ON END-TO-END ENCRYPTION FOR CLOUD-BASED SERVICES

Suryadipta Majumdar

A thesis

in

The Department

of

Concordia Institute for Information Systems Engineering

Presented in Partial Fulfillment of the Requirements

For the Degree of Master of Applied Science

in Information Systems Security at

Concordia University

Montréal, Québec, Canada

September 2014

Concordia University

School of Graduate Studies

This is to certify that the thesis prepared

By: **Suryadipta Majumdar**

Entitled: **On End-to-end encryption for Cloud-based Services**

and submitted in partial fulfillment of the requirements for the degree of

Master of Applied Science (Information Systems Security)

complies with the regulations of this University and meets the accepted standards with respect to originality and quality.

Signed by the final examining committee:

Dr. Yong Zeng _____ Chair

Dr. Amr Youssef _____ Examiner

Dr. Ashutosh Bagchi _____ External Examiner

Dr. Mohammad Mannan _____ Supervisor

Approved _____

Chair of Department or Graduate Program Director

_____ 2014 _____

Dr. Amir Asif, Dean

Faculty of Engineering and Computer Science

Abstract

On End-to-end encryption for Cloud-based Services

Suryadipta Majumdar

Cloud-based services are now an integral part of everyday lives of many users. Indeed, users who do not use Facebook, Gmail, Dropbox, GoogleDrive, QQ, Baidu or similar services are now a rarity. These services offer seamless integration of user data with multiple user-owned devices, reliable online backup, and enable easy and instant communications between users. Such features, at an affordable price of zero dollars, make these services very popular, even though they are an antithesis to user privacy, and help create large-scale surveillance programs such as NSA PRISM. Several mechanisms have been proposed and implemented to make these services privacy-friendly. Most past proposals rely on public key systems with user-managed private keys, or password-based symmetric encryption. We explore a symmetric-key approach without password-derived keys to facilitate end-to-end encryption of stored user data (e.g., cloud storage) and communication messages (e.g., web-based email). We propose Keyfob, a key management scheme for easy key transfer between user-owned devices, and between users. Keyfob uses high-entropy random keys for encryption instead of password-derived keys, and leverages DH-EKE (Bellovin and Merritt, IEEE S&P 1992) with weak secrets for secure key transfer. Each user needs to manage one user-master key, and all other keys are derived from that master key or a pairwise shared master key. We implemented Keyfob as a Firefox extension using the Firefox Sync service, which implements an EKE variant. Keyfob can make several

applications and services privacy-friendly, if appropriate intermediate layers are implemented, e.g., as plugins between a target cloud-service application and the Keyfob extension. We have implemented two such plugins to support encrypted Dropbox (in desktop and Android) and Gmail (in desktop). Our hope in proposing Keyfob with a symmetric-key approach is to highlight challenges in such a lesser-explored mechanism, and attract researchers towards the long-standing problem of enabling end-to-end encryption in a cloud-dominated environment.

Acknowledgments

At the very beginning, I would like to thank my adviser Dr. Mohammad Mannan. His continuous availability and guidance helped me the most to finish this thesis work. I am grateful to him for introducing me to the research world and feeding me the basics of this world. He always listens to my problems patiently and tries his best to rescue me with his inspiring speeches. Moreover, he gave his attention to almost all of my requests, which shows his great kindness. I feel very lucky to have him as my supervisor.

I also like to thank all other faculty members of the CIISE department. I really enjoyed and learned a lot from the courses that I attended during my master's program. I would like to show my gratitude to Ian Goldberg and Jeremy Clark for their insightful advice, along with my all fellow labmates specially Vladimir Rabotka, Lianying Zhao, Xavier de Carné de Carnavalet and Briti Sundar Mondal for their enthusiastic discussions.

At the end, I would like to acknowledge the unconditional affection and continuous support of my parents and sister in my life. They always fulfill all my silly demands and keep faith on my ability. They have always been the best inspiration in my life.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

At present, more than 1.25 billion people use different cloud services for backing up data and sharing information, e.g., cloud storage services[1], social networks[2], e-mail[3], etc. All of this services deal with sensitive personal information and most user data is stored in plaintext at a provider's server. It seems that the majority of users finding it acceptable that their personal photographs, payslips, intimate messages are accessible by service providers. Possible privacy consequences are not taken into account due to lack of awareness. A common misconception is that personal information of an individual user is of no interest to giant service providers like Google or Dropbox. Also, most people are unaware of the infrastructure and the underlying technology for cloud services. They think they can hide themselves in the crowd of billion users, which may not be true. Throughout the later half of 2013, disclosure of NSA's wide-reaching surveillance programs has caught attention and increased our concern

[1] http://techcrunch.com/2014/04/09/dropbox-hits-275m-users-and-launches-business-product-to-all

[2] http://socialmediatoday.com/jonathan-bernstein/1894441/social-media-stats-facts-2013

[3] http://www.theverge.com/2012/6/28/3123643/gmail-425-million-total-users

towards user privacy. At least, it is now established that user data is being leaked from many large corporations (see [39], [24]). Even considering unawareness of privacy among mass people, we believe that effective solutions of end-to-end encryption for this services is a demand of time.

To facilitate end-to-end encryption in a cloud ecosystem, the main challenge is to manage keys among different devices of the same user and between different users. In recent years, several encrypted cloud storage services (e.g., Mega, Viivo, Wuala and Boxcryptor) have been launched. Main approaches are:

1. encryption with public key without offering several popular features like syncing, sharing (e.g., Syme [74]);
2. encryption with user chosen password (e.g., Boxcryptor [11]); and
3. relying on a third party key server (e.g., CaaS [21], FriendlyMail [61]).

Apart from cloud storage services, email communication is also a potential candidate for end-to-end encryption. PGP and S/MIME are the best solutions we currently have for email encryption, but unfortunately they have not got much popularity for several reasons. Firstly, they are not designed for cloud mail. In a cloud-based system, syncing a private key to user-owned devices is equally challenging as transferring a symmetric key. Another major reason is their usability shortcomings ([34, 72]). Both PGP and S/MIME require complex initial configuration and handling with cryptographic artifacts [81]. Apparently, most users are not interested to protect their own privacy with the price of significant reduction in usability. Also, service providers have less incentive to offer end-to-end encryption, because many of their existing features rely on unencrypted data storage. End-to-end encryption is preferable to be added as an essential security feature in most of this services by keeping the minimal impact.

We observe that most existing approaches are either vulnerable to a malicious service provider or relying on a trusted party. Public key cryptosystem is the most popular in handling key management for end-to-end encryption. Apart from above-mentioned limitations, troubles adapting public key in cloud services are:

1. protecting keys from malicious providers; and
2. non-trivial key management.

1.2 Thesis Statement

In this work, our main objective is to propose a key management mechanism for cloud services to facilitate end-to-end encryption with high-entropy key without relying on a trusted party. To achieve this goal we explore following research questions:

Question 1. How can we design and implement a key manager which makes existing popular cloud services privacy-friendly?

Question 2. What are the benefits and shortcomings of using symmetric encryption for cloud-based services?

Question 3. What are the popular cloud features we can offer with the proposed key management mechanisms?

1.3 Contributions

At first, we identify 6 targeted features and challenges to facilitate end-to-end encryption for cloud services. We design and implement a key management scheme which includes key transfer mechanisms and key derivation methods to transfer master keys and derive other related keys respectively among user-owned devices as well as other user devices. We also implement three encryption plugins for cloud storage services

(Dropbox) and email communication (Gmail) to facilitate end-to-end encryption in two different platforms (Windows and Android). We keep our key management module as less dependent on cloud storage and email services as possible so that it can be pluggable with other services easily. Additionally, our system is minimally deviated from the existing process for the convenience of users.

Main contributions of this work are following:

1. We propose Keyfob, a key manager to facilitate privacy-friendly cloud services without relying on user-chosen passwords for data confidentiality, and requiring trusted third parties. Keyfob is application and service agnostic, i.e., no changes are needed in official client applications or in the server-end; it can be used with any specific cloud service, as long as a service-specific software plugin (e.g., application or browser extension) is implemented.

2. To show feasibility of Keyfob's design, we implement plugins for Dropbox (in desktop and Android) and Gmail (in desktop) to enable encrypted cloud storage, encrypted shared content and confidential emails. Our Dropbox plugin can also be used with several other popular cloud storage services, e.g., GoogleDrive, OneDrive. To facilitate gradual adoption, these plugins allow encryption support for selected content (files or emails); i.e., users can still send plaintext messages or store files unencrypted.

3. Although Keyfob relies on authenticated Diffie-Hellman key exchange (DH-EKE) [7], there are no long-term public-private key pairs to maintain. Public key based systems are numerous, but gained very little adoption. With Keyfob, we explore a symmetric-key based system, a less-explored approach in academic literature. We require each user to back up only one master key (a high-entropy symmetric key) for all cloud service accounts.

4. Apparently as a result of the Snowden-effect, several encrypted cloud storage mechanisms have been proposed/implemented in the recent time. We evaluate these techniques, and compare Keyfob with existing key management approaches for cloud services.

1.4 Outline

The rest of this thesis is organized as follows. Chapter 2 covers some necessary background and literature related to this dissertation. In Chapter 3, we first illustrate the threat model for this work and then define the goals for the Keyfob design. We present the details of Keyfob design including the details of our approach of solving different key management issues in Chapter 4. Chapter 5 includes all the implementation details of Keyfob. In Chapter 6, we discuss the security analysis of our proposal with the respect to our threat model. Discussion on analytical evaluations based on user steps and deployability as well as future work are placed in Chapter 7. Chapter 8 concludes.

Chapter 2

Background and Related Work

In this chapter, we discuss the necessary background and literature related to our work.

2.1 Background

This section includes the background discussion mainly on the protocols and tools we use in this work extensively.

2.1.1 Password-authenticated key exchange (PAKE)

PAKE is a key establishment method utilizing the knowledge of password by both parties. The basic idea behind this approach is not to include or not to be leaked any verifiable information to stop an attacker to conduct an offline guessing attack on the password. Many PAKE-based proposals (e.g., [7, 8, 29, 30, 40, 41, 49, 82] are available, where two parties want to authenticate to each other using a pre-shared password and agree on a high-entropy session key to be used to protect their subsequent communication.

EKE [7] is the first protocol in its kind. In EKE, one party sends an ephemeral

public key protected by pre-shared password and other party replies with the session key encrypted by the ephemeral public key. Here, the assumption is that both parties share a password P in advance. A simplified version of EKE is the following (see Table 1 for the description of notations used):

1. Alice generates a random ephemeral public key E_A and encrypts it with the pre-shared password P. Alice sends to Bob
   $\{E_A\}_P$.

2. Bob decrypts to retrieve E_A from the previous message with shared password P; generates a random session key R; and encrypts R with both E_A and P. Bob sends to Alice
   $\{\{R\}_{E_A}\}_P$.

3. Alice decrypts the message with P and her private key for E_A to obtain the session key R.

After these three steps, both parties agree upon a high-entropy session key R. The strength of this protocol is that any dictionary attack on "weak" secret P is infeasible. There are three versions (RSA, ElGamal and Diffie-Hellman) of EKE available depending on the way the public/private key pair is generated. Patel [60] formed number-theoretic attacks on all versions of EKE. For DH-EKE, the attack can be avoided by not allowing specific choices of the generator g and prime p.

In this work, we heavily rely on EKE (more specifically, DH-EKE). To bypass the patent on EKE, several variants of EKE, e.g., J-PAKE [37], SRP [82], SPEKE [40], have been proposed. According to our knowledge, only J-PAKE and SRP are being used in public systems (e.g., Firefox Sync and TLS-SRP). The patent on EKE expired in late 2011. In this work, we focus more on DH-EKE, as DH-EKE is one of the simplest variants of password-authenticated key agreement and also a secure option

(see e.g., [60]). The simplified steps are:

1. Alice generates a random exponent x; calculates g^x mod n , where g is a generator and n is a prime; and encrypts it with pre-shared password P. Alice sends the encrypted result.

2. Bob decrypts to retrieve g^x mod n from the previous message with the shared password P; generates a random key y; encrypts g^y mod n with P; and sends the encrypted result.

3. Alice decypts the message with P to obtain g^y mod n.

After the third step, both parties agree upon the same session key, g^{xy} mod n. Unlike regular Diffie-Hellman, DH-EKE prevents man in the middle attack (MiTM) by authentication.

2.1.2 Firefox Sync

We use the Firefox Sync server instead of a direct user-to-user DH-EKE because, it is unable to connect peer-to-peer in most cases (e.g., due to the use of NAT, firewall). Firefox Sync [56] is one of the very few real world services that use an EKE-like protocol. It is a built-in service in Mozilla Firefox for synchronizing user data and preferences (e.g., history, password) to all user-owned devices. Firefox Sync uses J-PAKE [37], to establish a session key. Sync uses this session key to transfer any data from one device to another via the Firefox Sync server. So, user data is never exposed to the Sync server. We mainly rely on the "easy setup" step of Firefox Sync, where a session key is established through a J-PAKE session. J-PAKE invalidates offline dictionary attacks on the pre-shared weak secret. Also, it considers the Sync server untrusted. Below we describe the "easy setup" mechanism in Firefox Sync (Firefox 28).

1. The Firefox Sync client requests a channel from the server. In response, the server provides a channel ID, which is used for any further request during a session.

2. Then, the Sync client generates a random weak $Secret_{ij}$ and prepares a PIN by appending the $Secret_{ij}$ with the channel ID.

3. From another user device, the user enters the PIN generated in the previous step. The channel ID is retrieved from the PIN.

4. Once both devices know the PIN, they can establish a session key using the J-PAKE protocol.

The recent version of Firefox features a new Firefox Sync design [12], where the "easy setup" step is excluded. Mozilla found that the previous design with "easy setup" is not convenient for users having one device. Also, if users lose their master keys, then all contents are irretrievable. The current design relies on the user-chosen password so that encrypted contents are recoverable as long as a user has the knowledge of the password. Additionally, the new version allows users to store contents in the Sync server in the clear to offer full recovery. So, the security of the data depends on the user-chosen password. As we need Firefox Sync for multiple devices, we find the previous design more applicable for this work. Mozilla announces to continue to host the old Sync server for a limited period of time [54]. After that to avail the old Sync service we must run our own Sync server (see [57]). As our design only needs the Firefox Sync server to establish the key, and does not consider the server as trusted, running our own server should not be a concern in terms of performance or security i.e., no data transfer or user registration (see Section 7.2 for estimated server load).

