Convenient Password Manager

Group Project

Manuel Eggimann, Christelle Gloor
meggimann@student.ethz.ch cgloor@student.ethz.ch

Distributed Computing Group
Computer Engineering and Networks Laboratory
ETH Zürich

Supervisors:
Pascal Bissig, Philipp Brandes
Prof. Dr. Roger Wattenhofer

January 10, 2016

Abstract

This paper describes an application approaching password management in a different manner than what is popularly used by a lot of people at the moment. The application works with multiple devices communicating via Bluetooth to increase security of password management. It uses Shamir's secret share algorithm instead of locking passwords behind more passwords while still keeping the convenience of a password safe. The implementation reached a bottleneck concerning the possibilities to establish more than two stable peer-to-peer connections via Bluetooth.

Contents

1 Introduction

The management of passwords plays an important role in our highly digitalized world. Passwords protect our data, secrets, and even our identities. In the past few years we have witnessed a rise in hackers targeting our passwords. Too many people use a few simple passwords and pins for all their services. One leak can lead to a disaster. In this paper we propose to change that.
Password safes are the most common applications managing the world's passwords. Apps like Efficient Password Manager[1], 1Password[2] , Dashlane[3] and KeePass[4] are some popular examples. The idea is to keep all of the users passwords locked behind a master key (one password which unlocks all the other passwords) thus avoiding to have to remember all of them individually. If used properly, this can lead to strong passwords being used for the services. But more often than not, the average user will just keep his existing unsafe passwords out of convenience, not using the full potential of these applications. In this case leaks (passwords unveiled through a security hole) will still be potentially as dangerous as before. The possible revelation of the master password is an additional security hazard.

1.1 Improvement proposal

Our approach is a different one. First of all, we want to eliminate easy passwords. Every password, used by our application is random. No password is ever reused for different services. A leak now means that only one service is compromised instead of many. We do not want to lock all the passwords with one master key as it is cumbersome to use long master passwords with smartphone keyboards. We chose a multi-device approach. The passwords will be split between any number of devices depending on a security level the passwords get assigned to. Only if enough devices are physically nearby, a password can be recovered and shown to the user.
The amount of devices needed to recover one specific password should be adjustable. For example the user might want to have a password category for his social media services. Since those might not be as important to the user he could choose a security level of two. This means that at least two of his devices need to be in reach to recover any passwords inside this group. A different category could be used for various banking services like PayPal or the Online-Banking service of the users' bank. For this category the user might want to choose a higher security level since this service will not be used as often as the social media ones and some inconvenience will be accepted for increased security.
The passwords themselves are never saved anywhere except if the user chooses to export them when all devices are present.
This would typically not happen very often. When starting to use our

application the desire to keep a backup somewhere safely locked is understandable. This backup would be updated only at a point when a lot of passwords have been added.
With this approach we eliminate the risk of hackers spying out the users devices at the very moment when he enters his master password (this kind of malware is called keylogger). Using our solution, a device on its own is information-theoretically not able to recover a password of a higher security level.

2 Secret sharing

Secret sharing is the very basis of our application. The idea is to split a secret to several shares and distribute them to different participants so that at least k of them have to cooperate in order to recover the secret. It does not matter which shares one uses to recover the secret as long as there are enough of them.
In our approach we use Shamir's secret sharing scheme [5][6]. The passwords are generated by salting a 256-bit secret with the service- and the login name and hashing them with SHA-256[7]. In our application the threshold parameter k is called *Security Level*.

2.1 Basic idea

The idea behind Shamir's secret sharing scheme is to encode the secret in the coefficients of a polynomial over a finite field. A polynomial function of the form $a_0 + a_1x^1 + a_2x^2 + ... + a_nx^n$ is exactly defined by $n + 1$ points on this curve. In our application the polynomial of the desired order is randomly generated and the coefficient a_0 is defined as the secret. As many points on the polynomial-curve as there are devices using the application are evaluated. Each device receives one point ((x,y) value). This means that the security level is limited to the number of added devices because if there were not enough points the secret would be lost. After the shares were distributed, the coefficients of the polynomial and therefore the secret is destroyed. To recover it, one must gather as many points as the order of the polynomial was in the first place and solve a system of linear equations to interpolate a_0. Hence it does not matter which devices are present at the time of the recovery attempt. If we further operate over a finite field by choosing a prime number which is larger than the largest value we allow the secret to be, the secret sharing scheme is proven to be information theoretically secure. [5]

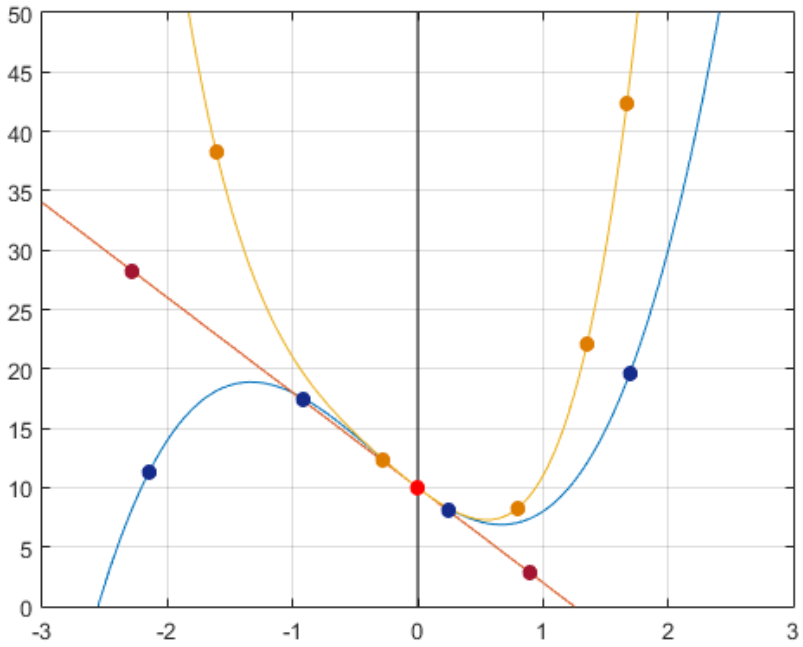


Figure 1: Polynomials of the order two, three and four with the same a_0 coefficient (bright red dot)

