Bachelor of Science in Computer Science & Engineering

Developing a Password Breach Detection Technique for Serving the Purpose of Honeywords

by

Audri Banik

ID: 1504096

Department of Computer Science & Engineering

Chittagong University of Engineering & Technology (CUET)

Chattogram-4349, Bangladesh.

August, 2022

Developing a Password Breach Detection Technique for Serving the Purpose of Honeywords

Submitted in partial fulfilment of the requirements for
Degree of Bachelor of Science
in Computer Science & Engineering

by

Audri Banik
ID: 1504096

Supervised by

Ashim Dey
Assistant Professor
Department of Computer Science & Engineering

Chittagong University of Engineering & Technology (CUET)
Chattogram-4349, Bangladesh.

The thesis titled '**Developing a Password Breach Detection Technique for Serving the Purpose of Honeywords**' submitted by ID: 1504096, Session 2019-2020 has been accepted as satisfactory in fulfilment of the requirement for the degree of Bachelor of Science in Computer Science & Engineering to be awarded by the Chittagong University of Engineering & Technology (CUET).

Board of Examiners

Supervisor

Ashim Dey
Assistant Professor
Department of Computer Science & Engineering
Chittagong University of Engineering & Technology (CUET)

Head of Department

Dr. Md. Mokammel Haque
Professor & Head
Department of Computer Science & Engineering
Chittagong University of Engineering & Technology (CUET)

External

Dr. Md. Mokammel Haque
Professor
Department of Computer Science & Engineering
Chittagong University of Engineering & Technology (CUET)

Declaration of Originality

This is to certify that I am the sole author of this thesis and that neither any part of this thesis nor the whole of the thesis has been submitted for a degree to any other institution.

I certify that, to the best of my knowledge, my thesis does not infringe upon anyone's copyright nor violate any proprietary rights and that any ideas, techniques, quotations, or any other material from the work of other people included in my thesis, published or otherwise, are fully acknowledged in accordance with the standard referencing practices. I am also aware that if any infringement of anyone's copyright is found, whether intentional or otherwise, I may be subject to legal and disciplinary action determined by Dept. of CSE, CUET.

I hereby assign every rights in the copyright of this thesis work to Dept. of CSE, CUET, who shall be the owner of the copyright of this work and any reproduction or use in any form or by any means whatsoever is prohibited without the consent of Dept. of CSE, CUET.

Signature of the candidate

Date:

Acknowledgements

The completion of this thesis would not have been possible without the support and supervision rendered by some people whom I would like to thank from the deep core of my heart.

Firstly, I want to convey my sincere gratitude to my supervisor for his guidance in making this thesis a reality. His given inspiration for doing research on the field of computer science that interests me, his infinite patience, understanding and willingness to let me find my own path, have all contributed to my humble development of this thesis's ideas. I really appreciate his constant mentoring and probing questions, which have pushed me to think outside the box.

I am grateful to the panel members for their support in approving my work. I appreciate my family's compassion in bearing with me at this difficult time. Their unwavering support and encouragement have been invaluable.

Abstract

Password-based authentication, although popularly adopted for its simplicity, is vulnerable to an inversion attack paradigm, in which the adversary obtains the plaintext password from the hashed value of the password. Honeyword-based authentication has been introduced to combat such attacks. Along with the user's original password, certain dummy passwords or honeywords are also saved in this strategy. Although this technique is adequate for dealing with the aforementioned security vulnerability, the requirement of additional storage to store the honeywords is still an overhead. In our proposed scheme, we tried to develop a password breach detection technique that will reduce this storage overhead while mimicking the concept of honeywords with much simplification. Maintaining the standard security and usability parameters is our concern also.

Table of Contents

Acknowledgements **iii**

Abstract **iv**

List of Figures **viii**

List of Tables **ix**

1 Introduction **1**

1.1 Introduction 1

1.1.1 Problem Statement 1

1.1.2 Objectives 3

1.2 Framework/Design Overview 3

1.3 Difficulties 4

1.4 Applications 5

1.5 Motivation 5

1.6 Contribution to the thesis 5

1.7 Thesis Organization 6

1.8 Conclusion 7

2 Literature Review **8**

2.1 Introduction 8

2.1.1 Security Standards 8

2.1.1.1 Flatness 8

2.1.1.2 MSV 8

2.1.1.3 DoS resiliency 9

2.1.2 Usability Standards 9

2.1.2.1 System interference 9

2.1.2.2 Stress on memorability 9

2.1.2.3 Typo safety 10

2.2 Related Literature Review 10

2.3 Conclusion 14

2.3.1 Implementation Challenges 14

3 Methodology **15**

3.1 Introduction 15
3.2 Diagram/Overview of Framework 15
3.3 Detailed Explanation 15
3.4 Implementation 18
3.4.1 create.ipynb 18
3.4.2 utils.py 18
3.4.2.1 username_availability_check(username) 18
3.4.2.2 pattern check 19
3.4.2.3 passwd_char(username, passwd) 20
3.4.2.4 find_positions(username, passwd) 20
3.4.2.5 passwdch_in_password_file(given_passwdch) . . 21
3.4.2.6 pos_in_positions_file(pos_match) 21
3.4.2.7 Graph plotting 22
3.5 Conclusion 22

4 Results and Discussions 23
4.1 Introduction 23
4.2 Evaluation of Framework 23
4.2.1 Evaluation of storage cost (theoretically): 23
4.2.2 Evaluation of storage cost (experimentally): 23
4.2.3 Evaluation of functionality: 24
4.2.3.1 Functionality testing of registration module: . . . 26
4.2.3.2 Inspecting a login routine: 27
4.2.4 Evaluation of usability and security standards: 30
4.2.4.1 Resistance against MSV attack: 30
4.2.4.2 Achieving flatness: 31
4.2.4.3 System interference: 31
4.2.4.4 Stress on memorability: 32
4.2.4.5 Typo-safety: 32
4.2.5 Social and Ethical Impact 32
4.3 Conclusion 32

5 Conclusion 33
5.1 Conclusion 33
5.2 Future Work 33

List of Figures

1.1 Design Overview 4

3.1 Registration Module Architecture 16
3.2 Login Module Architecture 16
3.3 contents of create.ipynb 19
3.4 Contents of username_availability_check(username) 19
3.5 block of code for pattern check 20
3.6 contents of passwd_char(username, passwd) 20
3.7 contents of find_positions(username, passwd) 20
3.8 contents of passwdch_in_password_file(given_passwdch) 21
3.9 contents of pos_in_positions_file(pos_match) 21
3.10 block of code for graph plotting 22

4.1 files generated for tweaking digits method 24
4.2 contents of passwords file in tweaking digits method 24
4.3 contents of index file in tweaking digits method 25
4.4 files generated for our proposed method 25
4.5 contents of passwords file in our proposed method 25
4.6 contents of positions file in our proposed method 25
4.7 contents of space.csv file 25
4.8 graphical representation of space required by our proposed model and tweaking digits model (k=5) 25
4.9 content of the user file while evaluating case 1 and case 2 26
4.10 response by system on case 1 registration routine 26
4.11 response by system on case 2 registration routine 27
4.12 a new user having username = 'user7' is registered onto the system (case 3 registration routine) 27
4.13 updated user file(case 3 registration routine) 28
4.14 updated passwords file(case 3 registration routine) 28
4.15 updated positions file(case 3 registration routine) 28
4.16 system response on case 1 (login routine) 29
4.17 system response on case 2 (login routine) 29
4.18 system response on case 3 (login routine) 30
4.19 breach detected message displayed in the screen(case 4 login routine) 30
4.20 MSV attack in tweaking digits method 31

4.21 MSV attack in our proposed method 31

List of Tables

3.1 Given username and password 17
3.2 Parsing and storing password 17
3.3 Given username and password 18
3.4 Login routine for table 3.3 18

4.1 space required for general tweaking digits method 23
4.2 space required for our proposed method 24

Chapter 1

Introduction

1.1 Introduction

In today's era, people use digital devices more than ever before. To protect the confidentiality, integrity and availability of computer system data from those with malicious intentions, information security is designed which may use basically 4 types of user authentication techniques namely, physical token authentication (e.g. an ID card), authentication by some user known attribute (e.g. a password), biometric authentication (e.g. a retina or fingerprint scan) and peer-level or human-intermediate authentication [1]. Among these types, password based authentication has the highest appeal for ease of memorability as well as for its cost-effectiveness and simple login functionality. Yet password based authentication is ill-famed because users choose poor passwords; making it prone to different attacks. Traditionally, passwords are stored in hashed form which were once thought to be irreversible. But recent studies proved that this is not the case anymore. Hence, to detect password cracking by an adversary, the concept of honeywords have been brought to light by some researchers where fake passwords are stored with the real password to confuse the attacker. This fulfills the objective but not without any cost. Additional storage overhead as well as diluted security and usability features come with it. Our project aims at reducing this storage overhead while not compromising the security or usability standards.

1.1.1 Problem Statement

Password based authentication system is adopted in almost all the sites where information security is to be maintained. But it is a weak method of authentication because users tend to set poor and identical passwords in all their accounts.

To give passwords extra security, their corresponding hashed values are stored in password-files instead of the actual passwords, with the idea of the hashed values being unalterable. But there have been reports of leaked passwords files of users at many well known sites and social media thus raising an alarm[1] [2].

Inversion attack is one such lately developed attack model, the main objective of which is to retrieve user's password (p) from the hashed value (H(p)) stored in the password file (Fp) using brute force search. In 2009, Weir et al. [3] proposed an algorithm based on probabilistic context free grammar to further improve the inversion rate from 28% to 129%. Ma et al., in [4], has shown that hashes can be inverted with a very high success rate using hidden Markov chain model . So, only hashing the password does not provide a rich mechanism of security. One solution to this problem may be making password hashing more complex and time-consuming. This approach can help, but also slows down the authentication process for legitimate users, and does not detect successful password cracking easily.

In [2], Juels & Rivest introduced honeyword based authentication technique for improving the security of hashed passwords. Honeywords are dummy passwords which are stored along with the original password associated with each user account. An adversary who has succeeded in stealing the hashed password file and inverted the hashes cannot tell which one is the actual password. An auxiliary server called the honeychecker (Hc) can distinguish between the original and the dummy password for any login routine and will set off an alarm when a honeyword is submitted thus making password cracking detectable.

Here it is important to note that-

- honeywords will not stop the system from being compromised i.e. if the password file is stolen it is already a matter of big concern;
- honeywords will not stop the stolen hashes from being cracked;
- honeywords will not stop the adversary from guessing the users' passwords from social engineering or keylogger or some random guess.