2.1.3 Sharing secret over insecure channel

There is no efficient way to transfer a secret over an insecure channel without having any pre-shared secret. Public-key cryptography was invented to solve this problem. But, it fails when the integrity of the public key is not preserved in the transition and lets an attacker to be a transparent man-in-the-middle. Apparently, most convenient way to share a secret with a friend is to call or send an SMS to that person. But, eavesdropping by the mobile operators may leak the secret. Alexander and Goldberg [1] suggested to use any previously shared experience (e.g., last movie watched together) as secret. This approach is vulnerable if Eve gathers personal information about Alice and starts sharing a secret as Bob. The attack difficulty is increased when both parties ask two questions about two different past events to each other. We doubt that people can keep their past experience secret in this era of social networking and cloud services.

Quantum key distribution (see e.g., [9]) relies on the quantum channel to exchange keys between two parties. It enables both parties to detect if the key is eavesdropped, as the quantum message gets disturbed at the presence of an eavesdropper. But, this approach is vulnerable to man-in-the-middle attack when used without authentication that concludes that we can not use the Quantum key distribution safely without a pre-shared secret between two parties. We identify the de-centralized services (e.g., Tor, BitTorrent) as a potential candidate to transfer secret. Section 7.4 describes one of the possibilities by leveraging Tor hidden services [77].

2.2 Related Work

In recent years, end-to-end encryption solutions especially for cloud services are getting popular. In this section, we focus on the main approaches available to facilitate

end-to-end encryption for cloud services. We identify several shortcomings in existing approaches, which are addressed in our solution.

2.2.1 Protected email

Hallam et al. [35] and Yu et al. [42] discussed different challenges in designing an encrypted email system. The first challenge is to avoid any trusted party in the design. S/MIME [62] and PGP [13] are two well-known end-to-end solutions for email encryption. PGP decentralizes the chain of trust, whereas S/MIME relies on a trusted third party (e.g., certification authority). In PGP, email contents are encrypted by a random symmetric key. This symmetric key is then encrypted using the recipient's public key and sent with the encrypted email to the receiver. Unlike other public key based systems, PGP does not rely on PKI. PGP introduces the *web of trust* model for the key distribution, where users recommend each others' certificates. Though this model matches with the trust model of real life, it may not be always true. Because, users may need to certify other users, who are not known beforehand. Whitten et al. [81] first identified the user inconvenience in the PGP protocol. They identified that the initial configuration steps in PGP are too complicated for users. On the other hand, S/MIME relies on PKI, which makes the key management costly and insecure. Certificate authorities can replace public keys so that they can access user data later. Self certificate may seem a better option, but it is not free from its own security risks and usability issues [28]. Ryan et al. [65] proposed an encrypted email service using PKI without any trusted party. In this proposal, service providers maintain a public log to store public keys of all customers, so that any malice done by the service provider can not be undetected. FriendlyMail [61] is a Firefox extension offering encrypted email for the Gmail web interface. It encrypts email contents with random symmetric keys. Random keys are sent separately through another channel (e.g.,

a private message in Facebook). It requires the sender and receiver to be friends in Facebook. The main downside of this approach is that it fails to protect confidentiality when Google and Facebook collude or the government on top of them collects data from both. Lavabit, Silent circle and PrivateSky [47, 68, 15] were encrypted email services, which were shut down because of their reluctance to disclose user emails. As most of those services offered server-side encryption, all emails were accessible by the service providers. These incidents taught us not to rely on any trusted party including the service providers anymore. Dark mail technical alliance [16] is a recent effort to come up with a privacy-friendly email technology.

2.2.2 Password-based encryption systems

A popular approach is to use password to propose a privacy-friendly cloud service (e.g., [31, 45, 66, 11, 69]). Apple's iCloud stores all sensitive user data (e.g., password, encryption keys) into the iCloud Keychain. It offers users to sync their secrets to all user-owned devices. The first user device creates a public/private key pair. Public keys are stored as a "circle" in the iCloud server after being protected by user's iCloud password, and signed by the device's private key. To access the public key an attacker must authenticate with the user password, and to modify the public key an attacker must provide the original device's private key. To add a new device, user's existing device authenticates the new device with user's iCloud password. Once the authentication succeeds, the old device adds the public key of the new device to the "circle". The public key is signed by both the private key and iCloud's password. Then both devices can upload to the iCloud server by encrypting with each other's public key. According to our knowledge, iCloud Keychain is the only public-key based service which offers syncing without transferring the private key. As the pairing of a new device relies on the user password, only online attack on password is feasible.

Unfortunately, recent finding [83] on undocumented forensic services running on every iPhone device shows that any surveillance on iPhone users is feasible. Apparently, Apple knows about the backdoor and keeps updating these forensic services. The *file_relay* service is responsible for giving access to many sensitive user data and the dump of all metadata.

Google recently announced the Chrome extension *end-to-end* [31], which is a JavaScript implementation of OpenPGP. To support multiple user-owned devices, *end-to-end* transfers the private key protected by a user-chosen password or pass-phrase. So, a successful brute force attack on user password allow Google to access user emails. Yahoo recently announced to launch a similar encryption plugin for their email service [25]. Yahoo's plugin will be a modified version of Google's *end-to-end*, though the details of the design is still unavailable. Boxcryptor [11], which is a cloud storage encryption plugin, works with major cloud storage providers (e.g., Dropbox, Google Drive, OneDrive), similarly protects private keys by user-chosen password. In Boxcryptor, each file is encrypted by a random key and each user has a public/private key pair. Random keys are encrypted by public keys of each user, who has access to that file and stored in the local cloud storage directory. All public keys are in the clear and private keys are encrypted by user password; both keys are stored in the cloud storage server to facilitate sharing and syncing. To share a file with other users, the file key is encrypted with the receiver's public key and submitted to the cloud so that the receiver can retrieve the file key by decrypting using her private key.

MiniLock [52] is a file encryption service. It generates public/private keypairs from user pass-phrases. Public keys are considered as miniLock ID. Users exchange their miniLock IDs prior to sending any file. Using the elliptic curve cryptography, miniLock generates public/private keys small enough to share through tweets. To encrypt a file, the sender drag the desired file and recipient's miniLock ID. All files are

encrypted by the recipient's public key before leaving sender's device. The recipient can decrypt the file by entering her email address and passphrase.

There are several symmetric-key-based solutions, where in most of the cases encryption key is derived from a password. The Safebox [66] cloud storage service encrypts all user data with the password-derived key. To support syncing, Safebox does not need any additional key distribution. A user device can sync all data from the Safebox server, and decrypt them using the key derived from the user's master password. Wuala [45] is another encrypted cloud storage service, which derives the master key from the user-chosen password. The master key encrypts the root directory. Other encryption keys reside in the parent directory so that a user having the encryption key of a folder can access all the files and sub-folders inside the folder. SpiderOak [69] offers both syncing and sharing. All user data is encrypted by symmetric keys. Encryption keys are protected by the user password. To facilitate sharing, SpiderOak requires a user to create a *share room*. Each *share room* has a password, which is used to encrypt all data in that room. All the members of a room know the password for the room. Trrst [79] is an encrypted and anonymous blogging platform. Trrst transfers a private key protected by the user chosen-password. The main downside of these solutions is that the security of the system is dependent on the strength of user-chosen password. It is observed that people usually choose a weak password (see e.g., [70]).

2.2.3 Trusted third party

CaaS [21] introduced a semi-trusted third party to manage encryption keys. CaaS leverages one-time padding encryption to achieve associativity. The sender add her layer of encryption before sending to the CaaS server and remove the layer before sending to the service provider. All contents at the service provider are encrypted by

the CaaS key. In this proposal, a sender, receiver and CaaS server (third party) have their own encryption keys. At first a sender sends the file encrypted by her key to the CaaS server. CaaS encrypts the file again with its key and sends back to the sender. Then, the sender removes her encryption and leaves the file encrypted by the CaaS key for the cloud storage server. This system uses the one-time padding encryption algorithm to achieve associativity. The receiver of the file downloads it and forwards to the CaaS server after encrypting by the receiver's key. The CaaS returns the file to the receiver after removing its layer of encryption. Then the file is only encrypted by the receiver's key. Thus user data is never exposed to the cloud storage provider and CaaS server. CaaS requires user registration with email address and password so that the CaaS server can store encryption keys against user email addresses. Also, CaaS enforces the usage of SSL between client and service providers including the CaaS server to avoid any MiTM. Otherwise, an attacker can collect encryption keys as well as encrypted user data from the user communication with the CaaS server and service providers. If the CaaS server and cloud server colludes then user data will be exposed.

2.2.4 Other approaches

Syme [74] is an encrypted group conversation service. Each member of a group has a public/private keypair, and also knows other members' public keys. To exchange public keys with a new member, an ephemeral keypair is generated by the new member and transferred in the clear through the Syme server. Each file is encrypted with random keys. All the encryption keys are encrypted by the public keys of all members of a group and stored in a key file; the key file is available to all group members. Also, each group member knows other members' public keys. Whenever a new member is added to the group, an ephemeral keypair is generated by the Syme client. Ephemeral

keypairs are transferred through the Syme server to exchange public keys. Once a member has public keys of all the members, she can securely communicate with others. The main weakness of Syme is that it relies on the service provider during the steps of transferring public keys using an ephemeral keypair. As ephemeral public keys are transferred in the clear, the Syme server can replace ephemeral public keys so that false public keys of other members can be sent to a new member. Thus the Syme server will be able to access all the future posts from that the newly added member. If the server acts maliciously from the starting of a group, then all conversations for the group will be readable by the server.

SafeSlinger [22] offers a key exchange mechanism highly depending on the multi-value commitments and group Diffie-Hellman key agreement (e.g., STR [43]). The purpose is to share a secret over a group while they are meeting in person so that the secret can be used to protect future communication. SafeSlinger avoids $O(n^2)$ communications for key exchange between n members of a group. The major steps of SafeSlinger are:

1. Data selection. The user selects the contact information to be shared.

2. Group size. The user enters the size of the group.

3. Group selection. Each member of the group is provided a unique ID. To identify as a same group to the server, all members enter the lowest ID of the group.

4. Data and commitment distribution. Each device submits the encrypted data and commitments to the server.

5. Verification. By matching three word phrases among all members, users verify the integrity of commitments.

6. Shared secret. Each device derive the group key from the received commitments and own private key using the group Diffie-Hellman key agreement.

7. Data retrieval. User data is encrypted by random keys. Random keys are distributed after encrypting them by the group key. So, all legitimate members of the group is able to decrypt the data.

SafeSlinger is able to exchange keys between group members without exposing keys to the server. But, it requires all group members to be present at the same time. If the group size is 2, then the communication for key exchange will be $O(n^2)$.

CrowdShare [5] is a resource sharing mechanism using mobile tethering. A key server generates public/private key pairs for encrypting messages between mobile devices. As resource sharing steps do not involve the server, accessing the encrypted messages is difficult for the server. ARKpX [4] offers client side encryption without providing any key management support. A user is responsible to transfer or share encryption keys.

Chapter 3

Threats & Design Goals

In this chapter, we define our threat model and identify the goals to achieve with our proposal.

3.1 Threat model

3.1.1 Service-provider related assumptions

For user data at rest or in-transit, we assume service providers (cloud storage providers/ISPs) are a curious attacker, i.e., they would always prefer to have access to plaintext user data, for reasons including: profiling users and target ads. We also assume these providers are non-malicious in terms of availability and functionality (i.e., no DoS attacks on users), and take adequate steps to safeguard their resources (but possibly not enough to deter government agencies). During authentication, we assume all communications between a client and server are SSL/TLS protected. We also assume all out-of-band channels are eavesdropped (see e.g., [75]). We do not trust providers for not sharing user data with third parties, for legal or business reasons; i.e., service providers can collude with each other or with government agencies. For

password-protected keys (symmetric/private), we model the providers as malicious entities, when they have access to such password-protected keys; i.e., they will launch large-scale brute-force attacks to expose such keys, or use password cracking services for hire, e.g., the Stricture Consulting Group.[1] We also assume service providers as malicious if they are used for sharing public keys between users (e.g., [46]); i.e., they can replace user public keys to launch a simple man-in-the-middle attack.

3.1.2 End-user related assumptions

We have several types of keys in Keyfob: user master key (unique for each user), pair-wise shared master key (between each user pair), application keys, and per-item keys (for each data item, e.g., file or email content). Users must protect all keys from exposure. We assume all user devices are malware-free, which is a rather difficult goal to achieve, especially against targeted/government-sponsored attacks (e.g., TURBINE [27]). Backing up the user master key is critical for Keyfob. Several possibilities of backing up the master key are discussed in Section 4.4. On the recipient side, backing up the pair-wise master key is strongly recommended; otherwise, a new DH-EKE session must be performed to get the key from the initiator/sender. For the DH-EKE secret, we assume it is shared between users out-of-band (e.g., phone call, SMS), where both parties at least know each other's contact numbers, or through an online channel that cannot be tied to specific users (e.g., message transmitted via Tor hidden services). We assume an attacker will be detected if she tries to play the role of sender to receiver or vice-versa, when the DH-EKE secret is being shared.

[1] `http://stricture-group.com`

3.2 Design goals

The primary goal of Keyfob is to enable easy key management to support end-to-end encryption between all user devices and between users. Our target features include:

1. Secure existing cloud services, instead of introducing a new 'secure' service. Providing strong security guarantees of user data from a cloud provider is possibly impractical (cf. the Lavabit affair [17]).

2. Introduce no trusted third-parties for key transfer or as a key server. We use Firefox Sync for key transfer, where the Sync server is untrusted.

3. Support multi-device key syncing, assuming most users own more than a single device, and user devices may be unable to connect peer-to-peer (e.g., due to the use of NAT, firewall). We use the Firefox Sync service (modified on the client-side) to support this feature. Note that, we are not concerned of syncing encrypted user data, for which we rely on existing services.

4. Enable secure content sharing between users. For cloud storage services, sharing files between users is a very popular feature.

5. Support multiple services/applications (e.g., cloud storage, email, instant messaging, social networking) using the same master key (for single-user) or the same pair-wise master key (between users). We currently target cloud storage and web email services for our implementation. We design Keyfob for easy integration of other services.