2.2 Invalidation of shares

One problem that may very well occur while using our password manager scheme is the loss of one device that holds one of the shares. Although this is not necessarily a problem as a thief would have to steal at least as many devices as the security level of the least secure password requires, it would be preferable to render a stolen share useless without altering the password. However the problem is that it may be impossible to signal the stolen device to delete its share as one no longer has control over it.
Shamir's secret sharing scheme provides a very simple solution that does not even require the secret to be recovered by the remaining devices. One trustworthy device generates a new random polynomial with the same order as the one used for the creation of the secret shares. The coefficient a_0 however which normally would define the secret is set to zero. Now the device evaluates this polynomial at the same x-values used for the creation of the shares in the first place (the x-value of each share does not have to be kept secret). The newly created y-values are now distributed individually to all the devices except the one that was stolen in such a way that each device receives only the y-value of the new polynomial corresponding to the x-value of the share the device already owns. Each device updates its

share by adding the received y-value to the y-value of its share. Because the polynomial interpolation over a finite field is linear, the recovery using the updated shares leads to the same secret. If the old shares and the polynomial are deleted after the update procedure, the stolen share is rendered useless as the thief will not be able to recover the secret in combination with some of the updated shares. The old shares are incompatible with the updated ones.
As long as a thief is unable to steal several devices together at once or without the owner noticing it the passwords stay safe; even if the thief is able to decrypt any data stored on the stolen device.

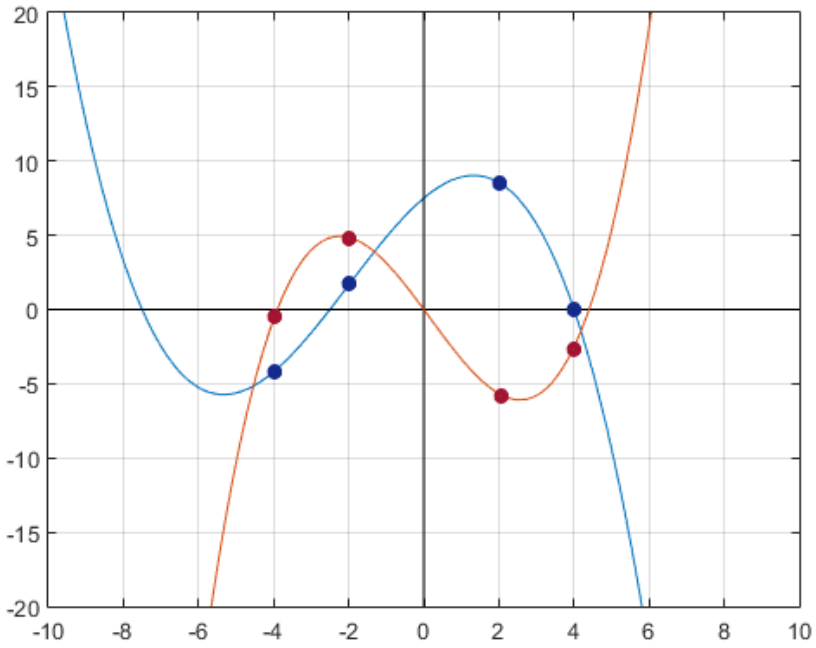


Figure 2: Original polynomial (blue) used for share generation and a new polynomial of the same order (red) used to generate update material.

2.3 Upgrading and downgrading of the threshold

It is possible that the user may wish to change the security level after the creation of a password. Our application makes this possible without the necessity to change the password. We have to distinguish two cases: If the user wants to downgrade the security level, there is no possibility other than recovering the secret and splitting it again with a new polynomial with a smaller order that has the same secret encoded in its a_0 coefficient.
If the user instead wishes to upgrade the security level, we use the same

technique as described in Section 2.2 with one small change. Rather than using a polynomial with the same order as the original one to update the secret, we use a polynomial with a higher order to produce the update values (Figure 1). This approach allows to increase the security level without recovering the secret. The used polynomial and the update material has to be deleted after it served its purpose.

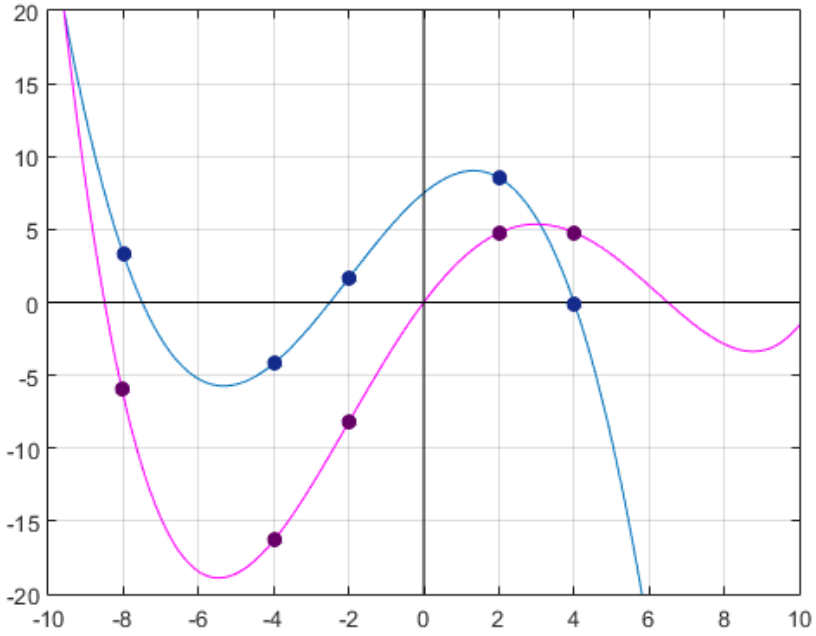


Figure 3: Original polynomial (blue) and a new higher order polynomial (pink) used for the upgrade.

3 Concept

3.1 Communication

The whole communication of the clients in our App does not solely rely on the security features of the transport channel (e.g. Bluetooth SSP[8]). The platform independent core introduces another layer of security that consists of our own protocol to ensure transport channel independent security. In order for stateless transport-channels (e.g. Android Wear messages) to work seamlessly with other protocols we use small messages that consist of plain-text data like routing information for the mesh-synchronisation or sequence numbers for connection multiplexing and encrypted data that contains sensitive information.

3.1.1 Pairing

To add a device to an existing setup, it has to be paired with another device that is already part of the given setup. We do not mean paring in the sense of Bluetooth SSP, which takes place independently on first connection attempt but our own protocol to exchange a Public-Private key pair for asymmetric encryption.
Each client has its own Key-Pair. During the pairing process both devices exchange their public key. After the user confirmed that he wishes to proceed, both devices independently generate a hash of the concatenation of both, their own and the other device's public key to produce some short human readable code for comparison on both devices to prevent man-in-the-middle attacks. If the devices were not subject to an attack, they calculate the same code. Now the device that was already part of the setup sends the Public-key of the new device to the other devices of the setup and will try to recover all secrets to redistribute them.