What honeywords do is making it a high-risk guessing game for the attacker

[1]https://www.upguard.com/blog/biggest-data-breaches

thereby an attempt to catch the event of password breach.
Although this technique is good enough to address the security breach mentioned earlier, the additional storage required to store the honeywords is still an overhead. Moreover, achieving the security and usability standards is also a matter of concern.

1.1.2 Objectives

The key objectives and possible outcomes of this work are mentioned as follows-

- To introduce a storage cost efficient password breach detection technique that satisfies the purpose of honeywords.
- To improve the security and usability standards compared to the existing honeyword generation schemes.
- To increase user-password's strength through the proposed scheme.

1.2 Framework/Design Overview

Our main idea behind the proposed scheme is not to let the attacker have the whole password through the password file. To achieve this we utilized the concept of extracting some portion of the original user password into another file named positions, which will hold the extracted characters values and their respective index positions in the actual password; thus executing a distributed security system with less storage cost as we do not have to store k number of honeywords' hashed form causing additional storage overhead.
The username's uniqueness is achieved through the registration module during the registration process. Every user trying to register with a unique username (that is not already included in the users list) are allowed to insert a password of their own choice with slightest interference from the system; like imposing general guidelines as- a password pattern containing of alphabets, digits and special case characters (each category must have their representative in the user's password). As the user still can provide a password of his own choice, it creates low stress on user's memorability. We chose to keep the alphabetic portion to the passwords

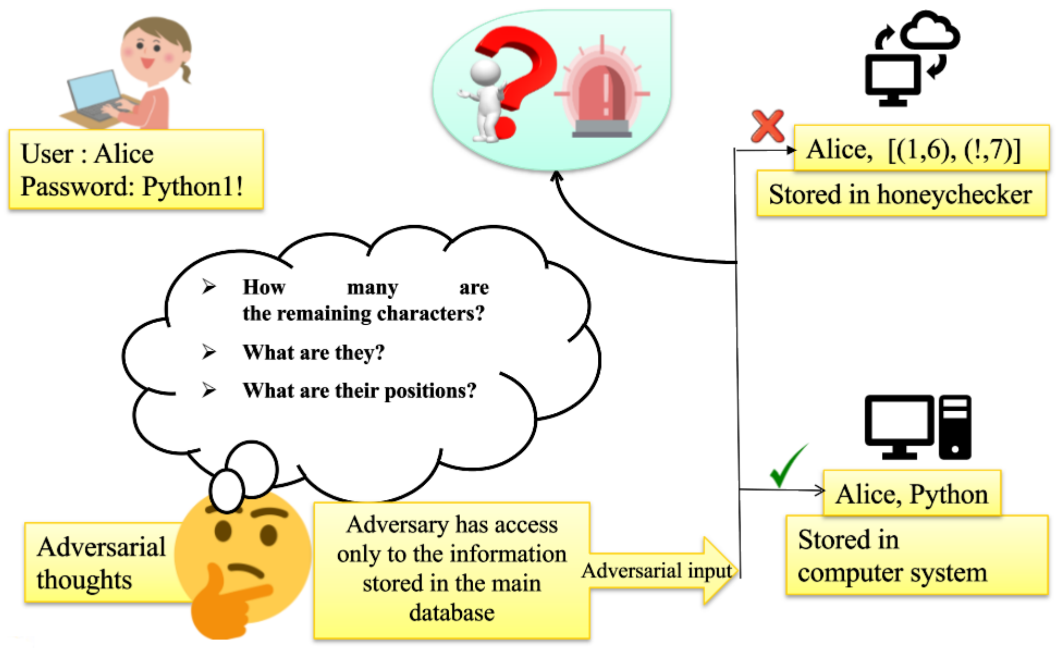

User : Alice
Password: Python1!
How many are the remaining characters?
What are they?
What are their positions?
Adversarial thoughts
Adversary has access only to the information stored in the main database
Adversarial input
Alice, [(1,6), (!,7)]
Stored in honeychecker
Alice, Python
Stored in computer system

Figure 1.1: Design Overview

file (which the adversary is likely to obtain through inverting hashes) and the portion containing a combination of digits and special characters in the positions file. Because the alphabet may form an dictionary word but the digit and special characters combination will not; hence providing security against inversion attack. The login module handles every login request to authenticate users and detect the likelihood of possible password breach. If a login attempt is made such that, against a registered username the password.char portion matches exactly (with the one stored in the passwords file) but password.digit/password.specialchar portion does not match by values or positions or both (with the ones stored in positions file), then an alarm is raised as a possible compromise of the actual user password, and the system may take actions corresponding to the system policy.

1.3 Difficulties

The proposed scheme is a very simple one both theoretically and practically. Many web system already impose different parameters on users' password choice. Hence adjoining this scheme to the existing system should not arise any difficulties. However, the systems not posing any rules on the users password choice, (thus allowing users to choose poor passwords like dictionary words, or their personal information like birth-date) will not be effectively protected by this scheme as random selection of positions-to-be-extracted may still give hints to the attacker of what the password might be; thus reducing the property of flatness.

1.4 Applications

Any authentication system, using passwords as the user-identification-token, can incorporate this technique at the back-end of their respective interface with 6(or without in some cases) slightest change in the logic as per their requirements.

1.5 Motivation

- Though honeyword based authentication provides significant assistance in detecting password breaches, almost all honeyword based techniques proposed so far have one inherent flaw, that is creating storage overhead. Reducing this storage overhead is our primary motivation behind this work.
- Apart from that, as the users tend to select weak and repetitive passwords for convenient memorizing, improving the password's strength without giving the user high stress on memory has also persuaded us in doing this work.
- We wanted to achieve the property of flatness completely i.e. probability of being the actual password should be equal for all sweetwords from the adversary point of view.
- To provide security against various attacks like Multiple System Vulnerability or MSV.

1.6 Contribution to the thesis

Driven by the aforementioned factors, we have made following contributions in this work:

- **Contribution-1:** We have introduced a password breach detection technique that fulfills the purpose of honeyword scheme with reduced storage overhead.
- **Contribution-2:** The property of flatness has been achieved.
- **Contribution-3:** The chance of MSV or intersection attack is completely removed.
- **Contribution-4:** Our introduced technique causes low stress on user's memorability with negligible system interference.
- **Contribution-5:** This technique also provides typo-safety in case of mistyping alphabets in upper/lower case due to mistiming of pressing CapsLock/Shift key.

1.7 Thesis Organization

This thesis report contains five different chapters which individually provide information regarding this whole project. The following sections are stated below-

- **Chapter 1:** In this chapter, the project is introduced by giving an overview of the system, describing the purpose and scopes of the project in present context, challenges, applications, motivation behind the work done and our contributions in this arena.
- **Chapter 2:** This chapter provides the literature review of this project which contains two sections- introduction and related literature review. The introduction section familiarizes readers with some critical terms related to this concept and the literature review specifies the pros and cons of the work done this far regarding the issue. These two sections are vital for proving the legitimacy of advancement towards the development of this problem's solution.
- **Chapter 3:** The third chapter focuses on the architecture and methodology of the proposed system. Corresponding modules along with the functions

they implement with the expected outcomes for different possible cases are discussed here.

- **Chapter 4:** This chapter studies some real case scenario according to the described ones and visualizes if the expected outcome has been achieved in each case with an analysis of the performance.
- **Chapter 5:** The fifth and last chapter covers the overall conclusion of our thoughts about this thesis project - how efficient the proposed system is in solving the arose question of security enhancement, as well as possible scopes of improvement.

1.8 Conclusion

This chapter presents an overview of the project. It also provides an overview of the proposed technique as well as the challenges in implementing this. The rationale for this work, and also the contributions made, are also mentioned here. The context and latest developments of the problem will be discussed in the following chapter.

Chapter 2

Literature Review

2.1 Introduction

In this chapter, we will shortly discuss the terminologies related to this project which are important to understand. To begin with, we first discuss what the security and usability standards mean, followed by storage overhead of honeyword based approaches. Various implementations of honeyword based models are also discussed here along with their pros and cons.

2.1.1 Security Standards

Any honeyword-based approach's security standard may be tested using three parameters: (a) Flatness, (b) Multiple System Vulnerability (MSV), and (c) DoS resiliency . These factors will be discussed next.

2.1.1.1 Flatness

Flatness corresponds to the probability of picking up the actual password from the set of sweetwords(i.e. set of honeywords along with the original password), against a username in a honeyword based approach. Perfect flatness demands that all the sweetwords must be equally probable from the adversary point of view while selecting the original user's password [2].

2.1.1.2 MSV

A honeyword generation algorithm normally creates different set of honeywords at each run for a given password. Thus for a given user password, two different systems using the same honeyword generation algorithm is likely to produce different lists of honeywords. Given the fact that these two lists of honeywords differ from each other, hence if the adversary manages to compromise both password files, then all he needs to do is to perform intersection operation to get the original password's hash [2]. This is known as MSV or Multiple System Vulnerability in honeyword generation methods.

2.1.1.3 DoS resiliency

DoS attack happens when an attacker already knows the actual password but enters honeywords intentionally to raise a false alarm in the system on the basis of which the system may disable a part of itself or all of it. Any honeyword generation scheme is said to provide moderate security against DoS attack if it can give protection against the second type of scenario [5].

2.1.2 Usability Standards

The usability standard of a honeyword generation approach may be described through three parameters namely (a) System interference, (b) Stress on memorability, and (c) Typo safety.

2.1.2.1 System interference

System interference property depicts whether the password choice of the user is influenced by the honeyword scheme. If it is not affected by the system then the user need not to remember anything other than his chosen password hence stress on memorability also becomes low [5]. But a study on real user password lists showed that the normal password choices of users are extremely poor where only 3.28% of the passwords contained both letters numbers, and only 1.20% of them contained both letters and special characters, and 50% users used 1 as a single digit in their passwords [3]. Hence, a trade-off is required between system interference rate and stress on memorability caused by it, so that moderately strong passwords are ensured.

2.1.2.2 Stress on memorability

The effect of stress on memorability is proportional to the system interference parameter. Additional information that must be remembered is a burden on the human memory. Depending on how much additional data is imposed upon user's memory, this feature can be grouped into two categories-

- *High stress on memorability*, where as part of her login credentials for n separate web accounts, the user have to recall n system produced information.
- *Low stress on memorability*, where user may remember no/single extra information of her own choice to login into n different accounts.

A good honeyword system always tries not to impose high stress on memorability to retain a standard level of usability.

2.1.2.3 Typo safety

When the honeywords are similar to the original password, an overly sensitive system may result in a false detection and block the actual user for his typing mistake while entering password, which is not desirable. Hence, typo safety forms an important criterion from the usability perspective [5].

2.2 Related Literature Review

The concept of honeyword was first introduced in [1], where the authors also described mainly 3 methods to generate honeywords.

