Password Manager Master

An Application of Number Theory

Ruotian Zhang

Zhiliang Yu

1. Introduction

1.1. What is cryptography

Recently, in our Number Theory module, we learned about the application of modular arithmetic in encryption technologies. This exploration naturally led us to the concept of cryptography, an area woven into the fabric of our daily lives. You might first think of cryptography as something from spy movies or secret war messages, something really cool but far from our daily life. But actually, it's way closer to us and pretty much everywhere.

For instance, each time we engage in an online transaction, send an email, or communicate via a secure messaging app, we are relying on cryptographic principles to protect our information. In simple terms, cryptography is the science of using mathematical principles to ensure information security.

1.2. The Question of “How to manage passwords better”

For modern individuals navigating the digital era, passwords are an inescapable part of daily life, encompassing everything from email and online shopping to bank accounts, educational platforms, and social media sites like Facebook and Instagram. The use of passwords not only secures our personal digital space, but also poses challenges such as the risk of privacy breaches and the inconvenience of forgotten passwords.

For instance, we may encounter situations such as data loss, due to a browser crash or something we used to rely on. Many of us tend to rely on our browsers, like Chrome, to manage our passwords for convenience. However, such an unexpected crash can result in significant time and effort spent in recalling or resetting various passwords for email, social media, and other online platforms. Such experiences highlight the potential risks of depending solely on something like browser-based password storage and underscore the importance of considering more secure and stable alternatives, like local password management solutions.

In this case, a question arises: How can we develop a more reliable and secure method for managing and storing passwords that reduces the risks associated with browser-based systems?

Our goal is to help people manage their passwords more effectively. This involves developing a user-friendly system that simplifies the process of creating, storing, and retrieving secure passwords. By providing a tool that encrypts password and reduces the risk of security breaches, we aim to make online security more accessible and manageable for everyone.

1.3. Rationale

Our project aims to develop a local password manager that effectively balances security and convenience. Therefore, we decided to develop a local password manager, a tool designed to securely store (super-duper highly encrypted) and manage your passwords directly on your computer. Unlike cloud-based solutions, this manager will keep all data locally, encrypting it to ensure maximum security. The encryption is solid, meaning that even if someone were to gain physical access to your device, they wouldn't be able to decipher your passwords. Additionally, the manager is designed to be user-friendly, making it easy for anyone to securely store and retrieve their password.

2. Analysis

2.1. Analysis Process

In this section, we provide an overview of the methodologies employed to address our password management challenge. Our approach involved the creation of a password manager using Python, drawing extensively from the concepts covered in the CS 5002 course, particularly in the domains of number theory and cryptography. We utilized Python programming to implement the functionalities required for effective password management.

We initiated the analysis by exploring fundamental encoding methods, including the Caesar cipher, Vigenère cipher, RSA, and AES. Each method was chosen based on its relevance to password management and its alignment with the principles covered in the CS 5002 course. Subsequently, we applied those methods above to implement different versions of the password manager, each as a distinct encoding method. This step allowed us to compare the strengths and weaknesses of each approach.

Throughout the analysis, we encountered challenges specific to each encoding method. These challenges provided valuable insights into the practical aspects and limitations of each technique. We summarized the difficulty levels associated with implementing each method and provided a concise overview of different strengths and weaknesses of them.

By following these steps, we not only addressed our password management problem but also gained a comprehensive understanding of the theories and practices related to cryptographic techniques, as taught in the CS 5002 course. This integration of theoretical knowledge and practical application has been instrumental in achieving our project objectives.

2.2. Methodology Overview

2.2.1. Caesar cipher

Concept:

Caesar cipher is a type of substitution cipher in which each letter in the plaintext is replaced by a letter some fixed number of positions down the alphabet.

Algebraic description:

The encryption can also be represented using modular arithmetic by first transforming the letters into numbers, according to the scheme, A → 0, B → 1, ..., Z → 25. Encryption of a letter *x* by a shift *n* can be described mathematically as

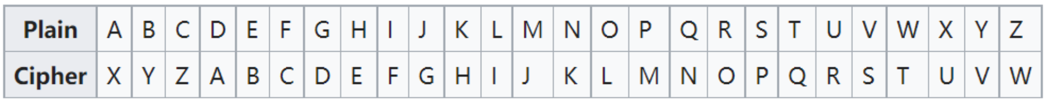
$$E_n(x) = (x + n) \mod 26.$$

Decryption is performed similarly

$$D_n(x) = (x - n) \mod 26.$$

Example:

The transformation can be represented by aligning two alphabets; the cipher alphabet is the plain alphabet rotated left or right by some number of positions. For instance, here is a Caesar cipher using a left rotation of three places, equivalent to a right shift of 23 (the shift parameter is used as the key):



Plain	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
Cipher	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W

When encrypting, a person looks up each letter of the message in the "plain" line and writes down the corresponding letter in the "cipher" line.

Plaintext: THE QUICK BROWN FOX JUMPS OVER THE LAZY DOG

Ciphertext: QEB NRFZH YOLTK CLU GRJMP LSBO QEB IXWV ALD

Deciphering is done in reverse, with a right shift of 3.

Advantages:

1. Simplicity and Ease of Use: The Caesar cipher is incredibly straightforward to understand and implement, even without any specialized equipment. This simplicity makes it an excellent educational tool for introducing the basic concepts of cryptography.

2. Effective Against Uninformed Attackers: For someone completely unaware of the concept of substitution ciphers, the Caesar cipher can effectively obscure information.

Disadvantages:

1. Vulnerability to Frequency Analysis: The cipher is extremely vulnerable to frequency analysis. Since it simply shifts the letters in the alphabet, the frequency of letters in the ciphertext corresponds to the frequency in the plaintext, making it easy to crack.

2. Limited Key Space: The Caesar cipher has a very limited number of possible keys (26 in the case of the English alphabet), making brute-force attacks trivial.

3. No Protection Against Modern Methods: In the context of modern computational power and cryptanalysis techniques, the Caesar cipher offers virtually no security and can be broken almost

instantaneously.

4. Predictability: Its method of encryption is highly predictable once the pattern is understood, as the same plaintext letter is always replaced by the same ciphertext letter for a given key.

2.2.2. Vigenère Cipher

Concept:

The Vigenère cipher is a method of encrypting alphabetic text where each letter of the plaintext is encoded with a different Caesar cipher, whose increment is determined by the corresponding letter of another text, the key.

Algebraic description:

If Σ is the alphabet of length *l*, and *m* is the length of key, Vigenère encryption can be written

$$C_i = E_K(M_i) = (M_i + K_{(i \bmod m)}) \bmod \ell$$

And decryption

$$M_i = D_K(C_i) = (C_i - K_{(i \bmod m)}) \bmod \ell.$$

M_i denotes the offset of the *i*-th character of the plaintext M in the alphabet Σ.

Example:

- if the plaintext is *attacking tonight* and the key is *OCULORHINOLARINGOLOGY*, then

- the first letter *a* of the plaintext is shifted by 14 positions in the alphabet (because the first letter *O* of the key is the 14th letter of the alphabet, counting from 0), yielding *o*;

- the second letter *t* is shifted by 2 (because the second letter *C* of the key means 2) yielding *v*;

- the third letter *t* is shifted by 20 (*U*) yielding *n*, with wrap-around;

and so on; yielding the message *ovnlqbpvt hznzouz*. If the recipient of the message knows the key, they can recover the plaintext by reversing this process.

Advantages:

1. Improved Security Over Simple Ciphers: Compared to the Caesar cipher, the Vigenère cipher offers enhanced security because it uses multiple Caesar ciphers in sequence, making it more resistant to straightforward frequency analysis.
2. Key Flexibility: The key can be any length and contain any letters, which allows for a vast number of possible keys, increasing the difficulty of a brute-force attack.

Disadvantages:

1. Vulnerability to Cryptanalysis: Despite being more secure than simpler substitution ciphers, the Vigenère cipher is still vulnerable to more sophisticated forms of cryptanalysis, such as the Kasiski examination or frequency analysis over longer texts.
2. Key Security: The security of the cipher is heavily dependent on the secrecy of the key. If the key is compromised, the cipher becomes easily breakable.
3. Complexity for Manual Use: While it offers better security, its complexity compared to simpler ciphers makes it more challenging to use without the aid of a computer, especially for longer messages.
4. Predictability with Repeated Key: If a short key is repeated for a long message, patterns can emerge, which can be exploited to break the cipher.

2.2.3. RSA

Concept:

RSA (Rivest–Shamir–Adleman) is a public-key cryptosystem, one of the oldest that is widely used for secure data transmission. In a public-key cryptosystem, each key pair consists of a public key and a corresponding private key. Key pairs are generated with cryptographic algorithms based on mathematical problems termed one-way functions.

Algebraic description:

The RSA algorithm involves four steps: key generation, key distribution, encryption, and decryption.