3.1.2 Encryption

Each message sent to other devices that contains sensitive data is encrypted. We use hybrid encryption by combining an asymmetric and a symmetric encryption algorithm. The asymmetric encryption scheme is used to encrypt a session key, that is used to encrypt all the sensitive data during one session.

3.2 Password generation, distribution and recovery

Each password managed by our Application must belong to one so called password group. Each password group has its own secret that is shared among the devices. Furthermore, each password has a service name, which describes where the password is used e.g. www.google.com and the login name. In order to generate the password, we recover the group's secret, concatenate the service and the login name, and hash the whole string. This makes it impossible to deduce another password of the same group even if one password of the group was uncovered. The safety against data loss is improved as well because the user does not have to create and hide away backups of his password every time he adds a new password. As long as the user does not lose more devices than necessary to recover the secrets, he will be able to restore every password.
The reason we do not use one secret for each password was to make it easier for the user to backup his passwords. If the user keeps a copy of the group secret hidden and safe, he does not have to create a new backup every time he changes or adds a new password. The only thing the user must remember is the service name, the login name, and to which password group the password he wishes to recover belonged to. No backup of individual passwords is necessary if no password groups are added.

3.3 Attack countermeasures

Our application was designed to withstand a few common attacks. By design we tried to minimize the possibilities of interaction between the user interface and the part of our program that stores, manages, and recovers all the secrets.

3.3.1 Man-in-the-middle attack

The man-in-the-middle attack is an attack in which the attacker tries to place herself in between of two clients that try to communicate with each other. If both devices already shared a key in the past to encrypt each message this would be impossible. If, however, an attacker replaces the key sent by each device to the other during the important phase of asymmetric key exchange with her own public key, she will be able to read and manipulate any message she wants.
In our application this critical phase takes place during device pairing where the public keys are exchanged. To prevent this kind of attack we hash the concatenation of both public keys and derive a short alphanumeric public-key verification word by using the first few characters of the result. Each device must show this verification word to the user and the user must confirm whether the words on both devices are identical. An attacker that tries to replace the public key sent by one device with her own will therefore change the verification word on the receiving device as well. If we assume that the attacker is unable to produce collisions for cryptographically strong hash algorithms, she will not be able to alter the messages during public key exchange phase without the user noticing it.

3.3.2 Message replay

If an attacker is unable to decrypt messages or produce messages that will be accepted by the victim, she may still generate harm by using an attack called message replay. The message replay attack is executed by recording valid messages sent by one of the victims and sending the recorded message again at a later time to the same or another receiver. Although the attacker should not be able to decrypt the message, she still may be able to guess what the message was intended for e.g. that during the synchronisation of device a command to delete a password the user removed on another device may be sent. If the attacker replays this or other sorts of messages at another time to the wrong receiver, one may get scenarios for which this results in unintended behaviour.
To prevent message replay attacks in the first place, we use a challenge-response system. Each message we send during a session contains a random number chosen by the sender. The message for session key negotiation at

the beginning of each session is no exception. We call this number the challenge. Each message also contains another number, the so called challenge-response. In order to be accepted by the receiver, each message must contain the challenge number of the preceding message as the challenge response number in the message to be sent. The sender simply copies the challenge of the last message received and adds a new random number by its own as the new challenge for the receiver should he wish to answer. As each message, with the exception of the public key negotiation during device pairing, is either symmetrically or asymmetrically encrypted, the attacker is unable to send messages that will be accepted by the receiver because no message will be considered valid a second time again (if we neglect the chances that the 32-bit random number is chosen by the victim as the current challenge at just the right moment again).

3.3.3 Denial of service attack

Denial of service attacks intend to make a service unavailable for normal use. Our application clearly is unable to prevent DoS attacks aimed at the connection channel. A Bluetooth jammer of course may prevent the devices from communicating with each other. Nevertheless, it is not possible to prevent the system from working over a longer period of time and without close physical proximity to the victim. Every message that is not encrypted with the private key of a valid user of the system will be discarded upon decryption attempt, long before the message is going to be interpreted. In the current version of our application it is though possible to flood a device with pairing request thus effectively preventing connections from other devices. Because only one Bluetooth connection is possible at a time and only one incoming request is processed at any time it should not be possible to make the application crash through multiple pairing requests during a short amount of time. Another obstacle for an attacker is going to be the fact that he has to find out the Bluetooth MAC-address as the devices are normally undiscoverable.

3.3.4 Compromised client

The most severe gateway for attacks against the whole system are compromised devices. If an attacker manages to gain control of a device that is paired and can communicate with other devices of the set up, she has much more possibilities to attack the whole system. However, the attacker must gain access to the database where all the sensitive data is stored. The database by itself is not encrypted as a hard coded encryption key would not be much of a hindrance and using a password to encrypt it would oppose the whole idea of master-password-less access to the user passwords.
If we assume an attacker to be able to send malicious messages without the

owner of the compromised device being aware of it, the most dangerous operations on the data is still prevented. A hacker may be able to alter, add, and delete login names and service names but as recovering the password requires user interaction on all devices to send their shares it is not possible to gain access to the passwords or group secrets (assuming the user does not accept every request that pops up on his device without any wariness). Deletion and updates affecting groups and therefore the group secrets are not processed on any device without user interaction. As we already mentioned in Section 3.2, as long as the user remembers the login name and group the passwords belonged to he will be able to recover the passwords even if an attacker deleted this information from all devices. The choice to not require user interaction for these type of changes was a trade-off between security and convenient background synchronisation we considered reasonable.
There are a few attacks against our system we are aware of. If the device that executes the splitting of the secrets or calculation of the update material mentioned in Section 2.2 and 2.3 has already been compromised, we have a severe problem. First of all the attacker may have access to the secret as it was generated on the compromised device. Furthermore, she can replace the correctly calculated update material with random data, effectively destroying the secrets if the unsuspecting user accepts the updates on the other devices. The most critical situation for this kind of attack arises if the attacker has remote access to a device on which a user just initiated the paring of a new device as it is necessary to recover every single secret to produce a new share for the newly added device. There are algorithms preventing the latter of these two scenarios considered. These counter measures are covered under the term of verifiable secret sharing but we do not yet use these algorithms in the current version of our program. Another scenario we cannot prevent consists in an attacker that spies out the device at the moment the user recovers a password and uses it to login into the service. At this moment the password, and in the current version of the program the group secret as well, is available in the volatile memory of one single device. An attacker with unlimited access to the device, possibly with the help of a trojan installed on the device that the victim is unaware of, will gain access to the password just recovered and every other password belonging to the same group. Passwords of other groups however stay safe.

