computes c_B= Scrypt (P_B).

2. The client retrieves ${s^d}_B$ for device *d* from Keybase server and computes ${k^d}_B = {s^d}_B \oplus c_B$.

3. Then Bob's client can encrypt or decrypt his device-specific private keys using $k^d{}_B$ and "NaCl's SecretBox" [5, 6, 41]. NaCl Secretbox is a package that uses XSalsa20 cipher and Poly1305 message authentication code to encrypt and authenticate small messages [40].

In the event of a password update, Bob should be able to encrypt and decrypt all of his device-specific keys with the new password. In order to do so, Bob needs to perfom some actions, explained below, while changing his password from P_B to P'_B [16].

The client computes both c_B= Scrypt (P_B) and c'_B= Scrypt (P'_B) in order to compute $\delta = c_B \oplus c'_B$ which needs to be sent to the Keybase server. On the Keybase server s'_B is computed where: $s'_B = s^d{}_B \oplus \delta$ is computed for all of Bob's devices. The sever stores s'_B, replacing all s_B for all of Bob's devices. Now if Bob tries to use the new password from another device that was offline during password change, the server will provide the new s_B (that is s'_B) instead of the device's old s_B and the steps would be identical to the steps shown above for encryption and decryption on device d [16].

2.4.3 Signature chain (sigchain)

There is a public signature chain for every Keybase account called *sigchain*. Sigchain is a verifiable list of ordered statements that indicate the changes made to the account over time [20]. Every user's sigchain is stored on Keybase's servers. When a user adds a key, connects a social media account, or "follows" another user, their Keybase client signs a new statement which is called a "link" and embeds the link (with the signature included in it) into their sigchain (creating a new statement and appending to user's current sigchain). Each of these links are signed by one of the users's keys described in Section 2.4.1. Each link has a sequence number and the hash of the previous link

in the sigchain (hash over the entire link). The use of this hashed sequence number prevents a dishonest Keybase server from creating or omitting links without making the entire sigchain invalid. A full link statement with some fields removed is shown below. This can help the readers understand the key values that are stored in each statement of Keybase sigchains.


```
1   {
2   "status":
3   {
4     "code":0,
5     "name":"OK",
6   },
7
8   "sigs":
9   [
10    {
11      "seqno":1,
12      "payload_hash":"cf1f5993b254029132690594f95
13            12ee6fc49340fc347234fe4000b95e162bb18",
14      "merkle_seqno":2742703,
15      "sig_id":"c145cf3c44d8d8d3898f2098e3e2103
16              9c95effb7e92e22351fbeb0724eedbb630f",
17      "sig_id_short":"wUXPPETY2NOJjyCY4-IQOcle_7fpLiI1H76w",
18      "kid":"0120f92d63fe30bdc2702bce66ccf5cf5
19             180b850af8e12e50620c5b0944ffa8a5e9e0a",
20      "sig":"hKRib2R5hqhkZXRhY2hlCpHR5c...kea3s5UGr+w
21             UPPaK1wirbov6o08iBKN0YWfNAgKndmVyc2lvbgE=",
22      "payload_json":"{\"body\":{\"device\":{\"id\":\"330044a21db4ad
          ... \",\"kid\":\"0120f92d63fe30bdc270ull,\"seqno\":1,\"tag
          \":\"signature\"}",
23
24      "sig_type":1,
25      "ctime":1526405128,
26      "etime":2030981128,
27      "rtime":null,
28      "eldest_seqno":0,
29      "sig_status":0,
30      "prev":null,
31      "is_eldest":1,
32      "fingerprint":"",
33    }
34  ],
35 }
```


Listing 2.1: A full link statement with some fields removed

All Keybase users' sigchains are publicly available (both within their own Keybase profile and through the Keybase API) and any user can verify the cryptographic validity of the signatures on them using Keybase's API (that finds the user's public key from their sigchain) or by using a web browser. Users can add or remove their

own sibkeys by adding new links to their existing sigchain, and any of user's sibkeys can be used to sign links [20]. To be valid, a link must be verifiable based on the valid user's keys at that particular point in time (and the time that the link statement has been generated is "ctime" field in the link). Thus, if the old sibkeys were to be revoked, old links remain valid if they were signed by keys that were valid at that point in time and the signed statements were legitimate (although the keys may be revoked afterwards and new statements with revoked keys are not valid). This means that revoking one particular key pair will not have an effect on a user's other keys, previously signed identity proofs, and the snapshots that other users following this user, have.

To see if a Keybase account is valid and to find the keys that are currently valid for that account, a Keybase client needs to receive the Merkle tree (described further in Section 2.4.4) from the Keybase server. In that Merkle tree, the last statement of all the Keybase users are present. When the Keybase client finds the last statement of the sigchain of the account that its trying to validate, it assumes that the key that are specified on the last statement is a sibkey. Then the client moves from one statement to the previous statement based on the sequence number of the links in the sigchain (from the last statement to the first statement). While navigating from one statement to the other, the Keybase client verifies the signature on each of the statements based on the valid keys (sibkeys) that it has already encountered. In the process the Keybase client keeps track of all the valid sibkeys and takes into the account the effect of other links on those sibkeys (if the key was revoked afterwards or a new key has been added) [20]. The listing 2.2 that is shown below, is the first statement of a user's sigchain. All the statement types and related fields are explained in the following paragraphs.

```
1        {
```

```
2    "body": {
3      "device": {
4        "id": "330044a21db4ad9c20f8e86849323218",
5        "kid": "0120f92d63fe30bdc2702bce66cc
6         f5cf5180b850af8e12e50620c5b0944ffa8a5e9e0a",
7        "name": "Mobile Device",
8        "status": 1,
9        "type": "mobile"
10     },
11     "key": {
12       "host": "keybase.io",
13       "kid": "0120f92d63fe30bdc2702bce66ccf5
14           cf5180b850af8e12e50620c5b0944ffa8a5e9e0a",
15       "uid": "4a35d1e3eb7a1542babcad5035458719",
16       "username": "samanfar"
17     },
18     "merkle_root": {
19       "ctime": 1526405127,
20       "hash": "833871618c3a0827e83d832d79afb03eca3
21               ab3cbe8b0623612062091b094c8b3c755d5f033b074
22               52aec2adbfeb2359c0330cf95487b8f4b993183c7f96bbe7c0",
23       "hash_meta": "09a1a7649d1e425fb03f2f74c6d1bca50e0055375d46ad
24       6fc7c0e7443e53cca5",
25       "seqno": 2742702
26     },
27     "type": "eldest",
28     "version": 1
29   },
30   "client": {
31     "name": "keybase.io go client",
32     "version": "1.0.47"
33   },
34   "ctime": 1526405128,
35   "expire_in": 504576000,
36   "prev": null,
37   "seqno": 1,
38   "tag": "signature"
39 }
```

Listing 2.2: The first statement of a user's sigchain

There are various types of links (statements) in a sigchain as they are listed below. We are going to describe each of these links based on the official description on Keybase's documentation [20]:

1. **Eldest statement:** It holds the first key for the account and the signing key for the link becomes the first sibkey. It is the first statement in a Keybase user's sigchain (unless there has been an account reset by the user). This type of links is specified by the *"type": "eldest"* in the body of the statement.

2. **sibkey statement:** It is a statement that adds a new sibkey to the account.

This type of links is specified by the *"type": "sibkey"* in the body of the statement.

3. **subkey statement:** It is a statement that adds a subkey to the account. subkeys are used for encryption only. This type of links is specified by the *"type": "subkey"* in the body of the statement.

4. **revoke statement:** It is a statement that removes a key from the list of the account's valid keys. This is done by removing the key from the list of Key-IDs (*kids*). Removing a key however, does not affect the validity of the statements that were signed using this key (each link has the information of the keys that are used for signing it). After revocation, these keys can no longer sign new statements. This mean when Keybase clients of other users are navigating this user's sigchain, they should remove these keys from the list of the valid keys while coming across a revoke statement. This is all done by the Keybase client and users do not need to remember which keys are revoked. This type of links is specified by the *"type": "revoke"* in the body of the statement.

5. **PGP update statement:** It is a statement for updating the PGP key for the account and replacing it with a new one (which may result in adding or removing subkeys etc. to the account). This is because Keybase currently only supports PGP and there is no support for S/MIME. This type of links is specified by the *"type": "pgp_update"* in the body of the statement.

   Currently, Keybase client only lets users generate a PGP key pair. Because both PGP and S/MIME use similar algorithms to generate public and private keys (such as RSA [35]), it is possible to extract those keys from the generated PGP key block on Keybase and use them in a S/MIME certificate. A theoretical example is using The Monkeysphere Project[5] to extract the public and private

[5] http://web.monkeysphere.info

keys from a PGP key block and using them to get an X.509 certificate. This is yet another challenge from a usability point of view, but this problem can be solved by enabling support for key types other than PGP keys within Keybase, or including this functionality within the ACME client. In this way, the ACME client would receive user's PGP keys and extract public keys from the PGP blocks. But, private keys should be extracted locally. In this thesis, in order to demonstrate the benefits of this design, we assume that both keys on Keybase and ACME client are the same type (S/MIME).

6. **Web service binding:** It is a statement that indicates the ownership of a certain "username" on a website "domain" (a supported social media page or a supported account). When the proof has been posted for the first time, the server looks for the proof and verifies the signature on that post. If it was verified, the server keeps the address of that post to help the Keybase clients for further checks. The Keybase client also checks that proof for itself and verifies if the signature on that post is valid. This type of links is specified by the *"type": "web_service_binding"* in the body of the statement.

7. **track and untrack statement:** Tracking statement is making a snapshot (more information for tracking can be found in Section 2.4.6) of another users identity that is provided by the Keybase server. This can be used by a users other devices to trust the same user without the need to manually verify the user again (however, the Keybase client will perform the checks anytime that it is needed). This type of links is specified by the *"type": "track"* in the body of the statement. However, if a user stops following another user, a untrack statement is required. This statement makes all of the user's devices to not trust the snapshot that they already have and check the other users identity proofs and present them to the user whenever the user is interacting with them.

This type of links is specified by the *"type": "untrack"* in the body of the statement.

8. **cryptocurrency statement:** Is a statement for advertising a cryptocurrency address. This type of links is specified by the *"type": "cryptocurrency"* in the body of the statement.

9. **per user key statement:** It is a statement for adding or rotating (revoking a "PUK", creating a new one, and sharing that with all of a user's devices) two per-user signing and encryption keys. With the introduction of *teams* on Keybase, they introduce a new type of key. The Per-User Key (PUK) which is used for signing and encrypting messages and files within a team. The public part of each user's PUK is encrypted for all of other team members using their public PUK. The private per-user key is encrypted for all of a user's active devices so the user can interact with the team from all of his active devices. The public PUK is also advertised in the user's signature chain. When a user adds a new device, a private PUK is encrypted for the new device. When the user revokes a device, a new PUK is generated, all of user's registered devices get the new private PUK. The generation number starts at one and increments whenever the per-user keys are rotated, typically after a device revocation (a device revocation happens when a user removes one of the previously verified devices from his list) [19]. This type of links is specified by the *"type": "per_user_key"* in the body of the statement.

It is noteworthy that the "type" mentioned in "device" section on line 9 of Listing 2.2 is different from the type of the statement which is shown on line 26 of the same listing. The first "type" indicates the type of the device that is creating the link, and the second "type" indicates the type of the link that is being created.

As it is shown in the listing 2.2, a statement has many fields. There are many

fields that are repeated in all types of the links. We will discuss these common fields in details below based on the official Keybase documentation [20].

All the information that is regarding a link type is encapsulated in **body** of Keybase statements. Body usually includes information about the link type, as it was mentioned in the link types above, the device that is creating the link and the information about the key that is signing the statement. One of the other fields includes information about the Keybase client that is creating the link which is encapsulated in the **client** field [20].

There are other important information regarding the time of the link creation and the expiry date of the statement. These are indicated in the **ctime** and **expire_in** fields respectively.

As we need to navigate the sigchain there is need for two main components to be included in all of the statements. These are **seqno** and **prev**. Each link in the sigchain has a sequence number that is include in the "seqno" field. If the sequence number was 0, the statement would be considered the first link in the user's sigchain. "prev" on the other hand, holds the hash of the previous link in the user's sigchain. This is required when a Keybase client is navigating from the last statement to the first statement in a user's sigchain [20]. Another field that is within all the statements is **merkle_root**. It encapsulates information regarding the time of the creation of the Merkle tree when the statement was created and it also includes the sequence number and the hash of the root of that Merkle tree [20].

2.4.4 Is Keybase a trusted server?

Our assumption is that Keybase is acting as an honest server. In the rest of this document, we treat Keybase as a trusted server. There are methods to check Keybase's honesty through manual checks, and if Keybase was not honest, it would become evident. Being honest means that Keybase has not deleted or removed any links that

users have posted on their sigchains and has not refused any request by a user to add new links to their sigchains.

Keybase users can, by using command line to interact with Keybase, check if Keybase has been honest by interacting with Bitcoin blockchain and performing manual decryption procedures and checks as follows: They can write their own scripts (for example, using Python) that check the expenditures on Bitcoin blockchain, open json files, compute and verify hashes, and connect to Keybase API. Assuming that some users perform these checks implies that if Keybase acts as a dishonest server, these users will be notified and alert others about the error in the Keybase server's integrity [15].

To enable users to verify that Keybase has remained honest, Keybase pushes the hash of its latest global Merkle tree's root to the Bitcoin blockchain (more details about Bitcoin blockchain and working with it can be found in Section 2.6). This Merkle tree has the most recent statement of the sigchains of Keybase users as leaves (each time that any user signs a link, the Merkle tree is updated and pushed to the Blockchain). The leaf nodes include the length of a user's sigchain, the hash of the payload of his latest statement, and the hash of his signature on the latest statement. The length is calculated based on the "seqno" (line 36 in listing 2.2) of the latest statement [15]. The hash of the payload is shown in line 12 of listing 2.1 and the signature on it is shown in line 20.

In a Merkle tree such as Figure 2.1, every leaf node is labeled with the hash of a data block (here the data blocks are the most recent statements of each user's sigchain), and every non-leaf node contains the hash of its child nodes. This means any changes to the tree (adding new statements to any sigchain) will affect the tree's root. Keybase's Merkle tree is like the tree in Figure 2.1, but each of the data blocks are for a single Keybase user and contain a full statement (user's latest statement). Keybase's Merkle root is regularly pushed into the Bitcoin blockchain. Whenever a

user adds a link to their sigchain, the Merkle tree is updated and the new Merkle root is calculated, signed by Keybase, and pushed to the blockchain. This means that any user can consult the blockchain to find the most recent Merkle tree root pushed to the blockchain in order to verify Keybase's honesty.

We use the following notations to explain how Keybase posts the root of its Merkle tree to the Bitcoin blockchain and how users can attempt to verify Keybase's honesty:

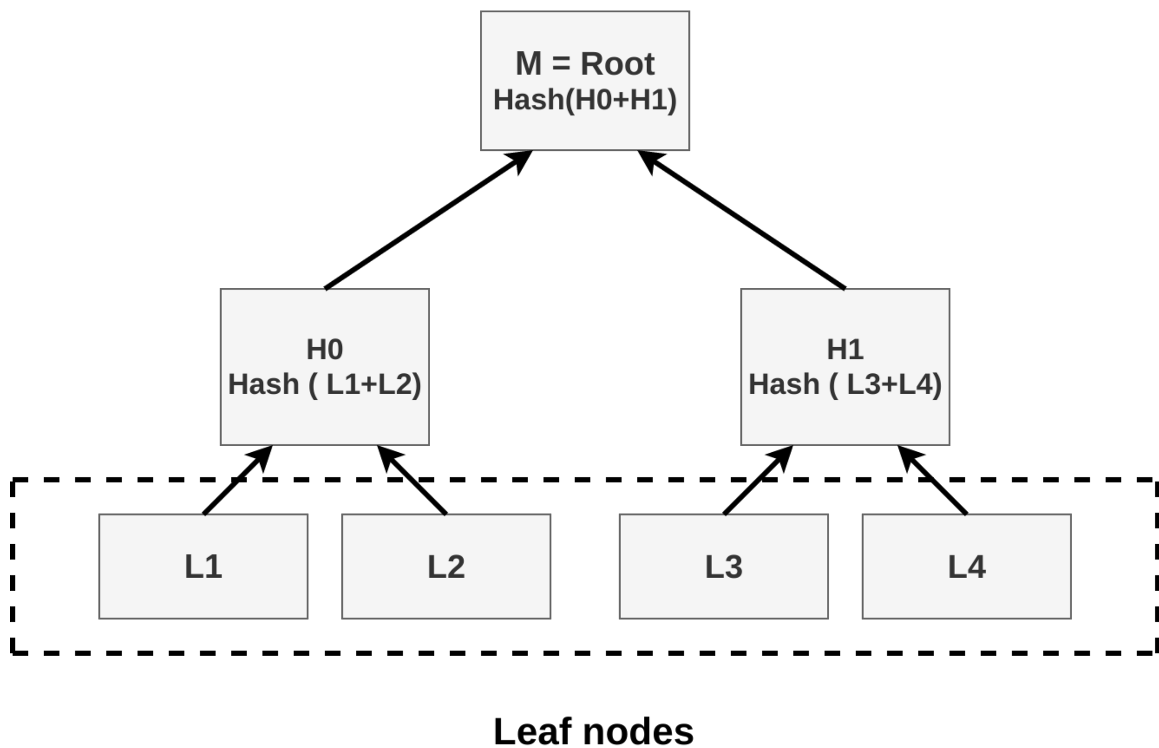

Figure 2.1: A sample Merkle tree structure. In Keybase's Merkle tree, the latest statements of the sigchains of each Keybase user are the leaves.