6. Simplify key management tasks for users. We choose a symmetric-key approach instead of public-key, as apparently understanding a two-key (public-private) scheme is less straightforward than a single-key symmetric scheme (symmetric keys are more similar to real-world keys). We also must enable easy backup of

master keys. We want to keep the key-establishment procedure a one-time step, for adding each user device, or a peer user.

Chapter 4

Keyfob Design

This chapter describes the details of key management mechanisms for Keyfob.

4.1 Design overview

Figure 1 illustrates major steps in Keyfob for single-user and two-user cases. For both cases, a master key (either K_{U_i} or $K_{U_{ij}}$) encrypted by a session key is transferred through the Firefox Sync server; the session key is established using a PIN by a DH-EKE session. The PIN is sent out of band. For the single-user case, the encrypted file and encrypted key database are synced through Dropbox. Device 2 gets the master key through Sync; retrieves the key database; and finally decrypts all files. For the two-user case, the encrypted file and encrypted key database are sent through the Dropbox server. User 2 receives the shared master key through the Firefox Sync server; retrieves encryption key; and finally reads the file. Figure 2 illustrates the steps of encrypted email service with Keyfob. The encrypted email and encrypted key are sent through the Gmail server. User 2 receives the shared master key through the Firefox Sync server; retrieves encryption key; and finally reads the email. Below we discuss our key management mechanisms for both single-user and two-user cases

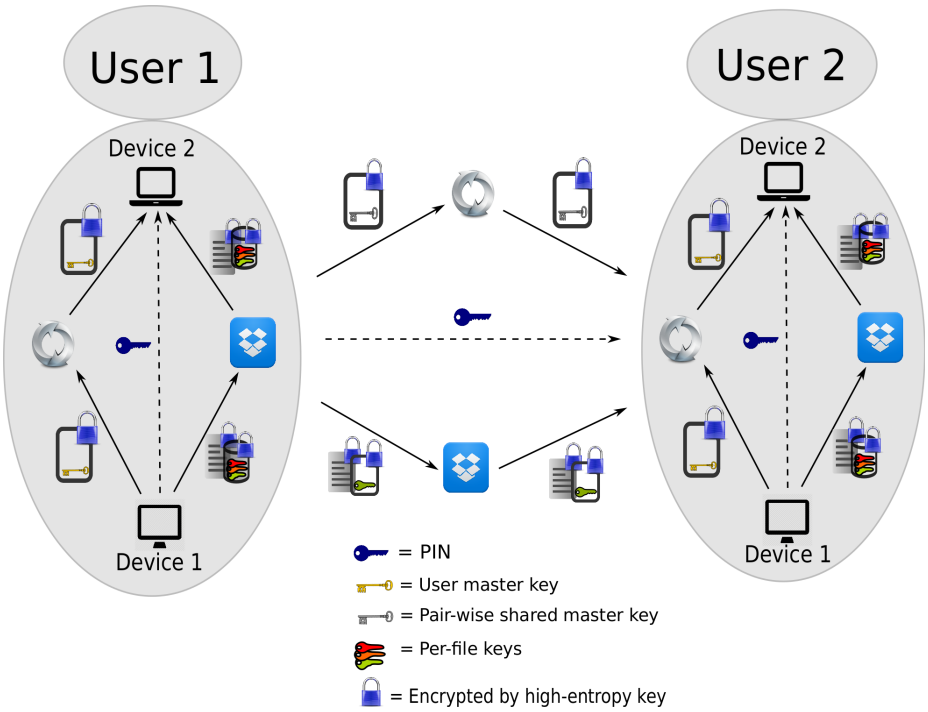

Figure 1: Keyfob overview: for a single user, the master key is transferred via Firefox Sync using a PIN, and other keys and encrypted files are synced through the cloud storage server to another user-owned device; and for multiple users, a shared master key is transferred via Firefox Sync using a PIN, and both the encrypted key and the encrypted file are sent through the cloud server. In both cases, a dotted arrow indicates out of band transmission. Each file is encrypted by a per-file key and the per-file key is encrypted by the application-specific key, which is derived from a master key.

focusing mainly on the cloud storage services. Our design supports other services (e.g., email) involving two users in a similar manner.

4.2 Key derivation

We choose to use key derivation to limit the number of keys needed to be stored. There is only one master key for each user. Table 1 defines the notation used.

We categorize the main features of cloud services into two groups: 1) features involving single user (e.g., file syncing) and 2) features involving two (or more) users

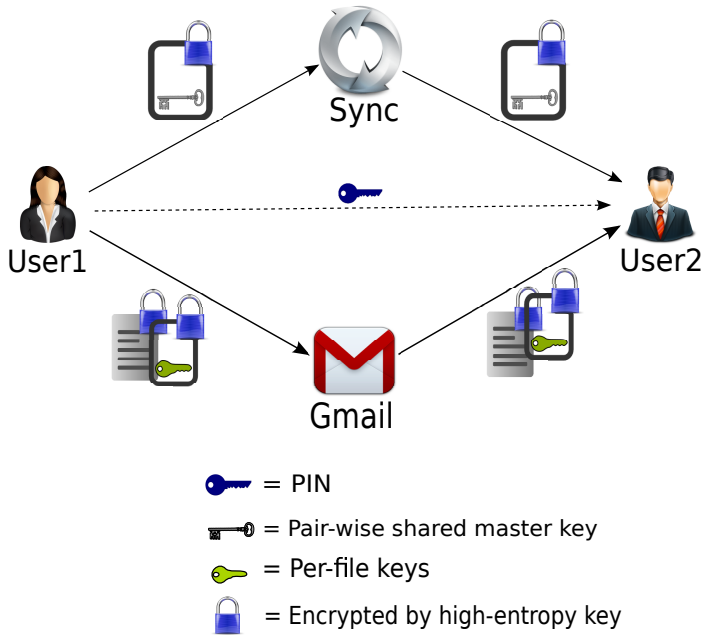

Figure 2: Keyfob overview for Gmail: a shared master key is transferred via Firefox Sync using a PIN, and both the encrypted key and the encrypted email are sent through the mail server. In both cases, a dotted arrow indicates out of band transmission. Each email is encrypted by a per-file key and the per-file key is encrypted by the application-specific key, which is derived from a master key.

(e.g., file sharing). Key derivation for both categories is handled similarly. The master key (K_{U_i}) for a user is randomly generated on the first run of Keyfob. Whenever a new file is added (or email is composed), a random key (K_{F_i}) is generated. K_{F_i} is used to encrypt $item_i$ (e.g., file or email). For the single-user case, at the first use of each application an app-specific master key (K_{A_i}) is derived where K_{A_i}= HKDF(K_{U_i}, App_i). Encryption keys (K_{F_i}) are protected by this K_{A_i}. For the two-user case, a shared master key ($K_{U_{ij}}$) between $User_i$ and $User_j$ is derived as $K_{U_{ij}}$= HKDF(K_{U_i}, $User_i \parallel User_j$), where $User_i$ is the initiator. App-specific master key ($K_{A_{ij}}$) is calculated as $K_{A_{ij}}$= HKDF($K_{U_{ij}}$, App_i). $K_{A_{ij}}$ is further used to encrypt any random keys between $User_i$ and $User_j$. This key derivation requires one-time key transfer of K_{U_i} for single user and one-time key transfer of $K_{U_{ij}}$ for a pair of users.

App_i	Name of application i (e.g., "Dropbox")
$User_i$	Name of user i (unique for a contact list)
$ChannelID$	As defined in Firefox Sync
$Secret_{ij}$	Pre-shared secret between $User_i$ and $User_j$
PIN	$(ChannelID \parallel Secret_{ij})$, where $\parallel$ represents string concatenation
K_S	Derived from a DH-EKE session key
K_{U_i}	Master key of $User_i$
K_{A_i}	Application specific key of $User_i$
K_{F_i}	Randomly generated encryption key for $item_i$
$\{X\}_K$	X encrypted by K using authenticated encryption
$K_{U_{ij}}$	Pair-wise sharing key between $User_i$ and $User_j$
$K_{A_{ij}}$	Application specific key between $User_i$ and $User_j$

Table 1: Notation used

4.3 Key transfer

Our key transfer process mainly consists of transferring K_{U_i} to all user-owned devices and sharing $K_{U_{ij}}$ with a peer user. We leverage Firefox Sync and DH-EKE for key transfer. We replace J-PAKE with DH-EKE as the key establishment protocol, as DH-EKE is simpler in terms of computational complexity and total steps in the protocol than J-PAKE. Details of DH-EKE and Firefox Sync are discussed in Section 2.1. Figure 3 shows the steps of establishing a DH-EKE session key for both single-user and two-user cases. This session key is then used for key transfer.

A PIN is the pre-shared secret, which is formed by concatenating the ChannelID and a $Secret_{ij}$. Firefox Sync generates the $Secret_{ij}$ randomly. We introduce user-chosen $Secret_{ij}$ as an option so that the users might share the $Secret_{ij}$ prior to the key transfer session. The PIN is shared with Device 2 through any other channel. Sharing secret over insecure channel is an open problem. We discuss existing methods in Section 2.1. Once the connection is established between two devices, Device 1 initiates the DH-EKE protocol to establish a session key (K_S).

Single-user. Figure 4 shows the steps of a single-user case, when peer-to-peer communication between user-owned devices is not possible. Device 1 sends master key

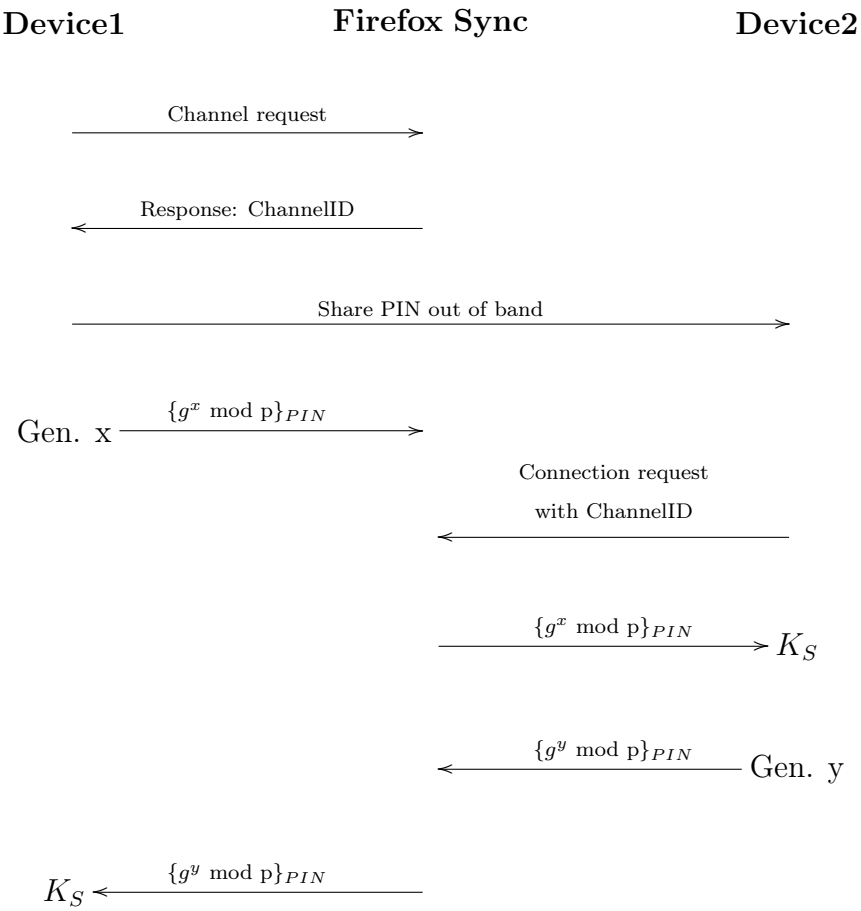

Figure 3: Key establishment protocol: session key establishment for both single-user and two-user cases. Here, $K_S = \text{KDF}(g^{xy} \bmod p)$.

(K_{U_i}) encrypted by session key over the same channel through the Firefox Sync server. As discussed in Section 4.2, for a single user, after having the master key, all other keys are derivable or retrievable. Device 2 next can derive the application specific key (K_{A_i}). All encrypted files and encrypted file keys are synced through the regular cloud service. So, Device 2 is able to decrypt all the files properly.

Two-user. Figure 5 shows the steps of a two-user case. Once a pair of users establish a session key (K_S), User1, having the master key ($K_{U_{ij}}$), sends it encrypted by the session key over the same channel through which the session key is established. From Section 4.2 we know that for multiple users after having $K_{U_{ij}}$ all other keys used

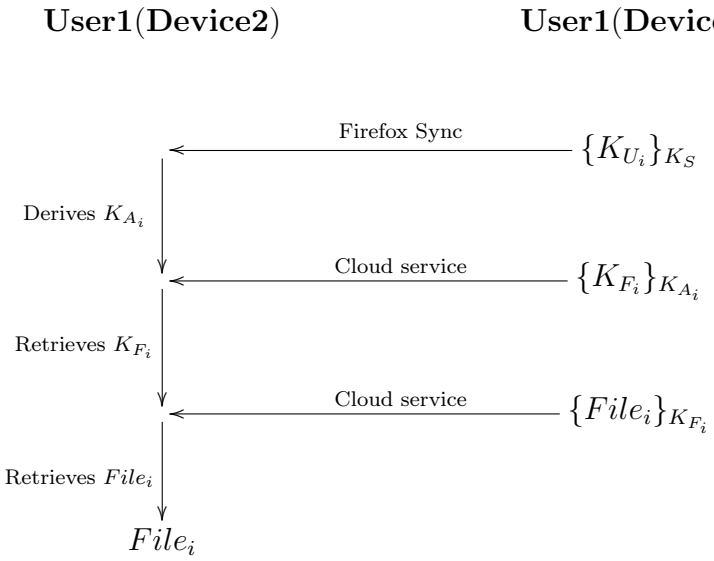

Figure 4: Key and data transfer protocol for the single-user case

between that pair are derivable or retrievable. User 2 next can derive the application specific key ($K_{A_{ij}}$). All encrypted files and encrypted file keys are synced through the regular cloud service. So, User 2 is able to decrypt all the files properly.

