Configuration Manual

MSc Research Project
Master of Science in Cybersecurity

Arpit Dharod
Student ID: x22186964

School of Computing
National College of Ireland

Supervisor: Eugene McLaughlin

National College of Ireland

MSc Project Submission Sheet

School of Computing

Student Name: Arpit Mukesh Dharod

Student ID: X22186964

Programme: MSc in Cybersecurity **Year:** 2024-25

Module: Research Project

Lecturer: Eugene McLaughlin

Submission Due Date: 12th August 2024

Project Title: HoneyBlow - An enhanced hybrid encryption method for messages

Word Count: **776** **Page Count: 7**

I hereby certify that the information contained in this (my submission) is information pertaining to research I conducted for this project. All information other than my own contribution will be fully referenced and listed in the relevant bibliography section at the rear of the project.
ALL internet material must be referenced in the bibliography section. Students are required to use the Referencing Standard specified in the report template. To use other author's written or electronic work is illegal (plagiarism) and may result in disciplinary action.

Signature: Arpit Mukesh Dharod

Date: 12th August 2024

PLEASE READ THE FOLLOWING INSTRUCTIONS AND CHECKLIST

Attach a completed copy of this sheet to each project (including multiple copies)	□
Attach a Moodle submission receipt of the online project submission, to each project (including multiple copies).	□
You must ensure that you retain a HARD COPY of the project, both for your own reference and in case a project is lost or mislaid. It is not sufficient to keep a copy on computer.	□

Assignments that are submitted to the Programme Coordinator Office must be placed into the assignment box located outside the office.

Office Use Only	
Signature:	
Date:	
Penalty Applied (if applicable):	

Configuration Manual

Arpit Dharod
X22186964

1 Introduction

This configuration manual consists of all the requirements for the setup of the model with the software, language of programming and libraries. It also contains the steps to setup the required model is also in this model.

2 System Info

2.1 System Specification

Model: MacBook Air M2 @ macOS Sonoma Version 14.5 (23F79)
Processor: Apple M2 chip (8 core CPU)
Memory: 8 gigabytes

2.2 Tools, Software and Programming Language

Tool: Visual Studio Code Editor.
Software: MacOS Operating System
Programming Language: Python 3.12

3 Installation

The required Python Libraries needs to be installed for performing encryption and decryption for this model.

- Python Cryptography Toolkit
    a. Pycrypto – pip install pycrypto
    b. PycryptoDome - pip install PycryptoDome
    c. Blowfish – pip install blowfish

4 Implementation

There are multiple Python files used in this model are:

1. honeyencryption.py
2. blowfish.py
3. Main.py
4. bits_by_bits.py
5. Aes.py

I. Initially, the main.py which contains all the libraries which are required are imported.

```
import hashlib
import random
from pprint import pprint
from cryptography.hazmat.primitives.ciphers import Cipher, algorithms, modes
from cryptography.hazmat.backends import default_backend
from cryptography.hazmat.primitives import padding
from os import urandom
import timeit
from Crypto.Cipher import Blowfish
from Crypto.Random import get_random_bytes
import time
from os import urandom
from Crypto.Cipher import AES
from Crypto.Util.Padding import pad, unpad
```

II. After all imports needed. The test file is used for taking input data with all the inputs for the results of main program model.

```
###Some random words with different Size bytes used for testing and analysing the Encryption methods.

# For Byte Size 7
# "Passout"

# For Byte Size 10
# "MangoBerry"

# For Byte Size 12
# "Hello_World!"

# For Byte Size 15
# "Thesis@Semester3 "

# For Byte Size 16
# MSc_Cybersecurity

# For Byte Size 65
# "Hello, this is a test sentence to fit approximately sixty-five bytes."

# For Byte Size 81
# "Sample text that should be around eighty-one bytes long for testing."

# For Byte Size 128
# "This example is designed to be exactly one hundred twenty-eight bytes, taking into account various characters, including punctuation, spaces, and special characters, to ensure accurate byte siz

# For Byte Size 209
# "A longer string of characters, spanning over two hundred bytes to reach the specific byte count required, might include diverse punctuation and spaces, depending on the exact encoding."

# For Byte Size 257
# "A longer and more detailed text example, stretching to approximately two hundred fifty-seven bytes, encompassing various letters, punctuation marks, spaces, and potentially some special charact

# For Byte Size 299
# "Creating a text block of about two hundred ninety-nine bytes involves crafting a detailed passage, which includes an array of characters, punctuation marks, and spaces to ensure the exact byte
```

