Distributed Personal Password Repository using Secret Sharing

Merete Løland Elle, Stig F. Mjølsnes*, and Ruxandra F. Olimid

Department of Information Security and Communication Technology, Norwegian University of Science and Technology - NTNU, Trondheim, Norway
{sfm,ruxandra.olimid}@ntnu.no

Abstract

Secret sharing based systems can provide both data secrecy and recoverability simultaneously. This is achieved by a special cryptographical splitting of the data, where the parts, called *shares*, are distributed among a group of entities. A classical solution would be to first encrypt the data (*confidentiality*), then to copy and store the result for backup (*recoverability*). However, by using a secret sharing system, the complete data can be recovered even when only a sufficiently sized subset of shares can be supplied, while any smaller subset of shares does not leak any information about the original data (*perfect secrecy*). For instance, the shares can be distributed across several distinct cloud providers, thus enabling a secure and recoverable storage. Following this idea, we design and propose a novel application for secure and recoverable management of personal passwords by distributing secret shares to cloud storage entities. We have made an experimental smartphone implementation that validates the expediency of the design. The Android application implementation distributes the shares to three cloud providers (Dropbox, Google Drive and Microsoft OneDrive). We note that several mobile password managers exist, but they mostly use the classical solution of encrypted data for storage.

1 Introduction

1.1 Motivation

Authentication is the most popular method to allow access to resources and services. Despite the tremendous effort to replace the usage of passwords by introducing other mechanisms (e.g.: electronic tokens, one-time keys, biometrics), passwords remain the most common, but also the most vulnerable factor in a personal authentication process. This might be caused by the implementation of the password authentication itself, but very often is caused by bad practices at the user's side. A common method to alleviate people's problem of remembering many passwords is to use the same password for multiple platforms. To prevent people from using this and other bad practices, the very practical question of how to securely manage all our private passwords comes naturally. A solution can be a *password manager*, which should provide at least *confidentiality* of the stored passwords and *recoverability* in case of incidents and faults.

1.2 Contribution

The contributions presented in this paper are based on the Android application design and implementation described in Elle's master's thesis work [5], where we investigated the usability

*The author presented this paper at the NISK 2018 conference.

Proceedings of the 11th Norwegian Information Security Conference, Sep. 19-20, 2018.
NISK Journal, e-ISSN 1894-7735. https://ojs.bibsys.no/index.php/NISK

Figure 1: A password is divided in three shares, where each share is stored in a cloud; Reconstruction is possible from any two out of three shares

of secret sharing for password storage, as an alternative to the classical solution that requires both encryption (for data *confidentiality*) and backup (for data *recoverability*). Unlike this classical backup method that creates copies, secret sharing does not increase the risk of exposure, but facilitates secrecy by dividing trust between multiple entities. This approach is thus a possible solution for secure distributed cloud storage, where the clouds are located with distinct providers.

Even though the concept of secret sharing has been around for almost 40 years now and has been considered for long-term secure storage, it has been given very little attention for password storage. By using the concept of storing by secret sharing, and implementing it in a mobile application, one could create an application that acts like the conventional password storage manager while providing a solution that is information-theoretically secure. Considering secret sharing as a basic for a distributed personal password repository and implementing a working Android experimental application for sharing passwords among cloud providers brings novelty to this work. This particular application distributes secret shares among three clouds (Dropbox, Google Drive and Microsoft OneDrive) in a theoretically-secure manner. Using Shamir's threshold secret scheme, we obtain recoverability from any two out of three shares. See Figure 1 for a schematic representation of password sharing and reconstruction.

In our literature survey, we found only one other mobile application that uses secret sharing for password management [21]. That Android application is apparently under development, but with no software update since 2016. Moreover, its design does not distribute the shares to third party cloud servers, but to other personal devices, which makes that application's functionality dependent on the availability of the personal devices at any given time.