4.4 Key backup

Backing up of user master key (K_{U_i}) is essential in Keyfob. If a K_{U_i} is lost, then all encrypted contents of the corresponding user ($User_i$) becomes irretrievable. We lack an efficient way to store a cryptographic key offline. Not relying on any device limits the available options. Most existing solutions ask users to take a note or print of the key. Mercury [50] derives public/private key from user-selected content (i.e., private image, audio, video). A user backs up the file instead of the private key. Alternatively, it offers key generation from random seed. The random seed in this case can be converted into a QR code and printed for offline storage. Krivoruchko et al. [44] derives the RSA key from a user-chosen pass-phrase so that users can avoid storing the private keys. But, using pass-phrase may weaken the whole system. As

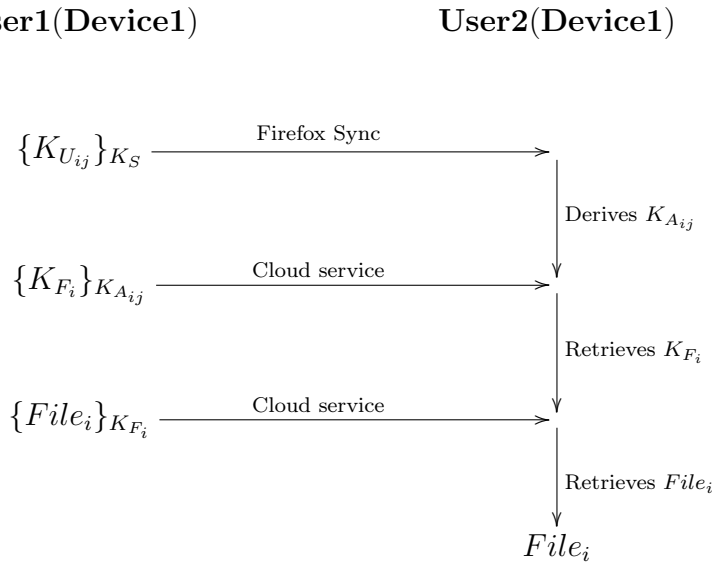

Figure 5: Key and data transfer protocol for the two-user case

Keyfob generates 256 bits long master keys, it is difficult for users to write them for backup without any mistake. S/Key [36] converts random bits into meaningful words. We adapt their approach and represent master keys as a collection of meaningful words so that user-mistake is less probable and also more detectable. Keyfob shows this converted master key in big fonts for user convenience in case the user wants to take a note. We offer users to take a print of the QR code of their master key to store somewhere secure. In case, the QR code can be scanned and the master key can be retrieved. As Keyfob derives pair-wise master keys ($K_{U_{ij}}$) from the identifier of the contacts and requires users to use the same identifier always for a specific peer, we feel that many users might want to store the contacts offline. In Keyfob, users can import their contacts into a file and later can refer to the file to use the same name.

4.5 Miscellaneous key management issues

Key recovery. Key recovery is one of the important features for any privacy-friendly system. Most systems depend on the password to recover the user data. In this work, we avoid any usage of password for recovery. We also notice that, a weak recovery mechanism can lead to undesired taking over of any account (see e.g., [38]). To avoid compromising with the security, Keyfob asks users to back up their own master key (K_{U_i}). Firefox Sync also required users to backup a master key to recover encrypted data. But, recently Sync has introduced a password-based key recovery mechanism, because the previous key recovery requirements were found unpopular to the users (see [80]). We acknowledge that the overhead of storing a master key only for the browser data might not be acceptable. In our case, we ask users to store a master key so that they can recover their data from different cloud services.

Note that we want a user to back up a single key (K_{U_i}) to recover all other keys including shared master keys. All keys required for the single-user case are derivable from K_{U_i}. So, as long as a user can present her K_{U_i}, she is able to retrieve all her data in different cloud services. Next, we discuss a key recovery mechanism for the two-user case. There are three possible scenarios, when a user might need to recover her shared master key:

1. All but one user-owned devices have lost the shared master key. Those devices can recover the key through Keyfob from the only device without requiring the user to input her master key.

2. All user-owned devices have lost the shared key ($K_{U_{ij}}$), but the peer has it. The user having the key can share it through Keyfob with the other peer. In such a case, the master key is transferred during a complete DH-EKE session through the Firefox Sync server from one device to another.

3. None of the users in a pair has the key. As the shared master key ($K_{U_{ij}}$) is derived from the master key (K_{U_i}) of the initiator and identifiers of both users, it is recoverable by the initiating user. So, we need to store the initiator's identity. At any point, the initiator can derive the key, and transfer it to her peer using a similar Keyfob session as the regular key transfer.

Key deletion & key reset. Our solution stores the master key in the local machine. In case of changing ownership, a user might want to delete the master key so that the new owner can not retrieve any of his contents. We provide an option to delete the key, even though it does not ensure complete unavailability of the key (see [63]). To reset a master key, Keyfob derives app-specific keys (K_{A_i}) for single-user case and shared master keys for two-user case from the new K_{U_i}; and re-encrypts all item-specific keys (K_{F_i}). This process does not require re-encryption of user data.

Chapter 5

Implementation

The major part of this thesis work is to implement the Keyfob tool and encryption plugins. In this chapter, we discuss the details of our implementation along with the challenges we faced during the implementation phase. Two major modules of our implementation are: the key transfer tool and encryption plugin. Keyfob is the key transfer tool, which is implemented as a Firefox extension for desktops and an Android app for mobile devices. There are three encryption plugins implemented in this work to offer end-to-end encryption for Dropbox and Gmail. To benefit a large number of users, we choose Windows and Android environments for the Dropbox plugin. This Dropbox plugin works with other cloud storage services (e.g., GoogleDrive and OneDrive), except the sharing feature, which requires service-specific API calls. The Gmail plugin is a Firefox extension, which works in a desktop environment.

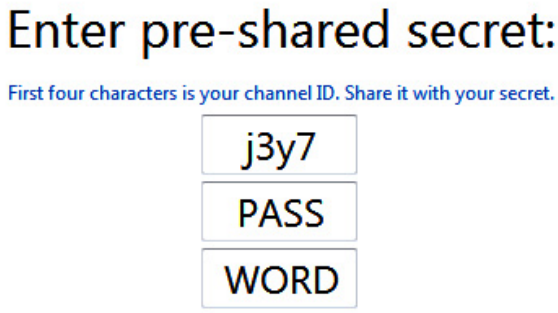

Figure 6: Keyfob UI: entering pre-shared secret while pairing a device

5.1 Keyfob implementation

5.1.1 Firefox extension for desktop

Keyfob is a Firefox extension, which is developed on top of the Firefox Sync source code. We take the *services/sync* module from *mozilla-central* [58], which is the main working branch of Mozilla Firefox. We want to use the code of "easy setup" (see [53]), which includes key establishment and key transfer in Firefox Sync. Note that Firefox Sync uses the "easy setup" only for the single-user case. While porting the code, we notice that there are dependencies between Sync and other modules. Our first task is to port all the necessary sources to make Sync working from the Keyfob extension. At the end, we imported 18 source files from 4 different modules. Note that we use the Firefox Sync server only for the key establishment. Unlike Firefox Sync, Keyfob does not require any registration. Keyfob stores all keys in a SQLite database ($keyDB$), which is populated with shared master keys and identifiers of peers.

5.1.1.1 Sending a key

To share a key between users and devices, a user initiates the process by selecting the *send key* option from the Keyfob menu. Under the *send key* option, a user is asked to input a name ($User_i$) as an identifier. In case of transferring own key, a

user inputs "me" as an identifier. Otherwise, the identifier is the receiver's $User_i$. Keyfob returns the key stored under that identifier. If the identifier is not found in $keyDB$, Keyfob generates a random key, and stores it into $keyDB$. Before the key establishment, Keyfob forms the PIN by concatenating 8-character long $Secret_{ij}$ (random or user-chosen) and 4-character long $ChannelID$. Once the session key is established, the master key (K_{U_i} or $K_{U_{ij}}$) encrypted by the session key is transferred via the same channel that establishes the session key. We generate a 256-bit random exponent by Mozilla crypto API [58]. Other parameters (generator and modulus) of DH-EKE are chosen from RFC 5114 [48]. We choose the bigInt.js [6] library for computing and sjcl.js [71] for encrypting messages for DH-EKE. To hide the fact that we are using DH-EKE instead of J-PAKE, we replace the g^{x1} value of J-PAKE with g^x and keep the format of the message similar as J-PAKE. To maintain the synchronization of messages transferred between a sender and a receiver, we follow Firefox Sync and build an asynchronous callback chain. Several times we got blocked (each time for 5-10 minutes) by the Firefox Sync server while implementing Keyfob. To test the connection with the Sync server and also to test Keyfob features, we created hundreds of connections without completing each session. We assume that highly frequent requests and incomplete sessions to the Sync server cause blocking (see [53]). A user should not face any blocking, as Keyfob does not issue incomplete sessions in the regular mode, and also our solution requires infrequent key transfer through the Sync server.

5.1.1.2 Receiving a key

In Keyfob a receiver is either another device of the same user, or a device from another user. At the very beginning, the receiver is asked to enter a name ($User_i$) as an identifier and pre-shared PIN; the PIN is sent to her out of band. In case of

transferring own key, a user inputs "me" as an identifier. Otherwise, the identifier is the sender's $User_i$. Then Keyfob extracts $ChannelID$ from the PIN and requests a connection from the Firefox Sync server. After that both parties can start exchanging DH-EKE messages to establish a session key (K_S). Thus at the receiver end, Keyfob gets K_{U_i} or $K_{U_{ij}}$ and stores the key under the identifier entered at the first step.

5.1.2 Android version of Keyfob

We intend to port the Keyfob extension for the Android environment. Firstly, we notice that Firefox introduces bootstrapped add-ons for the Android browser. As our desktop extension is an overlay-based add-on, simple porting is not possible from desktop to Android. Next, we identify that essential mozilla services (e.g., rest, util, constants) for Firefox Sync are unavailable in Firefox Android, which stops us building an extension with existing desktop implementation even by careful merging. Finally, we look into the Firefox Sync source code for Android from the "/mobile/android/base/sync" module under the mozilla-central code branch. Unlike desktop implementation, Firefox Sync for Android is developed in Java. This observation redirects us to import the Firefox Sync implementation for Android to an Android app. We implement Keyfob as an Android app, where the key establishment and UI are mostly ported from the Firefox Sync source. Unavailibility of mozilla crypto service in Java increases the size of the code for Android.

5.2 Dropbox plugin

5.2.1 User experience

Figures 7, 8, 9, 10, 12 illustrate required user steps for different use cases. Figure 7 shows the step of setting up Keyfob in the first device for Dropbox. To use our

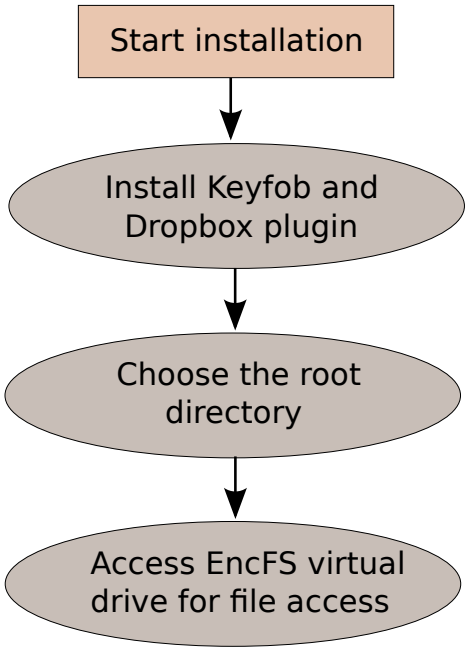

Figure 7: Use case: to start installation

Dropbox plugin, a user must also install the Keyfob Firefox extension. On the first run of the Dropbox plugin, a user is asked to choose the paths where the encrypted files and the plaintext files will be stored. We do not force users to encrypt everything in their Dropbox storage. The user-chosen root directory for the encrypted files must reside in a Dropbox local folder, so that the files get uploaded to the Dropbox cloud storage server by the Dropbox desktop client. The user accesses plaintext files from the EncFS [32] virtual drive.

The steps of adding a new device is illustrated in Figure 8 and Figure 9. To add a device, the user selects the "Send key" option from an existing device using the Keyfob extension; enters the identifier as "me"; and either notes the PIN generated with the random $Secret_{ij}$ or enters the pre-shared password with the channel ID depending on the user choice.

From the new device, the user must use the Keyfob extension; select the "Receive key" option; enter the identifier as "me"; and input the PIN to sync her master key.

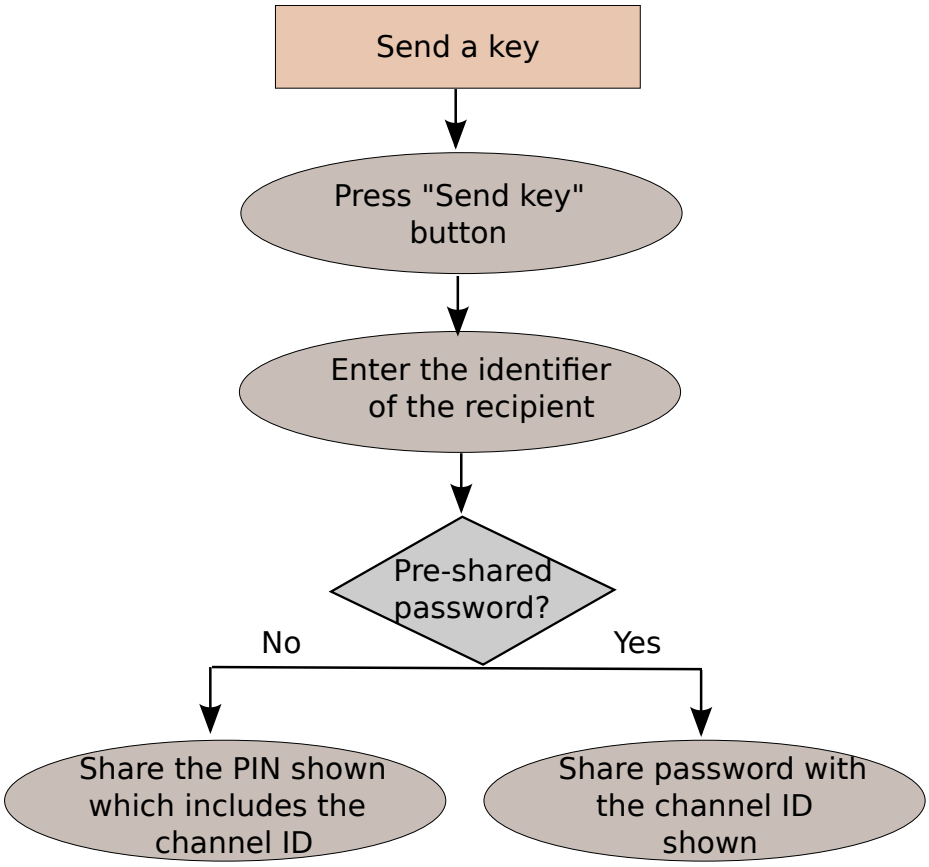

Figure 8: Use case: to send a key to other device or user

To share an encrypted file with a peer, the peer must have the Keyfob extension and our Dropbox plugin installed. The user experience to a share key is almost the same as adding own devices. In this case, users enter $User_i$ of the receiver. To share a file with a peer (see Figure 10), the user selects the file; chooses the "Share with Keyfob" option from the right click menu (same as Figure 11); and shares the generated file URL with the intended peer (e.g., via email). To receive a file (see Figure 12), the user downloads the file from the given URL; stores the file in the EncFS virtual drive; chooses the "Shared with Keyfob" option from the right click menu selecting the file; and enters the identifier of the sender. Our Dropbox plugin then retrieves the encryption key from the shared key file, and stores it in the local key database so that the file is considered same as other user-owned files.