A basic principle behind RSA is the observation that it is practical to find three very large positive integers *e*, *d*, and *n*, such that with modular exponentiation for all integers *m* (with $0 \leq m < n$):

$$(m^e)^d \equiv m \pmod{n}$$

and that knowing *e* and *n*, or even *m*, it can be extremely difficult to find *d*. Here the symbol ≡ denotes modular congruence: i.e. both $(m^e)^d$ and *m* have the same remainder when divided by *n*.

In addition, for some operations it is convenient that the order of the two exponentiations can be changed: the previous relation also implies

$$(m^d)^e \equiv m \pmod{n}.$$

RSA involves a *public key* and a *private key*. The public key can be known by everyone and is used for encrypting messages. The intention is that messages encrypted with the public key can only be decrypted in a reasonable amount of time by using the private key. The public key is represented by the integers *n* and *e*, and the private key by the integer *d* (although *n* is also used during the decryption process, so it might be considered to be a part of the private key too). *m* represents the message (previously prepared with a certain technique).

Advantages:

1. Asymmetric Encryption: One of the biggest strengths of RSA is its use of asymmetric encryption, which uses two keys – a public key for encryption and a private key for decryption. This eliminates the need to securely share a single secret key between parties.
2. High Security: RSA is considered very secure due to its use of large prime numbers and the difficulty of factoring the product of two large primes, a problem on which its security is based.
3. Widespread Use and Trust: RSA is widely used and trusted in many secure communication protocols, including SSL/TLS for secure internet communications.
4. Digital Signatures: RSA can be used for digital signatures, ensuring the authenticity and integrity of a message or document.

Disadvantages:

1. Computational Intensity: RSA requires significantly more computational resources compared to symmetric key algorithms, making it slower for certain applications, especially where large amounts of data are involved.

2. Key Size: To maintain security, RSA requires much larger key sizes than symmetric key algorithms. As computational power increases, the required key size for RSA continues to grow.

3. Vulnerability to Quantum Computing: In theory, RSA is vulnerable to attacks by quantum computers, which could potentially factor large primes much more efficiently than classical computers, although practical quantum computers of this scale do not yet exist.

4. Implementation Challenges: Incorrect implementation of RSA, such as weak random number generation or poor key management, can lead to vulnerabilities.

2.2.4. AES

Concept:

AES (Advanced Encryption Standard) is based on a design principle known as a substitution–permutation network, and is efficient in both software and hardware. Unlike its predecessor DES, AES does not use a Feistel network. AES is a variant of Rijndael, with a fixed block size of 128 bits, and a key size of 128, 192, or 256 bits.

Description:

AES operates on a 4 × 4 column-major order array of 16 bytes $b_0, b_1, \ldots, b_{15}$ termed the *state*:

$$\begin{bmatrix} b_0 & b_4 & b_8 & b_{12} \\ b_1 & b_5 & b_9 & b_{13} \\ b_2 & b_6 & b_{10} & b_{14} \\ b_3 & b_7 & b_{11} & b_{15} \end{bmatrix}$$

The key size used for an AES cipher specifies the number of transformation rounds that convert the input, called the plaintext, into the final output, called the ciphertext. The number of rounds are as follows:

- 10 rounds for 128-bit keys.
- 12 rounds for 192-bit keys.
- 14 rounds for 256-bit keys.

Each round consists of several processing steps, including one that depends on the encryption key itself. A set of reverse rounds are applied to transform ciphertext back into the original plaintext using the same encryption key.

Advantages:

1. Strong Security: AES is known for its high level of security. Its complex algorithm is resistant to most forms of cryptanalysis, making it a reliable choice for securing sensitive data.
2. Efficiency in Various Platforms: AES performs well on a wide range of hardware, from high-end servers to low-power mobile devices. This efficiency makes it suitable for many applications and technologies.
3. Flexibility with Key Sizes: AES allows for multiple key lengths (128, 192, and 256 bits), offering flexibility based on the level of security required.
4. Widely Tested and Trusted: Since its adoption as a standard, AES has undergone extensive analysis and testing by the global cryptographic community, earning a high level of trust.
5. Suitability for Bulk Encryption: With its efficient performance, AES is ideal for encrypting large volumes of data, which is common in corporate and government environments.

Disadvantages:

1. Vulnerable to Side-Channel Attacks: AES can be susceptible to side-channel attacks, where an attacker gains information from the physical implementation of the cipher, such as power consumption or electromagnetic leaks.
2. Resource Intensive for Small Devices: Despite its overall efficiency, AES can still be resource-intensive for very small or low-power devices, impacting performance.
3. No Built-In Authentication: AES, being a symmetric key encryption algorithm, does not provide authentication or non-repudiation, which are often required in secure communication

systems.

4. Key Management Challenges: In symmetric encryption, secure key distribution and management can be challenging, as the same key is used for both encryption and decryption.

5. Potential Vulnerability to Quantum Computing: Like many encryption methods, AES could potentially be vulnerable to attacks by future quantum computers, although the practicality of such attacks remains theoretical as of now.

To cater to diverse needs, we plan to employ the aforementioned methods to develop different versions of our application. Each version will be tailored to specific requirements, ensuring that we can provide solutions that are most suitable for various scenarios.

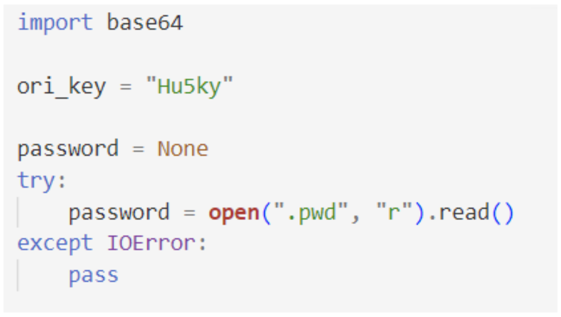
2.3. Step-by-Step Analysis

Within this section, we delve into the practical application of three distinct encoding methods, seeking a profound understanding through their real-world implementation in addressing our password management challenge. Through hands-on exploration and problem-solving, we aim to gain deeper insights into the intricacies of each encoding approach and assess their effectiveness within the context of our project.

2.3.1. Implementing the Vigenère Cipher

In our initial testing phase, we began by implementing a simple version of the Vigenère Cipher as one of the encoding methods for our password manager.

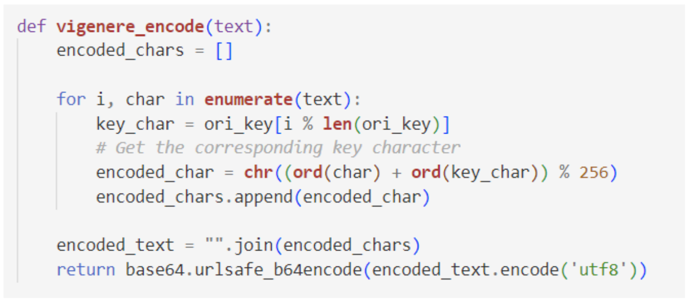
To initiate the Vigenère Cipher implementation, we defined a basic key, "Hu5ky" (from Husky). Our first step involved handling the retrieval of an existing password from the ".pwd" file, establishing a foundation for subsequent encoding and decoding processes.



```python
import base64

ori_key = "Hu5ky"

password = None
try:
    password = open(".pwd", "r").read()
except IOError:
    pass
```

Figure 1 Vigenère Cipher (1)

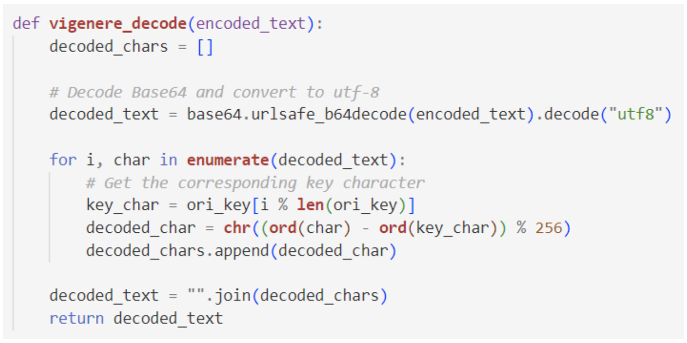
We then crafted a preliminary encoding function, *vigenere_encode(text)*, which aimed to encode text using the Vigenère Cipher. The algorithm cycled through characters of both the input text and the key to produce the encoded output.



```python
def vigenere_encode(text):
    encoded_chars = []

    for i, char in enumerate(text):
        key_char = ori_key[i % len(ori_key)]
        # Get the corresponding key character
        encoded_char = chr((ord(char) + ord(key_char)) % 256)
        encoded_chars.append(encoded_char)

    encoded_text = "".join(encoded_chars)
    return base64.urlsafe_b64encode(encoded_text.encode('utf8'))
```

Figure 2 Vigenère Cipher (2)

Finally, we developed the *vigenere_decode(encoded_text)* function to decode the encoded text back to its original form using the Vigenère Cipher.



```python
def vigenere_decode(encoded_text):
    decoded_chars = []

    # Decode Base64 and convert to utf-8
    decoded_text = base64.urlsafe_b64decode(encoded_text).decode("utf8")

    for i, char in enumerate(decoded_text):
        # Get the corresponding key character
        key_char = ori_key[i % len(ori_key)]
        decoded_char = chr((ord(char) - ord(key_char)) % 256)
        decoded_chars.append(decoded_char)

    decoded_text = "".join(decoded_chars)
    return decoded_text
```

Figure 3 Vigenère Cipher (3)

Several challenges emerged during the implementation. Efficiently cycling through characters of the key during encoding and decoding required careful consideration. We addressed this challenge by utilizing the modulo operator to ensure proper cycling. Additionally, managing the retrieval of an existing password from the ".pwd" file presented complexities, particularly when the file might not exist. We implemented exception handling to gracefully address this scenario.

