Aalto University
School of Science
Degree Programme in Computer Science and Engineering

Nguyen Hoang Long

Securely accessing encrypted cloud storage from multiple devices

Master's Thesis
Espoo, October 2, 2015

Supervisor: Professor N.Asokan, Aalto University
Advisor: Sandeep Tamrakar M.Sc. (Tech.)

Aalto University
School of Science
Degree Programme in Computer Science and Engineering

ABSTRACT OF
MASTER'S THESIS

Author:	Nguyen Hoang Long		
Title:	Securely accessing encrypted cloud storage from multiple devices		
Date:	October 2, 2015	**Pages:**	97
Major:	Data Communication Software	**Code:**	T-110
Supervisor:	Professor N.Asokan		
Advisor:	Sandeep Tamrakar M.Sc. (Tech.)		

Cloud storage services like Dropbox, Google Drive and OneDrive are increasingly popular. They allow users to synchronize and access data from multiple devices.

However, privacy of cloud data is a concern. Encrypting data on client-side before uploading it to cloud storage is an effective way to ensure data privacy. To allow data access from multiple devices, current solutions derive the encryption keys solely from user-chosen passwords which result in low entropy keys.

In this thesis, we present OmniShare, the first scheme to allow client-side encryption with high-entropy keys combined with an intuitive key distribution mechanism enabling data access from multiple devices. It uses a combination of out-of-band channels and cloud storage as a communication channel to ensure minimal and consistent user actions during key distribution. Furthermore, OmniShare allows the possibility of reducing communication overhead for updating encrypted data. OmniShare is freely available on popular platforms.

Keywords:	Cloud storage, Key distribution
Language:	English

Acknowledgements

I wish to express my deepest gratitude to my supervisor, Professor N. Asokan for his guidance and insightful comments. Similarly, I thank Sandeep Tamrakar, my advisor, for his valuable time and patience with this thesis.

I am grateful to Mihai Bucicoiu from TU Darmstadt for his early development of OmniShare. I am also thankful to him for many valuable discussions that helps me understand and enrich my ideas.

I am also indebted to the members of the Secure Systems research group with whom I have interacted during my thesis. Particularly, I would like to acknowledge Dr. Hien Truong, Dr. Andrew Paverd, Thanh Bui, Jian Liu and Swapnil Udar for many valuable discussions that helped me understand my research area better.

Most importantly, none of this would have been possible without the love and patience of my family. I would like to express my heart-felt gratitude to my family and especially my fiance, Thanh Tran, who has aided and encouraged me throughout this endeavor.

Espoo, October 2, 2015

Nguyen Hoang Long

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Motivation

Over the last few years, cloud computing has become more and more popular. It brings revolutionary innovation with regards to cost, resource management and utilization. Cloud computing offers nearly unlimited resources, highly reliable on-demand services with minimal infrastructure and operational cost. For these reasons, we now see the wide use of cloud computing in organizations as well as by end users.

As a part of cloud computing, cloud storage services are becoming more and more popular. These services provide users a way to store their data on central servers instead of their local storage. Cloud storage brings many advantages — They allow users to synchronize and access their data files from multiple client devices. Additionally, they allow multiple clients to collaborate on a single file.

At the same time, there is a privacy concern for data stored remotely on cloud storage. While using cloud storage services, users have to trust the service providers for security and privacy of their data. Anyone with access, legitimate or otherwise, to the storage providers' servers will be able to read or modify data without being noticed. Mainstream services like Dropbox [3], OneDrive [8] or Google Drive [5] may encrypt users' data on server side so there is no way for users to know when and how their data is protected. There is also a possibility that these services get compromised or deliberately give away users' data to unauthorized third parties. Client-side encryption is an effective way of preserving users' data privacy. By encrypting users' data locally before uploading to cloud storage, privacy concern about data stored remotely on the cloud can be minimized.

Over the last decade, a wide range of personal cloud storage services,

including the ones which offer client-side encryption, have emerged. Services such as Wuala [18] or SpiderOak [14] attract millions of users by these features. However, they are incompatible with other popular services, like Dropbox which offer much more storage capacity and features such as collaboration with other users or deep integration with multiple operating systems. Other solutions such as Boxcryptor [1] or Viivo [17] allow users to easily create encrypted files and synchronize seamlessly via Drobox, OneDrive or Google Drive. They provide applications for multiple platforms which offer strong client-side encryption. Users download the applications, install them and register for accounts using credentials such as usernames and user-passwords. These applications uses the user-passwords as secret inputs for their encryption scheme. As a result, the privacy of users' data can be guaranteed as long as the password is safe.

Using passwords to encrypt data is extremely common nowadays due to its simplicity. It is an easy way for people to access their encrypted data on cloud storage from any device of their choice. However, people might pick low-entropy passwords. They might reuse or even write them down somewhere. One solution for this problem is to replace passwords with strong cryptographic keys. Nonetheless, this approach brings inconvenience to users since they have to carry the keys and exchange them manually among devices in order to access their encrypted data from multiple devices.

From the service providers' point of view, client-side encryption causes another problem. The purpose of encryption is to produce a randomized output, even when the inputs are fully or partly identical. Therefore, whenever there is a change in a file, for example, when a user edits a file, the updated file needs to be re-encrypted and re-uploaded. Similarly, two identical files will be uploaded twice since their encryption are totally different. This also causes problems for clients — Clients have to download and re-synchronize updated files again across all of their devices. Such problems consume huge amount of network bandwidth, download/upload time and server's storage capacity.

To our knowledge, there has been no single complete solution that offers client-side encryption with secure, easy-to-use key distribution mechanism, works seamlessly with existing cloud storage services, and consumes minimal network bandwidth as well as storage capacity.

1.2 Structure of the thesis

The rest of this thesis is divided into eight chapters. Chapter 2 describes relevant background knowledge. Chapter 3 presents problem statement and

requirements of the thesis. Chapter 4 and chapter 5 present the design and implementation of our solution respectively. In chapter 6, we conducted a thorough analysis of security, performance and usability of the solution. Chapter 7 presents various existing client side encrypted cloud storage products. Finally, Chapter 8 provides a summary of the thesis and suggests potential future work.

Chapter 2

Background

This chapter presents different topics and concepts that set the background for our thesis.

2.1 Cloud storage

Basically, cloud storage systems are networks of data centers which use cloud computing technology and provide interfaces for storing data [24]. They also provide applications to interact with store data. Cloud storage services allow users to access their stored data from anywhere at any time over the Internet. They also enable users to share data with others. OneDrive, Dropbox and Google Drive are among the most popular cloud storage providers available today.

However, one major problem of cloud storage services is data security and privacy. By uploading (sensitive) information to cloud storage, users have to trust the storage providers for protecting their data. Nonetheless, cloud providers can peek into data or transfer data to other parties in certain circumstances (subpoena for example). Cloud servers can also get compromised [1], resulting in exposure of users' sensitive data. It also can be the consequence of poor security design, e.g. before September 2013, Google did not encrypt data while moving among data centers [2].

[1] http://en.wikipedia.org/wiki/2014_celebrity_photo_hack

[2] http://www.washingtonpost.com/business/technology/google-encrypts-data-amid-backlash-against-nsa-spying/2013/09/06/9acc3c20-1722-11e3-a2ec-b47e45e6f8ef_story.html

2.2 Client-side encryption

There are different mechanisms that can be implemented to enhance security of data stored on cloud storage. Companies can rely on building private/hybrid cloud architecture where data are stored entirely/partially in their local infrastructure hence greatly enhancing security. However, this approach is too expensive for small/medium organizations and especially for normal users.

A more common approach is to encrypt data locally in client devices (client-side encryption) before uploading to the cloud storage. The cloud storage is used to store, synchronize and distribute the encrypted data across multiple devices. The encryption keys are only distributed to devices which users intend to access data from. Those devices form a *domain of devices* which users authorize to access their encrypted data. This way, users retain control over their encrypted data and make sure that no one except the key holders (i.e. devices in the domain) can access the encrypted data.

There are numerous applications providing client-side encryption such as BoxCryptor [1], SpiderOak [14] or TrueCrypt [16]. They share similar core functions (encrypt/decrypt using keys derived from users' password) but are very different on architecture and user experience. Chapter 7 provides a more detailed look on those client-side encryption solutions.

2.3 Message authentication code

A *message authentication code* (MAC) is a short value used to ensure integrity and authenticity of a message. One way to construct a MAC value is to use a *cryptographic hash function*.

In cryptography, a *cryptographic hash function* is an one-way function which takes an arbitrary length message as an input and produce a fixed length output called *hash* [55]. Hash function maps the message input to the output in a way that all outputs are equiprobable and random. Constructing a hash from a given message is easy while it is extremely difficult to reconstruct the input from a given hash. For this reason, hash function is used as a way to ensure integrity of the input message. SHA-2 [52] is one of the most widely used set of cryptographic hash function.

A MAC algorithm, which is constructed using cryptographic hash function is also called a *keyed hash function*. A keyed hash function takes two different inputs, a message and a secret key to produce a fixed length output. It is practically impossible to produce the same MAC given the algorithm and the input message without knowing the secret key.

2.4 Encryption scheme

In [38], Goldreich stated that an encryption scheme is a protocol that allow two parties to *securely* communicate over an insecure channel. The communication channel can be tapped by an *adversary*. An encryption scheme consists two algorithms. The sender uses an *encryption* algorithm to transform the secret message, which is called the *plaintext*, to a *ciphertext* which does not leak any information about the plaintext and sends the ciphertext to the receiver. The receiver applies a *decryption* algorithm to the ciphertext to recover the original plaintext. In order for this scheme to be secure, the receiver must know an input to the decryption algorithm called the *decryption key* that is unavailable to the adversary. On the other hand, the sender must provide the encryption algorithm with an auxiliary input parameter, called the *encryption key*, that relates to the decryption key. There are two types of encryption schemes: *symmetric key encryption* and *public-key encryption*.

2.4.1 Symmetric key encryption

In symmetric key encryption, the encryption key and the decryption key are the same. For this reason, the adversary must not learn about the encryption key. Consequently, there is a problem of how two parties (sender and receiver) can agree on a same secret key over an insecure communication channel. This is called the *key distribution* problem. There are many widely used symmetric encryption algorithms such as AES [29] and 3DES [19].

Symmetric-key encryption can either use stream cipher or block cipher. A stream cipher is an algorithm which encrypts one bit of the plaintext at a time with a corresponding bit of the key to get one bit of the ciphertext. Meanwhile, block cipher takes a specific number (*block size*) of bits of the plaintext and encrypts them as a single unit. To do so, a block cipher encryption algorithm splits the plaintext into multiple *blocks* and encrypts each block using a block cipher *mode of operation*, e.g. CFB, CBC, GCM [44].

Authenticated encryption

Authenticated Encryption (AE) is a block cipher mode of operation which provides confidentiality, integrity and authenticity of the data. The encryption function using AE mode takes a plaintext message, a secret key and optionally a plaintext header. The corresponding output includes a ciphertext and a MAC. The optional plaintext header is not encrypted but is used to produce the MAC value. On the other hand, the decryption function takes the ciphertext, the secret key and the header as inputs to produce

the plaintext output. ISO/IEC 19772:2009 [44] suggests the use of authenticated encryption mode such as Galois/Counter Mode (GCM) or Counter with CBC-MAC (CCM).

2.4.2 Public-key encryption

In public-key encryption scheme, also known as *asymmetric encryption*, the encryption key does not need to be secret. Therefore, the encryption key can be called *public key* while the decryption key is called *private key*. These two keys form an asymmetric *key pair*. Asymmetric encryption is much more computationally intensive than symmetric key encryption scheme. RSA [58] is among the most widely used asymmetric encryption schemes.

2.4.3 Hybrid encryption

While public-key encryption scheme has an advantage that the sender and the receiver do not need to share a common secret, it involves heavy computation in comparison to symmetric key encryption scheme. Hybrid encryption scheme combines the advantage of public-key encryption and the efficiency of symmetric key encryption. In a hybrid encryption scheme, there are two components: *key encapsulation scheme* and *data encapsulation*. A key encapsulation scheme is a public-key encryption scheme which is used to encrypt/decrypt a short symmetric key. Meanwhile, a data encapsulation scheme is a symmetric key encryption scheme which is used to encrypt a long input message. The key encapsulation scheme protects the key used in data encapsulation scheme.

There are many standards that suggest how to use encryption (public and hybrid) schemes correctly in practice. One of the most widely used standards is the *PKCS* standards family. In particular, PKCS#1 [45] provides recommendations for implementing RSA algorithm. PKCS#1 suggests the use of *optimal asymmetric encryption padding* (OAEP) to enhance the security of RSA encryption. PKCS#5 [48] suggests the use of key derivation functions from passwords such as *PBKDF2*. PKCS#7 [47] defines various padding schemes for the data encapsulation scheme.

Plain RSA is a *deterministic* encryption algorithm. That means encrypting two identical plaintexts with the same key yields the same ciphertext. As a result, adversaries can launch *chosen plaintext attacks* by encrypting some plaintexts with the same public key to check if the results are equal to a ciphertext. To avoid this attack, a *padding scheme* that randomizes the message before encrypting is necessary. Standard padding schemes like OAEP

securely pad the plaintext inputs in such a way that the ciphertext outputs are always different. That makes RSA encryptions *semantically secure* [21].

2.5 Key establishment

When two parties communicate with each other over an insecure channel, they need a mechanism to establish trust among themselves. The process of establishing trust between communicating devices which do not have any pre-shared secret is called *secure device association*. The first step is to introduce these two devices to each other. Next, they run a key establishment protocol to agree on a shared secret that can be used to secure the communication channel [60].

As mentioned above, when two parties communicate using symmetric key encryption scheme, the secret key needs to be distributed between them. Key establishment is an effective way to establish a secure communication channel for key distribution.

Suomalainen et. al. [60] mentioned three approaches for key establishment: key transport, key agreement and key extraction. *Key transport* is when one device transfers the key directly to the other device with the help of a secure *out-of-band* (OOB) channel. *Key agreement* is when communicating devices agree on a cryptographic key by running a key agreement protocol. *Key extraction* measures environment specific parameters such as radio signal or noise to extract shared secret. In the scope of this thesis, we only consider key transport and key agreement methods.

2.5.1 Authenticated key agreement

Suomalainen et. al. [60] give a classification of different key agreement protocols. Key agreement can be either *unauthenticated* or *authenticated*. Unauthenticated key agreement is vulnerable to *man-in-the-middle* (MitM) attacks where an adversary intercepts communication messages between two parties and trick them to agree on an adversary-controlled key. An example of an unauthenticated key agreement protocol is the Diffie-Hellman (DH) key exchange protocol [30].

On the other hand, authenticated key agreement provides communication parties with assurance that they know each other's true identities [31]. In particular, authenticated key agreement protocols bind the agreed cryptographic keys to data that can be used to authenticate the client such as pre-shared secret, passwords or public/private keypair. In practice, there are three ways of performing authenticated key agreement [60] as listed below:

1. **Authentication by exchanging key commitments**: In this method, the communicating devices authenticate each others' public keys via an auxiliary channel which is hard to spoof without being detected.

2. **Authentication by short integrity checksum**: Each device computes a short checksum based on knowledge exchanged during the key agreement protocol. The checksum is also known as the Short Authenticated String (SAS). The final check can be performed by asking users to visually compare the checksums. A state-of-the-art protocol in this category is the Manual Authentication IV (MANA IV) protocol proposed by Laur and Nyberg in [50].

3. **Authentication by (short) shared secret**: the pre-shared secret passcode such as user password or an *one-time-password* can be used to authenticate the key. There are different ways of generating the passcode, e.g. requiring users to enter the passcode into one or both devices. Another example is to transfer the passcode via secret auxiliary channels or derive the passcode from the shared environment. State-of-the-art protocols for this method are the Manual Authentication III (MANA III) [37] and password-authenticated key exchange.

2.5.2 Password-authenticated key exchange

Password-authenticated key exchange (PAKE) is a family of authenticated key agreement protocols using shared secrets. The scheme allows two parties to establish a shared key using only the knowledge of a secret password. Especially, this secret password does not need to be a high-entropy password. There are two types of PAKE, *balanced PAKE* and *augmented PAKE*. Balanced PAKE requires both parties to have the same plaintext version of the password, e.g. the Encrypted Key Exchange (EKE) introduced by Bellovin et. al. [22]. On the other hand, augmented PAKE only requires the plaintext password on one party. The other party, instead, only stores the cryptographic hash of the password calculated with a salt, e.g. Augmented Encrypted Key Exchange (A-EKE) [23] and Secure Remote Password protocol [65].

Secure remote password protocol

The Secure Remote Password (SRP) protocol is an augmented PAKE introduced by Thomas Wu in [65]. In comparison with other augmented PAKE protocol (e.g. A-EKE), SRP has better performance and fewer message

rounds. The current version of SRP is *SRP-6a* which is specified in RFC 5054 [61]. The SRP protocol is described as follows.

All computations are performed in a finite field $GF(N)$ where N is a large prime number and g is a generator in $GF(N)$. H is a hash function. k is a multiplier parameter where $k = H(N, g)$. a, b are secret ephemeral values. At first, the client registers with the server with client identity I and password P. The server generates a random salt value s and stores these following information in a tuple.

$$< I, \quad s, \quad x = H(s, P), \quad v = g^x >$$

During authentication, client initiates the SRP protocol with the server as depicted in figure 2.1. We denote A as the client and B as the server, the SRP protocol runs as follows.

Client (A)		Server (B)
$I, \quad \alpha = g^a$	$\xrightarrow{I,\alpha}$	(lookup s,v)
	$\xleftarrow{\beta,s}$	$\beta = kv + g^b$
$\downarrow$		$\downarrow$
$u = H(\alpha, \beta')$		$u' = H(\alpha', \beta)$
$\sigma = (\beta' - kg^x)^{(a+ux)}$		$\sigma' = (\alpha' v^{u'})^b$
$K_{ses} = H(\sigma)$		$K'_{ses} = H(\sigma')$
$M1 = H(\alpha, \beta', K_{ses})$	$\xrightarrow{M1}$	$verify(M1, \alpha', \beta, K'_{ses})$
$verify(M2, \alpha, M1, K_{ses})$	$\xleftarrow{M2}$	$M2 = H(\alpha', M1, K'_{ses})$

Figure 2.1: The secure remote password protocol

1. A generates a secret ephemeral value a and calculates its public value $\alpha = g^a$. A then sends α to B along with its identity I.

2. First, B looks up the tuple with key I to find corresponding values s and v. B then chooses a secret ephemeral b. After that, B generates its own public value $\beta = kv + g^b$. B sends β and s back to A.

3. Both devices compute the common value σ and the hash value K_{ses} of σ which A and B use as the shared session key. Now, they proceed to confirm the session key to complete the protocol. A generates confirmation message $M1 = H(\alpha, \beta, K_{ses})$ and sends to B.

4. Since B has access to α, β and K_{ses}, it verifies the received $M1$. After verification, B sends the final confirmation value $M2 = H(\alpha, M1, K_{ses})$, to A.

5. A verifies $M2$ with α, β and K_{ses}. The successful verification of $M2$ indicates that the session key K_{ses} is authenticated. K_{ses} is used to secure further communication between A and B.

2.5.3 Out-of-band channel

A common approach to associate two devices is to form an out-of-band channel (OOB) between them, which can be physically authenticated by users in close proximity. The aim of OOB is to exchange information that can be used to authenticate the key establishment protocol messages exchanged over an insecure channel. There are various association models which involve the use of OOB channels such as *Compare & Confirm*, *Copy* [62] or *Scan Barcode* [46](Scan). In practice, there is hardly any model that fits in every scenario since they often depend on device hardware capabilities or user's environment. In the scope of this thesis, we only consider two OOB channels that have the least security failures: **Copy** and **Scan**.

2.5.3.1 Quick Response Code

Quick Response Code (QR code) is a method of representing data in a form of a two-dimensional matrix barcode. Unlike a standard barcode, a QR code can store more bits of information in a single compact image. Furthermore, users can easily scan a QR code with their smartphone camera to read the embedded information. It is also readable from any direction. In key establishment context, scanning a QR code is an effective OOB channel for key transport due to its increasing popularity nowadays. Figure 2.2 illustrates an example of QR code.

2.5.3.2 Copy

In this method, a device displays a short, memorable passcode which users type into the other device. The passcode can be treated as an ephemeral shared-secret between two devices. Thus, it allows us to run various key agreement protocols which are based on *short shared-secret* (SSS) as described in section 2.5.1. Uzun et. al. [62] mentioned that *Copy* channel is inherently resistant to fatal errors and is the most preferred personal choice as the OOB channel that users want to use on their devices.