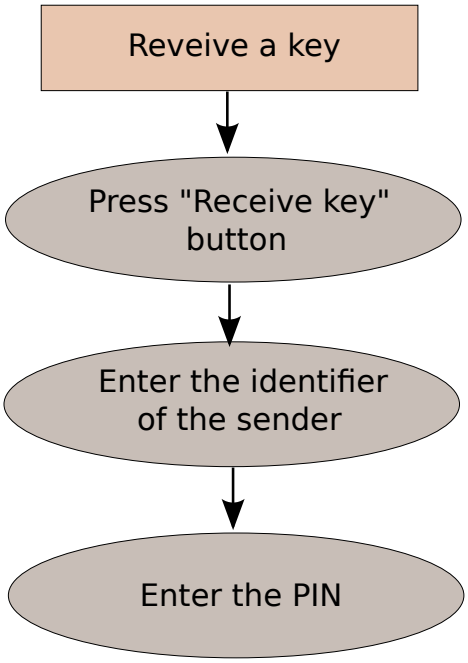

Figure 9: Use case: to add a new device at the receiver end

5.2.2 Desktop plugin

Our Windows encryption plugin is built on top of EncFS 1.7.4, which is an encrypted virtual filesystem in user-space originally implemented for Linux. We use encfs4win [84] (open-source EncFS project for Windows) for our implementation. Main dependencies of EncFS are FUSE [26], RLog [64], OpenSSL [59] and Boost [10] libraries. We modified EncFS significantly in terms of user experience, encryption configuration and key handling. A user-chosen password is not required, so we remove password prompts. Apart from password, EncFS originally asks users for encryption configurations (e.g., encryption algorithm, key size, block size) on the first run. Majority of everyday users lack in-depth security knowledge and may be unable to choose these configurations in a proper manner. By default, we set AES-CBC with 256-bit key. There is an option to change the encryption configuration later. We exclude filename encryption from our current implementation. EncFS originally stores all encryption configurations including encryption key protected by a password

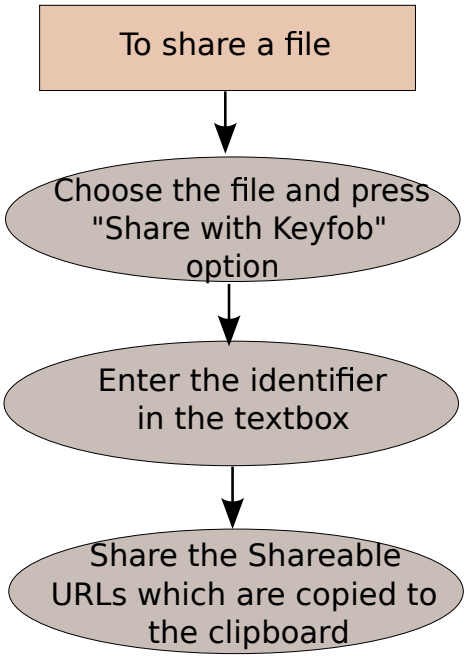

Figure 10: Use case: to share a file

in a configuration file generated by the Boost serialization library. We keep this file containing all configurations except the key. Unlike EncFS, our plugin generates per-file encryption keys; encrypts them with app-specific key (K_{A_i}); and stores them in a SQLite database ($keyDB$). Encrypted files and $keyDB$ reside in a Dropbox directory, so that they are automatically synced to all user-owned devices. The master key, which decrypts per-file encryption keys, is transferred through Keyfob. Each time a user wants to access files, she must access the EncFS virtual drive instead of a Dropbox local directory.

In Windows, we follow the regular Dropbox desktop client to provide almost the same user experience while sharing files with peers. To access the user Dropbox folder, our app must authenticates and takes user permission. A successful authentication process provides an access token, which is used for later access to the Dropbox folder. Note that our app has access only to the user-chosen root directory at the very first step in a user's Dropbox storage, which contains encrypted files generated by our

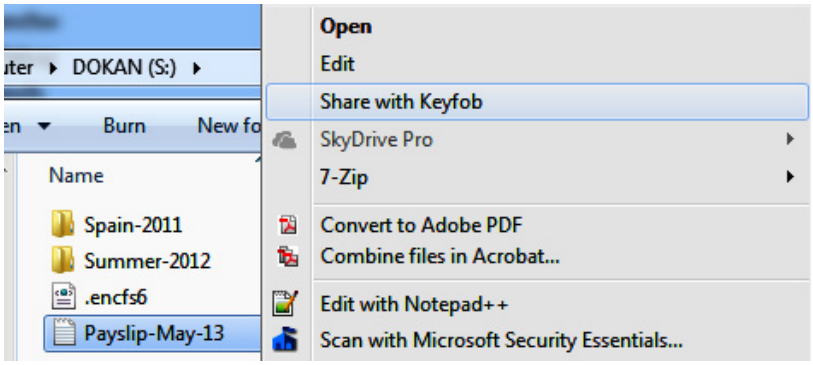

Figure 11: Keyfob UI: selecting file for sharing

Dropbox plugin. We add the "Share" option on the right click menu which triggers the Dropbox API (createShareableUrl) [18] call to generate a shareable link for the file, which is copied to the clipboard by our plugin. Peers can access the encrypted file using the URL and download it to their encrypted virtual drive. A user provides the identifier of the sender of the file. The shared master key is transferred through Keyfob and stored in *keyDB* under the same identifier. The plugin fetches the encryption key from *keyDB* and decrypts the file.

5.2.3 Android plugin

Our Dropbox plugin is an Android app implemented on top of Cryptonite [67] (open-source EncFS project for Android), which contains the EncFS 1.7.4 source code along with all dependency libraries. Cryptonite implements their own user interface for Android. One of the major changes we made in desktop plugin is UI-related e.g., disabling password prompts, fixed the encryption configuration for the first time. So, a simple swapping of the EncFS module did not work. Also, we notice that the EncFS module in Android contains additional code under the flag "ANDROID". We merge our modified EncFS for desktop and the EncFS module in Cryptonite carefully so that no Android-specific code is affected. Figures 13, 14, 15 and 16 are few screenshots of our Android plugin.

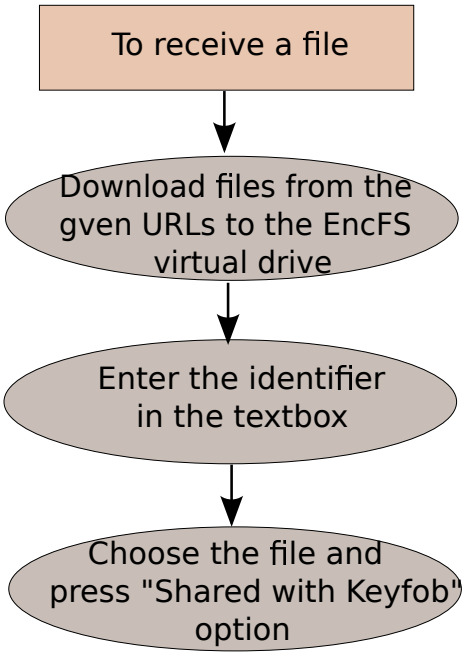

Figure 12: Use case: to receive a shared file

5.3 Gmail plugin

5.3.1 User experience

We require users to use the Keyfob extension and our Gmail encryption plugin to get the encrypted email support in Gmail's web interface. If the shared master key is not already transferred to the receiver side, a sender must use Keyfob to transfer the shared master key with the receiver. To send an encrypted email (see Figure 20), a user chooses the "Secret Compose" (see Figure 17) button instead of the regular "Compose" button.

Figure 18 shows a compose window, which looks the same except it contains a "Secret send" button. A user has the same experience while composing the email. Once the "Secret send" button is pressed the encrypted message (see Figure 19) and the encrypted key are sent to the recipient. Except the key transfer steps through Keyfob, we offer almost the same experience as the regular Gmail. At the recipient

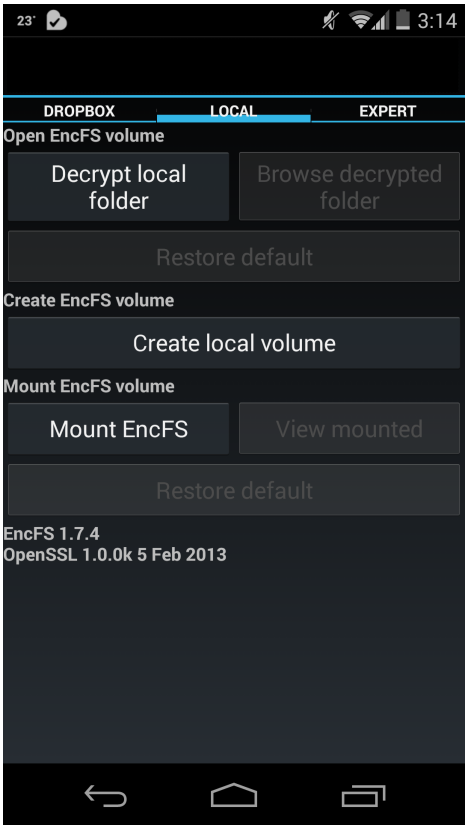

Figure 13: Android plugin UI: home screen

end, the encrypted email appears as the same as other regular emails. The receiver reads the decrypted email without any additional effort.

5.3.2 Implementation details

We build our Gmail encryption plugin on top of FriendlyMail [61], which is a Firefox extension offering encrypted Gmail service with key transfer through Facebook. The source of the project is publicly available. As we have Keyfob to transfer the master key to the receiver side, we do not require any trusted third party (e.g., Facebook) to transfer encryption keys. So, we exclude the key transfer implementation of FriendlyMail in our Gmail plugin. Each email is encrypted using a 128-bit random encryption key. For encryption and random key generation, we use the sjcl.js library [71]. We

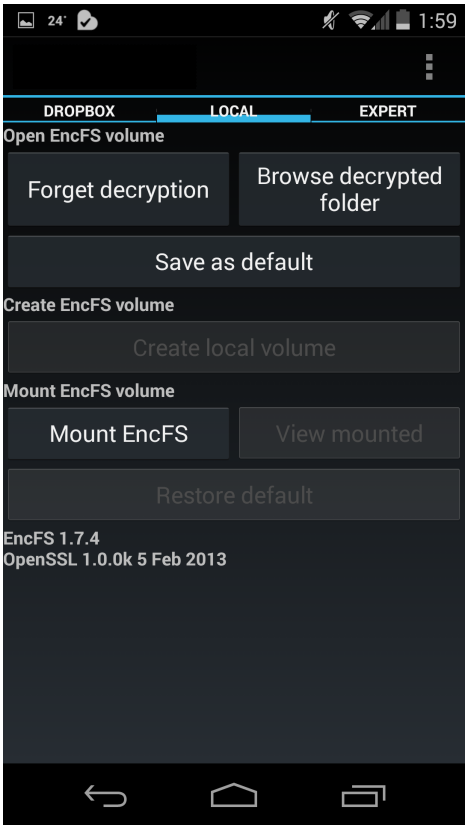

Figure 14: Android plugin UI: encrypted directory

extract the recipient email address to fetch the corresponding shared key from the key database. Then using HKDF, we derive the Gmail-specific key ($K_{A_{ij}}$) to encrypt the per-email encryption keys. We append the encrypted per-email key at the end of the encrypted message with an appropriate flag so that we can identify the key data portion in the encrypted message. Figure 19 shows an encrypted email produced by our plugin. For message encryption, we use authenticated encryption offered by the sjcl.js library so that we can verify the integrity of the encrypted message at the recipient side. At the receiver side, we extract the encrypted key from the whole message. We also collect the sender's email address from the message body, and fetch the corresponding shared master key from the key database. The shared master key reveals the per-email key, which is used to decrypt the email message. The whole encryption or decryption process does not involve the user. Currently we do

Figure 15: Android plugin UI: uploading file from local directories to the encrypted directory

not support encryption for attachments or emails to multiple recipients. To support multiple email addresses of a $User_i$, we allow users to update their existing contact information to add two or more email addresses.