3.4 Mesh synchronisation

Synchronisation of all the devices is an important an non-trivial aspect of our application. Login names, shares, share-updates and public keys of newly added devices have to be synchronized between all the devices. We did not want to rely on one device that acts as a central server for synchronisation. Our goal was to synchronize the devices in such a way that each device has

equal rights and the unavailability or even the loss of any one of the devices did not have too much of an impact on the whole system.
We did not want to calculate the difference between two device data sets and merging the sets together as this would require to exchange the whole history of the devices during every synchronisation attempt. Even if we used some sort of hash to identify the changes and only exchanged the list of these IDs, the amount of data to transmit every time would grow bigger and bigger, with the additional changes performed by the user over time. The user could change something on one device and a short time later do another change on another device. This would result in synchronisation problems if the devices did not have the chance to communicate with each other between the changes. An approach for decentralized versioning like Git[9] would therefore require to much bandwidth.
Our approach was to create a data structure that represents one single change to the data, contains a time stamp of its creation and information about the progress of synchronization. If changes affect the same data set e.g. login name we discard the change with the older time stamp. This means that during the creation of the change it is already clear which devices have to receive the change. We distinguish two kinds of these objects. The first kind contains changes that concern all devices in the same way e.g. changes to login names or deletion of devices. The second kind contains information that is intended for only one device.
The change objects that are intended for all the devices includes a map of all devices that were known at creation time and a flag whether the corresponding device already applied the change. We provide algorithms to handle cases were a device gets added to the setup after the change object was created and before it is fully synchronized. Within reasonable time intervals every device sends these change objects to all the other devices if there are still devices in the map that are not yet marked as aware of the change. The devices receiving the objects apply the changes if not already done and answer with an updated map now containing the information about synchronization status of all the devices as known to both devices. Every change object gets exponentially distributed trough the whole mesh-network and will finally disappear as soon as all devices are aware of the fact that every device applied the change.
The change objects only intended for one receiver however are handled differently. These objects typically contain new secret-shares or updates for the shares. Because only the intended receiver of the change is allowed to read the data, the change itself is encrypted by the creator with the public key of receiver. The metadata like time stamps, intended receiver and type of change is visible to everyone (the messages are still encrypted as a whole again as described in Section 3.1.2). The change object is still sent to every device in the network in order to increase the chances to reach the intended receiver.

Because shares are encrypted with the public key of the receiver as soon as they are created, neither the secret nor more than one share per device is ever stored on persistent memory. In order to ensure the authenticity of the shares and updates, the encrypted data is furthermore signed with the private key of the creator.

3.5 Message routing

Implementing the application for the Android Wear platform exposed us to some major problem. The Wear API does not allow the Smartwatches to communicate via Bluetooth with any other handheld device than the one it is associated with upon configuring it the first time. To work around this restriction we had to add message routing to our application.
Our application contains a route table were routes to every single device available at the moment are stored. In order to populate this table, the devices exchange small messages with routing information at regular intervals. The messages sent for routing purposes are transmitted in plain text to decrease network overhead for key negotiation but they do not contain any sensitive information. Furthermore, it is not possible to route the messages to any other than the paired devices and maliciously formatted route configuration messages are filtered out. Every time a message has to be sent the communication module tries to communicate with the receiver directly. If this is not possible, the route table provides the communication module with the shortest route to the destination according to a cost function, which evaluates all the information about the network known to the sender. The message is sent to the next intermediate node of the network and will travel trough the mesh until it reaches its destination. Our implementation contains logic to delete routes from the table which repeatedly fail to reach the intended receiver.

4 Implementation

The user interface is kept as simple as possible. Our main goal was convenience and an intuitive handling of the application. The colours are kept plain in shades of grey and white with an accent colour of green. Only in the wearable interface did we allow a more colourful design for the notifications, which is the event that gets triggered most often.

4.1 Architectural overview

Figure 4 gives a high level overview of the architecture of our architecture. We emphasised modularity over anything else in order to be able to port the application as easily as possible onto multiple platforms. This is why we

made the clear distinction between platform independent parts of the logic and the platform dependent ones.

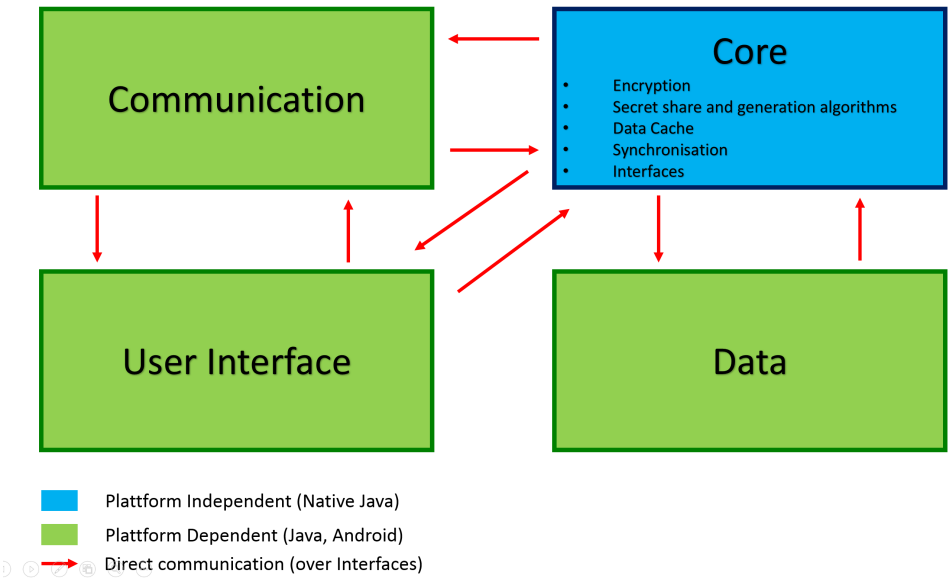


Figure 4: A rough overview of the applications architecture.

