Enhancing Security in a Password Manager Using RSA Encryption

Zaki Yudhistira Candra - 13522031[1]
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jl. Ganesha 10 Bandung 40132, Indonesia
[1]13522031@itb.ac.id

***Abstract*—In this digital age, the use of passwords is deeply engrained in our digital systems as they serve as a primary means to prove our identity upon gaining access to personal accounts, sensitive information, and online services. People with a high-level of digital involvement tend to have multiple accounts, using the same password for each account would lead to vulnerabilities, thus a variety of unique passwords is needed. A password manager is necessary in order to keep track of those different sets of passwords. Traditionally, password managers store passwords as plaintext in a text file, making them susceptible to hackers. This paper proposes the integration of RSA Encryption into a password manager's database. In any case, when a hacker got hold of a password manager's database, the information retrieved would be encrypted, rendering it useless to the perpetrator.**

***Keywords*—Encryption, Number Theory, Password, RSA Algorithm.**

I. Introduction

In recent years, the rapid digitalization of the world is linear to the involvement of people in the digital landscape. This has led to the common practice of a person owning multiple accounts from multiple online services. The leisure of using a single password for all accounts comes with a significant trade-off; these accounts are highly inclined to information leaks and even potential theft. With those issues in mind, a unique password for each account is necessary for security concerns. However, memorizing a password for each account could be troublesome, especially when managing numerous accounts.

Considering the urgency of improving online security, the adoption of a password manager application becomes crucial in simplifying the task of managing multiple unique passwords. By using the password manager, a person could create a list of two elements, one containing the account name and the other the password to that particular account. If the person requires a password upon logging into a specific account, the corresponding password to that account would be returned, saving the hassle of memorizing it.

There are multiple password managing apps that are available at this moment of time, but since those apps manages your private data, there are no guarantee for the data's safety and secrecy, hence it is highly advised to create our own tailored password manager software. This could ensure our data's safety and prevent any leakage.

A general password manager software is equipped with password storing mechanism and a database that consists of passwords and their corresponding accounts. A password manager serves as a storage in case a user needs to manage multiple accounts.

This paper is all about diving into the world of password management, specifically looking at how we can make it more secure by using RSA (Rivest–Shamir–Adleman) encryption. RSA is a big deal in the encryption world, known for its super-strong protection of data[3]. By adding RSA encryption to the mix, we want to fix the weak points in how passwords are usually stored which we have previously explained.

We will explore what RSA encryption is all about and its practical implementation in the realm of password management and what it means for making things more secure. We'll also delve into the advantages and the shortcomings, making sure we find a good balance between super-strong security and keeping things easy for people to use.

As we dig into making our digital identities safer, this research aims to give useful information about password management and keeping things secure online. The goal is to use RSA encryption to make password keepers stronger, helping people feel more confident and secure in the digital world.

II. Fundamental Theorem

I. *Number Theory*

Number theory is a branch of pure mathematics that deals with the properties and relationships of numbers, particularly integers. It has a long and rich history, dating back to ancient times, and has applications in various areas of mathematics and computer science.

There are several key concepts and theories in this mathematical branch that would be used in this research which will be discussed in the next section[1].

II. *Integers*

An integer is a whole number that can be positive, negative, or zero. It does not include fractions or decimals. The set of integers is denoted by $\mathbb{Z}$. Examples of integers are -5, 1, 5, 8, 97, and 3,043. Integers can be represented on a number line, and arithmetic operations such as addition, subtraction, multiplication, and division. Numbers such as $\sqrt{5}$, π, and 2.25 are not integers[1].