1.3 Outline

The rest of the paper is organized as follows. Next section introduces the theoretical background in terms of secret sharing and the two different approaches for secure storage. Section 3 introduces the implementation background in terms of clouds usage and connectivity, as well as some basic Android implementation notions. Section 4 presents the experimental Android application that implements the distributed password repository among clouds. Last section concludes.

2 Theoretical Background

2.1 Secret Sharing

Secret sharing is a method to divide a secret into parts, called *shares* that are distributed among a group of entities such that the secret can only be reconstructed when an *authorized* set of participants combine their shares. Individual shares or combinations of shares belonging to *unauthorized* sets of participants do not reveal the secret. A large variety of secret sharing schemes exist in the literature and are classified based on the secrecy level they guarantee for unauthorized sets, the structure of the authorized sets, the type of the algorithms used for generating the shares or reconstructing the secret, and others.

We will only describe Shamir's scheme [16] here because it satisfies the necessities for our work, and it is further used in the implementation. It is a *threshold* secret sharing scheme, which means that the secret can be reconstructed from any set of at least k shares, where k is a given threshold. This implies redundancy directly: if at least k shares remain accessible, the secret can be reconstructed even if the other shares are lost or destroyed. Moreover, Shamir's scheme is *perfectly secure*, which means that, theoretically, less than k shares reveal absolutely no information about the secret. Shamir's scheme is based on the fact that a $(k-1)$-degree polynomial $f(x)$ is uniquely defined by (at least) k distinct pairs $(x, f(x))$. In the sharing phase, the free coefficient is set to the value of the secret, and the rest of the coefficients of the polynomial are randomly chosen modulo q, where q is a large prime number. Notice that the secret itself must lie in $\mathbb{Z}_q = \{0, \ldots, q-1\}$, because it is also acts like a coefficient. This might seem a limitation, but is not the case, because any sequence of bits can be encoded into one or more values in $\mathbb{Z}_q$ that can be shared independently. The i-th share is simply set to $f(i)$, the value of the polynomial in i. Reconstruction is possible from any k (or more) shares by polynomial interpolation.

2.2 Secure Storage Types

The classical solution for secure storage assumes strong encryption, and so decryption becomes *computationally* infeasible in the absence of the decryption key. This means that the data remains confidential to any adversary with bounded computational power that gains access to the ciphertext. Contrastingly, a solution that uses a *perfect* secret sharing scheme (such as Shamir's) maintains confidentiality against any adversary, even with unbounded computational power, under the assumption that the adversary can gain access to up to $k-1$ shares.

The classical solution uses backup or other replication mechanisms to assure data redundancy in case of incidents and faults, while threshold secret sharing based solutions provide data recoverability by construction: the data can be fully reconstructed as long as a minimal number of shares (equal to the threshold) are accessible.

For a more detailed comparison of the two approaches, see [17, 18, 15]. Although many solutions for long-term data storage that use secret sharing mechanisms have been proposed in the literature, surprisingly few investigations have been done to employ secret sharing mechanisms for passwords management.

3 Implementation Background

3.1 Clouds

Three of the major cloud storage providers on the market today are Dropbox, Google Drive and Microsoft OneDrive [1]. They provide from 2 to 15 GB of free storage. Using an API (Application Programming Interface), an application can interconnect to a cloud by sending and receiving data in a secure manner, using encryption for the data in transit [20]. All the three mentioned clouds use the standard OAuth 2.0 authentication scheme, a framework that enables applications to obtain limited access to resources, without the users having to share their log-in credentials with the application [11].

In our implementation, the clouds are used to store the shares. The amount of necessarily storage space occupied by the shares is very small compared to the overall free storage space of a user, so there is no negative impact with respect to storage capacity. Storing the shares to the cloud does not introduce any additional costs and in principle, gives the user high flexibility in selecting its own preferable providers. Although the experimental work of the current application allows connectivity to the above-mentioned clouds only, in principle the application can be extended to allow connectivity to other cloud providers as well.