Figure 2.2: An example QR code

2.6 Analysing security protocols

After designing and implementing security protocols, it is mandatory to analyse them in a systematic way to ensure it's security. First, an adversary model is specified. Then protocol designers state various security properties that the protocols must satisfy and finally, the protocols are analysed to check if their security properties hold.

2.6.1 Dolev-Yao adversary model

Improperly designed security protocols are vulnerable not only to "passive" eavesdroppers, but also to "active" adversaries who can impersonate legitimate users, modify or replay messages. It is important to explicitly state the capabilities of adversaries so that we can analyse the protocols correctly. A formal specification of the capabilities of the adversary is introduced by Danny Dolev and Andrew C. Yao in their work on the security of public key protocols [32]. The Dolev-Yao adversaries are capable of the following:

1. The attacker can read, modify and send any message in the network.
2. The attacker is also a legitimate network user. He can initiate a conversation with any other user in the network.
3. The attacker can act as a receiver or sender of any conversation with any user in the network.

2.6.2 Security property

To design and analyse key establishment protocols, Menezes et. al. [55] suggest the use of *security properties*. For each protocol, we need to state which

of these properties it satisfies. We can consider various security properties for a key establishment protocol between two entities A and B as defined in [27].

- **Implicit key authentication from A to B** — Assurance for B that no parties other than A may gain access to the established secret key. This property does not require A to actually possess the key. For this reason, it is specified as "implicit".
- **Key confirmation from A to B** — Assurance for B that A is in possession of the correct key.
- **Explicit key authentication from A to B** — is security property that implies both implicit key authentication and key confirmation from A to B hold. In this case, it assures B that A is the only other entity that is in possession of the correct secret established key.
- **Entity authentication of A to B** — Assurance of the identity of A to B that A has actually participated in the key establishment protocol with B.
- **Perfect forward secrecy** — compromising a long-term secret key of an entity does not compromise the keys established in previous sessions.

2.6.3 Scyther tool

Menezes et. al. [55] suggest different approaches for analysing security protocols such as *complexity-theoretic analysis* or *formal method*. Formal method analysis uses logics of authentication and various other methods which combines algebraic and state-transition techniques to prove the correctness and security of the protocols. In the scope of this thesis, we look at the tool-supported formal method which is a formal method analysis using software tools. In particular, we use Scyther [26], an automatic tool for verifying, falsifying and analyzing security protocols. The tool uses protocol description as input then outputs a summary report along with graphs for each attack on the protocol. The protocol description is written in *spdl* language [20]. Scyther then allows user to perform different functionalities as described below.

Verification of claims: While describing the security protocol using *spdl*, users can specify different security properties in the protocol using claim events. For example, given an agent *Alice* in role X, *Alice* can claim that a nonce α it generated is confidential (secrecy). The correctness of this claim will be assessed by the tool.

Automatic claims: If there is no specific claim in the protocol description, Scyther can automatically generate claims and evaluate them. As described above, these claims are security properties of the protocol. This functionality allows Scyther to quickly generate an overview of the whole protocol.

Characterization: Scyther generates and models all traceable paths of protocol execution based on the protocol description. Users can analyse those paths manually to gain insight of the protocol.

2.6.3.1 Role script

Listing 2.1: Scyther minimal protocol description

```
1  function f;
2  protocol ExampleProtocol(I,R) {
3          role I {
4                  fresh N: Nonce;
5                  send_1(I, R, f(N));
6          };
7          role R {
8                  var N;
9                  recv_1(I, R, f(N));
10         };
11 };
```

Role scripts are specification of the protocol written in *spdl* by users. Listing 2.1 depicts a sample role script. A protocol description contains a finite number of roles e.g. role I and R are protocol participants. Scyther then executes the roles multiple times separately in a finite number of runs. Roles consist of sequences of send, receive and claim events. Events have term parameters which are constructed using role names, function names (e.g. f), variables (e.g. N) and constants. Generally, it can be understood as two roles are sending messages to each other via pairs of send/receive events.

2.6.3.2 Claim

Claim events are used to model the intended security properties of the protocol. There are several pre-defined type of claim that we will use in our modeling; they are all defined in Cas Cremers's PhD thesis about the Scyther tool [25].

- *Secret:* A role can claim that a parameter is secret. Secrecy implies that the information in the parameter is not revealed to an adversary, even though the underlying protocol communicates over an untrusted network.

- *Alive:* Aliveness means that whenever I completes a run with R, then R has been running the protocol previously. In fact, R can run the protocol with other malicious party instead of I. R can also run the protocol even before I runs. As a consequence, aliveness can only guarantee the entity authentication property of the protocol.

- *Nisynch:* Non-injective synchronisation is a property for a protocol and a claim event. Synchronisation means that in every successful execution of a protocol, all role players exactly follow their roles defined in the description, i.e. exchange the expected messages, variables in intended sequences.

- *Niagree:* Agreement is a weaker form of synchronization. Originally from [53], given I is in agreement with R implies that I and R are alive and they agree on all data variables. Nisynch is stronger than Niagree since it ensures the sequence of operations inside the protocol. Agreement is also equivalent to key confirmation security property.

- *Reachable:* Scyther will check whether this claim can be reached at all. The claim is true if there exists a trace of the protocol where the claim can be reached. It is used to check if there is any obvious error in the protocol specification.

2.6.3.3 Approximating equality

In Scyther, "two terms are equal if and only if they are syntactically equivalent"[3]. However, there are common cryptographic constructions that are equal but require to be modeled differently. An example is the following exponential equation:

$$g^{ab}(mod\ N) = g^{ba}(mod\ N) \tag{2.1}$$

Such equation can be modeled in Scyther by introducing a function g with input parameters a and b. However, $g(a, b)$ is obviously not syntactically equivalent to $g(b, a)$, thus the model cannot be prove correctly.

[3] `https://github.com/cascremers/scyther/raw/master/gui/scyther-manual.pdf`

Scyther introduces the approximating equational theory to work around this problem. The idea is that we can provide the adversary an ability to learn both the equivalent classes if he knows one of them. If we model the adversary that knows $g(a, b)$, we can allow him to learn $g(b, a)$ and vice versa. Basically, we can model this behavior in Scyther role script using a helper protocol denoted by prefix @ as follows:

Listing 2.2: Diffie-Hellman helper protocol

```
1 hashfunction g;
2 protocol @DH(X) {
3         role X {
4                 var a,b: Ticket;
5                 recv_!1(X,X,g(a,b));
6                 send_!2(X,X,g(b,a));
7         };
8 };
```

In Listing 2.2, the helper protocol $@DH$ allows agent X to receive $g(a, b)$ from anywhere. X then has the ability to produce $g(b, a)$ and send to itself. As a result, X learns both values in the Diffie-Hellman equivalent term.

2.7 Deduplication

Deduplication is a method of identifying multiple copies of data and storing only a single copy of identical data. During the deduplication processes, unique chunks of data such as files or block of bytes are analyzed to identify duplication. In case of data duplication, only one instance of the data is stored/transferred hence it greatly improves storage utilization and transfer time/bandwidth. Initially, deduplication was introduced to reduce multiple copies of files during backups [56]. Nowadays, data deduplication is a critical part of any storage services since it reduces the total cost of ownership [43] and user experience.

As mentioned above, encryption is a secure method to protect data in cloud storage. However, a semantically secure encryption scheme encrypts plaintext messages to generate different ciphertext even when the encryption key and plaintext are same. Thus, the cloud storage cannot benefit from deduplication scheme. In 2012, Douceur [33] introduced the concept of *Convergent Encryption* where the encryption key is derived from a file itself. Convergent encryption is simply encrypting a file using symmetric encryption scheme with a key which is the cryptographic hash of the plaintext file. Using convergent encryption, encrypting identical files produce the same ciphertext

thus enabling deduplication. However, convergent encryption is not semantically secure as it suffers from chosen-plaintext attacks. There have been efforts to improve the security of convergent encryption [63]. Nevertheless, they stop at limiting the scope of the attack, not preventing it.

2.8 Data differencing

Data differencing is the process of computing the invertible differences between two sources of data namely "source" and "target". The computed value is called the "delta" [49]. In a situation where differences are small, such as modifying a few bytes on a large file, the delta is also very small in comparison with the original source or target. Thus, it is more efficient to transmit or store the delta rather than the whole big file. A standard format for storing a delta is **VCDIFF** [49]. The format is compact, portable, generic and efficient.

Delta encoding is a method to generate delta. It involves two algorithms, encoding and decoding. The encoding algorithm takes the source as an input and the target to generate delta as an output. On the other hand, the decoding algorithm uses the source and the delta to reproduce the target. **open-vcdiff** [10] is a tool developed by Google, which follows VCDIFF standard to provide delta encoding.

Chapter 3

Problem Statement

3.1 Problem description

Client-side encryption is an effective way of preserving security and privacy for users' data in cloud storage. However, enabling access to users' encrypted storage from multiple client devices requires an efficient key distribution mechanism. State-of-the art solutions rely on users' passwords to derive and distribute the keys across client devices. However, this is not a competent solution since it is hard for users to generate and remember good passwords.

An effective client-side encryption solution designed for the cloud storage should exhibit the following features.

- Automatically encrypt files before uploading to cloud storage.
- Securely distribute the keys across users' domain of devices.
- Work seamlessly with existing cloud storage services.
- Offer well balanced security and usability by providing **consistent** and **intuitive** user interactions without sacrificing security.
- Reduce the cost of network bandwidth and storage capacity for encrypted storage.

3.2 Requirements

In order to design such a solution, we define the adversary model and identify different system requirements as explained in the following subsections.

3.2.1 Adversary Model

The adversary model is as follows. Authorized devices in users' domain of devices access plaintext files on encrypted cloud storage. We assume the devices themselves are trustworthy. The adversary has Dolev-Yao capabilities over the cloud storage. That means the adversary has full access to users' cloud storages. They can read, create and modify files on users' storage without being noticed. We do not attempt to protect against denial-of-service attacks, e.g. by deleting files in the cloud storage. We also assume that the adversary cannot listen/observer or tamper with the local communication among legitimate devices and users.

3.2.2 Security requirements

S1. **Strong client generated keys:** The system should use strong keys generated locally on the client device.

S2. **Authenticated file encryption:** The system should encrypt files at client side before uploading to cloud storage using authenticated encryption.

S3. **Plaintext access from multiple devices:** All authorized devices should be able to download files from the cloud storage and decrypt them to access plaintext files.

- S3.1 **Secure key distribution:** The system should offer a key distribution mechanism for transferring the keys across authorized devices. Key distribution protocols must be operational and secure. In particular, the protocols should satisfy security properties defined in 2.6.2.
- S3.2 **Avoiding third-party servers:** The system should not depend on any third-party server except the cloud storage provider to store and distribute the keys.

3.2.3 Usability requirements

U1. **Consistent user experience:** The system should exhibit consistent user experience (instructions, workflow) across different platforms.

U2. **Intuitive user interface:** The system should have the following characteristics.

U2.1 **Familiar user experience:** The system should provide a familiar user experience with other related products so that we can avoid the problem of the user not knowing what to do and the overhead of explanation.

U2.2 **Clear instruction:** Instructions given by the system should be clear and simple. We want to avoid technical terms that are unfamiliar to normal users.

U3. **Minimum interaction:** The system should automatically handle most of the processes and decisions. We want to minimize user involvement in configuring the system. As a result, the system is more convenient to use and resilient to users' mistakes.

3.2.4 Additional system requirements

A1. **Multiple platforms**

A1.1. **Multiple device types:** The system should work on multiple device types. In particular, it should work on both desktop and mobile devices. It should also works with devices which have different hardware features such as camera or NFC.

A1.2. **Multiple cloud storage:** The system should work with multiple popular cloud storage providers such as Dropbox, Google Drive or OneDrive.

A3. **Extensibility and maintainability:** The system should be modularized into loosely-coupled components and well-documented. This way, it is easy to re-use, maintain and add new features to the system.

A4. **Reducing network overhead:** File updates should minimize network communication overhead of the cloud storage service.

Chapter 4

Approach

This chapter introduces and explains the design decision of our client-side encryption solution for cloud storage.

4.1 High level architecture

We present *OmniShare* as our solution that is designed based on requirements defined in section 3.2. OmniShare is the first scheme to allow client-side encryption with high entropy keys for cloud storage combined with an intuitive key distribution mechanism enabling data access from multiple client devices.

Dropbox, OneDrive, etc, provide client-side applications that run on client devices and allow users to operate on their storage. Since we want OmniShare to work with multiple cloud storage (requirement **A1.2**), we design OmniShare as a software client working on top of existing cloud storage. Basically, OmniShare introduces an intermediate layer providing data encryption/decryption between user data and cloud storage client. OmniShare takes users' data, encrypts them and feeds them to the cloud storage client application as input. Figure 4.1 describes a high level architect of OmniShare which consists of four components: *OmniShare domain*, *key hierarchy*, *key distribution*, and *synchronization*.

4.2 OmniShare domain

We present the concept of *OmniShare domain* which creates a *container* in a user's cloud storage that allows user to store his encrypted data and a domain of devices that he authorized. OmniShare protects users' data by encrypting them with standard encryption algorithm and storing ciphertext files inside the domain container. Since the container is entirely on the users' cloud

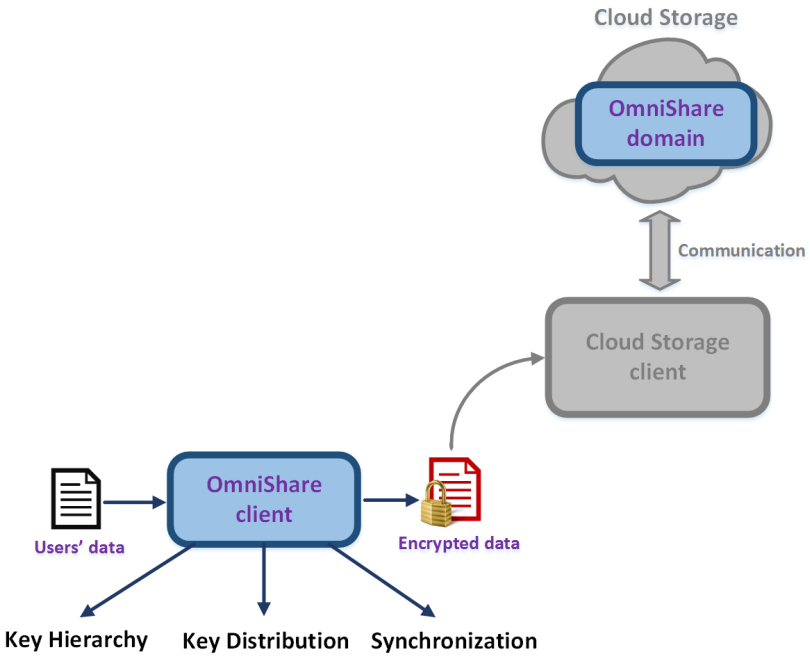

Figure 4.1: OmniShare components

storage, it is accessible from all devices of the domain owners as long as they are authenticated to the cloud service provider. However, only authorized devices that belong to the user's domain of devices can decrypt ciphertext files inside the container to gain access to the plaintext files. We also call this container an *encrypted storage*.

Users provide access permission to their plaintext files by authorizing devices that they wish to access the encrypted storage from. To do this, we introduce the *authorization* process where users authorize their personal devices. Figure 4.2 depicts an OmniShare domain and its components. We call a user device that is authorized into the domain as an *authorized device*. Whereas the *new device* is the one that the user wants to introduce to the domain and allow access to plaintext files. As in the picture, the new device is outside of the users' domain of devices, thus cannot access the plaintext files.

When a user initiates OmniShare from a device, it creates a new OmniShare domain. The domain creation process involves (i) creating the container which is a directory called *OmniShare directory* in users' cloud storage, (ii) creating a domain specific symmetric *Root key*, (iii) creating a *domain descriptor* file called *.OmniShare* inside OmniShare directory to maintain a list

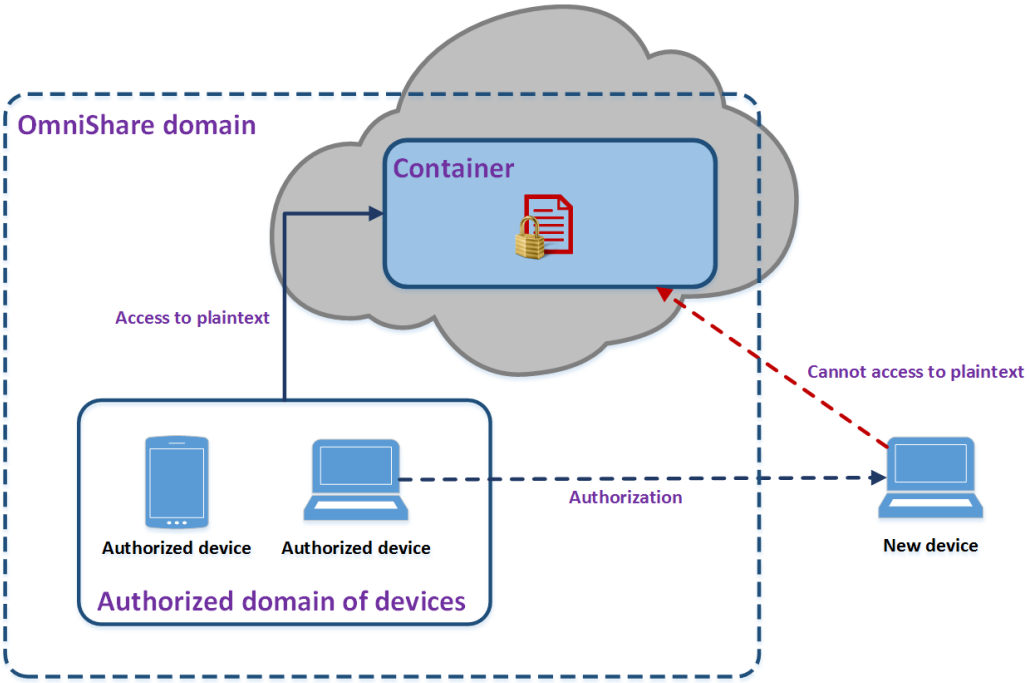

Figure 4.2: OmniShare domain

of authorized devices for that domain. After that, OmniShare automatically adds the device as the authorized device in the domain descriptor file.

4.3 Key hierarchy

For each user's authorized device, OmniShare generates an asymmetric key pair (*Device Keys*) which consists *Device Private Key* and *Device Public Key*. Device Private Key is stored and protected using platform specific mechanism.

When a file is first created, OmniShare generates a new *File Key* to encrypt the file. OmniShare also generates a new *Directory Key* for each directory inside the OmniShare container. OmniShare maintains a key hierarchy where key at each level encrypts the key at the level below starting from RK. It uses a *lock-box* data structure to limit Root Key access only to users' authorized devices. Root Key is encrypted separately with the Device Public Key of each authorized device.

Figure 4.3 depicts an example of a key hierarchy that belongs to an OmniShare domain including the lock-box data structure that limits Root Key

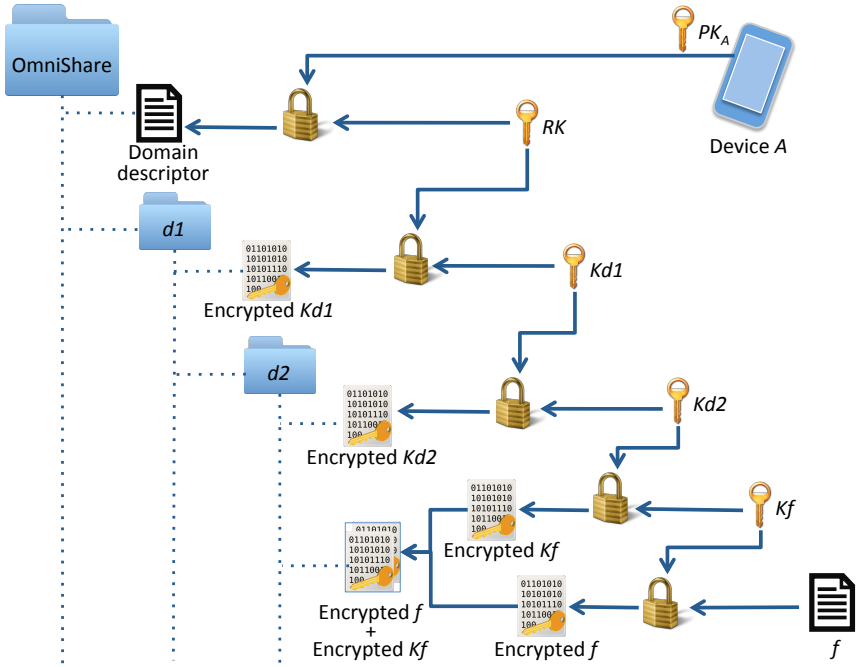

Figure 4.3: OmniShare key hierarchy example.