Figure 16: Android plugin UI: encrypted directory

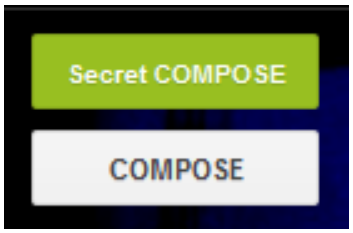

Figure 17: Gmail plugin UI: "Secret compose" button

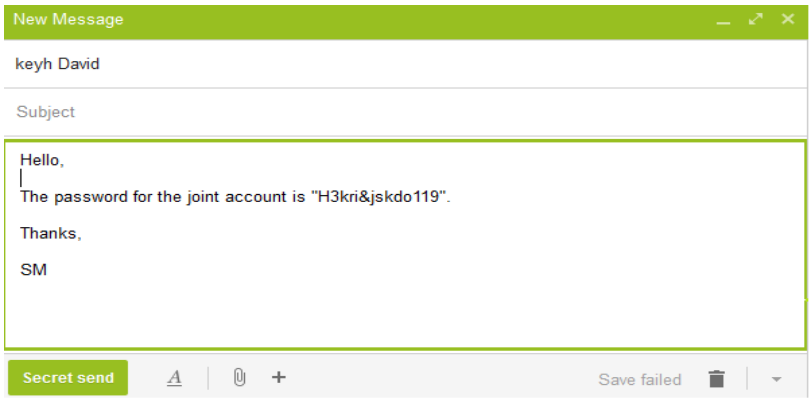

Figure 18: Gmail plugin UI: A plaintext Gmail message

```
********** Start of protected message **********
eyJpdiI6IjZycXV1ZDhZUzhPTkhkYUVXWjA3Z2c9PSIsInYiOjEsIml0ZXIiOjEwMDAsImtzI
joxMjgsInRzIjo2NCwibW9kZSI6ImNjbSIsImFkYXRhIjoiIiwiY2lwaGVyIjoiYWVzIiwiY3
QiOiJDSWE1UkxFMU95M0VSZzV4RXo5SVNsTnp0dz09In0=
********** End of protected message **********

:::::::::::::::::::: start of key data ::::::::::::::::::::
DkN4FcJeinqi2jUBDwIYoA==
:::::::::::::::::::: end of key data ::::::::::::::::::::
```

Figure 19: Gmail plugin UI: Sample encrypted Gmail message

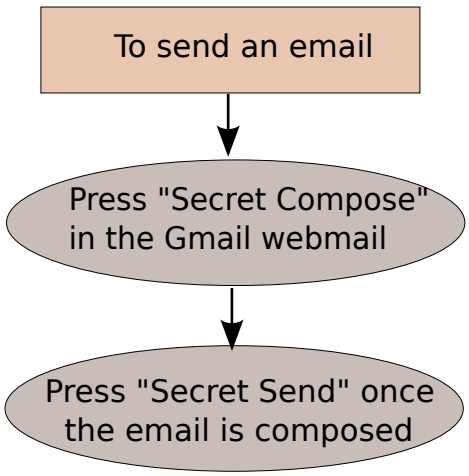

Figure 20: Use case: to send an email

Chapter 6

Security Analysis

In this chapter, we discuss the threats that are addressed by Keyfob and also the ones that we can not address. In this work, we mainly consider the threats from the curious and malicious providers. We also cover the threats from the eavesdropper and man-in-the-middle.

6.1 Threats from malicious service providers

We consider the Firefox Sync server (or any alternative implementation of it), cloud services (e.g., Dropbox, Gmail) and network providers (e.g., ISPs) as service providers. None of these providers can access plaintext user content or keys in Keyfob. User content is encrypted by unique random keys; service providers see only encrypted content and encrypted keys, and master keys are transferred through DH-EKE, which prevents offline attacks on the PIN. All keys are long (e.g., 128 bits), and generated from a good RNG source or derived from a random key. Thus, brute-forcing a key is considered infeasible.

6.1.1 Online guessing of PIN

First, we consider that the attacker does not know the ChannelID. If a wrong PIN is entered, both parties end up with two different session keys. According to our current implementation of DH-EKE during the challenge response phase, the enter of wrong PIN is identified and the session is aborted with an exception message (jpkae.error.keymismatch [53]). So, to discard a single password from the dictionary, the attacker must exchange all DH-EKE messages with a legitimate user. This requirement makes consecutive try difficult. If an attacker knows the ChannelID and wants to guess the $Secret_{ij}$, then the attacker must exchange all DH-EKE messages, which includes 6 HTTP GET requests, to know the outcome of a guess. The Firefox Sync server deletes the ChannelID after 6 successful GETs. So, the attacker only gets one chance to guess the PIN with a known ChannelID.

6.1.2 Leaked Channel IDs

Currently, ChannelIDs are deleted after 6 successful GETs, or after 5 minutes by the Sync server. Consider a scenario, where a ChannelID is leaked and Firefox intentionally stops deleting ChannelIDs. It lets an attacker to try all possibilities of the random part of a PIN. However, due to the use of DH-EKE, the attacker cannot verify his guesses offline. So, it is not possible to brute force the PIN without brute forcing the long DH-EKE session key.

6.1.3 Leaked PIN

Exposure of a PIN before key establishment allows a man-in-the-middle attack for both random and user-chosen $Secret_{ij}$. If the PIN is leaked after the DH-EKE session key is established, then we have the following two cases. When a PIN is generated with a unique random $Secret_{ij}$, deriving the session key from the leaked PIN and

encrypted messages is infeasible. However, when a user-chosen password is used for PIN generation in several DH-EKE sessions, an exposed password can lead to man-in-the-middle attacks for subsequent key establishment sessions; in this case, an attacker also must know the per-session ChannelID.

6.1.4 Eavesdropping out-of-band channels

As soon as a ChannelID is requested from the Sync server, all out-of-band channels may be eavesdropped to get the PIN for that session, assuming the PIN will be shared by such a channel in real-time (i.e., the PIN is generated using a random $Secret_{ij}$, not a pre-shared password). Here, an attacker's goal is to impersonate a receiving user. To target a specific user, the attacker must correlate two users who are trying to establish a DH-EKE session with a given ChannelID, using IP addresses or other user-identifying information. We recommend using a pre-shared $Secret_{ij}$ (preferably shared via an in-person meeting) to generate the PIN; see also Section 2.1 or use a channel that is difficult to attribute to specific user (e.g., SMS from a phone with a cash-purchased SIM card, Tor hidden services).

6.1.5 Impersonation attack

In Keyfob, a user, who initiates a session, is responsible to share the PIN with the other peer in case they do not have any pre-shared $Secret_{ij}$. A possible attack scenario is that Eve, pretending to be Bob, initiates and completes a key establishment session with Alice. At the first step, Eve transfers a PIN to Alice as Bob through an out-of-band channel (e.g., phone call, SMS). According to our assumption, both parties at least know each other's contact numbers. With this assumption, the above mentioned attack is feasible if an attacker has physical access to Bob's phone or if the mobile network operators eavesdrop. If the service provider impersonates any of the users,

the scenario will be the same as above except the provider already knows the channel ID. But, still the provider must know the PIN. Note that sharing a secret over an insecure channel (i.e., without integrity protection) is yet an open problem. We discuss existing and possible solutions to this problem in Section 2.1. If users have a pre-shared $Secret_{ij}$, then Eve's only option is to launch an online guess attack on the pre-shared $Secret_{ij}$.

6.2 Other threats

Leaked master key. If the K_{U_i} of $user_i$ is leaked, then all data individually owned by $user_i$ will be exposed. Additionally, the shared data, which is encrypted by the keys derived from the leaked K_{U_i}, can be also uncovered.

Brute-force key. We do not send any password protected messages to cloud service providers. All user files and key files are encrypted by high-entropy keys (256 bit), so exhaustive key search is possibly infeasible.

Data leakage. The example of public access to the sensitive user data due to the vulnerabilities in the design of the cloud services is also available (see e.g., [39]). In Keyfob all user files and emails are encrypted before being sent to the cloud server. So, users no longer need to rely on the provider for data leakage. User data is no longer subject to be leaked, given that the encryption algorithm does not have any backdoor (e.g., [73]).

Key manipulation. Encrypted key files are stored at the cloud server. By modifying the keys in key files, an attacker can form a DoS attack by restricting users from retrieving original data, but can not retrieve as well.

Collusion attack. Few prior works (e.g., [61, 21]) rely on multiple parties to transfer encrypted data and encryption keys. In those solutions, if relying parties collude, or

a government on top of them collects data from them, then user data will be leaked. Our system does not send encryption keys unencrypted out of a user machine. We also do not rely on any trusted party. So, collusion attacks are no longer possible.

6.3 Limitations

Any threats from a malicious browser extension or malware in the local machine is beyond the scope of this work. Also, we do not hide content metadata (e.g., filename, encryption algorithm, email header parameters) from the service providers. Our Gmail plugin currently does not encrypt email attachments. Keyfob does not achieve forward secrecy and non-repudiation, as it uses the shared key for content encryption, instead of the DH-EKE session key.

Malicious browser extensions. The Keyfob browser extension accesses a local key database, where master keys are stored. That allows other browser extensions to access local files and retrieve master keys. We argue that if malicious extensions have access to local files, they are able to read all files in a local drive. They do not need keys anymore.

Threats on metadata. For cloud storage, filenames remain as plaintext in our current implementation, as we use full filepath including filenames to map per-file encryption key. We acknowledge that filenames can let a provider know the possession of certain files. Other file metadata (e.g., filesize, timestamps) is also exposed to providers. Also, encryption configurations e.g., key size, block size, mode of operation are stored in plaintext, and not verified upon retrieval. Any modifications of these parameters can cause DoS. For web-email, header parameters (e.g., `From:`, `To:`, `Subject:`) are also exposed to email providers.

Several serious privacy threats from metadata are known (see e.g., [76, 23]). There

also exist few solutions to hide the metadata that Keyfob exposes. EncFS originally offers filename encryption. Changing in the Keyfob design will allow to adopt filename encryption. Several PGP and S/MIME based solutions (e.g., [20, 33, 51]) offer attachment encryption for emails. The Darkmail technical alliance [16] is a recent proposal to radically change the current email infrastructure to enable private email communications. One of the major goals of this project is to hide content metadata. Full details of this proposal is still unavailable, as of writing.

Chapter 7

Discussion

This chapter includes the discussion on usability and deployability issues for adopting Keyfob.

7.1 User steps & software requirements

	Features		
	No registration	No password	No configuration
Firefox Sync			✓
EncFS	✓		
Keyfob*	✓	✓	✓

Table 2: Comparing Firefox Sync and EncFS with Keyfob*. Keyfob* represents the Keyfob extension and our encryption plugins together. Here, empty box indicates the stated feature is not offered.

7.1.1 User steps

We leverage Firefox Sync and EncFS to build Keyfob and the Dropbox plugin. For rest of the report, we use Keyfob* to represent Keyfob and our encryption plugins together. Table 2 illustrates the modifications we added in terms of user experience. Firefox Sync requires user registration. Even though Keyfob establishes session keys and transfers master keys through the Firefox Sync server, Keyfob simplifies user experience by removing the user registration requirement. EncFS and Firefox Sync both require a separate password. EncFS uses the user-chosen password to encrypt the master key and Firefox Sync authenticates a user with the password. We exclude the requirement of both passwords. We protect the master key with a session key and authenticate a user with the pre-shared PIN, so we exclude all password prompts from our implementations. On the first run, EncFS lets a user to choose the encryption configuration (e.g., encryption algorithm, key size, and block size). We assume choosing encryption configuration on the first run inconvenient for most users and remove from our implementation (Section 5.2 lists default configuration). There is a provision of changing the encryption configuration later.

	Total # of		
	Auth. steps	Client app(s)	Master key(s)
Dropbox	1/1	1/3	0/1
Gmail	1/1	0/2	0/1
Dropbox + Gmail	2/2	1/4	0/1

Table 3: Comparing total number of operations and resources required for regular cloud services (e.g., Dropbox and Gmail) and our proposal for single-user in the format r/e, where r is the regular mode of operation and e is the encrypted mode. Here, the encrypted mode includes Keyfob*.

7.1.2 Software requirements

Table 3 compares user experience between the regular cloud services and our implementation. Keyfob does not introduce any additional authentication. So, a user faces only regular authentication steps from Dropbox and Gmail. For regular Dropbox, a user installs a client app to use Dropbox in a desktop environment. For Gmail, we consider its web interface. So, we consider that it does not require any client application in the regular mode. Being application-independent, the Keyfob extension lets our solution require a single browser extension to facilitate key transfer for different cloud services. The number of required client app grows with the encryption plug-ins, which support encryption for different services. To offer encrypted cloud storage and email services using Gmail and Dropbox, we need 4 client apps in total, which includes the regular Dropbox desktop client and Keyfob*. The regular mode of cloud services does not need any master key. Our solution demands users to store and manage (e.g., backup) a master key, which is later used to derive all other keys. We keep Keyfob independent of any encryption plugin in a sense that the key establishment and encryption can be performed in either order. Encrypted email can be sent before encryption key is transferred. Of course, the receiver would require encryption keys to decrypt an email.

7.1.3 Usability issues

Usability issues in our solution include the following. We fix few of them in our implementation.

Random PIN vs. pre-shared password. Originally in Firefox Sync, a PIN is randomly generated at the beginning of a key transfer. So, the users are unable to share the PIN in advance. To allow the users sharing a $Secret_{ij}$ during any prior meeting or communication, we introduce user-chosen $Secret_{ij}$ as another option (see

Figure 6).

Restrictions on changing filepath. As we store the per-file key mapped with the filepath, any change in the location of the encrypted files from Dropbox's web interface would fail our encryption plugin. We plan to develop an encryption plugin for the Dropbox web interface in the future.

Offline key back up. A user must store the master key (K_{U_i}) offline to recover all her contents. Also, to derive the shared master key ($K_{U_{ij}}$), an initiator needs to present her own master key. In Section 4.4, we discuss few possible ways to back up a key offline.

Service-specific intermediate application. For each cloud service, users are required to use a separate encryption plugin. Web interfaces for several cloud services can be possibly facilitated in a single browser extension.

Increased number of keys. To facilitate sharing, symmetric-key based system increases the number of keys significantly ($O(n^2)$). Whereas, the public key based solutions can restrict this number to $O(n)$. Keyfob only requires the receiver side to back up the shared key to avoid asking the sender to transfer the shared key ($K_{U_{ij}}$) via a new DH-EKE session. The sender side can derive the shared key from the master key (K_{U_i}). Thus we half the total number of keys required to be stored.

7.2 Deployability

To transfer the master key, we utilize the Firefox Sync server. Although the server allocates a channel for each session for Keyfob and forwards "easy setup" messages, the load on the server increases very slowly as key transfer is not a frequent event in our system. Still, if Firefox decides to disallow such external requests or changes its design, we or anyone else can easily host a low-cost Sync server (discussed in Section 2.1).

Hosting our own server does not lose security, as the server is considered untrusted in our system. We calculate the total number of requests to the server considering that there are 100,000 Keyfob users having 2 devices on average and everyday there are 5,000 new users for Keyfob. Also, we assume that each user on every two days contact a new person, which requires them to establish a key. Each day, only new users use the server to add their second device. The first device does not communicate with the Firefox Sync server without the need of adding a new device or establishing a key with a new user. So, new users generate 5,000 requests to the server to add the second device. Among 100,000 Keyfob users, 50,000 users on average add a new person to their contacts everyday. So, to transfer keys among pair of users, Keyfob generates 50,000 requests to the Firefox Sync server daily. In total, the server will handle 55,000 DH-EKE requests per day to support Keyfob. During each DH-EKE session, a sender sends two messages (one is about 1,218 bytes long and another is about 616 bytes long), and a receiver sends a message of 1,218 bytes through the Sync server (see [55] for the message size). To process a DH-EKE session, the Sync server handles about 3 KB data. So, per day on average the server will process approximately 160.08 MB, which is affordable even by a low-cost server.