- The first procedure is to do *chaffing-by-tweaking* in which the positions to be changed are selected and the values are randomized. The method has 2 variations-
  - Tweaking-digits: An example of this technique for t = 2 and the actual password being '18*flavors' : sweetword list = [42* flavors, 57* flavors, 18* flavors]
  - Tail-tweaking: For t = 3 and the actual password being '18*flavors', this technique may generate following sweetword list: [18*flavors, 18*flavgrn, 18*flavtcz]

  The new characters must be of the same type as the replaced ones so that the attacker cannot readily distinguish the original password from the honeywords. The disadvantage of this approach is that this method is only flat if the numbers chosen by the user are completely random. But it is not the usual case as the user prefers to choose the numbers having significance for her like birthday or any special date. Report [6] revealed that around 3.6% of hacked adobe passwords were comprised of date.

- The second method is *chaffing-with-a-password-model* which generates honeywords using a probabilistic model of real passwords. Though this method does not necessarily need the password in order to generate honeywords (unlike the previously discussed ones), and can generate honeywords of widely varying strength, yet this approach is not feasible because such a list may also be available to the attacker who could use it for honeyword identification.
- The third method is *take-a-tail* where the system proposes a random tail that is to be appended to the original password. The system thus creates honeywords that are completely flat but in this process increases system interference and stress on user's memorabilty.

Another approach to achieve flatness was described in [7] where the honeywords against a username are selected from the actual passwords of other users. But this approach has some demerits as well, because passwords of other users chosen as honeywords may be insignificant to the target user and distinguishable from his password of interest.
In [5] the authors proposed 4 methods for honeyword generation as follows-

- The first one is *modified-tail.* In this approach, the user is given a set S of length |S| from which she has to choose |S|-1 characters as the tail of her password and the system saves the password after appending the remaining character to the entered one; e.g. S=[@ < |], user chosen password: 'alex@|', sweetword list: [alex|<@ , alex@|<, alex@<|]. Though the user is given a choice on tail selection, the stress on memorability is still not satisfactory here because the provided set of characters for tail selection is chosen by the system itself. Moreover, it is a weak DoS resilient scheme.

- The second method is *caps-key* where user is asked to select m alphabets in capital if the password has n (>m) alphabets E.g. n=6, m=2, user given password: 'AniMal', sweetword list: [aNiMal . AnImal , AniMal]. This method shows less typo-safety as when the users are typing mixed case passwords, they may suffer from too soon/late press of caps lock/shift key.

- The third method is *close-number-formation.* This method is very similar to *chaffing-by- tweaking-digits* except for the fact that the numbers chosen in the honeywords are not absolutely random ones rather are much closed to the number in the actual password. E.g. user given password: 'alex1992', sweetword list: [alex1992 , alex1996 , alex1998]. Yet the password is easily guessable if the user has used a number related to her or a number related to a popular brand.

- The last method described is *pre-processing technique.* If a password has digit d (d>2), d times in it, it falls under category1 in which system generates same sets of patterns for other digits (d>2). For example, if the original password 4444, the generated sweetwords for k=5 are 333, 4444, 55555, 666666, 7777777. Category2 has d digits repeated for d1 times and system generates honeywords with other digits of d1 length e.g. 333, 444, 555, 666, 777 for d1 = 3. Mounting DoS attack is easy for this approach as maximum value of k becomes 7 for category1 and 9 for category2 which is below the standard value k=20.

All these methods along with their respective limitations have the limitation of space overhead as well, as in each case, k sweetwords are needed to be stored. Some approaches are proposed in [8] to reduce storage cost of the methods described in [5] –

- The first approach described is *storage-optimized-modified-tail* in which Fp stores the paired-distance D (clockwisely), betn chosen tail characters from a round set S provided by the system and the first character of the tail is stored in Hc.Thus the attacker is confused among all the tail combinations that can have paired distance D. The method has the same usability standard as *modified-tail*, as user has to remember a tail from system given set.

- The second method is *storage-optimized-caps-key* which is the modification of *caps-key* method formerly discussed. In this case, the user-given-password is stored after converting all characters into small case, and the positions holding upper case letters are stored in Hc; thus creating confusion among upper case letters' positions. Though this method is storage optimized but still does not provide typo-safety.

- The third method is *storage-optimized-close-number-formation* where the digits in the original user password are replaced with some other digits within a range ±r, determined by the system administrator. The system stores the distance, |D|, between the original and replaced digits and stores it in Hc. Thus the attacker gets confused among 2*r possibilities after obtaining the password file. But this approach has a major flaw as only |D| is stored. If the original password is X, then X-D and X+D both give the same distance |D| from X. Hence, login is possible in case of false password having distance |-D| as well.

- The fourth method is *storage-optimized-pre-processing*, which is the modified version of *pre-processing* technique described earlier. Here, Fp contains the category under which the digit-combination falls and Hc contains the digit that is repeated thus optimizes storage. The usability and security standard is same as before i.e. does not match the standard value of k.

In [9], Akishima et al. introduced some honeyword generation procedure namely-evolving password model, user-profile model and appending-secret model. These models, when studied, tend to reveal degraded usability standards along with making storage overhead. Another method is shown in [10] in which topological graphic matrix is constructed for generating honeywords whereas representation learning techniques is used to generate high-quality honeywords from a massive collection of unstructured data in [11]. Although the authors claim to have achieved the property of flatness, additional storage cost associated with these honeywords still remains.

An elaborate discussion on strengths and weaknesses of existing honeyword schemes has been made by the authors in [12] where the cost of storing the honeywords produced by these techniques is predominant.

2.3 Conclusion

From the above discussion, it is clear that any honeyword based approach will rank top if it can maintain the security and usability standards as well as have minimal storage overhead. Till now, the work done in this domain lack in one or another. In contrast to the previous work, a storage cost reduced password breach identification technique is introduced here which will also improve the security usability standards through slight system interference (low stress on memorability) on user's password choice so as to increase the password's strength moderately.

2.3.1 Implementation Challenges

The proposed approach is very easy to implement whether in theory or in practice. Many web systems currently place restrictions on users' password selection. As a result, integrating this approach into the existing system should be as simple as maintaining or adding new features to the system.
However, systems that do not impose any restrictions on users' password selection (allowing users to use poor passwords such as- personal information or dictionary words of small lengths) will not be effectively guarded by this scheme due to diluted flatness property, as random extraction of characters (from the password string) may still provide attackers with clues that will lead to guessing the correct password.

Chapter 3

Methodology

3.1 Introduction

The suggested system's design and working principle are discussed in this chapter. The corresponding modules, as well as the functions they perform and the predicted or expected outcomes for various scenarios, are addressed here.

3.2 Diagram/Overview of Framework

The scheme mainly work through 2 modules- registration module and login module.

- **Registration module:** This module is where the main task of security enhancement is done during the registration process of any new user. The following 3.1 figure is the basic framework of this module.
- **Login module:** In this module, the login routine of any user/non-user(mostly attacker) is monitored for detecting the possible occurrence of password breach. The architecture of this module is shown in fig. 3.2

3.3 Detailed Explanation

- **Working procedure of Registration module:** The registration module is intrigued whenever a new user tries to register into the system. The provided username by the user is checked for its existence in the registered-users list. If the given username is already taken, the want-to-be-user gets a negative response. If not, a string is taken from her as password input. If the password string fulfils the pattern requirement (imposed by the system

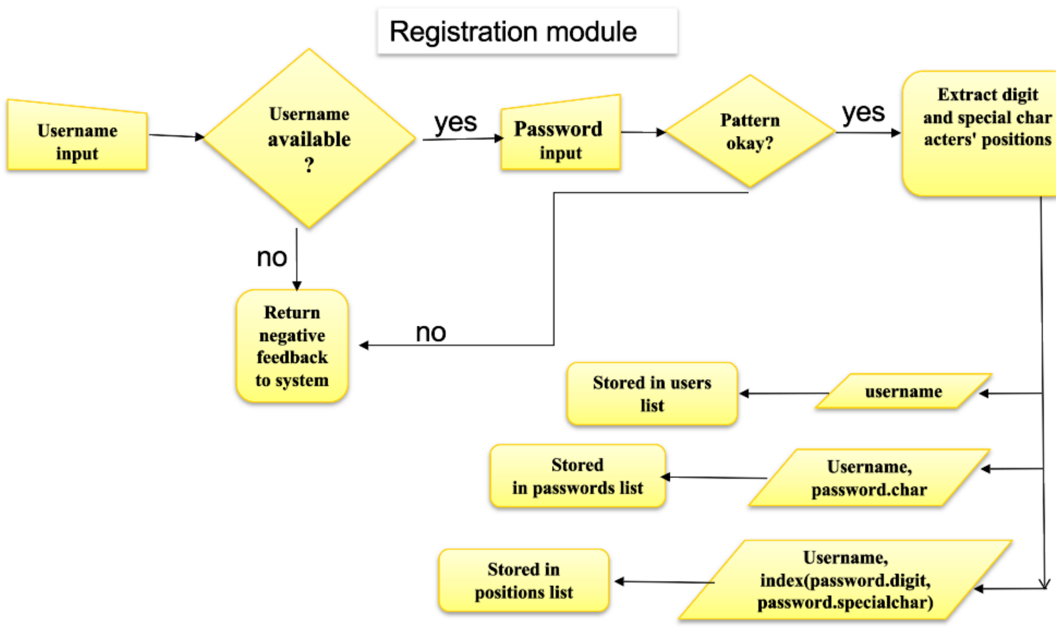


Figure 3.1: Registration Module Architecture

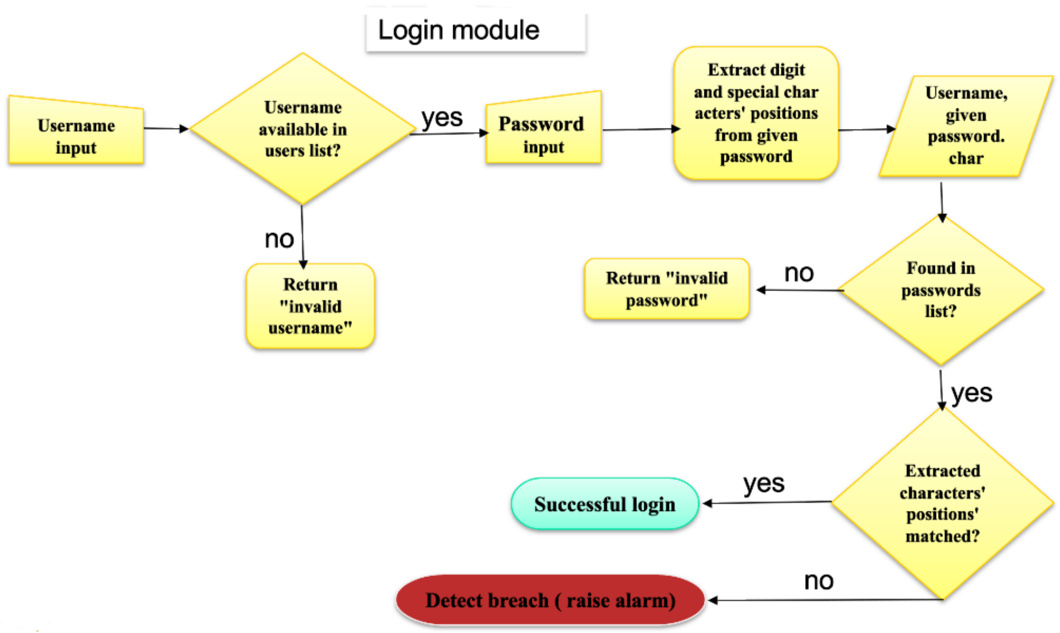

Login module
Username input
Username available in users list?
yes
Password input
Extract digit and special char acters' positions from given password
Username, given password. char
no
Return "invalid username"
Return "invalid password"
no
Found in passwords list?
yes
Successful login
yes
Extracted characters' positions' matched?
Detect breach (raise alarm)
no

