



***Design, implementation and analysis of a
theft-resistant password manager based on
Kamouflage architecture***

by

Torstein Haugum & Lars-Christian K. Rygh

Supervisor

Vladimir A. Oleshchuk

This master's thesis is carried out as a part of the education at the University of Agder and is therefore approved as a part of this education. However, this does not imply that the University answers for the methods that are used or the conclusions that are drawn.

The University of Agder
Faculty of Engineering and Science
Department of Information and Communication Technology

Grimstad, May 26, 2015

Abstract

As a solution for helping companies and users in the constant security dilemma of obtaining and using passwords in the securest ways possible, password managers have become custom around the globe. The design architecture on what development of password managers are based on, preserving authenticity, usability and reliability are principles that keep systems secure and defend against attacks or unfortunate circumstances. The design principles however, have changed little over time. After researching password managers and analyzing overall security, we use our findings to develop a design based on the implementation of decoys, customized for Android. This development was inspired by a paper named "Kamouflage: Loss-Resistant Password Management" [27].

Preface

The work carried out in this thesis is done at the University of Agder and is our final task before completion of the master programme "Information and communication technology" with security as the main profile.

We will use this chance to thank our supervisor Professor Vladimir A. Oleshchuk for helping us academically and giving valuable input regarding our task.

This thesis is written by Torstein Haugum & Lars-Christian K. Rygh.

Contents

Contents	ii
List of Figures	v
1 Introduction	1
1.1 Motivation	2
1.2 Goals	4
1.3 Field of research	4
1.4 Statement of the Problem	4
1.5 Contributions	5
1.6 Target audience	6
1.7 Report outline / Thesis Organization	6
1.8 Delimitations and key assumptions	7
2 Background	8
3 State of the art	11
3.1 General information	11
3.2 Local password managers	12
3.3 Web-based password managers	12
3.4 Proposed solutions using decoy-based architecture	13
3.4.1 Kamouflage	13
3.4.2 NoCrack	14

CONTENTS

4	Design	15
4.1	Application design	16
4.1.1	PIN code	16
4.1.2	Master Password	17
4.1.3	Login	18
4.1.4	Creating passwords	20
4.1.5	Calculating the correct index	20
4.1.6	Database	24
4.2	User operations	26
4.2.1	Add	26
4.2.2	Read/Show	26
4.2.3	Update/Edit	26
4.2.4	Delete	27
4.2.5	Change master password	27
4.3	Cloud design	28
4.3.1	Production of decoys	28
4.3.2	Database	31
4.4	Connection between android application and the cloud	32
5	Proposed Solution	34
5.1	Software used	36
5.2	Random functions	36
5.3	Android development	37
5.3.1	Tools used	37
5.4	Hexstring to decimal	37
5.5	Cloud - production of decoys	39
5.6	Decoy generation	43
5.7	Discussion of design issues	45
5.7.1	Login	45
5.7.2	Internet / No Internet connection	46
5.7.3	Using modulo arithmetic for indexing	47

CONTENTS

6	Performance testing	48
6.1	Android application performance	49
6.1.1	How much MB space do the decoys produce?	49
6.1.2	Do large amounts of decoys slow down the application?	49
6.1.3	Probability of hitting the correct index	50
6.1.4	Hit rate with 1000 decoys	53
6.1.5	Hit rate with 5000 decoys	54
6.1.6	Hit rate with 10000 decoys	55
6.2	Cloud performance	57
6.2.1	Decoy generation	63
7	Security analysis	69
7.1	Attack vectors	69
7.1.1	Attack vectors targeting the application	69
7.1.2	Attack vectors targeting the cloud	72
7.2	Additional functions to enhance security	74
7.2.1	Honeywords	74
8	Discussion, conclusion and further work	76
8.1	Discussion of results	76
8.1.1	Hit rate results	76
8.1.2	Decoy production in Cloud	78
8.1.3	Storing passwords	81
8.1.4	Modulo as index function	81
8.1.5	Performance	82
8.2	Conclusion	82
8.3	Further Work	85
	Bibliography	86

List of Figures

4.1	Login with master password	19
4.2	Indexing	22
4.3	Even distribution	24
4.4	Database design	25
4.5	Decoy production overview	28
4.6	Original password: Password123!	29
4.7	Database design in the cloud	32
5.1	General overview of our prototype	35
5.2	Code snippet of SHA-256 hex value converted to decimal	38
5.3	Check with 0-100 000 decoys in the database	41
5.4	Average over 10 runs: 1,036 Sec	42
5.5	Screenshot from the database in the cloud	42
5.6	Passwords being sent in JSON-format	42
6.1	Application performance	50
6.2	Hit rate with 1000 decoys	53
6.3	Hit rate with 5000 decoys	54
6.4	Hit rate with 10000 decoys	55
6.5	Decoy production 10000 - 10 times	58
6.6	Decoy production: 15.5828 sec	58
6.7	Improved time Decoy production: 9.575 sec	59
6.8	Improvement: 6 sec	59

LIST OF FIGURES

6.9	Production of 10 000 x 5 (Time in seconds)	60
6.10	Iterations	61
6.11	Unique Lookup check with 0-100 000 decoys in the database . . .	62
6.12	Comparison of production 10 000	63
6.13	Original password:PASSWORDDEViCE	65
6.14	Original password:PASSWORDDEViCE	66
6.15	Original password: passwOrDDeViCE	66
6.16	Step 1: Choice of words, separated with '-'	67
6.17	Step 2: Choice of uppercase letters	67
6.18	Step 3: Choice of numerals	68
6.19	Step 4: Choice of special characters, final result	68

List of Abbreviations

- **APK** Android Application Package
- **CA** Certificate Authority
- **PHP** PHP: Hypertext Preprocessor
- **KB** Kilobytes
- **MB** Megabytes
- **GB** Gigabytes
- **RAM** Random access memory
- **DoS** Denial of service
- **IP** Internet Protocol
- **AES** Advanced encryption standard
- **SHA** Secure hash algorithm
- **USB** Universal serial bus
- **PBKDF2** Password-Based Key Derivation Function 2
- **SSL** Secure socket layer
- **PBE** Password based encryption

Chapter 1

Introduction

Password managers is a solution that helps companies and private users in structuring their password and login credentials. The amount of systems we access daily have increased, and restrictions in the creation of secure passwords become harder to satisfy [7, 36, 49]. Remembering all of these passwords is difficult, and only having to remember one single password with the use of password managers, is a solution many prefer [47].

Password managers have to be secure, as people trust these applications with their secrets [47]. The challenge is to make an application secure enough, but still user friendly so that security is not indirectly compromised because of complexity [30]. It has been proven several times that password manager solutions are not impenetrable, and in some cases security is decreased as a sacrifice for usability [26, 22, 41].

One of the contributions and new solutions we based our research on is the implementation of decoys. The theory is that by preventing brute force attacks (a highly popular method of penetrating systems), a higher level of security is obtained, without a decrease of usability. There have been proposed solutions that

help protect against these types of attacks earlier by slowing the process down, but none of them actually stop the possibility of attacking in this manner. We believe that by implementing a decoy based design in the correct way, brute force attacks can be categorized as a minimal threat [27].

1.1 Motivation

By studying Computer Security, it has become apparent that there is always a balance between absolute security, and usability. Creating systems that are secure will not contribute if they are so hard to use that they are not used correctly, or disregarded by users [30].

Systems are often protected by passwords and encryption, in one form or another [34]. Some systems implement more steps for login; one example would be second factor authentication [31]. Even though these solutions help, they are basically the same, nothing revolutionary.

While developers are in the process of creating secure systems, the main motivation is to prevent security breaches, which is of course important. The reason for this is based on three points:

First point: Systems that demand some form of security, often implement the use of passwords in one way or another. There are few, if any ways around this because of authentication [31].

Second point: Because of the implementation of passwords restricting systems, security is based on how secure the password mechanisms are, either calculated from the probability of guessing a password, or gaining access in other forms [49, 22].

Third point: As most systems use passwords to protect their system, a well known method of penetrating these types of systems, is brute force [49]. By brute force we mean automated scripts or applications used offline to generate enough permutations / dictionary combinations for guessing passwords, which will result in success. If an attacker has enough time and the system is offline, the attacker will eventually be successful in obtaining the password. This is why encryption is highly important, creating an immense amount of possible combinations, resulting in a situation where the amount of time needed to guess the password is categorized as secure. As technology and resources evolve, encryption in itself does not stop the problem, it only delays the process [27]. This customary way of securing systems with the implementation of passwords might one day have to change, especially in regard to the possibilities of renting botnets [33] to perform tasks, gaining unthinkable amounts of resources.

We ask ourselves, what if in the development process of designing a secure application, it is expected that attackers will gain access to the system, then what?

The motivation for this thesis was to challenge this way of thinking head on, potentially uncovering new approaches that can benefit the industry. Our design implements solutions based on the unwanted. Attackers penetrating the system won't know if the information given is false or real information. The result may be categorized as fooling, or even trapping the attacker, even if they are aware of the design.

1.2 Goals

By researching decoy-based architecture we wish to achieve the following goals:

- Research known weaknesses in password managers and focus on the weaknesses that relate to architecture and design.
- Implement a proof of concept application on a mobile device with our own decoy-based design.
- Investigate by proof of concept if enough decoy passwords can be produced in a feasible amount of time.

1.3 Field of research

The field of research in this master thesis is focused on password managers and security in relation to brute force attacks. We examine a new architecture, built on the implementation of decoys. It is believed that with the use of decoys, a higher level of security is achieved, as information is camouflaged and it becomes harder to confirm if a brute force attack is successful [27].

1.4 Statement of the Problem

Brute force and dictionary attacks are expected in regard to systems that demand different forms of authentication for access. Protecting systems for users, making it difficult for attackers to gain unauthorized access is an ongoing struggle as dictionaries used in brute force attacks, and computing power increases [26]. Having users implement the use of strong passwords that are hard to guess often help prevent dictionary attacks [49]. Implementing slow hash functions and salts

also help slow down the attack process, but they do not eliminate the possibility of dictionary attacks from being successful [27]. Password managers are increasing in popularity by users because of higher demands in using strong passwords, making passwords harder to remember. An increase of password managers being used leads to the assumption that password managers can become popular targets for attackers. If an attacker was to gain unauthorized access to a managers credentials, they obtain highly valuable information [47, 37].

1.5 Contributions

With the implementation of decoys, it is believed that a higher level of protection is achieved against brute force attacks. To be categorized as a valid solution, implementation must take place on different platforms for analysis.

Sufficient decoy production is 'key' in developing a secure solution. The decoys generated must successfully camouflage stored information. This task can be challenging as users' passwords often contain words [36].

By implementing a prototype on a mobile device running Android, we are able to analyze and evaluate if the implementation of decoy-based architecture is feasible. Decoy-based designs demand a large amount of decoys to achieve an acceptable level of security. Resources needed, as available space, hardware power and communication speeds, could make implementation more difficult. We have proven that producing enough decoys of sufficient quality is possible with the use of cloud servers and efficient programming, and shown that running our decoy-based solution on a mobile device is highly possible.

1.6 Target audience

The target audience are users in the security industry, application developers and others who share the common interest of Computer Science.

1.7 Report outline / Thesis Organization

The report outline of our thesis is structured as follows:

Chapter 1 - This chapter is the introduction to our thesis. This chapter addresses our motivations for this thesis including our goals.

Chapter 2 - This chapter is the background of our thesis. Information that is needed in order to understand the thesis is mentioned here.

Chapter 3 - This chapter is the state of the art mentioning overall password managers and their different architectures.

Chapter 4 - This chapter is about the design of our password manager. How the application functions is mentioned here.

Chapter 5 - This chapter is the proposed solution which mentions important functions necessary for our design. It also contains information about design issues.

Chapter 6 - This chapter show results of our application in how it performs.

Chapter 7 - This chapter is a security analysis of our password manager mentioning attack vectors for our application.

Chapter 8 - This chapter discusses the results performed by our application including conclusion and further work.

1.8 Delimitations and key assumptions

We have not implemented the following into our solution:

- Encryption
- SSL between the application and the cloud environment
- Internet / No Internet connection regarding synchronization of decoy production (Explained further in 4.3.1)
- Obfuscation of source code in the application
- Strictly hash storage in the cloud, preserving privacy for the user (used for testing)

We have not incorporated machine learning to conduct an analysis of our decoy production.

Chapter 2

Background

In this thesis we emphasize security, not safety. These terms are not easy to distinguish from each other as both terms often are defined by context or presentation. The word 'safe' comes from the Latin word 'saluses', translated as 'uninjured', where 'safety' is a condition of being protected against risk or injury. The word 'secure' comes from the Latin word 'securus', a condition of being free from danger or threat, subtle differences, but within computer science we recognize the term security as - "protects against both deliberate and unintended attacks", while safety generally protects against unintended attacks [19].

The importance of strong passwords is well known not only in the security industry, but also by regular citizens as well. This has become common knowledge. However, creating a strong password, and actually using strong passwords are two different matters. The dilemma is easy, stronger passwords are harder to remember, and while the amount of different services used increases, a larger amount of passwords have to be remembered [53]. This results in users implementing if not the exact same passwords on multiple services, then very comparable passwords, that in them selves are not necessarily strong at all. [38, 37, 55].

If complete words are used in the creation of a password, this helps an attacker in the scenario of disregarding possible password combinations, especially if the attacker knows what language the user speaks. This way, a massive amount of possible combinations needed to brute force the application is decreased as attackers may implement the use of dictionaries. If a user has a password with a length of 8 characters, it is assumed that the possible permutations are in relation to the amount of characters available for use, in the power of the length of the password [49], but this is not necessarily correct.

(An example would be if a password consisted of 8 characters, containing only lowercase letters from the English alphabet, possible permutations would be 26^8 .)

If we were to assume that all passwords were completely randomly generated, this would be true, but surveys have discovered that most people use passwords implemented with words, drastically decreasing the possible combinations in a users password [49].

While companies and developers try to enforce strong password creation on to users by implementing restrictions when creating passwords, this may lead to passwords being reused on multiple services, being forgotten, or even worse, users writing the passwords down on paper or their smart phone. All of these scenarios combined, results in the creation of Password Managers, a secure solution of saving valuable information such as passwords [41, 7, 37, 55].

Password managers are applications that secure valuable information, with restricted access. Many of them are also implemented with functionality making them easier to use, examples being auto fill when using a web browser, or syncing to other devices so the information is accessible everywhere. If a password manager allows the use of passwords automatically, then the passwords can be so strong that there is no need to remember them [26]. Password managers can be

popular objectives for attackers, as the information to be gathered is of high value. By researching the security in these types of applications, patterns start to occur.

Several research papers prove that password managers are not impenetrable, which is a scary truth. If we trust these applications with all of our personal access information, a successful attack could be catastrophic [27, 22, 41, 47, 37].

Standard systems store real information and nothing else. Valuable information is often encrypted, relying on the fact that the encryption used is secure. The result of this is that if a brute force attack was to be successful, the attacker would know with certainty that all information obtained is real. By implementing a decoy-based architecture, security will be increased in the situation of an attacker penetrating the system, as information obtained will be camouflaged. The attacker can not distinguish between what is false or real information. This increase of security has little to no sacrifice of usability for the user as the changes only become apparent in the situation of an attack [27].

Chapter 3

State of the art

3.1 General information

A traditional password manager protects sensitive information with a master password. This master password is used to encrypt the information it is supposed to protect [37]. Protecting the information in this way makes it vulnerable to brute force attacks. If an attacker manages to obtain a copy of the encrypted database he can perform an offline brute force attack trying to guess the correct key for decryption. When a wrong decryption key is used the encryption will fail and the attacker will keep on trying until the decryption is successful. Upon success the attacker will retrieve the correct information and is then able to authenticate himself. Whereas a password manager using decoy-based architecture the attacker will never know if the information he receives is real or decoy information. The only choice left is to try the results online.