- H is the hash function
- S_k is signing with key k
- X is the value pushed to the bitcoin block chain

Here is the process in which Keybase pushes a hash to blockchain [15]:

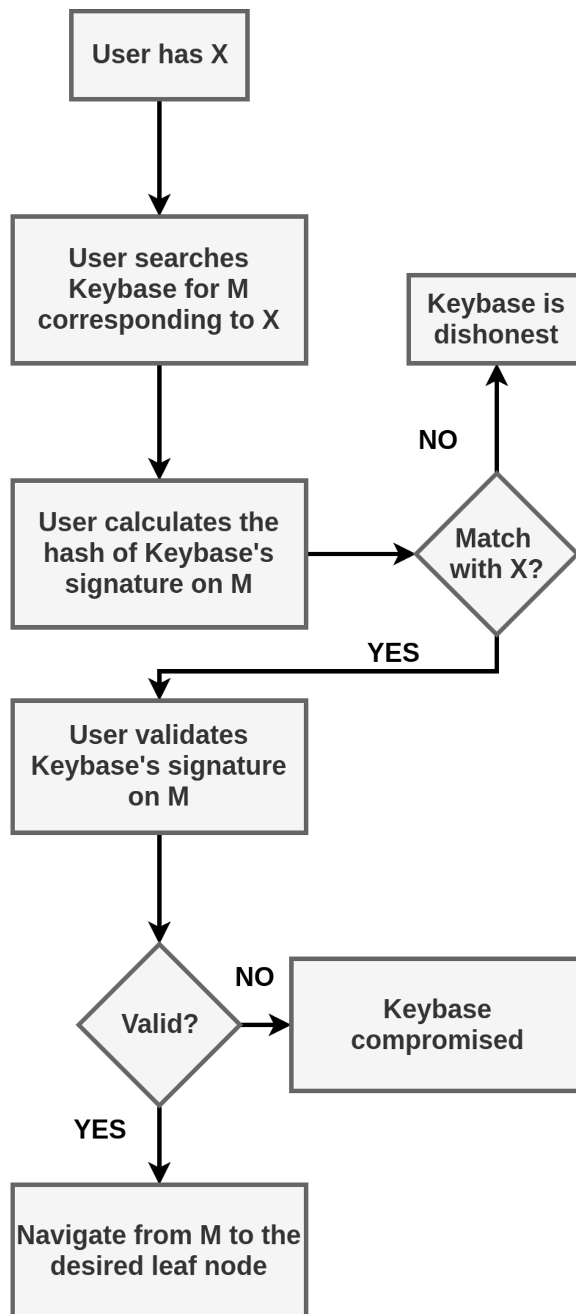

Figure 2.2: Process of validating a Merkle tree's root pushed to the Bitcoin blockchain. User is the person trying to verify the honesty of Keybase using various tools to interact with Bitcoin blockchain and to compute hashes and other data. $X=H(S_k(M))$ from Equation 2.2. The user's client recovered the value X in step 3 in Section 2.6.3 based on the process in Keybase documentation [15]

1. Calculate M, the hash at the root of the Merkle tree (top hash at Figure 2.1)

2. Sign M with k, where k is Keybase's private key and K is its public key

3. Compute the hash X of this signature and push it to the block chain

$$X = H(S_{\mathrm{k}}(M))) \tag{2.2}$$

In order to push each X to the blockchain, Keybase has to spend a small amount of bitcoin each time. Keybase sends that small amount to an address and that address is going to be used for verification purposes. From hereon we call Keybase the *sender* of the bitcoin, and X, the address of the receiver of that bitcoin [15].

For users to verify the validity of Keybase's root block (an example is showed in Section 2.6.3), users can find the latest expenditure from Keybase's address on bitcoin block chain. Then, they can get the receiver's address (X) and treat it as a hex-encoded hash value. X is calculated as shown in Equation 2.2. Users can search Keybase servers (through its API) for the root block corresponding with X, and Keybase will return the M (a sample root block is shown on Figure 2.1) and $S_{\mathrm{k}}(M)$. Then X the hash of Keybase's signature on M should match the value X retrieved from the blockchain. After verifying that, users can verify the signature $S_{\mathrm{k}}(M)$ on that root block (Keybase's public key is advertised on their website and the private part is kept offline [6]). This is a standard PGP signature created using Keybase's key k. If the signature is valid, users can navigate the Merkle tree from top to bottom to find the leaf node which is the latest statement of a particular Keybase user [15]. The process is shown in Figure 2.2.

```
1        {
2     "merkle_root": {
3         "ctime": 1526405127,
4         "hash": "83387161879afb03ab...3cbe8f96bbe7c0",
5         "hash_meta": "09a1a762f74c6d1bca5...0e0046ad6fc7",
6         "seqno": 2742702
7     }
```

[6] Available on https://keybase.io/docs/server_security/our_merkle_key

```
8 }
```

Listing 2.3: A sample Merkle tree root field in a sigchain including the hash value, the sequence number of the root block, the time of creation in ctime format, and meta data of the hash. This code block is part of Keybase API's response

Users can navigate from the latest link in a user's sigchain and move link by link to the oldest link on the sigchain to see if any links are missing, and thus the integrity of sigchain is verified. As mentioned before, each link has the signed hash value of its previous link which provides integrity while navigating the sigchain. This is because, while navigating, the Keybase client checks the hash in the "prev" field with the "payload_hash" of the previous link to see if it matches and it also verifies the signatures while doing so. The actual steps for performing these checks by users are demonstrated in Section 2.6.3 [15].

2.4.5 Keybase's goal

Keybase's goal is to make public keys available in a way that users can trust them without any back channel communication [23]. By this, we mean that one user should be able to obtain a trustworthy copy of another user's public key, and know it's the correct key, without the need for an out-of-band communication. This is a difficult service to provide, given servers might become compromised or make an attempt to lie about a key. Keybase client—whether it is one provided by Keybase itself or a client that has been developed by someone else — should not simply trust the validity of statements asserted by the server. When the reply from Keybase server indicates that "this public key is for Facebook user @r.samanfar", there should be a means to independently verify the validity of the statement. For this reason, Keybase was designed such that any cryptographic interaction with a Keybase user named Reza follows these steps [23]:

1. The server provides information regarding Reza. This includes: Keybase username, account public key (PGP public key), the account handles for bound social media, links to the actual proofs for bound social media (stored as posts on those social media accounts).

2. User's client proceeds to verify Reza's identity proofs (see Section 2.4.7.)

3. User performs a manual review of Reza's usernames (visiting Reza's social network profile).

So, first, the Keybase client queries the Keybase server regarding Reza's identity. Keybase server, provides a response with all its information about "Reza". The user's Keybase client does not simply trust the server's response. Keybase server has claimed that specified Keybase user Reza and *@r.samanfar* (for example his Facebook account handle) are actually the same entity. The question remains "is it true?" In step 2, the client perform his own checks. This can be done with the link that the server included in the response. For example in the case of Facebook, to convince the Keybase client, the post on Facebook must be a signed statement that is claimed to be from Reza on Keybase [23]. In other words, the Keybase client provides assurance to the user that "Reza" has access to 3 things:

1. His Keybase account

2. His Facebook account

3. The private key corresponding to the public key mentioned in the first step 1.

Now the user proceeds to review the verified usernames, i.e. to check if it is Reza, their intended recipient.

2.4.6 Following on Keybase

As mentioned in Section 2.4.5, in order to interact with a user, the first step is to ask Keybase server for some information about that particular user. This enables others to trust the Keybase server and be confident that advertised user is actually their intended user. While other users may switch between devices or add new devices to their trusted devices on their Keybase account, performing these steps every time may be overwhelming. "Following" is Keybase's solution to this problem. Following on Keybase means that the user's Keybase client takes a "snapshot" of another user's identity and signs that snapshot with the user's (follower) private key [23].

Taking a snapshot means that after going through those three steps, Keybase client signs the data with user's private key (specifically, the data from step 1 in Section 2.4.5, along with some extra information about the client's review of the other user such as the user that has taken the snapshot, the time of the snapshot, the time in which the identity proofs were posted on social media, and the hash of the link that has been posted) [23].

The client then posts the snapshot to the Keybase server. This does not mean that user's Keybase client will not perform further checks but, rather means that whenever a new device is used by the user, the Keybase server can provide user's own signed snapshot of the user that they have followed. User's own signature on that snapshot would verify the integrity of the snapshot itself [23].

If the user's Keybase client perform a check and the status of the intended user's sigchain did not match with the snapshot (i.e., the Facebook post on that user's social media page disappears), the client would notify the user of that change. Due to the fact that these snapshots are stored on the Keybase server and the contents of these snapshots are identical to each other (because the account hadn't changed, i.e., no revocation or no new proof addition or removal), users that want to follow

other users with a high number of followers can rest assured that the account that they are trying to follow has kept its integrity over a span of time due to the fact that those snapshots were taken at different times by different users. Keybase keeps the snapshots, and if they were not valid anymore, they would be removed by the following user's client [23].

It is important to keep in mind that this is not a web of trust. Web of trust is a decentralized trust model in PGP. In a web of trust users will trust another user based on decision of other users. We do not trust another user's keys or account status based on decision of other users. A new follower's client can verify Reza's identity proofs (see Section 2.4.7), even if no one else was following Reza. Although, as the number of Reza's followers grow, the confidence in the fact that the account has not been compromised over a span of time also grows. As mentioned above, an older follower statement is more valuable than a new one. It is hard for an adversary to maintain control of all of a user's accounts over a long period of time because the user himself or his friends would notice the compromise at some point [23].

2.4.7 Registering on Keybase

A user that wishes to register on Keybase is required to have a working email address and to have an account on one of the social media platforms that Keybase supports. Then, they should install the Keybase client software on their system (downloadable through Keybase's website).

Protocol (Detailed Steps for Registering on Keybase)

For a user to start the registration process on Keybase website, they should follow the steps (the steps are created by following the instructions appointed to the user while registering):

1. Visiting Keybase.io website and completing the Registration form on the website.

2. Downloading Keybase client software from the link provided on the website and installing it on their device.

3. Logging in the client software with the credentials that are used in the registration process.

4. The Keybase client software generates two public-private key pairs for signing and encryption. The user could upload their own public key through the client.

5. User will specify one of their social media account handles supported by Keybase. Keybase performs verification process on the specified account.

6. User will receive a challenge from Keybase (which the client signs using user's signing private key). The signed response needs to be posted on the previously-mentioned social media account.

7. User will be prompted to sign in to their social media account and post the signed response with a public visibility setting (the posting procedure may vary based on different social media platforms).

8. User will prompt Keybase client to verify the posted challenge on that social media (a sample is shown in Figure 2.3) and the client asks Keybase server to verify the challenge.

9. If Keybase server can find the post on the social media and the signature on that can be validated, the challenge is completed and that social media account will be added to user's Keybase profile page by Keybase server.

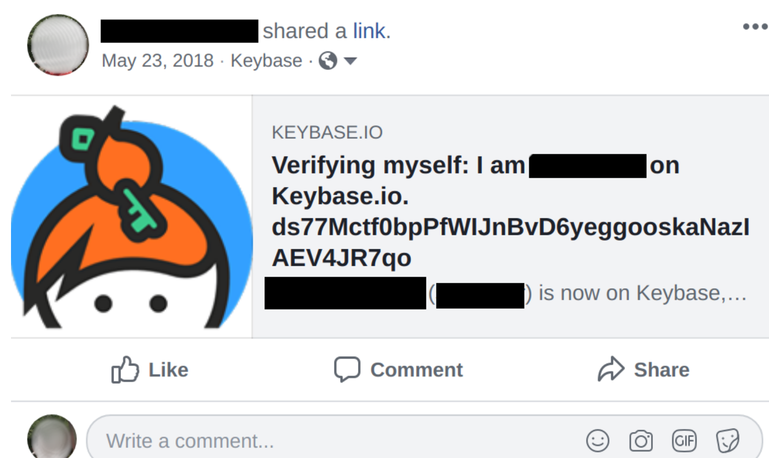

Figure 2.3: A sample posted Keybase proof on Facebook with public visibility setting. The name of the account and some parts regarding the identity of this post's owner have been blacked out.

Messages

In the messages shown in Figure 2.4, "Request" is the initial request that user sends to Keybase in order to indicate a specific social media account handle that they want to be associated with. "Challenge" is a challenge that Keybase server presents to user's Keybase client. "Signed challenge" is the challenge signed by Keybase client using user's signing keys and being posted on the social media account. If the post process was successful user lets Keybase know that the posting process is complete, sending a "Post complete" message. Verification of the challenge is the process where Keybase checks the social media for that specific signed challenge. If the post was found and Keybase verifies the signature on that is valid, it signals the user that the the account is verified and that account is added to their Keybase profile page.

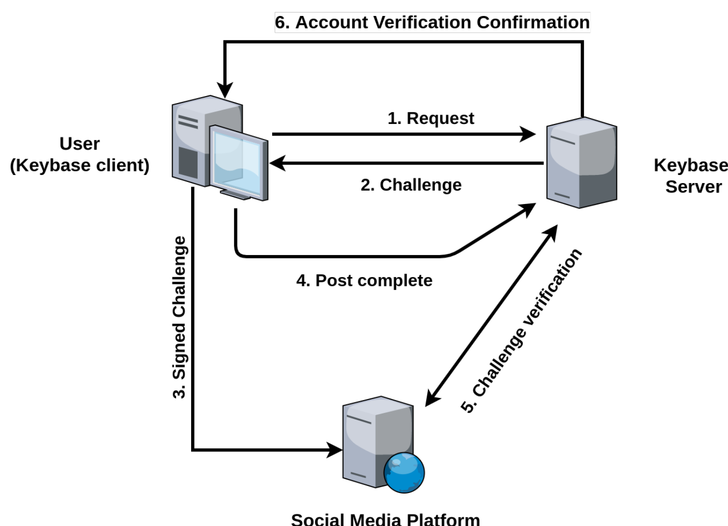

Figure 2.4: Social media account binding process. Social Media Platform will be involved in the validation process over Validation channel with ACME. Figure based on steps on page 32.

2.5 ACME protocol

2.5.1 Let's Encrypt

Let's Encrypt[7] has increased the adoption of HTTPS by providing an easy and fully automatic process for getting a trusted TLS certificate for web servers. These certificates are domain validated (DV) and are issued free of charge. Before Let's Encrypt, a typical user experience for acquiring a DV certificate for a particular web domain would be [4]:

1. Generating a PKCS#10 Certificate Signing Request (CSR) [42].

2. Copying the Certificate Signing Request to a CA web page

3. Proving the ownership of the domain - for which the DV certificate is requested- by any of these means:

[7] https://letsencrypt.org/

(a) Responding to a challenge from the CA which is typically a challenge placed on a specific directory of the web server.

(b) Putting a challenge posed by the CA in a DNS record of the specified domain.

(c) Receive a challenge posed by the CA via email (the email address should be domain administered and within the domain) and responding to CA's email.

4. Downloading the certificate after issuance and installing it on the web server.

Besides CSR part, the rest of the procedure is ad-hoc and it is based on human interactions with the CA that is performed differently from a published and implemented automated protocol. Informal usability tests conducted by authors of ACME suggests that webmasters often require 1-3 hour to obtain and install a certificate on their domains. However ACME protocol can change this situation [4].

2.5.2 ACME

ACME or "Automatic Certificate Management Environment" is a protocol that is currently being used to automate the interaction between CAs and web servers. It is based on the exchanging of JSON-formatted[8] messages over HTTPS. ACME has been standardized by IETF [4]. It has been designed by ISRG (Internet Security Research Group) and it was initially used by Let's Encrypt, a non-profit certificate authority run by the same group. The process of certificate issuance in ACME , i.e., in the case of a traditional CA that issues domain-validated certificates, is similar to a traditional CA [9]. User creates an account and requests a certificate for a specific domain (in our scenario it will be an email address instead of a domain). If the process completed

[8]JavaScript Object Notation

successfully (explained in detail in Section 4.1) a certificate for user's email address will be issued. After creating an account, the steps required to obtain a certificate are:

1. Submission of a certificate issuance order.
2. Proving control of the identifiers indicated in the certificate issuance order (email address).
3. Finalizing with submission of a CSR (Certificate Signing Request).
4. Awaiting issuance and downloading the certificate through ACME client.