The code used for connecting to the Dropbox API is provided by the Dropbox Core SDK (Software Development Kit) for Java 6+ [3]. Similarly, the code used for connecting to the OneDrive API for Android is provided by the sample code from the OneDrive SDK [13], while the code used for connecting to the Google Drive API for Android is provided by Google [2].

3.2 Android

Android is an open-source mobile OS (Operating System) owned by Google [10]. The popularity of Android devices (approximately 69% of the global market in April 2018 [14]) and the ease to program makes it a good choice for developing mobile applications.

The Android Studio is the official Integrated Development Environment (IDE), providing tools for building applications for all Android devices [7].

The security mechanisms for Android includes code signing, application isolation, permission model, file system encryption, and other security protections. The purpose of the code signing is to prove the source of the software, by signing the application with the developer's private key, and only allowing updates to be made if the signature can be verified. Furthermore, each Android application resides in its own security sandbox, which isolates the application data and code execution from other applications [8]. If the application needs to use resources outside of the sandbox, the application has to request permission. The permissions are declared by listing them in the *manifest file* [9].

Note that this work did not go through all steps required for releasing an application because our Android experimental application was considered for testing purposes in a limited environment. The application was made available for download outside the Play Store. The manifest file includes access to the Internet, the network state, reading and writing to the external storage, and managing access to documents. It also contains additional metadata, such as the API key for the Google API and the authentication activity for Dropbox with the application key.

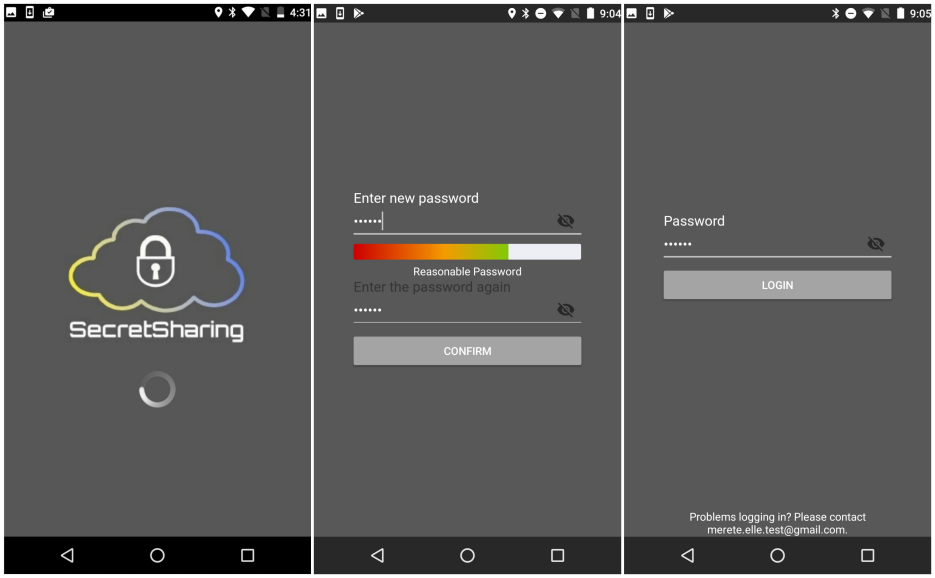

Figure 2: Screenshot of the application; left: welcome screen; middle: creating a new application password; right: log-in screen

4 The Application

The distributed password repository is implemented in the shape of an Android application that uses Shamir's secret sharing scheme with threshold $k = 2$. The application divides each password into three shares and distributes each to three different cloud providers (Dropbox, Google Drive and Microsoft OneDrive). Android Studio has been used in the developing process, and the connectivity to the clouds was performed as previously explained.

4.1 Functionality

We describe next the basic functionality of the experimental implementation in terms of user actions and results. We keep in mind that our fundamental goal is not to describe the implementation itself, but to consider secret sharing as a basis for a distributed personal password repository, so we also mention capabilities that have not been implemented in the application, improvements and future work.