3.2 Local password managers

Local password managers, also referred to as stand-alone password managers, are applications that let users store sensitive information inside a single vault, that is protected with a master password. If a user needs to access his sensitive information, he must decrypt the vault by entering the correct master password. These password managers store the encrypted file locally, so if a user wants to use this file on another computer it can be saved on a USB flash drive and moved over to the desired computer. The disadvantage is that every computer has to have the application installed [29].

A well known application with this type of architecture, is Keepass [9]. Keepass is an open source password manager that encrypts sensitive information with the help of a master password or a key file. To protect against dictionary attacks Keepass uses the encryption algorithm AES-256 [32] and the hash algorithm SHA-256 [40].

3.3 Web-based password managers

Web-based password managers differ from local password managers in the way that they run in the browser. Web-based password managers store the users passwords in a cloud where the users have to authenticate themselves in order to retrieve the desired information [47].

A popular web-based password manager is LastPass [10]. LastPass both encrypts and decrypts sensitive data on the client side before synchronization, ensuring that the data is only accessible to the user with the correct master password. Lastpass uses the combination of AES and "Password-Based Key Derivation Function 2 [35]", to secure the sensitive information.

3.4 Proposed solutions using decoy-based architecture

3.4.1 Kamouflage

In 2010 four researchers introduced a new architecture for password managers based on decoys. The solution they proposed is named "Kamouflage" [27]. The idea behind this architecture is to make password managers safer in regard to anti-theft. With this design an attacker is forced to perform a great amount of work in order to determine if a password is correct or not. Traditional password managers only store a single set, which is encrypted with a master password. If an attacker is able to steal or obtain the mobile device, he can run a brute force attack in order to find the correct password, and eventually decrypt the database to obtain all the sensitive information stored.

The idea behind "Kamouflage" architecture, is to store N plausible (decoy) sets along with the real set. The purpose of these sets is to camouflage the real set so an attacker can not with certainty confirm if the correct set is found. Each set is encrypted with a decoy master password generated from the original master password. If an attacker is able to steal a mobile device with a password manager using the "Kamouflage" architecture, he would have to decrypt each set separately, and try to use the information given online in order to determine if he successfully decrypted the correct set or not. Ultimately the idea is to lure an attacker to think that he has successfully decrypted the real set only to find that it was decoy information. The researchers in "Kamouflage" also mention the use of honey words [27, 46, 39] which can be used to detect when an attacker is actually trying to decrypt the database. A user can take a portion of the decoys as honey words and add them to websites. When these decoys are used in the attempt of accessing a given website, an alarm would go off or block the user account. This type of technique has also been used in "ErsatzPasswords—Ending Password Cracking" [20]

where the researchers propose a solution named "ErsatzPasswords" that protects stored password hashes. This scheme is designed so an attacker thinks he successfully decrypted the password file of hashes. When an attacker tries to crack the hashes in the password file he will receive "ErsatzPasswords", also known as "fake" passwords or decoys. If the attacker tries to use these passwords when logging into the system an alarm will be triggered, exactly as with the intended use of honey words. This may imply that login credentials are leaked and that an attacker is trying to gain unauthorized access to the system [20].

3.4.2 NoCrack

Other researchers have recently proposed a new solution regarding cracking-resistant password vaults named NoCrack [28]. This solution is based on weaknesses they identified in the proposed solution by "Kamouflage". They mention that "Kamouflage" is degrading overall security relative to a traditional PBE (Password based encryption). One of the weaknesses the researchers mention is that in "Kamouflage" the decoy sets are encrypted with decoy master passwords generated with the same template as the original master password. If an attacker is able to successfully decrypt one of the decoy sets he would then know the template of the original master password. This would initially help an attacker to narrow down the search of decrypting the remaining sets. The second vulnerability they mentioned is how "Kamouflage" produces decoy master passwords. NoCrack differs from "Kamouflage" in the way it produces decoys. The researchers were inspired by a theory named honey encryption [45]. The main idea behind honey encryption is that when an incorrect master key is used in the attempt of decrypting a cipher text, it will avoid producing an error and instead return a result in plain text that looks real.

Chapter 4

Design

The design of our application will always grant access, regardless of what you enter as a master password. The traditional design of a password manager today, one example being KeePass[9], only grants access if the master password is correct. This master password is used to encrypt the sensitive information that is stored. When a user types in the correct master password the database is decrypted and the information is revealed. If the input (master password) is wrong, the decryption will fail and the application will cast an error. The main flaw with this traditional design is that an attacker knows when he typed a faulty password. If an attacker manages to steal a device with a password manager installed, based on a standard design, or if the attacker obtains a copy of the encrypted database file, he will definitely try a technique called brute forcing [51].

Brute-force attacks are techniques used by attackers were they attempt to guess passwords [51]. An attacker attempting to brute-force has to have in mind that the password he is trying to break can contain all possible combinations of upper and lower case letters, numbers, and special characters. Password space is infinite and it could take years before finding the correct password. Therefore attackers use tools that help narrow down the exhaustive search to speed up the process, often

implementing the use of dictionaries. It is known that users tend to use passwords that are human-memorable and that makes them vulnerable to brute-force dictionary attacks [49, 55].

To prevent brute-force attacks we have based our design on an architecture called "Kamouflage" [27]. Our application is designed to grant access to anyone regardless of the master password given. By designing the application in this way the attacker will never know when he has typed in the correct master password, thus making brute-force attacks difficult. In a traditional design, every time the attacker receives an error, the probability of guessing the correct password will increase. With our design the attacker does not know whether the information given by the application is correct or wrong.

4.1 Application design

4.1.1 PIN code

Since the application allows all inputs, anyone will get access. We could have implemented a PIN code of four digits, but this would not enhance the security of the application in any way. With ten digits to choose from and with a length of four digits, math shows that there are only 10^4 combinations. Users also tend to pick guessable PIN codes, such as their birthday or certain patterns [24], which is the first patterns an attacker would try. From a security point of view this would not stop an attacker for a very long time, and therefore we have chosen not to implement this. The only reason to implement this would be to add an extra step to help prevent friends or others from updating, deleting or adding random data for fun. It is also worth mentioning that most smart phones are implemented with a PIN code to access in general. Users often implement the use of the same PIN codes on multiple access restrictions. If an attacker is able to open the application

he already has access to the phone, giving the assumption that the attacker already knows the general PIN code.

4.1.2 Master Password

The first time the user opens the application he will be asked to create a master password. The master password must meet some specific criteria before it will be approved by the application. This quality check ensures that the user gets a strong and secure master password. A user should be able to remember a master password of length 8 combined with these restrictions. Even though the application will allow all arbitrary input, it is important that the user is forced to create a strong password containing different types of characters. This will enhance security in relation to an attacker guessing what the correct master password is with the use of side information about the user.

To increase security even further the master password is not stored in the application. If we chose to store the password in the application, this would result in precautions having to be made in terms of storing the password securely. An example would be to run the password through a cryptographic hash function such as SHA-256 [40]. If we had done this, the attacker would know that if he was able to generate the same hash, he would obtain the master password. This could be achieved with the use of brute force, opposing the principles of our application.

The master password has to fulfill the following criteria:

- Does it contain a digit(s)
- Does it contain upper case letter(s)
- Does it contain lower case letter(s)
- Does it contain special character(s) specified by us
- Is the length greater or equal to 8 characters.
- No occurrences of white space is allowed

4.1.3 Login

Figure 4.1 shows the login design of the application, how a user accesses and retrieves information from the application. This example shows the scenario where a user has logged in before and created a master password.

1. In order to access the application to retrieve the correct information, the user has to enter a master password that has already been created by the user.
2. Upon success the user will gain access to the application and will be met with a list of options. As shown in this example the user has chosen the option "Show entries"
3. Choosing the option "Show entries", the application will open a new window with more detailed information about the selected entry.
4. The new window will contain information such as service, username, password and url, for the selected entry.

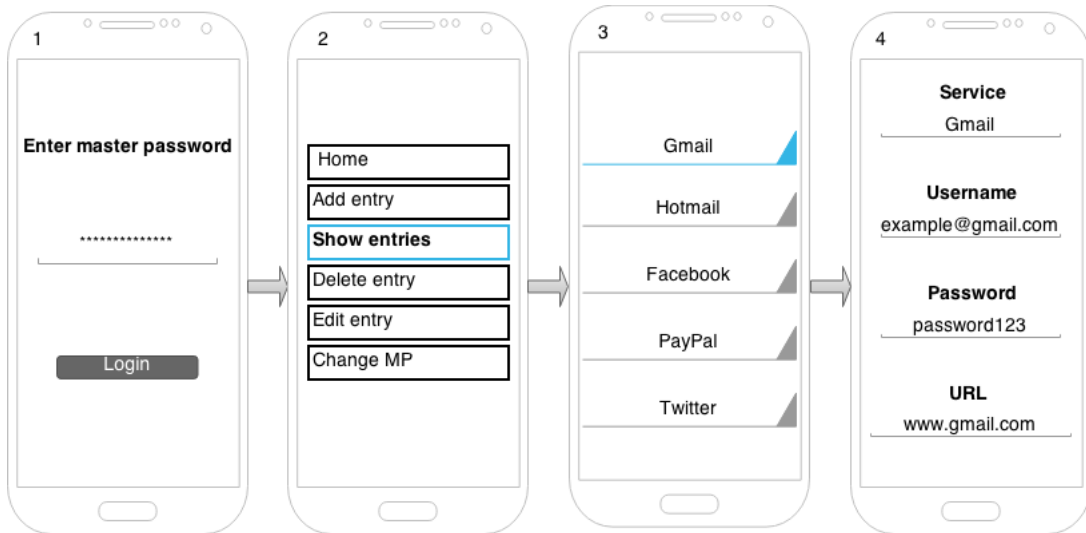


Figure 4.1: Login with master password

4.1.4 Creating passwords

A user has the possibility to choose a password that he wants, but there are some restrictions in order to make it a strong password. Since users tend to create memorable passwords [55] we are using dictionaries to create decoys based on the original password. This means that the original password must include a word from a dictionary. The requirements are the same as for the master password except that the password has to include a word from a dictionary so that we can produce similar decoys. It would not be a problem to implement a more strict password policy to make the passwords even stronger. An attacker could use this information about the requirements to his knowledge by omitting all decoys not satisfying our requirements. To solve this we restricted the production of decoys to match restrictions in the application, fulfilling password requirements.

4.1.5 Calculating the correct index

Our application is designed in such a way that no matter what master password that is tried, the application will return a password. The idea is that the attacker will never know when the correct master password is entered, since the application is always returning positive feedback. The only choice left for the attacker is to try the password given by the application. Hopefully, the attacker won't be lucky and get the correct password. This will be discussed more in chapter 7 - Security analysis. Since we are searching in a database with a lot of records/decoys, and we also want the calculated index to be within a certain range, we are using the modulo function [50]. The modulo function will help us to always return a password from the database. The equation we are using for calculating the correct index to store and find the correct password for a given entry is: $x \bmod N$, where N is the number of decoy passwords plus the correct password. To generate an index for the correct password we are using the cryptographic hash algorithm SHA-256 [40] together with the parameters master password and service acting as the key.

We do not have to index the decoys because we only care about the correct password created by the user. The final equation for locating and storing the correct password is defined as $\text{SHA-256}(\text{Master password, service}) \bmod N$. As long as the correct master password is entered the index calculated by the equation will always be the same. This ensures that the application will select the correct password for the selected service when the correct master password is entered. Since we are using with modulo we have to work with numbers. To solve this we are taking the hex value of the SHA-256 result and converting it to a decimal number. To make this more understandable let's perform an example where we explain how this works in practice. The master password is "password123", number of decoys is one hundred, and the service selected is "Gmail". Using our equation the correct password for the Gmail account is located at index $\text{SHA-256}(\text{password123, Gmail}) \bmod 101 = 89$. Let's say, for the master password an attacker is trying the password "qwerty". This will point to a password located at index $\text{SHA-256}(\text{qwerty, Gmail}) \bmod 101 = 13$. The attacker will believe that this is the correct password for the Gmail account and will definitely try this, but will find out that this is not the correct password when he is not granted access online.

Figure 4.2 shows how the design is working.

1. If the correct master password is entered the application will calculate the correct index with our equation and locate the correct password in the database.
2. If a wrong master password is entered the application will calculate an index that most likely is a decoy password. The worst-case scenario is if the input master password given by the attacker results in the same index as the correct password.

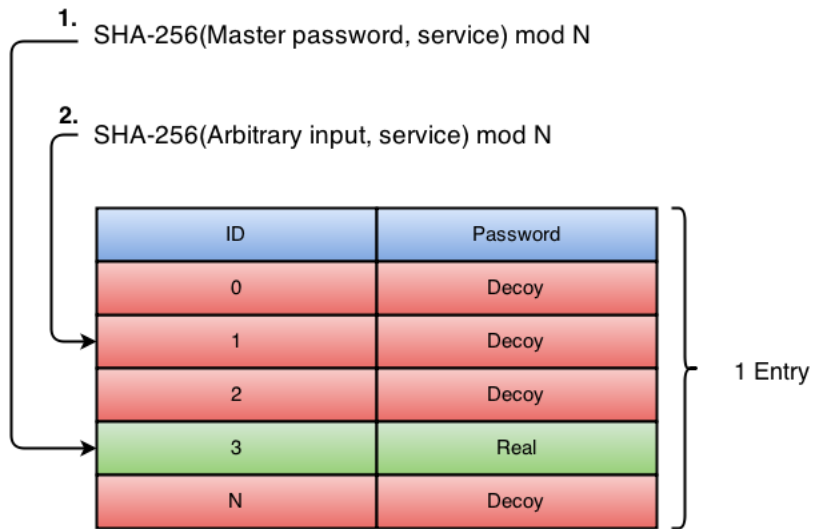


Figure 4.2: Indexing

Modular arithmetic

When we are calculating an index we want to select records from the database within a certain range, depending on the number of decoys. To accomplish this we are using a mathematical operation known as modulo [50]. If the correct password is camouflaged in 999 decoys then the amount of records in the database are 1000. The modulo will therefore be 1000, ensuring that the generated index is between 0 and 999.

Below is an example of the modulo where the remainder is 4. We do not deal with decimals in modulo operations. The modulo equation is as follows: $194 \bmod 10 = 4$

1. $194 / 10 = 19$
2. $19 * 10 = 190$
3. $194 - 190 = 4$

Regarding this example the password or decoy would be located at index 4. The modulo operator restricts the possible indexes to range from 0 to N. When an attacker tries to brute force this application in search of the correct password we want to utilize all decoys. If our index design only utilized half of the indexes the other half would not have to be there camouflaging. Therefore it is important that there is an even distribution among all of the indexes. Figure 4.3 is a print screen from one of our tests, showing that all of the indexes are hit and the number of times they are hit. The figure is simulating the modulo function where mod is 10 ranging from 0-9.