The Core is written in native java language and exported as a jar library that communicates with the other modules through a set of interfaces. The graphical user interface as well as the communication module has no direct access to the data module were sensitive information is stored. This adds an additional layer of security. Our goal was to create software, were every part of it could be easily replaced should the need arise. For example temporarily disabling encryption to make some debugging easier. In order to achieve this each module is further divided in smaller sub-modules, which interact through well defined interfaces as well.

4.2 User interface

4.2.1 Mobile application

Main Screen
The main screen (Figure 5) directly shows a list of all the passwords grouped in the different categories which are sorted by security level the lowest and most easily accessible being on top and the most secure on the bottom.

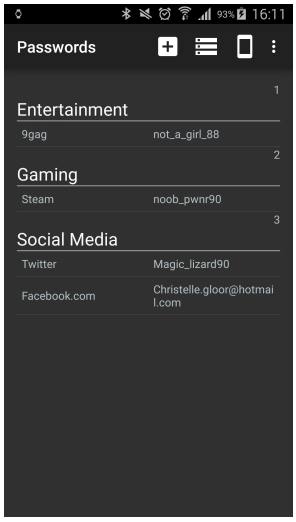
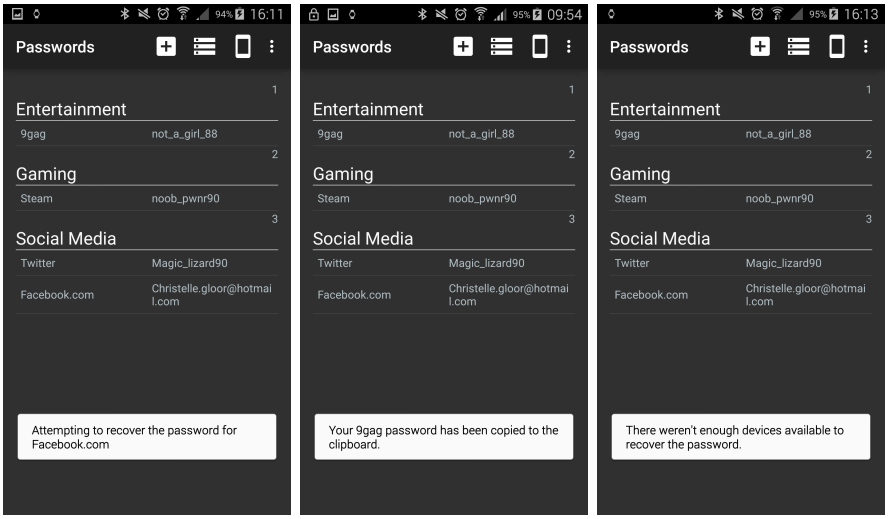


Figure 5: The launching screen of the application. The saved passwords are displayed already sorted by the categories they belong to. The lowest and therefore most easily accessible is on top. Touching the password triggers a password recovery.

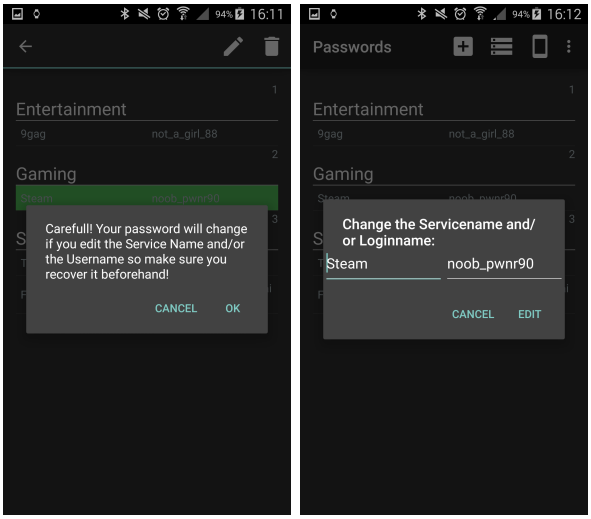
A short touch of the password triggers a password recovery. If Bluetooth is deactivated the user is notified to activate it via a toast (little messages at the bottom of the screen). After completion the password is conveniently copied to the clipboard ready for use. For security reasons the password is overwritten with the last clipboard entry after thirty seconds.
This whole process is accompanied by toasts notifying the user about what exactly the application is attempting to do after each command as a feedback (Figures 6a, 6b, 6c).



(a) Password Recovery (b) Recovery Success (c) Recovery Failure

Figure 6: The launching application with its toasts which appear (a) directly after a password recovery is triggered, (b) when a password was successfully recovered and copied to the clipboard and (c) if a password recovery failed.

The service and/or login name are easily deleted or edited by touching the password the user wants to modify for a longer period of time.



(a) Warning Message (b) Editing

Figure 7: The editing process of a service name and login name pair. The user triggers it with a long touch. First he is warned about the consequences of the edit (a). If he agrees, the editing window appears (b) allowing him to directly change the service name and login name.

The action bar (Figure 8) features simple material design icons allowing the user to add a password or to access the category management and device management. This manner of action bar design is consistent with the rest of the application.

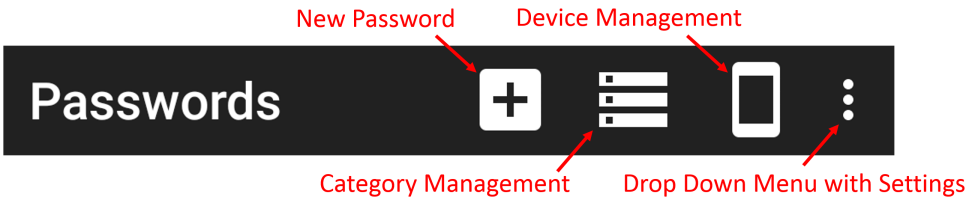


Figure 8: The action bar of the launching screen allows access to the category management, device management and the settings. The first icon is used to add a new password.

New Password

Touching the New Password icon in the launching screen action bar (Figure 8) generally leads to this screen. If no password category has been created before adding a new password the New Category screen (Figure 11 on page 20) is shown instead followed by the New Password screen after a group has been added.