Login. After the welcome screen (or *splash screen*), the user is either asked to introduce a password (if no application password has been set before), or to log-in by typing the correct password. When a new password is introduced, a progress bar gives an indication of its strength. The evaluation is based on the length of the password, as well as the number of symbols, digits, uppercase and lowercase letters. Afterwards, the login screen is prompted every time the application is launched, to create an extra barrier for a potential attacker. This would disallow for example direct access to the application to someone that has physical access to the phone. Figure 2 illustrates related screenshots of the application.

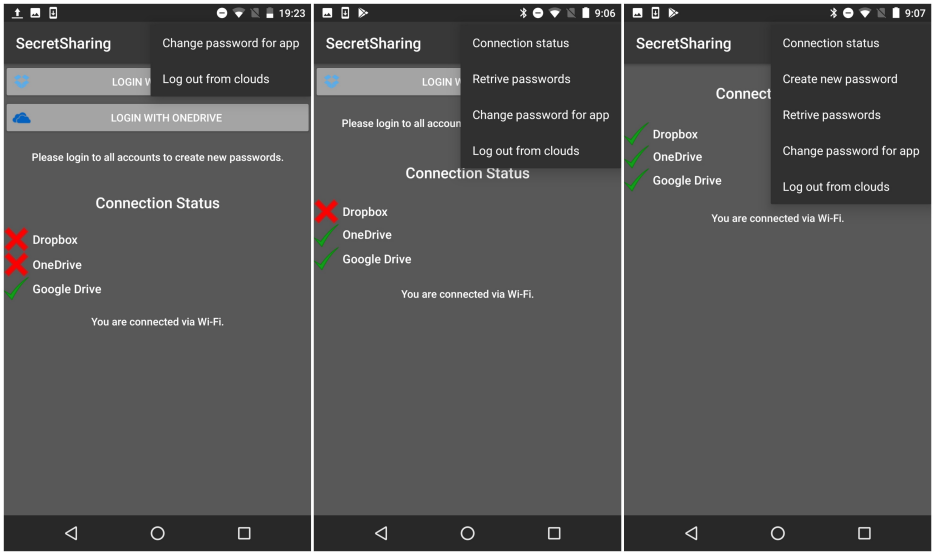

Figure 3: Screenshot of the application; left: drop-down menu when one cloud is connected, middle: drop-down menu when two clouds are connected, right: drop-down menu when all clouds are connected

Connectivity to the clouds. The entry point of the application is the connection status screen, displayed after the user is successfully logged in. Is here where the user has an overview of whether it is connected or disconnected to the clouds, and also if there is Internet connectivity. For each cloud there is displayed a login button. The login buttons contain the names and icons for the clouds, in accordance with the guidelines for branding [4, 12, 6]. If the user is not logged to any cloud, then all of the login buttons are visible and all the clouds are marked with a red cross. When the user logs to any of them (using his personal credentials), the red cross changes to a green tick symbol, indicating that the application is connected to the cloud. If so, the login button becomes invisible. Figure 3 illustrates related screenshots of the application.

The elements in the drop-down menu change depending on how many clouds are connected. If none, the user only has access to change the password for the application. If at least one cloud is connected, the user can in addition log out from the clouds. If at least two clouds are connected, the user can reconstruct passwords. Only when all clouds are connected, the user has full functionality and can store a new password. The restriction of storing a password only when all clouds are up is to achieve redundancy: a password can be recovered from any two shares, but the splitting is done in three shares. Of course, this is a particular implementation, which is somehow hardcoded. To properly allow flexibility, the user might choose the overall number of shares and the threshold needed for reconstruction. But the implementation should be done with care, do disallow the user to poorly parametrize (e.g.: to get no redundancy).