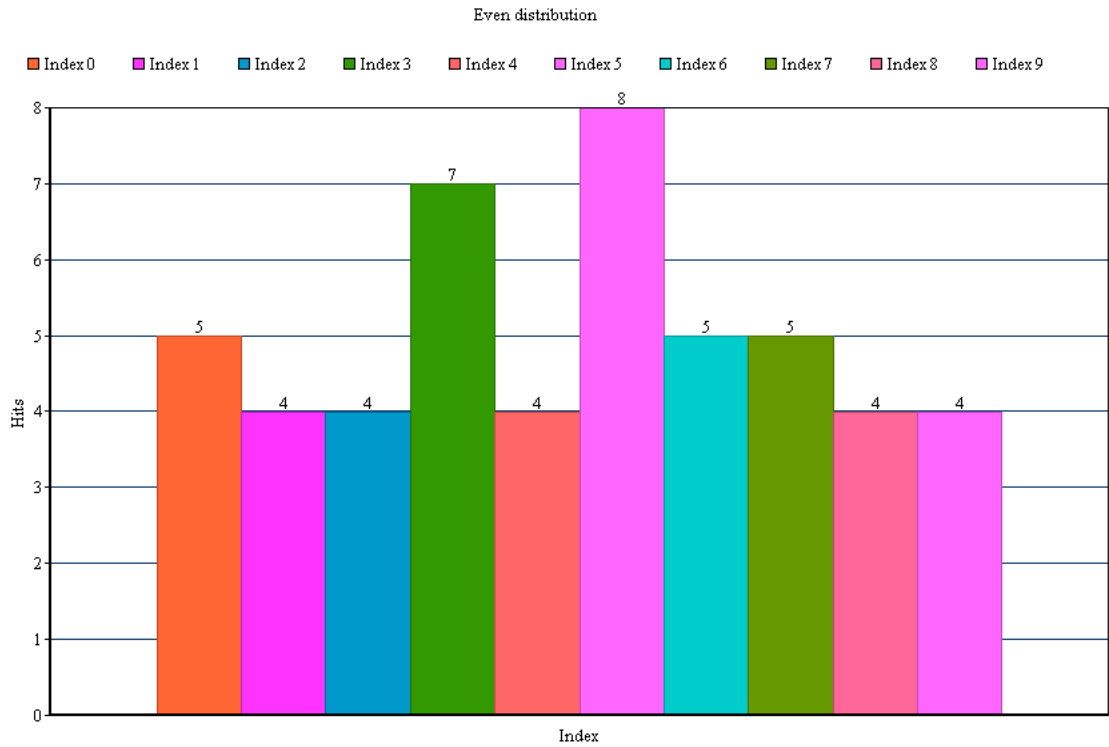


Figure 4.3: Even distribution

4.1.6 Database

Android SQLite

To provide storage for the application we used the inbuilt solution for Android which is SQLite [3]. SQLite is a light weight database which supports common database functions like add, delete, update, remove etc. The database is saved in a SQLite 3 format.

Database design

The database design is very simple and consists of one database with a table named entry. The database has two columns, id and password. The id column is of the type integer and is the primary key. This column is used in combination with the modulo operation for finding an index. The password column is of type text and is used to store passwords and decoys at the index of the column id. Each entry is added as a set, where each set is equal in size and consists of N records (correct password + decoys). An entry consists of a service, username, password and uniform resource locator (URL). As figure 4.4 shows, every set has a correct password camouflaged in N decoy passwords. The correct password is placed based on our equation of calculating the correct index. We do not have to do anything special about the insertion of records, since the id column is auto incrementing for each record. The rest of the data such as service, username and url is stored in a built in interface in Android called Shared preferences [2].

	ID	Password
Password set 1	1	Real password
	2	Decoy 1
	3	Decoy 2
	4	Decoy 3
Password set 2	5	Decoy 1
	6	Decoy 2
	7	Real password
	8	Decoy 3

Figure 4.4: Database design

4.2 User operations

Similar for the operations add, edit and change master password is that the passwords are randomly shuffled so an attacker can not find the correct password by using an image of the database before and after these operations.

4.2.1 Add

Creating an entry in our design will be dependent on whether the user has Internet connection or not. This is explained in more detail in 4.3 - Cloud design. When a user creates a new entry the application will send the selected password to the cloud. The cloud will return a set with decoys including the correct password back to the application. The original password together with the decoys are inserted into the database.

4.2.2 Read/Show

To show the correct information when a user chooses an entry from the list of entries the application has to locate the right password set which is solved by storing a counter for each entry. The application will calculate the correct index based on our equation $\text{SHA-256}(\text{Master password, service}) \bmod N$ for the correct index.

4.2.3 Update/Edit

When a user wants to update or edit his password the application will perform the same operation as creating a new entry. The new password specified by the user will be added to the database with new generated decoys. The correct set to edit is located by the counter stored uniquely for each entry.

4.2.4 Delete

Deleting an entry will delete everything such as service, username, url and password including decoys and the correct password from the database. The correct set to delete is located by the counter stored uniquely for each entry.

4.2.5 Change master password

When a user wants to change master password the application will do the following for each entry:

1. Select all passwords in each set
2. Locate the correct password by calculating the correct index with the current master password
3. Calculate the new index based on the new master password
4. Randomly shuffle the password list
5. Locate the correct password and change its place with the decoy that is located at the index generated from the new master password

By only changing the correct password with the decoy password located at the new index, an attacker could have easily found the correct password if he had a before and after image of the database, since these two passwords would be the only one changing index. Therefore we have to shuffle the list so that all passwords change index. It is a possibility that some decoys have the same index after the shuffle, but that is only positive, since the new index for the correct password could also have been generated to the same index as the old. Making it harder for an attacker to locate the correct passwords.

4.3 Cloud design

4.3.1 Production of decoys

By implementing a cloud environment, we have the possibility to move functionality from the mobile device, and up into the cloud. By doing this we prevent an attacker from getting access to source code that could weaken overall security regarding the production of decoys. If an attacker is able to analyze the production of decoys, this could ultimately help the attacker in separating correct passwords from decoy passwords. The solution Internet connection / Awaiting Internet connection is not implemented in our solution as it is today. Now the application requires Internet connection. Figure 4.5 shows the general design of how the decoys are produced with and without Internet connectivity.

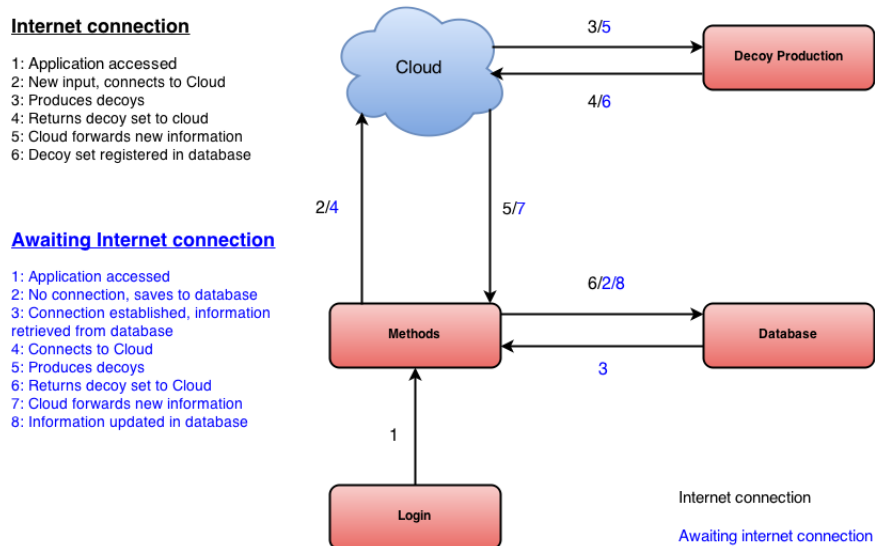


Figure 4.5: Decoy production overview

There are two main reasons for producing decoys in a cloud.

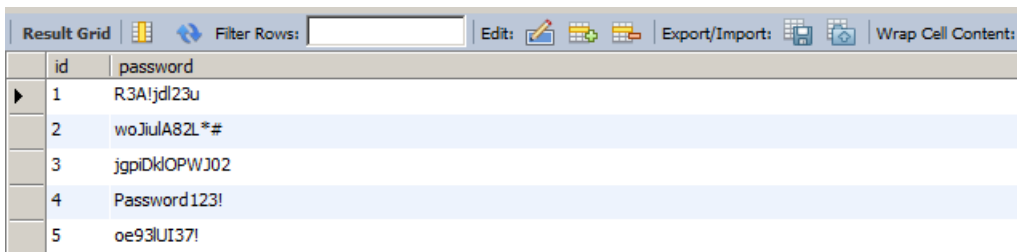
First point: The process is not revealed in the source code, which an attacker could use to his advantage

Second point: Even though mobile phones have become more powerful over the years, the amount of resources available on a server is much larger and expandable.

Generating strong passwords has never been a hard task, but it is proven that people tend to use passwords they have created themselves, built up of words. [49, 27, 55]

We wish to implement the use of decoy passwords that look like they are produced by a human, not a machine.

Creating random passwords is not difficult, but producing passwords that first of all look like they have been produced by a human, and second of all help camouflage the users original password, is not an easy task. The reason for this is human produced passwords are unlikely random as users tend to create easy memorable passwords. [49, 27, 55]. If the decoys produced were random, it would be clear to an attacker what password was produced by a human. (See figure 4.6)



id	password
1	R3A!jdl23u
2	woJiulA82L*#
3	jgpiDkiOPWJ02
4	Password123!
5	oe93lUI37!

Figure 4.6: Original password: Password123!

Requirements:

- All passwords have to be unique
- All passwords include one or more words from one or multiple dictionaries
- The original password has to be checked against already existing passwords and produced decoys
- Generated decoys have to be checked against all passwords, both in the database and in the same session of generation
- The original password received from the device is shuffled with generated decoys before sending them all back to the device
- The amount of time should be as short as possible with a production of 10 000 decoys. A feasible amount of time would be maximum 5 seconds

Internet connection

The application only requires Internet connection when the user wants to create a new entry. Since the production of decoys is happening in the cloud the mobile phone has to have Internet connection in order to request and retrieve decoys.

No Internet connection (not implemented)

When the user wants to create a new entry and the application does not have Internet connection, it will store the correct password temporarily. Once the application detects that it has Internet connection it will carry out the same tasks as if it had Internet connection in order to produce decoys for the newly created entry.

4.3.2 Database

To provide storage in the cloud environment we are using a MySQL database from the software MAMP [11]. The purpose of this database is not the same as the one stored on the android device. This database is primarily used to correlate newly produced decoys with old ones ensuring that we are not creating equal decoys for different sets. Worst case scenario would be if each set had the same decoy passwords camouflaging the correct password. It would have been easy for an attacker to detect the correct passwords since they would be the only ones differing when looking at all the sets in context.

Database design

The database design in the cloud consists of a single table with three columns. The first column is named "ID" and acts as the primary key. The second column is named "Hash", this is a unique identifier for each password, both decoys and correct passwords. This ensures unique passwords / decoys avoiding duplicates.

The third column is named password and is used to store the passwords. It is important to mention here that the column "Password" is implemented in our design for testing purposes, and it is not necessary in a real implementation. By only storing the unique hashes in the database, the cloud is able to check uniqueness of new productions, and privacy is obtained in relation to users passwords.

Figure 4.7 shows a figure of the database design. The column "Hash" in the figure is only showing a part of an SHA-256 value. Including the whole value would make the figure go out of range and therefore we have shortened it.

ID	Hash	Password
1	e3b0c44298fc1c	Real password
2	4c8996fb92427a	Decoy 1
3	49b934ca49599f	Decoy 2
4	01f6a3bc01c12d	Decoy 3
5	04972fe71088gd	Decoy 4
6	52b49b6cbf5d33	Decoy 5
7	035f96ab343c70	Real password
8	eee3b0dbc2632	Decoy n

Figure 4.7: Database design in the cloud

4.4 Connection between android application and the cloud

An Apache web server is used to simulate a cloud in our solution. The Apache server is running a PHP-script [16] that is responsible for processing incoming

data from the android application and performs tasks based on the input it is receiving. The Apache web server is communicating with a MySQL database using the same PHP-script [16].

Chapter 5

Proposed Solution

Figure 5.1 shows the general design of our prototype. The prototype consists of two main elements, the Android application and the cloud environment. The Android application consists of the application itself and a SQLite database to store its data. The cloud environment consists of a local Apache web server that is simulating the cloud and a MySQL database to store its data. To establish a connection between the Android application and the cloud we used the scripting language PHP.

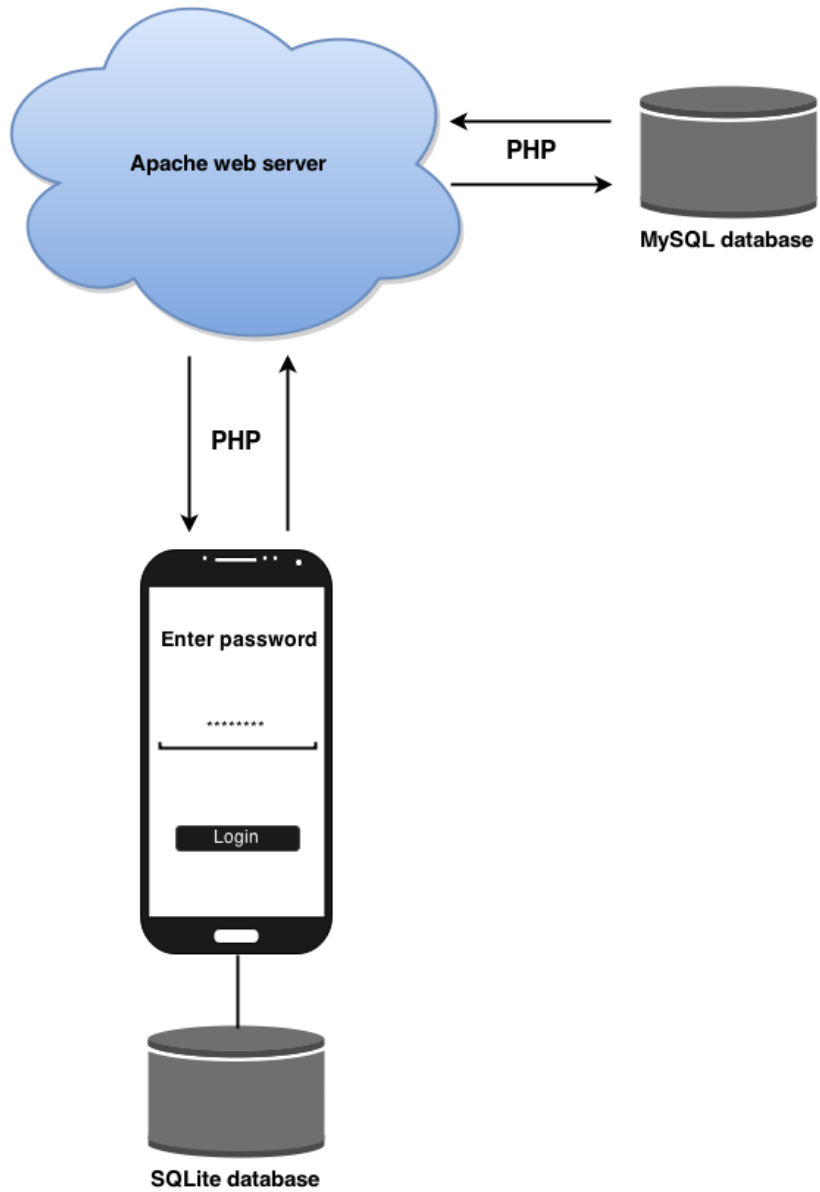


Figure 5.1: General overview of our prototype

5.1 Software used

The programming language we have worked mostly with is Java and therefore it was an easy choice when it came to choosing a programming language. With Java as our preferable choice, we knew that Android's programming language is Java. Android also have a new great software called "Android Studio" [4] with a lot of development tools. We have worked with all of these tools earlier in a previous course, but now they are all implemented in "Android Studio", which makes it easy to use and more productive when creating an application. In order to simulate a cloud we used a software called MAMP [11]. MAMP contains software such as an Apache web server and a MySQL database. With these tools we only had to use PHP on the Apache server, being the connection between the application and the cloud.