(RK) access to only device A. RK is encrypted using PK_A and stored in the domain descriptor file. The key hierarchy corresponds to the directory structure inside OmniShare directory. In particular, we use RK to encrypt Directory Key $Kd1$ of directory $d1$. $Kd1$ is used to encrypt Directory Key $Kd2$ of the next level directory $d2$. Then, $Kd2$ is used to encrypt File Key Kf which, in turn, is used to encrypt plaintext file f. The encryption of Kf is stored along with the encrypted file f while encrypted $Kd1$ and $Kd2$ are stored in their corresponding directories.

OmniShare key hierarchy offers several benefits. First, it minimizes the number of keys that need to be distributed. In particular, we only need to distribute Root Key securely among authorized devices so that these devices can decrypt other encrypted keys and ciphertext files. Secondly, since we use different File Keys and Directory Keys, it is easy to implement *share* features. For example, to share files or directories with other users' devices, we can simply give these devices the corresponding File Keys or Directory Keys. Finally, the key hierarchy minimizes the impact when a specific File Key is compromised, in this case, other keys remain secure.

4.4 Key distribution

As we introduced before, device authorization is when users authorize their personal devices for accessing plaintext files inside OmniShare domain. To do this, on a new device, OmniShare presents users with an interface to select one of their previously authorized devices from their domains. After that, OmniShare runs an *automatic capability discovery* process to establish an OOB channel between the new device and the selected authorized device. Then, it runs a key distribution protocol depending on the OOB channel.

4.4.1 Cloud storage communication channel

Running a key distribution protocol between two devices which do not have pre-shared secret requires a communication channel. An approach to establish such channel is to use auxiliary bi-directional channels such as *NFC* or *bluetooth*. However, these auxiliary channels are not available on all devices. For this reason, we decided to use the cloud storage itself as a communication channel for our key distribution protocol. Basically, two devices exchange messages by uploading/downloading files containing the messages to/from the cloud storage.

The advantage of cloud storage communication channel is that it is available for all devices (requirement **A1.1**) since it only requires internet connection. It is also can be fully-automated since devices can upload and download files automatically without requiring users' involvement. Thus, it ensures the minimum interaction requirement **U3**. However, since adversaries can access the cloud storage, this communication channel is insecure. Therefore, we still need an OOB channel to authenticate information exchanged via the cloud storage communication channel.

4.4.2 Automatic capability discovery

OmniShare supports two types of OOB channels, i.e. *Copy* and *Scan* channels. These channels rely on device auxiliary hardware features that can be used as input/output interfaces called device *capabilities*. In particular, for Copy channel, one device needs a monitor to display the passcode while the other device needs a keyboard for users to type the passcode in. On the other hand, for Scan channel, a QR code is displayed on one device's monitor whereas the other device scans the QR code using its camera.

Table 4.1 lists capabilities that OmniShare uses, their usages and priorities. *Input* means that the feature can be used for receiving information. *Output* indicates that the feature is used for sending information. Priority

is a value starting with 1 as the highest priority that we hard-coded so that the application can sort and choose the highest priority capability. However, these values can be re-defined at will.

Feature	Usage	Default priority
Camera	Input	1
Monitor	Output	1
Keyboard	Input	2

Table 4.1: Device capabilities used in OmniShare

As shown in table 4.1, *Monitor* is the only output capability since we assume that all devices have monitors. On the other hand, we choose *Camera* as the first priority input capability and *Keyboard* is the secondary one. The first reason is that we want to experiment with different OOB channels. If keyboard has higher priority than camera, the Copy channel will always be prioritized over the Scan channel since we assume that all devices have keyboards. The second reason is that our key distribution protocol using Scan channel is much faster than the one using Copy channel. Details about the measurement are presented in chapter 6. If a device does not have any camera, it's priority for keyboard is set to 1.

OmniShare stores a list of capabilities for each authorized device in OmniShare domain in the domain descriptor file. For a new device, at first, it downloads the list of capabilities of the authorized device that users select. The new device then chooses an OOB channel by running a matching algorithm with inputs are capabilities list of both devices. After that, the new device sends it's capability list to the selected authorized device by uploading the list to the cloud storage. The selected device runs the same matching algorithm with the same inputs as the new device. Thus, they select the same OOB channel. This process is called the *automatic capability discovery* process.

Currently, we implement a simple matching algorithm which only depends on the input capability of the authorized device. In particular, if the highest priority input capability of the authorized device is camera, the algorithm returns Scan channel. Otherwise, the algorithm returns Copy channel. However, this algorithm can be generalized in the future to adapt with different capabilities and OOB channels.

We then developed two key distribution methods — *key distribution via QR code* for Scan OOB channel and *key distribution using SRP* for Copy OOB channel. The two protocols communicate using *cloud storage commu-*

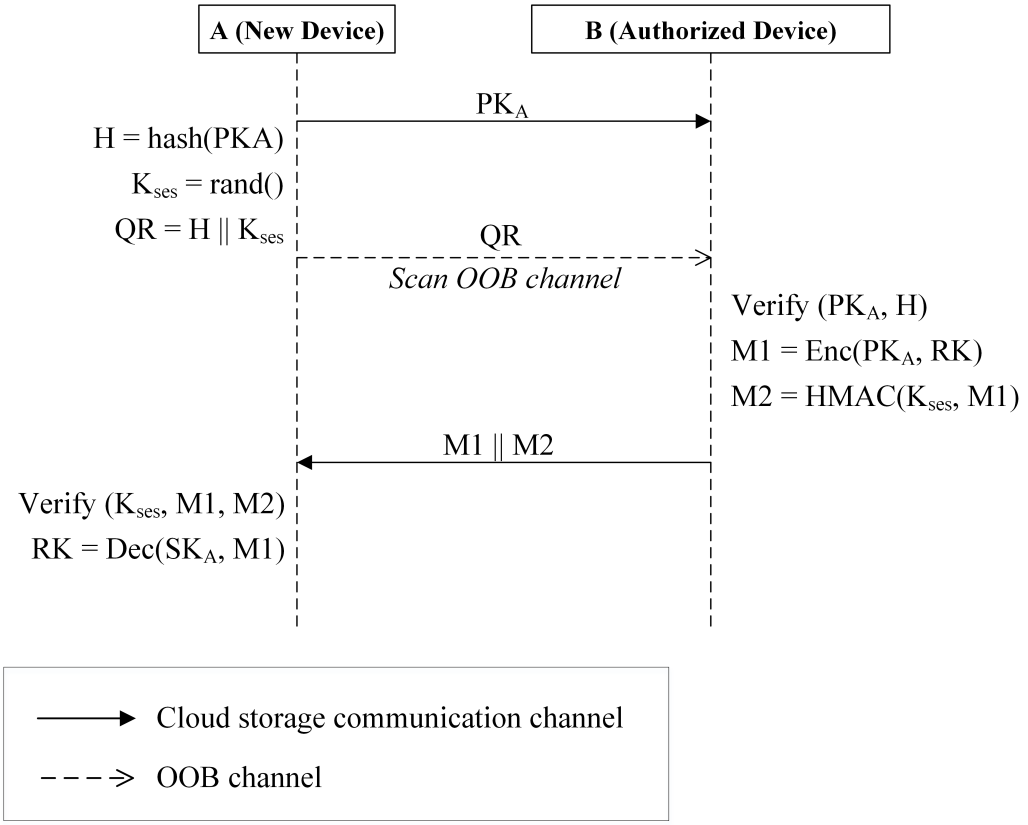

Figure 4.4: Key distribution using QR code

nication channel.

4.4.3 Key distribution via QR code

Our key distribution via QR code protocol is depicted in Figure 4.4. We denote the new device as A and the selected authorized device as B. The protocol runs as follows.

1. First, A sends its public key PK_A to B.
2. Since the cloud communication channel is insecure, B needs to verify PK_A. To do this, A displays a QR code consisting of the cryptographic hash value H of PK_A and a random session key K_{ses}. B requires the user to scan the QR code. B can then verify PK_A using H. B must abort the process if the verification fails.
3. After verifying PK_A, B encrypts RK with PK_A to get $M1$ and sends back to A via the cloud storage communication channel. Here, A also

need to verify the authenticity of $M1$. Therefore, B also appends $M1$ with an HMAC value $M2$ of message $M1$ using K_{ses} as the key. B informs the user that the new device (A) has been authorized.

4. In the final step, A verifies the $M2$ with K_{ses} to ensure the integrity and authenticity of $M1$. After that, A can decrypt $M1$ with its Device Private Key (SK_A) to obtain RK. Finally, A informs users that the key exchange process is completed. A can start using RK to access encrypted data on the storage.

In this protocol, we use H and $M2$ to authenticate PK_A and $M1$ respectively. Without H, the protocol is vulnerable to MitM attack where adversary C can replace PK_A with its public key PK_C so that B is tricked to send RK to C. Similarly, C can send $M1' = Enc(PK_A, K_{fake})$ to A. In the later case, device A is tricked to use K_{fake} as its Root Key. Thus, it allows users to encrypt and upload files using the key which is available to the adversaries.

4.4.4 Key distribution using SRP

In this key distribution protocol, we use the Copy OOB channel where users enter a short share secret, which is displayed on one device's monitor, to another device using a keyboard. Popular authenticated key establishment by short shared secret methods are MANA III and PAKE protocol family.

[60] describes a variant of the MANA III protocol which is used in *Wi-Fi Protected Setup* (WPS). The protocol splits the short shared secret into k pieces. Both devices repeatedly prove their knowledge of each piece by exchanging *commitment* values. The probability of a successful attack from a adversary is inversely proportional to the length of the shared secret and k. However, the protocol uses too many rounds of message exchange, i.e. $4 * k$. Thus, it is not suitable for our cloud storage communication channel.

On the other hand, PAKE protocols are better in terms of minimizing message exchange rounds. Figure 4.5 depicts key distribution between two devices using PAKE on an abstract level. The protocol runs as follows:

1. A generates a short passcode P and displays to the user. The user also is required to enter passcode P from A.

2. A and B run PAKE protocol to derive session key K_{ses}.

3. B uses authenticated encryption to encrypt RK with key K_{ses} to get M and sends to A. A then decrypt M with K_{ses} to get RK. We use

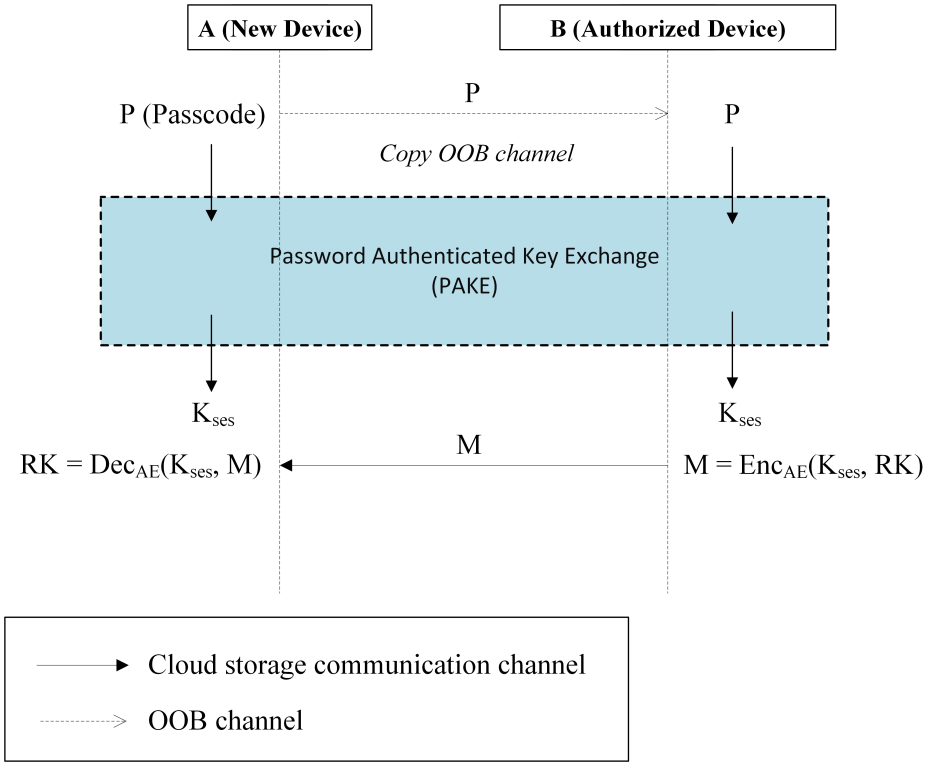

Figure 4.5: Key distribution using PAKE

authenticated encryption in this case to ensure both confidentiality and integrity of RK.

Among PAKE protocols, we choose SRP protocol which not only has best performance in comparison to other augmented PAKE protocols but has been widely deployed in mainstream systems such as OpenSSL [1] or GnuTLS [2]. Specifically, we use SRP version 6a [64]. Figure 4.6 depicts key distribution between two devices using SRP. In our SRP set-up, the new device (A) plays the role of the client while the selected authorized device (B) is the server.

In section 2.5.2, B generates and stores the salt value s after A's registration. In our SRP set-up, the password is generated and entered manually by users on both devices hence there is no need for the registration step. Thus, B generates s on-the-fly and sends to A. Additionally, we also concatenate the final message M with the final message $M2$ of the original SRP and send them together to A. As a result, we save one round of communication.

[1] `https://www.openssl.org/`

[2] `http://www.gnutls.org/manual/html_node/Authentication-using-SRP.html`

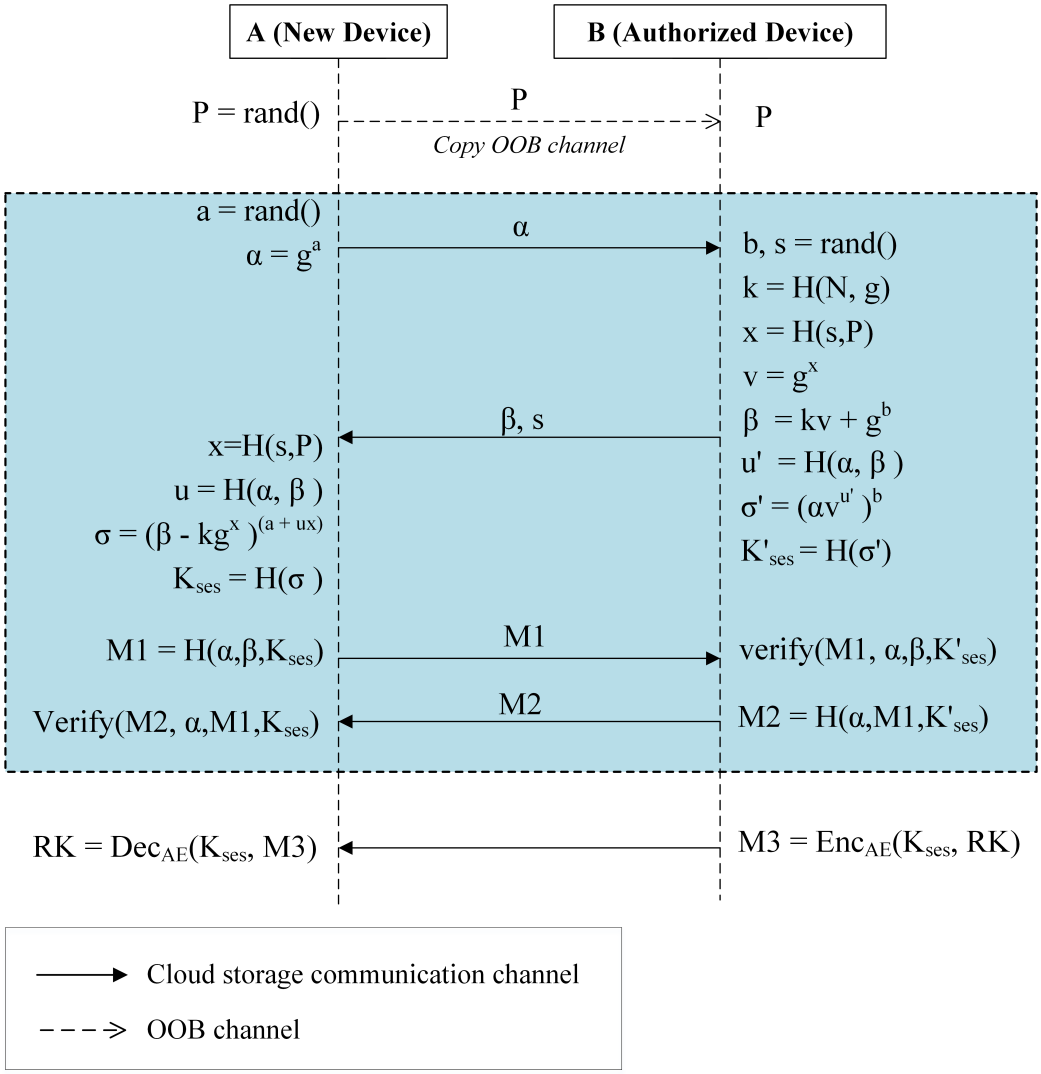

Figure 4.6: Key distribution using SRP

4.5 Synchronization

Existing cloud storage solutions automatically take care of data synchronization among their client devices. OmniShare introduces an intermediate layer providing data encryption/decryption between user data and cloud storage client. It is important to ensure a reliable synchronization between OmniShare and the cloud storage clients without requiring any change from the cloud providers. Here, we designed two approaches to synchronization between unencrypted data and encrypted data.

4.5.1 The interceptor pattern

A straightforward solution for synchronization between user data and cloud storage clients is to intercept and modify interactions (e.g. creation, modification, deletion of files and directories) between users and the cloud storage clients. Some cloud providers provide APIs to interact with their clients or servers [3]. Using these APIs, we can intercept and encrypt files before uploading to the cloud storage. Similarly, we can decrypt files after downloading ciphertext from cloud storage servers. This solution is easy to implement since most of the operations are provided by the storage APIs. It makes the encrypt/decrypt process become transparent to the rest of the system; thus, other features such as versioning, conflict handling can be left to the storage client to handle.

However, it is not practical for situations where there are many interactions between users and their data in a short period of time. In these situations, data need to be re-encrypted/decrypted frequently. That increases network traffic and resource consumption, hence user experience may reduce. It also increases dependency of the system to a specific cloud storage's APIs.

4.5.2 The periodic synchronization pattern

Another approach to synchronization between users' data and cloud storage clients is to provide a local *plaintext directory* on a user's device. The idea is to periodically encrypt user data in this plaintext directory and put encrypted data into a *ciphertext directory*. The ciphertext directory will be used as inputs for the cloud storage clients to synchronize with the OmniShare container. This approach eliminates the scalability issue with frequent updates on files in comparison with the previously described approach. However, it introduces more complexity. As a result, it is error-prone and much more complicated to implement. We call this approach as *periodic synchronization pattern*.

Assume that each user device has access to a consistent clock. In the periodic synchronization pattern, a simple version control system records changes in both plaintext and ciphertext directory with regards to the time that those changes happen. We use a meta-data database for this purpose. The database has maps of simple (*key*, *value*) tuples where *key* is the relative path to the file/directory inside a directory (plaintext or ciphertext) and *value* which is the last modification time (LMT) of that file. LMT is the last time that file is written to. For example, a plaintext file f inside a sub-directory d of the plaintext directory created at time x is stored in *PlaintextMap* as

[3] `https://www.dropbox.com/developers/core`

$< /d/f, \quad x >$. Meanwhile, its encrypted counterpart $f.os$ in the ciphertext directory created at time y is stored in *CiphertextMap* as $< /d/f.os, \quad y >$.

There are two phases in periodic synchronization pattern. The first phase is *upload* where changes from *plaintext directory* are applied to *ciphertext directory*. The second phase is *download* phase where changes are applied in the reverse order.

In *upload* phase, first we run *DetectPlaintextChanges* algorithm to detect all changes in plaintext directory. Algorithm 1 describes *DetectPlaintextChanges* algorithm. Basically, we loop through all files in the plaintext directory. For each file, we get its LMT $C1$. If there is no equivalent *CiphertextFile* (same relative path) in the ciphertext directory, we can conclude the *PlaintextFile* is a new file. Therefore, we put it into *NewFileList*. If there is a *CiphertextFile* in the ciphertext directory, we get LMT value C from the *PlaintextMap* in database using the path of *PlaintextFile* as the key. Now, there are three options. If C is NULL or C is bigger than $C1$ (i.e. the current file is older than the file in database), there must be problem with the database. Therefore, we return an error as *corrupted database*. $C < C1$ indicates that the plaintext file has been modified. Thus, it is added to the *ModifiedFileList*. Otherwise, if C and $C1$ are equal, the plaintext file does not change. We add unchanged files to a *SynchronizedFileList*. After the loop, we check all files in the *PlaintextMap*. If there is a file that is not in any list that we just constructed, it indicates the file is *deleted* so we can add the file to the *DeletedFileList*. The return value of the algorithm is a list combined by *NewFileList*, *ModifiedFileList*, *DeletedFileList* and *SynchronizedFileList*.