Store a password. To store a new password, the user needs to press the *Create new password* button in the drop-down menu. This action redirects to a new screen, as shown in Figure 4. Here, the *platform name* field corresponds to the system the password is used for, e.g. "Facebook", and the *password* field is the password to be stored. When the save button is pressed, a pop-up is prompted to confirm the action. The application will split the password into shares

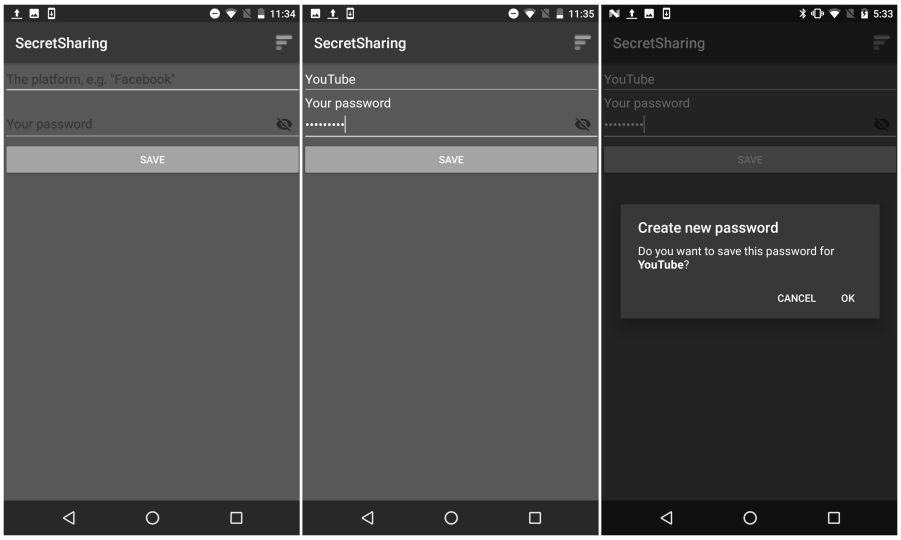

Figure 4: Screenshot of the application; left: create new password, middle: filled in platform name and password, right: pop-up for confirmation

and distribute to each cloud a share titled with the platform name, which can be later used to reconstruct the password. Again, an implementation decision has been made here not to hide the name of the platform, assuming that only the password protection is important. Of course, to some extend the platforms themselves can be considered private, as they disclose something about the user behavior, so the name of the shares should be independent of the platform name. This can easily be achieved by keeping a file that matches the names of the platforms to random or generic sequences. These sequences are then used for naming the shares in the clouds, and the matching file can be stored split between the clouds by secret sharing. Each reconstruction would then additionally imply first the reconstruction of the matching file, and only after the reconstruction of the password by the corresponding shares. In the absence of this matching file, a similar file containing just the names of the platforms can be stored in the clouds in case of the phone is stolen, lost or dysfunctional and the user needs to reinstall the application on another device and restore the passwords. This is what we call the *backup file*.

Reconstruct a password. To reconstruct a password, the user needs to press the *Retrieve passwords* button in the drop-down menu. This action redirects to a new screen that contains a list of the stored passwords, as shown in Figure 5. The user selects the platform's name that corresponds to the password he wants to recover and confirms in the pop-up window. This triggers the reconstruction of the password from shares, using the secret sharing scheme.

Deleting a password. To delete a password, the user needs to click on the platform name and hold it for a short while before a pop-up with a warning message is displayed. After the user confirms deletion, the application triggers the deletion of all the shares belonging to the given password in the clouds.

Figure 6 is an overall representation of the activities and the options shown to the user. On the Android platform, an activity serves as the entry point for an applications interaction with