5.2 Random functions

In the choice of random implementation, an equal distribution as well as efficiency was highly important. In PHP the default random generator is based on "rand()", which produces good results in small scenarios. However, in our case we noticed that it did not produce sufficient distribution as patterns started to occur. We implemented the use of "mt_rand()", an improved random generator based on the original Mersenne Twister (a large linear-feedback shift register), producing statistically good distribution, as well as being 4 times more efficient than the original "rand()" function in PHP [15, 12].

5.3 Android development

5.3.1 Tools used

As mentioned Android studio contains a lot of tools that can be used to analyze Android applications [5]. This helped us a lot when developing the application. The tools we used were mostly from the Android development kit:

- **Android Device Monitor** This tool makes it possible to view the file hierarchy of the mobile phone and pull or upload necessary files. We used this mostly to pull the database and shared preferences file to view the data as we added, updated or deleted data. The tool was also used to take screenshots of the application in action, to show what it looks like and works.
- **Android Virtual Device Manager** This tool makes it possible to create a mobile emulator that is identical to a physical device.
- **Database Browser for SQLite** This tool helped us to open and read the database file we pulled from the mobile phone using the device monitor tool.

5.4 Hexstring to decimal

Calculating the index by our index equation cannot be done directly. Since we are dealing with mathematical operations and the result has to be a number we must convert the SHA-256 output to a number. The code snippet shown below shows how we are taking a string as input parameter. The input string is the master password + service in the function SHA-256 (Master password, service). The result is a unique SHA-256 value represented as a hex value. To use the modulo function, this hex value has to be represented as a number. Therefore we are converting the hex value to a unique number.

CHAPTER 5. PROPOSED SOLUTION

```
public static BigInteger sha256HexToDecimal(String base) {
    MessageDigest mda;
    BigInteger bi = null;
    try {
        mda = MessageDigest.getInstance("SHA-256");
        mda.reset();
        //important part
        byte[] array = mda.digest(base.getBytes());
        StringBuffer sb = new StringBuffer();
        for (byte b : array) {
            sb.append(String.format("%02X", b)); // print
            the byte as a 0 padded, two digit,
            hexadecimal String
        }
        bi = new BigInteger(sb.toString(), 16);
    } catch (NoSuchAlgorithmException e) {
        e.printStackTrace();
    }
    return bi;
}
```

Figure 5.2 shows a print screen from the output of our function mentioned above. The input master password is "Password123%" and the service is "Gmail". These strings are concatenated and run through the SHA-256 algorithm. The result will be a unique hex value that is transformed to a number. Using online tools we have confirmed that the SHA-256 value shown in figure 5.2 is correct based on the input and also that the number value of the SHA-256 result is correct.

Hex value of Password123%Gmail is: 98707C6FB22412F3D5F039F8BE11F69FA9AF64EE2ABEF7EED033F1EDCDE6C042
Decimal number: 68950298683773180074774603160307782585575303721222097075669461515168345669698

Figure 5.2: Code snippet of SHA-256 hex value converted to decimal

5.5 Cloud - production of decoys

When a user adds a service on the Android device the cloud is prompted with the new password. This should initially be sent over by SSL to keep attackers from sniffing the traffic (we simply use JSON communication as it suffices for our concept testing). The password is received by the cloud in form of a JSON object (other options would off course be possible) and analyzed. The mobile device would have restrictions for what passwords are acceptable in relation to length and restrictions. The analysis in the cloud would be to check how long the password is, how many numbers are used, how many uppercase letters are used and how many special characters are used. This is for the production of decoys. It is highly important that the decoys produced are similar. This means that the dictionaries used must be in the same language as the original password, and that the dictionary also includes names.

In the cloud, dictionaries are imported and 'cleaned' so certain characters and encoding won't trouble functions in relation to encoding, one example being insert procedures to the database. After this, the checks begin in producing unique decoys.

In the last chapter it is mentioned that multiple checks have to be carried out to complete a successful decoy set. In the production of large amount of decoys, an example being 10 000, the amount of iterations needed highly impact the time.

- Check original password against database
- Check each decoy password against database and within production itself

We implemented a manual form of associative arrays in PHP (Lookup table, HashMap). By producing a unique value from the decoy generated, the decoy passwords were indexed with this unique value. In the event of checking if a decoy is produced earlier, all that is needed is a single lookup on the index.

CHAPTER 5. PROPOSED SOLUTION

```
$DECOYLIST = array();
$Password = "pasSword";
//Variable $Password get's value "pasSword"

$PasswordSHA = hash('sha256', $Password);
/*Unique value is produced with SHA256

$PasswordSHA value =
    '218c913efad5c7a174cb30255ec63ad68c3692b51ea77fe940c17624a198a12f'
*/

$DECOYLIST[$PasswordSHA] = $Password;
//Password is inserted into array with unique index

var_dump($DECOYLIST);

/*
Dump of array values :
array(1) {
    ["218c913efad5c7a174cb30255ec63ad68c3692b51ea77fe940c17624a198a12f"]=>
    string(8) "pasSword" }
*/
```

CHAPTER 5. PROPOSED SOLUTION

By doing this, after each decoy creation, a single check can check the uniqueness before inserted into the decoy table.

```
if (!$DECOYLIST[$PasswordSHA]) {  
    $collision = false;  
} else {  
    $collision = true;  
}
```

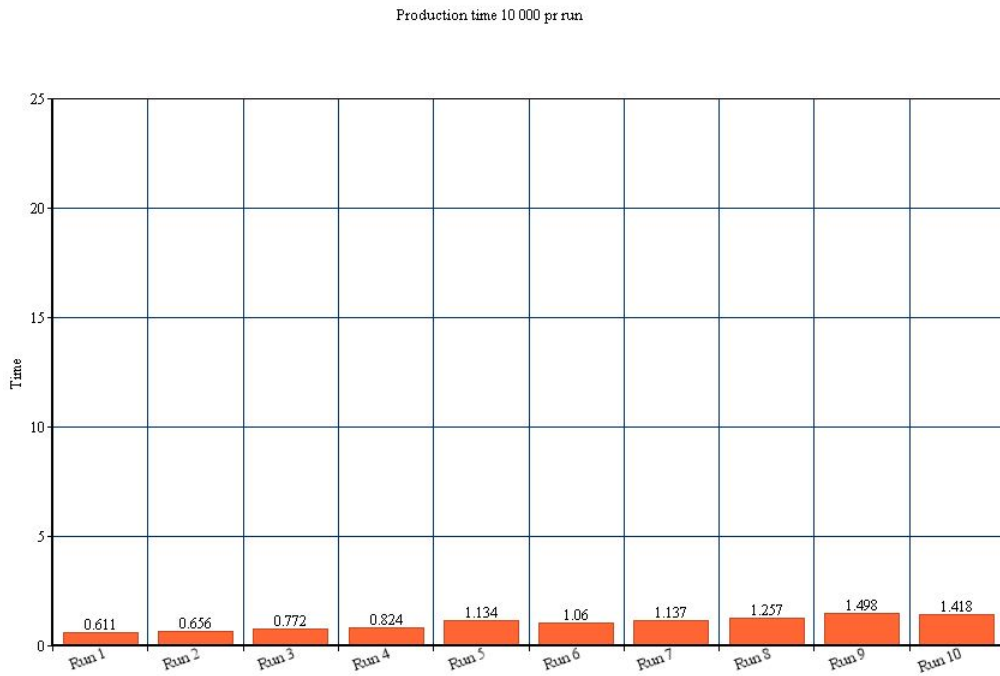


Figure 5.3: Check with 0-100 000 decoys in the database

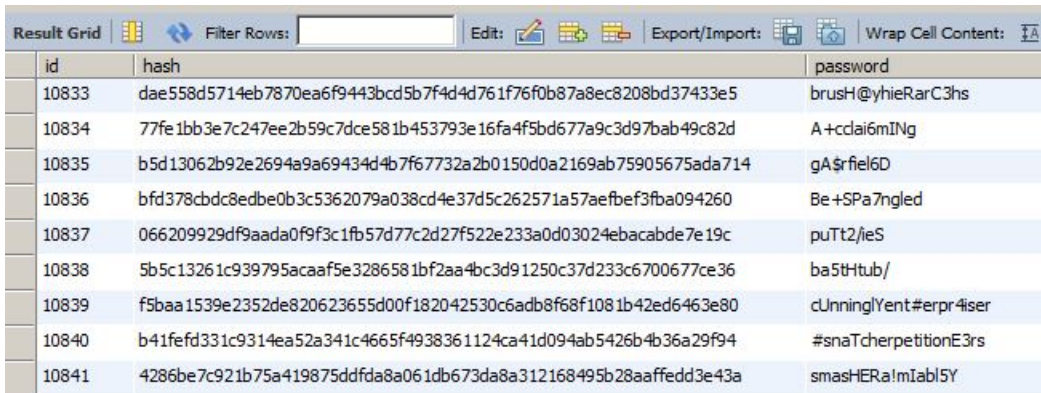
CHAPTER 5. PROPOSED SOLUTION

As shown in the figure, there is little to no increase of time used in relation to how many decoys there are in the database.

$$\frac{1}{10} (0.611 + 0.656 + 0.772 + 0.824 + 1.134 + 1.06 + 1.137 + 1.257 + 1.498 + 1.418)$$

Figure 5.4: Average over 10 runs: 1,036 Sec

Figure showing database in cloud with unique keys and produced decoy passwords. (Hash = unique ID, password = the passwords)



id	hash	password
10833	dae558d5714eb7870ea6f9443bcd5b7f4d4d761f76f0b87a8ec8208bd37433e5	brushH@yhieRarC3hs
10834	77fe1bb3e7c247ee2b59c7dce581b453793e16fa4f5bd677a9c3d97bab49c82d	A+cclai6mING
10835	b5d13062b92e2694a9a69434d4b7f67732a2b0150d0a2169ab75905675ada714	gA\$rfiel6D
10836	bfd378cbdc8edbe0b3c5362079a038cd4e37d5c262571a57aefbef3fba094260	Be+SPa7ngled
10837	066209929df9aada0f9f3c1fb57d77c2d27f522e233a0d03024ebacabde7e19c	puTt2/ieS
10838	5b5c13261c939795aacaaf5e3286581bf2aa4bc3d91250c37d233c6700677ce36	ba5Htub/
10839	f5baa1539e2352de820623655d00f182042530c6adb8f68f1081b42ed6463e80	cJnningYent#erpr4iser
10840	b41fefd331c9314ea52a341c4665f4938361124ca41d094ab5426b4b36a29f94	#snaTcherpetitionE3rs
10841	4286be7c921b75a419875ddfda8a061db673da8a312168495b28aaffedd3e43a	smasHERa!mIabl5Y

Figure 5.5: Screenshot from the database in the cloud

When a user inputs a password it is important that it is not a random sequence, but that it includes a minimum of one word, with minimum 1 uppercase letter, 1 lowercase letter, 1 special character, and some numbers. This password is sent to the cloud, decoys are produced and sent back to the device as a shuffled array. The device remembers in temporary memory what the original password is in the array. (Simplified passwords shown in figure 5.6)

```
["aMm0n!a", "Th3U&gGish", "di#SconN3ectEd", "con#9stIpAtes", "rE3la+bel", "libe&liS1t"]
```

Figure 5.6: Passwords being sent in JSON-format

5.6 Decoy generation

To start out with, the cloud analyzes the original password, and makes a choice of words to use from a dictionary. It is randomly chosen what words are to be used together, with a choice between one and two words. Words can be used multiple times in different decoys, allowing a larger amount of decoy combinations in comparison to removing a word from the dictionary after first use.

When choosing the amount of uppercase letters, our solution does the following to conceal original structure, but equally achieves a sufficient amount of variance:

Choosing the amount of uppercase letters is done randomly. The process is done by choosing 1 of 3 choices for each decoy produced.

Definition: Amount = Amount of characters in generated decoy password.

- Choice 1: Random between $(1, ((\text{amount} - 1) / 3))$
- Choice 2: Random between $(1, ((\text{amount} - 1) / 2))$
- Choice 3: Random between $((\text{amount} / 2), (\text{amount} - 1))$

There is a $5/8$ chance that choice 1 is selected, $2/8$ chance that choice 2 is selected and a $1/8$ chance the last choice is selected, each choice in increasing order having a larger probability of producing larger amounts of uppercase letters.

As passwords commonly contain more lowercase letters than uppercase, there is a larger probability that choice 1 is used, and least likely that choice 3 is used.[49] The reason for this is that passwords that include a large amount of uppercase letters look computerized, but in a scenario where a user actually does implement many uppercase letters, it is highly important that some decoys produced also have this structure.

Another common pattern that occurs in password creation by users is that it is more likely that users implement an uppercase letter in the beginning of words, rather in the middle [49]. Even though this is true, if all decoys were to begin with uppercase letters, if a user did the opposite, the original password would be the only one starting with lowercase. As the amount of decoys produced per run are set to a minimum of 10 000, the choice of uppercase letters are done randomly, thereby all combinations will occur. In the situation of inserting numbers and special characters, the amount is chosen between 1 and (amount in original password), the reason being that a large amount gives a strong impression of the decoy being computerized.

Numerals are chosen from 0-9, and special characters are chosen from !, #, %, &, /, =, +, @, \$, .

5.7 Discussion of design issues

5.7.1 Login

When logging into the application it is important that the user uses the same password every time. Using different passwords will cause problems for the user, since the operations in the application are dependent on the master password. Also, when the user installs the application for the first time, a master password has to be created. After this is done, the user can still login with another master password, since the application will allow any master password input. The user is not forced to use the master password that was created upon installation.

Add entry

When the application is launched it requires a master password. Accidentally entering a wrong master password can cause problems for the user. If an entry is added with a different master password rather than the intended one, issues will occur the next time the user is entering the correct master password. The wrong master password that was used to create the entry will result in an index most likely differing from the correct master password and the application will therefore return a wrong password when the correct master password is used. The user can be lucky and hit the correct index if the wrong master password and the correct master password is calculated to the same index.

Change master password

If a user wants to change the master password some parameters are required. The required parameters are the current master password, new master password and a confirmation of the new master password. The current master password is used to locate the correct password from each entry. The new master password is used to

place the correct password at the correct index based on the result of the index calculation. If the user accidentally enters a wrong master password in the "current password" field the application will locate wrong passwords from the different sets. In worst case, this can cause the user to "lose" all of his passwords.

Also, if the current master password and the new master password results in the same index, then the real password won't change its place in the password set for that entry, while the decoys will. This can be a problem if an attacker has a before and after image of the database. The probability of this will decrease by using more decoys, since the space will be larger. The decoys are also randomly shuffled so the probability of only the correct password being on the same place after this operation is very small, as they also can have the same index before and after this operation.

Edit entry

Entering an incorrect master password and editing an entry will also cause problems. The index of the correct password will be dependent on the master password that was used. When the user enters the correct master password the next time, he will most likely calculate another index and therefore retrieve an incorrect password.

5.7.2 Internet / No Internet connection

If a user does not have Internet connection when creating an entry the application saves the record and awaits a successful connection before receiving decoys. This is not implemented into our solution, but it is a weak point as the password is stored without decoys.

This results in the entry in question not being protected by decoys in this pe-

riod of time (until connection is established), but as mobile devices are connected to a networks most of the time, we categorize this as a minimal threat.

5.7.3 Using modulo arithmetic for indexing

Since we are using modulo in our design when indexing there will always be collisions. This problem is known as the "Pigeonhole principle" [18]. In our case we have n possible numbers and only k decoys. Since n in our case is greater than k , collisions will occur. Our number n is dependent on the SHA-256 algorithm which gives 2^{256} unique values. The number 2^{256} is a much larger number than the number of decoys we are using. Therefore, some of the hash values will map to the same index.