Beyond the realm of encryption and decryption, we faced additional challenges related to character format types, specifically in the conversion between UTF-8 and Unicode. This aspect became crucial when handling different character sets within our application. Additionally, the usage of *base64.urlsafe_b64encode* introduced complexities. Its specific functionality involves converting input data to a URL-safe, base64-encoded string. These intricacies required a nuanced approach to ensure seamless integration and compatibility within our password manager.

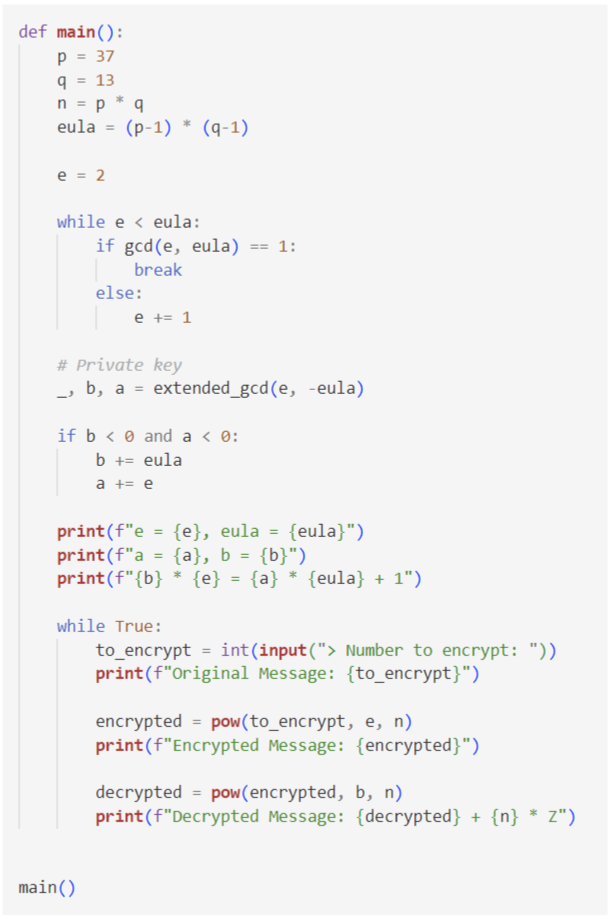
The initial test of the Vigenère Cipher implementation in our password manager was successful. Both the encoding and decoding functions demonstrated effectiveness, providing a secure and practical method for handling passwords within our application.

However, it's essential to note that this encryption method, while functional, proved to be relatively simple. Its simplicity raises concern about susceptibility to potential decryption attempts. Acknowledging this limitation, we recognized the need for a more robust and sophisticated encryption approach. Consequently, we embarked on further exploration and experimentation with more complex encoding methods to enhance the security of our password manager.

2.3.2. Implementing the RSA Algorithm

In this section, we ventured into implementing the RSA Cipher as part of our password manager's encryption methods. The RSA algorithm, known for its robust security through public-key cryptography, offered a sophisticated alternative to our previous attempts.

To better understand the idea of RSA Algorithm, we wrote a simple demo by python first. We originally intended to write an algorithm by hand and use it for our program, while we realized that it would not be powerful enough after we run our demo. We need to give big enough p and q manually to handle complicated messages. But during this process, we did understand how RSA works step by step.

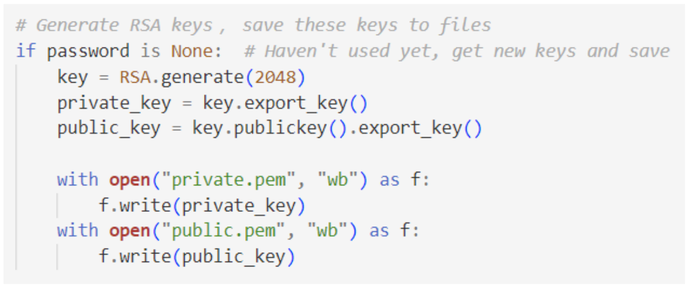


```python
def main():
    p = 37
    q = 13
    n = p * q
    eula = (p-1) * (q-1)

    e = 2

    while e < eula:
        if gcd(e, eula) == 1:
            break
        else:
            e += 1

    # Private key
    _, b, a = extended_gcd(e, -eula)

    if b < 0 and a < 0:
        b += eula
        a += e

    print(f"e = {e}, eula = {eula}")
    print(f"a = {a}, b = {b}")
    print(f"{b} * {e} = {a} * {eula} + 1")

    while True:
        to_encrypt = int(input("> Number to encrypt: "))
        print(f"Original Message: {to_encrypt}")

        encrypted = pow(to_encrypt, e, n)
        print(f"Encrypted Message: {encrypted}")

        decrypted = pow(encrypted, b, n)
        print(f"Decrypted Message: {decrypted} + {n} * Z")


main()
```

Figure 4 RSA Algorithm Demo

Therefore, our approach changed to integrate the RSA algorithm using the Crypto library. We generated RSA keys, saved them to files, and utilized these keys for encryption and decryption. The process aimed to enhance the security of our password manager by employing public-key cryptography.

If the password has not been set, we generated a new pair of RSA keys (2048 bits) – a private key and its corresponding public key. The keys were saved to files named "*private.pem*" and "*public.pem*" for later use.

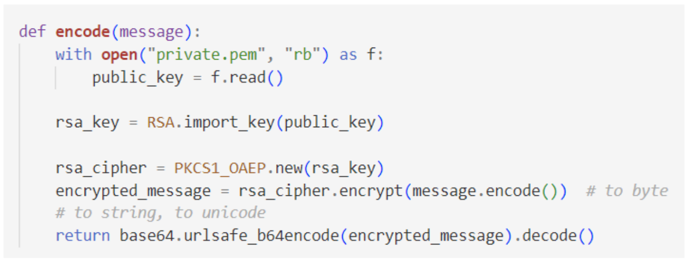


```python
# Generate RSA keys , save these keys to files
if password is None:  # Haven't used yet, get new keys and save
    key = RSA.generate(2048)
    private_key = key.export_key()
    public_key = key.publickey().export_key()

    with open("private.pem", "wb") as f:
        f.write(private_key)
    with open("public.pem", "wb") as f:
        f.write(public_key)
```

Figure 5 RSA Cipher (1)

To encrypt a message, the public key from "private.pem" was loaded. The RSA algorithm using

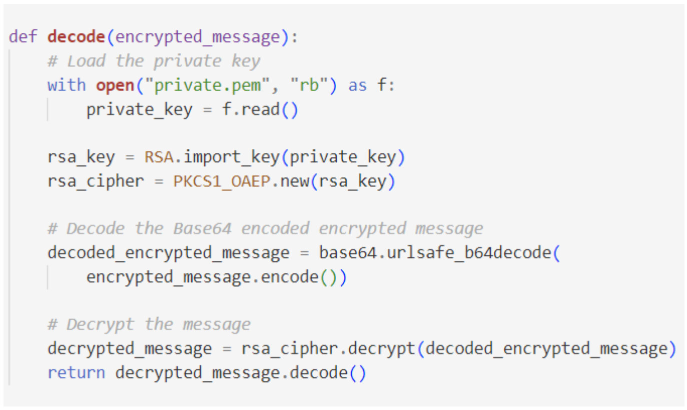
PKCS1_OAEP was applied to encrypt the message, and the result was Base64 encoded for secure transmission.



```python
def encode(message):
    with open("private.pem", "rb") as f:
        public_key = f.read()

    rsa_key = RSA.import_key(public_key)

    rsa_cipher = PKCS1_OAEP.new(rsa_key)
    encrypted_message = rsa_cipher.encrypt(message.encode())  # to byte
    # to string, to unicode
    return base64.urlsafe_b64encode(encrypted_message).decode()
```

Figure 6 RSA Cipher (2)

To decrypt an encrypted message, the private key from "private.pem" was loaded. The Base64 encoded encrypted message was decoded, and the RSA algorithm was used for decryption.