III. *Integers Division*

Suppose a and b are two integers that satisfy $a \neq 0$. a is said to divide b if there exists an integer c such that $b = ac$, or it can be denoted in mathematical notation as follows:

$$a \mid b, \text{ if } b = ac, c \in \mathbb{Z}, a \neq 0$$

From that theory, emerged an algorithm that utilizes the divisibility characteristic of integers named *Euclidian Algorithm*. Suppose *m* and *n* are two integers that satisfy n > 0. If *m* is divided by *n*, then the division result is *q* and the rest *r* so that:

$$m = nq + r \text{ with } 0 \leq r < n$$

IV. *Greatest Common Divisor*

Suppose a and b are two non-zero integers. The greatest common divisor (GCD) of a and b is the largest integer d such that:

$$d \mid a \text{ and } d \mid b$$

Then the GCD(a, b) = d is obtained. In addition, using the Euclidian, suppose m n, and q are two integers with the condition n > 0 such that:

$$m = nq + r \text{ with } 0 \leq r < n$$

Then the GCD(*m*,*n*) = GCD(*n*,*r*)

V. *Modulo Arithmetic*

The modulo operation calculates the remainder when a number is divided by another. It is represented as the *mod* operator and is used to find the "leftover" value after performing integer division.

Suppose a and m are two integers with m > 0, the "*mod*" operation returns the remainder of *a* (*r*) divided by *m* as written below:

$$a \; mod \; m = r$$

It is interpreted as *a* modulo *m* such that *a* = *qm* + *r*, with 0 ≤ *r* < *m*. *m* is called *modulus* or *modulo* and the arithmetic result of *modulo m* is located in the set {0,1,2,…, *m* – 1}.

VI. *Congruences*

Congruency describes a condition where two integers have the same remainder when both of them are divided by a certain non-zero integer.

Let a, b, and c be a non-negative integer and c ≠ 0. It is said that a is congruence to b *mod* c if the following situation is satisfied: $a = nc + r, b = mc + r$.

Congruency is denoted in this mathematical notation:

$$a \equiv b \text{ (mod c)}$$

.

VII. *Prime Number*

Let p be a positive integer that satisfies p > 1. It is concluded that p is a prime number if and only if it could be only divided by 1 and p. An example is the number 23.

Since a prime number has to be greater than 1, then the prime series starts from 2, 3, 5, 7, 11, 13, … and so on.

Fermat theorem is a prime related theorem that states that if *p* is a prime number and there exists an integer a such that GCD(*p*,*a*) = 1, then this rules applies:

$$a^{p-1} \equiv 1 \text{ (mod p)}$$

VIII. *Relatively Prime*

Let *a* and *b* be two positive integers. It is said that *a* and *b* is relatively prime or coprime if and only if their greatest common divisor is 1.

If a and b is relatively prime, there would exist such integers *m* and *n* that satisfies the following condition:

$$am + bn = 1$$

IX. *Modulus Inverse*

If *a* and *m* are coprime, and m > 1, then there exists an inverse of *a* (mod *m*), meaning there exists an integer *x* that satisfies the condition:

$$ax \equiv 1(\text{mod } m)$$

or it can be expressed in the notation a^{-1} (mod *m*) = *x*. The method for determining the value of the modulo inverse is by solving it in the 'equal to' notation as follows:

$$ax = 1 + km \rightarrow x = \frac{1 + km}{a}$$

Then, try values for k = 0, 1, 2, … and k = -1, -2, -3, … with the solutions being all integers that satisfy the equation.

X. *Cryptography*

Cryptography, or cryptology, is the practice and study of techniques for secure communication in the presence of adversarial behavior. It involves constructing and analyzing protocols that prevent third parties or the public from reading private messages. Cryptography is derived from mathematical concepts and algorithms, and it is used to transform messages in ways that are hard to decipher. This field is crucial for ensuring data privacy, secure online communication, and protection against various types of cyberattacks. Modern cryptography encompasses various objectives, including confidentiality, integrity, and non-repudiation. It is used in numerous practical applications such as electronic commerce, chip-based payment cards, digital currencies, computer passwords, and military communications.

Cryptography can be categorized into three types: secret key cryptography, public key cryptography, and hash function cryptography. Examples of cryptographic algorithms include the Rivest-Shamir-Adleman (RSA) algorithm and the Advanced Encryption Standard (AES).

In essence, cryptography is mainly divided in two process, *encryption* and *decryption*.

Encryption is the process of transforming plaintext information into a coded form known as ciphertext, while decryption is the reverse operation, converting ciphertext back into its original plaintext form.

Both processes will require a key to generate its own unique cyphered text and to decode a specific cyphered text. The key often has 2 values, a public-key and private-key, the public key is used for encryption, hence everybody could use it, whereas the private key is used for decryption and it is kept secret.