The disadvantage of using modulo for indexing is that the attacker can hit the correct password even if the master password is incorrect. It does not matter what the input master password is, if the result of the modulo calculation results in the same index as the correct master password, then the attacker will retrieve the correct password for the selected entry. But, it does not necessarily mean that the same master password will locate the correct password of another entry.

Chapter 6

Performance testing

Tests regarding the application were done on an android device with the following specifications:

- **Name:** Android Galaxy S3
- **Model number:** GT-I9300
- **Android version:** 4.3
- **Internal storage:** 16 gigabytes

Tests regarding decoy production were done on a computer with the following specifications:

- **Processor:** Intel Core i7-3610QM CPU 2.30 GHz
- **RAM:** 12 Gigabyte
- **Operating system:** 64-bit
- **System type:** Windows

6.1 Android application performance

6.1.1 How much MB space do the decoys produce?

When the application is first installed on the mobile device its total size is 3,35 megabytes, where the application is 3,34 MB and additional data is 12 kilobytes. After adding ten entries where each entry had ten thousand decoys the size of the application increased to 5,82 MB in total. The additional data stored in the application after adding these ten entries increased to 2,49 MB. These results show that space is not a problem regarding the amount of decoys that is needed. It is normal that a user has between ten and twenty services. With these results, space would not be a problem. Even increasing the number of decoys would also be possible.

6.1.2 Do large amounts of decoys slow down the application?

The below results in figure 6.1 show that ten thousand decoys are not a problem regarding speed in the different operations. These operations can be heavy for the mobile device and are therefore done in a separate thread. This will ensure that the main thread is not overloaded and that the user experience is not slow and lagging. Below is a diagram showing the number of seconds each operation takes. This is seconds after the application has retrieved the requested information from the cloud. These tests were done when the application was loaded with ten entries having ten thousand decoys each. All operations completed in one second, except when we changed the master password. This is a much heavier task than the others, since it has to change every index of each entry. Changing master password is not something that is done that often so this result is acceptable. It is possible that the test results are better but we are only testing in seconds and not in a smaller scale.

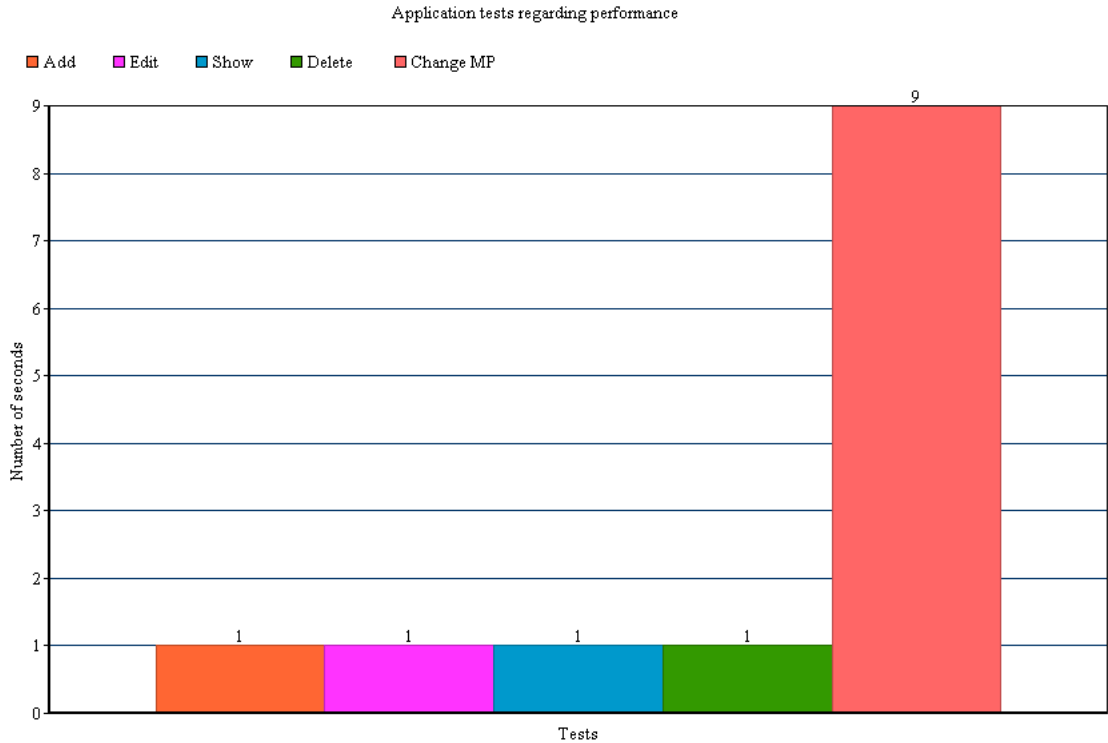


Figure 6.1: Application performance

6.1.3 Probability of hitting the correct index

The security of every real password is dependent on how many decoys are used to camouflage it. Below is statistics to show how many times an attacker is able to hit the correct index where the correct password is stored. We had to create a custom test to be able to simulate an attacker trying various master passwords attempting to find the correct password. We are generating unique random master passwords, whereas an attacker would probably use other methods of attacking the application. The results below are only to show the statistics of whether an attacker hits the correct password, depending on how many decoy passwords that

are being used to camouflage the correct password. The biggest chance to hit the correct password is if the modulo calculation based on the master password input results in the same index as the correct password, rather than guessing the correct master password. Therefore it is important to increase the number of decoys to decrease this probability. The test was run 10 times where each run generated 10000 unique master passwords. Below is an explanation of the test that is used to compute the results in our graphs. For this test we used "Gmail" as the service when calculating an index.

1. The real users master password is initialized and the index of the correct password for the service "Gmail" is calculated, hence the index is $\text{SHA-256}(\text{Master password, Gmail}) \bmod N$ (Number of decoys plus the correct password).
2. Two tables are created. The first table is used to add the value of the index calculation to the appropriate index in the table, so we can count how many times the correct index is hit. If we are testing the hit rate for 1000 decoys, then the table size is ranging from 0-1000. The second table is only used to check whether the generated password is generated before. An attacker would not try the same password twice, therefore we had to check that the passwords were unique before trying them. Although, in our scheme it could be that the master password hits the correct index for one entry but not another. This is because the value changes when the master password is concatenated with another service.
3. A loop is run 10000 times, where a unique master password is generated for each iteration. To generate a random password we are using the secure random function in Java [8]. This function picks out random characters from a defined string we have defined, which contains all alphabetical letters, both upper and lower case, numbers ranging from 0-9 and some special characters. The length of each password generated is of length 8. The master

password plus the service "Gmail" is used to calculate the index of the correct password. If the calculated index is the same as the index of the users, then the attacker hit the right password for that entry. It does not necessarily mean that he hit the correct master password, but the modulo calculation resulted in the same index as the correct password. Nor is it certain that this master password would generate the correct index of another service, since the hash value may be different when concatenating the master password with a different service.

6.1.4 Hit rate with 1000 decoys

Figure 6.2 shows the outcome of our test with 1000 decoys. The highest number of occurrences where the "attacker" hit the correct index was fifteen times in one run and that was in the fourth run. The third run was the one that hit the correct index first and that was on attempt 15. The eighth run was the one with the most attempts before hitting the correct index the first time. It did not hit until 3491 attempts.

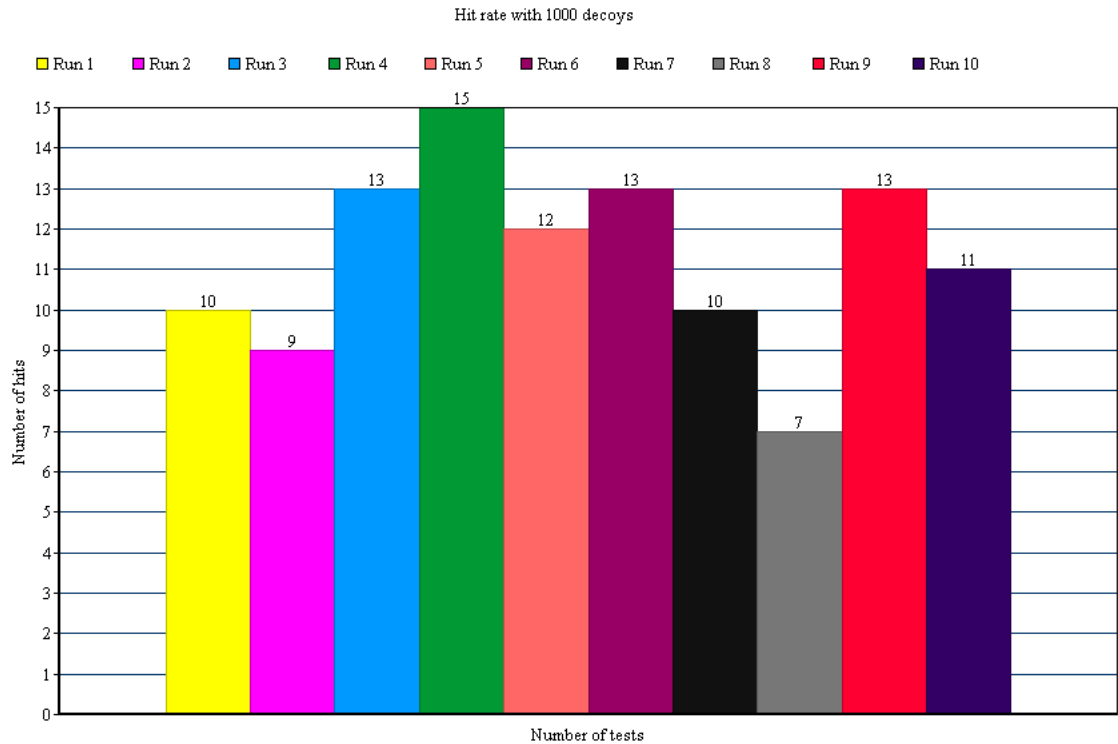


Figure 6.2: Hit rate with 1000 decoys

6.1.5 Hit rate with 5000 decoys

Figure 6.3 shows the outcome of our test with 5000 decoys. The highest number of occurrences where the "attacker" hit the correct index was three times in one run and that was in the sixth and seventh run. The seventh run was the one that hit the correct index first and that was on attempt 1004. The first run was the one with the most attempts before hitting the correct index the first time. It did not hit until 4625 attempts.

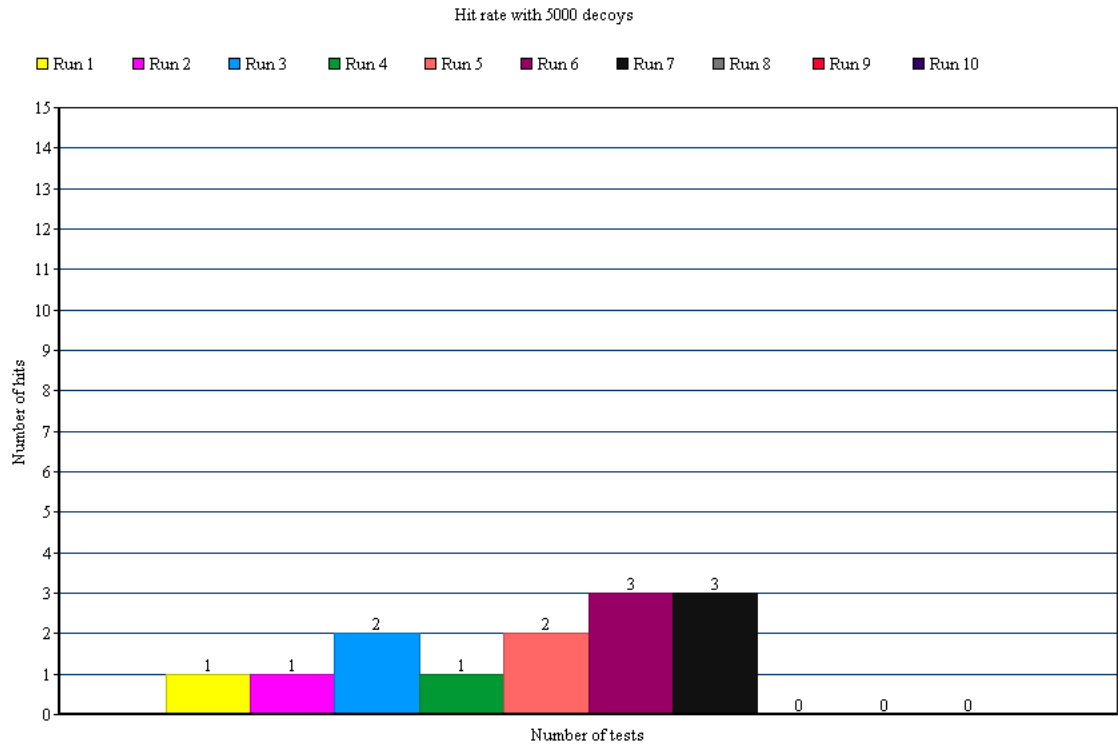


Figure 6.3: Hit rate with 5000 decoys

6.1.6 Hit rate with 10000 decoys

Figure 6.4 shows the outcome of our test with 10000 decoys. The highest number of occurrences where the "attacker" hit the correct index was four times in one run and that was in the eighth run. The eighth run was the one that hit the correct index first and that was on attempt 3410. The sixth run was the one with the most attempts before hitting the correct index the first time. It did not hit until 8137 attempts.

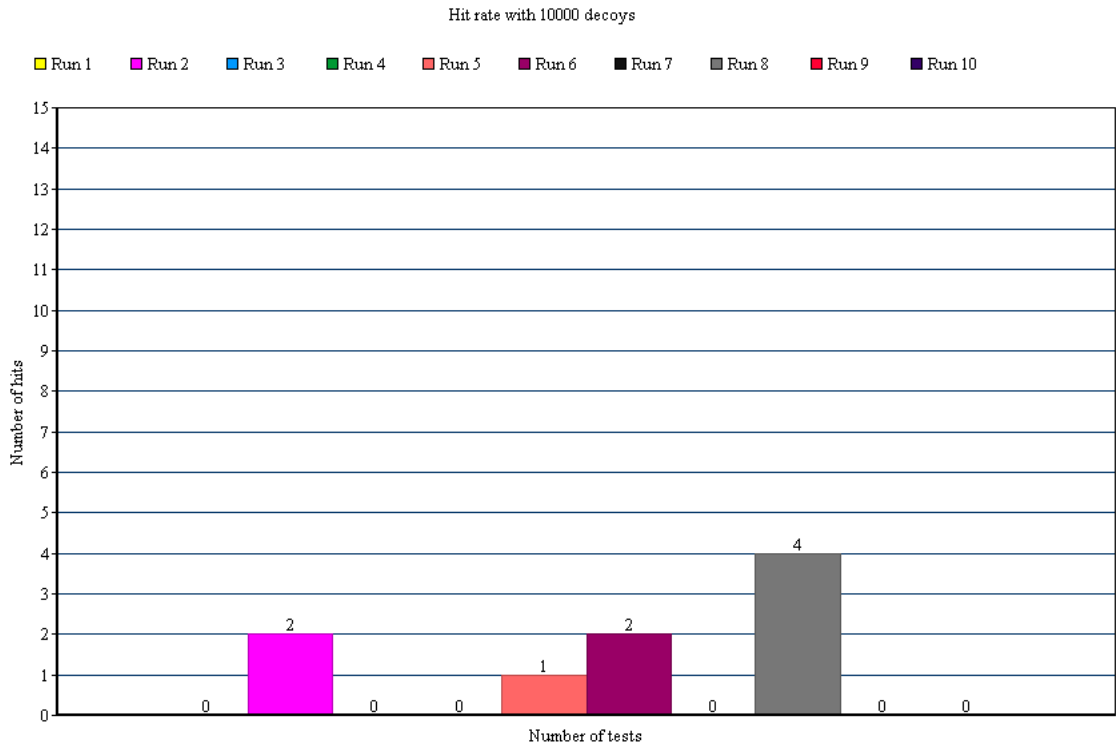


Figure 6.4: Hit rate with 10000 decoys

CHAPTER 6. PERFORMANCE TESTING

Analyzing the data in the graphs shows that there is a clear difference between the number of decoys. This is logical since an increase of decoys will give a smaller probability of hitting the correct index. However, this does not exclude the possibility that the number of hits is higher when the number of decoys is 10000 versus 1000.