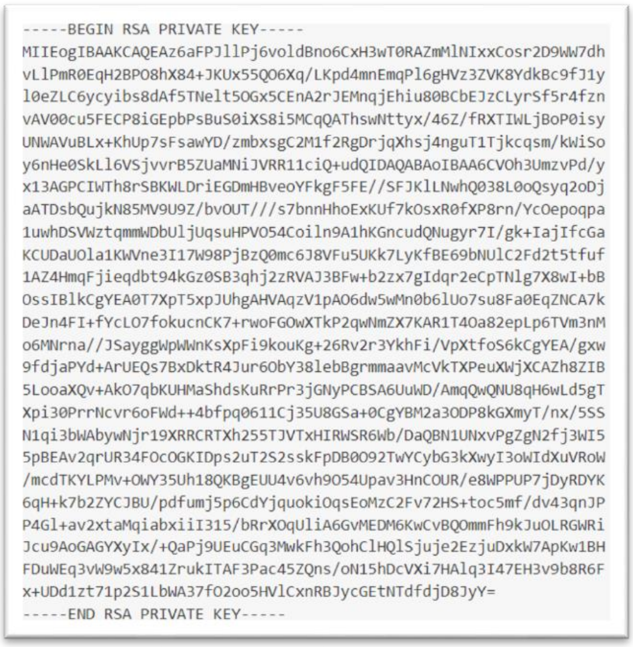


```python
def decode(encrypted_message):
    # Load the private key
    with open("private.pem", "rb") as f:
        private_key = f.read()

    rsa_key = RSA.import_key(private_key)
    rsa_cipher = PKCS1_OAEP.new(rsa_key)

    # Decode the Base64 encoded encrypted message
    decoded_encrypted_message = base64.urlsafe_b64decode(
        encrypted_message.encode())

    # Decrypt the message
    decrypted_message = rsa_cipher.decrypt(decoded_encrypted_message)
    return decrypted_message.decode()
```

Figure 7 RSA Cipher (3)

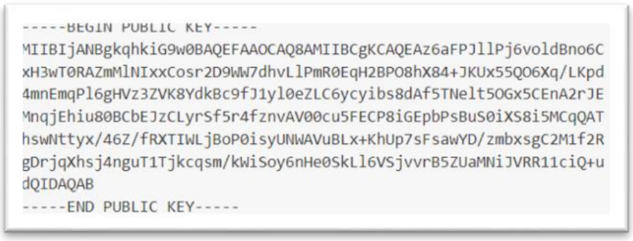
While implementing the RSA Cipher, we faced several challenges that required careful consideration and solutions.

The first is about key management. To ensure consistent decryption, we needed to use the same set of public and private keys. However, generating random keys for each encryption posed difficulties. To solve this, we opted to store the keys in files ("*private.pem*" and "*public.pem*") to maintain consistency between encryption and decryption processes. But in the meantime, storing keys in files introduced a potential security risk, as unauthorized access to these files could compromise the integrity of the encryption.



```
-----BEGIN RSA PRIVATE KEY-----
MIIEogIBAAKCAQEAz6aFPJllPj6voldBno6CxH3wT0RAZmMlNIxxCosr2D9WW7dh
vLlPmR0EqH2BPO8hX84+JKUx55Q06Xq/LKpd4mnEmqPl6gHVz3ZVK8YdkBc9fJ1y
l0eZLC6ycyibs8dAf5TNelt5OGx5CEnA2rJEMnqjEhiu80BCbEJzCLyrSf5r4fzn
vAV00cu5FECP8iGEpbPsBuS0iXS8i5MCqQAThswNttyx/46Z/fRXTIWLjBoP0isy
UNWAVuBLx+KhUp7sFsawYD/zmbxsgC2M1f2RgDrjqXhsj4nguT1Tjkcqsm/kWiSo
y6nHe0SkLl6VSjvvrB5ZUaMNiJVRR11ciQ+udQIDAQABAoIBAA6CVOh3UmzvPd/y
x13AGPCIWTh8rSBKWLDriEGDmHBveoYFkgF5FE//SFJKlLNwhQ038L0oQsyq2oDj
aATDsbQujkN85MV9U9Z/bvOUT///s7bnnHhoExKUf7kOsxR0fXP8rn/YcOepoqpa
1uwhDSVWztqmmWDbUljUqsuHPVO54Coiln9A1hKGncudQNugyr7I/gk+IajIfcGa
KCUDaUOla1KWVne3I17W98PjBzQ0mc6J8VFu5UKk7LyKfBE69bNUlC2Fd2t5tfuf
1AZ4HmqFjieqdbt94kGz0SB3qhj2zRVAJ3BFw+b2zx7gIdqr2eCpTNlg7X8wI+bB
OssIBlkCgYEA0T7XpT5xpJUhgAHVAqzV1pAO6dw5wMn0b6lUo7su8Fa0EqZNCA7k
DeJn4FI+fYcLO7fokucnCK7+rwoFGOwXTkP2qwNmZX7KAR1T4Oa82epLp6TVm3nM
o6MNrna//JSayggWpWWnKsXpFi9kouKg+26Rv2r3YkhFi/VpXtfoS6kCgYEA/gxw
9fdjaPYd+ArUEQs7BxDktR4Jur6ObY38lebBgrmmaavMcVkTXPeuXWjXCAZh8ZIB
5LooaXQv+AkO7qbKUHMaShdsKuRrPr3jGNyPCBSA6UuWD/AmqQwQNU8qH6wLd5gT
Xpi30PrrNcvr6oFWd++4bfpq0611Cj35U8GSa+0CgYBM2a3ODP8kGXmyT/nx/5SS
N1qi3bWAbywNjr19XRRCRTXh255TJVTxHIRWSR6Wb/DaQBN1UNxvPgZgN2fj3WI5
5pBEAv2qrUR34FOcOGKIDps2uT2S2sskFpDB0O92TwYCybG3kXwyI3oWIdXuVRoW
/mcdTKYLPMv+OWY35Uh18QKBgEUU4v6vh9O54Upav3HnCOUR/e8WPPUP7jDyRDYK
6qH+k7b2ZYCJBU/pdfumj5p6CdYjquokiOqsEoMzC2Fv72HS+toc5mf/dv43qnJP
P4Gl+av2xtaMqiabxiiI315/bRrXOqUliA6GvMEDM6KwCvBQOmmFh9kJuOLRGWRi
Jcu9AoGAGYXyIx/+QaPj9UEuCGq3MwkFh3QohClHQlSjuje2EzjuDxkW7ApKw1BH
FDuWEq3vW9w5x841ZrukITAF3Pac45ZQns/oN15hDcVXi7HAlq3I47EH3v9b8R6F
x+UDd1zt71p2S1LbWA37fO2oo5HVlCxnRBJycGEtNTdfdjD8JyY=
-----END RSA PRIVATE KEY-----
```

Figure 8　Private.pem



```
-----BEGIN PUBLIC KEY-----
MIIBIjANBgkqhkiG9w0BAQEFAAOCAQ8AMIIBCgKCAQEAz6aFPJllPj6voldBno6C
xH3wT0RAZmMlNIxxCosr2D9WW7dhvLlPmR0EqH2BPO8hX84+JKUx55Q06Xq/LKpd
4mnEmqPl6gHVz3ZVK8YdkBc9fJ1yl0eZLC6ycyibs8dAf5TNelt5OGx5CEnA2rJE
MnqjEhiu80BCbEJzCLyrSf5r4fznvAV00cu5FECP8iGEpbPsBuS0iXS8i5MCqQAT
hswNttyx/46Z/fRXTIWLjBoP0isyUNWAVuBLx+KhUp7sFsawYD/zmbxsgC2M1f2R
gDrjqXhsj4nguT1Tjkcqsm/kWiSoy6nHe0SkLl6VSjvvrB5ZUaMNiJVRR11ciQ+u
dQIDAQAB
-----END PUBLIC KEY-----
```

Figure 9　Public.pem

The second is about File-Based encryption. Implementing file-based encryption required careful handling of file formats, ensuring proper encoding and decoding during encryption and decryption. So we carefully managed file operations, encoding data for secure storage and decoding when retrieving information.

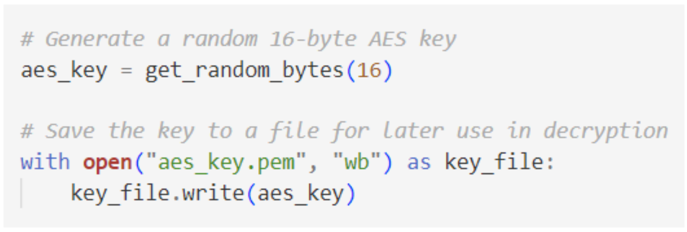
The implementation of the RSA Cipher marked a significant improvement in the complexity and security of our password manager. The adoption of public-key cryptography addressed vulnerabilities present in simpler encryption methods, offering enhanced protection for user passwords. The RSA algorithm's effectiveness in secure communication was evident through its successful integration into our system.