XI. ASCII Text Encoding

ASCII, or the American Standard Code for Information Interchange, is a 7-bit character encoding standard used for electronic communication. It comprises 128 code points, with 95 being printable characters such as letters, digits, and punctuation.

ASCII served as the primary character encoding for data processing, providing a universally accepted set for basic data communications. While still in use, modern systems often prefer Unicode, which encompasses ASCII and supports a broader range of characters for global communication. Unicode is backward-compatible with ASCII, ensuring a smooth transition between the two standards[4].

XII. RSA Cryptography Algorithm

The RSA algorithm, named after its creators Ron Rivest, Adi Shamir, and Leonard Adleman, was introduced in 1977. It functions as an asymmetric cryptographic algorithm, employing public and private key pairs for data encryption and decryption. Based on modular arithmetic, the algorithm relies on the challenge of factoring large numbers into their prime components. It is an asymmetric cryptography algorithm[3].

Here's how the RSA algorithm operates:

1. Key Generation: The user generates two large prime numbers, typically more than 200 digits, p and q, and calculates n as their product, n is not hidden.
2. Public Key: A public key e, that is relatively prime with $\varphi(n)$. GCD(e, φ(n)) = 1.
3. Private Key: A private key, represented by a number d satisfying $de \equiv 1 \pmod{\varphi(n)}$, is kept for decrypting data. $\varphi(n)$ is a number that is relatively prime with p and q and is kept hidden.
4. Encryption: The sender uses the public key (n and e) to compute a ciphertext by raising the message m to the power of e and dividing by n.
5. Decryption: The receiver uses the private key (d) to compute the plaintext by raising the ciphertext to the power of d and dividing by n.

This algorithm is one of our most powerful cryptography algorithm to date since there are no efficient ways of calculating the factor of n, n is a very large prime number.

The value of n is open for public and same goes for the public key (e). On the other hand, φ(n), p, q, and the private key is kept hidden .

The encryption procedure is as follow:

Define a plain-text in a string of its corresponding ascii values and divide those ascii text into smaller blocks (p_n), for example a 3 characters block. Then encrypt each block into a value using the public key with the following formula:

$$c_n = p_n^e \bmod n, \text{ with } c_n \text{ as the encrypted block}$$

To decrypt the cyphered text, it should follow this formula:

$$p_n = c_n^d \bmod n$$

The value n indicates the block index in the target plain-text.

III. Problem Analysis

A. Password manager implementation

A simple password manager program implementation would take form in a command line interface (CLI) manner and built using the python programming language. To operate and function as a proper password manager, it would need to have some baseline features as follows:

1. Store accounts and its corresponding password

   Users could register an account and its password into the software which will be later be saved in the software's database.
2. Delete an account from the database

   If a user's account is not in use anymore, a user could easily delete it to improve the program's readability and save storage.
3. Save registered accounts into a database and have multiple databases

   The registered account and its password would be saved in a database in a form of encrypted plain-text, for the sake of simplicity, the database will be stored in a local folder.
4. Encrypt database

   Before saving the registered accounts into the database, the texts that hold such information in the memory will firstly be encrypted using the RSA Algorithm. Each database will have its own public key and private key for the encryption-decryption process.

The program would have a structure as follows:

```
Folder
|----src
|       |-main.py
|       |-encryption.py
|       |-extra.py
|       |-passwordManager.py
|
|----files
|       |----database1
|       |       |-AccData
|       |       |-Key
|       |----database2
|               |-AccData
|               |-Key
|
|----run.bat
```

Figure 1. Program structure (Source: writer's Archive)

B. RSA Database Encryption

The plain-text encryption would have a process implementation as follow:

1. User registers a particular account
2. The accounts would be saved in the system's memory
3. If the user wants to exit the program, the temporary database would be written into a text file in an encrypted format by using the public key generated
4. The encrypted database is saved in the local folder

By following the procedure above, the plain-text would never be written and would only exist in the local memory, thus making it harder for a data leak.