Based on the identifier (the email address in our design), ACME poses challenges for users to prove ownership of the email address and the private key at the same time. ACME allows any user with an announced public key that has registered an account with ACME to request a certificate for any identifier, but will only grant the request after proper verification. In our design, ACME server produces a "nonce" and encrypts it with users public key and sends that to the indicated email address. The user, by accessing the email message and decrypting it, proves the ownership of the account, and by signing that challenge and sending it back to ACME server (by copying that message and sending it through ACME client software) provides evidence of ownership of the key (details of the design are in Chapter 4). ACME server verifies the signature on that nonce using users public key and, if it verifies, proceeds with certificate issuance. Since ACME is currently being used to issue DV certificates, there are various client software available to interact with an ACME server. Currently, none of those clients provide a graphical user interface and this may have a negative effect on usability of our proposed design. However, this can be addressed by adding GUI support to ACME clients.

All of the ACME functions are accomplished by the client software. As the ACME client acts as an HTTPS client, it sends a series of messages to the ACME server (acting as a HTTPS server) that are HTTPS requests and include a JSON message. While performing HTTP validations (process of validation over a HTTP connection where ACME server tries to validate a challenge posted on a web server) over validation channel, ACME server also acts as an HTTP server besides being a HTTPS server [4]. Requests that do not have a empty body, have their payload encapsulated within a JWS [9] [14]. This means that requests without this body format and simple GET requests will not be authenticated. Since the server has received a malformed request, it would return error 405 "Method Not Allowed" and would specify the type as "malformed" [4].

For protecting ACME server's resources from replay attacks, a mechanism has been put in place. As mentioned above, the ACME server produces and keeps track of its nonces. These nonces are used in all of the signed requests (mandatory) that are being transferred between the client and the server.

Since ACME protocol can be used to automate the process of issuing X.509 certificates, we claim that with minor changes on the existing client and server software (Pebble[10] which is a test ACME server provided by Let's Encrypt[11]), they can be modified to issue email certificates and perform verification. Our proposed method for doing so is discussed on Section 4.1.

2.6 Bitcoin background

We mentioned earlier that users can consult Bitcoin blockchain in order to verify that sigchains of Keybase users have maintained their integrity. These users can

[9] JSON Web Signature

[10] A miniature version of Boulder, Pebble is a small ACME test server

[11] https://github.com/letsencrypt/pebble

consult Bitcoin blockchain in order to retrieve the hash of Keybase's signed Merkle tree's root that has been pushed into the Bitcoin blockchain. From there, they can navigate through the Merkle tree and find the latest statement on sigchain of any particular user that they are trying to check. The steps for this process is shown in Figure 2.2.

In this section, Bitcoin is introduced and some sample code for performing aforementioned checks have been included.

2.6.1 What is Bitcoin?

Bitcoin is a decentralized digital currency and it is an example of a cryptocurrency. Cryptocurrencies inherit some characteristics of traditional currencies but they use mathematical proofs based on cryptography as a means of verification. One of the key features of Bitcoin is that it is not administered by a single entity and it does not rely on any central bank. The monetary units (called *bitcoin*) can be transferred from any user to another user within a peer-to-peer Bitcoin network. This eliminates the need for presence of a mediator [31] [2].

Bitcoin was proposed by a software developer named "Satoshi Nakamoto" in 2008. His goal was to introduce a payment system that became an electronic means of exchange independent from any central authority. bitcoins (the token) needed to be transferred in a secure manner and the transactions needed to be verifiable and unchangeable [7].

To avoid confusion, we clarify the differences between bitcoin (the token) and Bitcoin (the network and protocol). "bitcoin" is a binary object that represents ownership of a virtual unit of value. "Bitcoin" refers to a distributed network that acts as a registry that maintains records of all bitcoin balances.

2.6.2 Block chain

For maintaining a public distributed registry as mentioned in the Section 2.6.1, we need to have a distributed database that keeps track of all digital events and transactions that have occurred. This distributed database is *blockchain* and the events and transactions are shared among parties that participate in maintaining this database. Every transaction has a record that can be verified. In order to verify a transaction, consensus of a majority of the participating parties is required. Once a transaction record enters the blockchain, it will remain unchanged and can not be deleted afterwards. Despite the fact that Bitcoin is the most famous example that utilizes blockchain, its uses are not limited to cryptocurrency and financial applications [10].

The blockchain mechanism used by Bitcoin provides a digital system in which occurrence of a digital event can be recorded. All the participating parties can rely on the fact that the record can not be tampered with and will not be lost. Having this distributed system that is always based on the agreement of the majority of the participating parties, has helped many applications to publish their records to the blockchain in order to enable other users to verify the records and to provide transparency with their users. Users could trust in the records based on the fact that they are confirmed by the consensus of the digital entities that are in this distributed system [7, 30].

2.6.3 Interaction with Bitcoin blockchain for verification on Keybase

These are the steps taken in the verification process mentioned in Section 2.4.4 (the code that is shown below is in Python) and these codes are replicates from the code demonstrated in Keybase's official documentation on their website [15].

In order to follow the steps mentioned in Section 2.4.4 we need to find the address

that Keybase has sent a small amount of bitcoin to. We know the address that Keybase is sending the bitcoin from is[12]:

```
"1HUCBSJeHnkhzrVKVjaVmWg2QtZS1mdfaz"
```

In order to find the receiver's address, we would consult the "blockchain.info" and give Keybase's address as an argument as shown in line 14. We find the receiver's address as it would be the *receiver_address* in line 15 of the code listing 2.4. If we would print the value of the *receiver_address* at February 20, 2020 at 19:37 it would be : "17t2qANeNzJS9hzLoYER7Un6mz6RaPdFYq"

We then decode it with "base58" and convert it to a hex-encode hash value in line 16. Now we need to pass the "receiver_address_hash" to the Keybase API and request the matching root block of a Merkle tree as it is showing in line 23 of listing 2.4. We then receive the corresponding Merkle tree's root and put it in a variable in line 24.

When we receive the value for root, we look for the "sig" field that is the signature of the hash of the Merkle trees root block. This value should match with the value that we found in "receiver_address_hash". If they match we can be sure that the hash of that signature that Keybase is claiming to have is correct and is correctly published to the blockchain. After getting the root we can use a "username" to query the Keybase API to receive the user data corresponding to username specified in the line 31. In order to do so, we give the Keybase API the root hash and we get the actual root block as it is shown in line 36. Now we can start to descend from the root to the leaf node we have to keep in mind that the path from the root to a user's leaf node starts with the node that is indexed with the first character of that user's user id. The next node would be indexed with the first two characters of that user's user id and with this pattern we would reach the leaf node corresponding to that user as

[12]This address in mentioned in Keybase official documentation website and the expenditures are followed from https://www.blockchain.com/btc/address/1HUCBSJeHnkhzrVKVjaVmWg2QtZS1mdfaz

it is shown in the lines 41 to 49. If the last statement matches the last statement on our sigchain, we can have confidence that Keybase is pushing the sigchains to the blockchain with honesty.

```
1         # Finding the latest Keybase insertion
2       # into Bicoin blockchain
3       #------------------------------------------------------
4       import re         # Regular expression operations
5       import base58
6       from bs4 import BeautifulSoup
7       import json            # to enable working with json files
8       from base64 import b64decode    # for decoding a Base64 encoded
           string
9       from urllib import request      # to enable working with urls
10      from binascii import hexlify    # to convert binary data to hex-
           encoded
11      from hashlib import sha512, sha256
12      #------------------------------------------------------
13      Keybase_address = "1HUCBSJeHnkhzrVKVjaVmWg2QtZS1mdfaz"  #This is
            Keybase's wallet address
14      url = "https://blockchain.info/address/%s?format=json" % (
           Keybase_address)
15      receiver_address = json.load(request.urlopen(url))['txs'][0]['
           out'][0]['addr']
16      #The line above gives us the address that Keybase has sent the
           bitcoin to
17      receiver_address_hash= base58.b58decode_check(receiver_address).
           hex()
18      receiver_address_hash = '38482
           d2daf98ee6c04b2e2fd32981de6e78a3b60'
19      # receiver_address_hash = '38482
           d2daf98ee6c04b2e2fd32981de6e78a3b60' was used
20      # in the Keybase documentation for Monday 14 Jul 2014 as an
           example
21      #The line above converts it to a hex-encoded hash value
22      #----------------------------
23      keybase_api = "https://keybase.io/_/api/1.0"
24      url= "%s/merkle/root.json?hash160=%s" % (keybase_api,
           receiver_address_hash)
25      root = json.load(request.urlopen(url))
26      # looking for the root block matching with the value we found
           for receiver_address_hash
27      # looking for the 'hash160' value that should match the value we
            found in receiver_address_hash
28      #--------------------------------------------
29      username = "samanfar"
30      url = "%s/user/lookup.json?username=%s" % (keybase_api, username
           )
31      user_id= json.load(request.urlopen(url))['them']['id']
32      #user_id was "4a35d1e3eb7a1542babcad5035458719"
33      root_hash = 'c6f881691c7a5126d5334c8fd9af2d240025e34cd5c6618ac6
34      dae0737abad20354104df6c2c8ebbbc              4970
           aedf3a92bdb42d20441d639309a8f7d375c80f7bf09'
35      #root hash can be found when we receive the root on line 24
36      url = "%s/merkle/block.json?hash=%s" % (keybase_api, root_hash)
37      block = json.load(request.urlopen(url))
38      block = json.dumps(block) #converting block to string
39      #This gives us the root block of the cooresponding root hash
```

```
40      #with data regarding username="samanfar"
41      #---------------------------------------------
42      for i in range(1,len(user_id)):
43        temp = json.loads(block)
44        pfx = user_id[0:i]
45        next = temp.get(pfx)
46        if next == None:
47          break
48        url = "%s/merkle/block.json?hash=%s" % (keybase_api, next)
49        block = json.load(request.urlopen(url))['block']
50        data = temp[user_id][1]
51        print(data)
```

Listing 2.4: - Code is adopted from Keybase official website [15]

With these steps, a user can consult the Bitcoin blockchain and retrieve the aforementioned user data and check if the latest statements on sigchains of Keybase users are actually the same as the ones that are retrieved from Keybase server (ssumming that the value that Keybase wrote to blockchain was the correct value). This prevents Keybase from discarding new link statements to sigchains undetected due to the fact that the latest statement in a user's sigchain will not match the one that can be retrieved from Bitcoin blockchain.

In this section we have covered some fundamental concepts regarding the components that are being used in our proposed design in Chapter 4.

2.7 Related work

Lerner et al. [24] have pursued the same goals that we aim to achieve, making end-to-end encrypted email accessible for users by proposing a solution to address key management issue. They developed a prototype mail client that makes use of Keybase to enable users to send encrypted email. Users input their recipient's Keybase account handle (or search for them using their names or supported social media accounts) alongside their email address. Their client, called Confidante, then shows them their recipient's Keybase account within a drop down menu and users are required to manually verify the Keybase account of their recipients by clicking on them.

Our design differs from Confidante in three aspects:

1. We do not require a new mail client, but rather enable users to send end-to-end encrypted email to their recipients by configuring and using any mail client that supports X.509v3 certificates. Hence, both parties need not use the same mail client or a specific platform. For our test purposes, as mentioned in Chapter 4, we have used Microsoft Outlook mail client.

2. We use standard, self-signed X.509v3 certificates that have user's Keybase account handle embedded in the "Subject Alternate Name" field. This binds a user's public key, email address, and social identity all together.

3. We rely on a manual check of the recipient's Keybase profile by the user either on the first time that they either download the certificate/have received it from any other channel or while a certificate renewal happens (e.g., attachment from an email). Our users can save those certificates within their mail clients (assuming that the mail clients support saving certificates) and will not need to check their recipient's Keybase profile unless the certificate is detected by the mail client to be invalid either because of a change or a revocation.

Chapter 3

Threat model and requirements

In this chapter we describe the threat model and requirements. Our proposed design consists of two main parts. The first part is Keybase, which is being used to bind users' social identities with their public keys. The second part is ACME, that is being used to automate the certificate issuance which binds users' email addresses to their public keys. Our threat model consists of three main parts:

1. threats concerning interaction with Keybase.
2. threats concerning interaction with ACME .
3. threats that may arise while these two entities are interacting in our model.

The remainder of this chapter is as follows. Section 3.1 discusses Keybase's threat model, followed by ACME's threat model in Section 3.2. Section 3.3 presents our design's threat model.

3.1 Keybase threat model

Based on Keybase's documentations on their website, Keybase has identified and addressed some attack types that can threaten its integrity. Keybase documentation

states that DDos attacks targeting their servers ,and attacks that would target their server-side code resulting in the compromised server sending corrupted data to the legitimate Keybase clients could be vectors of attack against Keybase [21]. They also point out that a compromised Keybase client could be another threat. However, this could be achieved by corrupting Keybase servers to distribute the corrupted clients to the users [21].

We discuss each of these threats in the following section.

3.1.1 DDos attacks against Keybase servers

DDos attacks are a concern for many web servers and online service providers nowadays. In these types of attacks, attackers target web servers and exploit their limited connectivity resources to cripple their online services [12] [11]. As for Keybase servers, if the Keybase clients can not reach the Keybase server to retrieve the sigchains of the other Keybase users, verification process that is performed by the Keybase client and the validity of the statements in other user's sigchains can not be verified [21]. However, DDos attacks along with all other network-based attacks (such as DNS flooding, DNS hijacking and etc.) against Keybase server is out of our scope in this thesis.

3.1.2 Keybase server compromise

This kind of attack aims to prevent legitimate clients from retrieving sigchains of other honest users [21]. In this kind of attack if the server code and its signing keys are compromised, the corrupted server can manipulate the legitimate data or manufacture corrupted data. The legitimate Keybase clients that receive the corrupted data from the presumably legitimate Keybase server would perform the verification on false data [21]. This will be discussed in more details in Server-side threats, in Section

3.3.2.

3.1.3 Defence mechanisms for defending against server corruption

Keybase claims that a feature in Keybase's sigchain design and with the help of third parties that observe the status of Keybase's Merkle root pushed to the Bitcoin blockchain it can defend against server compromise threats.

In the design of sigchain, each of the statements have a sequence number. It is crucial to keep in mind that these sequence numbers should be incremented by one each time that there is a new statement being added to a specific sigchain and none of the statements will be removed from the sigchain (by design) at any point in time. In other words even for removing a key or a device the statement is not just simply removed from the sigchain but a revocation statement is added to the sigchain [21]. Also as discussed in Section 2.4.4, Keybase frequently updates its Merkle tree and pushes a value to the Bitcoin blockchain in order to enable third parties to perform validation and verification on the integrity of the Merkle tree that Keybase keeps. This implies that regardless of the source of the data that the Keybase client has acquired, as long as the data is signed using Keybase's signing key and the Merkle tree that its hash has been posted to the Bitcoin blockchain is identical to the acquired Merkle tree, the client can trust it [21].

3.2 ACME Threat Model

ACME, falls into the Internet threat model. Best current practices for the internet community are discussed in RFC 3552 [34]. Consulting RFC 3552, we use the following partitioning and divide them into three main categories: Confidentiality, Data Integrity, and Peer Entity Authentication. The ACME RFC (RFC 8555) also

follows these principles and addresses security issues in its security consideration section. These security issues will be discussed in this section exactly as they have been pointed out by McCarney [27].

While discussing ACME, the RFC 8555 has analyzed ACME server's communication with other hosts on the Internet in two different channels as shown below:

- An ACME channel: ACME HTTPS requests are exchanged within this channel.
- A validation channel: In this channel ACME server transfers requests in order to validate a client, and seeks to verify a client's control of a specific identifier (in this thesis, user's email address and Keybase account handle bound together).

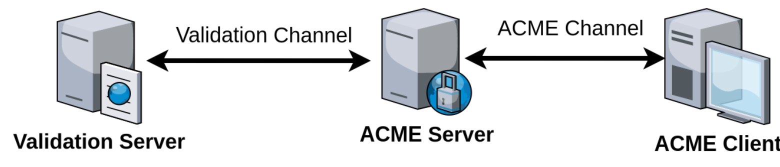

Figure 3.1: Communications Channels Used by ACME. ACME channel mostly consists of inbound HTTPS connections towards the ACME server, and the validation channel is outbound connections such as HTTP and DNS requests that involve demonstrating control of DNS-related resources for validation purposes. The picture is adopted from RFC 8555

Threats against ACME can target each of these channels. However, due to the differences between these channels the attacks would be different. ACME channel is used for communication between the ACME server and the ACME client. The communication on this channel is inbound HTTPS connections that the ACME client makes towards the ACME server [4]. On the other hand, the Validation channel is a channel in which the ACME server performs validation on client's control over a specified identifier. These communications are the HTTP connections made from the ACME server (outbound) or in some cases DNS requests [4]. It is noteworthy that security concerns are not always independent from the other channel. Although the

RFC 8555 claims that ACME's goal is to be secure about active and passive attackers on each of those channels but when an attacker can have control on both channels at the same time some concerns may arise [4].