6.2 Cloud performance

In the process of developing a system that has to evaluate such a large number of decoys, a standard checking mechanism with loops would work, but demand an increasingly large amount of resources. This would also lead to time increasing in relation to the amount decoys produced. Having to check a database of 200 000 decoys would take much longer time than a database with 1000, resulting in the application becoming slower and slower as more decoys are produced.

PHP: String compare for each decoy produced: Returns 0 if equal:

```
for($s=0; $s<count($DECOYLIST); $s++){  
    if(strcmp($generatedPassword,$DECOYLIST[$s])==0){  
        $collision = true;  
    }//if  
} // For-loop
```

We tested this to analyse expectations on time used. By implementing string comparison checks in PHP with the use of arrays, we concluded with the following times.

Without storing decoys to the database (Strictly production)

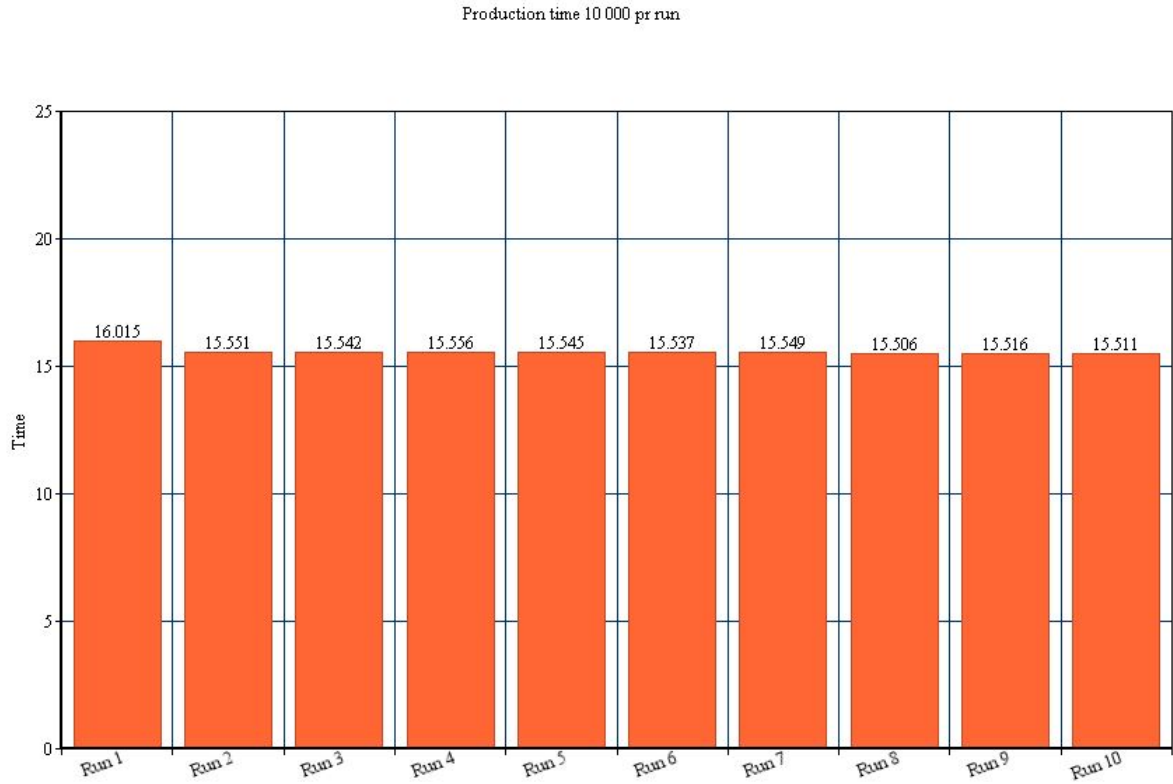


Figure 6.5: Decoy production 10000 - 10 times

By calculating the average we can conclude that the production of 10 000 decoys without saving to the database would take over 15 seconds (15.5828 sec) with the use of iterations and string comparison.

$$\frac{1}{10} (16.015 + 15.551 + 15.542 + 15.556 + 15.545 + 15.537 + 15.549 + 15.506 + 15.516 + 15.511)$$

Figure 6.6: Decoy production: 15.5828 sec

CHAPTER 6. PERFORMANCE TESTING

We improved methods and loops used on the server to make the production more efficient, one example being the implementation of 'for each' loops, decreasing time used by 38.5 %. $((15.5828 - 9.5759) / 15.5828) = 38.5 \%$

$$\frac{1}{10} (9.59 + 9.602 + 9.523 + 9.594 + 9.604 + 9.581 + 9.602 + 9.533 + 9.579 + 9.551)$$

Figure 6.7: Improved time Decoy production: 9.575 sec

Even though this increases efficiency, it is apparent that the resources needed to produce decoys in a reasonable amount of time are large.

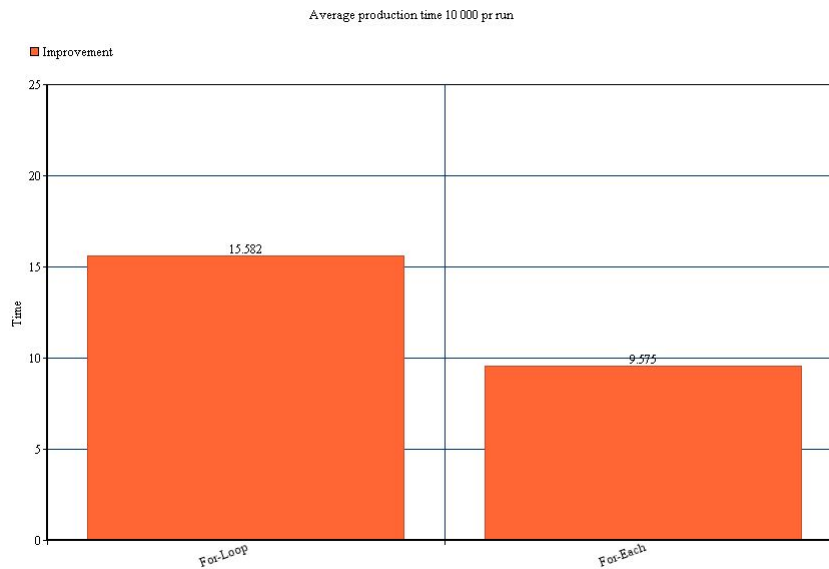


Figure 6.8: Improvement: 6 sec

CHAPTER 6. PERFORMANCE TESTING

The checks to validate if a decoy has been produced earlier are efficient, but they are restricted in relation to how many decoys that have to be compared (an increase of decoys result in an increase of checks needed). Having to check every single value in the database, and doing this in the production itself is highly time consuming. An analysis starting initially with an empty database through the production of 50000 decoys total (10 000 x 5 stored) show the increase of time needed.

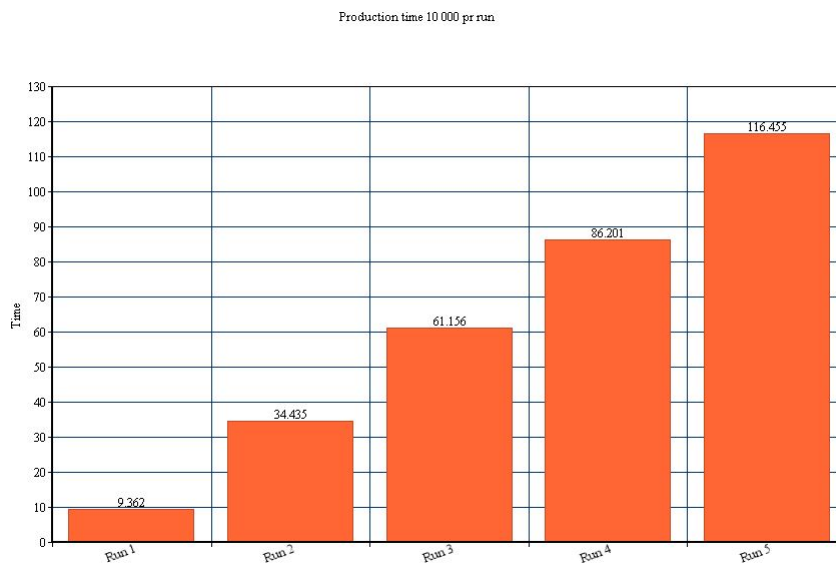


Figure 6.9: Production of 10 000 x 5 (Time in seconds)

If a standard iterative design like this was to be implemented, a production of 5 decoys would result in 15 iterations as every new decoy has to be checked against already existing decoys. There would be a check after 1 decoy is produced, 2 checks after the second decoy is produced, and 3 checks after the third, and so on. A simplified version of decoy production is shown in the example under to illustrate how (no restrictions), original password is shuffled with decoys in the result.

CHAPTER 6. PERFORMANCE TESTING

Original password: paSsword

Iteration: 1 Generated Password= volTs - check against nr: 1 : paSsword

Iteration: 2 Generated Password= undisplayeD - check against nr: 1 : paSsword
Iteration: 3 Generated Password= undisplayeD - check against nr: 2 : volTs

Iteration: 4 Generated Password= tRepid - check against nr: 1 : paSsword
Iteration: 5 Generated Password= tRepid - check against nr: 2 : volTs
Iteration: 6 Generated Password= tRepid - check against nr: 3 : undisplayeD

Iteration: 7 Generated Password= kendrA - check against nr: 1 : paSsword
Iteration: 8 Generated Password= kendrA - check against nr: 2 : volTs
Iteration: 9 Generated Password= kendrA - check against nr: 3 : undisplayeD
Iteration: 10 Generated Password= kendrA - check against nr: 4 : tRepid

Iteration: 11 Generated Password= Intercapillary - check against nr: 1 : paSsword
Iteration: 12 Generated Password= Intercapillary - check against nr: 2 : volTs
Iteration: 13 Generated Password= Intercapillary - check against nr: 3 : undisplayeD
Iteration: 14 Generated Password= Intercapillary - check against nr: 4 : tRepid
Iteration: 15 Generated Password= Intercapillary - check against nr: 5 : kendrA
Amount of iterations 15

All passwords: ["kendrA","paSsword","undisplayeD","Intercapillary","volTs","tRepid"]

Figure 6.10: Iterations

Production of 10 decoys would lead to 55 iterations, 100 decoys would result in 5050 iterations, and 1000 decoys would result in 500500 and so on. This is calculated without the situation of collisions. If collisions were to happen, the number of iterations would increase. The solution to avoiding iterations is to implement unique identifiers for decoys. By doing this, string comparison iterations can be replaced by single look-ups, decreasing overall time by 93%. ((15.5826 -

CHAPTER 6. PERFORMANCE TESTING

1,036) / 15.5826). It is also worth mentioning that the improved time using unique look up is timed with the storage of decoys in the database. The iteration methods (15.5826) are timed without storing to the database, as it would take longer time for each run. This is explained in chapter 5.

Implemented lookup method

```
if(!$DECOYLIST[$PasswordSHA]) {  
    $collision = false;  
} else {  
    $collision = true;  
}
```

As values are stored to the database, there is an increase of decoys that need to be checked after every run. The graph shows that the increase of decoys has little or no impact on time needed to check uniqueness with the implementation of look up checks.

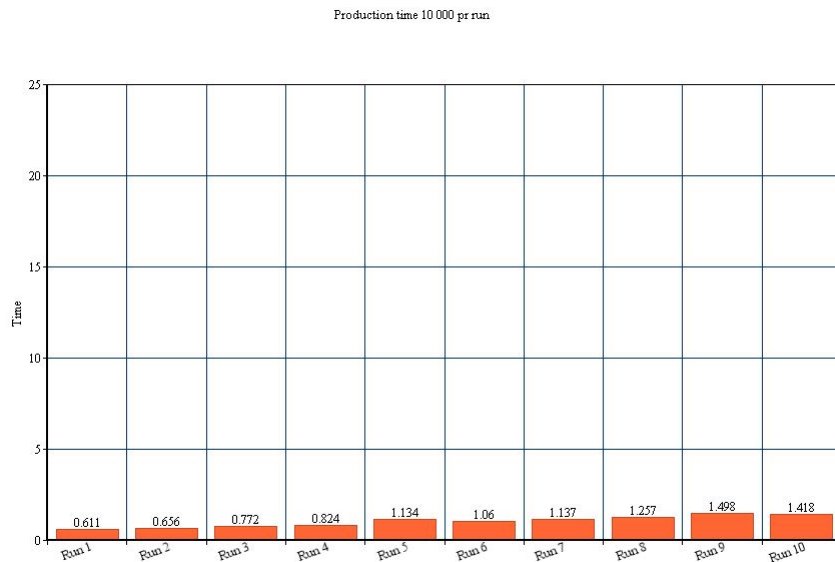


Figure 6.11: Unique Lookup check with 0-100 000 decoys in the database

Comparison of string comparison vs associative array lookup.

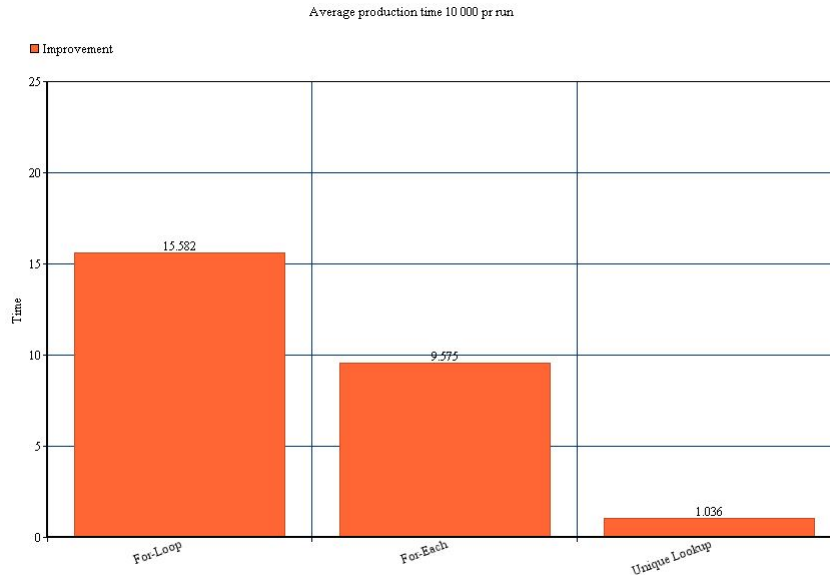


Figure 6.12: Comparison of production 10 000

6.2.1 Decoy generation

(All examples showed are produced by decoy generation, but are **simplified** for illustration purposes. If sufficient camouflage is obtained with examples of 10 decoys, it is assumed that camouflage quality is higher in the production of 10 000.)

In our testing we used a dictionary with a total of 114746 words and names.[6, 13] By allowing several words to be combined, possible combinations increase (2 words = 114746×114746), a sufficient number of possible outcomes for testing. In a real scenario there should be implemented dictionaries in a much larger scale. There are also many possibilities in the decoy production itself [20]. If it is desired that decoys only were to contain a word once, the words may be removed from the dictionary after use. The length of passwords could be adjusted to contain

one to several words, with other variations as well. The database should also be merged with random passwords in cases where users implement the use of random generated passwords, keeping an equal spread of both dictionary passwords and random passwords in the database.