Subsequently, loading a database will have these following procedures:

1. User would be asked to prompt a database folder
2. User would be asked to prompt a private key
3. The database would be decrypted and the result would only be stored inside the system memory

IV. Implementation

As mentioned in the previous section, the program will be implemented using the programming language python. The program has a *main* function that would be executed upon starting the program. The *main* function will call all the necessary *function* to complete a certain task.

This implementation uses the python 3.10.0 version.

There are several libraries used in implementing the main RSA Algorithm: *sympy* and *random.*

Sympy is a python library that allows user to manipulate mathematical expressions. In this implementation, *Sympy* is used to generate the prime numbers.

Random is a python library that is used to generate random integers from a designated value. In this implementation, *Random* is used to generate a random number for finding the *th* prime number.

All the functions regarding the encryption algorithm, its implementation, and its purpose will be listed in the following section:

1. **gcd**

   Is a function that is used find the greatest common divisor between 2 given integers. It uses the Euclidian algorithm recursively.

```
def gcd(a, b):
    if (b == 0):
        return a
    else:
        return gcd(b, a % b)
```

Figure 4.1. gcd function (Source: writer's Archive)

2. **get2Prime**

   Is a function that returns two different prime integers. These integers would later be used in finding the *n* value and the *m* value.

```
def get2Prime():
    p = rd.randint(100,1000)
    q = rd.randint(100,1000)
    while p == q:
        q = rd.randint(100,1000)
    return sy.prime(p),sy.prime(q)
```

Figure 4.2. get2Prim function (Source: writer's Archive)

This function only uses the first 1000 prime numbers due to the lack of computing powers, ideally it should at least use the 10^15th prime number (200 digits).

3. **getPub**

   Is a function that returns a public key value of the encryption algorithm. The value of the public key in capped between 1 to 200 due to the lack of computing power.

```
def getPub(m):
    while(True):
        e = rd.randrange(1,200)
        if (gcd(e,m) == 1):
            return e
```

Figure 4.3. getPub function (Source: writer's Archive)

4. **inverseMod**

   Finds an inverse of a given modulo equation, the inverse would later be used as the hidden private key. The minimum number of the inverse returns is 10 for complexity purposes.

```
def inverseMod(pub,m):
    for i in range(10,m):
        if((pub*i)%m == 1):
            return i
```

Figure 4.4. getPub function (Source: writer's Archive)

5. **rsaAlgorithm**

   This is the main implementation of the RSA algorithm. It first takes *p* and *q* as its two main prime numbers, then a public key (*pub*) would be retrieved followed by its private counterparts. Lastly it will return the public key, private key, and the modulus *n* as a 3 elements tuple.

```
def rsaAlgoritm():
    p,q = get2Prime()
    n = p*q
    m = (p-1)*(q-1)

    pub = getPub(m)
    while pub == 1:
        pub = getPub(m)

    priv = inverseMod(pub,m)
    return pub,priv,n
```

Figure 4.5. rsaAlgorithm function (Source: writer's Archive)

6. **encrypt**

   This function encrypts a single ASCII character using the public key into an integer value that could be decrypted using its private complement. It encode a character into its ASCII value and modify its literal value.

```
def encrypt(char,pub,n):
    return int(pow(ord(char),pub,n))
```

Figure 4.6. encrypt function (Source: writer's Archive)

7. **decrypt**

   Contrary to the encrypt function, the decrypt function as it name suggests, decrypt an integer value using the private key into a readable ASCII character.

```
def decrypt(value,priv,n):
    return int(pow(value,priv,n))
```

Figure 4.7. decrypt function (Source: writer's Archive)

8. **encryptString**

   Processes each individual character in a string into an encrypted data, used to convert a database and save it in its encrypted state.

```
def encryptString(stringInput,pub,n):
    result = ""
    first = True
    for c in stringInput:
        if(c == '\n'):
            result += c
        else:
            if first:
                result += 
str(encrypt(c,pub,n))
                first = False
            else:
                result += '!' + 
str(encrypt(c,pub,n))
    return result
```

Figure 4.8.encryptString function (Source: writer's Archive)