3.2.1 Authorizations on ACME

Anyone that needs authorization from the ACME server can attempt to send an authorization request for an identifier of their choosing to ACME servers (in this thesis, an email address or Keybase account handle are both identifiers). They then receive challenges from ACME server that they need to complete in order to be authorized for the specified identifiers. This is done by account and key registration, followed by sending a "new-order" request with the registered account key [4]. ACME has put in place two challenges for authorization, and the intent is that these challenges can only be completed by someone who [4]:

1. is in possession of the private key that corresponds to the registered account key pair on the ACME server; and

2. has control over the specified identifier (i.e., an email address in this thesis)

There should be a bond between the validation responses and the registered account key pair in order to avoid attacks in which a middle-person tries to target ACME HTTPS requests (on validation channel) and advertise an account key of his choosing instead of legitimate user's key [27]. Such attacks can be mounted as follows [4] (Figure 3.2):

1. Legitimate user (legitimate owner of the identifier specified) registers a key pair (A) for the account.

2. Middle-person registers key pair (B) for an account.

3. Legitimate user sends a new order request that is signed using their key (A)

4. Middle-person intercepts the legitimate user's request and substitutes it with the same new-order request for user's identifier, signed using his account key (B).

5. Middle-person forwards challenges posed by ACME server to the legitimate user.

6. The user responds to the validation challenge.

7. ACME server proceeds to validate the response and verifies the response provided by the legitimate user as shown in Figure 3.1 (this can be checking a DNS record for a TLS web server, or some other methods).

8. Due to the fact that these challenges were regarding a response that was signed using account key (B), ACME server authorizes account key B (the middle-person) as the legitimate registered key instead of account key A (the legitimate owner of the identifier). This results in issuing a certificate for the middle-person's identifier with key B.

As noted in step 6, the user needs to provide a validation response to ACME server. For TLS web certificates, these could be posting a challenge on a specific DNS directory, or some alternative methods. This scenario would be different if the identifier was an email address and the response that the user provided required email owner's signature on it. As discussed in Chapter 5, this is one of the reasons that, in our proposed design, some attacks based on these methods are mitigated.

All of the challenges that are posed to the user in the standard ACME process, are validation queries made by the server that need to be signed with user's private key for proving key ownership and control over the specified identifier at the same

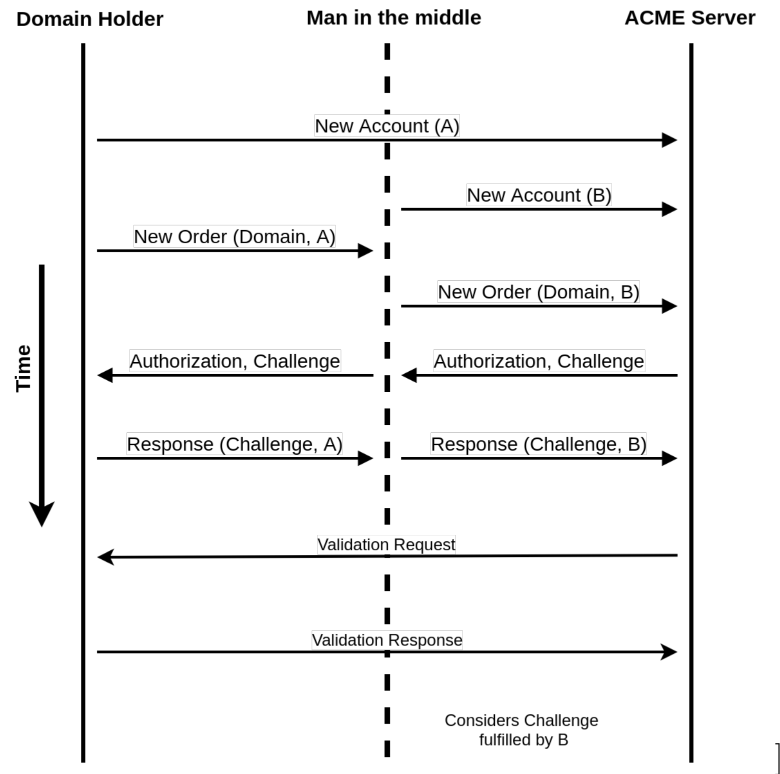

Figure 3.2: Middle-person Attack on validation channel without Account Key Binding showing the steps 1-8. Redrawn from RFC 8555.

time. Proving the ownership of the key and the identifier is done over validation channel with in validation response step. The main point of this challenge regarding identifiers is requiring the client to perform such an action that can be performed only by the legitimate owner of that identifier can perform. Some examples of these challenges are [27], [4]:

- HTTP: Putting some files under the ".well-known" directory[1] of a web server that host's the specified domain.
- DNS: Providing DNS resource records for the specified domain.
- Email: signing over a received challenge (by email) and sending that to the server (this may involve using ACME clients to send that response)

[1] /.well-known is a URI prefix in a website for well-known locations that is defined by IETF

Attacks are not the only source of violations in these assumptions. Misconfiguration (on servers) also can lead to a violation. As an example, a user can falsely “prove” ownership of a web server that has not been configured correctly. If the server allows other users (not only the administrators) to write entries to *.well-known* directories, any user can prove the ownership of the server, responding to the HTTP challenge that was mentioned above [27]. However, in case of email addresses as identifiers, this concern would not be applicable unless the user in any way gives the access to his email address to another entity so the entity can also access the contents of the messages sent to that email address in order to try to prove the ownership on that email address.

This issue is close to an issue with hosting providers. Those who use hosting providers are at risk because if the hosting server is compromised by an attacker, the attacker can request the ACME server for certificate issuance and they would succeed based on the fact that they have access to the hosted domain [4].

An attacker that is a middle-person on the ACME channel needs to also convince the validation server that he has control over a specific identifier. This can be attempted through validation channel and the requests are bound with the keys that are advertised to ACME server over key registration phase. Thus, it is not possible for that attacker to validate the identifier with the key that he has chosen. In other words, a passive middle-person on the validation channel, or any attacker (active or passive) performing middle-person on ACME channel, can not proceed to the certificate issuance since it can not pass the validation challenges with the key of his choosing [4]. A passive middle-person on the verification channel can only reply the responses that are being transmitted over the channel, provided that, by design, those responses are bound to the keys that are registered—they can not be used for any other purpose [4].

On the other hand, an active middle-person on the verification channel can pursue

normal ACME transactions and prove the ownership of the identifier with his own key as it was mentioned above regarding users that are using public hosting providers and has outsourced their DNS or web service operations. In order for a middle-person attack to succeed, the middle-person is required to act as an active attacker on both channels [27], [4].

RFC 8555 also mentions that another point that attackers can target is existing vulnerabilities in Internet routing protocols (e.g., RFC7132 discusses classes of threats against path security in routing) and attempting middle-person attacks over the validation channel. These kinds of attacks usually depend on the position of the attacker in the route to the server. As it is quite difficult for attackers to maintain control over various locations of the route, they are usually localized. These threats can be mitigated if the ACME server tries to perform DNS and HTTP queries through various routes, and any inconsistencies could result in detection [27], [4].

Since ACME is a protocol for managing certificates that performs key-identifier binding, the goal is to ensure that bindings are correct and only authorized entities are capable of managing certificates. This can be divided into two main objectives [4]:

1. Authorization over an identifier should only be possible by an entity that legitimately controls the identifier.

2. An account should never use another accounts key authorization

3.2.2 Denial-of-Service attacks

Denial-of-service (DoS) attacks are beyond our scope in this thesis. However, due to the fact that ACME protocols operate over HTTPS, rate limiting for TCP and HTTP-based connections should be put in place [4]. An example mentioned in the RFC 8555 is that an attacker may cause ACME server to send many outgoing connections (for the purpose of the validation) to a victim domain. This can be done by sending

ACME server repeated authorization requests for that victim domain which could cause the domain to suffer insufficient resources [4].

3.2.3 Request Forgery

As mentioned in Section 3.2.1, ACME server performs an HTTP request to domains for HTTP validation challenge purposes. This can be used by an attacker to force ACME server to perform an HTTP request to a domain that the attacker has specified. The validation process starts with a HTTP GET request from ACME server to the specified domain. Any ACME client is capable of making a query to a specific domain name using ACME server. It is known that some ACME implementations include some information gathered from the response from the validation server. This enables an adversary to get some information from his target web server even though that domain may have been inaccessible for him directly. This is called Server-Side Request Forgery (SSRF). The only limit to these kinds of attacks is that the attacker can only specify the target URL and has no control over the path that ACME server takes to perform the HTTP request [27], [4].

3.2.4 Certificate issuance policy

Checking if an entity has control over an identifier isn't the only check that is required before issuing a certificate for an identifier. There are various other checks that ACME server as a CA must perform before proceeding to certificate issuance [4]. Most of these additional checks mentioned in the RFC 8555 are regarding issuance of certificates for web domain. Some of the examples are checking if the clients have agreed to subscriber agreements or checking various constraints regarding the web domains such as the domain name including "*" in their leftmost label [4]. However, in our case, most of these check would not be applicable and simple checks such as

the user agreements, the identifier (email address) being valid, or not being from the domains that have been banned would simply suffice.

3.2.5 Replay protection

ACME servers make uses of nonces to protect against replay attacks. This is done by adding an obligatory nonce to all ACME POST requests and the signed responses from the clients must, by design, include that nonce within them. The list of used nonces are kept with the server which allows comparing the nonces within the requests with the ones that server has issued [4]. All of the JSON Web Signature (JWS) received from ACME clients should have the nonce in their protected header encoded according to the Base64 URL encoding[2] [14]. The method used for creating and comparing nonces is up to the ACME server, as we have our own method in Chapter **??**. ACME server transfers the nonces to the clients by an HTTP header field called "Replay-Nonce". This header field is required in both the successful and unsuccessful responses to POST requests [4].

3.3 Our Design's threat model

Both Keybase and ACME protocol have their own threat models and have aimed to eliminate various threats. The assumed threats for our design based on the threat model of its key components (Keybase and ACME) are categorized as follows:

(A) middle-person attacks

(B) server-side threats (both Keybase and ACME)

(C) client-side threats (both Keybase and ACME)

[2]It is described in section 2 of RFC 7515

(D) impersonation attacks

(E) adversary controlling user's accounts

For each of these categories, three main parts of our threat model mentioned in Chapter 3 will be considered. DDoS attacks against Keybase or ACME servers are beyond our scope. We denote threats with "T" and will address the effects of our proposed design on these threats in Chapter 5.

3.3.1 Middle-person attacks

T1: Middle-person attacks over ACME channel and the Validation Channel, shown in Figure 3.1.

T2: A middle-person attack over the communication channel between Keybase client and Keybase servers during initial registration with ACME and later communications.

3.3.2 Server-side threats

T3: An advanced attacker showing Alice's and Bob's clients different Keybase Merkle roots that are signed.

The attacker must fork the Merkle root and must always keep these forks. This fork can not be merged with the legitimate Merkle root since the inconsistency would allow detection by the users who communicate out-of-band and the duplicate roots would become obvious [21].

A server that has been compromised can either continue to be honest and avoid detection, or it could try to provide corrupt data that can be detected by users. A server that is controlled by an attacker can [21]:

1. Ignore updates to users' sigchains or try to rollback a specific user's sigchain.
2. Performing a fake key update and adding signatures to the end of a user's signature chain.
3. Maintain different site states of Keybase and providing these different versions to different users.

For checking Keybase server's honesty, Keybase clients are of a great importance. These clients always check the integrity of users' sigchains and, in the event of a rollback, these clients can detect the malicious act and prove Keybase server's dishonesty [21]. While Reza is following Alice, if there was a malicious roll back on Alice's sigchain or the integrity of Alice's sigchain was compromised (either by server compromise or Alice being compromised), Reza's client would alert him about the situation. The clients do this by constantly checking the Merkle tree root is being published by Keybase and crosschecking that with known sigchains. If the checks were completed, clients would proceed to sign proofs and these proofs would act as reference for other checks in the future. This means that Keybase clients play a crucial role in safety of the Keybase environment, and their integrity is critical [21].

Keybase has put in place several methods to increase the safety and integrity of its clients. The first one is that they have an open API and they refer to their open source client as a reference client. They have allowed software developers to create their own clients in the language of their choosing that can interact with their open API. Second, all the updates to Keybase client is signed by Keybase's private key and they claim that the key is kept offline to reduce the chance of key compromise. This allows users to not just trust the security of their download over HTTPS, but to trust the downloaded client based on the integrity of Keybase's signature on it [21]. It is noteworthy that users who are not using the Keybase client, and are instead opting for the Keybase web client, do not benefit from these assurances and are

vulnerable to server compromise. However, as mentioned before, if there were enough users that were using Keybase's clients, they could detect the server misbehaviour or compromise and alert other users [21].

Also on the ACME side, the compromise of the server would be one of threats. The adversary will try to compromise the server and try to issue an incorrect certificate with the public key of the adversary instead of the public key of the legitimate user.

T4: ACME server compromise resulting in issuance of false certificates, signing false values for their Merkle root, and sending a false root value to the Bitcoin blockchain.

T5: Keybase server compromise.

3.3.3 Client-side threats

Both Keybase and ACME have client software that must be installed on a user's device. Keybase does have an official client software, which is open-source with updates signed by Keybase. ACME does not have an official client software, but a few recommended unofficial ones. ACME does not take responsibility for those clients and recommends caution while downloading and using them to interact with an ACME server.

T6: User installing a malicious Keybase client.

T7: User installing a malicious ACME client.

3.3.4 Impersonation attacks

An adversary may try to impersonate a legitimate user by creating social media accounts that resemble the legitimate user's accounts, or attempt to impersonate the

legitimate user to either Keybase or ACME server.

T8: Attacker impersonating a legitimate user to interact with Keybase.

T9: Attacker impersonating a legitimate user to interact with ACME server.

T10: Attacker impersonating a legitimate user by creating fake social media accounts.

3.3.5 User account compromise

Adversaries may gain access to user's accounts by password compromise or other means. These adversaries may gain access to a user's email account, Keybase account, social media accounts, or all of these at the same time. Alternatively, they can try to infiltrate one of victim's devices and attempt to send and receive email from that device, or look for the private key stored locally. Further, they could try to add their own device to the list of the victim's devices on Keybase.

T11: Attacker gaining password access to a user's email address.

T12: Attacker gaining password access to a user's Keybase account.

T13: Attacker gaining password access to a user's social media accounts.

T14: Attacker gaining physical access to a user's devices either by possession or by using client-side malware.

In Chapter 5 we argue that these attacks are addressed by our design and the attacker needs to gain access to multiple components at the same time (email account, Keybase account, user's registered devices) to undermine the system.

Chapter 4

Design and Proof of Concept

In this chapter, we discuss the mechanism of our design. We describe the required actions and assumptions. It will be followed by a discussion on a proof of concept to demonstrate the feasibility of our proposed design; aiming to emphasize the changes that are required to enable an ACME server and an ACME client to interact with Keybase in order to issue email certificates for users.

Keybase interactions with the user are done through the Keybase client software. The user completes the aforementioned registration process with Keybase (Section 2.4.7) and puts two keys on their profile. One of them to be used in the encryption certificate, and the other one to be used in the signing certificate. For this proposed design, the functionality of Keybase client does not need to change except for enabling it to work with S/MIME instead of PGP as mention on page 21. The changes required to the ACME test server are located in Section 4.1.3. In Section 4.1.4, we proceed to discuss the changes required in an ACME client to interact with the test server. In Section 4.2, we demonstrate the process of two parties sending an email to each other. Figures 4.2 and 4.5 will be the main references for following the steps and to show all the interactions between the components of our proposed design.

The required actions for a user to take in the specific order are:

1. User needs to register on Keybase and perform the verification process on a

supported social medium.

2. User needs to interact with ACME server to obtain certificates.

3. User needs to upload their acquired certificates to their Keybase public folder to be accessible by other Keybase users.

4. User needs to download a recipient's certificates (for each party they wish to send or receive email from) from that recipient's Keybase profile (within their public folder) and manually verify that the social media accounts listed in the Keybase profile referred to by each certificate match the user's personal knowledge upon cross-checking these social media accounts of that recipient in real time.

5. User needs to integrate his own acquired certificates and his recipient's certificates with his mail client.

Three main assumptions should be taken into account:

1. Users should have a working email address connected through a secure channel.

2. Users should be active on at least one of the social media platforms supported by Keybase to enable Keybase to perform verification on that social media account.

3. Both key pairs used on Keybase and ACME should be the same key formats (e.g., S/MIME).

Any user that wants to send or receive email from another user securely (as shown in Figure 4.5), is required to have two clients installed on their system: a Keybase client, to register on Keybase and interact with it; and an ACME client, to interact with the ACME server in order to perform account registration, perform validation

process, acquire a certificate, and have certificate management (i.e., renewing an expired certificate).

4.1 ACME Server

The objective of the ACME protocol is to automate the process of obtaining and managing certificate with very little human intervention. This is done using the ACME client on user's device acting as a certificate management agent. As it was mentioned in Chapter 2, most ACME servers currently issue domain-validated (DV) web certificates. However, with a small change to the server and client-side code, they can issue X.509 email certificates as well. ACME server receives an email address instead of a valid domain. ACME server will identify the user with their email address. The first time the client software interacts with the ACME server, the client proceeds to create fresh pairs of public-private keys or lets users reuse the keys that they already have in the existing ACME server that is commercially used, one signing key pair is generated for DV certificate issuance but in our design we will discuss that we need two key pairs for signing and encryption. ACME client advertises the user's public keys to the server, and in the process the user is required to prove to the ACME server that it controls the email address and has the private keys corresponding to the advertised public keys. Since the user has previously generated two pair of keys for their Keybase account, it is crucial that they use the same keys in this process. This can be done by exporting the previously generated key pairs from Keybase client and importing them into the ACME client.