Deduplication is a very popular storage and bandwidth optimization mechanism, where a file is uploaded only if it is not already stored in the cloud server. As our solution encrypts files with random encryption keys, it cannot support cross-user deduplication. However, deduplication for the same user is partially possible. When the user changes an existing file then modified blocks are uploaded only if block-level deduplication is performed. Encrypted files and emails restrict service providers to benefit from content-aware advertising. However, non-targeted ads can be served.

Currently, offering encryption support for each cloud service requires an intermediate application to work as an encryption plugin for that service. Currently user

		Features							Attacks			Requirements		
		Syncing	Sharing	Client-side-encryption	Symmetric encryption only	Password-free	No trusted party	Auto-key-transfer	Curious provider	Malicious provider	Collusion	Pre-key-exchange	User-level keys	Key server
Services	Dropbox [19]	✓	✓						✗	✗				
	iCloud Keychain [3]	✓		✓		✓	✓	✓		✗		✗	✗	
	End-to-end [31]	✓	✓	✓			✓	✓		✗		✗	✗	
	PGP [13]		✓	✓		✓						✗	✗	
	Syme [74]		✓	✓		✓		✓		✗		✗	✗	
	FriendlyMail [61]	✓	✓	✓	✓	✓		✓			✗			
	CaaS [21]		✓	✓	✓	✓					✗			✗
	Safebox [66]	✓		✓	✓		✓	✓		✗			✗	
	Wuala [45]	✓	✓	✓	✓		✓	✓		✗			✗	
	BoxCryptor [11]	✓	✓	✓			✓	✓		✗		✗	✗	
	SpiderOak [69]	✓	✓	✓	✓		✓	✓		✗			✗	
	ARKpX [4]	✓	✓	✓	✓	✓	✓						✗	
	Keyfob*	✓	✓	✓	✓	✓	✓	✓					✗	✗

Table 4: Comparing different existing solutions. Here, a checkmark (✓) for features means the corresponding service offers that feature; a crossmark (✗) for attacks indicates that a specific attack is not addressed; and a crossmark (✗) for requirements marks that a specific requirement is needed. An empty box indicates the stated feature/requirement is not offered/not needed or the stated attack is prevented.

interface, APIs (if available) and offered features from different vendors for a specific cloud differ significantly; thus it might not be possible to support different vendors with a single encryption plugin. Our Dropbox plugin is implemented in such a way that it is usable with other popular services (e.g., GoogleDrive, OneDrive) for syncing without any change. For the sharing support they require additional implementation, as it involves service-specific API calls.

7.3 Comparison

Comparison among different approaches are presented in Table 4. The solutions having more checkmarks (✓) and less crossmarks (✗) are generally considered good, though each feature does not have equal importance. For example, ARKpX offers all features but *auto-key-transfer*, which we consider a major drawback. To compare existing services with Keyfob, we select 7 features, 3 attacks and 3 requirements. *Syncing* and *sharing* are very common features for cloud services. They allow users to transfer contents between multiple user-owned devices and between different users. *Client-side-encryption* is to offer encryption for any user-content leaving the user machine. We put *symmetric encryption only* as a feature to identify the services solely depending on symmetric encryption. All asymmetric-based solutions is rely on password protection for private keys or trust the provider not to change the public keys. The *password-free* feature includes the services that do not use user-chosen password for data confidentiality. If a service does not introduce any trusted third party (e.g., key server), then we check the *no trusted party* feature. The *auto-key-transfer* feature is marked when keys are transferred by the tool rather than the user. We consider that *auto-key-transfer* might need minimum user involvement (i.e., pressing few buttons, entering a password or name). Attacks from a *curious provider* include the access of user-data when it is stored in the clear. A *malicious provider* is one-step ahead, which may employ significant effort (e.g., brute forcing password, replacing public keys intentionally) compared to a curious provider to access user-data. Under the *collusion attack*, we consider the attacks possible when different providers collude, or user-data from different providers is collected. The requirements are considered as the overheads and limitations of a system. The services, having *pre-key-exchange* crossmarked, requires key exchange prior to any encrypted data transfer. The services with *user-level-keys* means that they create and manage key at the client

end, and requires user involvement. The *key server* is a trusted or semi-trusted third party key management service. Next, we discuss our analogy while rating different solutions in Table 4.

iCloud Keychain. iCloud Keychain [3] is a key backup mechanism for the single-user case only. So, it does not offer the sharing feature. iCloud offers end-to-end encryption rather than relying on the Apple server. Each registered device generates its own public/private key pair to encrypt data before uploading to the server. Although, iCloud requires user password to authenticate a new user, iCloud never uses passwords to derive encryption keys. As iCloud never exposes user private key to the server or any third party, iCloud does not require any trusted party. The Keychain tool transfers public keys to the server with the involvement of users such as authentication and approval. Both curious and malicious providers apparently can not access user data as long as they do not possess any device private key. Private keys never leave a client device. Even to modify the public keys, iCloud requires to sign with the corresponding private keys. But, recent revelations (discussed in Section 2.2.2 show that Apple or any third party is able to conduct surveillance over users. So, we consider iCloud to be vulnerable to the malicious provider. As iCloud does not rely on any trusted party, it is collusion-proof. Like any other public key based systems, to avail the end-to-end encryption service users must exchange public keys in iCloud Keychain. iCloud generates user keys at the client side without requiring any additional key server.

PGP. PGP is mainly designed for multi-party email communication. PGP does not offer syncing originally. Few recent PGP-based plugins (e.g., *end-to-end*) offer syncing by encrypting the private key by a user password. Here, we uncheck the *syncing* feature considering the original PGP proposal. PGP provides end-to-end encryption between users, where each user has his/her own public/private key pair.

PGP does not involve user password in generating keys. Rather than relying on a trusted party, PGP creates *web of trust* among known users to certify the authenticity of public keys. Failing to create a trust path between users may cause the creation of fake public keys (cf. examples of fake PGP keys [2, 14]). High dependency on the trust model lets us uncheck the *no trusted party* feature. The key transfer in PGP is non-trivial and involves users. So, we consider that PGP is missing the *auto-key-transfer* feature. The *web of trust* mechanism in PGP helps to avoid the dependence on a trusted provider or third party e.g., certification authorities, and prevents threats from malicious providers. Like most public-key-based systems, PGP requires *pre-key-exchange* to share encrypted emails. Additionally, PGP generates public/private key pairs at the client end without requiring any key server.

Syme. Syme is a privacy-friendly group conversation service. Currently, Syme does not support multi-device syncing. Although, Syme uses symmetric keys for encryption, there is a public/private keypair per user to protect encryption keys. All encryption keys are generated randomly, and none is generated from user password. During key establishment period, Syme relies on the server to keep the integrity of the ephemeral keys. So, our rating considers that Syme is missing the *no trusted party* feature. As keys are generated and managed by the Syme client, the *auto-key-transfer* is checked. For the possibility of faking the ephemeral keys by a provider, our evaluation marks Syme as vulnerable to the malicious provider. Syme requires *pre-key-exchange* and *user-level-keys*.

FriendlyMail. FriendlyMail offers encrypted email service with the Gmail web service. Encryption keys are transferred through Facebook as a private message. From any device users can access emails by logging into Gmail and Facebook. So, we check *syncing*. FriendlyMail only generates random high-entropy symmetric keys. Facebook

is trusted for not disclosing encryption keys to Google or the government. *Auto-key-transfer* is offered requiring login to Facebook. Any provider alone possibly can not breach confidentiality of this system. But, as soon as Google and Facebook colludes, or the government of the same jurisdiction collects data from both, the user data is leaked. FriendlyMail does not demand any of the requirement listed in the table.

CaaS. CaaS is a third party key management service, which provides and stores encryption keys for shared data between users. As CaaS is not targeted for syncing keys between user devices, we uncheck *syncing*. CaaS facilitates end-to-end encryption between users. User password is only used to authenticate users to the server. Encryption keys are stored at the CaaS server considering the server trusted. CaaS does not require any key transfer, as encryption keys are not generated at the client end. Like FriendlyMail, any provider including the CaaS server can not retrieve user data. But, CaaS is vulnerable to collusion attack. CaaS is of the very few proposals, which require a key server.

Keyfob. Keyfob is designed to offer end-to-end encryption for different cloud based services. The single user case and two-user case in Keyfob cover *syncing* and *sharing* respectively. For encryption, Keyfob totally depends on high-entropy symmetric keys. Even though we use PINs during the key transfer, DH-EKE invalidates any offline attack on the PIN. During the key transfer phase, the Firefox Sync server is used without trusting for data confidentiality. Also, Keyfob offers the *auto-key-transfer* involving users minimally i.e., entering password and pressing few buttons. By achieving all the features in the list, Keyfob possibly bypasses all three attacks listed in the table. Not involving user password and trusted third party apparently avoids threats from the malicious provider. The *no trusted party* feature allows Keyfob to be collusion-proof. Keyfob achieves all the features, and possibly avoids all

the attacks enlisted with the price of two major requirements. Keyfob keys are generated, maintained and stored at the client end. To recover user data, backing up the master key (K_{U_i}) is mandatory. Additionally, Keyfob needs a server to conduct the key transfer. Unlike most other approaches, the server in Keyfob is untrusted for data confidentiality.

Other approaches. We include Dropbox in our comparison to see where the most popular cloud services (e.g., Gmail, Facebook, OneDrive) stand. Google's *end-to-end* is a JavaScript implementation of OpenPGP, we consider it separately. *End-to-end* uses symmetric encryption for content encryption. But, we uncheck the *Symmetric encryption* field, as it leverages public key systems for key transfer. Unlike PGP, *End-to-end* offers syncing between devices. To support syncing, *End-to-end* involves user-chosen pass-phrases to encrypt private keys. There are other password-based solutions. Wuala and SpiderOak derives the master key from the user-chosen password. BoxCryptor transfers the private key to multiple user machines by protecting it by the user password. ARKpX relies on the user to transfer keys. It does not offer automatic key transfer.

All approaches except Keyfob and ARKpX either use public key system, password-based symmetric-key, or a trusted third party. ARKpX does not offer the key transfer feature, and users are responsible for key transfer. We notice that all public key based systems that offer syncing is password dependent, as they transfer private keys protected by a user password. Except Keyfob, we found CaaS and Syme password-free. The systems (e.g., Caas, Syme), which rely on a trusted party (e.g., third party or service provider), are vulnerable to either malicious provider or collusion attack. Ignoring recent revelations of iCloud vulnerabilities in forensic services, iCloud Keychain is able to protect public key from integrity loss. But, iCloud requires as many keys as any symmetric-encryption-based approaches, even though iCloud uses

public key systems. Comparing with others Keyfob outstands by providing all listed features, protecting from main attacks, and requiring less infrastructure.

7.4 Future work

Sharing a secret over an insecure channel is an open problem. We identify finding an efficient way to share secrets as a potential future work. In this thesis work, we consider the Tor hidden services [77] to share secrets over insecure channel. This work is ongoing. We assume that the reader is familiar with Tor (see [78] for details).

Tor hidden services allow users to offer different services hiding their locations. The steps of registering and offering/accessing a hidden service are:

1. To start a hidden service, Alice selects few nodes from the Tor circuit as introductory points (IP), generates a public key, and derives the onion address from the public key. Rest of this section refers an introductory point of a Tor circuit using IP.

2. Alice advertises her IPs, onion address and public key to the global Tor hidden service database.

3. Bob inquires Alice's IPs and public key using the onion address.

4. Bob sends a one-time secret and a rendezvous point (RP) protected by Alice's public key to one of the Alice's IPs.

5. Alice presents the one-time secret to Bob's RP. Thus, a Tor circuit is built via Bob's RP.

6. Further communication continues using this circuit.

To share a secret, we may leverage this infrastructure and build a circuit between two parties before sending the secret. We identify several assumptions so that we can simplify the problem. Our assumptions are:

1. IDs (onion addresses) and introductory points of friends are available without any integrity loss. Apparently, this is the most challenging part to achieve. We discuss challenges in details later of this section. Email address is one choice as ID. In this case, we do not need confidentiality.

2. All messages are sent over mixed network.

3. No relay including introductory points will be repeated for two consecutive runs for the same source-destination pair.

Figure 21: Secret sharing using Tor hidden service, Here, HS(A) and HS(B) are the hidden services of Alice, and Bob and IP(A) and IP(B) are introductory points of Alice and Bob respectively. Also, $package_A = id_A \parallel cookie_A \parallel rp_A \parallel pubKey_A$.

Next, we discuss the steps (see Figure 21) to share a secret between two users. If Alice wants to share a secret with Bob, then

1. Bob will derive an onion address from his email address and try to connect to that address as a client to check whether a hidden service with this onion address already exists. If this request fails, then Bob knows that this onion address is free and he registers his hidden service with that address.

2. Alice sends a nonce and a package to Bob's onion address using Bob's IPs. The package consists of her rendez-vous point (rp_A), one time secret ($cookie_A$) and

public key ($pubKey_A$). We assume that Alice can derive Bob's onion address from Bob's email address, which is assumed to be available to Alice and retrieve introductory points. She sends this first message through Bob's introductory point (IP(B)).

3. After getting the first message, Bob does not know from whom he gets it. So, Bob keeps the nonce part as it is and encrypts the rest part with his key (key_B) and broadcasts to the hidden services of his all contacts.

4. Bob's friends who have not initiated any session will ignore the second message. Others will decrypt the first part of the message with their keys and verify with their nonce. Alice will find a match and then remove her key from the rest part of the message. Now, the package is only encrypted by Bob's key. Alice sends this message to Bob's hidden service.

5. Bob can decrypt the third message and retrieve rp_A, $cookie_A$ and $pubKey_A$. Further communication with Alice will be conducted through rp_A encrypted by $pubKey_A$.

This approach challenges us with following issues:

1. An onion address should be derived from a publicly available ID (i.e., email address) so that peers can derive the address without requiring any additional information. Also, user email address is required to be reached to all peers without any integrity loss.

2. If a publicly available ID is used, then one can open a hidden service using another one's ID. Eve can open a hidden service with Bob's email address before Bob. Our current proposal can not stop Eve, but Bob is notified that already a hidden service is open with his email address. Any link with the email ownership might make the task of eve more difficult.