9. **decryptString**

   Works in similar fashion with the encryptString function, only this time it acts as a decryptor of an encrypted database.

```
def decryptString(stringInput, priv, n):
    result = ""
    stringInput = stringInput.split('\n')
    for line in stringInput:
        parse = line.split('!')
        for num in parse:
            if(num != ''):
                result += 
chr(decrypt(int(num),priv,n))
        result += '\n'
    return result
```

Figure 4.9.decryptString function (Source: writer's Archive)

10. **strToKeys**

    Used to convert a string into keys.

```
def strToKeys(strKey):
    strKey = strKey.replace('\n',"")
    strKey = strKey.split(';')
    return int(strKey[0]), 
int(strKey[1]), int(strKey[2])
```

Figure 4.10. strToKeys function (Source: writer's Archive)

11. **keysToStr**

    Used to convert keys into a string for storing its values.

```
def keysToStr(pub, priv, n):
    return 
str(pub)+";"+str(priv)+";"+str(n)
```

Figure 4.11. strToKeys function (Source: writer's Archive)

The rest of the implemented functions revolves around building a proper password manager with its features that supports its ease of use. For readability purpose, it would only be partially listed in this paper. The following figure describes the main implementation of the program:

```
import encryption as enc
import passwordManager as pwd
import extra as ex
import os

# ex.writeBanner("misc/ascii.txt")
accName = "AccData"
keyName = "Key"
param, path = ex.getListDir("database")

if not param:
    pubKey, privKey, nKey = enc.rsaAlgoritm()
    accMatrix = []
    catArray = []
else:
    try:
        keyString = ex.loadFile(path,keyName)
        pubKey, privKey, nKey = 
enc.strToKeys(keyString)
        mainString = ex.loadFile(path,accName)
        mainString = enc.decryptString(mainString, 
privKey, nKey)
        catArray, accMatrix = 
pwd.stringToDat(mainString)
    except FileNotFoundError:
        print("File is either corrupted or doesnt 
exist, terminate program")
        exit()
print("\nThe program has been sucessfully loaded !\n")
userInput = -1
while(userInput != "4"):
    ex.writeMenu()
    userInput = str(input("|-> Please input your 
command : "))
    if userInput == "1":
        pwd.addAccount(accMatrix, catArray)
    elif userInput == "2":
        pwd.deleteAccount(accMatrix, catArray)
    elif userInput == "3":
        pwd.printAccount(accMatrix, catArray)
    elif userInput == "4":
        print("")
    else:
        print("Invalid Command!")

mainString = pwd.datToString(accMatrix, catArray)
mainString = enc.encryptString(mainString, pubKey, 
nKey)

ex.saveFile(path,accName,mainString)
```

Figure 4.12. the main program (Source: writer's Archive)

The program starts by asking the user to choose a database from the available ones. Users will be asked to create a database if there are none present. Next, it would try to load its content, decrypt its database, and obtain the important account and password data. When closed, it will encrypt its temporary database which is stored in the memory, and then write and update them into the existing database folder and files. For simplicity, even though private keys are meant to kept hidden, its value is stored in a “Keys” file in the database folder.

The main functions of the password manager are as follows:
1. Storing account and passwords
2. Deleting accounts
3. Displaying accounts and their corresponding password

The program consists of two main data structure which are an array of categories and a matrix of accounts. each registered account has 4 main sub elements: its account name, username, password, and category for better managing. The existing categories are stored in an array.

V. Testing and Analysis

The program could be executed by simply running the batch executable or clicking the “run.bat” file, make sure that the required python and its libraries are installed. Here is the sample database that is going to be used in the testing of this program.

```
Google;Gaming;Entertainment {Existing categories}
Steam;steamUsername1;steamPassword1;Gaming {Accounts}
Uplay;ubiHard1;passUbi;Gaming
Google;anEmail90@gmail.com;emailPass1;Google
Spotify;bestSong@gmail.com;spotJuked;Entertainment
OnlineShop;ImABuyer;buyer555;E-Commerce
```

Figure 5.1. Sample Uncrypted Database (Source: writer’s Archive)