Our focus was to produce decoys both efficiently, and with good enough quality to perform a sufficient amount of camouflage, indistinguishable by the human eye. Further down the line decoys produced should be analyzed by machine learning software and other pattern recognition tools to avoid exposure of structure, but for our testing purposes in relation to production efficiency, our results were sufficient. [28]

Scenarios to avoid in relation to exposing original password structure, or elimination.

- Decoys are all uppercase letters - (Violates restrictions)
- Decoys are all lowercase letters - (Violates restrictions)
- All decoys are of the same length as the original - (Expose structure)
- Decoy length is shorter than requirement in application
- (Violates restrictions)
- Decoys have unrealistic length - (Look computerized)
- Decoys are of another language - (Look computerized)
- Decoys have unrealistic amount of numbers or special characters - (Look computerized)
- Decoys do not contain words - (Look computerized)

By letting a function choose if one or more words are used, the length of the original password is not exposed. By limiting the amount of words put together to avoid unrealistically long passwords, our results show sufficient camouflage.

An example of insufficient camouflage:

1	cauCAsian
2	pRopPiNg
3	prognosticAting
4	engendereDbodybuilDeR
5	imparTiALItY
6	unfOrgiviNg
7	pointblanknuThouse
8	misarraNgebRoadcasted
9	PASSWORDDEViCE
10	CurETTeSADoptErs
11	ARtistes

Figure 6.13: Original password:PASSWORDDEViCE

In this example, the original password (PASSWORDDEViCE) could easily be distinguished from the decoys. The reason for this is, even though the original password contains at least 1 uppercase letter, the decoys produced fail in producing good camouflage.

Example of sufficient camouflage with same input:

1	INTIMating
2	aMBuLANCe
3	uLTimaTeShisTorlcITY
4	DleseS
5	dwAnAaggie
6	daNSeUrS
7	cROAkERs
8	shArpeneRcOnfabulAtiOn
9	BiENnIAIs
10	PASSWORDDEViCE
11	ScuTCHEOn

Figure 6.14: Original password:PASSWORDDEViCE

Example showing sufficient camouflage with less uppercase letters in original:

1	cOCKneYS
2	finaliSm
3	exotERictRUsteRs
4	tEachable
5	PrEvoCATioNal
6	bulWarkS
7	crystallograpHY
8	bLasphemes
9	passwOrDDeViCE
10	auDItiOninguniNToxicaTeD
11	saddlERY

Figure 6.15: Original password: passwOrDDeViCE

CHAPTER 6. PERFORMANCE TESTING

Production example of 10 decoys with valid strong input, illustrated in steps:

Original input = 1PAS2sworDdeVi@Ce

1	radio - chemistry
2	pocketing - biosyntheses
3	prepositions
4	parturitions-derails
5	ultrahigh - gunshots
6	realtors
7	quotational - leggins
8	attract - calumets
9	shul- reaffirmed
10	subentries

Figure 6.16: Step 1: Choice of words, separated with '-'

1	radiOchemIStry
2	pockEtIngBiosyntHesEs
3	prepositiOnS
4	parTuritiOnsderailS
5	ultraHIGHgUNshoTs
6	ReALTOrs
7	quotatiONalleGgins
8	attracTcalumeTs
9	shULrEaffIrmEd
10	sUbenTries

Figure 6.17: Step 2: Choice of uppercase letters

1	ra3diOcheml5Stry
2	p7ockEtIngBiosyntHesEs
3	7prepo3sltioS
4	parTuritionsdera8iLs
5	u3ltraHIGHgUNsh1oTs
6	ReALTO74rs
7	qu4otatioNalleGgins
8	attracTc7alumeTs
9	s3hULrEafFlrmed
10	3sUbenT8ries

Figure 6.18: Step 3: Choice of numerals

1	ra3diOcheml5+Stry
2	p7oc&kEtIngBiosyntHesEs
3	7prepo3sltio@nS
4	parTurit+ionsdera8iLs
5	u3ltraHIGHgUN&sh1oTs
6	Re+ALTO74rs
7	qu4otat&ioNalleGgins
8	attracTc7alume\$Ts
9	s3hULrE!afFlrmed
10	3sU+benT8ries
11	1PAS2sworDdeVi@Ce

Figure 6.19: Step 4: Choice of special characters, final result

We feel this implementation of decoys give sufficient results for testing the solution, both with weak and strong password input.

Chapter 7

Security analysis

7.1 Attack vectors

7.1.1 Attack vectors targeting the application

Granting access to the application

If an attacker manages to steal a victims mobile phone he will definitely try to access the application to obtain sensitive information that is stored in the password manager. When the attacker tries to log in he will always gain access with any password he tries. What he does not know is whether the password for the given entry is correct or a generated decoy password. Always giving the attacker positive feedback regarding his input forces him to try the password online in order to find out if it is correct. These failed attempts are easier to notice online and actions can be taken to block the user account. This can be done with honey words which is explained in 7.2.1. The probability of hitting the correct password is based upon the number of decoys that is generated for each password. By looking at our tests the probability of hitting the correct index is clearly dropping from a thousand to

ten thousand decoys. If the attacker is not lucky, he has to do a lot of work in order to hit the correct password. The number of decoys can also be increased. Hitting the correct index does not mean that the attacker has found the correct master password. In order to get the correct password for all services the attacker has to find the correct master password or that the SHA-256 algorithm produces a collision.

Probability of hitting the correct password

As mentioned, an attacker will gain access to the application no matter what master password he types in. Depending on what service that is chosen the index will be calculated and point to a password in the database. The attacker has the possibility to be lucky enough to get the correct password, but this probability is very low, depending on how many decoys that are produced for each password. Even though the application is displaying passwords to the user he still has to try them online in order to find out if it is the correct password for the selected service. As our results show, an increase of decoys will lead to a lower probability of hitting the correct password.

Attacker analyses the source code to find vulnerabilities

It is not hard to reveal the source code from an android application. To learn more about our application and to find potential vulnerabilities to exploit, an attacker can reveal the source code and analyze it in hope of finding vulnerabilities that we have unintentionally made while developing the application. This can give the attacker an upper hand in attacking our application to retrieve sensitive information. To make it harder for an attacker reverse-engineering the source code, obfuscation can be implemented in order to prevent this [42]. The point of obfuscation is to make the source code more difficult to understand in order to protect it. This technique is usually used to protect intellectual property [21]. Obfuscation is not

implemented into our solution as it is today.

Attacker gets access to the database file

Another possibility is if the attacker gets hold of the database file. This is almost the same attack vector as trying to hit the correct password by typing in arbitrary master passwords as input. The one thing that differs is that the attacker has a better overview and can view all passwords at once instead of the application revealing one at a time. Even though the attacker has full overview of the passwords he does not know which password is the correct one, since it is camouflaged in decoy passwords similar to the correct one. The only possibility is to try random passwords and hope he chooses the correct password. To know which password belongs to the proper service, the attacker also needs the shared preferences file.

Getting hold of the database file is not that easy, even when the mobile device is stolen. The database file is stored in the internal memory of the mobile device and is therefore protected by security features provided by the Android platform. The file is private only to the application and the file is inaccessible by other applications and also the user [1]. However, there are known vulnerabilities regarding the internal storage in Android [14].

Collision probability using SHA-256

The importance of always creating a unique value based on an input, in our case the master password concatenated with the service is crucial for our application, since this is used to locate the correct password for a given service. If we were using hash algorithms with a high collision probability the chances of hitting the correct master password would be higher and this would have weakened the security of the application. For us it is important that the probability of an attacker typing in another master password that results in the same hash value as the original

master password is very low. Therefore we are using the cryptographic algorithm SHA-256 [40]. In a hash function where the output is m-bits it is possible to find a collision h in $2^{m/2}$ tries [23] if h is regular. The probability of collision with the SHA-256 algorithm requires 2^{128} computations on average to get a collision based on the birthday attack [40].

This number is so large that it is infeasible to try all combinations in order to get a collision and therefore we are confident using SHA-256 as a hash function in our design.

7.1.2 Attack vectors targeting the cloud

Attacker gains access to the cloud

If an attacker is able to gain access to the cloud it would be almost the same as accessing the database file on the mobile device. The database in the cloud contains only passwords to correlate with newly generated decoys by ensuring that we are making unique decoys. The attacker becomes aware of all the passwords but does not know which websites they belong to, since he needs the "Shared Preferences" file with the additional information regarding the services. The correct passwords are also camouflaged in decoys making it harder to locate. As mentioned, storing the passwords in the cloud is not necessary (testing purposes only). Only storing the hashes in the cloud database will make it harder for the attacker to retrieve the passwords. The attacker has to compute a lot of hashes in order find those that match the ones in the database.

Attacker eavesdropping on traffic to and from the cloud

Implementing a cloud environment into our solution opens up the possibility of an attacker eavesdropping on traffic between the application and the cloud. Not

using encryption between the application and the cloud will result in sending data in plain text. To solve this SSL [25] can be implemented in order to increase the security of sending data with the use of SSL. SSL is not implemented in the solution as it is today.

As explained earlier in 4.2.1, when the user wants to create a password it is sent to the cloud where it will be processed and decoys similar to the original passwords will be created. The original password being sent to the cloud has to be sent over a secure channel and therefore we have to implement SSL. Not only for secure data transfer but also for authenticity between client and server.

SSL is an abbreviation for "Secure socket layer" and is used for securing Internet communication [25]. It is used together with the HTTP-protocol (Hyper Text transfer Protocol), which is used to communicate on the World Wide Web. Usually traffic is sent over HTTP port 80 (insecure communication), but when implementing SSL the traffic is sent over HTTPS port 443 (secure communication). SSL is a protocol that provides two very important factors regarding Internet security:

1. **Encryption:** The SSL-protocol provides encryption between the communication from client and server, securing sensitive data and providing integrity and confidentiality. When the client initiates the connection, the web server will respond to the request with a certificate and a public key. The client will then create a symmetric session key, encrypt this key with the public key of the web server and send it to the web server. When the web server receives the encrypted message it can decrypt it with its private key. The symmetric session key will be used to encrypt and decrypt the traffic between the client and server.
2. **Authentication:** The SSL-protocol also provides authentication. When a user visits a website the website has to present a SSL-certificate to prove their identity. The certificate is signed and issued by a certified certifi-

cate authority. Whenever a client want to connect to the server, the server presents its certificate for the client. The client will then verify the validity of the certificate with the certificate authority that issued the certificate.

7.2 Additional functions to enhance security

7.2.1 Honeywords

Blocking an attacker trying to brute-force a user account is quite challenging. A user account could be blocked after a given of incorrect tries, but this would only lead to a Denial of Service attack [48, 54] which leads to administrators using a huge amount of time unlocking user accounts. There are several other possibilities such as blocking an IP-address after a given of incorrect logins but this can lead to other legitimate users being blocked, since several users can share the same public IP-address [44]. One could also add a delay that increases every time an incorrect password is entered, but this would only slow an attacker down and not ultimately disable him from his malicious intents. To solve the problem of Denial of service attacks, websites have implemented a system known as CAPTCHA [54, 52, 43]. This is a system that will distinguish people from computers. After a certain number of failed login attempts, the website will present a test that is easy for humans but difficult for computers to pass. This will only stop brute-force attempts towards the website, but some additional security is needed to recognize when an attacker is trying to access a user account with information from a stolen device.

In order to detect when an attacker is trying to use compromised information from stolen devices, some defense mechanisms are required. One defense is the use of honey words [27, 46, 39, 20]. Honey words are "fake" passwords that will set of an alarm if anyone of these "fake" passwords are used when trying to login to a particular account. In our case the "fake" passwords would be our decoy pass-

words. A user could add some of these decoys on websites to be used as honey words and when these passwords are used the account will be locked, preventing an attacker from accessing the account [27]. This would effectively detect an attacker trying to use sensitive information from a stolen device.

Chapter 8

Discussion, conclusion and further work

8.1 Discussion of results

8.1.1 Hit rate results

Hit rate with 1000 decoys

Figure 6.2 shows the hit rate from our test with 1000 decoys camouflaging the correct password. We clearly see that the probability of hitting the correct password is too high. The average of hitting the correct password from these ten runs is 11,3 times with our test. These results show that 1000 decoys are not enough to "protect" the correct password.

Hit rate with 5000 decoys

Figure 6.3 shows a significant difference when increasing to 5000 decoys. The hit rate decreases significantly when increasing from 1000 to 5000 decoys. The average of hitting the correct password from these ten runs is 1,3 times with our test. Increasing to 5000 decoys gives positive results when it comes to the chance of hitting the correct password.

Hit rate with 10000 decoys

Figure 6.4 does not show a significant difference from the results with 5000 decoys. The runs 2, 5, 6, and 8 are similar to some of the runs with 5000 decoys. The main difference is that there are more runs where the number of hits is zero. The average of hitting the correct password from these ten runs is 0,9 times with our test.

Overall discussion of hit rates

We see that camouflaging the correct password with 1000 decoys is clearly not enough. The chances of hitting the correct password are too high and would not be sufficient to protect the correct password.

Increasing to 5000 decoys shows a significant improvement from 1000 decoys. The average hit rate drops from 11,3 to 1,3, which is a acceptable improvement.

Furthermore, increasing to 10000 decoys does not show a significant improvement from using 5000 decoys. The average hit rate drops from 1,3 to 0,9. The most noteworthy difference is that there are more runs where the number of hits is zero.

Let's assume the following:

- The attacker has not the possibility of using machine learning in order to separate our decoys from the correct password
- Each login takes 10 seconds

Even without encryption an attacker has to do significantly of online work on average in order to determine the correct password. The average of hitting the correct password when camouflaging with 10000 decoys is 5000. This means that an attacker will in total use 50000 seconds, which equals 13,8 hours. This means that an attacker would have to sit 13,8 hours constantly trying passwords before hitting the correct one.

8.1.2 Decoy production in Cloud

In a scenario where a users' passwords were random, decoy production could also be performed on the mobile device, as the source code would not expose more than that the decoy production would use random functions. Random production is a spectacular solution, if only users were willing to use random passwords on all of their services, unfortunately, this is not the case [55, 36]. In the production of human readable passwords, the situation becomes quiet different, as dictionaries must be implemented, and that several parameters are used in the production. If these parameters were to be exposed, an attacker could use this to his advantage, as well as if an attacker got a hold of the dictionaries used.

As communication between a android device and a cloud can be categorized as secure, hiding the decoy production in a cloud acts as an advantage. If a larger amount of decoys was preferred, the amount of resources available in a cloud is

much larger than on a mobile device which is highly prioritized as the production has to be fast in order to be user friendly.

Discussion of decoy generation

There are several choices in relation to producing decoys. For testing purposes we did not implement the largest dictionaries, because the smaller the dictionary, the higher the probability of collisions happening, which is good for testing purposes. In a real implementation, the dictionaries should be much larger. Advantages are an increase of probability that a users password contains words from a dictionary, possible combinations would increase, thus resulting in a decrease of collision probability. Disadvantages are that larger dictionaries demand more resources.

For a higher level of uniqueness, words may be removed from a dictionary after use, thus resulting in a decrease of possible outcomes. Passwords may look alike, but still be unique because of uppercase/lowercase letters, numbers and special characters. In our solution the importance of performance and combination possibilities was chosen over the level of uniqueness.

In the production of decoys the camouflage effect is highly important. There is no point on hiding a password in between 10 000 decoys if the password lights up as different by the human eye. Avoiding worst-case scenarios is a minimum, examples being decoys not fulfilling the same requirements as the original password, or decoys that clearly are computerised. Decoy generation is an art in itself, and can always be optimized and improved. A choice of performance vs. quality must be settled before development as many choices must be made.

Our choice was efficiency over quality in regard to implementation on a mobile device.