2.3.3. Implementing the AES Algorithm

In this section, we ventured into the implementation of the AES Cipher, a symmetric encryption method chosen alongside the asymmetric encryption method RSA. Since AES was not covered in our coursework, we aimed to gain a basic understanding of its principles by incorporating it into our password manager.

We explored its implementation using the Crypto library in Python, employing modules such as AES, Random, and Util. The key steps in our implementation include steps below.

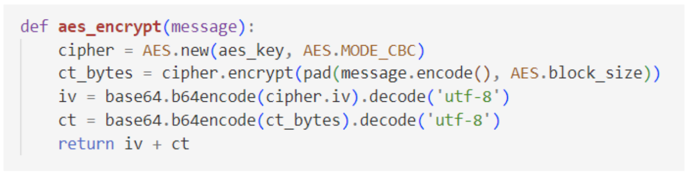
We generated a random 16-byte AES key using the *get_random_bytes* function. The generated key was saved to a file ("*aes_key.pem*") to ensure consistent key usage during both encryption and decryption processes.



```python
# Generate a random 16-byte AES key
aes_key = get_random_bytes(16)

# Save the key to a file for later use in decryption
with open("aes_key.pem", "wb") as key_file:
    key_file.write(aes_key)
```

Figure 10 AES Cipher (1)

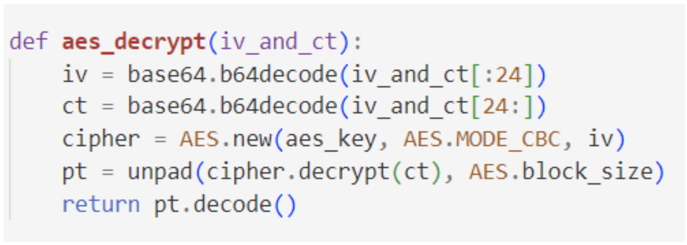
For encrypting user passwords, we employed AES in Cipher Block Chaining (CBC) mode. The *aes_encrypt* function takes the user's password as input, generates a cipher text (ct) and initialization vector (iv), and returns the concatenation of iv and ct.



```python
def aes_encrypt(message):
    cipher = AES.new(aes_key, AES.MODE_CBC)
    ct_bytes = cipher.encrypt(pad(message.encode(), AES.block_size))
    iv = base64.b64encode(cipher.iv).decode('utf-8')
    ct = base64.b64encode(ct_bytes).decode('utf-8')
    return iv + ct
```

Figure 11 AES Cipher (2)

The *aes_decrypt* function receives the concatenated iv and ct, decodes them, and performs AES decryption. It ensures proper unpadding to retrieve the original plaintext password.



```python
def aes_decrypt(iv_and_ct):
    iv = base64.b64decode(iv_and_ct[:24])
    ct = base64.b64decode(iv_and_ct[24:])
    cipher = AES.new(aes_key, AES.MODE_CBC, iv)
    pt = unpad(cipher.decrypt(ct), AES.block_size)
    return pt.decode()
```

Figure 12 AES Cipher (3)

Amongst these three methods, the Vigenère Cipher proved to be relatively straightforward, characterized by a simple encoding and decoding process. The method's advantages include ease of implementation and understanding, making it suitable for scenarios with short texts. However, it has notable disadvantages, such as vulnerability to frequency analysis and known-plaintext attacks. The periodic repetition of the key limits its security for longer texts.

The RSA Algorithm implementation posed a moderate level of difficulty, involving tasks like key generation, encryption, and decryption with careful consideration for secure file handling. RSA offers strong security due to its use of asymmetric keys, making it suitable for secure communication and digital signatures. However, it comes with disadvantages, including relatively slower processing for large data and complexities in key management, especially with longer key lengths.

Implementing the AES Cipher ranged from moderate to high difficulty, requiring attention to key generation, encryption, and decryption, including considerations for padding and mode of operation. AES is efficient for handling large amounts of data and provides strong security with proper key management.

In summary, the Vigenère Cipher, while simple, lacks robust security and is suitable for shorter texts. RSA, with its moderate complexity, offers strong security for various applications but may face challenges in processing large data and key management. AES, with moderate to high complexity, excels in efficiently securing large datasets but requires careful consideration in key distribution and management. We choose RSA Cipher for our final implementation, while other methods can still be valid choices for different use.

2.4. Tools Used

In the development of our password manager, we harnessed a combination of tools and technologies to achieve a robust and user-friendly application. Python served as the programming language foundation, enabling us to implement core functionalities, cryptographic operations, and file handling. The Tkinter library played a pivotal role in crafting a graphical user interface (GUI), offering a seamless interaction experience. It's important to note that Tkinter is compatible primarily with Windows. Additionally, we enriched the user interface with self-designed icons, elevating the visual appeal and usability of the password manager.

2.4.1. Tkinter Library

In order to enhance user experience and facilitate seamless interaction, we leveraged the Tkinter library in Python for designing the graphical user interface (GUI). Tkinter provides a versatile set of tools for creating intuitive and user-friendly interfaces, allowing us to organize distinct windows for functionalities such as log in, add, list, and search. Each of these windows is tailored to specific user actions, providing a clear and organized structure that contributes to an overall efficient and enjoyable user experience. The integration of Tkinter demonstrates our commitment to crafting a visually appealing and user-centric password management solution.

2.4.2. Python

As introduced above, we use python as our main language to implement our program, and to make our developing clear, we split it into different parts to make it easier for implementing and debugging.

The primary workflow of our program follows a straightforward process. When the user executes manager.py, the initial interface prompts the user to either set up a password (if not done before) or enter the verification password (if already set). Upon successfully setting up or verifying the password, the main window displays three core functionalities: add, list, and search.

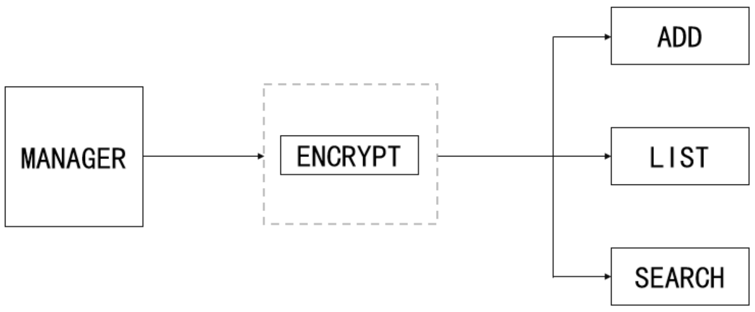


Figure 13 Structure of the Password Manager Master

Below are the modules of the project.

(1) Manager

The purpose of this module is to serve as a main entry of the application, to initialize and manage the entire application work flow. This module contains the creation of the main interface, handling user login and registration, generating several main windows, and integrating other modules (such as List, Add, Search). It deals with user inputs, navigation, and state management of the entire application.

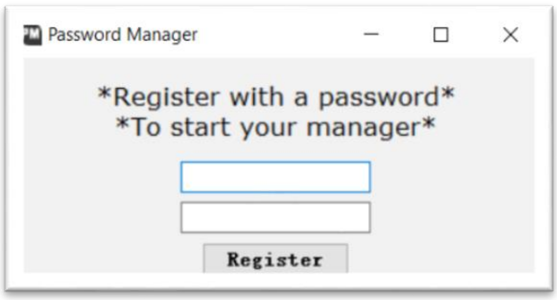


Figure 14 Start Window

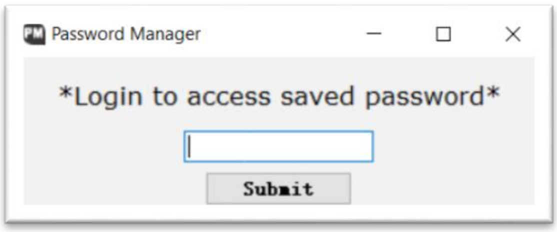


Figure 15 Log in Window

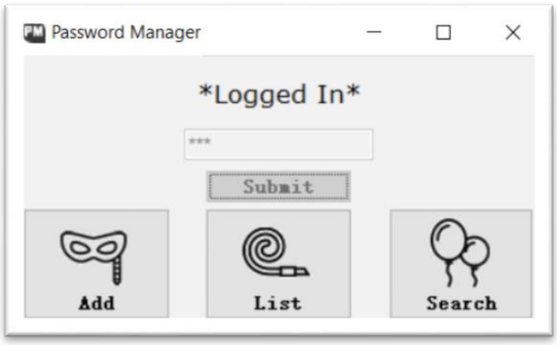


Figure 16 Function Window

(2) Add

This module is to enable user to add new password records, including new keywords, usernames and passwords. Then the passwords and related information will be stored in a local database after encrypted.