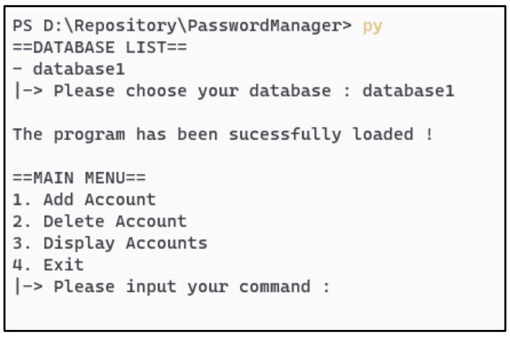
Note that in practice, the database would be encrypted, here is the example of the encrypted database using the public key “183”, the private key “5684407”, and the n modulus “16013551”.

```
11626207!661998!661998!3276739!7615300!15935780!9330872!11626207!15270737!
15703646!7541054!12118223!3276739!9330872!2589233!12118223!13639187!159357
80!5943667!13639187!15270737!7541054!12118223!15703646!15935780!12118223!1
3639187!2102464!5221045!2589233!1489821!7541054!587294!13639187!7541054!12
118223!3276739!2102464!1111605!15270737!13639187!15935780!3276739!661998!5
943667!7541054!15935780!587294!6720756
!12627805!13639187!15935780!15270737!15703646!9330872!587294!13639187!1593
5780!15270737!15703646!7341682!587294!15935780!5943667!12118223!15270737!1
5703646!15935780!11710804!9330872!587294!13639187!15935780!15270737!157036
46!7629877!15270737!587294!587294!14459421!661998!5943667!9144584!11710804
!9330872!11626207!15270737!15703646!7541054!12118223!3276739!2102464!52210
45!7128817!1111605!1111605!661998!449858!12118223!13639187!587294!6720756
!7341682!957058!7615300!15270737!4848718!9330872!449858!3706281!7541054!15
855359!15270737!5943667!9144584!11710804!9330872!957058!15270737!587294!58
7294!7341682!3706281!7541054!9330872!11626207!15270737!15703646!7541054!12
118223!3276739
!11626207!661998!661998!3276739!7615300!15935780!9330872!15270737!12118223
!2589233!15703646!15270737!7541054!7615300!2614196!11812759!4254291!327673
9!15703646!15270737!7541054!7615300!2664596!1111605!661998!15703646!933087
2!15935780!15703646!15270737!7541054!7615300!7629877!15270737!587294!58729
4!11710804!9330872!11626207!661998!661998!3276739!7615300!15935780
!12627805!957058!661998!13639187!7541054!7012639!4848718!9330872!3706281!1
5935780!587294!13639187!12627805!661998!12118223!3276739!4254291!3276739!1
5703646!15270737!7541054!7615300!2664596!1111605!661998!15703646!9330872!5
87294!957058!661998!13639187!9850367!449858!12526026!15935780!9144584!9330
872!2589233!12118223!13639187!15935780!5943667!13639187!15270737!7541054!1
2118223!15703646!15935780!12118223!13639187
!3662525!12118223!7615300!7541054!12118223!15935780!12627805!4723026!66199
8!957058!9330872!1897970!15703646!7128817!15068923!449858!4848718!15935780
!5943667!9330872!3706281!449858!4848718!15935780!5943667!11575397!11575397
!11575397!9330872!2589233!4552316!6987153!661998!15703646!15703646!1593578
0!5943667!1111605!15935780
```

Figure 5.2. Sample Encrypted Database (Source: writer’s Archive)

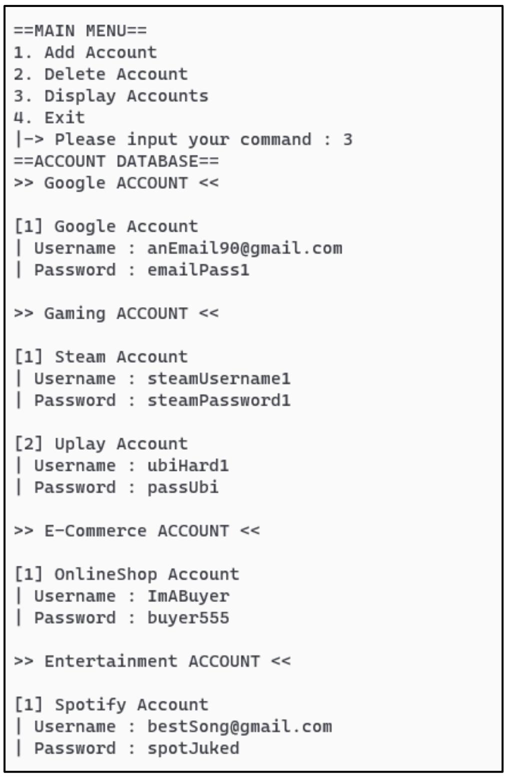
As we can see, it could be almost impossible to read the encrypted database using the naked eye.