Figure 3.2: Login Module Architecture

username	**password**
user1	Python1! 01234567(the positions of the characters of the given password)

Table 3.1: Given username and password

users.csv	**passwords.csv**	**positions.csv**
user1	user1, Python, h(Python)	user1, [('1',6), ('!', 7)], h([('1',6), ('!', 7)])

Table 3.2: Parsing and storing password

on user's password such as- standard guidelines for making password), the string is parsed into two portions- one in the passwords file and the other in the positions file. The passwords file contains the alphabetic part of the password(in hashed form) against the given username. And the positions file contains record of the hashes of the extracted characters values along with their respective indices. For example, if a user with userneme: user1 and password: Python1! tries to register the registration module will work as shown in table 3.1 and 3.2

- **Working procedure of Login module:** The login module authenticates the user for every login attempt against a provided username. If it is a registered one, then it is found in the users file otherwise "invalid username" is displayed as the response. For a username registered on the system, a password input is asked from the person trying to log into the system, which is passed through alphabetic-characters-extraction method to match the outcome with the one stored in the passwords file against that specific username. If not matched, the system simply response back as invalid password provided. But if it gets matched, it means that the person who is trying to log into the system is either the actual user or an adversary who has managed to compromise the passwords file, successfully cracked the hashes and is now attempting a guess attack. There is only one way to find out and that is through the final verification method i.e. extracted characters (digits and special characters) index matching. If this returns a positive result then the system identifies the login attempt to be done by a valid user; else the system takes it as a password breach occurrence and takes actions according to the system policy.

  For example, if a person with userneme: user1 and password: Python00

Given username	**Given password**
user1	Python00

Table 3.3: Given username and password

content against given username	**found in file**
username: user1	yes (in users.csv)
extracted alphabetic fragment: Python	yes (in passwords.csv)
extracted digits' and special characters' values and index in the given password: [('0',6), ('0',7)]	no (in positions.csv)

Table 3.4: Login routine for table 3.3

tries to log into the system, the login module will work as shown in table 3.3 and 3.4. As seen in table 3.4, a possible password breach will be detected by the system in this case and an alarm may be raised.

3.4 Implementation

In our implementation of the above idea, the two modules utilize the methods described below to perform specific actions.

3.4.1 create.ipynb

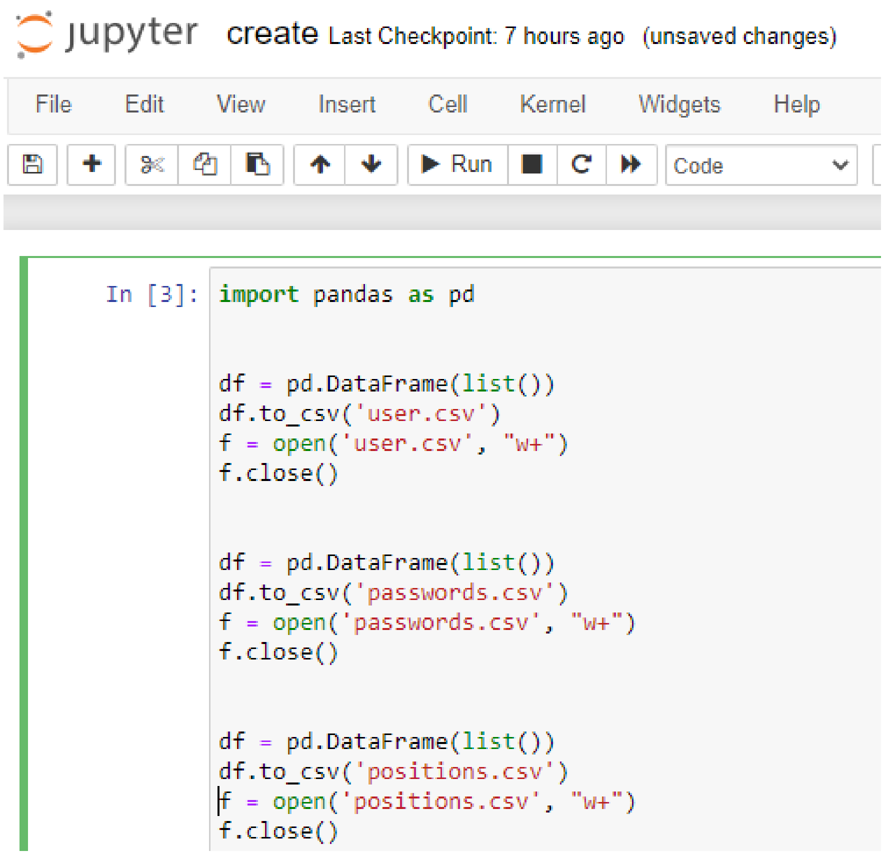
This block of code is compiled at the initial stage only for once, to generate empty users file, passwords file and positions file. The users file holds the list of users, the passwords file holds the list of alphabet-fragment of the passwords against respective usernames, and the positions file records the extracted characters values and positions against the usernames after they have registered on the system.

3.4.2 utils.py

This is a file containing some methods which are used by the registration and login module. Instead of readdressing them individually in both the modules, this file is generated to save space. The methods in this file are-

3.4.2.1 username_availability_check(username)

This method checks for the existence of a username in a file named users, which is basically a list of the registered users of the system. Based on the availability


jupyter
create Last Checkpoint: 7 hours ago (unsaved changes)
File Edit View Insert Cell Kernel Widgets Help
Run
Code
In [3]:

```
import pandas as pd


df = pd.DataFrame(list())
df.to_csv('user.csv')
f = open('user.csv', "w+")
f.close()


df = pd.DataFrame(list())
df.to_csv('passwords.csv')
f = open('passwords.csv', "w+")
f.close()


df = pd.DataFrame(list())
df.to_csv('positions.csv')
f = open('positions.csv', "w+")
f.close()
```

Figure 3.3: contents of create.ipynb

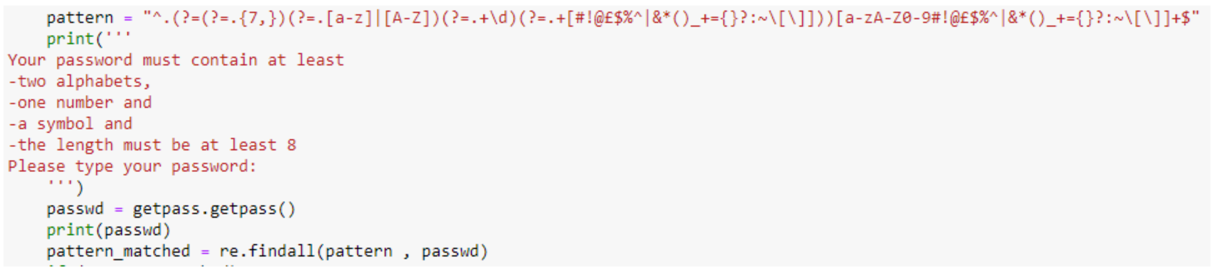
of the username-asked-for, this returns True or False(a boolean value). Both of registration and login modules use this method.

3.4.2.2 pattern check

This block of code is used by the registration module to check if the user given password satisfies the criteria set by the system. This also returns a boolean True or False to indicate positive and negative results respectively.

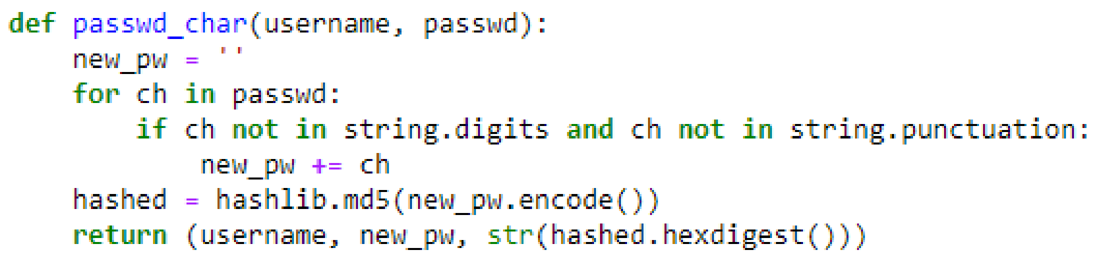
```
def username_availability_check(username):
    username_exists = False
    with open('user.csv', 'rt') as f:
        reader = csv.reader(f, delimiter=',')
        for row in reader:
            if username in row:
                username_exists = True
    return username_exists
```

Figure 3.4: Contents of username_availability_check(username)



```
    pattern = "^.(?=(?=.{7,})(?=.[a-z]|[A-Z])(?=.+\d)(?=.+[#!@£$%^|&*()_+={}?:~\[\]]))[a-zA-Z0-9#!@£$%^|&*()_+={}?:~\[\]]+$"
    print('''
Your password must contain at least
-two alphabets,
-one number and
-a symbol and
-the length must be at least 8
Please type your password:
    ''')
    passwd = getpass.getpass()
    print(passwd)
    pattern_matched = re.findall(pattern , passwd)
```

Figure 3.5: block of code for pattern check



```
def passwd_char(username, passwd):
    new_pw = ''
    for ch in passwd:
        if ch not in string.digits and ch not in string.punctuation:
            new_pw += ch
    hashed = hashlib.md5(new_pw.encode())
    return (username, new_pw, str(hashed.hexdigest()))
```

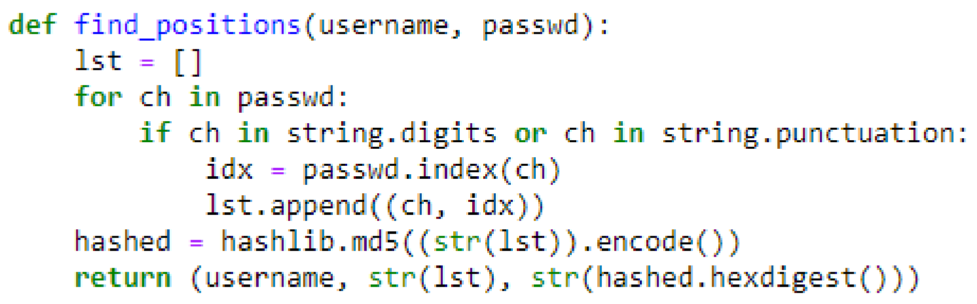
Figure 3.6: contents of passwd_char(username, passwd)

3.4.2.3 passwd_char(username, passwd)

This method, used by both registration and login modules, takes two arguments through the 'username' and 'passwd' parameters, and returns the hash of the alphabetic portion of the given password against the username.