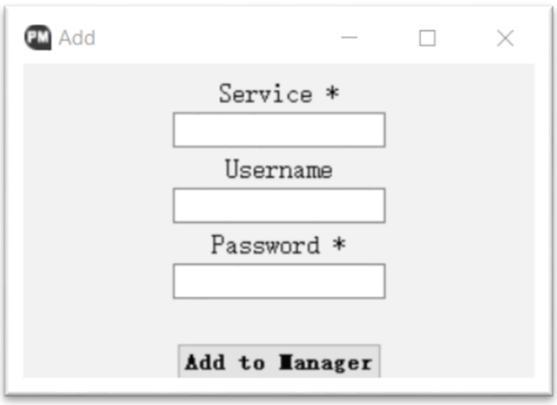
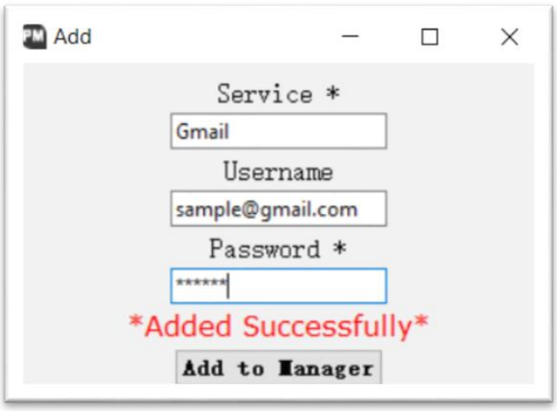


Figure 17 Add Window


Service *: Gmail

Username: sample@gmail.com

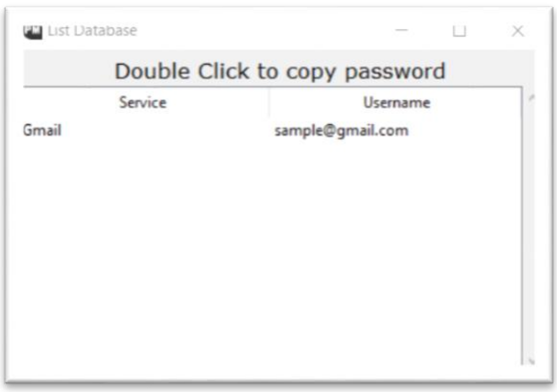
Password *: ******

Added Successfully

Figure 18 Added successfully Window

(3) List

The purpose of this module is to store, manage and display the stored passwords. It is responsible to read the user's input password, pass it to the Encode and Add module and parse the stored data. It also provides the user with an interface to view the saved password list and browse and copy it. The user can also double click to copy the password, which can avoid the password to be shown apparently.



Service	Username
Gmail	sample@gmail.com

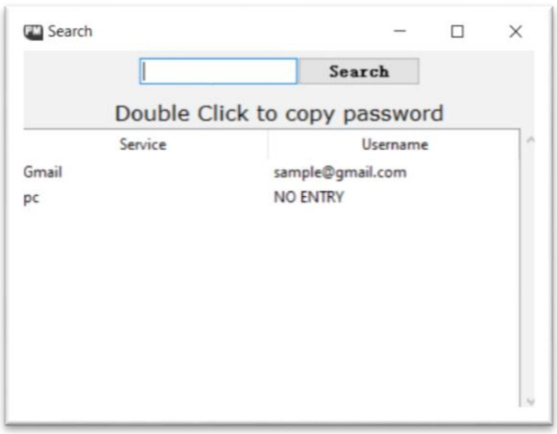
Figure 19 List Window

(4) Encode

This part manages the encryption and decryption of passwords, and ensure the stored data is not easily accessible by unauthorized parties. This process includes not only the encryption but also the string converting into binary numbers, as well as encoding and decoding strings in formats like UTF-8 or UTF-16. We also use hash function to check whether the password for the manager is correct. This part has been detailed introduced in section 2.3.

(5) Search

This module is designed for user to quickly find specific password in the stored records. The user can enter the keyword to search the particular result, and the search result will be displayed.



Service	Username
Gmail	sample@gmail.com
pc	NO ENTRY

Figure 20 Search Window

2.4.3. Adobe Photoshop and Illustrator

In addition, we have designed some small and refined icons for the user interface, including the top navigation bar and unique icons for each functionality button. This subtle detail injects vitality and charm into the entire program, making the user interface more dynamic, reader-friendly, and interactive.

Figure 21 Icon for top bar (representing Password Master)

Figure 22 Icons for Add, List, Search

3. Conclusion

3.1. The result of our program

The outcome of our program is indeed successful, fulfilling our initial goal of enhancing password management through effective local storage, browsing, and searching functionalities. The program not only meets but surpasses our expectations, providing a seamless and efficient solution for users to manage their passwords with ease and security.

Throughout the process, we gained insights into various encryption methods. Upon careful consideration, we conclude that RSA or AES stands out as superior encryption methods, contributing to the overall security and robustness of our password management solution.

3.2. Project Weaknesses and Limitations

Our program currently functions as a basic password manager, and there are several areas that can be enhanced and refined. These include implementing multi-user password management, exploring methods to encrypt key files for increased security, and establishing connections with websites or applications to enable automatic password filling. These avenues present opportunities for future improvements and developments in our password management system.

3.3. What we have learned from this project

Ruotian Zhang: This project has addressed a fundamental issue for me—providing a secure storage solution for the passwords I lost after my Google Chrome browser crashed a month ago. Beyond that, it marks my first attempt at developing a project from scratch, albeit a relatively simple one. Personally, aspiring to become a software engineer and dreaming of developing my own independent game someday, I and my teammates share a greater interest in software development than data processing. We chose a distinct approach, primarily relying on programming to bring our ideas to life.

Throughout the process, I gained valuable experience, including breaking down a large project into manageable and executable parts, possibly incorporating some object-oriented programming concepts, and systematically completing tasks within a framework. I learned how to flexibly utilize Python libraries to achieve our goals, and, drawing on my past design knowledge, I established interactive interfaces to enhance visual effects. These insights not only provided me with a clearer and deeper understanding of program development but also sparked more interest. Perhaps, they will guide and assist me in the upcoming 5004 Object-Oriented Programming course next semester, instilling greater confidence in realizing my dream of developing an independent game.

Zhiliang Yu: What I have learned from this project is truly invaluable. I and my teammate had the opportunity to delve deep into the mysteries of cryptography, a journey that allowed us to experience the romance inherent in the mathematical realm. Together with my teammate, we crafted codes, a process that was not just technically enlightening but also creatively fulfilling. Our proficiency in programming was significantly enhanced through this endeavor. Implementing various cryptographic algorithms challenged and refined our coding skills, fostering a deeper understanding of the logic and structure behind effective programming.

Equally important was the lesson in teamwork. The success of this project hinged on our ability to work cohesively as a unit, sharing insights, and supporting each other's efforts. This collaboration fostered a harmonious working environment and was instrumental in achieving our project goals.

In conclusion, this project was a comprehensive learning experience, encompassing cryptography, programming, and teamwork. Each aspect contributed to our overall growth, equipping us with valuable skills and knowledge applicable in our future academic and professional pursuits.

4. Appendix

4.1. Reference and Citation

1. Vigenère Cipher Concept: https://en.wikipedia.org/wiki/Vigen%C3%A8re_cipher
2. Vigenère Cipher: https://www.geeksforgeeks.org/vigenere-cipher/
3. RSA Algorithm Cryptography Concept: https://en.wikipedia.org/wiki/RSA_(cryptosystem)
4. RSA Algorithm Cryptography: https://www.geeksforgeeks.org/rsa-algorithm-cryptography/
5. AES Algorithm Cryptography Concept: https://en.wikipedia.org/wiki/Advanced_Encryption_Standard
6. AES Algorithm Cryptography: https://www.geeksforgeeks.org/advanced-encryption-standard-aes/
7. Password Manager structure: https://github.com/ashutoshkrris/Password-Manager-using-Tkinter https://github.com/mukkachaitanya/Password-Manager/tree/master
8. Tkinter Usage: http://stupidpythonideas.blogspot.in/2013/12/tkinter-validation.html
9. Tkinter Usage, binding a key to a button: http://stackoverflow.com/questions/11456631/how-to-capture-events-on-tkinter-child-widgets