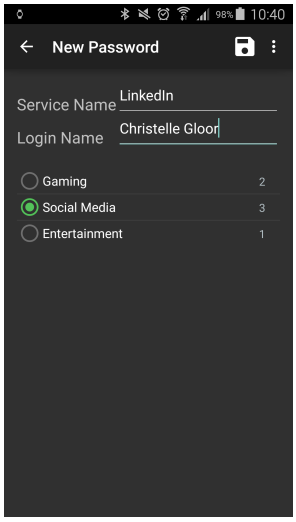
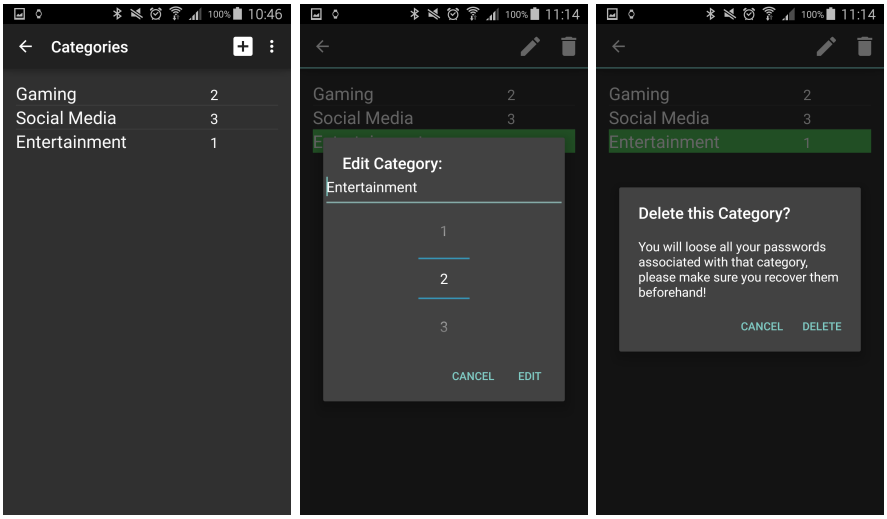


Figure 9: This screen provides the means to enter the data needed for a new password. An intuitive material design save icon is displayed in the action bar. If some data is left empty the application will not allow the user to save the password and again notify him with a toast.

Categories

A simple list view of all password groups and their security level is displayed here. The categories are easily deleted or edited through long touch. (Figure 10)



(a) Plain View (b) Edit (c) Delete

Figure 10: A list view of all the password categories and their respective security levels is found here (a). They can be edited (b) or deleted (c) through a long touch. A downgrade or upgrade can be performed. For the downgrade the category secret needs to be recreated. This is communicated via a toast. The upgrade can be done without the secret recovery. Deleting a category results in losing all the passwords associated with it. The user is warned about it.

New Category

This screen is generally accessed by touching the add icon in the Categories action bar (Seen in Figure 10a).

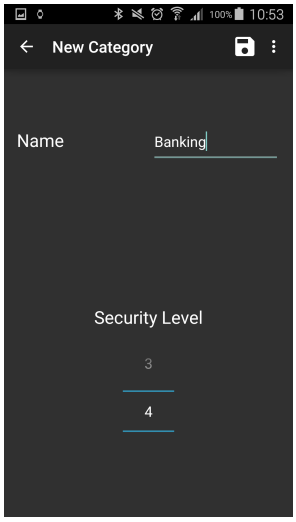
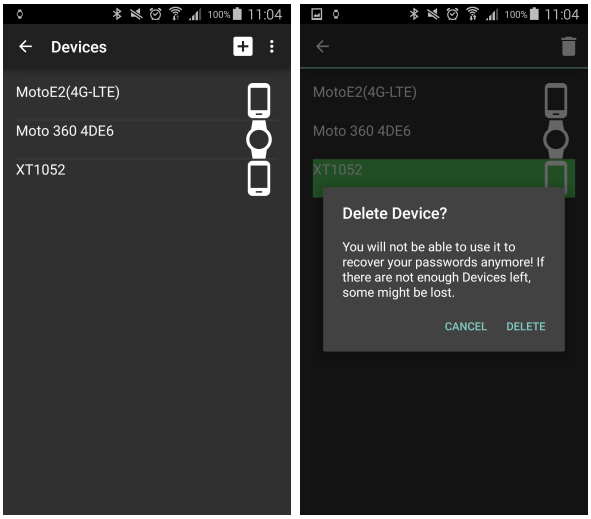


Figure 11: This screen allows to enter the data for a new category. The security level can conveniently be picked using a spinner, that only allows values from 0 to the number of paired devices. Here as well the user is not allowed to leave the name blank.

Devices

This screen is accessed over the launch screen action bar (Figure 8 page 17). All connected devices are displayed here. This is also where the user can delete an unwanted or stolen device. This results in the invalidation of the shares on the deleted device as explained in Section 2.2.

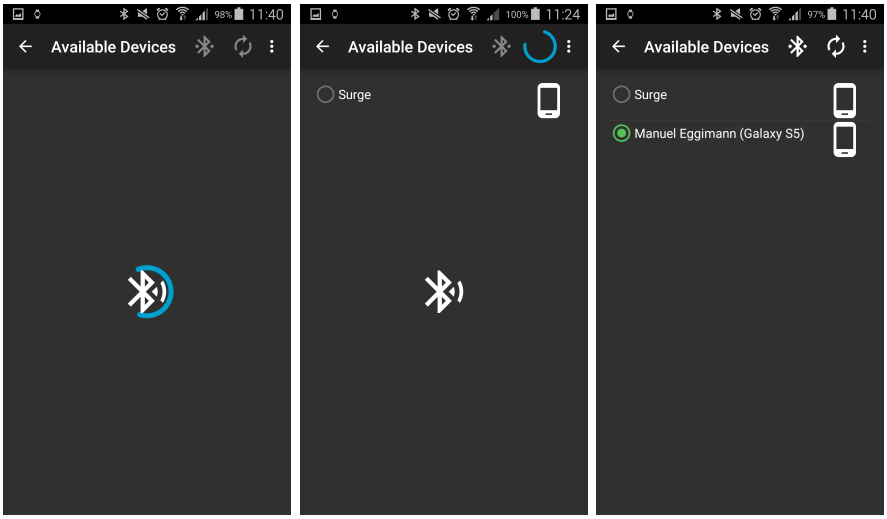


(a) Plain View (b) Deleting

Figure 12: A list view of all devices is found here (a). Android material design icons indicate the type of the paired device. The deletion process is triggered with a long touch. Before executing it the application warns the user about the consequences and asks for confirmation (b).

Available Devices

This screen is accessed over the add device icon in the Devices screen action bar (Seen in Figure 12a on page 21).



(a) Initial Scan (b) Rescan (c) Devices

Figure 13: When this screen is opened, the application directly starts scanning for visible Bluetooth devices (a). If Bluetooth is deactivated, the user is notified to activate it via toast. If the scan returned any results, they are displayed in the same manner than as in the Devices screen with additional radio buttons to select the device the user wants to add to his setup (c).