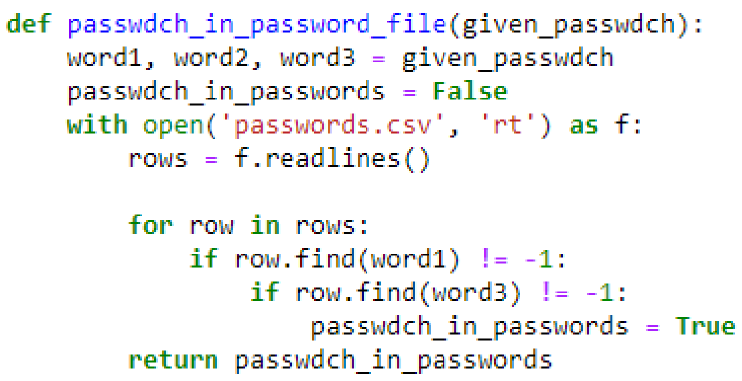
3.4.2.4 find_positions(username, passwd)

This is used by both modules as well. It extracts the positions of digits and special characters from the given password string, and returns them in hashed format with their respective positions, along with the given username as a list.



```
def find_positions(username, passwd):
    lst = []
    for ch in passwd:
        if ch in string.digits or ch in string.punctuation:
            idx = passwd.index(ch)
            lst.append((ch, idx))
    hashed = hashlib.md5((str(lst)).encode())
    return (username, str(lst), str(hashed.hexdigest()))
```

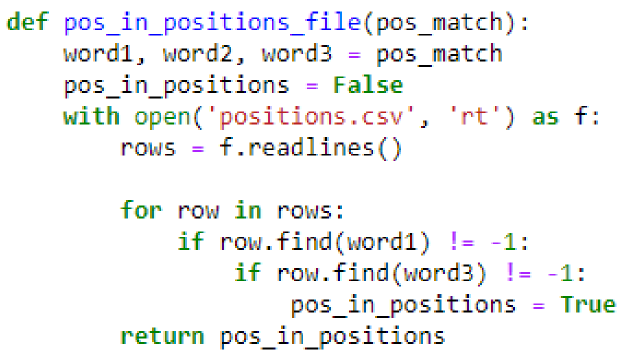
Figure 3.7: contents of find_positions(username, passwd)



```python
def passwdch_in_password_file(given_passwdch):
    word1, word2, word3 = given_passwdch
    passwdch_in_passwords = False
    with open('passwords.csv', 'rt') as f:
        rows = f.readlines()

        for row in rows:
            if row.find(word1) != -1:
                if row.find(word3) != -1:
                    passwdch_in_passwords = True
        return passwdch_in_passwords
```

Figure 3.8: contents of passwdch_in_password_file(given_passwdch)



```python
def pos_in_positions_file(pos_match):
    word1, word2, word3 = pos_match
    pos_in_positions = False
    with open('positions.csv', 'rt') as f:
        rows = f.readlines()

        for row in rows:
            if row.find(word1) != -1:
                if row.find(word3) != -1:
                    pos_in_positions = True
        return pos_in_positions
```

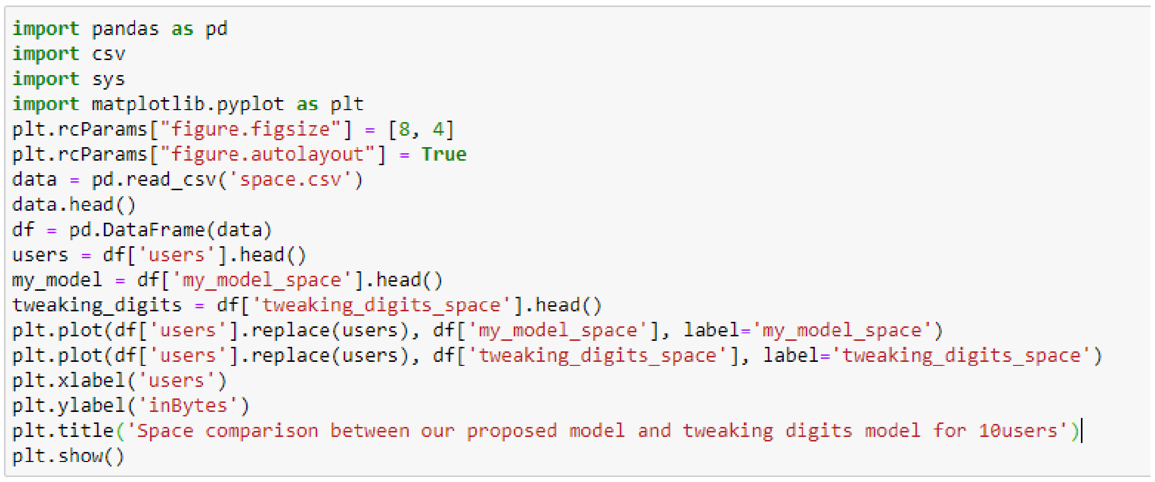
Figure 3.9: contents of pos_in_positions_file(pos_match)

3.4.2.5 passwdch_in_password_file(given_passwdch)

This method is used by the login module after it has checked the given username against a registered username and parsed the given password through passwd_char(username, passwd) method. This method checks the existence of the list returned by passwd_char(username, passwd) method in the passwords file.

3.4.2.6 pos_in_positions_file(pos_match)

The login module uses this method as the final step of verification. After passwdch_in_password_file(given_passwdch) method returns a boolean True, this method comes into action to check whether the given password is exactly same as the actual password or the password file has been compromised thus letting the attacker know the alphabetic portion of the password and perform a guess attack.



```
import pandas as pd
import csv
import sys
import matplotlib.pyplot as plt
plt.rcParams["figure.figsize"] = [8, 4]
plt.rcParams["figure.autolayout"] = True
data = pd.read_csv('space.csv')
data.head()
df = pd.DataFrame(data)
users = df['users'].head()
my_model = df['my_model_space'].head()
tweaking_digits = df['tweaking_digits_space'].head()
plt.plot(df['users'].replace(users), df['my_model_space'], label='my_model_space')
plt.plot(df['users'].replace(users), df['tweaking_digits_space'], label='tweaking_digits_space')
plt.xlabel('users')
plt.ylabel('inBytes')
plt.title('Space comparison between our proposed model and tweaking digits model for 10users')
plt.show()
```

Figure 3.10: block of code for graph plotting

3.4.2.7 Graph plotting

fig 3.10 depicts the code written to show the space comparison between our proposed model and tweaking digits model.

3.5 Conclusion

This chapter gives an insight into the proposed scheme's architecture and functioning procedural. It also includes the actual steps performed by us in order to implement the given idea. The next chapter will discuss the outcomes of the performed tasks and verify them against the predicted ones.

Chapter 4

Results and Discussions

4.1 Introduction

This chapter is all about evaluating whether our proposed scheme stood up to the expectations. At first, this chapter studies the storage cost improvement theoretically and then experimentally. Next this chapter inspects some real case scenarios and visualizes if the expected outcome has been achieved in each case with an analysis of the performance measure. The improvement of the usability and security standards are also discussed later.

4.2 Evaluation of Framework

4.2.1 Evaluation of storage cost (theoretically):

If we consider the general tweaking digits method, it has the space requirement as table 4.1. Whereas, our proposed method requires space as followed by table 4.2.

4.2.2 Evaluation of storage cost (experimentally):

Files generated for tweaking digits method where username: *user1* and password: *Python1!* (for k=5) are shown in figure 4.1, 4.2 and 4.3. Figure 4.4, 4.5 and 4.6

User no.	**Password length**	**Honeywords produced**	**Total sweetwords**	**Hash bytes required for each sweetword**	**Total space required in main database**	**Total space required in honeychecker**
1	p	k-1	k	h	k*h	c1+h
n	n*p	n*(k-1)	n*k	h	n*k*h	n*(c1+h)

Table 4.1: space required for general tweaking digits method

User no.	Honeywords produced	Stored bits of password	Removed bits of password	Hash bytes required for m	Total space required in main database	Total space required in honeychecker
1	none	m(<p)	p-m	h	h	c1+h
n	none	n*m	n*(p-m)	h	n*h(<n*k*h)	n*(c1+h)

Table 4.2: space required for our proposed method

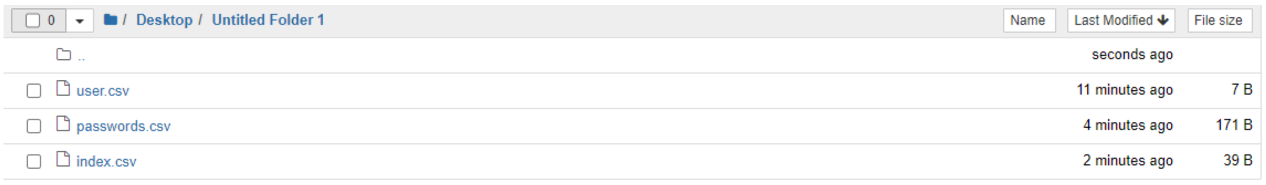

```
0 ▾  / Desktop / Untitled Folder 1                 Name   Last Modified ↓   File size
   ..                                                    seconds ago
   user.csv                                              11 minutes ago        7 B
   passwords.csv                                         4 minutes ago       171 B
   index.csv                                             2 minutes ago        39 B
```

Figure 4.1: files generated for tweaking digits method

display the files produced for the same username and password in our proposed scheme. In both case MD5 hash function has been used.

- **Calculation:**

  Now for general tweaking digit method:

  Total space required = user.csv file size + passwords.csv file size + index.csv file size = 7B + 171B + 39B = 217B

  For our proposed method:

  Total space required = user.csv file size + passwords.csv file size+ positions.csv file size = 7B +39B +39B = 85B

  Hence space overhead for 1 user = 217B - 85B = 132B (for k=5 only)

  If there is n users then space overhead n*132B (approx.)

  fig 4.7 and 4.8 illustrates the space comparison (in bytes) between our proposed model and tweaking digits model for 10user (where k= 5 in tweaking digits method)

4.2.3 Evaluation of functionality:

To justify that our implemented system works, we have checked it against some test cases.