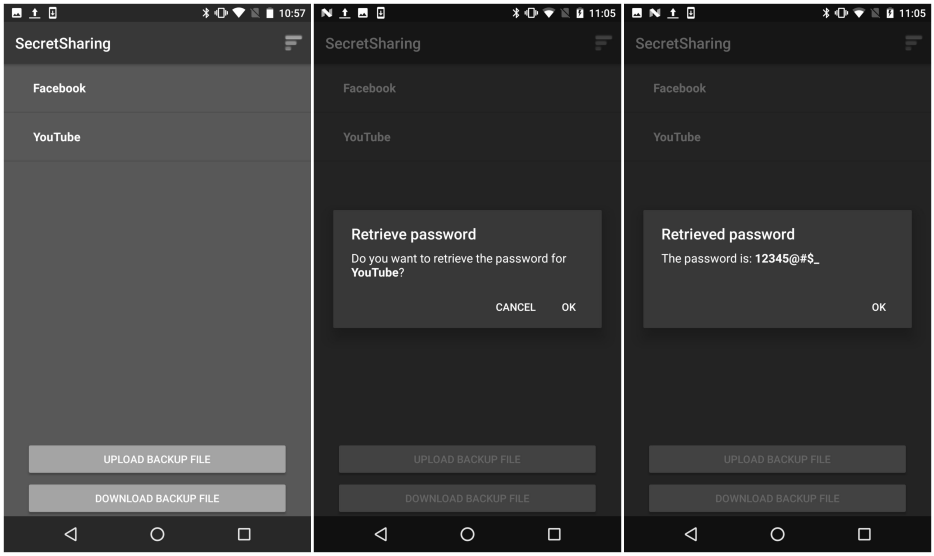

Figure 5: Screenshot of the application; left: retrieve passwords, middle: pop-up asking if the user wants to retrieve the chosen password, left: the password retrieved

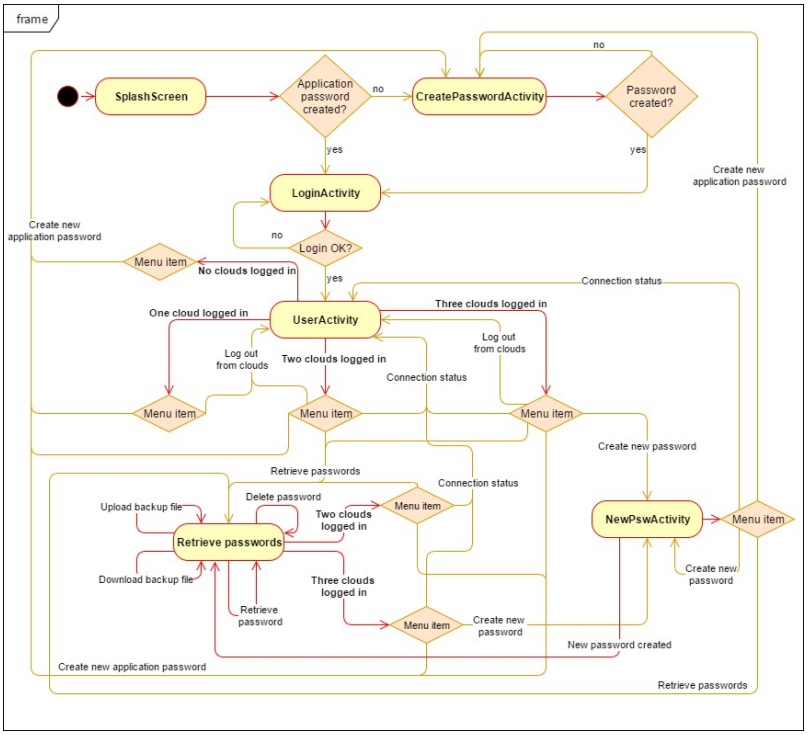

Figure 6: An overall representation of the activities and menu options shown to the user

the user. The implementation contains six activities visible to the user, which are illustrated in the figure.

Completing the design. One could immediately observe the absence of the usernames here. Even if usually the username is easier to remember (e.g.: the e-mail address), the user should not be asked to remember them, but they must be managed by the password manager application too. This could be simply solved by concatenating the username to the password (for instance delimited by a space) before sharing. This is if the application prototype implementation is to be followed. A simpler approach would be to store all platforms' names, together with their associated usernames and passwords, in a single file that is to be shared between the clouds. This would imply the storage of a single share (of larger size) to each cloud. Each time a new password needs to be stored, the whole file is reconstructed, the new string *(platform-name, username, password)* is appended to the file and the new version is secret shared, updating the new shares to the clouds. A similar procedure should be followed in case of deletion of passwords. Of course, this solution would require a higher computational cost for a single password storage or deletion, but it should not introduce a significant overload, so it should not affect the usability of the application. It would also eliminate the need of a backup or matching file, and it would keep private the platforms' names by design. For any of the solutions, a versioning system should be put in place, and timestamp of all performed actions must be available.