The amount of decoys produced for each service is set to 10 000. The reasons being testing, time and probability. For our testing purposes we achieved acceptable performance in regard to hit rate and showing low likelihood of hitting the correct passwords unintentionally.

When a user creates a new service and saves a new password, it is important that the decoy production is efficient. One reason being user friendliness, but also keeping the needed connection time to a cloud at a minimum. Testing shows that with production set to 10 000, time needed for production is approximately 1 second. This shows that producing decoys in a much larger scale is also feasible.

Camouflage quality

In the analysis of a user's password, it is important that the decoys produced are similar enough to camouflage the valid information, but not so similar resulting in exposing patterns or structure of the original password. If all decoys produced were of the same character length as the original, the attacker would know the original length immediately. Other examples are if all decoys contain the same amount of numbers, uppercase letters or special characters, this would expose a great deal of valuable information in regard to penetrating the system.

Finding a balance between good camouflage and variance is not easy, but our implementation is acceptable in its results.

We feel this produces sufficient camouflage without exposing much of the original password.

In the process of choosing the amount of numbers and special characters, it is apparent that many of the decoys expose them selves as computerized (Not humanly readable) which is a big disadvantage. Our choice concluded with keeping

numbers and special characters to a random choice between 1 and the amount in the original password. This produces good camouflage quality, and is efficient, but does in some manner expose the maximum amount of numbers and special characters in the original password by analyzing the decoys. Value of camouflage was chosen over possible exposure of original structure.

8.1.3 Storing passwords

Decoy production in the cloud has to check for uniqueness and avoid the production of duplicates. Both the unique hash of the passwords and the passwords are stored in the database. This would expose a user's password for the developers. Storing the passwords in plain text as shown on earlier figures is not necessary for the cloud to work, and is only done for testing purposes in this "proof of concept". Storing the hashes would suffice.

8.1.4 Modulo as index function

Using modulo in our index function makes it possible for us to always give a positive feedback independent of what master password that is being tried. This results in an attacker never knowing if he typed in the correct password. The drawback of using modulo as our index function is that the attacker has the possibility of hitting the correct password if the master password being tried is wrong. The results show that increasing the number of decoys have a great impact whether the attacker hits the correct password or not. We also see that the authors of "NoCrack" [28] used modular arithmetic in one of their honey encryption schemes.

8.1.5 Performance

Application performance

Looking at figure 6.1, our solution is performing very well on the different operations. Add, delete, show, and edit takes only one second to perform which is very acceptable. The operation that takes the most time is changing the master password. This takes nine seconds, and since this is not an operation carried out so often makes it acceptable.

Overall discussion of performance

The tests we have performed are done on equipment with relatively good specifications. The results we are showing regarding performance will of course increase or decrease depending on the specifications of the devices that uses our solution. Mainly it will only be dependent on the mobile device used since the cloud environment is to be run on a powerful server.

We did not test the time from the application to the cloud and vice versa, since this is dependent on what type of Internet connection the user is using. The two most important elements for us to test are the application and the cloud performance.

8.2 Conclusion

Endorsing the use of passwords to protect systems, rest on developed encryption methods being classified as secure. As of today, a large amount of faith rests in AES-256 [32] encryption being unbreakable. (Encryption standard approved by the US government in June 2003 to protect information categorized as TOP SE-

CRET). In the year 2015, despite flaws found in the encryption algorithm, security keys still haven't been broken.

Brute-force attacks are categorized as a threat to all systems that are implemented with passwords, as they break encryption and are difficult to protect against.

The success of brute-force attacks is based on the results given. A normal system would either return error (No access), or success (Full access) in the situation of an attack. If information is returned, other than error, the information is confirmed as valid. The reason being, in standard systems, all information stored is valid. The whole system is secured by one single factor, the encryption. If the encryption is broken, everything is revealed.

By elimination the possibility of error, the attacker has no reference as to if an attack is successful or not. Furthermore, by eliminating the possibility of confirming a result as valid, confirmation is impossible, eliminating the possibility of successful brute force attacks all together.

Creating a system categorized as secure, but also a solution that users are willing to implement, is a challenge. Higher security often involves an increase of difficulty in user-friendliness, resulting in users either skipping steps or disregarding the system all together [30]. It is also important that a user feels the system is both safe and secure to use, eliminating the possibility of a system being so complex that it may result in users performing unintended actions (mistakes), resulting in poor security.

Our system is designed so no errors are given in the situation of an attack, eliminating the possibility of confirming a brute-force attack as successful. Even if the whole database is returned, the attacker has no possibility of separating valid information from decoys, as everything is camouflaged. Overall security does not

depend on one single factor, but rather on multiple factors. In a scenario where honey words were to be implemented, the whole situation could lead to the attacker in being trapped, or locking down the system, eliminating further progress in the attack as well.

Implementing our application on a mobile device is highly feasible as decoy production is efficient and produced in the cloud. Even though this requires a stable Internet connection, it does not affect the user in a large degree if no connection is possible. There are many scenarios where it is troublesome to demand Internet connection all the time. One example being, looking up access information in the password manager when located far underground with no Internet connectivity. The production of decoys is based on dictionary combinations, gaining users the possibility of using their own passwords. If random passwords are wanted these can easily be generated for the user on the mobile device, and even implemented in the decoy database in combination with humanly readable passwords.

The implementation of decoys does not affect the user experience, increasing overall security without an increase of complexity for the user. The architecture the system is built on also allows further development of features, a few examples being SSL traffic and random generated passwords among others.

Even without encryption, our design architecture and implementation of decoys, result in preventing brute-force attacks to a large degree, categorizing these types of attacks as a minimal threat.

We have achieved successful results in relation to our goals.

8.3 Further Work

There are three elements we consider as important for further work.

The first element is implementation of SSL between the application and the cloud environment to provide encryption and authenticity. Anti-theft functions such as remote wipe and track my mobile device can also be implemented.

Develop the production of decoys further, so that it becomes more generic and produces decoys that are even more plausible and not easy to rule out using different techniques, such as machine learning and pattern recognition.

The third element is implementation of encryption into our solution needs to be investigated. Implementation of encryption must be done in a correct manner [17], and an architecture that is based on decoys creates some challenges. Further investigation should be to examine whether it is practicable to implement encryption into our solution. By this we mean that there is no point in implementing encryption if the design does not allow it to perform seamlessly. It may turn out that some changes in the design are needed in order to implement encryption in a correct way.

Bibliography

- [1] “Android internal storage,” <http://developer.android.com/guide/topics/data/data-storage.html#filesInternal>, [Online: accessed: 29-04-2015].
- [2] “Android shared preferences,” <http://developer.android.com/reference/android/content/SharedPreferences.html>, [Online: accessed: 18-04-2015].
- [3] “Android sqlite database,” <http://developer.android.com/reference/android/database/sqlite/SQLiteDatabase.html>, [Online: accessed: 24-02-2015].
- [4] “Android studio,” <http://developer.android.com/tools/studio/index.html>, [Online: accessed: 20-04-2015].
- [5] “Android tools,” <https://developer.android.com/tools/help/index.html>, [Online: accessed: 22-04-2015].
- [6] “Dictionary,” <http://www-01.sil.org/linguistics/wordlists/english/wordlist/wordsEn.txt>, [Online: accessed: 23-04-2015].
- [7] “The importance of using strong passwords,” <https://msdn.microsoft.com/nb-no/enus/library/ms851492%28v=winembedded.11%29.aspx>, [Online: accessed: 14-04-2015].
- [8] “Java - secure random class,” <https://docs.oracle.com/javase/7/docs/api/java/security/SecureRandom.html>, [Online: accessed: 4-05-2015].
- [9] “Keepass - official website,” <http://keepass.info>, [Online: accessed: 30-04-2015].
- [10] “Lastpass - official website,” <https://lastpass.com/>, [Online: accessed: 19-05-2015].

BIBLIOGRAPHY

- [11] “Mamp,” <https://www.mamp.info/en/>, [Online: accessed: 22-04-2015].
- [12] “Mersenne twister - creator,” <http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/emt.html>, [Online: accessed: 1-05-2015].
- [13] “Names,” <http://deron.meranda.us/data/census-derived-all-first.txt>, [Online: accessed: 23-04-2015].
- [14] “Palo alto - insecure internal storage in android,” <http://researchcenter.paloaltonetworks.com/2014/08/insecure-internal-storage-android/>, [Online: accessed: 23-05-2015].
- [15] “Php - mtrand,” <http://php.net/manual/en/function.mt-rand.php>, [Online: accessed: 1-05-2015].
- [16] “Php - official website,” <http://php.net>, [Online: accessed: 1-04-2015].
- [17] “Use cryptography correctly,” <http://cybersecurity.ieee.org/center-for-secure-design/use-cryptography-correctly.html>, [Online: accessed: 25-05-2015].
- [18] M. Ajtai, “The complexity of the pigeonhole principle,” in *Foundations of Computer Science, 1988., 29th Annual Symposium on.* IEEE, 1988, pp. 346–355.
- [19] E. Albrechtsen, “Security vs safety,” 2003.
- [20] M. H. Almeshekah, C. N. Gutierrez, M. J. Atallah, and E. H. Spafford, “Ersatzpasswords—ending password cracking,” 2015.
- [21] A. Balakrishnan and C. Schulze, “Code obfuscation literature survey,” *CS701 Construction of Compilers*, vol. 19, 2005.
- [22] A. Belenko and D. Sklyarov, “secure password managers and military-grade encryption on smartphones: Oh, really?” *Blackhat Europe*, 2012.
- [23] M. Bellare and T. Kohno, “Hash function balance and its impact on birthday attacks,” in *Advances in Cryptology-Eurocrypt 2004.* Springer, 2004, pp. 401–418.

BIBLIOGRAPHY

- [24] N. Ben-Asher, N. Kirschnick, H. Sieger, J. Meyer, A. Ben-Oved, and S. Möller, “On the need for different security methods on mobile phones,” in *Proceedings of the 13th International Conference on Human Computer Interaction with Mobile Devices and Services*. ACM, 2011, pp. 465–473.
- [25] M. S. Bhogade, “Secure socket layer,” in *Computer Science and Information Technology Education Conference*, 2002.
- [26] M. Blanchou and P. Youn, “Password managers exposing passwords everywhere,” <https://isecpartners.github.io/whitepapers/passwords/2013/11/05/Browser-Extension-Password-Managers.html>, 2013.
- [27] H. Bojinov, E. Bursztein, X. Boyen, and D. Boneh, “Kamouflage: Loss-resistant password management,” in *Computer Security—ESORICS 2010*. Springer, 2010, pp. 286–302.
- [28] R. Chatterjee, J. Bonneau, A. Juels, and T. Ristenpart, “Cracking-resistant password vaults using natural language encoders,” 2015.
- [29] M. Ciampa, M. Revels, and J. Enamait, “Online versus local password management applications: An analysis of user training and reactions,” *Journal of Applied Security Research*, vol. 6, no. 4, pp. 449–466, 2011.
- [30] L. F. Cranor and S. Garfinkel, *Security and usability: designing secure systems that people can use*. ” O’Reilly Media, Inc.”, 2005.
- [31] A. Czeskis, M. Dietz, T. Kohno, D. Wallach, and D. Balfanz, “Strengthening user authentication through opportunistic cryptographic identity assertions,” ACM, pp. 404–414, 2012.
- [32] J. Daemen and V. Rijmen, *The design of Rijndael: AES—the advanced encryption standard*. Springer Science & Business Media, 2002.
- [33] M. Fabian and M. A. Terzis, “My botnet is bigger than yours (maybe, better than yours): why size estimates remain challenging,” in *Proceedings of the 1st USENIX Workshop on Hot Topics in Understanding Botnets, Cambridge, USA*, 2007.
- [34] S. Fahl, M. Harbach, M. Oltrogge, T. Muders, and M. Smith, “Hey, you, get off of my clipboard,” in *Financial Cryptography and Data Security*. Springer, 2013, pp. 144–161.

BIBLIOGRAPHY

- [35] C. Fruhwirth, “Tks1-an anti-forensic, two level, and iterated key setup scheme,” 2004.
- [36] S. Furnell, “An assessment of website password practices,” *Computers & Security*, vol. 26, no. 7, pp. 445–451, 2007.
- [37] P. Gasti and K. B. Rasmussen, “On the security of password manager database formats,” pp. 770–787, 2012.
- [38] S. Gaw and E. W. Felten, “Password management strategies for online accounts,” ACM, pp. 44–55, 2006.
- [39] Z. A. Genc, S. Kardas, and M. S. Kiraz, “Examination of a new defense mechanism: Honeywords.” *IACR Cryptology ePrint Archive*, vol. 2013, p. 696, 2013.
- [40] H. Gilbert and H. Handschuh, “Security analysis of sha-256 and sisters,” in *Selected Areas in Cryptography*. Springer, 2004, pp. 175–193.
- [41] R. Gonzalez, E. Y. Chen, and C. Jackson, “Automated password extraction attack on modern password managers,” *arXiv preprint arXiv:1309.1416*, 2013.
- [42] S. Hada, “Zero-knowledge and code obfuscation,” in *Advances in Cryptology ASIACRYPT 2000*. Springer, 2000, pp. 443–457.
- [43] A. Hurkała and J. Hurkała, “Authentication system for websites with paid content: An overview of security and usability issues,” *IJCSNS International Journal of Computer Science and Network Security*, vol. 13, no. 7, pp. 42–49, 2013.
- [44] P. Johnston, “Authentication and session management on the web,” *Retrieved December*, vol. 13, p. 2009, 2004.
- [45] A. Juels and T. Ristenpart, “Honey encryption: Security beyond the brute-force bound,” in *Advances in Cryptology–EUROCRYPT 2014*. Springer, 2014, pp. 293–310.
- [46] A. Juels and R. L. Rivest, “Honeywords: Making password-cracking detectable,” in *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*. ACM, 2013, pp. 145–160.

BIBLIOGRAPHY

- [47] Z. Li, W. He, D. Akhawe, and D. Song, “The emperor’s new password manager: Security analysis of web-based password managers,” in *23rd USENIX Security Symposium (USENIX Security 14)*, 2014.
- [48] D. Moore, C. Shannon, D. J. Brown, G. M. Voelker, and S. Savage, “Inferring internet denial-of-service activity,” *ACM Transactions on Computer Systems (TOCS)*, vol. 24, no. 2, pp. 115–139, 2006.
- [49] A. Narayanan and V. Shmatikov, “Fast dictionary attacks on passwords using time-space tradeoff,” in *Proceedings of the 12th ACM conference on Computer and communications security*. ACM, 2005, pp. 364–372.
- [50] K. Rosen, *Discrete Mathematics and Its Applications 6th edition (International edition)*. McGraw-Hill, 2007.
- [51] G. Sowmya, D. Jamuna, and M. V. K. Reddy, “Blocking of brute force attack,” in *International Journal of Engineering Research and Technology*, vol. 1, no. 6 (August-2012). ESRSA Publications, 2012.
- [52] —, “Blocking of brute force attack,” in *International Journal of Engineering Research and Technology*, vol. 1, no. 6 (August-2012). ESRSA Publications, 2012.
- [53] W. C. Summers and E. Bosworth, “Password policy: the good, the bad, and the ugly,” in *Proceedings of the winter international symposium on Information and communication technologies*. Trinity College Dublin, 2004, pp. 1–6.
- [54] L. Von Ahn, M. Blum, N. J. Hopper, and J. Langford, “Captcha: Using hard ai problems for security,” in *Advances in CryptologyEUROCRYPT 2003*. Springer, 2003, pp. 294–311.
- [55] J. Yan *et al.*, “Password memorability and security: Empirical results,” *IEEE Security & privacy*, no. 5, pp. 25–31, 2004.