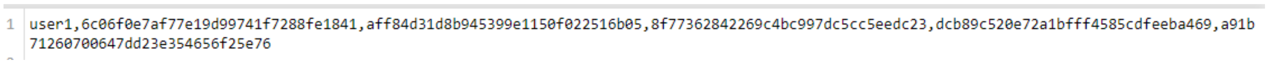
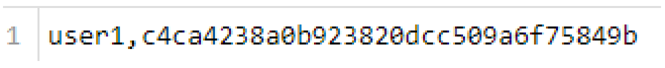
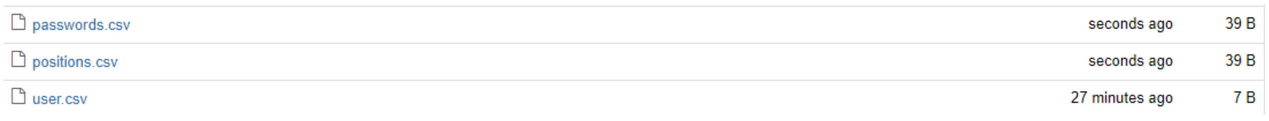

```
user1,6c06f0e7af77e19d99741f7288fe1841,aff84d31d8b945399e1150f022516b05,8f77362842269c4bc997dc5cc5eedc23,dcb89c520e72a1bfff4585cdfeeba469,a91b
71260700647dd23e354656f25e76
```

Figure 4.2: contents of passwords file in tweaking digits method



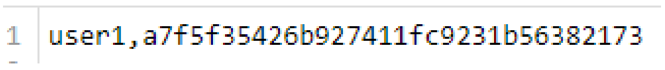
```
user1,c4ca4238a0b923820dcc509a6f75849b
```

Figure 4.3: contents of index file in tweaking digits method



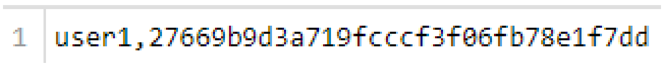
```
passwords.csv    seconds ago      39 B
positions.csv    seconds ago      39 B
user.csv         27 minutes ago    7 B
```

Figure 4.4: files generated for our proposed method



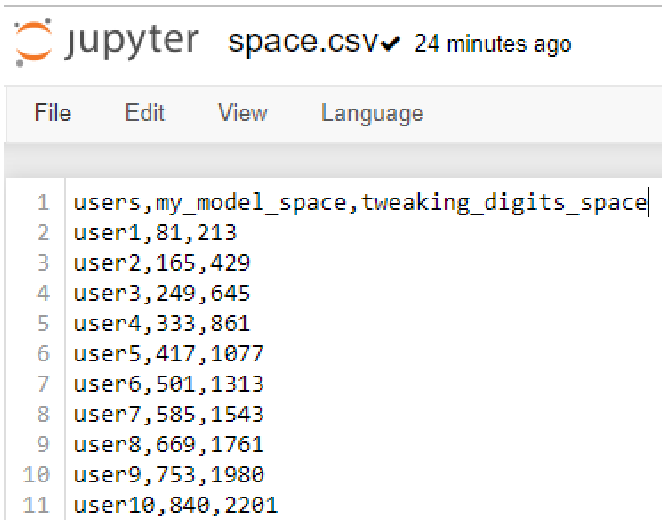
```
user1,a7f5f35426b927411fc9231b56382173
```

Figure 4.5: contents of passwords file in our proposed method



```
user1,27669b9d3a719fcccf3f06fb78e1f7dd
```

Figure 4.6: contents of positions file in our proposed method



```
users,my_model_space,tweaking_digits_space
user1,81,213
user2,165,429
user3,249,645
user4,333,861
user5,417,1077
user6,501,1313
user7,585,1543
user8,669,1761
user9,753,1980
user10,840,2201
```

Figure 4.7: contents of space.csv file

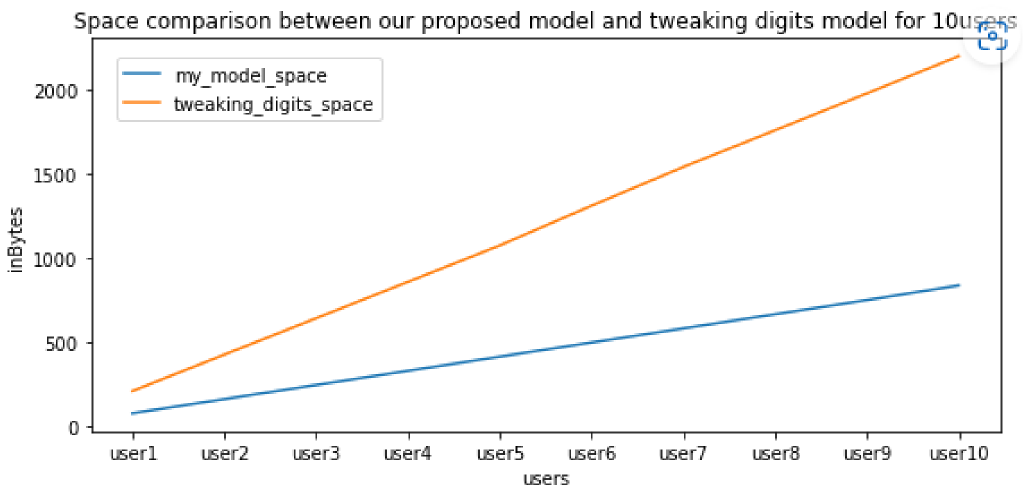
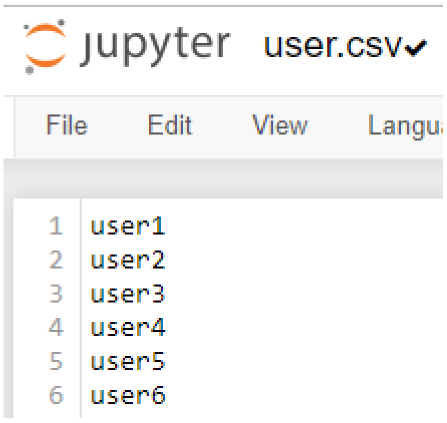
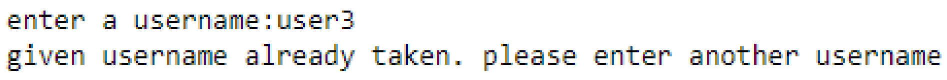


Figure 4.8: graphical representation of space required by our proposed model and tweaking digits model (k=5)



```
user1
user2
user3
user4
user5
user6
```

Figure 4.9: content of the user file while evaluating case 1 and case 2



```
enter a username:user3
given username already taken. please enter another username
```

Figure 4.10: response by system on case 1 registration routine

4.2.3.1 Functionality testing of registration module:

While trying to get a new user registered onto the system the following cases can happen and the result is checked against the expected outcomes.

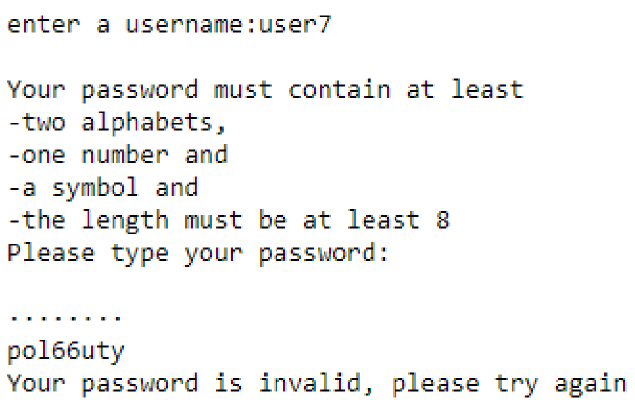
- **case-1** Invalid username: If an input username is given as such it already exists in the user file then the system should response as "the username is already taken". We tried to give an username 'user3' (which was already in the registered users list) in the registration process(figure 4.7)

  result: Username already taken message displayed in the screen.(fig 4.8)

- **case-2** Invalid password pattern: If an input password is given as such it does not satisfies the pattern imposed by the system on passwords, then the system should response as "invalid password". We tried to register with a given username 'user4' and passed a password value that is not in compliance with the pattern.

  result: Invalid password message displayed in the screen.(fig 4.9)

- **case-3** Valid username and valid password pattern: If an username is given that is unique (is not in the users list) and the password provided also

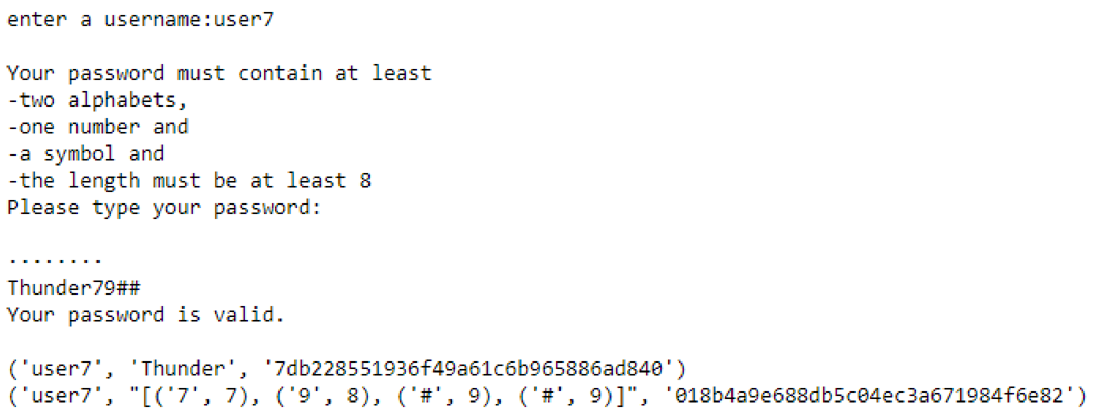


```
enter a username:user7

Your password must contain at least
-two alphabets,
-one number and
-a symbol and
-the length must be at least 8
Please type your password:

........
pol66uty
Your password is invalid, please try again
```

Figure 4.11: response by system on case 2 registration routine



```
enter a username:user7

Your password must contain at least
-two alphabets,
-one number and
-a symbol and
-the length must be at least 8
Please type your password:

........
Thunder79##
Your password is valid.

('user7', 'Thunder', '7db228551936f49a61c6b965886ad840')
('user7', "[('7', 7), ('9', 8), ('#', 9), ('#', 9)]", '018b4a9e688db5c04ec3a671984f6e82')
```