4.2 Sharing and Reconstruction

An available java implementation was used as a base for the secret sharing algorithm [19].

A 384-bit prime q is used for the split and reconstruction operations, and the password is encoded to an integer in $\mathbb{Z}_q$. As already mentioned, this is not a limitation because the password string can be broken into smaller pieces that lie in $\mathbb{Z}_q$. Anyway, this should not be the case, because no usable password can exceed 48 characters in length.

Share a password. To split a password into shares, a method called *splitSecretIntoPieces* is called with parameters the password *secret*, the total number of secrets n and the number of shares required for reconstruction k. The return value is a string array containing the shares, that will later be distributed to the clouds. Listing 1 shows the splitting of the password into shares.

Listing 1: Splitting password into shares, code from *UploadSharesTask.java* file

```
// Split password into shares.
    final String secret;
    secret = [THE_PASSWORD];
    final int n = 3, k = 2;
    String[] pieces = splitSecretIntoPieces(secret,n,k);
```

Reconstruct a password. To reconstruct the password, a method *mergePiecesIntoSecret* is called with parameter a string array obtained from the shares used for reconstruction. As an implementation example, Listing 2 shows the scenario where the password is reconstructed when all three shares are used.

Figure 7: The overall score of the user tests: "PASSED" 98,2% and "FAILED" 1,8%

Listing 2: Retrieving password, code from *RetrieveActivity.java* file

```
String[] pieces = new String[3];
pieces[0] = [OneDrive_SHARE];
pieces[1] = [Dropbox_SHARE];
pieces[2] = [Google Drive_SHARE];

// Create arraylist out of the formatted strings,
// shuffle and reconstruct secret.
List<String> list = new ArrayList<String>(Arrays.asList(pieces));
Collections.shuffle(list);
String[] kPieces = list.toArray(new String[0]);
final String reconstructed = mergePiecesIntoSecret(kPieces);
```

4.3 Testing and Feedback

An evaluation was conducted to measure how well the application performed while being used by regular users. The users were able, and encouraged, to comment on the application's interface and behavior. The aim was to have a focus group of five subjects, preferably with various degrees of technical skills.

The subjects were given a test sheet containing 70 steps that covered all aspects of the functionality, including for example logging in, storing and recovering passwords. The users gave each functionality a score of "PASSED" or "FAILED", where "PASSED" means that the application responded as described in the test sheet, and "FAILED" means that there were some deviations from the description in the test sheet. By evaluating these tests, a quantitative score, the *overall score*, was used to describe how well the application performs. Figure 7 shows the overall score of the user test, with a "PASSED" score of 98,2%. The comments provided by the users served as inputs for further development and improvement of the application.

The sharing and reconstruction of the secret is performed fast and hence introduces no disturbing delay for the user. Therefore, from a performance point of view there was no negative feedback with respect to the response time of the application.

5 Conclusions

The paper presents a distributed personal password repository prototype that uses secret sharing implemented in the form of an Android experimental application. The usage of secret sharing as an alternative solution for password storage brings novelty. By using Shamir's threshold secret sharing, the application provides redundancy by default. In the presented implementation, two out of three shares are required for reconstruction, but a more flexible approach might allow the user to parametrize the overall number of shares, as well as the shares required for reconstruction. Also, flexibility should be added to allow connectivity to other clouds. In a complete version, usernames must be stored for each platform together with the passwords, and a versioning system could be set into place.

As a risk, there is a certain chance that the shares can be deleted outside the application, directly from the cloud. If this is the case, and the remaining shares are below the reconstruction threshold, then reconstruction is not possible and an error message is displayed.