4.2. Source

1. Add.py

```python
import json
import encode

try:
    from tkinter import *
    from tkinter import ttk
except ImportError:
    from Tkinter import *
    import ttk

LABEL_FONT = ("Monospace", 12)
BUTTON_FONT = ("Sans-Serif", 10, "bold")
INFO_FONT = ("Verdana", 12)


class AddWindow(Toplevel):
    ''' Add Window'''
    def __init__(self, *args):  # A tuple of many arguments
        Toplevel.__init__(self, *args)

        # Add Credentials
        self.title("Add")
        self.setFrames()

    def setFrames(self, **kwargs):
        # a Frame called 'add'
        add = Frame(self, padx=2, pady=2, bd=3)
        add.pack()

        # Three labels
        # Service
        Label(add, text="Service *", width=30, bd=3, font=LABEL_FONT).pack()
        service = ttk.Entry(add)
        service.pack()  # Put the Entry into the add frame

        # Username
        Label(add, text="Username", width=30, bd=3, font=LABEL_FONT).pack()
        username = ttk.Entry(add)
        username.pack()

        # Password
        Label(add, text="Password *", width=30, bd=3, font=LABEL_FONT).pack()
        password = ttk.Entry(add, show="*")
        password.pack()
```

```python
        # Style of submit button, font and align
        sty = ttk.Style()
        sty.configure("Submit.TButton", font=BUTTON_FONT, sticky="s")

        # label for spacing extra information in red
        info = Label(add, width=30, bd=3, fg="red", font=INFO_FONT)
        info.pack()

        addBtn = ttk.Button(add, text="Add to Manager", style="Submit.TButton",
                            command=lambda: self.addClicked(
                                info=info, username=username,
                                password=password, service=service))

        addBtn.pack()
```

```
    def addClicked(self, **kwargs):  # A dict of many arguments
        fileName = ".data"
        data = {}
        # Writing data to a secrete file

        if (kwargs['password'].get() != "" and kwargs['service'].get() != ""):
            details = [kwargs['username'].get(),
                    #  encode.encode(kwargs['password'].get()).decode('utf-8')]
                       encode.encode(kwargs['password'].get())]

            # read present data
            try:
                with open(fileName, "r") as outfile:
                    data = outfile.read()  # in json format
            except IOError:
                # Create file if doesn't exits
                open(fileName, "a").close()

            # load new data
            if data:
                data = json.loads(data)  # from json into dictionary
                data[kwargs['service'].get()] = details
            else:
                data = {}
                data[kwargs['service'].get()] = details

            # Writing back the data
            with open(".data", "w") as outfile:
                outfile.write(json.dumps(data, sort_keys=True, indent=4))
                # from dictionary to json, to write into the file
```

```
            # To delete contents of the input Entry
            for input_frame in ('username', 'service', 'password'):
                kwargs[input_frame].delete(0, 'end')

            kwargs['info'].config(text="*Added Successfully*")

        # No input
        else:
            kwargs['info'].config(text="*Please enter Service and Password*")
```

2. List.py

```
try:
    from tkinter import *
    from tkinter import ttk
except ImportError:
    from Tkinter import *
    import ttk

import encode
import json
import pyperclip


NORM_FONT = ("Helvetica", 10)
LARGE_FONT = ("Verdana", 13)


class ListWindow(Toplevel):
    def __init__(self, *args):
        Toplevel.__init__(self, *args)
        self.title("List Database")

        self.frame = getTreeFrame(self, bd=3)
        self.frame.pack()
```

```python
class getTreeFrame(Frame):
    def __init__(self, *args, **kwargs):
        Frame.__init__(self, *args, **kwargs)
        self.addLists()

    def addLists(self, *arg):
        dataList = self.getData()
        headings = ["Service", "Username"]

        if dataList:
            # Adding the Treeview
            Label(self, text="Double Click to copy password",
                    bd=2, font=LARGE_FONT).pack(side="top")

            # Scroll bar
            scroll_bar = ttk.Scrollbar(self, orient=VERTICAL, takefocus=True)

            # Creates a Treeview widget with specified columns and headings.
            self.tree = ttk.Treeview(self, columns=headings, show="headings")
            scroll_bar.config(command=self.tree.yview)
            # Configures the Treeview to use the scrollbar.
            self.tree.configure(yscroll=scroll_bar.set)

            scroll_bar.pack(side=RIGHT, fill=Y)
            self.tree.pack(side=LEFT, fill='both', expand=1)

            # Adding headings to the columns and sets commands for sorting
            for heading in headings:
                self.tree.heading(
                    heading, text=heading,
                    command=lambda c=heading: self.sortby(self.tree, c, 0))
                self.tree.column(heading, width=200)

            for data in dataList:
                self.tree.insert("", "end", values=data)

            self.tree.bind("<Double-1>", self.OnDoubleClick)

        else:
            self.errorMsg()

    def getData(self, *arg):
        fileName = ".data"
        self.data = None
```

```python
        try:
            with open(fileName, "r") as outfile:
                self.data = outfile.read()
        except IOError:
            return ""

        # If there is no data in file
        if not self.data:
            return ""

        self.data = json.loads(self.data)  # JSON-formatted string to dict
        dataList = []

        for service, detail in self.data.items():
            if detail[0]:
                usr = detail[0]
            else:
                usr = "NO ENTRY"
            dataList.append((service, usr))

        return dataList
```

```python
    def errorMsg(self, *args):
        msg = "*No saved data*"
        label = Label(self, text=msg, font=NORM_FONT, bd=3, width=30)
        label.pack(side="top", fill="x", pady=10)
        B1 = ttk.Button(self, text="Okay", command=self.master.destroy)
        B1.pack(pady=10)

    def OnDoubleClick(self, event):
        item = self.tree.focus()

        # Copies password to clipboard
        service = self.tree.item(item, "values")[0]
        var = self.data[service][1]
        var = encode.decode(var)
        pyperclip.copy(var)

    # for search
    def updateList(self, regStr, *args):
        for x in self.tree.get_children(''):
            self.tree.delete(x)

        for data in self.getData():
            if regStr is None or any(
                    (isinstance(regStr, re.Pattern) and regStr.search(str(value))) or
                    (not isinstance(regStr, re.Pattern) and re.search(regStr, str(value), re.IGNORECASE))
                    for value in data
                ):
                self.tree.insert("", "end", values=data)

    def sortby(self, tree, col, descending):
        # Sort tree contents when a column header is clicked on
        # Grab values to sort
        data = [(tree.set(child, col), child)
                for child in tree.get_children('')]

        # Sort the data in place
        data.sort(reverse=descending)
        for ix, item in enumerate(data):
            tree.move(item[1], '', ix)
        # switch the heading cmds so it will sort in the opposite direction
        tree.heading(col,
                     command=lambda col=col: self.sortby(tree, col,
                                                         int(not descending)))
```

3. Search.py

```python
try:
    from tkinter import *
    from tkinter import ttk
except ImportError:
    from Tkinter import *
    import ttk
import List
import re

BUTTON_FONT = ("Sans-Serif", 10, "bold")
```

```python
class SearchWindow(Toplevel):
    def __init__(self, *args):
        Toplevel.__init__(self, *args)

        self.title("Search")

        self.frame = Frame(self, padx=2, pady=2, bd=3)
        self.frame.pack()

        self.to_search = StringVar()
        self.to_search.trace('w', self.onUpdate)
        search_win = ttk.Entry(self.frame, textvariable=self.to_search)
        search_win.grid(row=0, columnspan=2)
        search_win.focus_set()

        sty = ttk.Style()
        sty.configure("Submit.TButton", font=BUTTON_FONT, sticky="e")

        search_btn = ttk.Button(self.frame, text="Search",
                                style="Submit.TButton",
                                command=lambda:  self.onUpdate())
        search_btn.grid(row=0, column=3, sticky="e")

        self.tree = List.getTreeFrame(self, bd=3)
        self.tree.pack()

    def onUpdate(self, *args):
        # *args = [name, index, mode]
        content = self.to_search.get()
        search_re = re.compile(content, re.IGNORECASE)
        self.tree.updateList(search_re)
        return True
```