After running *DetectPlaintextChanges*, we also need to run *DetectCipherChanges* which is basically the same algorithm but for *ciphertext directory*. The result is a list consists of *new*, *modified*, *deleted* and *synchronized* ciphertext files.

After detecting changes, we run the *UploadSync* algorithm to apply the changes from *plaintext directory* to *ciphertext directory*. Algorithm 2 describes the synchronization process for *new* files and *deleted* files. We loop through the list of *new* file from *DetectPlaintextChanges* result, encrypt and store them in the ciphertext directory. For *deleted* plaintext files, we can simply remove the corresponding *CiphertextFile*.

Algorithm 3 describes how we synchronize *modified* files from *plaintext directory* to *ciphertext directory*. Basically, for each *PlaintextFile* in $L1$, we find the corresponding *CiphertextFile* in *ciphertext directory*. $L2$ is the result of *DetectCipherChanges*. If $L2$ does not contain *CiphertextFile*, there is problem with the meta-data database since in *DetectPlaintextChanges*, we make sure that *CiphertextFile* exist. Otherwise, we check the *status* of *CiphertextFile*. If the status is *synchronized* (i.e. CiphertextFile belongs to

Algorithm 1 Detect changes in plaintext directory

```
1: procedure DetectPlaintextChanges
2:     for each File PlaintextFile in PlaintextDirectory do
3:         Time C1 = GetLMT(PlaintextFile);
4:         File CiphertextFile = GetCiphertextFile(PlaintextFile);
5:         if CiphertextFile NOT exist then
6:             NewFileList.Add(PlaintextFile);
7:         else
8:             Time C = PlaintextMap.GetValue(PlaintextFile.Path);
9:             if C == NULL then
10:                Error(corrupted DB);
11:            end if
12:            if C < C1 then
13:                ModifiedFileList.Add(PlaintextFile);
14:            else
15:                if C > C1 then
16:                    Error(corrupted DB);
17:                else
18:                    SynchronizedFileList.Add(PlaintextFile);
19:                end if
20:            end if
21:        end if
22:    end for
23:    File[] DBList = PlaintextMap.GetAllKeys();
24:    for each File f in DBList do
25:        if f NOT in NewFileList
26:            && f NOT in ModifiedFileList
27:            && f NOT in SynchronizedFileList  then
28:            DeletedFileList.Add(f);
29:        end if
30:    end for
31:    Result = Join(SynchronizedFileList, NewFileList,
32:    DeletedFileList, ModifiedFileList);
33:    Return Result
34: end procedure
```

SynchronizedFileList), we can safely encrypt *PlaintextFile* and store in the *ciphertext directory*. On the other hand, if the status is *new* or *modified*, there is a conflict between the change in *PlaintextFile* and in *CiphertextFile*. In the scope of this thesis, we do not address problems such as *corrupted DB*

Algorithm 2 Synchronize new and deleted plaintext files

```
1: procedure UploadSync
2:     L1 = DetectPlaintextChanges();
3:     L2 = DetectCipherChanges();
4:
5:     for each PlaintextFile in L1.NewFileList do
6:         Encrypt(PlaintextFile);
7:     end for
8:
9:     for each PlaintextFile in L1.DeletedFileList do
10:        CiphertextFile = GetCiphertextFile(PlaintextFile);
11:        if CiphertextFile exist then
12:            Delete(CiphertextFile);
13:        end if
14:    end for
15:    ...
16: end procedure
```

or *conflict*. However, our algorithms detect those problems and report to the main system runtime. As a result, we can tackle those problems in our future work.

After running *upload* phase OmniShare executes the *download* phase. Basically, we run *DetectCipherChanges* and *DetectPlaintextChanges* first. Then, we run *DownloadSync* algorithm which is similar to *UploadSync* but for ciphertext files.

In summary, cloud storage client application synchronizes encrypted files in the cloud while OmniShare with meta-data database ensures the consistency between plaintext files and ciphertext files using the periodic synchronization pattern.

4.5.3 Incremental synchronization

As pointed out before, deduplication is an important feature of cloud storage service. However, conventional encryption defeats the whole purpose of deduplication. Even if two data units are slightly different, encryption makes the outputs completely different. Figure 4.7 describes this situation. Let δ_P is the difference between two version of a file. If there is just a small change to the original file, δ_P is equal to the size of the change, which is considerably smaller than the size of the original file size. Nonetheless, after encryption, the two outputs are totally different, the difference between those two files

Algorithm 3 Synchronize modified files

```
1:  procedure UploadSync
2:      L1 = DetectPlaintextChanges();
3:      L2 = DetectCipherChanges();
4:
5:      ...                                          ▷ synchronize new file
6:      ...                                          ▷ synchronize deleted file
7:      for each PlaintextFile in L1.ModifiedFileList do
8:          CiphertextFile = GetCiphertextFile(PlaintextFile);
9:          if L2.exist(CiphertextFile) then
10:             if CiphertextFile.status == synchronized then
11:                 Encrypt(PlaintextFile);
12:             end if
13:             if CiphertextFile.status == new then
14:                 Report(conflict);
15:             end if
16:             if CiphertextFile.status == modified then
17:                 Report(conflict);
18:             end if
19:         else
20:             Report(corrupted DB)
21:         end if
22:     end for
23: end procedure
```

are approximately the same as the file size. As a result, even thought cloud storage services might implement deduplication, they still need to upload the whole file again every time there is a change to the original file.

To tackle this problem, we propose a simple solution, called incremental synchronization. In short, we extract the binary difference between two versions of a file to construct *Diff file*. Diff file is then encrypted and stored separately from the original file. With this approach we can save considerable amount of network bandwidth used for uploading/downloading files considering the size of an encrypted diff file are much less than that of an original file (Requirement **A3**).

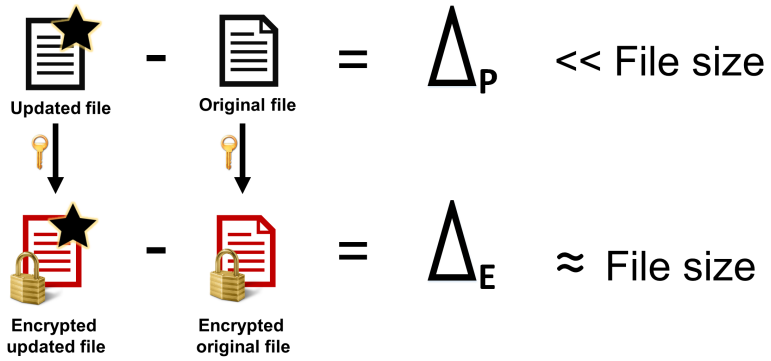

Figure 4.7: Encryption does not keep the update difference.

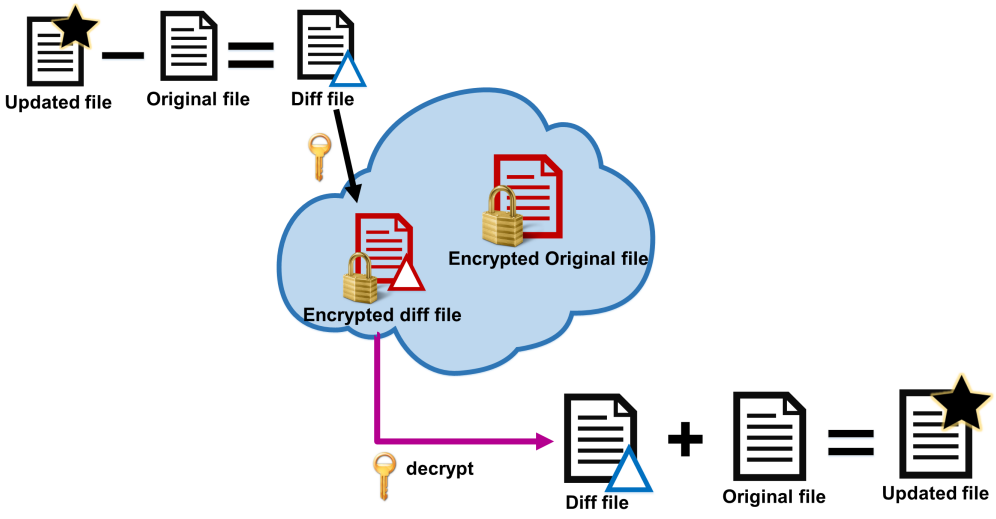

Figure 4.8: OmniShare incremental synchronization for deduplication

Chapter 5

Implementation

This chapter describes the implementation details of OmniShare on two different platforms: Android and Windows.

5.1 Features

As in any software development project, we define user stories [54] that are built around the list of requirements described in section 3.2. Each user story explains a descriptive insight of the feature that we need to implement in order to fulfil the requirements. User stories define how actors interact with the system and why they do so. The purpose of writing user stories is to simplify the high level requirements into concrete, small features that can be developed independently during development phase.

There are two actors, **User** and **Application**. By application, we mean the OmniShare application itself. Features used by the application work automatically without requiring interactions from users.

Each user story has 3 parts:

1. **Story name**: a short name for each user story.
2. **Formal description**: each user story is written using a common template:
   As a {*type of actor*},
   I want {*to perform some task*}
   so that I can {*achieve some goal/benefit/value*}.
3. **Informal explanation**: verbal explanation of each story using informal, less-technical words.

The list of user stories is written below:

Story 1: Linking OmniShare to Dropbox

As a user, I want to link my OmniShare application to my Dropbox account so that I can use Dropbox as my encrypted storage.
In order to use Dropbox as storage, users need to provide their Dropbox credentials to OmniShare. After that, OmniShare can operate (download or upload files) on Dropbox on behalf of users.

Story 2: Upload encrypted file

As a user, I want to encrypt files using OmniShare then upload them to Dropbox so that nobody can access my files without my permission.
OmniShare encrypts all files before uploading to users' Dropbox cloud storage. This story addresses requirement **S2**.

Story 3: Access plaintext file

As a user, I want to download my ciphertext files from my encrypted storage on Dropbox and decrypt them so that I can access their plaintext content.
OmniShare downloads ciphertext files from Dropbox then decrypts them locally so that users can access the plaintext content. This story matches requirement **S3**.

Story 4: Unlink from OmniShare

As a user, I want to unlink OmniShare on a device from Dropbox so that the device can no longer access plaintext files protected by OmniShare domain.
By unlinking the device from OmniShare, the application deletes the local Device Keys. As a result, the device is unable to decrypt the Root Key to decrypt ciphertext files in the container. The application also removes its entry from the domain descriptor file.

Story 5: Request for authorization

As a user, I want to request for authorization from a new device to OmniShare domain so that I can access plaintext files in OmniShare domain from the new device. OmniShare allows user to select a suitable device from the list of previously authorized devices. After that, the new

device sends a request called *admission* to the chosen authorized device and initiates a key distribution protocol. This story is a part of requirement **S3.1**.

Story 6: Authorize a new device

As a user, I want to authorize a new device from an authorized device in OmniShare domain so that I can access plaintext files in OmniShare domain from the new device.
After receiving request admission request from a new device, the selected authorized device runs in the authorization process. As a result, the new device receives Root Key and is added to the domain descriptor file as an authorized device. This story is a part of requirement **S3.1**.

Story 7: Create directory structure

As a user, I want to create directories in my OmniShare domain so that I can put my files inside the new directories
Users can create directory structure inside their OmniShare container. For each newly created directory, OmniShare creates a Directory Key and stores it inside the directory.

Story 8: Check authorization status

As an application, I want to check if I have access to plaintext files inside OmniShare container or not, so that I can request for authorization
After logging into Dropbox account, OmniShare can detect if the current device can access the plaintext files on OmniShare container by checking the content of the domain descriptor file. If it does not have the accessibility, OmniShare informs users to start the authorization process. On the other hand, if the device already has access rights, users can start downloading/uploading files to the encrypted storage. This story is a part of requirement **S3.1**.

Story 9: Get device list

As a user, I want to read the list of authorized devices in my OmniShare domain so that I can choose one to authorize my new device
A new device needs to know about authorized devices listed in the domain descriptor file. It also learns device information such as device name, device

public key, capabilities and device identity. This story is a part of requirement **S3.1**.

Story 10: Detect authorization request

As an application, I want to automatically detect new device authorization request sent to me so that I can start the authorization process
To reduce the number of interactions from users, OmniShare automatically detects authorization requests from a new device. When the new device sends a request to an authorized device, the authorized device needs to react automatically without requiring any interaction from the user. This story is a part of requirement **S3.1** and **U3**

Story 11: Establish OOB channel

As an application, I want to automatically establish an OOB channel between a new device and an authorized device so that I can determine which key distribution protocol to use
OmniShare compares lists of device capabilities from the new device and the select authorized device to select a suitable pair of capabilities for establishing OOB channel. This story addresses a part of requirement **S3.1**, and **U3**

5.1.1 User story analysis

Based on OmniShare requirements and our high level design, we have divided the system into eleven specific user stories. First, users link OmniShare to their Dropbox storage (**story 1**) on their devices. Users can upload ciphertext files (**story 2**), read plaintext files (**story 3**) as well as create directory structures inside OmniShare container (**story 7**) on that device. When users want to use another device, they need to identify their list of authorized devices (**story 9**). Then, users can request for authorization(**story 5**) and accept that request from the selected authorized device (**story 6**). The authorization process is fully automated with the help of **story 8**, **story 10** and **story 11**. Finally, users can unlink an authorized device to remove it from their OmniShare domain when they do not want to use that device any more (**story 4**). These stories combine into a fully functional client-side encryption application for cloud storage (Dropbox). We developed and tested them independently on both Windows and Android (Requirement **A1.1**).

5.2 Software design specification

When users access OmniShare from a device for the very first time, the application creates a Device Key pair. It also creates an *Authentication Key*

5.2.1 Device Key pair

OmniShare Device Keys are 2048-bit RSA key pair. OmniShare stores private part of the key pair securely using platform specific secure storage mechanism, whereas Device public key is listed in the domain descriptor file. In particular, to store the Device private key, we use Android Keystore [40] in Android and Next Generation Cryptography API (CNG) [57] in the Windows implementation.

5.2.2 Authentication key

In each device, OmniShare generates a random 128-bit authentication key to use as the key for **HMAC-SHA256** calculation. Authentication key is generated at the same time as Device Key pair. Similar to Device Key pair OmniShare stores authentication key using platform specific secures storage mechanism.

5.2.3 Domain descriptor file

The domain descriptor file is called *.OmniShare*. OmniShare creates the file and stores it inside the container. The .OmniShare file contains a JSON array. Each array element contains devices specific information: identity, name, public key, capabilities and encrypted Root Key. We encrypt the Root Key with Device Public Key using **RSA with Optimal Asymmetric Encryption Padding scheme (OAEP)**. Each array element is integrity-protected by a **HMAC-SHA256** value using the corresponding Authentication Key of the device. The structure of .OmniShare is described in Listing 5.1 while Table 5.1 defines JSON data for each array element.

Listing 5.1: .OmniShare file JSON structure

```
1 [
2  {
3   "Entry":{
4           "DeviceId":"",
5           "DeviceName":"",
6           "PublicKey":"",
```

```
7           "RootKey":"",
8           "Capabilites":[
9                       {"Name":"",
10                       "Priority":"",
11                       "Direction":""
12                      },
13                      ...
14          ]
15   },
16   "Hmac":""
17  };
18  ...
19 ]
```

Entry	Information for each device in the domain
DeviceId	Unique identify for each device
DeviceName	Name of the device
PublicKey	Public Key of the device
RootKey	Encrypted RK using Device Key
Capabilites	Device's hardware capabilities, Capabilities is a JSON array
Name	Name of the capability: NFC, CAMERA, KEYBOARD or MONITOR
Priority	priority preference of the capability
Direction	direction of data exchange: IN or OUT
Hmac	HMAC value of Entry object

Table 5.1: *.OmniShare* JSON data definition

5.2.4 Key hierarchy

In OmniShare key hierarchy, all keys are 128-bit AES keys. Unlike Root key, OmniShare encrypt all other keys using **128-bit AES in Galois/Counter Mode (GCM) with 96-bit (12 bytes) Initialization Vector and 128-bit (16 bytes) MAC length**. Encrypted keys are stored as follows: 12 bytes Initialization Vector, 16 bytes ciphertext and 16 bytes MAC. The format of an encrypted key is depicted in figure 5.1.

When a new directory is created, OmniShare generates a new Directory Key and encrypts it following the key hierarchy structure. Encrypted Di-

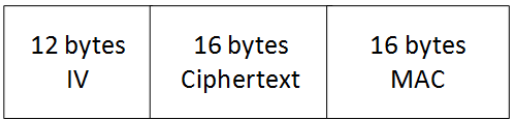

Figure 5.1: Format of an encrypted key

rectory Key is stored in a file named *.envelope.omnishare* inside the newly created directory. When a file f is created, OmniShare encrypts f with a new File Key Kf. Kf is then encrypted using the Directory Key stored in *.envelope.omnishare* file of the directory under which the file is stored. Finally, OmniShare prepends this encrypted File Key to the encrypted file. The format of an encrypted file is depicted in figure 5.2. We also use AES-GCM to encrypt f.

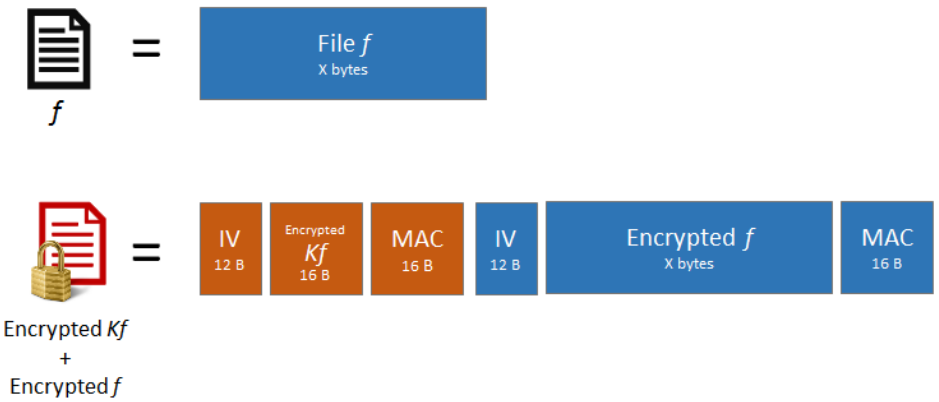

Figure 5.2: Format of an encrypted file

5.2.5 Device authorization work flow

To minimized users' interaction during authorization, we automated most of the authorization steps. In particular, OmniShare only requires users' actions in selecting the authorized device and transferring messages over the OOB channel. The rest of the authorization steps execute over the cloud storage communication channel (section 4.4.1).

5.2.5.1 Message structure

The message structure is described in listing 5.2. Each message is a file with a random Universally Unique Identifier (UUID) [51] as the filename. The file's content is a JSON object where *Body* is the message payload, *Hmac* is an optional keyed hash value of the body to protect its integrity and *AnswerFileName* is the name of the file that the sender is expecting as a response. Similar to the file name, *AnswerFileName* field is also a UUID.

Listing 5.2: Common message JSON structure

```
{
 "Body": "",
 "Hmac": "",
 "AnswerFileName": ""
}
```

5.2.5.2 Expected Polling

When a new device sends the first admission message to an authorized device, we need a mechanism to convey that the selected authorized device needs to response to the request. In particular, the new device uploads files to the cloud storage while the other device has to know that these files are available on the cloud storage to download.

Dropbox has a notification API on Android to allow application to subscribe to a specific Dropbox directory so that whenever there is any new files in the folder, the application can be notified. However, this feature is unavailable on other platforms. We also assume that this feature is not available on all cloud storages. In Windows, there is *FileSystemWatcher* [1] class which listens to the file system change notifications and raises events when a directory, or file in a directory, changes. However, as we experimented with *FileSystemWatcher*, the class is highly unreliable. *FileSystemWatcher* either creates too many events at once or occasionally omits some events. We suspect that there is an internal queue inside *FileSystemWatcher* that holds all events at a time. Therefore, *FileSystemWatcher* omits some events when the queue is full. When users modify files using specific application such as Microsoft Word [2], the application writes multiple chunks of byte at a time sequentially. Thus, *FileSystemWatcher* raises multiple events at once.