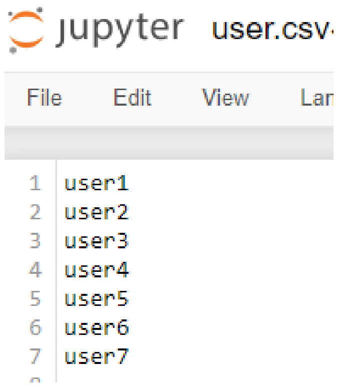
Figure 4.12: a new user having username = 'user7' is registered onto the system (case 3 registration routine)

satisfies the pattern then a new user should be created and the results produced by the corresponding methods should be saved in the respective files.

result: A new user 'user4' is created. username = 'user7' saved in the user.csv file. [username, h(password.char)] saved in the passwords file. [username, h(password.digits/password.schar , index)] is saved in the positions file. N.B. here the non-hashed versions are also shown to facilitate better understanding. (fig 4.10)

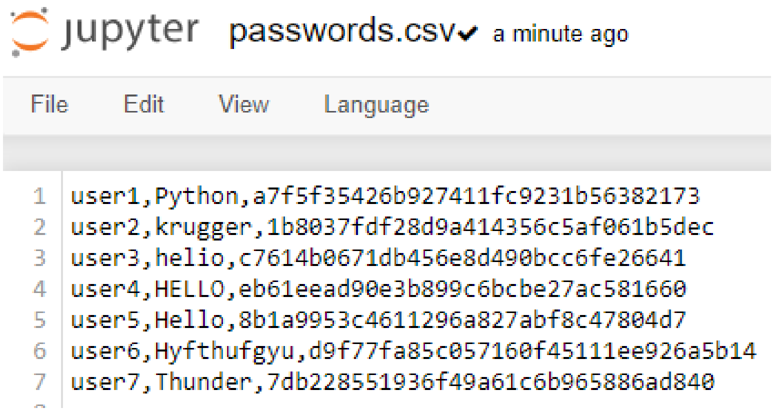
4.2.3.2 Inspecting a login routine:

While trying to check a login attempt made by a user/non-user into the system the following cases can happen and the results are checked against the expected outcomes.



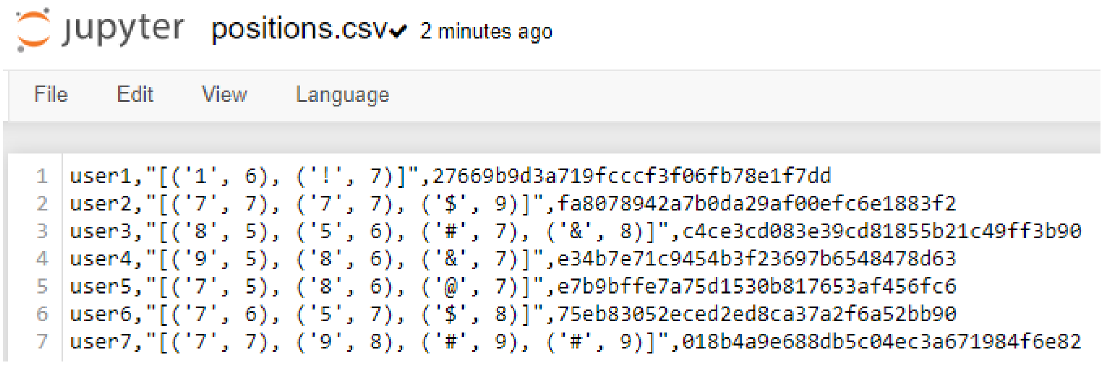
```
user1
user2
user3
user4
user5
user6
user7
```

Figure 4.13: updated user file(case 3 registration routine)



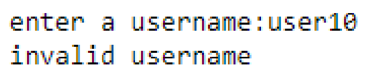
```
user1,Python,a7f5f35426b927411fc9231b56382173
user2,krugger,1b8037fdf28d9a414356c5af061b5dec
user3,helio,c7614b0671db456e8d490bcc6fe26641
user4,HELLO,eb61eead90e3b899c6bcbe27ac581660
user5,Hello,8b1a9953c4611296a827abf8c47804d7
user6,Hyfthufgyu,d9f77fa85c057160f45111ee926a5b14
user7,Thunder,7db228551936f49a61c6b965886ad840
```

Figure 4.14: updated passwords file(case 3 registration routine)



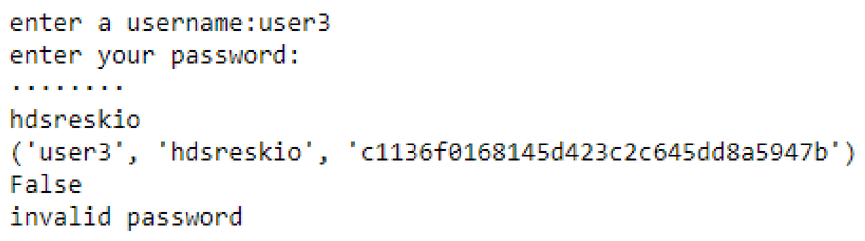
```
user1,"[('1', 6), ('!', 7)]",27669b9d3a719fcccf3f06fb78e1f7dd
user2,"[('7', 7), ('7', 7), ('$', 9)]",fa8078942a7b0da29af00efc6e1883f2
user3,"[('8', 5), ('5', 6), ('#', 7), ('&', 8)]",c4ce3cd083e39cd81855b21c49ff3b90
user4,"[('9', 5), ('8', 6), ('&', 7)]",e34b7e71c9454b3f23697b6548478d63
user5,"[('7', 5), ('8', 6), ('@', 7)]",e7b9bffe7a75d1530b817653af456fc6
user6,"[('7', 6), ('5', 7), ('$', 8)]",75eb83052eced2ed8ca37a2f6a52bb90
user7,"[('7', 7), ('9', 8), ('#', 9), ('#', 9)]",018b4a9e688db5c04ec3a671984f6e82
```

Figure 4.15: updated positions file(case 3 registration routine)



```
enter a username:user10
invalid username
```

Figure 4.16: system response on case 1 (login routine)



```
enter a username:user3
enter your password:
........
hdsreskio
('user3', 'hdsreskio', 'c1136f0168145d423c2c645dd8a5947b')
False
invalid password
```

Figure 4.17: system response on case 2 (login routine)

- **case-1** Invalid username: If an input username is given as such it does not exist in the user file then the system should response as "invalid username". We tried to give an username 'user10' (which is not a registered username).

  result: Invalid username message displayed in the screen.(fig 4.14)

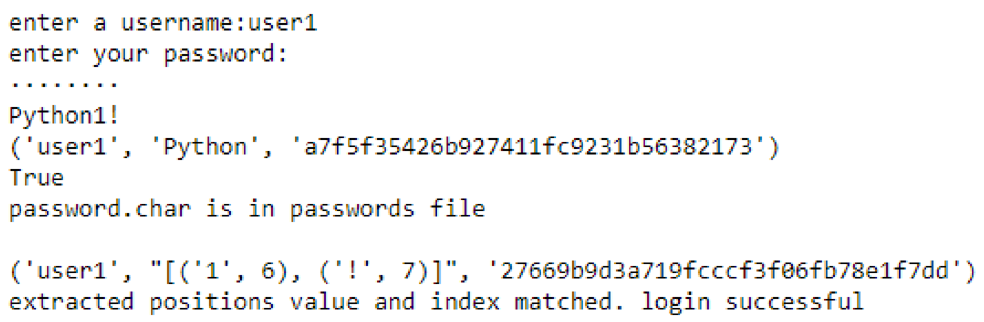
- **case-2** Invalid password: If an input password against a registered username is given as such password.char portion of the given password does not exist in the passwords file then the system should response as "invalid password". We tried to give a wrong password.char against a registered user 'user3' .

  result: Invalid password message displayed in the screen.(fig 4.15)

- **case-3** Whole password input matches with the stored one against a given username: If an input password against a registered username is given as such password.char portion of the given password exists in the passwords file and the values and positions of the extracted characters also match, then the system should response as "login successful". We tried to give a registered username and password .

  result: login successful message displayed in the screen.(fig 4.16)

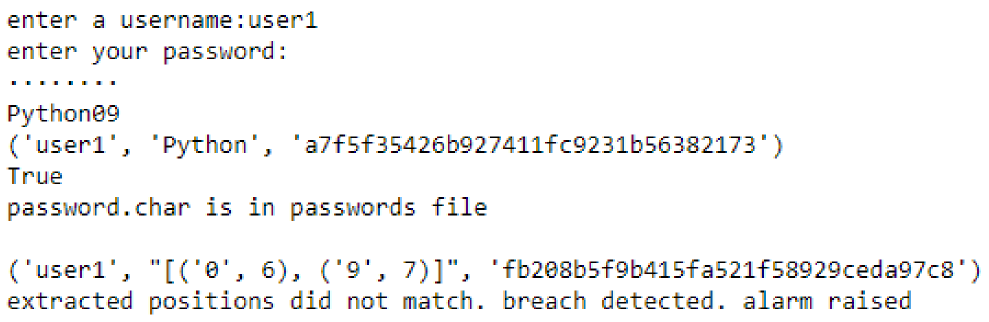
- **case-4** password.char portion of the given password matches with the stored one against a given username, but extracted characters values and indices don't: If an input password against a registered username is given as such password.char portion of the given password exists in the passwords file but



```
enter a username:user1
enter your password:
........
Python1!
('user1', 'Python', 'a7f5f35426b927411fc9231b56382173')
True
password.char is in passwords file

('user1', "[('1', 6), ('!', 7)]", '27669b9d3a719fcccf3f06fb78e1f7dd')
extracted positions value and index matched. login successful
```

Figure 4.18: system response on case 3 (login routine)



```
enter a username:user1
enter your password:
........
Python09
('user1', 'Python', 'a7f5f35426b927411fc9231b56382173')
True
password.char is in passwords file

('user1', "[('0', 6), ('9', 7)]", 'fb208b5f9b415fa521f58929ceda97c8')
extracted positions did not match. breach detected. alarm raised
```

Figure 4.19: breach detected message displayed in the screen(case 4 login routine)