3. If we consider the Tor hidden service database untrusted, then we can not store the public key in the clear any more.

Also, we need to consider about different attack points to address any MiTM.

Introductory point. Eve can reside in the network as an introductory point. She can intercept the first message and play the role of Bob. She prepares the second message with her key (key_E) and broadcasts to all Bob's contacts, which we assume is not a secret. Alice removes her key from the third message and sends back to Bob's hidden service. Now, the third message is the package encrypted by Eve's key. But, Eve will not get the message in step 3, because this time Alice will use another introductory point. Bob will receive this message and decrypt. As the sender ID is not valid, he will ignore this message.

Attacker friend. As all contacts of Alice will receive the first message, we consider that any of her friends might be malicious. A malicious friend can truncate the first message and encrypt with her own key (key_E). It will result an invalid user ID at Bob's end. So, it will not help the attacker.

The protocol discussed in this section requires further improvement.

Chapter 8

Conclusion

Few large for-profit corporations (e.g., Google, Facebook) are now brokering most user communications, and effectively in control of most stored user data. Besides, governments are accessing cloud-stored user data in bulk with questionable/shadowy legal apparatus, ignoring century-old privacy rights. Designing privacy-preserving mechanisms for existing cloud-based services, specifically a key management mechanism to facilitate crypto support in this ecosystem appears to be quite challenging. An obvious reason is the availability of several diverse features in cloud services that are difficult to accommodate with a privacy tool; such features include: syncing in multiple user-owned devices, online backup, sharing of stored data with peer users, online group editing of a shared document, and searching within plaintext user data. In this work, we proposed Keyfob, a key manager for end-to-end encryption in the cloud ecosystem, and leverage Keyfob to offer privacy-friendly cloud services (e.g., cloud storage, email). We are yet to cover few popular features (e.g., online editing). We acknowledge that user-experience-based features of Keyfob will be required to be validated by a user study. The design of Keyfob and implementation of Keyfob plugins for different services highlight challenges in less-explored symmetric-key approaches. Our hope is that Keyfob will motivate more researchers in solving the long-standing

problem of enabling end-to-end encryption in a cloud-based environment, especially when large-scale surveillance programs are targeting *all* user data, irrespective of any wrong-doing or nationality.

Bibliography

[1] C. Alexander and I. Goldberg. Improved user authentication in off-the-record messaging. In *WPES*, pages 41–47, 2007.

[2] G. Andresen. It ain't me! I've got a PGP imposter. Blog article (Mar. 21, 2014). `http://gavintech.blogspot.ch/2014/03/it-aint-me-ive-got-pgp-imposter.html`.

[3] Apple Inc. iCloud. `https://www.icloud.com`.

[4] ARKpX Inc. Official webpage. `http://arkpx.com`.

[5] N. Asokan, A. Dmitrienko, M. Nagy, E. Reshetova, A.-R. Sadeghi, T. Schneider, and S. Stelle. CrowdShare: Secure mobile resource sharing. In *Applied Cryptography and Network Security*, pages 432–440, 2013.

[6] L. Baird. Big integers in JavaScript. `http://www.leemon.com/crypto/BigInt.html`.

[7] S. M. Bellovin and M. Merritt. Encrypted key exchange: Password-based protocols secure against dictionary attacks. In *IEEE Symposium on Security and Privacy*, Oakland, CA, USA, May 1992.

[8] S. M. Bellovin and M. Merritt. Augmented encrypted key exchange: a password-based protocol secure against dictionary attacks and password file compromise. In *ACM Computer and Communications Security (CCS'93)*, pages 244–250. ACM, 1993.

[9] C. H. Bennett and G. Brassard. Quantum cryptography: Public key distribution and coin tossing. In *Proceedings of IEEE International Conference on Computers, Systems and Signal Processing*, volume 175. New York, 1984.

[10] Boost. Boost project website. `http://www.boost.org`.

[11] Boxcryptor. Official webpage. `http://boxcryptor.com`.

[12] D. Callahan. Firefox Sync's new security model. (Apr. 30, 2014). `https://blog.mozilla.org/services/2014/04/30/firefox-syncs-new-security-model`.

[13] J. Callas, L. Donnerhacke, H. Finney, D. Shaw, and R. Thayer. OpenPGP message format, Nov. 2007. RFC 4880.

[14] E. Clark. Another fake key for my email address. Tor talk (Mar. 9, 2014). https://lists.torproject.org/pipermail/tor-talk/2014-March/032308.html.

[15] CloudOne Services. Official webpage. https://www.cloudoneservices.com/privatesky.

[16] Dark Mail. Dark Mail Technical Alliance. https://www.darkmail.info.

[17] C. Doctorow. Kafka, meet Orwell: Lavabit's founder explains why he shut down his company. Blog article (May 20, 2014). http://boingboing.net.

[18] Dropbox. Dropbox core API SDK 0-SNAPSHOT API. http://dropbox.github.io/dropbox-sdk-java/api-docs/v1.7.x.

[19] Dropbox Inc. Official webpage. https://www.dropbox.com.

[20] Enigmail. Enigmail project. https://www.enigmail.net.

[21] S. Fahl, M. Harbach, T. Muders, and M. Smith. Confidentiality as a service–usable security for the cloud. In *TrustCom*, pages 153–162, 2012.

[22] M. Farb, Y.-H. Lin, T. H.-J. Kim, J. McCune, and A. Perrig. Safeslinger: easy-to-use and secure public-key exchange. In *Proceedings of the 19th annual international conference on Mobile computing & networking*, pages 417–428. ACM, 2013.

[23] L. Ferran. Ex-NSA chief: 'we kill people based on metadata'. ABC news blogs (May 12, 2014). http://abcnews.go.com/blogs/headlines/2014/05/ex-nsa-chief-we-kill-people-based-on-metadata.

[24] Forbes.com. The NSA's slideshow explaining its PRISM surveillance program. http://www.forbes.com/pictures/efdk45edd/prism-slide-4.

[25] Forbes.com. The Snowden Effect: Yahoo to join Gmail in offering users end-To-End encryption. Tech blog (Aug. 7, 2014). http://www.forbes.com/sites/kashmirhill/2014/08/07/yahoo-end-to-end-encryption.

[26] FUSE. Official webpage. http://fuse.sourceforge.net.

[27] R. Gallagher and G. Greenwald. How the NSA plans to infect 'millions' of computers with malware. News article (Mar. 12, 2014). https://firstlook.org.

[28] S. L. Garfinkel, D. Margrave, J. I. Schiller, E. Nordlander, and R. C. Miller. How to make secure email easier to use. In *Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 701–710. ACM, 2005.

[29] L. Gong. Optimal authentification protocols resistant to password guessing attacks. In *Computer Security Foundations Workshop, 1995. Proceedings., Eighth IEEE*, pages 24–29. IEEE, 1995.

[30] L. Gong, M. A. Lomas, R. M. Needham, and J. H. Saltzer. Protecting poorly chosen secrets from guessing attacks. *Selected Areas in Communications, IEEE Journal on*, 11(5):648–656, 1993.

[31] Google Inc. End-To-End. A Chrome extension `https://code.google.com/p/end-to-end`.

[32] V. Gough. EncFS Encrypted Filesystem. `http://www.arg0.net/encfs`.

[33] gpg4usb. gpg4usb project. `http://www.gpg4usb.org`.

[34] P. Gutmann and I. Grigg. Security usability. *Security & Privacy, IEEE*, 3(4):56–58, 2005.

[35] P. Hallam-Baker. Privacy protected email. In *A W3C/IAB workshop on Strengthening the Internet Against Pervasive Monitoring (STRINT)*, London, UK, 2014.

[36] N. Haller. The S/KEY one-time password system, 1995. RFC 1760.

[37] F. Hao and P. Ryan. J-PAKE: authenticated key exchange without PKI. *Transactions on computational science XI*, 6480:192–206, 2010.

[38] M. Honan. How Apple and Amazon security flaws led to my epic hacking. `http://www.wired.com/2012/08/apple-amazon-mat-honan-hacking/all`.

[39] Intralinks Inc. Your Sensitive Information Could Be at Risk: File Sync and Share Security Issue . `http://collaboristablog.com`.

[40] D. P. Jablon. Strong password-only authenticated key exchange. *ACM SIGCOMM Computer Communication Review*, 26(5):5–26, 1996.

[41] D. P. Jablon. Extended password key exchange protocols immune to dictionary attack. In *Sixth IEEE Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises, 1997.*, pages 248–255. IEEE, 1997.

[42] Y. Jiangshan, C. Vincent, and R. Mark. Challenges with end-to-end email encryption. In *STRINT'14*, 2014.

[43] Y. Kim, A. Perrig, and G. Tsudik. Group key agreement efficient in communication. *Computers, IEEE Transactions on*, 53(7):905–921, 2004.

[44] T. Krivoruchko, J. Diamond, and J. Hooper. Storing rsa private keys in your head. In *Dependable Computing, 2006. PRDC'06. 12th Pacific Rim International Symposium on*, pages 129–138. IEEE, 2006.

[45] LaCie AG. Wuala webpage. https://www.wuala.com.

[46] A. P. Lambert, S. M. Bezek, and K. G. Karahalios. Waterhouse: enabling secure e-mail with social networking. In *CHI'09*, Boston, MA, USA, Apr. 2009.

[47] Lavabit LLC. Official webpage. https://www.lavabit.com.

[48] M. Lepinski and S. Kent. Additional Diffie-Hellman groups for use with IETF standards. http://tools.ietf.org/html/rfc5114.

[49] S. Lucks. Open key exchange: How to defeat dictionary attacks without encrypting public keys. In *Security Protocols*, pages 79–90. Springer, 1998.

[50] M. Mannan, D. Barrera, C. D. Brown, D. Lie, and P. C. V. Oorschot. Mercury: Recovering forgotten passwords using personal devices. In *Financial Cryptography and Data Security*, pages 315–330. Springer, 2012.

[51] Microsoft Outlook. Encrypt email messages. http://office.microsoft.com/en-ca/outlook-help/encrypt-email-messages-HP010355559.aspx.

[52] miniLock. Official webpage. http://minilock.io.

[53] Mozilla Foundation. Easy setup. https://docs.services.mozilla.com/keyexchange.

[54] Mozilla Foundation. How to update to the new Firefox Sync. https://support.mozilla.org/en-US/kb/how-to-update-to-the-new-firefox-sync.

[55] Mozilla Foundation. J-PAKE password-authenticated key exchange (pure-python library). https://github.com/warner/python-jpake.

[56] Mozilla Foundation. Keep your Firefox in sync. http://www.mozilla.org/en-US/firefox/sync.

[57] Mozilla Foundation. Run your own Sync-1.5 server. https://docs.services.mozilla.com.

[58] Mozilla Foundation. Shared knowledge for the open web. https://developer.mozilla.org.

[59] OpenSSL. Project website. http://www.openssl.org.

[60] S. Patel. Number theoretic attacks on secure password schemes. In *IEEE Symposium on Security and Privacy*, pages 236–247, 1997.

[61] A. S. Pirouz, V. Rabotka, and M. Mannan. FriendlyMail: Confidential and verified emails among friends. Technical report (2013). http://spectrum.library.concordia.ca/978331.

[62] B. Ramsdell and S. Turner. Secure/multipurpose internet mail extensions (S/MIME) version 3.2 message specification, Jan. 2010. RFC 5751 (Standards Track).

[63] J. Reardon, D. Basin, and S. Capkun. Sok: Secure data deletion. In *IEEE Symposium on Security and Privacy*, pages 301–315, 2013.

[64] RLog. Official website. `http://www.arg0.net/rlog`.

[65] M. D. Ryan. Enhanced certificate transparency and end-to-end encrypted mail. In *Network and Distributed System Security Symposium (NDSS'14)*, 2014.

[66] Safebox. Official webpage. `http://safeboxapp.com`.

[67] C. Schmidthieber. Cryptonite: EncFS and TrueCrypt on Android. `http://code.google.com/p/cryptonite`.

[68] Silent Circle. Official webpage. https://silentcircle.com.

[69] SpiderOak. Official webpage. `https://spideroak.com`.

[70] Squelch Design. New tool reveals how bad people are at choosing passwords. `http://squelchdesign.com/web-design-newbury/new-tool-reveals-how-bad-people-are-at-choosing-passwords/`.

[71] E. Stark, M. Hamburg, and D. Boneh. Symmetric cryptography in JavaScript. In *ACSAC*, 2009.

[72] T. Straub. *Usability challenges of PKI*. PhD thesis, TU Darmstadt, 2006.

[73] N. Sullivan. How the NSA (may have) put a backdoor in RSA's cryptography: A technical primer. Blog article (Jan. 5, 2014). `http://threatpost.com/en_us/blogs/rsa-securid-attack-was-phishing-excel-spreadsheet-040111`.

[74] Syme Inc. Syme webpage. `https://getsyme.com`.

[75] TheGuardian.com. NSA Prism program taps in to user data of Apple, Google and others. News article (Jun. 7, 2013). `http://www.theguardian.com/world/2013/jun/06/us-tech-giants-nsa-data`.

[76] TheGuardian.com. NSA stores metadata of millions of web users for up to a year, secret files show. Guardian news archive (from September 30, 2013) `http://www.theguardian.com/world/2013/sep/30/nsa-americans-metadata-year-documents`.

[77] Torproject.org. Tor: Hidden Service Protocol, 2014. `https://www.torproject.org/docs/hidden-services.html.en`.

[78] Torproject.org. Tor project webpage, 2014. `https://www.torproject.org`.

[79] Trrst. Trrst official webpage. https://www.trsst.com.

[80] B. Warner. The new sync protocol. https://blog.mozilla.org/warner/2014/05/23/the-new-sync-protocol/.

[81] A. Whitten and J. D. Tygar. Why Johnny can't encrypt: A usability evaluation of PGP 5.0. In *USENIX Security Symposium*, Washington, DC, USA, Aug. 1999.

[82] T. D. Wu. The secure remote password protocol. In *Network and Distributed System Security Symposium (NDSS'98)*, 1998.

[83] J. Zdziarski. Identifying back doors, attack points, and surveillance mechanisms in iOS devices. *Digital Investigation*, 11(1):3–19, 2014.

[84] F. Ziglio. EncFS for Windows project page. http://members.ferrara.linux.it/freddy77/encfs.html.