[1] https://msdn.microsoft.com/en-us/library/system.io.filesystemwatcher

[2] https://www.microsoft.com/en-us/download/office.aspx

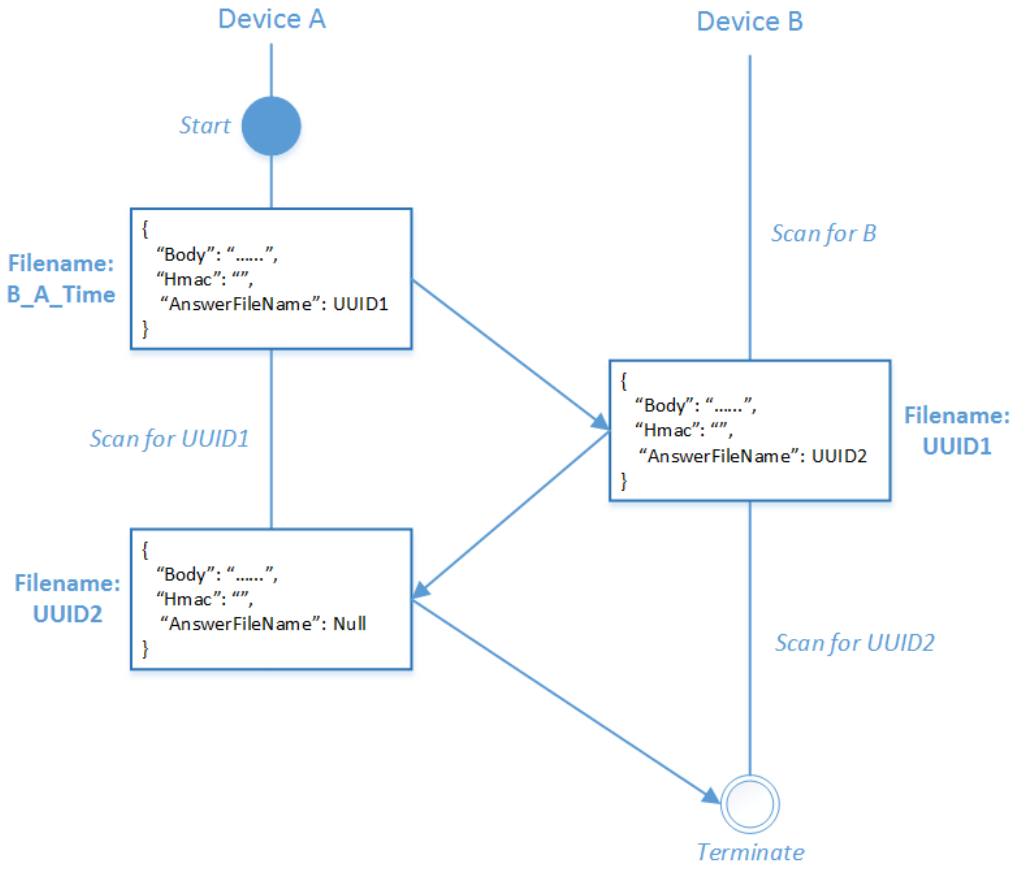

Figure 5.3: Expected polling pattern

For those reasons, we do not rely on notifications neither from the cloud storage API nor operating system. Instead, devices need to frequently poll the cloud storage server to check if there is any message for them to download. However, we cannot expect devices to keep polling the cloud storage server continuously, which may induce communication overhead. We designed "Expected Polling" (EP) as an interactive communication mechanism in cloud storage. Figure 5.3 depicts the mechanism for EP. There are two devices A and B where A want to communicate with B. We have a background process in B which frequently poll the cloud server for files with filename starting with B'. This process runs every 10 seconds so it does not cause communication overhead. The 10 second interval is also called *initial polling interval*

Consider the following scenario: A sends message 1 to B, B replies with

message 2. After receiving message 2, A sends message 3 to B as the last message. The EP pattern starts by A creating the first message as a file with filename B_A_Time where Time is the current timestamp. The AnswerFileName field in message 1 is a random UUID value (UUID1) which A expects as a response file.

Since B actively polls the cloud storage server for files with filename starting with B's identity every 10 seconds, B can recognize message 1 from A. B then constructs message 2 with filename UUID1 and add a new UUID 2 as AnswerFileName. B then uploads the file to the cloud storage.

After uploading the first message, A expects that B will reply with a file named UUID1 as specified. As a result, we can greatly reduce the polling interval from Device A to a *expected polling interval* value such as 5 seconds. Finally, A creates the last message (message 3). Because A does not expect any other reply message, AnswerFileName field can be set to NULL. A places message 3 into a file named UUID2 as specified in message 2 and uploads to the cloud storage.

Similarly, B expects a reply message from A with the name UUID2. B also polls cloud server every 5 seconds for the UUID2 answer. When B receive message 3, it notices that there should not be any further answer in the protocol since the AnswerFileName is NULL, B terminates EP process.

In case messages are lost or corrupted, we use a time-out value of 2 minutes for each expected polling sequence. In short, EP is our communication mechanism designed specifically for the cloud storage communication channel. The mechanism starts with a slightly long interval for the first message, then aggressively reduces the interval for other expected messages. Currently, we choose 10 seconds as the *initial polling interval* so that it is not too long for devices to recognize the admission message. It is also not too short to cause heavy computational overhead for the device. For *expected polling interval*, we choose 5 second window since we estimate that is the reasonable time for a device to upload a small message file (several bytes) to the cloud storage. However, these numbers can be changed depending on the network condition. A better approach would be combining EP and notification from the cloud storage API or operating system if they are available. This way, we can minimize the interval time between each poll.

We applied EP to our key distribution protocols. In both protocols, after the user selects an authorized device B, the new device A creates an Admission Message with file name B_A_Time. This file is called *admission file*. The admission file also follows the common message structure. The body element contains information such as device identification, name, public key and capabilities. There is also an $Extra$ field as a placeholder for any extra information.

Listing 5.3: Admission message JSON structure

```
{
 "Body": {
   "DeviceId":"",
   "DeviceName":"",
   "PublicKey":"",
   "Capabilities":[...],
   "Extra":""
 },
 "Hmac": "",
 "AnswerFileName": ""
}
```

OmniShare then store the file in a directory called *.Admission* then upload to the cloud storage. As we performed expected polling, the rest of the process happens automatically with very short delay between each message. The rest of the protocol runs as described in section 4.4.3 and 4.4.4.

5.2.6 OmniShare Android

5.2.6.1 Environments

We developed OmniShare Android using Eclipse Standard version 4.4 (Luna) with Android Development Toolkit extension 3.5.1 plug-in. The application was tested on multiple Android devices with different OS versions, hardware capabilities and screen sizes. In particular, we tested on Samsung Galaxy S3/S4/S5, HTC One M7 and Google Nexus 4. We also tested the application in a Google Nexus 7 for the case where the device does not have a back camera to scan QR code.

According to Android dashboards [39] provided by Google, Android versions 4.1 (Jelly Bean) and up occupy nearly 90 percent of the Android platform versions market share. For that reason, we built and tested OmniShare using Android SDK version 16 for Jelly Bean. We also tested the application on Android 4.4 (KitKat) and 5.0 (Lollipop).

5.2.6.2 Dependencies

We use these following open source libraries in our implementation: Dropbox sync SDK [35], greenrobot's EventBus [42], gson [41], ZXing [66] and Spongy

Castle [59]. An Android device also needs to have the Dropbox application [3] installed before using OmniShare.

We use Spongy Castle cryptographic library instead of the default library provided by Android. The Android operating system uses a customized version of the Bouncy Castle cryptographic API [15]. However, the bundled API is not up-to-date with the latest version of Bouncy Castle library [4]. For that reason, we decided to use Spongy Castle, which is a repackage of Bouncy Castle API for Android.

5.2.6.3 High level architecture

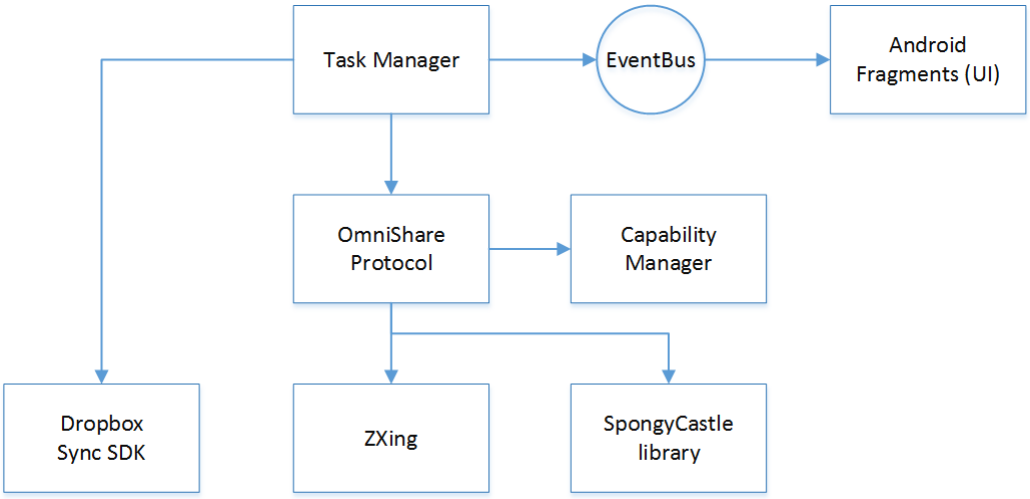

Figure 5.4: Software architecture of OmniShare on Android

OmniShare software architecture for Android is depicted in figure 5.4. The two core modules are *Task Manager* and *OmniShare Protocol*. *OmniShare Protocol* contains logical methods to handle key hierarchy an authorization in OmniShare. These methods are independent of the cloud storage API. OmniShare Protocol calls methods from other components, namely *ZXing*, *SpongyCastle* and *Capabilitiy Manager*.

We use *SpongyCastle* for all cryptographic algorithms. Specifically, we used hash function, Symmetric/Asymmetric cryptographic API, Key generation function, random generator and SRP protocol from *SpongyCastle*.

[3] `https://play.google.com/store/apps/details?id=com.dropbox.android&hl=en`

[4] `http://code.google.com/p/android/issues/detail?id=3280`

Capabilitiy Manager includes methods to detect device hardware capabilities and corresponding OOB channels. In case of QR code channel, we use *ZXing* to generate QR code. For scanning QR code, we make a call to "com.google.zxing.client.android.SCAN" intent of *ZXing* application. In case *ZXing* application is not available on the device, users will be redirected to Google Play to download *ZXing* application before scanning QR code.

Task Manager is a collection of classes that extend Android's *AsyncTask* [5]. These classes provide asynchronous operations, which are directly invoked from the user interface. The module acts as a bridge among user interactions, *OmniShare Protocol* and Dropbox API. It invokes *OmniShare Protocol* to handle logical operations. Interactions with Dropbox such as upload, download or create directory are provided by *Dropbox Sync SDK* for Android. To interact with the user interface, particularly Android Fragments, we use publish/subscribe architecture provided by *EventBus*. After finishing a task, *Task Manager* broadcasts an event to the main EventBus. All Android fragments which subscribe to that *EventBus* will be notified.

5.2.7 OmniShare Windows

5.2.7.1 Environments

We developed OmniShare for Windows using Microsoft Visual Studio 2013. The Windows application is tested on Windows 7, Windows 8.1 and the recent newly released Windows 10. We support both x86 and x64 Windows architecture. We tested primarily on Microsoft Surface Pro and Lenovo Yoga. We target .NET framework version is 3.5 which is pre-installed on Windows 7. For later versions of Windows where .NET 3.5 is not bundled by default, users can download the re-distribution packages through our installation wizard.

5.2.7.2 Dependencies

OmniShare for Windows depends on the following third party libraries: Microsoft Synchronization Framework 2.1 (MSF), Log4net 2.0.3, WPF NotifyIcon 1.0.5, Extended WPF Toolkit 2.3, ZXing.NET 0.14.0.1, Newtonsoft.Json 6.0.3, Open-VCDiff and Bouncy Castle C Sharp 1.7.

MSF is a synchronization platform provided by Microsoft. In our case, we use MSF for periodically synchronization between the plaintext directory and the ciphertext directory. The use of MSF helps us to avoid complexity in implementing periodic synchronization from the scratch. It is also possible

[5] `http://developer.android.com/reference/android/os/AsyncTask.html`

later to extend OmniShare to work with other data sources such as databases instead of only files and directories at the moment.

Various utility libraries are used throughout the project. Log4net is used for logging events. WPF NotifyIcon and Extended WPF Toolkit are used for advanced user interface features such as taskbar icon notification. We use Json.NET for serializing and deserializing JSON data structure.

Similar to OmniShare for Android, we generate QR code using ZXing library and Bouncy Castle for cryptographic operations.

5.2.7.3 High level software architecture

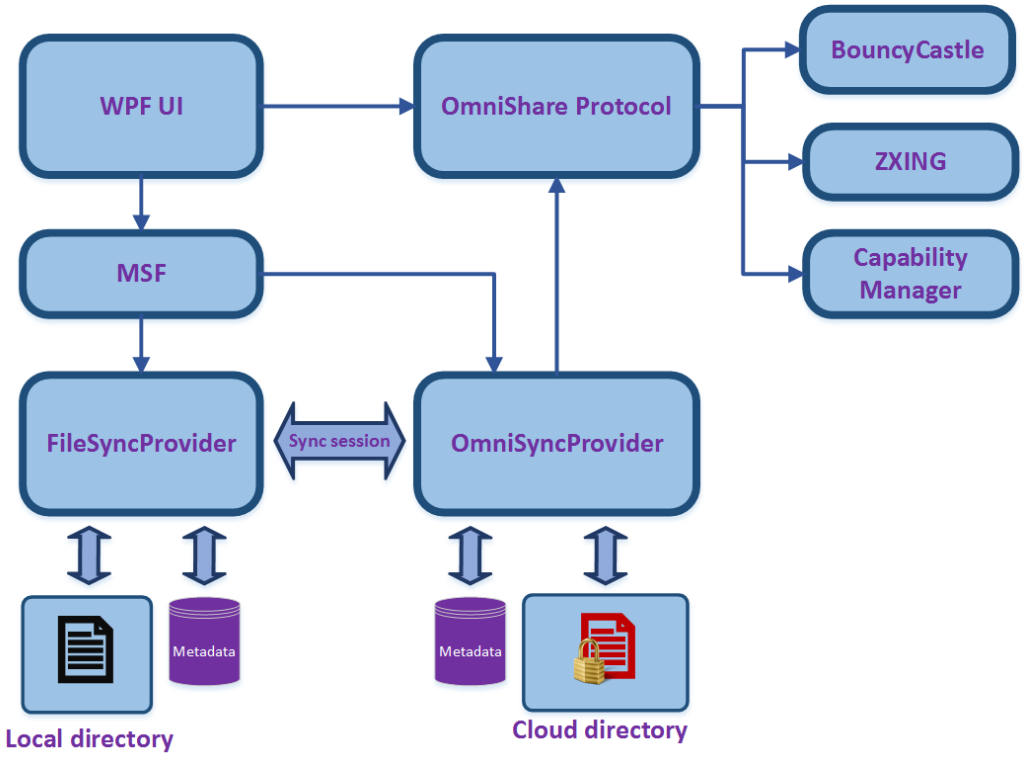

Figure 5.5: Software architecture of OmniShare on Windows

OmniShare software architecture for Windows is depicted in figure 5.4. Similar to OmniShare for Android, the core component is *OmniShare Protocol* which handles key management in OmniShare and is independent of the cloud storage API. *Capabilitiy Manager* discovers device hardware capabilities and matches the capability sets.

The user interface is implemented using *Windows Presentation Foundation* (WPF). Since the .NET framework already provides event handling as well as asynchronous mechanism, we do not have to rely on external interfaces like *EventBus* or *AsyncTask* as in Android. Instead, WPF controllers call directly to *OmniShare Protocol*.

As explained in 4.5.2, OmniShare on Windows uses periodic synchronization pattern to synchronize user files with Dropbox directory. In fact, we develop periodic synchronization based on *Microsoft Synchronization Framework* (MSF) [9]. MSF is a synchronization platform that enables developers to synchronize between various data storage types via different protocols and network channels. MSF enables developers to build a synchronization ecosystem by concentrating only on the sync logic.

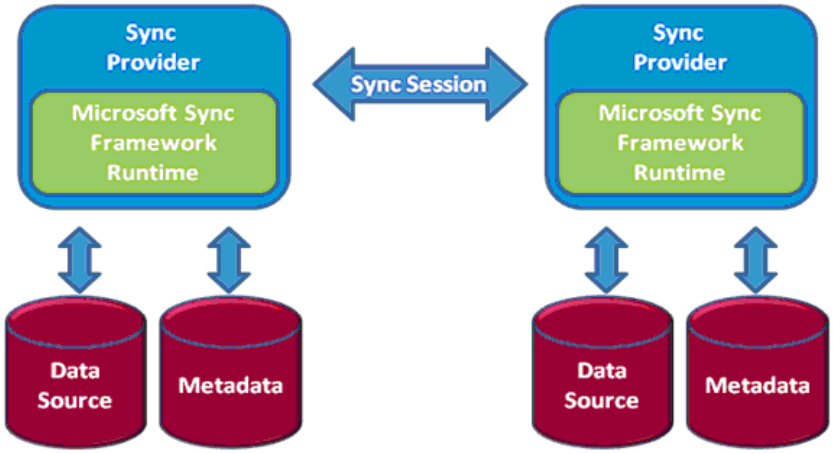

Figure 5.6: Microsoft Synchronization Framework core components [6]

Figure 5.6 shows core components of MSF and how they interact with each other. There are three basic building blocks: *Data Source*, *Metadata* and *Sync Provider*. *Data Source* is the container holding data that needs to be synchronized. In OmniShare context, *Data Source* is the plaintext directory and the ciphertext directory. However, there is no limit to the type of *Data Source*, it can also be a database or Web Service. Another building block is *Metadata* which contains information about the stored data in *Data Source*. *Metadata* can be stored in a file, database or inside the data itself. *Sync Provider* contains synchronization logic and policy. Based on Metadata, the Sync provider detects the changes in *Data Source* and passes it to the MSF runtime. The runtime takes care of exchanging that information in a *Sync Session* to other *Sync Provider*. After both Providers know what the changes are in *Data Source*, they can run their own logic to deal with those

changes.

FileSyncProvider is a built-in provider component for synchronizing files. *FileSyncProvider* detects changes that are made to the subscribed file or folders based on one of these attributes: last time modification, file hash (optional), file size, file/folder name (case-sensitive) and file attributes. It also guarantees the consistency of the synchronization process. We use *FileSyncProvider* as the provider for our plaintext directory.

We develop a custom *Sync Provider* called *OmniSyncProvider*, that behaves similar to the *FileSyncProvider* in detecting file changes from the encrypted folder and works seamlessly with the built-in *FileSyncProvider*. *FileSyncProvider* subscribes to the plaintext directory while *OmniSyncProvider* subscribes to the ciphertext directory.

5.2.7.4 Incremental synchronization

We developed a proof-of-concept on Windows to demonstrate the incremental synchronization approach. Figure 5.7 depicts our implementation. We extend *OmniSyncProvider* to be *OmniDeltaSyncProvider*. Apart from the plaintext directory and the encrypted OmniShare directory on the cloud storage, we have two more directories in the user's device called cache directory and delta directory. Basically, the cache directory is a replicate of the plaintext directory. Whenever users modify a file $f1$ in the plaintext directory to get $f2$, *OmniDeltaSyncProvider* executes open-vcdiff encoding algorithm with $f1$ as the source and $f2$ as the target to get *f.delta2*.

OmniDeltaSyncProvider then encrypts *f.delta2* and stores on the cloud storage directory as $f.delta2.os$. After that, *OmniDeltaSyncProvider* updates the cache directory by replacing $f1$ with $f2$. All information about delta, cache and plaintext files along with their encrypted counterparts on the cloud storage are stored in the metadata database.

However, there are several issues with *OmniDeltaSyncProvider*. First, storage size of the user's device is doubled due to cache files. Secondly, due to time constraint, we can only implement *OmniDeltaSyncProvider* as a proof-of-concept. Our implementation is not robust, i.e. the provider often crashes when there are many file modifications at once. Thirdly, In order to extract the delta, *open-vcdiff* has to read both $f1$ and $f2$ at the same time into the device memory. Thus, it is inefficient to work with large files. For those reasons, even though we might reduce the network communication overhead by only uploading delta files, our current implementation is error-prone and introduces significant local storage as well as computation overhead.