4. manager.py

```python
try:
    from tkinter import *
    from tkinter import ttk
except ImportError:
    from Tkinter import *
    import ttk

import Add
import os
import encode
import List
import Search


LARGE_FONT = ("Verdana", 13)
BUTTON_FONT = ("Sans-Serif", 10, "bold")


class Login(Tk):
    def __init__(self, *args):
        Tk.__init__(self, *args)

        if os.name == 'nt':  # Only Windows can use tk
            Tk.iconbitmap(self, default='icon.ico')

        Tk.wm_title(self, "Password Manager")
        self.state = {
            "text": "*Login to access saved password*", "val": False
        }

        if encode.password:  # Has logged in
            self.addLoginFrame()
        else:
            self.addRegisterFrame()
    def addLoginFrame(self):
        login = Frame(self, padx=2, pady=2, bd=2)
        login.pack()

        login_label = Label(login, text=self.state['text'],
                            bd=10, font=LARGE_FONT, width=30)
        login_label.grid(row=0, columnspan=3)

        entry = ttk.Entry(login, show="*")
        entry.grid(row=1, column=1, pady=3)

        entry.focus_set()

        s = ttk.Style()
        s.configure("Submit.TButton", font=BUTTON_FONT)
        submit_btn = ttk.Button(login, text="Submit", style="Submit.TButton",
                                command=lambda: self.checkPwd(
                                   login, label=login_label, entry=entry,
                                   btn=submit_btn))

        submit_btn.grid(row=2, column=1, pady=3)

    def checkPwd(self, frame, **kwargs):
        # **kwargs: label, entry, submit button
        to_check = kwargs['entry'].get()  # input

        # if passwords match
        # if hashlib.md5(chk.encode("utf8")).hexdigest() == encode.password:
        with open(".pwd", "r") as file:
            stored_encrypted_password = file.read()
        decrypted_password = encode.decode(stored_encrypted_password)

        if to_check == decrypted_password:
            self.state['text'] = "*Logged In*"
            self.state['val'] = True
            kwargs['label'].config(text=self.state['text'])
            kwargs['entry'].config(state=DISABLED)  # cannot use
            kwargs['btn'].config(state=DISABLED)

            # add buttons
            self.addConfigBtn(frame)
```

```python
        # Passwords don't match
        else:
            kwargs['label'].config(text=self.state['text'] + "\n*Try Again*")

    def addConfigBtn(self, login):
        btn_list = ["Add", "List", "Search"]
        btn_cmd_lst = [lambda: Add.AddWindow(self),
                       lambda: List.ListWindow(self),
                       lambda: Search.SearchWindow(self)]

        frames = []  # Frames array
        imgs = []  # image array
        self.img = []  # don't know how the bug fixed...

        for i in range(3):
            frames.append(
                Frame(login, padx=2, width=50, height=50))
            frames[i].grid(row=3, column=i)

            imgs.append(
                PhotoImage(
                    file=btn_list[i] + ".gif", width=48, height=48))
            self.img.append(imgs[i])

            ttk.Button(frames[i], image=imgs[i], text=btn_list[i], compound="top",
                       style="Submit.TButton",
                       command=btn_cmd_lst[i]).grid(sticky="NWSE")
    def addRegisterFrame(self, *arg):
        register = Frame(self, padx=2, pady=2, bd=2)
        register.pack()

        info = "*Register with a password*\n*To start your manager*"
        register_label = Label(register, text=info,
                               bd=10, font=LARGE_FONT, width=30)
        register_label.grid(row=0, columnspan=3)

        pw = ttk.Entry(register, show="*")
        pw.grid(row=1, column=1, pady=3)
        pw.focus_set()

        pwChk = ttk.Entry(register, show="*")
        pwChk.grid(row=2, column=1, pady=3)
        pwChk.bind('<Return>', lambda _: self.register(register,
                                                       pw, pwChk))

        sty = ttk.Style()
        sty.configure("Submit.TButton", font=BUTTON_FONT)
        submit_btn = ttk.Button(register, text="Register",
                                style="Submit.TButton",
                                command=lambda: self.register(register,
                                                              pw, pwChk))
        submit_btn.grid(row=3, column=1, pady=3)
    def register(self, frame, *pwd):
        # *pwd is a list containing password inputs
        if pwd[0].get() == pwd[1].get():
            # encode.password = hashlib.md5(pwd[0].get().encode()).hexdigest()
            encode.password = encode.encode(pwd[0].get())
            # Saving password for future use.
            open(".pwd", "w").write(encode.password)
            frame.destroy()
            self.addLoginFrame()

        else:
            error = "*Passwords dont match*\n*Try again*"
            error_label = Label(frame, text=error,
                                bd=10, font=("Verdana", 11), fg="red")
            error_label.grid(row=4, column=1, pady=3)

            # Removing entered Passwords
            for word in pwd:
                word.delete(0, 'end')


if __name__ == '__main__':
    new = Login()
    new.mainloop()
```

5. encode.py

```
from Crypto.PublicKey import RSA
from Crypto.Cipher import PKCS1_OAEP
import base64

# pwd = "Hu5ky"

password = None
try:
    password = open(".pwd", "r").read()
except IOError:
    pass
```

```
# Generate RSA keys ,  save these keys to files
if password is None:  # Haven't used yet, get new keys and save
    key = RSA.generate(2048)
    private_key = key.export_key()
    public_key = key.publickey().export_key()

    with open("private.pem", "wb") as f:
        f.write(private_key)
    with open("public.pem", "wb") as f:
        f.write(public_key)


# RSA
def encode(message):
    with open("private.pem", "rb") as f:
        public_key = f.read()

    rsa_key = RSA.import_key(public_key)

    rsa_cipher = PKCS1_OAEP.new(rsa_key)
    encrypted_message = rsa_cipher.encrypt(message.encode())  # to byte
    # to string, to unicode
    return base64.urlsafe_b64encode(encrypted_message).decode()


def decode(encrypted_message):
    # Load the private key
    with open("private.pem", "rb") as f:
        private_key = f.read()

    rsa_key = RSA.import_key(private_key)
    rsa_cipher = PKCS1_OAEP.new(rsa_key)
```

```
    # Decode the Base64 encoded encrypted message
    decoded_encrypted_message = base64.urlsafe_b64decode(
        encrypted_message.encode())

    # Decrypt the message
    decrypted_message = rsa_cipher.decrypt(decoded_encrypted_message)
    return decrypted_message.decode()
```

6. rsa_demo.py

```
def gcd(a, h):
    temp = 0
    while True:
        temp = a % h
        if temp == 0:
            return h
        a = h
        h = temp
```

```
def extended_gcd(a, b):  # a*x + b*y = 1
    if a == 0:
        return b, 0, 1
    else:
        gcd, x1, y1 = extended_gcd(b % a, a)
        x = y1 - (b // a) * x1
        y = x1
        return gcd, x, y
```

```python
def main():
    p = 37
    q = 13
    n = p * q
    eula = (p-1) * (q-1)

    e = 2

    while e < eula:
        if gcd(e, eula) == 1:
            break
        else:
            e += 1

    # Private key
    _, b, a = extended_gcd(e, -eula)

    if b < 0 and a < 0:
        b += eula
        a += e

    print(f"e = {e}, eula = {eula}")
    print(f"a = {a}, b = {b}")
    print(f"{b} * {e} = {a} * {eula} + 1")

    while True:
        to_encrypt = int(input("> Number to encrypt: "))
        print(f"Original Message: {to_encrypt}")

        encrypted = pow(to_encrypt, e, n)
        print(f"Encrypted Message: {encrypted}")

        decrypted = pow(encrypted, b, n)
        print(f"Decrypted Message: {decrypted} + {n} * Z")


main()
```

7. vigenere.py

```python
import base64

ori_key = "Hu5ky"

password = None
try:
    password = open(".pwd", "r").read()
except IOError:
    pass
```

```python
def vigenere_encode(text):
    encoded_chars = []

    for i, char in enumerate(text):
        key_char = ori_key[i % len(ori_key)]
        # Get the corresponding key character
        encoded_char = chr((ord(char) + ord(key_char)) % 256)
        encoded_chars.append(encoded_char)

    encoded_text = "".join(encoded_chars)
    return base64.urlsafe_b64encode(encoded_text.encode('utf8'))
```

```python
def vigenere_decode(encoded_text):
    decoded_chars = []

    # Decode Base64 and convert to utf-8
    decoded_text = base64.urlsafe_b64decode(encoded_text).decode("utf8")

    for i, char in enumerate(decoded_text):
        # Get the corresponding key character
        key_char = ori_key[i % len(ori_key)]
        decoded_char = chr((ord(char) - ord(key_char)) % 256)
        decoded_chars.append(decoded_char)

    decoded_text = "".join(decoded_chars)
    return decoded_text
```

8. aes.py

```python
from Crypto.Cipher import AES
from Crypto.Random import get_random_bytes
from Crypto.Util.Padding import pad, unpad
import base64

# Generate a random 16-byte AES key
aes_key = get_random_bytes(16)

# Save the key to a file for later use in decryption
with open("aes_key.pem", "wb") as key_file:
    key_file.write(aes_key)


def aes_encrypt(message):
    cipher = AES.new(aes_key, AES.MODE_CBC)
    ct_bytes = cipher.encrypt(pad(message.encode(), AES.block_size))
    iv = base64.b64encode(cipher.iv).decode('utf-8')
    ct = base64.b64encode(ct_bytes).decode('utf-8')
    return iv + ct


def aes_decrypt(iv_and_ct):
    iv = base64.b64decode(iv_and_ct[:24])
    ct = base64.b64decode(iv_and_ct[24:])
    cipher = AES.new(aes_key, AES.MODE_CBC, iv)
    pt = unpad(cipher.decrypt(ct), AES.block_size)
    return pt.decode()
```