Settings

The settings are accessible from all screens via drop down menu on the top right corner as is the norm with android applications (Seen in Figure 8 on page 17).

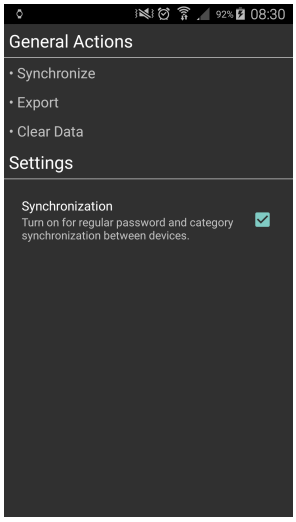
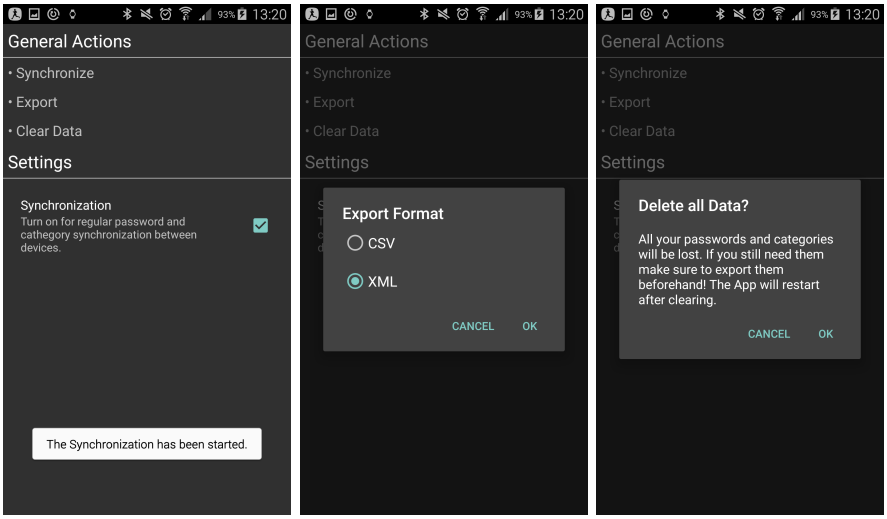


Figure 14: The application settings screen where some general actions may be triggered. This is also where the user has the possibility to turn the regular background synchronization (every 90 minutes) on and off.



(a) Synchronization (b) Export (c) Clear Data

Figure 15: *Synchronize* (a) manually initiates the synchronization between devices. *Export* (b) exports the data into one of two formats and allows the user to send it to an external application. The *CSV* format attempts to recover all passwords and saves them together with the service and login name. The *XML* format does the same but exports a lot more data. This includes the secret shares of the different groups, public keys, the private key, algorithms as well as encoding parameters and short descriptions on how the algorithms play together. If a password could not be recovered at the time of the export but the secret share is known, a person with some programming knowledge could still reconstruct the password. *Clear Data* (c) clears all the application data after the user confirmed.

Notifications

The application often has to directly communicate with the user. Whenever a password with a security level higher than 1 needs to be recreated (Figure 17) or a device is being added or any other event happens which for various security reasons needs confirmation from the user, notifications are used.

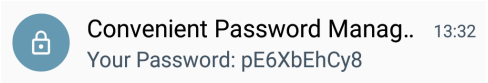


Figure 16: This notification pops up when a password request has finished successfully. The password is displayed in plain text until the user swipes away the notification.

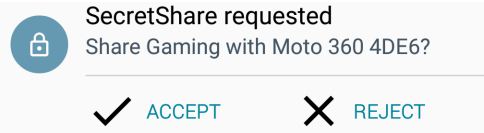


Figure 17: This notification appears when a different device has triggered a password request with a security level greater than 1. All reachable devices will be asked to share their secret. If enough of them accept, the password is recovered on the device that started the request.

4.2.2 Android Wear application

This application is much more rudimentary than the mobile application. A watch only has to be able to show what passwords are available to be able to send a recovery request. A smart watch is less convenient to use than a smart phone. The small display makes it especially hard to enter text. Since it can not be used without the associated phone we considered Device Management, Category Management and Settings to be unnecessary. All of these things can be accessed and used more easily on the phone.

Main Screen

After a launch screen providing an entry point to the password categories and possibly more in the future the main screen is reached (Figure 18).

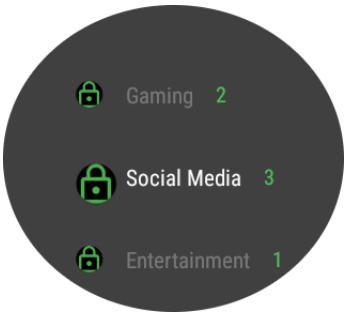


Figure 18: The password cathegories and their security levels are displayed here. Touching a category opens the Password View.

Password View

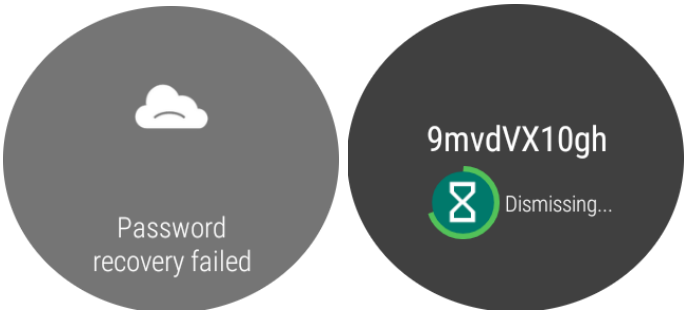
After touching a password category on the main screen the application shows all the passwords in that category (Figure 19).

Figure 19: All the passwords associated with the chosen password category. Touching a password triggers a password recovery.

Password Recovery

After a password recovery is triggered a colourful dynamical loading screen is shown. (Figure 20). The user is informed about the failure or success of his request (Figure 21).

Figure 20: Dynamical Loading Screen shown at the start of a password recovery attempt.



(a) Failure Screen (b) Success Screen

Figure 21: On completion of the password request the user is either informed that it has failed (a) (for example if there were not enough devices around for the needed security level) or the password is displayed on the screen for twenty seconds (b). If the user wishes to keep it on screen for a longer period of time, touching the hourglass resets the timer.

Notifications

Necessary communications with the user are also handled by notifications.