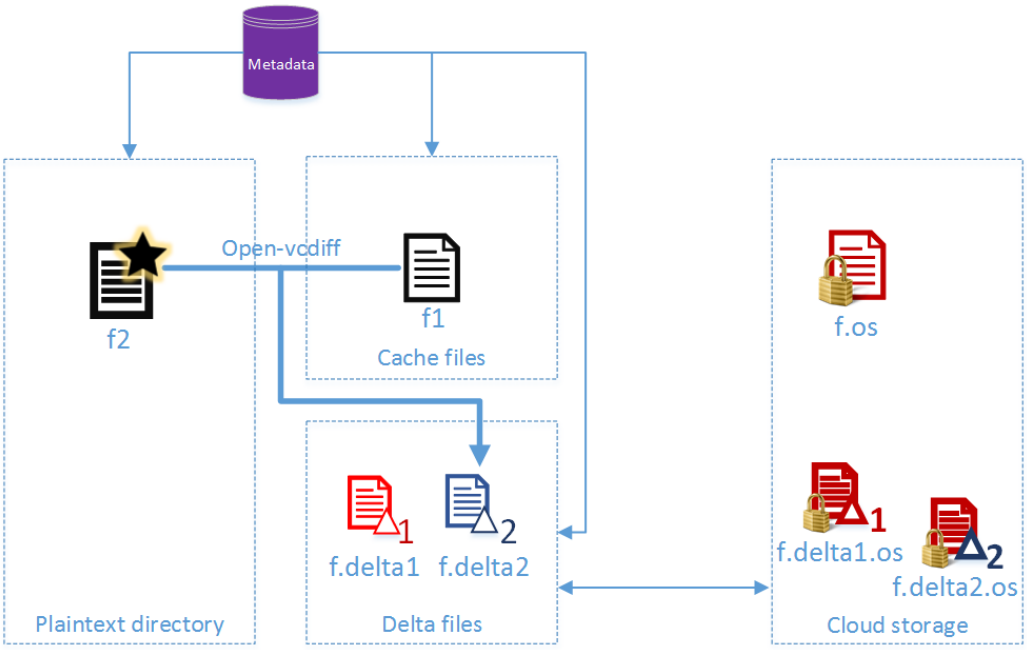

Figure 5.7: Incremental synchronization implementation

Chapter 6

Evaluation

6.1 Security evaluation

6.1.1 File encryption and key hierarchy

OmniShare uses standard cryptographic libraries for all cryptographic operations. We use authenticated encryption (AES-GCM) to encrypt files and keys in the key hierarchy. Therefore, OmniShare assures the confidentiality as well as the integrity of files on cloud storage (requirement **S2**).

Security of the Root Key is crucial to protect key in the key hierarchy. We encrypt the Root Key for each authorized device with its device public key. Therefore, only the authorized device can access the device private key, decrypt the Root Key and subsequently, decrypt keys in the key hierarchy. All keys in OmniShare are strong client-generated keys (requirement **S1**). OmniShare stores the key hierarchy on the cloud storage and distributes the keys using key distribution protocols which make use of the cloud storage communication channel and secrecy OOB channels. Therefore, OmniShare does not rely on any third-party server (requirement **S3.2**).

6.1.2 Device authorization

OmniShare protects users' data in cloud storage using a key hierarchy starting with Root Key (RK). **S3.1** requires that our key distribution protocols are secure. Thus, our key distribution mechanism must protect the secrecy and integrity of RK. The primary goal of our key distribution mechanism can be informally stated as follow. After key distribution, the selected authorized device believes that it hands over the key to the legitimate new device. On the other hand, the new device receiving the key believes that it receives the correct key from the user-selected authorized device. Formally, let A be the

new device and B be the selected authorized device. We consider the *Mutual Implicit Root Key authentication* security property (section 2.6.2) which provides assurance for B that A is the only other device that can possibly receive RK. Similarly, in the case of A, the property ensures the origin of the received RK so that A cannot be tricked to use a compromised RK.

In key distribution using QR code, we achieve implicit key authentication using data origin authentication technique: A's public key is authenticated using its cryptographic hash transferred to B via the secure OOB channel. Thus, RK is protected by an authenticated public key that can only be decrypted by the intended participant in the protocol. Subsequently, B's data origin is guaranteed through the use of message authentication code based on the randomly generated session key.

In key distribution protocol using SRP, the session key is generated using a widely recognized authenticated key establishment protocol SRP. Similar to key distribution protocol using QR code, the established share session key protects the origin of RK.

In both protocols, we exclude key confirmation for RK. Firstly it is important for A to assure the origin of RK and B to not reveal RK to adversaries. However, it is not necessary for B to confirm that if A has received RK or not. Instead, A will display the result directly to users. Secondly, RK confirmation requires another round of message exchange. Thus, introducing RK confirmation from A to B increases the protocol execution time which has a direct impact to the overall user experience. For those reasons, we avoid key confirmation and explicit key authentication for the RK. [55] also suggests considering **Forward secrecy** property when one of the long-term Device Key or RK is compromised. However, in the scope of this thesis, we assume that user devices are trusted so we can exclude **Forward secrecy** property. We also do not address other security issues. For example: errors such as deadlocks or weakness in the cryptographic algorithm itself. Instead, we assume that cryptographic algorithm implementation we used from BouncyCastle are correct.

6.1.2.1 Formal verification of key distribution via QR code

As explained in section 2.6, we use Scyther to formally prove our designed protocols. Listing 6.1 describes the key distribution via QR code protocol model. Global hash function H and asymmetric key pair $pk2, sk2$ are declared first. Scyther has predefined public key infrastructure (PKI). More specifically, each agent X has access to a long-term private key $sk(X)$ and a public key $pk(X)$ which is already shared among all agents. Similarly, predefined symmetric key infrastructure where $k(X, Y)$ denotes the long term

symmetric key shared between agent X and Y. However, in our protocol, there is no pre-established PKI. Thus, we defined an additional asymmetric key pair in order to model the key distribution process. $pk2$ is a public constant function that takes an agent identity as parameter. $pk2(X)$ then denotes the public key of agent X. $sk2$ is declared as secret function which also take an agent identity as its parameters. The output is the secret value $sk2(X)$ serving as the long-term secret key known only by agent X. $pk2$ maps to $sk2$ via *inversekeys* key word that allows value encrypted using $pk2(X)$ to be decrypted using $sk2(X)$.

Listing 6.1: Key distribution via QR code role script for Scyther

```
1  hashfunction H;
2  const pk2: Function;
3  secret sk2: Function;
4  inversekeys (pk2,sk2);
5
6  protocol QRP(A,B)
7  {
8    role A{
9      fresh sessionKey: Nonce;
10     var rootKey: Nonce;
11     var MAC: Ticket;
12     fresh A1: Nonce; //Nonce to correct Scyther
          behaviour of interpreting asymmetric key
13
14     send_1(A,B, pk2(A1));
15     send_2(A,B,{sessionKey, H(pk2(A1))}k(A,B));
16     macro m = {rootKey}pk(A);
17     recv_3(B,A, (m, MAC));
18     match(MAC, H(m, sessionKey));
19   }
20
21   role B{
22     var hash: Ticket;
23     var sessionKey: Nonce;
24     fresh rootKey: Nonce;
25     var A1: Nonce;
26
27     recv_1(A,B,pk2(A1));
28     recv_2(A,B,{sessionKey, hash}k(A,B));
29     match(hash, H(pk2(A1)));
```

```
30
31      macro m = {rootKey}pk(A);
32      send_3(B,A, (m, H(m, sessionKey)));
33    }
34  }
```

Since we declare our own asymmetric key pair, we can encrypt different *terms* using the following syntax $\{term\}pk2(X)$. The syntax denotes that the term value is encrypted using our defined public key of agent X. Thus, theoretically, only agent X who possess secret key $sk2(X)$ can decrypt the term. However, Scyther interprets the syntax as symmetric encryption instead of asymmetric encryption as we modelled. In other words, Scyther allows any agents use $pk2(X)$ to decrypt the term, even though $pk2$ is a public key.

To solve this problem, we apply a work around, suggested by Cas Cremers, the author of Scyther, that forces Scyther to interpret our syntax as asymmetric encryption using a fresh nonce $A1$. This way, $pk2(A1)$ is interpreted correctly as a public key from role A.

In order to model OOB communication, we use the Scyther's predefined symmetric key system. The following syntax, $\{term\}k(A,B)$, indicates the term value is symmetrically encrypted using the mutual secret symmetric key between A and B. As we assume attackers cannot sniff or tamper with the OOB channel; we model the QR code scanning process as sending the QR code data encrypted using $k(A,B)$. Thus, only A or B can decrypt the QR code data.

After receiving $pk2(A1)$ and the hash, B runs a *match* event to verify the public key. At this point, A and B have completed the public key distribution step from A to B. Therefore, we can use the keys in the pre-established PKI in the last step of the protocol.

Listing 6.2: claim events of the key distribution via QR code protocol

```
1  ...
2  protocol QRP(A,B)
3  {
4    role A{
5      ...
6      claim_a1(A,Secret,sk2(A1));
7      claim_a5(A,Secret,sessionKey);
8      claim_a6(A,Secret,rootKey);
9      claim_a8(A,Nisynch);
10     claim_a10(A,Reachable);
```

```
11    }
12
13    role b{
14      ...
15      claim_b1(B,Secret,sessionKey);
16      claim_b2(B,Secret,rootKey);
17      claim_b4(B,Nisynch);
18      claim_b6(B,Reachable);
19    }
20  }
```

Listing 6.2 captures the claims we made for the formal verification of key distribution using QR code. We want to make sure that *sessionKey* and *rootKey* remains secret during the protocol. Besides that, *Nisynch* and *Reachable* claims are also added to complete the security proof of the protocol. The complete role script for key distribution protocol using QR code is listed in Appendix A.1.

6.1.2.2 Formal verification of key distribution using SRP

In our key distribution using SRP protocol, there are many modular exponentiation operations involved. As we mentioned, Scyther requires helper functions in order to model these operations correctly. We introduced three support protocols in Listing 6.3, 6.4 and 6.5.

A. Modular exponential helper protocol

Listing 6.3: Helper protocol to simulate modular exponential equivalent

```
1  hashfunction g1, g2, H;
2  function f, plus;
3
4  protocol @exponentiation(BE, BM1, AM2)
5  {
6    role BE{
7      //Simulate (g^a)^b = (g^b)^a
8      var a,b: Ticket;
9      recv_!1(BE,BE,g2(g1(a),b));
10     send_!2(BE,BE,g2(g1(b),a));
11   }
12 ...
```

The first support protocol is a modular exponentiation equality to simulate

$$g^{ab} = g^{ba} \tag{6.1}$$

in a finite field $GF(N)$. Let $g1(x)$ denotes g^x and $g2(x, y)$ denotes x^y. Since g^{ab} can be written as $(g^a)^b$, equation 6.1 is equivalent to:

$$g2(g1(a), b) = g2(g1(b), a)$$

B. M1 equality helper protocol

Listing 6.4: Helper protocol to simulate M1 and M1' equivalent

```
...
  role BM1{
    //Simulate M1 equality
    var alpha, beta, a,b,x: Ticket;
    recv_!3(BM1, BM1, H(alpha,beta, H(f(g2(g1(b),a
       ), g2(g1(b),x)))));
    send_!4(BM1, BM1, H(alpha,beta, H(f(g2(g1(a),b
       ), g2(g1(x),b)))));
  }
...
```

The second support protocol is to model equality of $M1$ and $M1'$ messages of the protocol. In particular, we want to model

$$H(\alpha, \beta, K_{ses}) = H(\alpha, \beta, K'_{ses}) \tag{6.2}$$

K_{ses} is the hash of σ which is defined as

$$\sigma = (\beta - kg^x)^{a+ux} = g^{ba}.g^{bux}$$

While K'_{ses} is the hash of σ' which is defined as

$$\sigma' = (\alpha.v^{u'})^b = g^{ab}.g^{xu'b}$$

Since u and u' are both equal to $H(\alpha, \beta)$, they are syntactically equal. We can remove them from the approximating equality helper protocol. Let $f(a, b)$ denotes the multiplication of a and b. Thus, the helper protocol for $M1$ needs to provide approximating equality of the following equation:

$$\begin{aligned} &H(\alpha, \beta, H(f(g2(g1(b), a), g2(g1(b), x))) \\ &= H(\alpha, \beta, H(f(g2(g1(a), b), g2(g1(x), b))) \end{aligned}$$

C. M2 equality helper protocol

Listing 6.5: Helper protocol to simulate M2 and M2' equivalent

```
...
  role AM2
  {
    //Simulate M2 equality
    var alpha, beta, a, b, x, RK: Ticket;
    macro keyA = f(g2(g1(b),a), g2(g1(b),x));
    macro keyB = f(g2(g1(a),b), g2(g1(x),b));
    macro M1A = H(alpha, beta, H(keyA));
    macro M1B = H(alpha, beta, H(keyB));
    macro M2A = H(alpha, M1A, keyA);
    macro M2B = H(alpha, M1B, keyB);
    recv_!5(AM2,AM2,{RK}keyB, M2B);
    send_!6(AM2,AM2,{RK}keyA, M2A);
  }
}
```

This helper protocol is used to model $M2$ and $M2'$. Basically, we want to model the following equation:

$$H(\alpha, M1', K'_{ses}) = H(\alpha, M1, K_{ses})$$

As we already modeled $M1$, $M1'$, K_{ses} and K'_{ses} in M1 equality helper protocol, it is straightforward to re-use the modeled values in this helper protocol.

D. Main protocol

Listing 6.6: Key distribution using SRP role script for Scyther

```
hashfunction g1, g2, H;
function f, plus;

protocol SRP(A,B)
{
  role A{
    fresh P, a: Nonce;
    var s, beta, v, RK: Ticket;
    macro x = H(s,P);
    macro key = f(g2(beta,a), g2(beta,x));
    macro M1 = H(g1(a), plus(beta, v), H(key));
    macro M2 = H(g1(a), M1, key);

    send_1(A,B, g1(a));
```

```
15      send_2(A,B,{P}k(A,B));
16      recv_3(B,A,plus(beta,v),s);
17
18      match(v, g1(x));
19      send_!4(A,B, M1); //Sending out M1
20      recv_!5(B,A, {RK}key,M2);
21    }
22
23    role B{
24      fresh b, s, RK: Nonce;
25      var alpha, s, P;
26      macro x = H(s,P);
27      macro key = f(g2(alpha,b), g2(g1(x),b));
28      macro M1 = H(alpha, plus(g1(b), g1(x)), H(key)
          );
29      macro M2 = H(alpha, M1, key);
30
31      recv_1(A,B, alpha);
32      recv_2(A,B,{P}k(A,B));
33      send_3(B,A, plus(g1(b),g1(x)), s);
34      recv_!4(A,B, M1);
35      send_!5(B,A, {RK}key, M2);
36     }
37  }
```

Listing 6.6 shows the role script that we used to model our key distribution using SRP. The script uses the same syntax as described in figure 4.6 except that we use *key* to denote K_{ses} since Scyther does not allow underscore. In message 4, A sends out $M1$ which is a macro defined by $H(g1(a), plus(beta, v), H(key))$. When B receives $M1$, B triggers the M1 equality helper protocol. As a result, it also receives $M1'$ which is $H(alpha, plus(g1(b), g1(x)), H(key))$. Message 5 follows the same procedure except that it triggers M2 equality helper protocol.

Listing 6.7: claim events of the key distribution using SRP protocol

```
1
2  protocol SRP(A,B)
3  {
4    role A{
5      ...
6      claim_a1(A, Reachable);
```

```
7       claim_a2(A, Secret, a);
8       claim_a3(A, Secret, P);
9       claim_a4(A, Secret, key);
10      claim_a5(A, Secret, RK);
11      claim_a6(A, Nisynch);
12    }
13
14    role B{
15      ...
16      claim_b1(B, Reachable);
17      claim_b2(B, Secret, b);
18      claim_b3(B, Secret, P);
19      claim_b4(B, Secret, key);
20      claim_b5(B, Secret, RK);
21      claim_b6(B, Nisynch);
22     }
23  }
```

Listing 6.7 captures the claims we made for the formal verification of key distribution using SRP. Basically, we want to make sure that all SRP secret values, namely a, b, P, key as well as the RK remain secret during the protocol. Besides that, *Nisynch* and *Reachable* claims are also added to complete the security proof of the protocol. The complete role script for key distribution protocol using QR code is listed in Appendix A.2.

6.1.2.3 Result of formal verification

Figure 6.1 and 6.2 shows Scyther results after verifying our modelled protocols. All claims are successfully passed. In particular, we proved that Root Key is secret, only known by A and B after running both protocols. **Nisynch** claim also ensures that B to A ran the protocols in a correct message sequence and exchanged the intended variables. For these reasons, both key distribution protocols ensure mutual implicit Root Key authentication. We conclude that our key distribution protocol is secure (Requirement **S3.1**).

6.2 Performance evaluation

We performed performance evaluation for OmniShare on two aspects. The execution time of key distribution protocols and cryptographic operations. Measurements are performed on a Samsung Galaxy S6 running Android 5.1

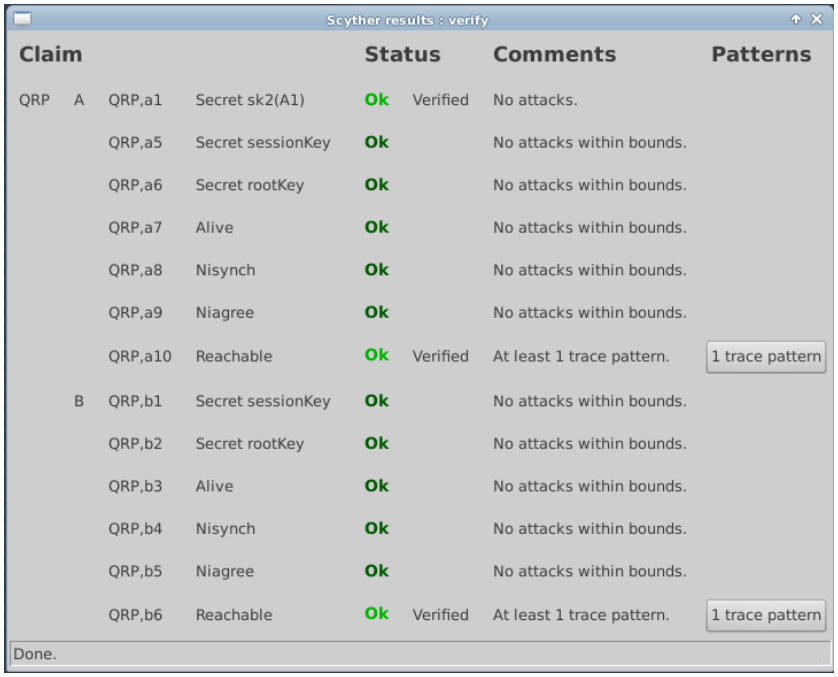

Figure 6.1: Scyther result of verifying key distribution via QR code protocol

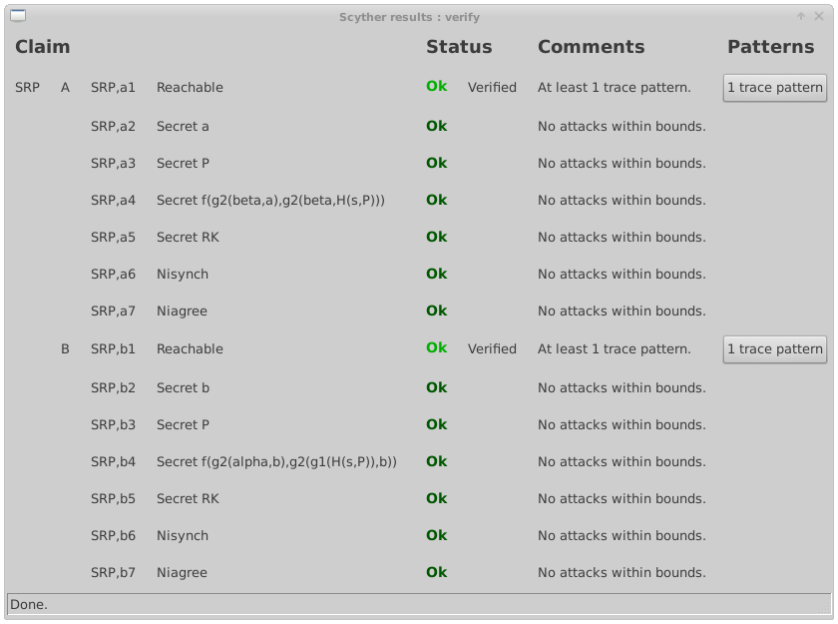

Figure 6.2: Scyther result of verifying key distribution using SRP protocol

with Quad-core 1.5 GHz Cortex-A53 & Quad-core 2.1 GHz Cortex-A57 and a Microsoft Surface Pro running Windows 10, core i5 1.7 GHZ.