4.1.1 Protocol for interacting with ACME server

The protocol for interacting with ACME server is:

1. The client software sends a request to ACME server including the email address,

user's signing and encryption public keys, and the Keybase account handle of the user.

2. The ACME server receives the request, creates a nonce, encrypts the nonce with user's public encryption key and sends it to the specified email address.

3. User proves ownership of the email address by being able to access the email sent to that address.

4. User receives the encrypted nonce and copies it the ACME client.

5. The client software decrypts the nonce using user's private decryption key, signs the nonce with user's signing private key and sends it to the ACME server, which checks the signed nonce using the signature public key provided by the user (for one of the certificates using ACME client) to see if the challenge is satisfied. If the signature was valid, the ACME client checks if the public keys on the specified Keybase account handle are the same public keys that are presented by ACME client. If both public keys are identical, the ACME client is authorized to perform certificate management for that email address.

6. The ACME server then sees the key pair used as "authorized key pair", and any request signed with that private key is accepted as validly representing the corresponding user.

7. Two certificates will be issued by the ACME server (one for signature public key, and one for the encryption public key) and downloaded through the ACME client.

4.1.2 Challenges from ACME server

The challenge from ACME server has three main aspects.

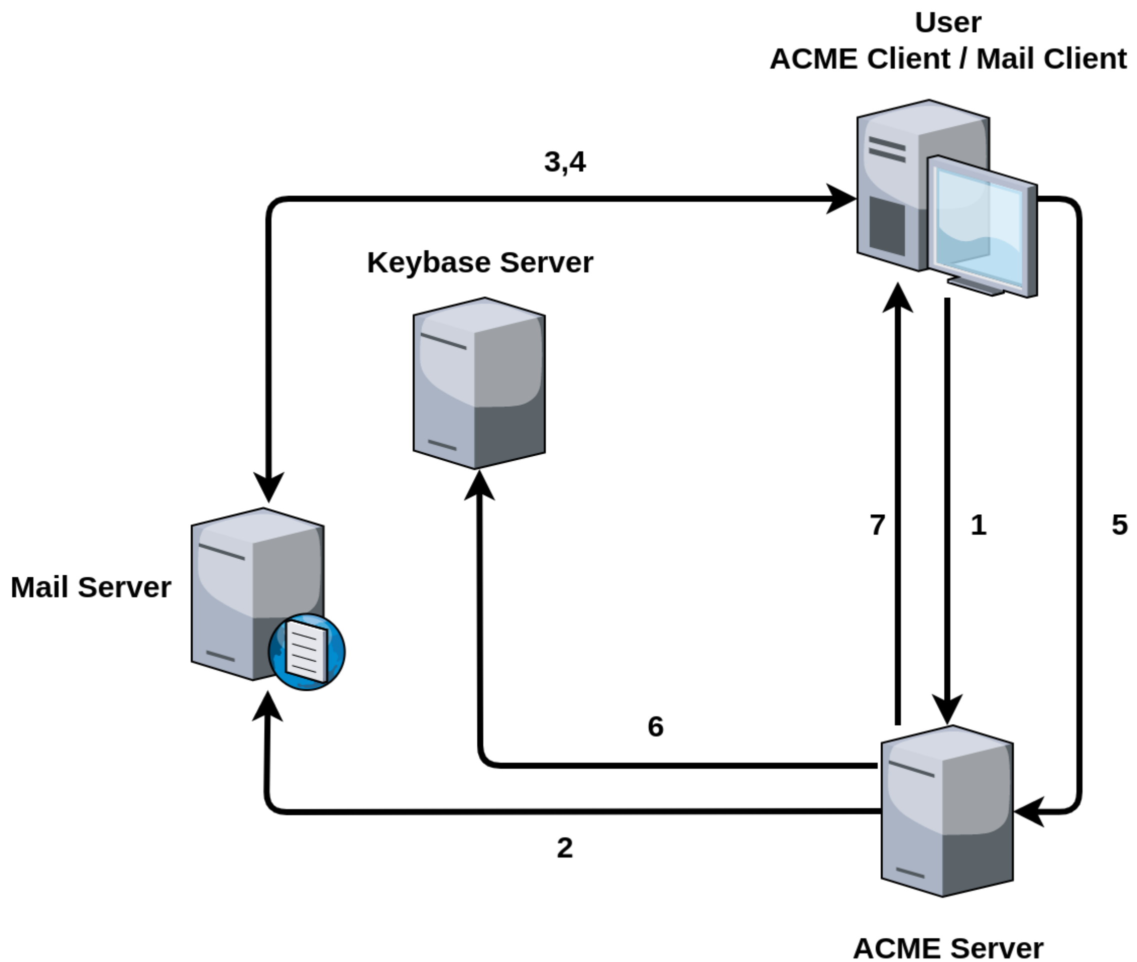

Figure 4.1: Interaction with ACME server

1. The challenge is proving the ownership over the specified email address, since the user needs to have access to the email account and open the mail sent by ACME .

2. It is proving the ownership of the private keys corresponding with the public keys that are presented to ACME server.

3. Since the public keys in the Keybase account and the public keys authorized by ACME server are identical, evidence of ownership of the Keybase account is provided as well.

Step 3 is based on Keybase registration process by which user has proven ownership of his public and private keys to Keybase. Also, due to the assumption that public keys are unique, no other user has the same key pair.

After a successful authorization process and completion of the challenges, certificate requests, renewing, and revoking certificates are done by the ACME client to issue certificate management messages and sign them with the authorized key pair.

4.1.3 ACME Test-Server

The test server is based on "Pebble"[1] which is a miniature version of the actual ACME server. Pebble is for testing purposes only. The code used for this implementation has been modified in Java by Tshepo Kgengwenyane [2] and myself. The changes required to make use of our proposed design, in an ACME server fit into two main categories:

- changes in validation process
- changes in certificate issuance process

Changes in validation process

Currently, Let's Encrypt ACME servers perform verification challenges using two methods [4]:

- **HTTP-01 challenge:** It is one of the most common methods in which, the ACME server provides a token to the ACME client. The ACME client then proceeds to put a file under ".well-known" directory of the user's website. The file has the token and a finger print of the user's account key. The ACME server then queries the web server to see the response [4].

[1] https://github.com/letsencrypt/pebble
[2] https://github.com/Poerilla/

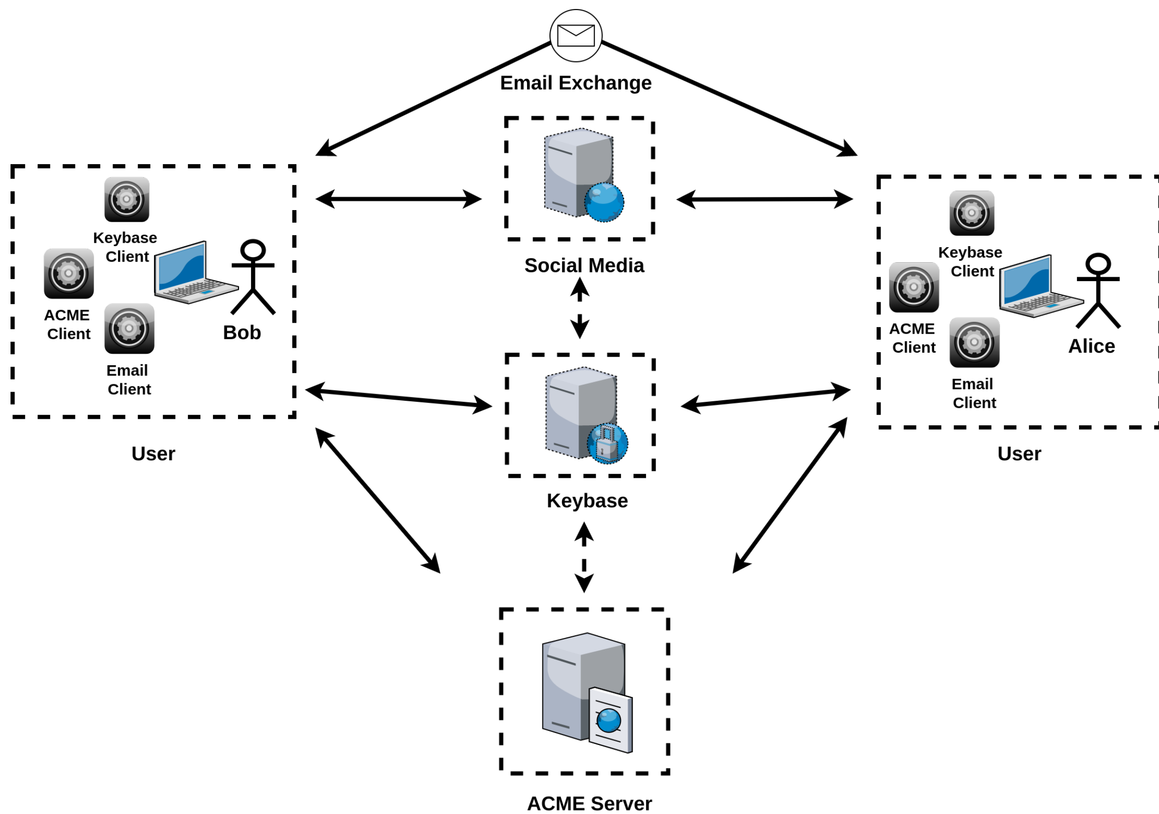

Figure 4.2: The components of our design. This picture is a copy of Figure 1.1

- **DNS-01 challenge:** In this challenge the user's ACME client receives a token from the ACME server and is required to create a text that is based on the received token and the finger print of the user's account key. The ACME client then puts it as a DNS record at "_acme-challenge.<DOMAIN_NAME> ". The ACME server then queries the DNS server to find the record [4].

In order to accomplish verification for email addresses, we introduce another method of authentication. In this authentication method, called "email-challenge", the ACME client needs to have other information about the user (instead of the domain name). The ACME server needs to receive the user's name, email address, and user's Keybase account handle in addition to the account key finger print (hash of the public keys that the user has given the ACME client in the registration process,

discussed in Section 4.1.4) that is used in the other methods as well.

We have developed a modified version of an ACME server for the users to create an account with, and have used "Gmail API"[3] to enable our ACME server to send and receive email to interact with the users that are signing up to our server. There is a simple interface web interface designed, as shown in Figure 4.3. Users input their email address, and are required to create a password for their account with our server. Users are also asked for their Keybase account handle to be added to their data. Their data then is saved in a database for future logins (to their ACME account) by the users. This process can be integrated with the ACME client, in which the user can complete the sign up process and input the required data within the ACME client. But in our tests, we have used a web interface to demonstrate a user's sign up process, and we have populated the database manually for test purposes, as shown in the Listing 4.1. The server needs to interact with an ACME client that supports the new verification method (email-challenge). The changes required for the ACME client are discussed in Section 4.1.4.

In the code block, we have put the email address as the *value* in line 11 that will be our "Subject Name". The following lines (from 16 to 45) are the information about the issuer of the certificate (our ACME server) that we have provided for our own test purposes. We note that lines 5, 19, 25, 31, 38, 45 are hard-coded, whereas in a functioning prototype, these would be inputs from another source.

Sign Up

Please fill in this form to create an account.

Username Enter Your Email Address **Create New Password** Create New Password **Repeat Password** Repeat New Password **Keybase Account Handle** Enter Your Keybase Accoun ☑ **Remember me**

By creating an account you agree to our Terms & Privacy.

Cancel Sign Up

Figure 4.3: A sample sign up page for our ACME server.

[3] https://developers.google.com/gmail/api

```
//Subject Name
    certificate.subject.typesAndValues.push(
      new pkijs.AttributeTypeAndValue({
        type: "2.5.4.6", // Country name
        value: new asn1js.PrintableString({ value: "CA" })
      })
    );
    certificate.subject.typesAndValues.push(
      new pkijs.AttributeTypeAndValue({
        type: "2.5.4.3", // Common name
        value: new asn1js.PrintableString({ value: userData.email })
      })
    );

    // Issuer
    certificate.issuer.typesAndValues.push(
      new pkijs.AttributeTypeAndValue({
        type: "2.5.4.6", // Country name
        value: new asn1js.PrintableString({ value: "CA" })
      })
    );
    certificate.issuer.typesAndValues.push(
      new pkijs.AttributeTypeAndValue({
        type: "2.5.4.3", // Common name
        value: new asn1js.PrintableString({ value: "CCSL" })
      })
    );
    certificate.issuer.typesAndValues.push(
      new pkijs.AttributeTypeAndValue({
        type: "2.5.8.1", // State
        value: new asn1js.PrintableString({ value: "ON" })
      })
    );
    certificate.issuer.typesAndValues.push(
      new pkijs.AttributeTypeAndValue({
        type: "2.5.4.10", // Organization
        value: new asn1js.PrintableString({
          value: "Carleton Computer Security Lab"
        })
      })
    );
    certificate.issuer.typesAndValues.push(
      new pkijs.AttributeTypeAndValue({
        type: "2.5.4.7", // Locality
        value: new asn1js.PrintableString({ value: "Ottawa South" })
      })
    );
```

Listing 4.1: Adding the required fields for a certificate for our ACME server with our own hard-coded data.

After creating an account and providing an email address, users are required to provide the signing and encryption public keys that they have on their Keybase profiles. This can be done through the ACME client by which users can advertise

their public keys (encryption and signing keys) to be associated with their ACME account. However, in our tests, key advertisement has been performed manually due to the difficulties mentioned in Chapter 4, page 60. After signing up, users then will need to prove the ownership of their email address. There are plenty of ways to proceed with the verification.

However, in our design, we have made use of the basic principle of the verification process in the two challenge types mentioned above. The ACME server generates a random nonce as a token and uses user's public encryption key to encrypt that nonce. Then, the ACME server sends it to the user's email address. The user checks their email address and copies the encrypted nonce included in the email's body to their ACME client. Then the ACME client decrypts the nonce locally using user's private encryption key and signs the nonce with user's private signing key on the local machine and sends that string to the ACME server. For this connection, ACME server acts as a HTTPS server and the ACME client acts as a HTTPS client. The ACME server checks the received string. If the signed string matches the token and the signature on that verifies, this validates to the ACME server the email address based on the fact that the user had access to that email address to receive an email from that ACME server and, and also confirms the ownership of the private encryption key used to decrypt the nonce and the private signing key, used to sign the message.