The following set of figures demonstrates the execution of the password manager program as a whole, note that all of the unencrypted texts are only stored in the memory:



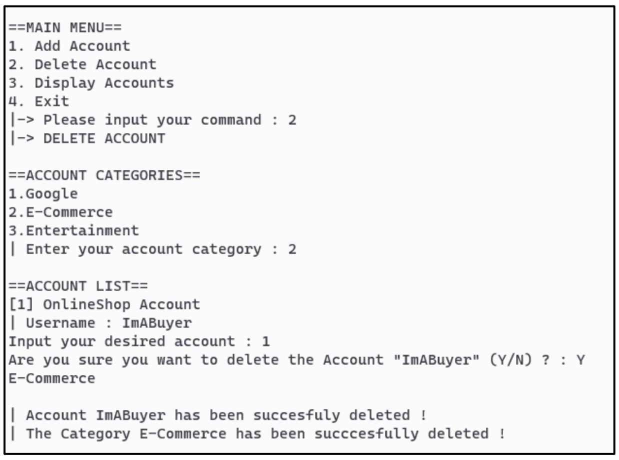
```
PS D:\Repository\PasswordManager> py
==DATABASE LIST==
- database1
|-> Please choose your database : database1

The program has been sucessfully loaded !

==MAIN MENU==
1. Add Account
2. Delete Account
3. Display Accounts
4. Exit
|-> Please input your command :
```

Figure 5.3. Choosing a database (Source: writer’s Archive)



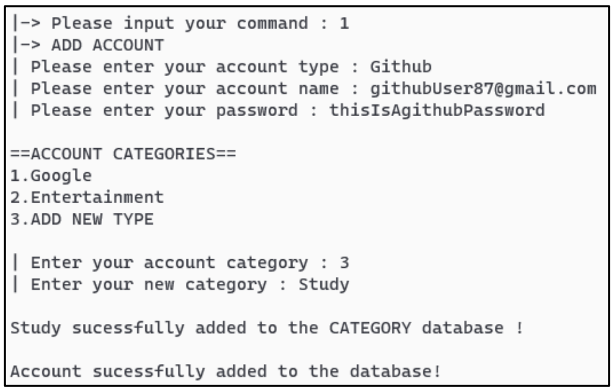
```
==MAIN MENU==
1. Add Account
2. Delete Account
3. Display Accounts
4. Exit
|-> Please input your command : 3
==ACCOUNT DATABASE==
>> Google ACCOUNT <<

[1] Google Account
| Username : anEmail90@gmail.com
| Password : emailPass1

>> Gaming ACCOUNT <<

[1] Steam Account
| Username : steamUsername1
| Password : steamPassword1

[2] Uplay Account
| Username : ubiHard1
| Password : passUbi

>> E-Commerce ACCOUNT <<

[1] OnlineShop Account
| Username : ImABuyer
| Password : buyer555

>> Entertainment ACCOUNT <<

[1] Spotify Account
| Username : bestSong@gmail.com
| Password : spotJuked
```

Figure 5.4. Displaying the accounts (Source: writer’s Archive)



```
==MAIN MENU==
1. Add Account
2. Delete Account
3. Display Accounts
4. Exit
|-> Please input your command : 2
|-> DELETE ACCOUNT

==ACCOUNT CATEGORIES==
1.Google
2.E-Commerce
3.Entertainment
| Enter your account category : 2

==ACCOUNT LIST==
[1] OnlineShop Account
| Username : ImABuyer
Input your desired account : 1
Are you sure you want to delete the Account "ImABuyer" (Y/N) ? : Y
E-Commerce

| Account ImABuyer has been succesfuly deleted !
| The Category E-Commerce has been succcesfully deleted !
```