A testing of the functionalities of the implementation was performed. The goal was not to test the security of the application, but mostly its functionalities and usability. Therefore, future work includes an in-depth security evaluation of the implementation. Furthermore, enhancements on the privacy should be considered. For example, this includes the secrecy of the names of the platforms the user has accounts for, which to some extent represent private information too.

References

[1] Martyn Casserly. The best cloud storage services. http://www.pcadvisor.co.uk/test-centre/internet/best-cloud-storage-services-2017-uk-3614269/. Accessed: April 2018.

[2] Dropbox. Authorizing Android Apps. https://developers.google.com/drive/android/auth, 2017. Accessed: April 2018.

[3] Dropbox. Dropbox Core SDK for Java 6+. https://github.com/dropbox/dropbox-sdk-java, 2017. Accessed: April 2018.

[4] Dropbox, Inc. Developer branding guide. https://www.dropbox.com/developers/reference/branding-guide. Accessed: April 2018.

[5] Merete Løland Elle. Long-term confidential data storage by distributed secret shares. Master's thesis, NTNU, Department of Information Security and Communication Technology, 7 2017. https://brage.bibsys.no/xmlui/handle/11250/2458158.

[6] Google. Use the Drive Badge and Brand. https://developers.google.com/drive/v3/web/branding. Accessed: April 2018.

[7] Google. Android Studio - The Official IDE for Android. https://developer.android.com/studio/index.html, 2017. Accessed: April 2018.

[8] Google. Application Fundamentals. https://developer.android.com/guide/components/fundamentals.html, 2017. Accessed: April 2018.

[9] Google. Declaring Permissions. https://developer.android.com/training/permissions/declaring.html, 2017. Accessed: April 2018.

[10] Google. Understanding Android. https://www.android.com/everyone/facts/, 2017. Accessed: April 2018.

[11] Internet Engineering Task Force (IETF). [RFC6749] The OAuth 2.0 Authorization Framework. https://tools.ietf.org/html/rfc6749, 2012. Accessed: April 2018.

[12] Microsoft. Branding guidelines. https://msdn.microsoft.com/en-us/onedrive/dn673556.aspx. Accessed: April 2018.

[13] Microsoft. OneDrive SDK for Android. https://github.com/OneDrive/onedrive-sdk-android, 2017. Accessed: April 2018.

[14] Net MarketShare. Operating System Market Share. https://www.netmarketshare.com/operating-system-marketshare.aspx?qprid=8&qpcustomd=1., 2018. Accessed: April 2018.

[15] Ruxandra F. Olimid and Dragos Alin Rotaru. On the security of a backup technique for database systems based on threshold sharing. *Journal of Control Engineering and Applied Informatics*, 18(2):37–47, 2016.

[16] Adi Shamir. How to share a secret. *Commun. ACM*, 22(11):612–613, 1979.

[17] Mark W. Storer, Kevin M. Greenan, Ethan L. Miller, and Kaladhar Voruganti. Potshards - a secure, recoverable, long-term archival storage system. *ACM Transactions on Storage (TOS)*, 5(2):5, 2009.

[18] Arun Subbiah and Douglas M. Blough. An approach for fault tolerant and secure data storage in collaborative work environments. In *Proceedings of the 2005 ACM workshop on Storage security and survivability*, pages 84–93. ACM, 2005.

[19] Tim Tiemens. Shamir's Secret Share in Java. https://github.com/timtiemens/secretshare, 2014. Accessed: April 2018.

[20] Davey Winder. How secure are Dropbox, Microsoft OneDrive, Google Drive and Apple iCloud cloud storage services? http://www.alphr.com/apple/1000326/how-secure-are-dropbox-microsoft-onedrive-google-\drive-and-apple-icloud-cloud-storage. Accessed: April 2018.

[21] ETH Zurich. Convenient Password Manager. https://play.google.com/store/apps/details?id=disco.ethz.ch.convenientpasswordmanager&hl=no, 2016. Accessed: April 2018.