Now that the ownership of the email account has been verified, the ACME client needs to verify if the keys that were advertised by the user in the ACME sign up process are identical to the keys that are advertised on the Keybase account handle mentioned by the user in the ACME registration phase. After verification of user's email address and key pairs, the ACME server makes a query through Keybase's API, specifying the account handle that the user has advertised. The Keybase API responds with the sigchain of the user that the ACME account has queried. The ACME client receives the response from the Keybase API and parses the text in

order to find the account handle. If the account handle in the response is identical to the account handle that the ACME server has queried, it means that the ACME server has received the right sigchain. The account handle can be found in line 18 in Listing 4.2. After parsing, since a user may have many keys on their Keybase account, the ACME client makes a list of all the finger prints of the public keys that are on the sigchain as one of them is shown in the code block in Listing 4.2 in line 25. The code block in the listing is one of the statements (links) of a Keybase account's sigchain in a JSON format as it is shown below:

```
1  {
2          "seqno":8,
3          "payload_hash":"b6ba46a2519a697128db3271e70d71ba87dbf4dd
4            7df02e2d0deb8b12e0ba411b",
5          "merkle_seqno":2743121,
6          "sig_id":"225e4c52414d207ad3a0e16c0336a351db0e61e77fa649
7            107366a24ab1595ce60f",
8          "sig_id_short":"Il5MUkFNIHrToOFsAzajUdsOYed_pkkQc2ai",
9          "kid":"012055ab14b5bdf7e527fcf53683add21d8fa9aa2ead5cc18
10           6f8858ea886954a2a0e0a",
11         "sig":"hKRib2RZNjODM9hZFxuNakJtT1RKa05qTm........1aVE13W
12           W1Sak1qY3dNbUpqWlRZMlkyTm1OV05tTZXJzaW9uAQ==",
13         "payload_json":
14         "{\"body\":
15           {\"key\":
16               {\"eldest_kid\":
17                   \"0120f92d63fe30bdc2702bce66ccf5cf5180b850af8e12e5
18                           0620c5b0944ffa8a5e9e0a\",
19                   \"host\":\"keybase.io\",
20                   \"kid\":\"012055ab14b5bdf7e527fcf53683add21d8fa9aa2
21                    ead5cc186f8858ea886954a2a0e0a\",
22                   \"uid\":\"4a35d1e3eb7a1542babcad5035458719\",
23                   \"username\":\"samanfar\"},
24                   \"merkle_root\":
25                   {\"ctime\":1526406245,
26                   \"hash\":\"ceac80890c5ea850dd83ed361fca192739865ae
27                       954ebd805f6dee95cd7078e35ad90392caf9a9af8b6cd4
28                       15c086e071bbf2a89cae0ed9e7b2334ac610278fd4f\",
29                   \"hash_meta\":\"b9c09fa304b6612ae20a34c70e067a6c34
30                       15ef3962601d96b63c860eef392b62\",
31                   \"seqno\":2743119},
32                   \"sibkey\":{\"fingerprint\":\"
                        c19f0091eec033eb7190f9ee17b0cb2be05f4e18\",
33                   \"full_hash\":\"d92ef85ed123fd34ddc5577e71d392f0c9
34                       57c30508ca6a97a5a394151286f18d\",
35                   \"key_id\":\"17B0CB2BE05F4E18\",
36                   \"kid\":\"01014a029a0645ada750aa8d3e4e3b7d7a1bbbe2
37                       1ac2b6ed60dac1c2951ffd70a8c20a\",
```

```
38                  \"reverse_sig\":\"-----BEGIN PGP MESSAGE-----
                       YjhlYzd5fzCyxiAtkQ4GU0SSK........EiPY8+
                       yqMTw7VWkjh26PoowgkK0I2hLikMjE02CIo=\\n=URSE\\n
                       -----END PGP MESSAGE-----\\n\"},
39                  \"type\":\"sibkey\",\"version\":1},\"client\":{\"
                       name\":\"keybase.io go client\",\"version
                       \":\"1.0.48\"},\"ctime\":1526406282,\"expire_in
                       \":504576000,
40                  \"prev\":\"5077394b3669c86aa5034b8ec7ed3aefd55becb
41                     3241121d4dcb723890f35c987\",
42                  \"seqno\":8,
43                  \"tag\":\"signature\"}",
44
45        "sig_type":1,
46        "sig_version":1,
47        "ctime":1526406282,
48        "etime":2030982282,
49        "rtime":null,
50        "eldest_seqno":1,
51        "sig_status":0,
52        "prev":"5077394
              b3669c86aa5034b8ec7ed3aefd55becb3241121d4dcb723890f35c987",
53        "proof_id":null,
54        "version":null,
55        "is_eldest":0,
56        "fingerprint":"",
57    }
```

Listing 4.2: Part of the response from the Keybase API that has the statement in which a user has added a PGP key to their account

The server finds all the “key_ids” (a key_id is the finger print of the key shown in line 25). If the key fingerprints of both keys that the user has advertised to the ACME server in the ACME registration phase are found in the list of the key fingerprints from user’s sigchain, the ACME server can verify that the user has ownership of the Keybase account based on three assumptions. First, the user has proven the ownership of the private keys previously in the ACME registration process. Second, the user has proven the ownership of those keys to Keybase while interacting with Keybase. Third, the assumption is that the user is the only person with access to those private keys and he hasn’t shared them with any third party.

We have used the *skip-validation* option of the Pebble server which is enabled by using the code below in the Pebble server [18]:

```
PEBBLE_VA_ALWAYS_VALID=1
```

This is because our ACME server doesn't accept PGP keys as mentioned in Chapter 2 on Page 21. In order to simplify our testing, we have skipped the validation and put in the validated keys manually in our ACME server. Although this did not allow us to fully run a test on ACME and Keybase API interaction but assuming that the keys on the Keybase account and the keys presented to the ACME server on the ACME registration phase are the same and the verification process has been successful helped us to proceed with the remainder of the test and issuing a sample certificate for an email address since, this is a functionality test and it is not a security test.

Changes in certificate issuance process

ACME Pebble is designed to issue TLS domain validated certificates. However, in order to enable it to issue email certificates we have changed some of the parameters required to issue a certificate.

- putting a user's email address in the "subject name" field of the certificate.
- adding the user's Keybase account handle to the "Subject Alternate Name" field of the certificate.

After the verification process and based on the manually-entered data as a user's information, the ACME server proceeds to issue a certificate with the aforementioned conditions and proceeds to sign the certificate with its private signing key. The server certificate and private key were generated using "OpenSSH"[4]. As you may have seen in Figure 4.4, we have acquired a certificate for our server and we assume that our server's public key is recognized by the relying parties email client software. As discussed in Chapter 4, email client users should know the public key of the ACME server issuing the email certificate that others must rely on. This can be accomplished

[4]https://www.openssh.com/

either by advertising this key through the ACME certificate server's website or by the ACME server becoming a trust anchor that mail clients would add to their databases.

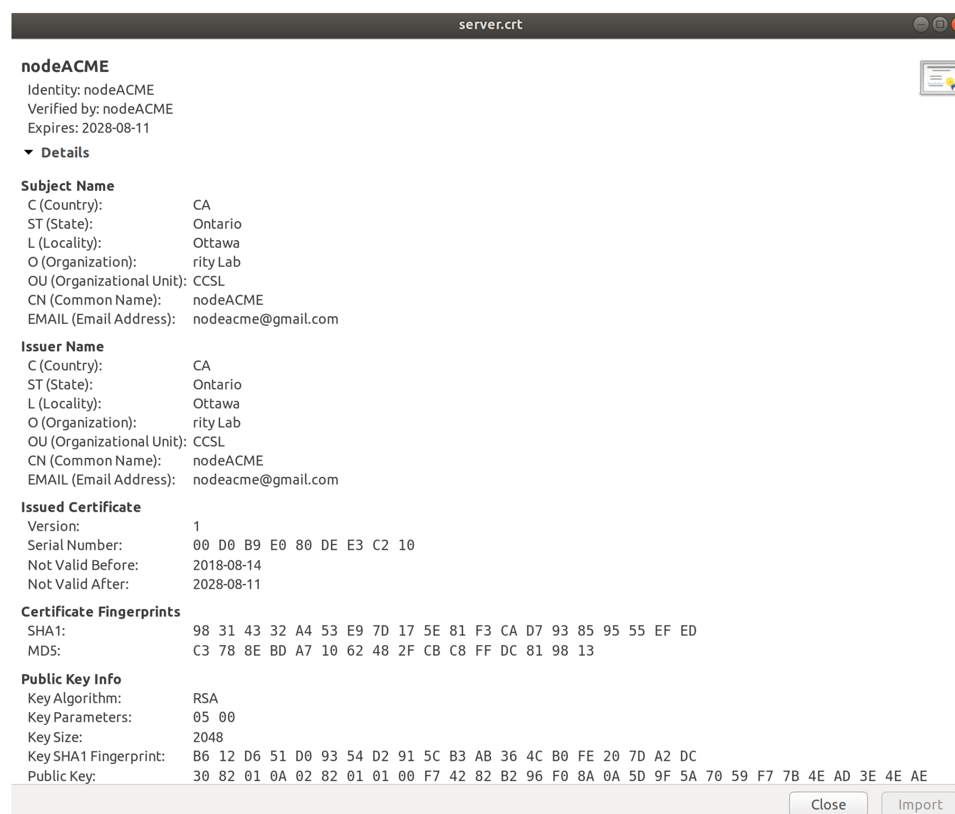

Figure 4.4: A self-signed certificate generated by OpenSSH for the ACME server

4.1.4 Required changes for the ACME client

In order to interact with the ACME server, ACME clients should be programmed to use corresponding verification methods. As we have changed the verification method in our ACME server, there are a few changes that are needed from the client-side. There are many clients that are recommended on Let's Encrypt's website but there is no official client endorsed by Let's Encrypt. It is mentioned on their website that: "*The ACME clients below are offered by third parties. Let's Encrypt does not control or review third party clients and cannot make any guarantees about their safety or*

reliability."[5]. In our tests we have not modified any ACME client, and we have manually input the data (hard-coded) to the ACME server in order to pursue the verification process by the server as mentioned above. However, the changes required to an ACME client are:

- changes in validation process
- changes in user input
- changes in interacting with users system

As an example, for clients that are written in Python, there is a python library that has implemented a class called, ACME validation challenge class [25]. This class is the implementation of the challenges accepted by ACME in RFC 8555 [4], as it is shown below:

```
class acme.challenges.DNS01(**arguments)
```

This is for a DNS01 challenge mentioned in Section 4.1.3 which is defined in the "acme-python" documentation [25]. To change an ACME client to work with our proposed design, a new email-challenge should be added to the acme-python library if the preferred language was Python. Then, the client can be modified as we mentioned in Section 4.1.3 to accept this kind of challenge instead of HTTP and DNS challenges.

There is a need to modify the user input in the ACME client. The ACME client should require a user to input other information such as their email address, Keybase account handle. These would be sent to the ACME server in order to proceed with the certificate issuance process.

Due to the fact that most of the ACME clients are developed to acquire a TLS DV certificate for websites, they are required to interact with web service software (i.e.,

[5]https://letsencrypt.org/docs/client-options/

Apache) on the user's machine. These requirements should be removed and instead, these clients should present the challenges generated by the ACME server to the user and the user can copy the output to the email that should be sent to the ACME server (mentioned in Section 4.1.3).

Other certificate management functionalities of the ACME client (e.g., renewing the expired certificate) should change due to the fact that the ACME client should notify the user about the new certificates so the user can upload the new ones in their Keybase file system. We have not made these changes as part of the work described in this thesis.

4.2 Sending Encrypted Email Between Two Parties

Assuming that user has a valid account on Keybase and has obtained two email certificates via ACME (one for encryption and one for signing), we now explain the steps involved for an end-to-end encrypted email exchange between the user and another user with the same conditions. Two parties are named Alice and Bob, and Alice intends to send Bob an email. Another assumption is that Alice can identify Bob's social media accounts on Bob's Keybase profile and verify that the owner of those social media accounts is Bob (this is what we mean by "manual verification"). For this scenario to work, the sender (Alice) needs to have receiver's (Bob) certificate in order to encrypt email with Bob's public key.

Keybase is acting as a key directory that doesn't require users to trust in the directory itself (described in Section 2). By using Keybase file system (KBFS), users can host their certificates on their Keybase profile. All Keybase users have a public and private folder on their account which can be used to upload files for public use

in their public folder, or for private use in their private folder. Users can access other Keybase users' public folder from their Keybase client or through Keybase website. A user must either know the other user's Keybase account *handle* or one of their supported social media usernames to be able to search for their Keybase profile and visit their public folders, either from their Keybase clients or through the website. Through the Keybase design by which all of the files on every user's KBFS is signed by users private key, if Alice is downloading Bob's certificates from Bob's KBFS, she has some assurance that the certificates are signed and uploaded by Bob himself. This is because all uploads to KBFS are signed on the Keybase client side. Alice's Keybase client is designed to check Bob's signature based on Bob's public key on his profile, while trying to download his certificates. In this scenario, we assume that both Alice and Bob have uploaded their email certificates on their Keybase profile under their public folders. The steps for Alice to send Bob an encrypted and signed email are shown below and in Figure 4.5

1. Alice registers an account on Keybase server. She proves ownership of at least one of her social media accounts to Keybase and creates two key pairs on her Keybase profile.

2. Alice Registers an account on ACME server and proceeds to acquire two certificates from ACME server (for signing and encryption purposes).

3. Alice's ACME client downloads the issued certificates from ACME server.

4. Alice uploads her certificates to her public folder on her Keybase profile (taking advantage of Keybase Filesystem mentioned on page 13). It is noteworthy that steps 1-4 should be performed by Bob as well in the same order. However, they are not illustrated in the figure.

5. Alice looks for Bob's Keybase profile on her Keybase client or on Keybase

website, either by searching for Bob's Keybase account handle or by searching Bob's known social media account handles that are bound with his Keybase account.

6. Alice proceeds to download Bob's email certificates (one for encryption and one for signing) from his Keybase public folder using her Keybase client or through the website. She saves the certificates within her mail client's saved certificates (the procedure of saving may vary through different mail clients but the client should have the ACME server as one of its trust anchors).

7. She then proceeds to use her local email client to send a signed and encrypted email to Bob. The email client will attach, to the email, Alice's own email public key certificates.

8. Bob receives Alice's email. His client finds Alice's certificates attached. The mail client checks the validity of the CA's (ACME server's) signature on the certificates. This relies on the mail client having the ACME CA public key as a trust anchor. If the certificates are valid, the mail client proceeds to decrypt the email. Bob needs to open Alice's attached email certificates within his mail client (Bob will need Alice's encryption certificate for sending her an encrypted email later). Bob should find Alice's Keybase account handle in Subject Alternative Name field. Bob proceeds to copy the Keybase account handle he found within the certificates into a browser address bar or a Keybase client and visit it. If Bob visits the profile and can identify the account owner Alice knowing her social media accounts, he can gain confidence that this email is originated from Alice herself. Then, he adds Alice's certificates to his mail client's saved certificates.

9. Bob can encrypt a message using Alice's encryption public key, sign the message

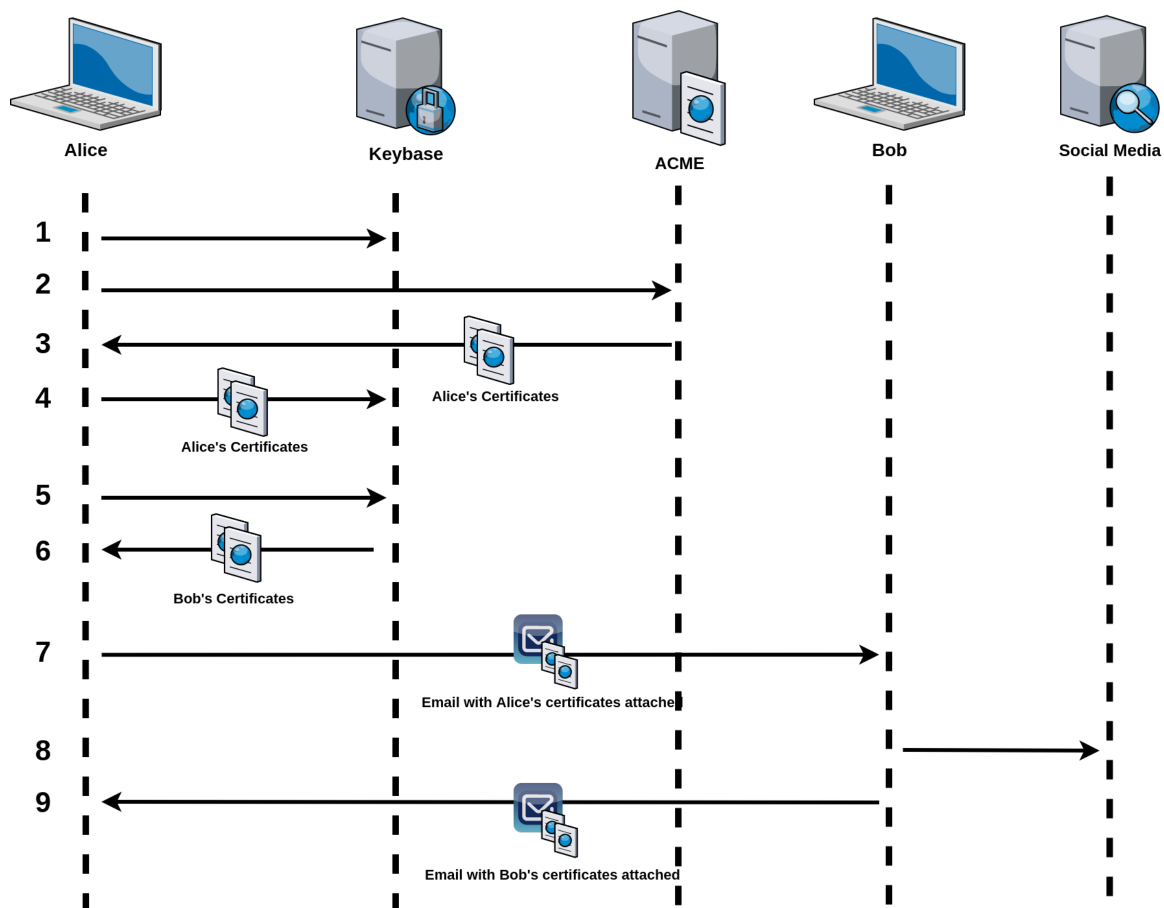

Figure 4.5: Mail exchange between Alice and Bob explaining steps above

using his own signature private key, and send the encrypted, signed message to Alice's email address. Alice will receive a signed and encrypted email from Bob, and since she had previously saved Bob's certificates, the mail client verifies the signature and decrypts the email (we haven't tested an email exchange on a specific mail client but, most mail clients such as Microsoft Outlook have the ability to perform these procedures already).