Figure 5.5. Account deletion (Source: writer’s Archive)



```
|-> Please input your command : 1
|-> ADD ACCOUNT
| Please enter your account type : Github
| Please enter your account name : githubUser87@gmail.com
| Please enter your password : thisIsAgithubPassword

==ACCOUNT CATEGORIES==
1.Google
2.Entertainment
3.ADD NEW TYPE

| Enter your account category : 3
| Enter your new category : Study

Study sucessfully added to the CATEGORY database !

Account sucessfully added to the database!
```

Figure 5.6. Account Addition (Source: writer's Archive)

Through the testing, it can be inferred that the encryption and decryption processes for the database have proven successful. Additionally, the written database exhibits a notable level of resistance to interpretation, rendering it effectively incomprehensible without the appropriate decryption keys. However, this testing also uncovers some notable drawbacks that should be considered:

Firstly, the system shows a drawback in terms of speed when computing large prime numbers. This could potentially impact performance, especially when dealing with extensive database.

Secondly, the password manager lacks certain features, indicating room for improvement. Enhancing the password manager's capabilities could contribute to a more comprehensive security infrastructure.

Lastly, there is a recognized need for a more reliable way to store encryption keys. The current method may pose a vulnerability, and implementing a more secure key storage solution is essential to fortify the overall integrity of the encryption system.

To implement this to its fullest potential, the database should be kept in the cloud and its public key displayed. When the password manager program is called, it would store a set of data locally and when the user is done using it, it would send the database to the cloud with its encryption. The user's private key would be stored locally.

Another improvement is to create a hash function to encrypt the private key, assuming the keys are stored locally.

VI. Conclusion

In conclusion, implementing the RSA algorithm to enhance security in a password managing app offers significant advantages by providing a sturdy encryption and decryption system. The successful testing of encryption and decryption processes showcases its effectiveness in safeguarding sensitive data. However, it is crucial to address potential drawbacks, such as the algorithm's speed when computing large prime numbers and the need for improvements in the password manager's features. Additionally, enhancing the reliability of key storage is essential to fortify the overall safety of the encryption system. Despite these considerations, the RSA algorithm remains a valuable tool for improving the security of a password managing app, and ongoing improvements can further optimize its performance and usability.

VII. Acknowledgements

The completion of this paper could not have been possible without the aid of all IF2120 lecturers, especially Dr. Nur Ulfa Maulidevi, S.T., M.Sc whom has taught the K01 for the discrete mathematics lecture. The writer has learned a hefty amount of information in the process of developing this paper. A debt of gratitude is also owed to Dr. Ir. Rinaldi Munir for providing a great amount of learning resources for the students to study from.

References

[1] Z. I. Borevich, I. R. Shafarevich, *Number Theory*, Newcomb Greenleaf, translator. New York: Academic Press, 1966, pp. 1–18, 155–170 .

[2] K. C. Chowdhury, *A First Course in Number Theory, 2nd Edition.* New Delhi: Asian Books Private Limited, 2007, pp. 1–86.

[3] Z. J. Luo, R. Liu, A. Mehta, *Understanding the RSA Algorithm*, Vol. 1, No. 1, Article. New Jersey: Rider University, 2023.

[4] Brytan, O' Hallaron, *Computer Systems : A Programmer's Perspective*, 2nd Edition. Boston : Pearson, 2011, pp. 46–69.

[5] https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2020-2021/TeoriBilangan-2020-Bagian1.pdf, diakses pada 3 Desember 2023

[6] https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2020-2021/TeoriBilangan-2020-Bagian2.pdf, diakses pada 3 Desember 2023

[7] https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2020-2021/TeoriBilangan-2020-Bagian3.pdf, diakses pada 3 Desember 2023

[8] https://linuxhint.com/generate-prime-numbers-in-pythonJ.

Statement

Hereby, I declare that this paper I have written is my own work, not a reproduction or translation of someone else's paper, and not plagiarized.

Bandung, 11 Desember 2023

[illegible]

Zaki Yudhistira Candra 13522031