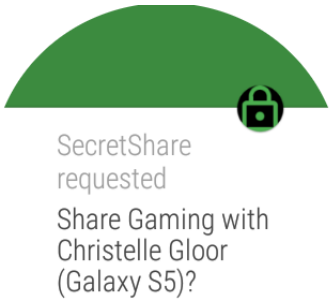
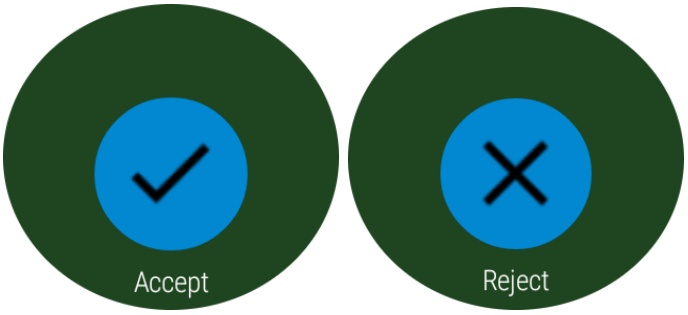


Figure 22: This notification pops up whenever a password is requested with a higher security level than one.



(a) Accept Screen (b) Reject Screen

Figure 23: The answering possibilities are handled with separate screens to accept (a) or reject (b). The user switches between them by sliding the screen to one side or the other and accepts or rejects by touching the appropriate icon.

4.3 Encryption

We use 2048-bit strong RSA public-private key pairs to negotiate a session key that is used to encrypt the messages through symmetric AES encryption. The 256-bit key to use for AES encryption of one session is chosen at random by both, the receiver and the sender of the first message. Both devices independently combine some random data sent to each other with the XOR-Operation to obtain the key to use for AES encryption. The random data to produce the session key is encrypted using each devices' private key, whose corresponding public key was exchanged during the paring phase. We use OAEP for padding the data encrypted with RSA and CBC to pad the messages encrypted with AES.

4.4 Transport channels

In the current version of the program we solely use Bluetooth for communication but everything is written in a modular fashion so adding new transport channels is as simple as adding a new class that implements a certain interface. We use two different APIs for communication through bluetooth.
For data exchanges between handheld devices we rely on low level RFCOMM sockets using classic bluetooth. The reason why we chose classical Bluetooth rather than the more modern Bluetooth Low Energy was mostly inexistent support of most of the current hendheld devices. Although most of the devices are able to use the so called *central mode*, very few of them until now are capable to use the *peripheral mode* needed to advertise themselves to other devices[10]. The disadvantage of classic bluetooth however is that connection attempts are quite slow and there is no possibility to detect whether a paired device is available (unless it is permanently in discoverable mode, which would be a privacy issue) other than trying to connect to the device.

During our own testing we saw that it may take several seconds before the connection attempt succeeds or fails. Therefore, we decided to use a hybrid approach: Devices that are able to advertise their availability with the Bluetooth Low Energy beacons will constantly advertise themselves. Most of the devices are able to scan for these small packets very fast. The advertisement beacons do not contain any personal information. A randomized MAC-address and a specific unique service identifier, that allows scanning devices to distinguish between devices running our application and other Bluetooth LE peripherals is advertised. The application now only tries to connect to devices for which it detected an advertisement packet or to devices that are marked as unable to send these kind of packets. However, the implementation of the Bluetooth connectivity was the biggest problem of the project. We did not manage to achieve stable Bluetooth communication when performing fast connection switches between the devices. Our observation was that connection failures tended to be more likely if the involved devices are already connected to other peripherals e.g. smartwatches or Bluetooth Speakers. Furthermore, the fail rate of the connection attempts was very unsteady and strongly dependant of the number of clients.
To communicate with Android Wear devices we rely on a high level API that provides fast availability detection and reliable message transmission.

4.5 Data storage

The persistent data is stored in a relational SQLite database without encryption. All the Public-keys and the devices own Private-key are also found here. We considered saving those data somewhere else. But considering an attacker with unlimited access to a device there would not be any security benefits. The Data module can only communicate with the user interface or the communication module via the core module making it less susceptible to various attacks.

5 Conclusion and outlook

The biggest problem during the development phase that is not solved until now is the unreliability of the Bluetooth stack on newer android devices. Fast switching or parallel connections are a huge hassle and extensive testing brought us to the conclusion that conventional Bluetooth just does not work well enough for low latency peer-to-peer communication. Therefore, the first challenge to tackle in further development on this application will be the search for a more reliable alternative to RFCOMM-sockets for communication between handheld devices.
Along with the development to further improve the stability, enhancing the user interface, and supporting more platforms there are a few features we were not able to implement yet. The fact that all passwords are generated

at random and user defined passwords cannot be used with our application was a design choice to force the user to switch to secure passwords instead of sticking with the easy-to-remember- and therefore easy-to-hack-passwords he used before. Still we realize that the complete absence of custom passwords is a disadvantage that we would like to address in the future. Another idea we had in mind since the beginning was to introduce the possibility to generate and scan QR codes as a substitute for devices. People that do not use many devices in daily life would benefit from this possibility as a small QR code printout easily fits into the wallet and a few QR codes hidden at physically separated places would be a much more secure backup of the passwords than a plain-text printout. Getting a hold of one QR code would just be the same as stealing one device.

References

[1] : Efficient password manager. http://www.computerbild.de/download/Efficient-Password-Manager-5149949.html (dec 2015)

[2] : 1password - password manager and secure wallet. https://itunes.apple.com/us/app/1password-password-manager/id568903335?mt=8 (dec 2015)

[3] : Dashlane password manager & secure digital wallet. https://itunes.apple.com/ch/app/dashlane-password-manager/id517914548?mt=8 (dec 2015)

[4] : Keepass a lightweight and easy-to-use password manager. http://sourceforge.net/projects/keepass/ (dec 2015)

[5] Shamir, A.: How to share a secret. Communications of the ACM **22**(11) (nov 1979) 612–613

[6] Sira Salim, Sruthy Suresh, R.G.R.S.: Application of shamir secret sharing scheme for secret data hiding and authentication. IJARCST (2014)

[7] Datenschutz, D.: Passwort sicherer mit hash und salt. https://www.datenschutzbeauftragter-info.de/passwort-sicherer-mit-hash-und-salt/ (mar 2013)

[8] Group, U.E.: Bluetooth user interface flow diagrams for bluetooth secure simple pairing devices. Technical report, Bluetooth special interest group (2007)

[9] Scott Chacon, B.S.: Pro Git. Apress

[10] Google: Bluetooth low energy. http://developer.android.com/guide/topics/connectivity/bluetoo
le.html