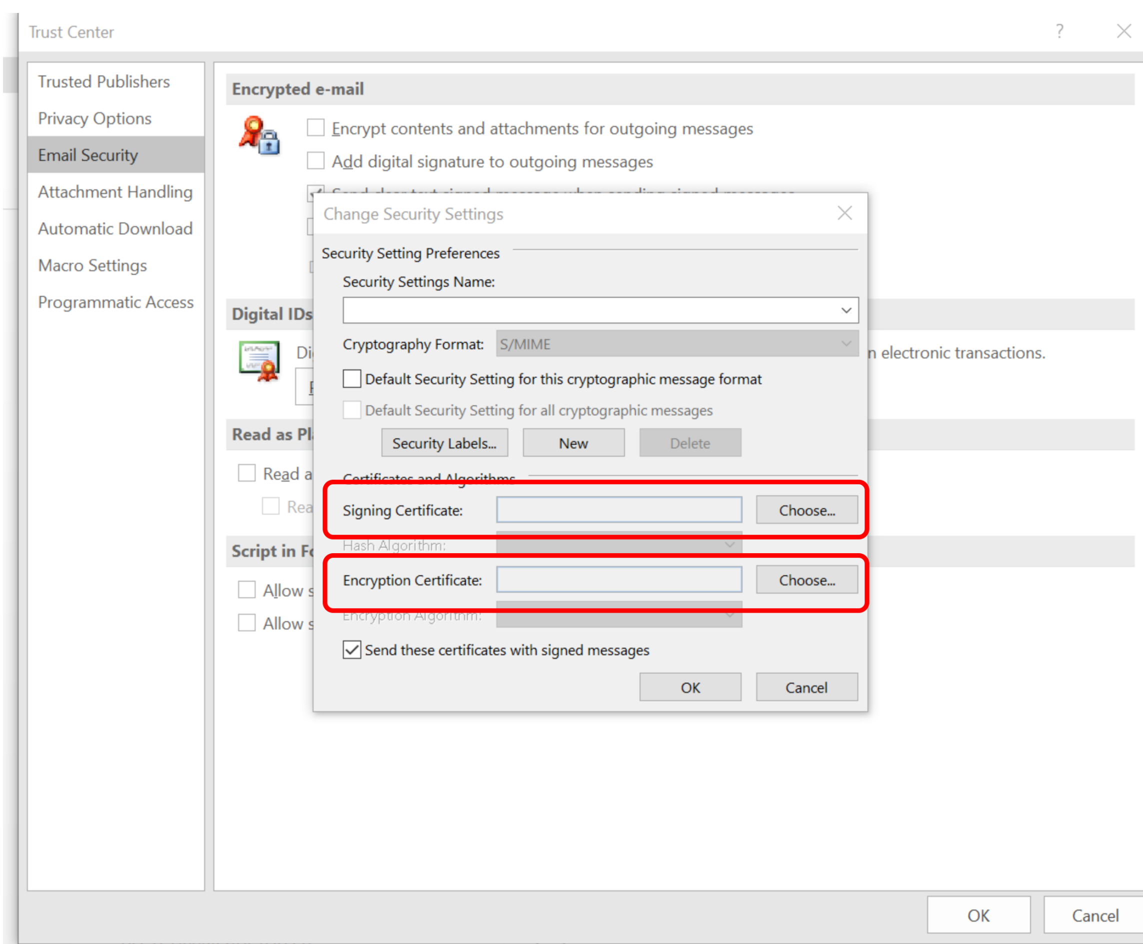

Figure 4.6: Account setting page for Microsoft Outlook 2016 (version: 16.0.4954) for encrypting and signing an email.

Our assumption is that both Alice and Bob have completed steps 1-4 at the time that they wish to exchange encrypted and signed emails with each other. As shown in the steps above, step 5 eliminates the need for checking a recipient's Keybase profile every time that a mail is received. Alice and Bob need to verify their recipients by manually checking their Keybase profile once (the first time, for each specific far-end party) while receiving or downloading their certificates. After saving those certificates to their mail clients there is no need to check their Keybase profile until the certificates are expired or revoked. So step 5 must be repeated, for each correspondent Bob, when Bob gets a new certificate (i.e., on certificate renewal).

When a user is trying to sign and encrypt an email within their mail clients, there are options by which the user can specify the certificates that he wants for signing and encrypting. As shown in the Figure 4.6, there are two fields within the Microsoft Outlook mail client that enables a user to specify the certificates to be used.

In this chapter, we have proposed a design that makes use of Keybase and an ACME server as its main components. We have discussed the steps needed for interacting with each of those components and the steps for exchanging encrypted and signed emails between two parties. We have also argued that to make our design possible, Keybase is required to have S/MIME support mentioned on page 21. The major changes on ACME side are adding the functionality to ACME server (and client) to perform validation checks for an email address and adding user's Keybase account handle to the issued certificate. Having a mail client with support of S/MIME, including a usable graphical user interface for users to easily import email certificates each time they interact with a new recipient, is required. Including social media account cross-checking capabilities when any recipient's email certificates are updated is also important. This could improve the usability of this design due to the fact that this would be more user-friendly than manually opening a user's certificate and copying the Keybase account handle for cross-checking purposes. Also, the mail clients should have the ACME server as a trust anchor. In the next chapter we will analyze our design based on the threat model that has been pointed out in Chapter 3.

Chapter 5

Security analysis

In this chapter, we discuss how our proposed design mitigates threats present in today's email ecosystem, and the threats that were mentioned in Chapter 3, where attack types were categorised into 5 main groups. Middle-person attacks are discussed in Section 5.1 followed by server-side attacks in Section 5.2. Then, client-side attacks and impersonation attacks are discussed in Sections 5.3 and 5.4 respectively. Account compromise is discussed in Section 5.5.

5.1 Middle-person attacks

While performing a middle-person attack, some adversaries may try to intercept and tamper with data during the Keybase registration process. They could attempt to introduce their own keys instead of a legitimate user's keys to Keybase, and to do the same in the registration process with ACME server over ACME communication channel. The registration process between the user and Keybase server is performed with an authentic Keybase client (downloaded from Keybase website and signed by Keybase) on a user's machine in an encrypted manner, and the assumption that public and private key pairs are created on the same machine (the private key is not transmitted to the server). The attacker will be detected due to the fact that his

introduced keys will not match the keys that the Keybase client uses to sign and post proofs for user's social media account.

On the second stage, when the user is registered on Keybase and is trying to interact with an ACME server, ACME server sends an email to the user's email address that includes a nonce that is encrypted with the users public key (that are both advertised to ACME server by the ACME client). By being able to decrypt the nonce, user proves the ownership of the corresponding private key. Also, when the user sends the signed nonce to the ACME server, ACME server checks the signature and verifies the ownership of the private signing key of the user. In this design, the middle-person attack can not succeed for two reasons. First, the communication is done over an encrypted channel between an ACME client and an ACME server; and second, the same public key which is registered on Keybase is advertised to the ACME server with the corresponding Keybase account handle. ACME server would check if both keys (advertised by the user and the one on the Keybase profile) match. Thus, a middle-person attack would fail since the attacker can not change the public keys on a user's Keybase profile, and the ACME server would refuse to issue a certificate if the keys did not match. It is noteworthy that a middle-person attacker may try to disrupt or intercept the set-up of an encryption session between an ACME client and the ACME server, but due to the second reason, unless an active attacker can be present on both ACME channel and ACME validation channel, the public key substitution would fail. These show that the design protects against **T1** regarding a middle-person attack over ACME channels and **T2** regarding a middle-person attack over the communication channel between the Keybase server and the Keybase client.

5.2 Server-side attacks

Although Keybase and ACME claim that they have mechanisms in place to be resilient against denial of service attacks, we assume that these types of attack are out of scope along with various other network-related types of attack.

A compromise within Keybase server or ACME server could be attempted. First, a compromise on Keybase server can not lead to removing and altering users' sigchains in an undetected manner [21]. This is due to the fact that Keybase is pushing all of the sigchains (discussed in Section 2.4) to the blockchain; we believe this would make it impossible for a single adversary to alter data without being detected. However, a compromise on the Keybase server may enable an attacker to obtain the keys that are stored on the server.

Users have the option to upload their private keys to the Keybase server in order to gain access to them from any device that they associate with their account. Private keys are stored on the local device in a password-protected manner. The keys are "TripleSec" encrypted on the client side [22]. TripleSec is a simple encryption library that uses "Salsa 20" and "AES" algorithms for encryption. Also, TripleSec uses "Scrypt"[1] for key derivation, and it claims to defend the encrypted data against password cracking and *rainbow table* attacks. This is in a case that the encrypted keys from a user's device (or from Keybase server should the user choose to upload their private keys to the server in an encrypted format) were stolen and an attacker tries to mount offline password cracking attack on them [43]. If the user chooses to upload his encrypted private keys to the Keybase server, the security of those solely rely on the security of the server and the strength of TripleSec. However, if the user chooses not to publish their private keys to the server, this addresses **T5** regarding a compromise on Keybase servers.

[1] http://www.tarsnap.com/scrypt.html

On the ACME side, an adversary may compromise the server and issue an incorrect certificate with the public key of the adversary instead of the legitimate user. In our proposed design, a user is asked to perform a manual check in the beginning of an email communication. This is done by visiting the Keybase account indicated in the certificate (Keybase account handle is within the certificate) and the public key stated on the Keybase account. Both keys in the certificate, and the Keybase account should be identical. User would notice the inconsistency, would know that the ACME server has been compromised, and the certificate is invalid. We believe that this addresses **T4** regarding ACME server compromise.

Regarding T3, Keybase documentation indicates that in a quite sophisticated attack, the attacker can show two different clients two different versions (forks) of Merkle roots. However, we assume that the users that are communicating out-of-band, will discover the inconsistency. In this kind of attack, the attacker can never merge the different versions or he would get caught [21], [26].

5.3 Client-side attacks

Another attack vector may be the client agents that a user installs on their local machine to interact with Keybase and ACME server. An attacker may try to deceive the user to install a client that is not genuine. This could lead to compromise of a user's private keys. Let's Encrypt states that they do not guarantee the safety of the clients that have been released by third parties [13]. They do recommend a few choices such as "CertBot". This brings us to a point where we have to assume safety of the installed ACME client-side agents. However, if the issue is addressed and a verification method of the client-side software is put into place (such as signing the packages or checking the hashes and fingerprints of the install packages) this can be resolved [13].

As for Keybase, they offer a few security measures to protect the integrity of their client software. First, they have an open API, and their client code is open source which brings in the benefits of open source software alongside the fact that users may try to create their own clients if desired. Second, all of the changes and updates on their client software is signed (by Keybase's private key which its corresponding public key is available on their website) and they have an update mechanism that downloads new clients without trusting HTTPS, but just trusting the integrity of their signature (Keybase claims that the private key is kept offline which prevents it from being compromised in case their server was under attack from a malicious party). In summary, users are vulnerable to **T7** (user installing a malicious ACME client) but there are some countermeasures to mitigate **T6** (user installing a malicious Keybase client) if a user has verified their installation of Keybase client.

5.4 Impersonation attacks

An attacker may try to create a new identity on a social media platform and pursue the legitimate registration process for Keybase and ACME. This would grant him a Keybase account and a valid certificate issued by ACME server. The issued certificate would include the Keybase account handle of the attacker and the Keybase account of the attacker would be bound to his social media account. However, trying to trick other users to trust a newly created account pretending to be a legitimate user should fail when users manually check the Keybase profile and realize it is different from their intended user's profile based on their previous knowledge of their intended user's social media handles. This mitigates **T10** in which an attacker tries to impersonate a user and acquire a certificate from ACME.

On the other hand, since the legitimate user is the owner of the public-private key pairs used in the registration process, an adversary trying to impersonate a legitimate

user to ACME server that doesn't have the legitimate user's private key will fail to complete the process with the ACME server, and in a case that they try to substitute their own public key with the legitimate user's public key in ACME registration process, it will be inconsistent with the public keys on user's Keybase profile. Thus, **T8** (impersonation while interacting with Keybase) and **T9** (impersonation while interacting with ACME) are mitigated.

5.5 Account compromise

In the case that an adversary gains access to a user's social media accounts, the attacker can remove the posted proofs on the legitimate user's social media (Section 2.4.7). As mentioned in Section 2.4.5, Keybase clients perform regular checks on sigchains of users that the legitimate user is trying to interact with. If the proofs were missing from the social media accounts, the Keybase server would not be able to provide the Keybase client a link to the proofs posted on them. While the client would not be able to check the proofs, it would alert the user. This could result in other users not being able to identify that Keybase user through that social media account, but it will not affect the integrity of the public-private key pair and everything related to it. This, addresses **T13** in which the attacker gains access to a user's social media accounts.

If the adversary gains control of a user's Keybase profile, they can only start using it from a list of registered devices that are associated with that Keybase account. This will prevent adversaries from making changes to a Keybase profile from a system that has not been associated with that Keybase account. This addresses **T12**. However, if the adversary gains control over a Keybase account from one of the devices that is associated with that Keybase account, the attacker can remove all of the legitimate user's social media proofs. This would render the proofs no longer valid when another

user's Keybase client tries to verify the user's sigchain. In the worst case scenario, if the user has backed up his private keys to the Keybase server, and those keys are stored in a password-protected manner, the adversary can request to download the private keys by using user's compromised Keybase password. This means an adversary mounting **T13** and **T14** together can defeat our proposed method. We therefore recommend users not to back up their private keys to Keybase server but rather store them locally on their devices in a password protected manner. This would prevent the compromise of private keys even in a case that an adversary has gained access to their Keybase account. Also, the attacker can remove the legitimate user's keys from their profile and add new keys to their account that can be used to request new certificates from the ACME server. This would require an attacker to have access to user's email address as well. This is because ACME requires email communication in order to issue a certificate for an email address. However, due to device compromise, if the user has saved his email credentials to the device, and the device compromise basically results in the email account compromise as well, the attacker can request new certificates with newly introduced keys.

However, if the user loses control of his Keybase profile, other users are prevented from verifying that user by looking for his social media accounts. An attacker who gains access to a user's email account, can not read and send emails because to do so, they would need the legitimate user's private key stored on the user's devices. This addresses **T11** in which an attacker gains access to a user's email address.

In this chapter, we addressed all the threats that were previously mentioned in Chapter 3. We have shown that many of the threats can be mitigated based on our proposed design and for some types of attacks that pose a threat to our design, we have seen that physical access to a user's devices and account compromise should happen simultaneously. In the next chapter we discuss limitations of our design, followed by conclusion and future work sections.

Chapter 6

Comparative Analysis and Conclusion

In this chapter, we present a comparison between the alternative secure email solutions and our proposed design. Then, we discuss the limitations that are present in our proposed design, and we go over the key points that were presented in this thesis. We mention the possible future work that can be pursued based on our proposed design, and we draw a conclusion on the benefits of adopting our design that aims to improve the security aspect of today's email ecosystem.

6.1 Comparison of secure email solutions

As shown in Figure 6.1, we compare alternative secure email solutions with each other on six categories. Within this comparison, we present our subjective opinion about the ease of performing tasks, giving ratings of: Easy, Medium and, Hard. The chosen alternative secure email solutions alongside our proposed design are manly PGP (with Enigmail chosen as an example candidate) and, S/MIME for both enterprise users and individuals. We also want to include Confidante that also makes use of Keybase in its design [24]. At the end we will have a brief discussion on the state of security of these solutions in comparison to our design.

The discussed categories are:

Secure email solution	Our Proposed design	PGP (Enigmail)	S/MIME for enterprises	S/MIME for Individuals	Confidante
Target users	Individuals	Individuals / small groups	Company/ Government employees closed groups	Individuals	Individuals / small groups
Certificate Format	X.509 v3 with specific fields being used	PGP keys	X.509 v3	X.509 v3	PGP keys
Software download requirements	Managed by users	Managed by users	Managed by dedicated IT team	Managed by users	Managed by users
Difficulty of Own Certificate acquisition	Medium	Easy (for technical users)	Easy	Medium(?)	Medium
Acquiring correspondents certificate	Medium	Medium to Difficult	Easy	Medium to Difficult	Easy
Ease of gaining trust	Medium to Difficult	Medium to Difficult	Easy	Medium(?)	Medium

Figure 6.1: Comparison of alternative secure email solutions

1. Targeted users of each of the solutions (i.e. individuals, employees, closed groups)
2. The format of certification that each of these solutions use
3. The requirements for download and installation of the required client software
4. The difficulty of certificate acquisition process
5. The difficulty of acquiring other correspondents certificates in order to initiate secure email communications with them
6. Possibility and difficulty of gaining trust on the acquired certificates (trusting other correspondents certificates).

6.1.1 Target users

Our proposed design targets individuals from a variety of technical backgrounds. The goal is to enable every email user to use secure email without a tremendous effort nor with a considerable technical background. PGP also targets individuals and is more usable while working within small and closed groups. We have divided S/MIME users into two main groups: S/MIME for enterprise users and S/MIME for individuals. The users that are working within an organization that communicate within the organization or within closed groups that have a shared certificate authority fit in the first category. The users that acquire an email certificate from one of the available certificate authorities themselves and are not bound to an organization fit into the second group. We aim to eliminate the localization of organization based S/MIME users and make our design a globally accessible service. In case of Confidante, the target users are individuals and small groups that use Keybase and Confidante mail client for their email communications.

6.1.2 Certificate Format

Both individual and organization-based S/MIME users receive X.509 v3 certificates that are issued by their trusted certificate authorities. In our design we use the same certificates with addition of taking advantage of additional fields within the certificate. As mentioned in Chapter 4, we embed users' Keybase account handles in the SAN field of the issued certificates. PGP users create PGP keys and distribute them amongst their correspondents. Confidante makes use of PGP keys as well. The keys used for Confidante are the same keys that are registered on a user's Keybase account. We have chosen S/MIME certificates over PGP for its popularity and the existing infrastructure and support amongs many service provider (the differences between S/MIME and PGP are discussed in Chapter 2 Section 2.3).