6.2.1 Key distribution protocols

Table 6.1: Key distribution protocol execution time (seconds) with Dropbox

Key distribution protocol	Average time (seconds)
Using QR code	
Windows - Android	16.31 (± 2.37)
Android - Android	21.66 (± 1.10)
Overall	19.72 (± 3.20)
Using SRP	
Windows - Android	39.77 (± 4.08)
Android - Windows	36.68 (± 9.21)
Windows - Windows	45.10 (± 5.71)
Android - Android	41.01 (± 3.94)
Overall	40.72 (± 6.03)

Table 6.2.1 shows execution time for our key distribution protocols. For example: Windows - Android means that the new device is a desktop running Windows and the authorized device is an Android phone. It includes message generation and exchange time via cloud storage but does not include time for user interaction over OOB channels. For expected polling, we use 10 seconds for *initial polling interval* and 5 seconds for *expected polling interval*.

Generally, key distribution protocol using QR code is twice as fast as using SRP due to fewer message exchange rounds. In particular, key distribution via QR code has two message rounds while key distribution using SRP has four rounds.

We also measured the execution time while reducing the *expected polling interval* time to 1 second. However, the total execution time does not change much in comparison with the 5 second, i.e. 22.43 ± 2.05 for key distribution protocol using QR code in Android - Android setup. Therefore, further performance evaluation is needed in order to improve the execution time of key distribution protocols.

6.2.2 Cryptographic operations

Table 6.2: Cryptographic operations time

Operation	Average time (milliseconds)
Windows	
2048-bit RSA keygen	93.0 (± 34.5)
RSA encryption (16 bytes RK)	<1
RSA decryption (16 bytes RK)	13.0 (± 1.4)
File encryption (1MB)	277.0 (± 19.5)
File decryption (1MB)	265.5 (± 14.3)
Android	
2048-bit RSA keygen	395.6 (± 184)
RSA encryption (16 bytes RK)	15.2 (± 3.96)
RSA decryption (16 bytes RK)	31.4 (± 2.79)
File encryption (1MB)	211.6 (± 16.27)
File decryption (1MB)	235 (± 4.47)

Table 6.2.2 shows execution time for each cryptographic operation in OmniShare for both Android and Windows. We evaluate time for generating device RSA key pair, encryption/decryption time using RSA for 128-bit root key and symmetric encryption/decryption of 1 MB file using AES-GCM. Other operations such as generating AES 128-bit key or encrypting File Keys using AES are relatively fast on both platforms (less than 1 millisecond).

6.3 Usability evaluation

In OmniShare, we ensure that all messages that are displayed to users are consistent on both platforms (**U1**). For example, figure 6.3 shows identical user instructions on both platforms.

OmniShare optimizes user experience by minimizing user interactions during device authorization (**U3**). Figure 6.4 depicts work-flow for authorizing a new device from a user perspective. To authorize a new device, OmniShare only requires two actions from users which are selecting a previously authorized device and transferring information over OOB channel. This work-flow is consistent on all platforms (**U1**). Both OOB channels are widely used and have least security failures according to Kainda et. al. [46] (**U2.1**).

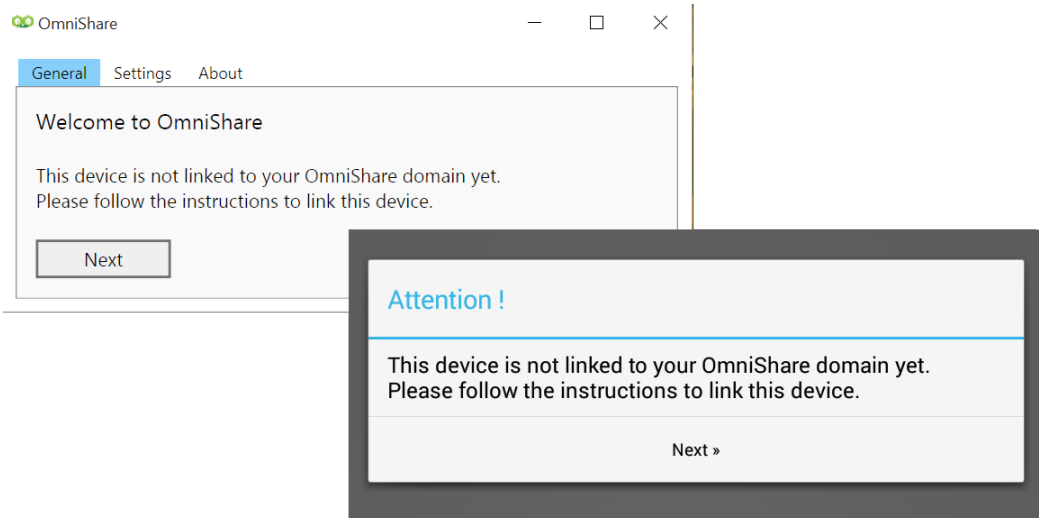

Figure 6.3: OmniShare consistent instructions between platforms

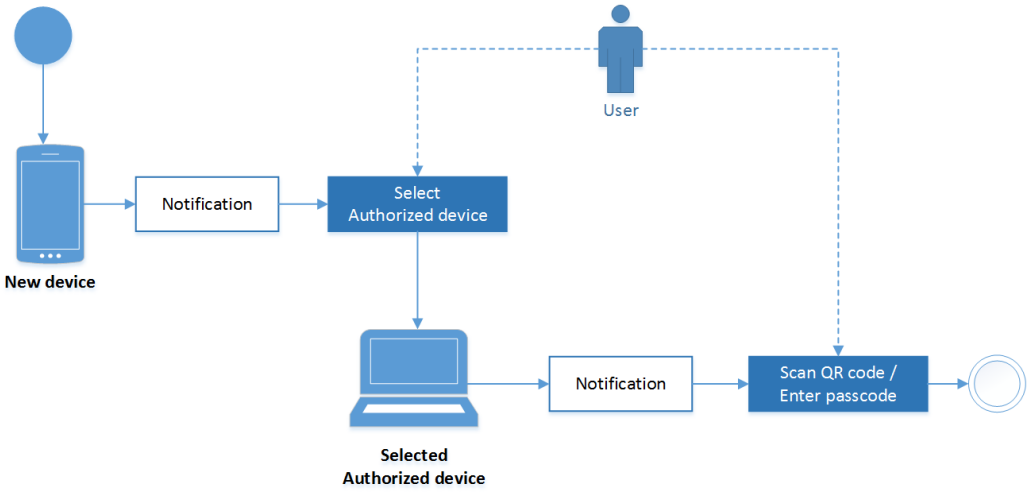

Figure 6.4: Work flow for device authorization from user perspective

We also designed OmniShare to behave similar to other cloud storage clients and client-side encryption utilities. On Windows, OmniShare is displayed as a small icon in users' taskbars (similar to Dropbox, OneDrive). Similar to Viivo, we automatically synchronizes a plaintext directory in users' machines to a ciphertext directory on the cloud storage. On Android, we present a file browser where users can browse directories in their encrypted

storages as well as download/upload files which is similar to BoxCryptor. These features are used commonly on other products that users are familiar with (**U2.1**).

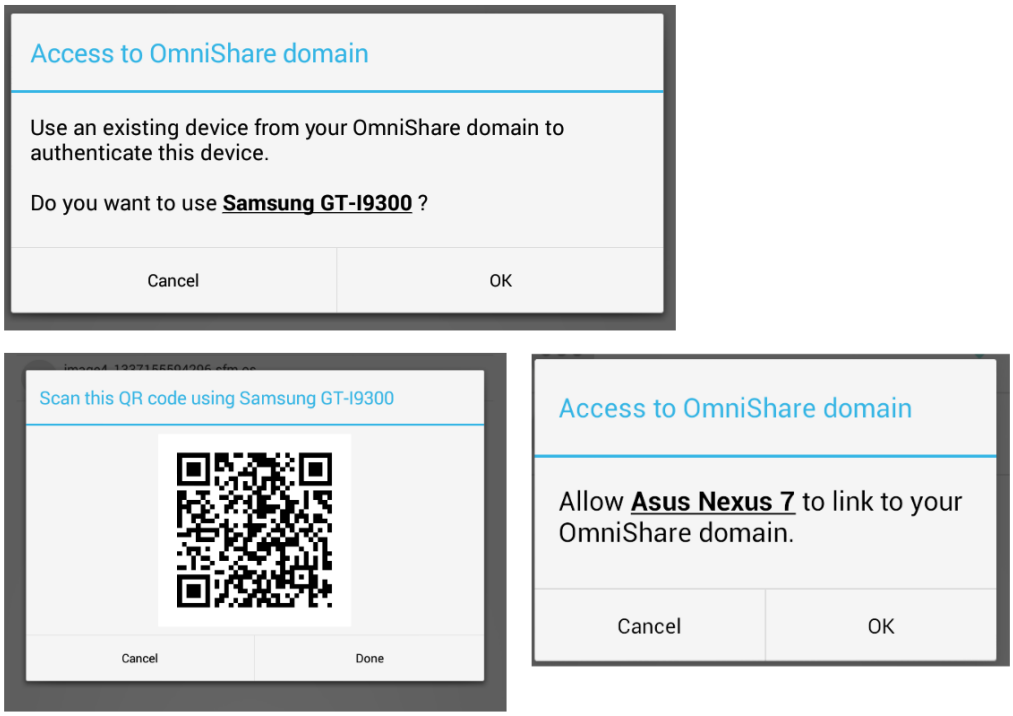

Figure 6.5: OmniShare sample dialog pop-ups

Finally, OmniShare present users with clear and simple instructions (**U2.2**). Most of OmniShare instructions are related to the authorization process. Figure 6.5 shows examples of pop-up dialogs that give instructions to users. Instructions are given as a sequence of pop-up dialogs. This feature guides users through a series of yes/no questions to simplify the authorization process.

6.4 System evaluation

We developed OmniShare on two widely used platforms, i.e. Windows 7 and Android 4.2. The application also works with devices which have different hardware capabilities, i.e. camera, monitor, keyboard (Requirement **A1.1**). The application currently only works on top of Dropbox. However, it is feasible to extend the application to work with other cloud storages, e.g. Google

Drive or OneDrive, since they have interfaces for third-party development (Requirement **A1.2**).

We designed and implemented OmniShare using loosely software modules where each module is designed and tested independently of other modules. We also use popular third-party framework such as *MSF* or *EventBus* to organize the structure of the code. For this reason, the application is easy to extend and maintain (Requirement **A2**).

6.5 Summary

Table 6.3 summarizes our evaluation result of OmniShare. It shows the predefined requirements of the system and the corresponding section where we evaluate the requirements.

Requirement	Section
S1. Strong client generated keys	section 6.1.1
S2. Authenticated file encryption	section 6.1.1
S3.1. Secure key distribution	section 6.1.2.3
S3.2. Avoiding third-party servers	section 6.1.1
U1. Consistent user experience	section 6.3
U2.1. Familiar user experience	section 6.3
U2.2. Clear instruction	section 6.3
U3. Minimum interaction	section 6.3
A1.1. Multiple device types	section 6.4
A1.2. Multiple cloud storage	section 6.4
A2. Extensibility and maintainability	section 6.4
A3. Reducing network overhead	section 5.2.7.4

Table 6.3: Summary of evaluation

Chapter 7

Related work

In this chapter, we take a closer look at some popular client-side encryption products and services. In particular, we investigate on how they encrypt users' data, what is their security mechanism as well as their drawbacks and potential risks while using them.

7.1 Product selections

Our study does not aim at providing comprehensive knowledge about existing cloud secure storage products or rating them. Instead, we choose a few popular products to analyze their approaches towards securing users' data. There are two categories for selections: (i) cloud storage service which offers client-side encryption and (ii) client utilities providing client-side encryption for other cloud storage. In particular, in category (i), we choose SpiderOak, Wuala and Mega. Meanwhile, in category (ii), there are Viivo, Boxcryptor, Sookasa, TrueCrypt, EncFS and PanBox.

Our methods for analyzing these products include: install and use the products, read their technical reports [24] or gather information from frequent-to-ask (FAQ) sections of the product manuals or support forums.

7.2 Cloud storage services offering client-side encryption

7.2.1 SpiderOak

SpiderOak [14] is an American-based cloud storage service. SpiderOak provides client applications on multiple platforms such as Windows, Mac, Linux,

Android and iOS. The client applications encrypt both file content and meta-data (i.e. file name, size or created date) using keys derived from users' passwords.

To use SpiderOak client application, users must create a password. The application then uses a key derivation function to generate a strong cryptographic key from the password called the *master key*. Files are encrypted with cryptographically strong keys called *file keys*. *Master key* is used to encrypt *file keys*. The encrypted *file keys* are stored along with the ciphertext files on SpiderOak servers. Meanwhile, the actual password is not stored anywhere. For this reason, as long as the password is safe, plaintext files are secure.

SpiderOak uses standard cryptographic algorithms to ensure confidentiality, authenticity and integrity of users' data. In particular, they use AES in CFB mode with 256-bit key length for encryption, HMAC-SHA256 for integrity protection. To derive keys from passwords, SpiderOak uses PBKDF2 with SHA256, minimum 16384 rounds and 32 bytes of salt.

SpiderOak enables deduplication within user domain [36]. Basically, if users upload the same file, the client application notices and provides the same encryption key that was previously used. Thus, the ciphertext is the same.

However, there are some problems we experienced while testing SpiderOak. First, the client application does not have any policy for creating password. That means password such as "12345" can be freely used. Second, since password is not stored on the server, it is impossible to recover data if users forget their passwords. Finally, SpiderOak allows users to login to their web application via web browsers using passwords and download plaintext files. During this process, encrypted files are decrypted temporarily on SpiderOak server thus raising our concern about the claimed "security and privacy" argument of the service.

7.2.2 Wuala

Wuala [18] is a European-based cloud storage service. Wuala uses PBKDF2 for deriving master keys from users' passwords as well as AES for encrypting files and meta-data. Unlike SpiderOak, Wuala uses convergent encryption to enable deduplication. As mentioned in section 2.7, convergent encryption is not semantically secure.

Wuala shares similar problems with SpiderOak. Wuala allows users to create "weak" password. They cannot restore data when users forget passwords. They also allow users to decrypt and download plaintext files from the server.

7.2.3 Mega

Mega [7] is a New Zealand based cloud storage service. Mega has the same security mechanism as SpiderOak. However, Mega does not provide client-side application. Instead, all encryption and decryption operations are performed locally using JavaScript on users' web browsers.

7.3 Client utilities providing encryption for cloud storage

While SpiderOak, Wuala and Mega are cloud service providers which offer client-side encryption to their clients. There are products acting as client utilities that encrypt client's data before uploading to other cloud services.

7.3.1 Boxcryptor

Boxcryptor [1] is a client-side encryption utility that works with a wide range of cloud storages: Dropbox, Google Drive, OneDrive, etc. Boxcryptor also supports *Web Distributed Authoring and Versioning* (WebDAV) [28]. WebDAV is a standard for collaborative authoring on the web. The standard contains a set of extension to the HTTP protocol that allows clients to create, modify or move files and meta-data (e.g. authors, modification date) on servers. Basically, Boxcryptor can work with any cloud storage service which uses WebDAV such as ownCloud [11]. Boxcryptor is available on multiple platforms such as Windows, Mac, Linux, Android and iOS.

Users first install both Boxcryptor and cloud storage client applications. Users then choose the cloud service they want Boxcryptor to work with. After that, Boxcryptor client application automatically takes care of encrypting files before passing the files to the cloud storage client application to upload.

Similar to other products that we mentioned so far, Boxcryptor uses standard cryptographic algorithms such as AES, PBKDF2 and HMAC-SHA 512. However, Boxcryptor hosts a central server for key management only, not for storing the ciphertext data. Specifically, the central server stores encrypted symmetric key, asymmetric private key and hash of the password hash. Users authenticate to the key server by sending the combination of their email addresses and password hashes. The key server is also used for sharing files, managing groups as well as synchronizing user information and encrypted keys across devices. Boxcryptor can be used without the key server in a *local mode*. In this mode, user information and encrypted keys are stored in *key files*. However, in this mode, users cannot share files and directories

with other users. They also have to manually transfer the key files to other devices if they want to use Boxcryptor on multiple devices.

7.3.2 Viivo

Viivo [17] is another client utility which provides the same functionality as BoxCryptor. Viivo has a central server for key management. They use standard encryption algorithms and work seamlessly with different cloud service providers. On desktops, Viivo creates "Lockers" which are directories that contain ciphertext files. Lockers can be synchronized automatically with directories containing plaintext. As a result, users can put these lockers to their favourite cloud storage.

There is one notable feature that Viivo supports that is not available to any other mentioned products. Viivo allows password recovery in some cases. Normally, it is impossible to recover users' data without knowing the password since password is not stored on the central server. However, when using the Viivo client application, there is a way to reset the users' password. Firstly, users need to start the reset password process from a desktop device that users have logged in before. In fact, Viivo keeps track of users' devices that use Viivo applications. From there, users can choose "Forgot Password" option in the client application. A box will pop up and users have to enter their email addresses that they used when creating Viivo account. Secondly, Viivo sends an email with a temporary passcode to the users. Finally, users can use the temporary passcode to log into the Viivo client. However, users cannot decrypt the ciphertext files that were encrypted with the previous password any more.

7.3.3 Sookasa

Sookasa [13] is an American based company provides client encryption utilities similar to BoxCryptor and Viivo. The product uses standard encryption techniques as well as supports multiple operating systems. It works only with Dropbox and Google Drive so far.

Sookasa applies a different approach to client-side encryption. They offer encryption service which is compliant with HIPAA [1] standard in healthcare and FERPA [2] standard in education. This allows schools and hospitals in the US to use popular cloud storage services like Dropbox to store their data without worrying about any standards.

[1] `https://www.sookasa.com/resources/hipaa-compliance-checklist/`
[2] `https://www.sookasa.com/resources/ferpa-compliance/`

Sookasa server manages user credentials, policies and key management. Encrypted files and encrypted file keys are stored separately on the cloud storage. However, Sookasa server stores the master key. As a result, files can be decrypted without user permission as long as the intruders have access to both Sookasa server and user's cloud storage.

7.3.4 EncFS

EncFS [4] is a free cryptographic file system. In the EncFS file system, there are two directories: *enc/* which contains ciphertext files and *clear/* containing plaintext files. These two directories are synchronized automatically. When using EncFS with cloud storage services, users can mount enc/ to cloud directories in client machines. All user interactions with clear/ will be reflected to enc/ vice versa. Users can mount EncFS directories using a master password. Anyone who has the password can mount and decrypt files inside the enc/ directory.

Similar to Boxcryptor, Viivo and OmniShare, EncFS synchronize files inside plaintext directory in the local users' devices and ciphertext directory in the cloud storage. Files are encrypted and decrypted separately. Therefore, this approach can also be called *directory-based* approach.

Overall, EncFS is a simple tool offering data encryption based on users' passwords. There is no mechanism for file sharing, deduplication or resetting passwords. Furthermore, EncFS is not up to date with modern cryptographic practices [3].

7.3.5 TrueCrypt

TrueCrypt [16] is another client-side encryption utilities based on users' passwords. However, different from EncFS, TrueCrypt is a *container-based encryption*. In container-based encryption, users first create a fixed container of a fixed size. Users can mount the container into the operating system using user password. The container then appears as a virtual drive in the users' machine. Files inside the container are automatically encrypted. Therefore, users can mount TrueCrypt container to a mount point on the cloud storage directory. Physically, the container appears as a file with the container size. The cloud storage application automatically synchronizes this file across users' devices.

However, with the container-based approach, every change to a file inside the container reflects to change of the whole encrypted container. As a result,

[3] `https://defuse.ca/audits/encfs.htm`

even though TrueCrypt offers better security by hiding the file structure inside the container, it has a serious synchronization problem when using with cloud services. Particularly, the whole TrueCrypt container has to be re-uploaded to the cloud and re-downloaded to other devices every time there are changes in the plaintext files.