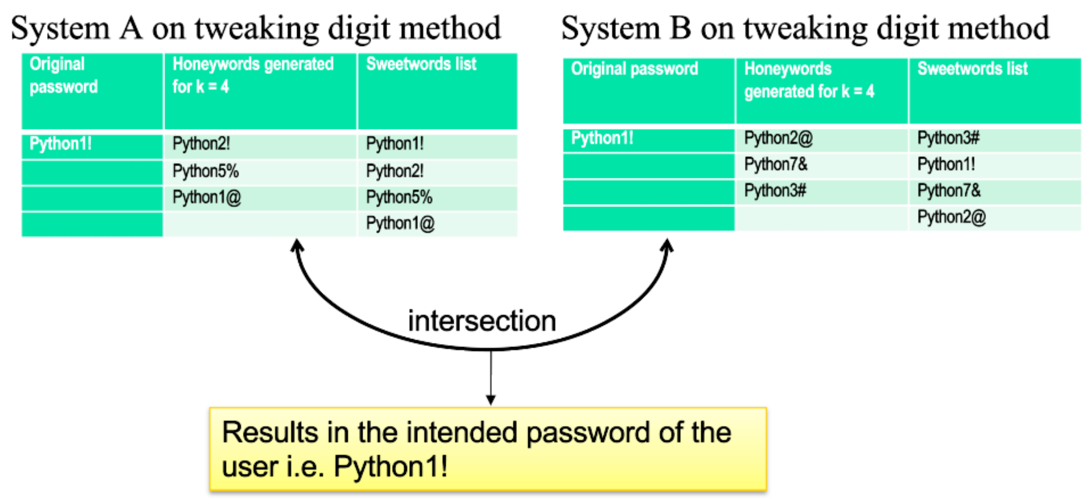
the values and positions of the extracted characters does not match with the one stored in the positions file against the given username, then the system should detect it as possible compromise of the users' passwords list and response as per system policy. We tried to give a registered username and password as such password.char matches but the extracted characters positions and values do not.

result: breach detected message displayed in the screen.(fig 4.17)

4.2.4 Evaluation of usability and security standards:

4.2.4.1 Resistance against MSV attack:

Two systems, both following tweaking digit method for generating sweetword list, are likely to produce different set of honeywords. Thus if an adversary manages to hack both password files and intersect them then he can get the actual password which is shown in fig 4.18 . On the other hand, our system does not generate or require any honeywords. The adversary, even if manages to compromise the password files, gets only a fraction of the actual password. Thus our proposed

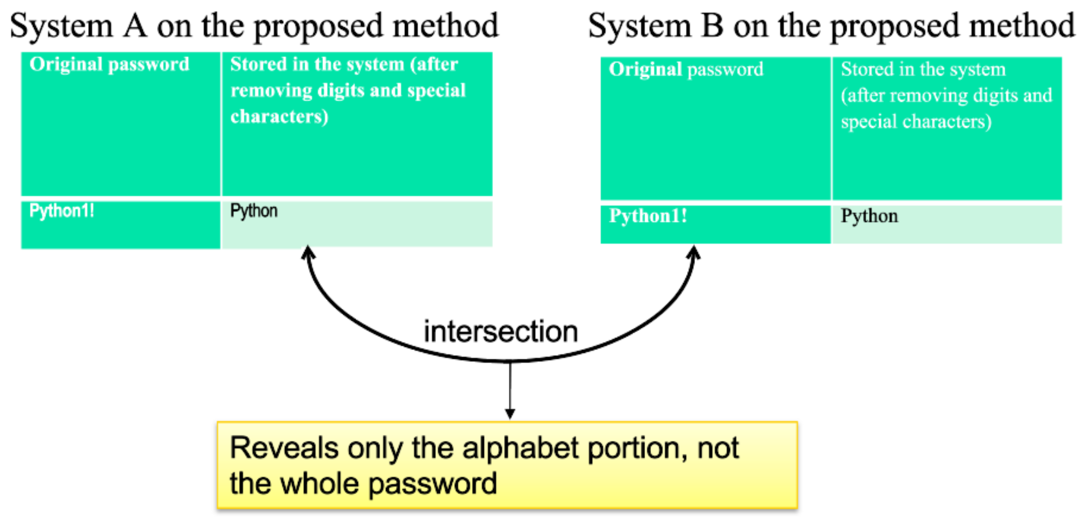

System A on tweaking digit method

Original password	Honeywords generated for k = 4	Sweetwords list
Python1!	Python2!	Python1!
	Python5%	Python2!
	Python1@	Python5%
		Python1@

System B on tweaking digit method

Original password	Honeywords generated for k = 4	Sweetwords list
Python1!	Python2@	Python3#
	Python7&	Python1!
	Python3#	Python7&
		Python2@

Figure 4.20: MSV attack in tweaking digits method


System A on the proposed method

Original password	Stored in the system (after removing digits and special characters)
Python1!	Python

System B on the proposed method

Original password	Stored in the system (after removing digits and special characters)
Python1!	Python

Figure 4.21: MSV attack in our proposed method

method provides complete security against MSV attack.(fig 4.19)

4.2.4.2 Achieving flatness:

For similar reason, the likelihood of stand out as the real password to the adversary is minimized as the extracted characters values can have any digit/special-character value and their indices are also arbitrary to the attacker.(fig 1.1)

4.2.4.3 System interference:

In our proposed system the user is not given any set of password altercation choices. The user only need to follow general guidelines of forming password. Hence there is no system interference whatsoever.

4.2.4.4 Stress on memorability:

No system interference leads to low stress on user's memorability.

4.2.4.5 Typo-safety:

Over sensitivity to typo is reduced in the proposed method as alarm is only triggered if the alphabet portion matches exactly (all cases of all letters) but not the rest of the characters (digit and special character values or positions).(fig 1.1)

4.2.5 Social and Ethical Impact

As password is a crucial component in authenticating an user, it should be well preserved for preserving the confidentiality of personnel information and to prevent crimes like identity theft. Although it was previously achieved by the complex computing hash functions (which returned a unique irreversible hash value for any given string), recent studies have shown that hash values are not irreversible anymore. Hence to add extra security to the users' passwords, a concept of storing dummy passwords with the actual password was introduced, in order to confuse the attacker among the values and detect a guess attack. But the honeyword generation technique does not always come with standard usability and security features and also causes much storage overhead. Our proposed scheme reduces this storage overhead with improved security and usability. And it is also easy to understand and integrate with the existing methods. Hence when adopted, it should give a layer of extra defense against the attacker.

4.3 Conclusion

From the above discussion, it can be said that the proposed approach might actually be a good one to be adopted.

Chapter 5

Conclusion

5.1 Conclusion

This approach is not terribly deep, but it should be quite effective, as it puts the adversary at risk of being detected with every login attempt using a password obtained by brute-force-solving of a hashed password file that does not even contain the whole password; which perfectly obtains the purpose of honeywords i.e. confusing and trapping the attacker, making it a high-risk guessing game. At the same time, it causes much less storage overhead than the original honeyword generation techniques with improved security and usability standards in many cases. Thus, this scheme when adopted can provide a very useful layer of defense in case the hashed passwords have been cracked by an adversary.

5.2 Future Work

While this method of password breach detection is somewhat effective in protecting the secrecy of users' passwords, this is not a replacement to a strong password policy and user awareness. However, there is much scope of improvement as such-

- How this system can best be designed to withstand active attacks, e.g., code modification (of the computer system or the honeychecker).
- How to provide more DoS resistivity without diluting the usability standards.
- How user-friendly this approach actually is in practice.

References

[1] A. Juels, 'Honeywords : Making Password-Cracking Detectable. In Proceedings of the 2013 ACM SIGSAC conference on Computer communications security, pages 145–160. ACM, 2013.,' pp. 0–19, 2013 (cit. on pp. 1, 10).

[2] J. Brainard, A. Juels, R. L. Rivest and M. Szydlo, 'Fourth-Factor Authentication : Somebody You Know Categories and Subject Descriptors,' *Proceedings of the 13th ACM conference on Computer and communications security*, pp. 168–178, 2006 (cit. on pp. 2, 8, 9).

[3] M. Weir, S. Aggarwal, B. De Medeiros and B. Glodek, 'Password cracking using probabilistic context-free grammars,' *Proceedings - IEEE Symposium on Security and Privacy*, pp. 391–405, 2009, ISSN: 10816011. DOI: 10.1109/SP.2009.8 (cit. on pp. 2, 9).

[4] J. Ma, W. Yang, M. Luo and N. Li, 'A study of probabilistic password models,' *Proceedings - IEEE Symposium on Security and Privacy*, pp. 689–704, 2014, ISSN: 10816011. DOI: 10.1109/SP.2014.50 (cit. on p. 2).

[5] N. Chakraborty and S. Mondal, 'Few notes towards making honeyword system more secure and usable,' *ACM International Conference Proceeding Series*, vol. 08-10-Sep-, 2015. DOI: 10.1145/2799979.2799992 (cit. on pp. 9–12).

[6] *The Pathetic Reality of Adobe Password Hints.* [Online]. Available: https://xato.net/the-pathetic-reality-of-adobe-password-hints-bb40fd92220f (cit. on p. 11).

[7] S. Pawar, S. Dhoble, N. Soni, S. Bhosale and I. Technology, 'International Journal of Advance Engineering and Research Achieving Flatness : Selecting the Honeywords from Existing User Passwords,' pp. 174–178, 2017 (cit. on p. 11).

[8] N. Chakraborty and S. Mondal, 'Towards Improving Storage Cost and Security Features of Honeyword Based Approaches,' *Procedia Computer Science*, vol. 93, no. September, pp. 799–807, 2016, ISSN: 18770509. DOI: 10.1016/j.procs.2016.07.298. [Online]. Available: http://dx.doi.org/10.1016/j.procs.2016.07.298 (cit. on p. 12).

[9] A. Akshima, D. Chang, A. Goel, S. Mishra and S. Sanadhya, 'Generation of secure and reliable honeywords, preventing false detection,' *IEEE Transactions on Dependable and Secure Computing*, vol. PP, pp. 1–1, Apr. 2018. DOI: 10.1109/TDSC.2018.2824323 (cit. on p. 13).

[10] Y. Tian, L. Li, H. Peng and Y. Yang, 'Achieving flatness: Graph labeling can generate graphical honeywords,' *Computers Security*, vol. 104, p. 102 212, 2021, ISSN: 0167-4048. DOI: https://doi.org/10.1016/j.cose.2021.

102212. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167404821000365 (cit. on p. 13).

[11] A. Dionysiou, V. Vassiliades and E. Athanasopoulos, 'Honeygen: Generating honeywords using representation learning,' in *Proceedings of the 2021 ACM Asia Conference on Computer and Communications Security*, ser. ASIA CCS '21, Virtual Event, Hong Kong: Association for Computing Machinery, 2021, pp. 265–279, ISBN: 9781450382878. DOI: 10.1145/3433210.3453092. [Online]. Available: https://doi.org/10.1145/3433210.3453092 (cit. on p. 13).

[12] Y. A. Yasser, A. T. Sadiq and W. AlHamdani, 'A scrutiny of honeyword generation methods: Remarks on strengths and weaknesses points,' *Cybern. Inf. Technol.*, vol. 22, no. 2, pp. 3–25, Jun. 2022, ISSN: 1314-4081. DOI: 10.2478/cait-2022-0013. [Online]. Available: https://doi.org/10.2478/cait-2022-0013 (cit. on p. 14).