6.1.3 Software download requirements

In our design we require users to install both Keybase and ACME client software on their system. Although installation of these client software does not differ from any other simple software installation, this could be considered as a notable effort in comparison with S/MIME. Specifically, organization-based S/MIME users have the easiest process in this area since, a dedicated IT team is usually responsible to provide and install the requirements for them. PGP users have various options to choose from. However, we have chosen Enigmail which is one of the most popular PGP secure email solutions. Enigmail is a mail client extension that is installed on either Mozilla Thunderbird or Postbox email clients. Enigmail users also need to have GnuPG installed on their system (minimum version currently required is 2.0.14)[1]. Individual S/MIME users need to acquire their certificates from any of the certificate authorities that commercially provide email certificates on the internet (some of which are in charge of creating users' public and private keys and sending them to users themselves).

Confidante on the other hand is a mail client itself. Users are required to install the mail client that currently works with Gmail accounts and required users to have a Keybase account (requiring users to install Keybase client) as well. This poses a limitation on other users that prefer other mail clients or are using other mail service providers than Google.

6.1.4 Difficulty of certificate acquisition

Acquiring a certificate while using S/MIME is considered to be quite easy for organization-based users. They usually do not have to perform any tasks and the IT team is usually in charge of issuance and renewal. For individuals that want to

[1] stated on https://enigmail.net/

acquire a certificate from certificate authorities, there are many services available by which they can fill the required forms and purchase a certificate. The difficulty of this process depends on the CA that the users have chosen for certificate acquisition and some accompanying factors. Thus a medium difficulty (with some uncertainty) has been assigned. PGP users, need to install proper software (mentioned in software download requirements section above) and create their keys within their installed software. Depending on the software they are using, some of them may require above average technical background knowledge to properly create the keys within the software. Since Confidante is using the keys available on users' Keybase profiles, users do not need to perform any specific tasks to acquire a certificate. In our design however, we have two registration processes that are required. Although we understand that these processes may require some user effort, none of the processes are different from daily tasks that users with average technical backgrounds can perform on a daily basis.

6.1.5 Acquiring correspondents' certificates

We believe that any of the mentioned solutions that do not require an out of band communication can easily handle this aspect of secure email communication. This means PGP users and Individual S/MIME users require a means to communicate with their correspondents to enable them to establish a secure email communication. These users need to have their certificates available in some form of online key directories to enable other users to find their public keys. However, we believe that this is a shortcoming that limits other users to easily acquire their intended recipients public keys. Alternatively, a preliminary email exchange with the keys attached can solve this issue. Other solutions that rely on a jointly trusted certificate authority can handle this aspect more easily. Organization-based S/MIME users will typically have a CA configured in their mail clients and usually are not required to perform any tasks

except for an occasional search for their recipients certificates within the organization certificate directory. Confidante users also use Keybase to find their intended recipient and the mail client itself acquires their PGP keys from their Keybase profile. In our proposed design we take advantage of Keybase file system; by requiring users to upload their certificates to their Keybase profile's public directory we enable other users to acquire their recipients certificates by searching Keybase for their profile and downloading their certificates after they are found.

6.1.6 Ease of gaining trust

Amongst organization-based S/MIME users the trust is based on the certificate authority used by the organization to acquire certificates for its users. This certification authority may or may not be embedded as a trust anchor in client software of other users outside that organization. Other individual S/MIME users that acquire their certificates from available certificate authorities and PGP users can not assure other correspondents about their identity unless an out-of-band communication occurs or unless their client software is configured to trust the same certification authority. This means that while the S/MIME certificate could be valid for a particular email address presumably belonging to a particular user, or a PGP key belonging to a particular user may seem trustworthy, it is difficult for users to determine if these email addresses or certificates actually belong to the intended user. Confidante, relies on Keybase requires users to also select the Keybase account handle of their correspondents while trying to compose an email. This can help users gain confidence in the identity of the correspondents. In our design, by binding users certificates, email address, and their social identities together, we present a method to the users to gain confidence in the identity of the certificate holders by cross-checking their social identities. However, these cross-checks make the difficulty level of gaining trust medium for Confidante and "Medium to Difficult" for our proposed design based on the cross-check process

of our design.

Based on the comparison above, we believe that our design takes advantage of the global support and existing infrastructure of S/MIME X.509 v3 certificates enabling a variety of users that use diverse mail provider services and mail clients to acquire email certificates with not much more than medium difficulty. Having benefits of S/MIME and not being limited to an organization while, being able to bind those certificates to social identities, reduces the chance of human error (i.e. sending an email to another person with a name similarity) and targeted impersonation attacks. Support for periodic certificate renewal that is within the technical capabilities of typical users would encourage them to adopt this practice and not use their keys for a prolonged period (many PGP users update their keys only rarely if at all). Also as mentioned in Chapter 5, we show that while having ACME and Keybase as two components and through cross checks between them, many of the middle-person attacks are prevented.

6.2 Limitations and Future work

A number of challenges remain in this proposed design.

There are no graphical-user-interface-enabled clients for ACME at the moment. There is need for a usable graphical ACME client that can be installed on a user's device. It would be more convenient for typical users if that client had a graphical user interface and all the functionality of the currently available ACME clients (in a user friendly manner the same way that Keybase's client software is).

Keybase currently only allows uploading PGP key pairs on their platform, and our design, requires a direct match between the public key presented on a user's Keybase profile and the public key presented to the ACME server. This challenge, can be handled either by enabling a new feature in Keybase, that allows user's to upload

different kind of public keys, or by integrating a mechanism within the ACME client to extract the public and private keys that are in a PGP public and private key block. Another problem that already exists (although it has been mitigated significantly by filtering) is spam email. Achieving end-to-end encrypted secure email may pose some challenges to limiting spam email [8](although this is not limited to our design but to all secure email solutions).

Currently ACME servers follow a 90 day certificate renewal policy. ACME TLS (web server) certificates are being renewed automatically. However, in our proposed design there is need for human interaction while acquiring email certificates. Users need to log into their email accounts and retrieve an email sent by ACME server. They also need to upload those certificates to their Keybase file system in order to make them available to other users. This can be an obstacle from a usability perspective.

As one possible solution for renewal, the ACME client could be granted access to the users email account by providing it with user's email account credentials. This can automate the process of receiving emails from the ACME server and the process of the email certificate issuance. Then the ACME client can present the new certificates to the user so they can upload those certificates to their Keybase profile through the Keybase client installed on their system. However, connecting the ACME client to user's email account can pose new threats to our proposed design due to the possibility that compromise in the ACME client could lead to compromise of user's email account. A security analysis of this matter would also be required.

As another item to consider, the users that receive a signed and encrypted email from another user for the first time need to manually check the sender's Keybase account handle and visit their profile. This means that they have to open the email certificate within their email clients and to find the sender's Keybase account handle in the SAN field of the certificate. They need to copy that account handle into a web browser address bar or within their Keybase client to visit the sender's Keybase

profile and check the connected social media accounts to that Keybase account based on their prior knowledge of the sender's social identities. Renewing the certificates every 90 days requires the users that receive emails encrypted and signed with new certificates to perform this manual check every 90 days. This may become a significant usability obstacle, and a disincentive for users to adopt our proposal.

We argue that automating the manual check, would defeat a major goal of our design which is the manual check of the involved parties' social identities. However, if the renewed certificates used the old public-private key pairs and the senders email address is unchanged, automatic acceptance of the renewed certificates based on the public key and email address combination may be possible. This could prolong the time intervals required to perform a manual check by the receiver of the email to a point that the key pairs for the sender's certificates have been changed. However, the idea of reusing key pairs in automated certificate renewal requires additional security scrutiny, as a major reason for short (e.g., 90 day) certificate lifetimes is to limit the damage if private keys are compromised, and as a simpler solution than traditional certificate revocation solutions [44]. The effects of reusing the old keys of the expired certificates and the optimal time for renewing the keys for the certificates from a security point of view for our design could be one of the topics that can be addressed as future work.

Although combination of cross-checking with a users social media accounts with their email certificates has a positive impact on the security of email exchanges between two parties, it is evident that the impact of this design on usability may be more than we initially anticipated. We have stated that the manual check performed by the users are crucial to ensure the security of the email communications within this design, but this manual check alongside the initial steps (registering on Keybase and ACME and installing their client software) may be too complicated for ordinary users. As maintaining a balance between security and usability has always been a

challenge, this negative impact on usability may discourage users from adopting this design.

For these reasons, a usability study on our proposed method would be useful, as well as a more complete and rigorous security analysis of this design. Together, these may increase the chances of adoption of this design by large organizations and by regular users with less advanced technical capabilities.

Also, turning our proof of concept into a functional prototype remains to be done. Developing an ACME client based on the new authorization challenge and adding a graphical user interface to the ACME client can yield valuable information both on deployability and usability of our design.

6.3 Conclusion

In this thesis, we have combined Keybase to bind a public key with a user's social identity with the use of ACME protocol to partially automate the process of issuing email certificates. We have discussed how registering on Keybase and embedding that Keybase account handle in an issued email certificate can lead to binding a user's email address, public key, and their social media accounts together. This binding aims to improve security of end-to-end encrypted email ecosystem, as argued in Chapter 5, and to reduce the attack surface on encrypted email users such that the attacker needs to gain control of a user's actual devices, Keybase account, and email account at the same time in order to undermine the entire system. We also argue that many forms of middle-person attacks are addressed by this design. Our intention is that this design will also improve key management issues and the certificate issuance process by partially automating various aspects of certificate acquisition and renewal.

List of References

[1] C. Adams and S. Lloyd. *Understanding Public-Key Infrastructure* (2nd edition). Addison-Wesley, 2002.

[2] Andreas M. Antonopoulos, *Mastering Bitcoin: Unlocking Digital Cryptocurrencies*, O'Reilly , Dec 2014

[3] Erinn Atwater, Cecylia Bocovich, Urs Hengartner, Ed Lank, Ian Goldberg, "Leading Johnny to Water: Designing for Usability and Trust", USENIX Symposium on Usable Privacy and Security, 20 pages, 2015.

[4] R. Barned, J. Hoffman-Andrews, D. McCarney, J. Kasten, Automatic Certificate Management Environment (ACME), IETF RFC 8555, March 2019.

[5] Daniel J. Bernstein, Tanja Lange, Peter Schwabe, NaCl: Networking and Cryptography library, http://nacl.cr.yp.to/index.html, visited on February 2020.

[6] Daniel J. Bernstein, Tanja Lange, Peter Schwabe, "The security impact of a new cryptographic library". Pages 159 - 176 in Proceedings of LatinCrypt 2012, edited by Alejandro Hevia and Gregory Neven, Lecture Notes in Computer Science 7533, Springer, 2012.

[7] Bitcoin, Bitcoin wiki, https://en.bitcoin.it/wiki/Main_Page, visited February 2020.

[8] J. Clark, P.C. van Oorschot, S. Ruoti, K. Seamons, D. Zappala, Securing Email, arXiv pre-print, Cornell University Library, 20 Apr 2018.

[9] D. Cooper, S. Santesson, S. Farrell, S. Boeyen, R. Housley, W. Polk, "Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile", RFC 5280, May 2008.

[10] Michael Crosby, Nachiappan, Pradan Pattanayak, Sanjeev Verma, Vignesh Kalvanaraman, “BlockChain Technology: Beyond Bitcoin”, Applied Innovation Review, Issue No. 2, June 2016.

[11] Lee Garber, “Denial-of-Service Attacks Rip the Internet”, Computer magazine, pp. 12-17, vol. 33, April 2000.

[12] Alefiya Hussain, John Heidemann, Christos Papadopoulos, “ A Framework for Classifying Denial of Service Attacks”, Proceedings of the 2003 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications , SIGCOMM’03, Pages 99-110, August 2003.

[13] Internet Security Research Group, ACME Client Implementations, https://letsencrypt.org/docs/client-options/, visited on February 2020.

[14] M. Jones, J. Bradley, N. Sakimura, “JSON Web Signature (JWS)”, RFC 7515, May 2015.

[15] Keybase, Inc., Keybase is now writing to the Bitcoin blockchain, https://keybase.io/docs/server_security/merkle_root_in_bitcoin_blockchain, Keybase official documentation, Visited February 2020.

[16] Keybase, Inc., Keybase Local Key Security, https://keybase.io/docs/crypto/local-key-security, Keybase Official documentation, Visited October, 2019.

[17] Keybase, Inc., Keybase overview and documentation, https://keybase.io/docs/server_security, Keybase Official documentation, Visited October, 2019.

[18] Keybase, Inc., Keybase Pebbl., https://github.com/letsencrypt/pebble/blob/master/README.md, visited November 2019.

[19] Keybase, Inc., Keybase Per-User Keys, https://keybase.io/docs/teams/puk, Keybase Official documentation, Visited October, 2019.

[20] Keybase, Inc., Keybase Sigchain, https://keybase.io/docs/sigchain, Keybase Official documentation, Visited October, 2019.

[21] Keybase, Inc., Server Security, https://keybase.io/docs/server_security, Keybase official documentation, Visited February 2020.

[22] Keybase, Inc., TripleSec, https://keybase.io/triplesec, Keybase official documentation, Visited February 2020.

[23] Keybase, Inc., Understanding following (previously called "tracking"), https://keybase.io/docs/server_security/following, Keybase Official documentation, Visited October, 2019.

[24] Ada Lerner, Eric Zeng, Franziska Roesner, "Confidante: Usable Encrypted Email A Case Study With Lawyers and Journalists", IEEE European Symposium on Security & Privacy, 2017.

[25] Let's Encrypt Project Revision, https://acme-python.readthedocs.io/en/stable/, ACME-Python documentation, visited November 2019.

[26] David Mazieres, Dennis Shasha,"Building secure file systems out of Byzantine storage", NYU computer science department technical report TR2002 826, May 2002.

[27] D. McCarney, Tour of the Automatic Certificate Management Environment (ACME), Internet Protocol Journal, Jun 2017.

[28] M. Mealling, L. Masinter, T. Hardie, G. Klyn,"An IETF URN Sub-namespace for Registered Protocol Parameters", IETF RFC 3553, June 2003.

[29] K. Moriarty, Ed., J. Jonsson, B. Kaliski, A. Rusch, PKCS #1: RSA Cryptography Specifications Version 2.2, IETF RFC 8017, November 2016.

[30] Satoshi Nakamoto, Bitcoin: A Peer-to-Peer Electronic Cash System, Unreviewed paper, October 31, 2008.

[31] Narayanan et al., *Bitcoin and cryptocurrency technologies: A comprehensive introduction*, Princeton University Press, 2016

[32] Colin Percival,"Stronger key derivation via sequential memory-hard functions", http://www.tarsnap.com/scrypt/scrypt.pdf.

[33] C. Percival, S. Josefsson, "The scrypt Password-Based Key Derivation Function", RFC 7914, August 2016.

[34] E. Rescorla, B. Korver, "Guidelines for Writing RFC Text on Security Considerations", RFC 3552, July 2003.

[35] Rivest Ronald L., Adi Shamir, Leonard M. Adleman, “Cryptographic communications system and method”, Patent Office, Patent US4405829A, December 14, 1977.

[36] Ruoti, S., Andersen, J. , Zappala, D., Seamons, K., “Why Johnny still Can't Encrypt: Evaluating the Usability of a Modern PGP Client, ArXiv, 2015.

[37] Saltpack, “Saltpack Binary Encryption Format [version 2]”, https://saltpack.org/encryption-format-v2, Visited June 2020.

[38] Q. Sceitle, T. Chung, J.Hiller, O. Gasser, J. Naab, R. van Rijswijk-Deji, O. Hohlfeld, R. Holz, D. Choffnes, A. Mislove, G. Carle, “ A First Look at Certificate Authority Authorization (CAA)”, ACM SIGCOMM Computer Communication Review, Volume 48 Issue 2, April 2018.

[39] J. Schaad, B. Ramsdell, S. Turner, “Secure/Multipurpose Internet Mail Extensions (S/MIME) Version 4.0 Message Specification”, RFC 8551, April 2019.

[40] The Go Authors, Secretbox package, https://godoc.org/golang.org/x/crypto /nacl/secretbox, visited on February 2020.

[41] The Go Authors, Secret-key authenticated encryption: crypto_secretbox, http://nacl.cr.yp.to/secretbox.html, visited on February 2020.

[42] S. Turner, “The application/pkcs10 Media Type”, RFC 5967, August 2010.

[43] Filippo Valsorda, On Keybase.IO and encrypted private key uploading, https://blog.filippo.io/on-keybase-dot-io-and-encrypted-private-key-sharing/, visited on February 2020.

[44] Paul C. van Oorschot, *Computer Security and the Internet: Tools and Jewels*, Springer, 2019.

[45] A. Whitten and J. D. Tygar, “Why Johnny can't encrypt: A usability evaluation of PGP 5.0”, USENIX Security, 1999

[46] P. R. Zimmermann. *The Official PGP Users Guide.* MIT Press, 1995.