7.3.6 PanBox

PanBox [12] is a new client-side encryption tool for cloud storage developed by Sirrix, Germany. PanBox creates a mount point where users can put their sensitive files and directories inside. The mount point is placed in the cloud service directory on client machines as a directory. User files within the mounted volume are automatically encrypted and decrypted. However, PanBox chooses a decentralized approach to implement key distribution so there is no need for a central key server like Boxcryptor.

PanBox keys are generated using a dedicated device key and a password. When users want to share files between two devices, they need to pair those devices. There are three options: Bluetooth, WLAN or files.

- **Bluetooth:** Given a pair of master-slave devices, both devices have to be paired manually first via Bluetooth. After pairing, the master device generates a QR code and the slave device scans it to trigger the key exchange process. The actual key exchange is via Bluetooth.
- **WLAN:** Users need to specify a network interface that they want PanBox to work on. Similar to Bluetooth, the slave device scans a QR code and the key exchange happens if the two devices are on the same network.
- **Files:** Users can export key files from the master and manually transfer to the slave.

PanBox works with Dropbox, Google Drive and all cloud storage services with WebDav interface. The application is available on Windows, Linux, MacOS and Android. Unfortunately, PanBox Android application is only available in Germany. The product manuals are also only available in German. Therefore, we could not analyse the solution in more detail. However, there are a few critical problems that we experienced while testing the solution.

First, the application does not work on Windows 8.1, version 64 bit. PanBox keeps reporting errors on our machine when we try to encrypt files. Panbox depends on a third party library called Dokan [2] which is a user

mode file system library for Windows. Dokan allows the application to create a virtual drive on Windows as well as intercepts all user interactions on the drive. However, the open source Dokan project is no longer maintained since 2011, thus it might not be compatible with new versions of Windows. Even though PanBox provides their own version of Dokan, the program does not work on our test machine.

Secondly, PanBox user experience is not intuitive. In fact, even after several tries, we could not make the application work at all. The product manual is lengthy and not available on English. PanBox user interface is also confusing.

7.4 Summary

Table 7.1 compares client-side encryption products that we have described so far. From end-users' perspective, in order to use cloud storage services such as SpiderOak, Wuala, or Mega, they have to purchase storage capacity and move sensitive data to these provider's *data server*. Therefore, they cannot use mainstream storage options like Dropbox, OneDrive or Google Drive.

On the other hand, Boxcryptor, Viivo or Sookasa provide client-side encryption utilities for popular cloud storage services. They have their own *key servers* for managing and distributing keys across users' devices while using cloud storage services' data servers for storing actual data. Nevertheless, these solutions still rely on third-party key servers.

EncFS and TrueCrypt are also client-side encryption utilities for other cloud storage services. They do not use third-party key servers like Boxcryptor. However, while EncFS is outdated with modern cryptographic practices, TrueCrypt uses container-based approach which causes serious synchronization problem for the cloud service.

One common factor of those described approaches is that they encrypt users' data using keys derived from user passwords. Password based keys can be easily re-generated on any device of the choice by simply typing in the same password. Thus, it is simple and easy to use. Even though the use of password stretching algorithms such as PBKDF2 helps increasing resistance to passive attackers, there are many problems with password based approach. Users might choose very simple passwords, i.e, "12345". High entropy passwords are difficult to remember so they easily forget, write them down or reuse frequently.

Overall, PanBox seems to be the solution that overcomes all those described challenges. It is a free client encryption utility, it does not require third-party server for any purpose and, most importantly, it does not rely on

user password. However, we found that the product is not functioning, its user interface is confusing and is unavailable world-wide.

	Data server	Key server	Encryption	Easy to use	Desktop	Mobile	Web
SpiderOak	✓	✓	password	✓	✓	✓	✓
Wuala	✓	✓	password	✓	✓	✓	✓
Mega	✓	✓	password	✓			✓
Boxcryptor		✓	password	✓	✓	✓	
Viivo		✓	password	✓	✓	✓	
Sookasa		✓	password	✓	✓	✓	
EncFS			password	✓	✓		
TrueCrypt			password	✓	✓		
PanBox			strong key		✓	✓	
OmniShare			strong key	✓	✓	✓	

Table 7.1: Comparison of client-side encryption products

Chapter 8

Conclusions

Cloud storage service is growing rapidly over the past few years. Dropbox was launched in 2008, currently it has more than 400 million users. These users contribute to 1.2 billion files synchronization every day and hundred thousand shared files every hour [34]. Cloud storage brings significant advantages such as the ability for multiple users to collaborate on one document or on-demand access to files from multiple device.

At the same time, data privacy has been a major concern in cloud storage since users have to trust the cloud service providers for security and privacy of their data. We address these problems by developing OmniShare as a generic mechanism to construct authorized domain of devices. OmniShare protects the confidentiality and integrity of users' cloud storage by providing authenticated encryption at the client-side using high-entropy keys. Unlike other similar services, it does not depend on any third-party server for key distribution. OmniShare uses a combination of OOB channel and the cloud storage to minimize user interaction during authorization process. OmniShare also has consistent user experience on all platforms.

On the other hand, OmniShare can be enhanced in future. For example, currently it lacks a mechanism to handle conflicts when a user modify the same files from multiple devices simultaneously. A simple solution would be to let the user choose the version that he wants to keep while discards others. OmniShare currently works only with Dropbox, one of the most popular cloud storage services today. However adding support for multiple cloud storage is straightforward provided that the cloud storage offers interfaces for third-party applications authorized by the users. Finally, although incremental synchronization is a promising approach to enhance OmniShare performance, a more thorough analysis and design of incremental synchronization is required.

OmniShare will have other applications beyond secure cloud storage. For

example, suppose an on-line banking application uses trusted hardware on mobile devices to protect user credentials for on-line banking access. To allow the credentials to be used from multiple devices belonging to the same user, the application could allow the user to define an authorized domain of devices using OmniShare and protect the banking credentials using the domain root key.

Bibliography

[1] Boxcryptor. https://www.boxcryptor.com/en.

[2] Dokan. https://code.google.com/p/dokan/.

[3] Dropbox. https://www.dropbox.com/.

[4] EncFS - an Encrypted Filesystem. https://github.com/vgough/encfs.

[5] Google Drive. https://www.google.com/drive/.

[6] Introduction to Microsoft Sync Framework. https://msdn.microsoft.com/en-us/sync/bb821992.aspx.

[7] Mega. https://mega.co.nz/.

[8] Microsoft OneDrive. https://onedrive.live.com/.

[9] Microsoft Synchronization Framework. https://msdn.microsoft.com/en-us/sync/bb736753.aspx.

[10] open-vcdiff: An encoder/decoder for the VCDIFF (RFC3284) format. https://github.com/google/open-vcdiff.

[11] OwnCloud. https://owncloud.org/.

[12] PanBox. https://www.sit.fraunhofer.de/de/panbox/.

[13] Sookasa. https://www.sookasa.com/.

[14] SpiderOak. https://spideroak.com/.

[15] The Legion of the Bouncy Castle. https://www.bouncycastle.org/.

[16] TrueCrypt. http://truecrypt.sourceforge.net/.

[17] Viivo. https://www.viivo.com/.

[18] Wuala. `https://www.wuala.com/`.

[19] Barker, W. C. *Recommendation for the triple data encryption algorithm (TDEA) block cipher*. US Department of Commerce, Technology Administration, National Institute of Standards and Technology, 2004.

[20] Basin, D., Cremers, C., and Meadows, C. *Model Checking Security Protocols*. Springer, ch. 24.

[21] Bellare, M., and Rogaway, P. Optimal asymmetric encryption. In *Advances in Cryptology - EUROCRYPT'94* (1995), Springer, pp. 92–111.

[22] Bellovin, S. M., and Merritt, M. Encrypted key exchange: Password-based protocols secure against dictionary attacks. In *Research in Security and Privacy, 1992. Proceedings., 1992 IEEE Computer Society Symposium on* (1992), IEEE, pp. 72–84.

[23] Bellovin, S. M., and Merritt, M. Augmented encrypted key exchange: a password-based protocol secure against dictionary attacks and password file compromise. In *Proceedings of the 1st ACM Conference on Computer and Communications Security* (1993), ACM, pp. 244–250.

[24] Borgmann, M., and Waidner, M. *On the security of cloud storage services*. Fraunhofer-Verlag, 2012.

[25] Cremers, C. *Scyther - Semantics and Verification of Security Protocols*. Ph.D. dissertation, Eindhoven University of Technology, 2006.

[26] Cremers, C. The Scyther Tool: Verification, falsification, and analysis of security protocols. In *Computer Aided Verification, 20th International Conference, CAV 2008, Princeton, USA, Proc.* (2008), vol. 5123/2008 of *Lecture Notes in Computer Science*, Springer, pp. 414–418.

[27] Cremers, C., and Horvat, M. Improving the iso/iec 11770 standard for key management techniques. In *Security Standardisation Research*. Springer, 2014, pp. 215–235.

[28] Daboo, C. Extended mkcol for web distributed authoring and versioning (webdav). RFC 5689, RFC Editor, September 2009.

[29] Daemen, J., and Rijmen, V. Aes proposal: Rijndael.

[30] Diffie, W., and Hellman, M. E. New directions in cryptography. *Information Theory, IEEE Transactions on 22*, 6 (1976), 644–654.

[31] Diffie, W., Van Oorschot, P. C., and Wiener, M. J. Authentication and authenticated key exchanges. *Designs, Codes and cryptography 2*, 2 (1992), 107–125.

[32] Dolev, D., and Yao, A. C. On the security of public key protocols. *Information Theory, IEEE Transactions on 29*, 2 (1983), 198–208.

[33] Douceur, J. R., Adya, A., Bolosky, W. J., Simon, P., and Theimer, M. Reclaiming space from duplicate files in a serverless distributed file system. In *Distributed Computing Systems, 2002. Proceedings. 22nd International Conference on* (2002), IEEE, pp. 617–624.

[34] Dropbox. 400 million strong. `https://blogs.dropbox.com/dropbox/2015/06/400-million-users/`.

[35] Dropbox. Dropbox Sync SDK for Android. `https://www.dropbox.com/developers/sync`.

[36] Fairless, A. Why SpiderOak doesn't deduplicate data across users. `https://spideroak.com/about/perspectives`.

[37] Gehrmann, C., Mitchell, C. J., and Nyberg, K. Manual authentication for wireless devices. *RSA Cryptobytes 7*, 1 (2004), 29–37.

[38] Goldreich, O. *Foundations of Cryptography: Volume 1*. Cambridge University Press, New York, NY, USA, 2006.

[39] Google. Android Dashboards. `https://developer.android.com/about/dashboards/index.html?utm_source=suzunone#Platform`.

[40] Google. Android keystore system. `https://developer.android.com/training/articles/keystore.html`.

[41] Google. gson. `https://code.google.com/p/google-gson/`.

[42] Greenrobot. EventBus. `https://github.com/greenrobot/EventBus`.

[43] Hitachi Data Systems. 4 principles for reducing total cost of ownership. `http://www.hds.com/assets/pdf/four-principles-for-reducing-total-cost-of-ownership.pdf`.

[44] ISO. Information technology – security techniques – authenticated encryption. ISO ISO/IEC 19772:2009, International Organization for Standardization, Geneva, Switzerland, 2009.

[45] Jonsson, J., and Kaliski, B. Public-key cryptography standards (pkcs) 1: Rsa cryptography specifications version 2.1. RFC 3447, RFC Editor, February 2003. `http://www.rfc-editor.org/rfc/rfc3447.txt`.

[46] Kainda, R., Flechais, I., and Roscoe, A. Usability and security of out-of-band channels in secure device pairing protocols. In *Proceedings of the 5th Symposium on Usable Privacy and Security* (2009), ACM, p. 11.

[47] Kaliski, B. Pkcs 7: Cryptographic message syntax version 1.5. RFC 2315, RFC Editor, March 1998. `http://www.rfc-editor.org/rfc/rfc2315.txt`.

[48] Kaliski, B. Pkcs 5: Password-based cryptography specification version 2.0. RFC 2898, RFC Editor, September 2000. `http://www.rfc-editor.org/rfc/rfc2898.txt`.

[49] Korn, D., MacDonald, J., Mogul, J., and Vo, K. The vcdiff generic differencing and compression data format. RFC 3284, RFC Editor, June 2002.

[50] Laur, S., and Nyberg, K. Efficient mutual data authentication using manually authenticated strings. In *Cryptology and Network Security*. Springer, 2006, pp. 90–107.

[51] Leach, P. J., Mealling, M., and Salz, R. A universally unique identifier (uuid) urn namespace. RFC 4122, RFC Editor, July 2005. `http://www.rfc-editor.org/rfc/rfc4122.txt`.

[52] Lilly, G. M. Device for and method of one-way cryptographic hashing, Dec. 7 2004. US Patent 6,829,355.

[53] Lowe, G. A hierarchy of authentication specifications. In *Computer Security Foundations Workshop, 1997. Proceedings., 10th* (1997), IEEE, pp. 31–43.

[54] Martin, R. C. *Agile software development: principles, patterns, and practices*. Prentice Hall PTR, 2003.

[55] Menezes, A. J., Van Oorschot, P. C., and Vanstone, S. A. *Handbook of applied cryptography*. CRC press, 1996.

[56] Meyer, D. T., and Bolosky, W. J. A study of practical deduplication. *ACM Transactions on Storage (TOS) 7*, 4 (2012), 14.

[57] Microsoft. Cryptography API: Next Generation. `https://msdn.microsoft.com/en-us/library/windows/desktop/aa376210(v=vs.85).aspx`.

[58] Rivest, R. L., Shamir, A., and Adleman, L. A method for obtaining digital signatures and public-key cryptosystems. *Commun. ACM 26*, 1 (Jan. 1983), 96–99.

[59] Rtyley. Spongy Castle - repackage of Bouncy Castle for Android. `https://rtyley.github.io/spongycastle/`.

[60] Suomalainen, J., Valkonen, J., and Asokan, N. Standards for security associations in personal networks: a comparative analysis. *International Journal of Security and Networks 4*, 1 (2009), 87–100.

[61] Taylor, D., Wu, T., Mavrogiannopoulos, N., and Perrin, T. Using the secure remote password (srp) protocol for tls authentication. RFC 5054, RFC Editor, November 2007.

[62] Uzun, E., Karvonen, K., and Asokan, N. Usability analysis of secure pairing methods. In *Financial Cryptography and Data Security*. Springer, 2007, pp. 307–324.

[63] Wilcox-O'Hearn, Z. Drew Perttula and attacks on Convergent Encryption. `https://tahoe-lafs.org/hacktahoelafs/drew_perttula.html`.

[64] Wu, T., et al. Srp-6: Improvements and refinements to the secure remote password protocol. *Submission to IEEE P 1363* (2002).

[65] Wu, T. D., et al. The secure remote password protocol. In *NDSS* (1998), vol. 98, pp. 97–111.

[66] zxing. Zebra Crossing. `https://github.com/zxing/zxing`.

Appendix A

Scyther role script

A.1 Role script for key distribution using QR code

```
1  hashfunction H;
2  /*
3   * Scyther assumes that all agents have access to
       all built-in public keys.
4   * We need to define custom keys for modeling key
       distribution
5   * as well as a temporary nonce variable for use
       with the keys.
6   */
7  const pk2: Function; //A public key, this is
     different from Pk(A)
8  secret sk2: Function;
9  inversekeys (pk2,sk2);
10
11 protocol QRP(A,B)
12 {
13   role A{
14     fresh sessionKey: Nonce; // random session key
15     var rootKey: Nonce;
16     var MAC: Ticket;
17     fresh A1: Nonce; // unused nonce to correct
         asymmetric behaviour
18     send_1(A,B, pk2(A1));
19
```

```
20      //simulation of OOB channel communcation using
            Scyther default symmetric key
21      send_2(A,B,{sessionKey, H(pk2(A1))}k(A,B));
22
23      macro m = {rootKey}pk(A);
24      recv_3(B,A, (m, MAC));
25
26      //Do a MAC integrity check
27      match(MAC, H(m, sessionKey));
28
29      claim_a1(A,Secret,sk2(A1));
30      claim_a5(A,Secret,sessionKey);
31      claim_a6(A,Secret,rootKey);
32      claim_a7(A, Alive);
33      claim_a8(A, Nisynch);
34      claim_a9(A, Niagree);
35      claim_a10(A, Reachable);
36    }
37
38    role B{
39      var hash: Ticket;
40      var sessionKey: Nonce;
41      fresh rootKey: Nonce;
42      var A1: Nonce; // unused nonce to correct
            asymmetric behaviour
43
44      recv_1(A,B,pk2(A1));
45
46      //receive OOB channel message
47      recv_2(A,B,{sessionKey, hash}k(A,B));
48
49      //OOB message verfication
50      match(hash, H(pk2(A1)));
51
52      //Send back the encrypted rootkey and a HMAC
            for integrity protection.
53      macro m = {rootKey}pk(A);
54      send_3(B,A, (m, H(m, sessionKey)));
55
56      claim_b1(B,Secret,sessionKey);
57      claim_b2(B,Secret,rootKey);
```

```
58      claim_b3(B,Alive);
59      claim_b4(B,Nisynch);
60      claim_b5(B, Niagree );
61      claim_b6(B, Reachable);
62    }
63  }
```

A.2 Role script for key distribution using SRP

```
1  hashfunction g1, g2, H;
2  function f, plus;
3
4  /*
5   * Support protocol for approximating equality.
6   * In this case, is DH exponentiation and the
       construct of session key
7   */
8  protocol @exponentiation(BE, BM1, AM2)
9  {
10   role BE{
11         //Simulate (g^a)^b = (g^b)^a
12     var a,b: Ticket;
13     recv_!1(BE,BE,g2(g1(a),b));
14     send_!2(BE,BE,g2(g1(b),a));
15   }
16
17   role BM1{
18     //Simulate M1 equality
19     var alpha, beta, a,b,x: Ticket;
20     recv_!3(BM1, BM1, H(alpha,beta, H(f(g2(g1(b),a
         ), g2(g1(b),x)))));
21     send_!4(BM1, BM1, H(alpha,beta, H(f(g2(g1(a),b
         ), g2(g1(x),b)))));
22   }
23
24   role AM2
25   {
26     //Simulate M2 equality
27     var alpha, beta, a, b, x, RK: Ticket;
```

```
28      macro keyA = f(g2(g1(b),a), g2(g1(b),x));
29      macro keyB = f(g2(g1(a),b), g2(g1(x),b));
30      macro M1A = H(alpha, beta, H(keyA));
31      macro M1B = H(alpha, beta, H(keyB));
32      macro M2A = H(alpha, M1A, keyA);
33      macro M2B = H(alpha, M1B, keyB);
34      recv_!5(AM2,AM2,{RK}keyB, M2B);
35      send_!6(AM2,AM2,{RK}keyA, M2A);
36    }
37  }
38
39  protocol SRP(A,B)
40  {
41    role A{
42      fresh s, P, a: Nonce;
43      var beta, v, RK: Ticket;
44      macro x = H(s,P);
45      macro key = f(g2(beta,a), g2(beta,x));
46      macro M1 = H(g1(a), plus(beta, v), H(key));
47      macro M2 = H(g1(a), M1, key);
48
49      send_1(A,B, g1(a), s);
50      send_2(A,B,{P}k(A,B));
51      recv_3(B,A,plus(beta,v));
52
53      match(v, g1(x));
54      send_!4(A,B, M1); //Sending out M1
55      recv_!5(B,A, {RK}key,M2); //Receiving M2, not
          from role B but from the helper protocol
56      claim_a1(A, Reachable);
57      claim_a2(A, Secret, a);
58      claim_a3(A, Secret, P);
59      claim_a4(A, Secret, key);
60      claim_a5(A, Secret, RK);
61      claim_a6(A, Nisynch);
62      claim_a7(A, Niagree);
63    }
64
65    role B{
66      fresh b, RK: Nonce;
67      var alpha, s, P;
```

```
68      macro x = H(s,P);
69      macro key = f(g2(alpha,b), g2(g1(x),b));
70      macro M1 = H(alpha, plus(g1(b), g1(x)), H(key)
           );
71      macro M2 = H(alpha, M1, key);
72
73      recv_1(A,B, alpha, s);
74      recv_2(A,B,{P}k(A,B));
75      send_3(B,A, plus(g1(b),g1(x)));
76      recv_!4(A,B, M1); //Receiving M1, not from
           role A but from the helper protocol
77      send_!5(B,A, {RK}key, M2); //Sending out M2
78
79      claim_b1(B, Reachable);
80      claim_b2(B, Secret, b);
81      claim_b3(B, Secret, P);
82      claim_b4(B, Secret, key);
83      claim_b5(B, Secret, RK);
84      claim_b6(B, Nisynch);
85      claim_b7(B, Niagree);
86
87    }
88  }
```