III. In Honey Encryption the passwords to seeds and seeds to true message is encoded. As it is referred as most possible secure method of honeyword creation by(Nguyen, 2016; Jordan, 2021)

```
# U.S. States as potential decoy messages
states = {
    'AK': 'Alaska', 'AL': 'Alabama', 'AR': 'Arkansas', 'AS': 'American Samoa', 'AZ': 'Arizona',
    'CA': 'California', 'CO': 'Colorado', 'CT': 'Connecticut', 'DC': 'District of Columbia', 'DE': 'Delaware',
    'FL': 'Florida', 'GA': 'Georgia', 'GU': 'Guam', 'HI': 'Hawaii', 'IA': 'Iowa', 'ID': 'Idaho',
    'IL': 'Illinois', 'IN': 'Indiana', 'KS': 'Kansas', 'KY': 'Kentucky', 'LA': 'Louisiana',
    'MA': 'Massachusetts', 'MD': 'Maryland', 'ME': 'Maine', 'MI': 'Michigan', 'MN': 'Minnesota',
    'MO': 'Missouri', 'MP': 'Northern Mariana Islands', 'MS': 'Mississippi', 'MT': 'Montana',
    'NA': 'National', 'NC': 'North Carolina', 'ND': 'North Dakota', 'NE': 'Nebraska', 'NH': 'New Hampshire',
    'NJ': 'New Jersey', 'NM': 'New Mexico', 'NV': 'Nevada', 'NY': 'New York', 'OH': 'Ohio', 'OK': 'Oklahoma',
    'OR': 'Oregon', 'PA': 'Pennsylvania', 'PR': 'Puerto Rico', 'RI': 'Rhode Island', 'SC': 'South Carolina',
    'SD': 'South Dakota', 'TN': 'Tennessee', 'TX': 'Texas', 'UT': 'Utah', 'VA': 'Virginia', 'VI': 'Virgin Islands',
    'VT': 'Vermont', 'WA': 'Washington', 'WI': 'Wisconsin', 'WV': 'West Virginia', 'WY': 'Wyoming'
}

# Generate specific decoy passwords and map them to seeds and state messages
passwordsToSeeds[password + str(trueSeed - 1)] = trueSeed + 1
seedsToMessages[trueSeed + 1] = states['AL']

passwordsToSeeds[password + str(trueSeed - 2) + "1"] = trueSeed + 2
seedsToMessages[trueSeed + 2] = states['CA']

passwordsToSeeds[password.lower()] = trueSeed + 3
seedsToMessages[trueSeed + 3] = states['FL']

passwordsToSeeds[password.lower() + str(trueSeed + 1) + "3"] = trueSeed + 4
seedsToMessages[trueSeed + 4] = states['TX']

passwordsToSeeds[password.upper()] = trueSeed + 5
seedsToMessages[trueSeed + 5] = states['TN']

passwordsToSeeds[password.upper() + str(trueSeed + 2) + "5"] = trueSeed + 6
seedsToMessages[trueSeed + 6] = states['WA']

# Encrypt the message (XOR each bit with the trueSeed)
cipher = ''.join(str(int(bit) ^ (trueSeed % 2)) for bit in ms)
```

IV. As honeyword is created the credential/passwords is encrypted with blowfish encryption algorithm. The image below shows the blowfish encryption and decryption algorithm(O, 2024):

```python
import time
from cryptography.hazmat.primitives.ciphers import Cipher, algorithms, modes
from cryptography.hazmat.backends import default_backend
from cryptography.hazmat.primitives import padding
from os import urandom

# Key and Initialization Vector generation
key = urandom(16)  # Generate a random 16-byte key
iv = urandom(8)    # Generate a random 8-byte IV (Initialization Vector)

# Creating a Blowfish cipher object
cipher = Cipher(algorithms.Blowfish(key), modes.CBC(iv), backend=default_backend())

# Get user input for data encryption
user_input = input("Please enter the data you want to encrypt: ")
data = user_input.encode()  # Convert the user input to bytes

# Prepare data for encryption (with padding)
padder = padding.PKCS7(64).padder()  # 64 bit (8 byte) padding for Blowfish
padded_data = padder.update(data) + padder.finalize()

# Encrypting data
print("Blowfish encryption started")
start_encrypt_time = time.time()
encryptor = cipher.encryptor()
encrypted_data = encryptor.update(padded_data) + encryptor.finalize()
end_encrypt_time = time.time()
encrypt_elapsed_time = end_encrypt_time - start_encrypt_time

print(f"Encrypted code: {encrypted_data}")
print(f"Encryption time in seconds: {encrypt_elapsed_time:.6f}")

# Decrypting data
start_decrypt_time = time.time()
decryptor = cipher.decryptor()
decrypted_padded_data = decryptor.update(encrypted_data) + decryptor.finalize()
end_decrypt_time = time.time()
decrypt_elapsed_time = end_decrypt_time - start_decrypt_time

# Remove padding after decryption
unpadder = padding.PKCS7(64).unpadder()
decrypted_data = unpadder.update(decrypted_padded_data) + unpadder.finalize()
decrypted_string = decrypted_data.decode('utf-8')  # Convert bytes to string
```

V. After using blowfish encryption and password the time is calculated and converted into milliseconds making it easy to track the record for the encrypted data(Yin, Indulska and Zhou, 2017). Using this execution time is calculated and compared with other techniques:

```python
# Blowfish encryption
encTimeStart = timeit.default_timer()
blowfish_encrypted_data, blowfish_key, blowfish_iv = blowfishEncrypt(encrypted_data['cipher'].encode())
print(f"Blowfish Encrypted Data: {blowfish_encrypted_data}")

# Blowfish decryption
blowfish_decrypted_data = blowfishDecrypt(blowfish_encrypted_data, blowfish_key, blowfish_iv)
print(f"Blowfish Decrypted Data: {blowfish_decrypted_data.decode()}")

# Display encryption time
encTimeEnd = timeit.default_timer()
encTotalTime = (encTimeEnd - encTimeStart) * 1000  # in milliseconds
print(f"Encryption Total Time: {encTotalTime} ms")
```

VI. There are 3 possibilities of decrypting and retrieving messages which are as follows:

i. Wrong Password: No data will be decrypted and notify user with wrong password.

ii. Honeyword matched: This will notify user that password used is honeyword making him know the bogus message.

iii. Correct Password: This will make user display correct message.

Below is code of decrypting message:

```
def honeyDecrypt(password, encrypted_data):
    passwordsToSeeds = encrypted_data['passwordsToSeeds']
    seedsToMessages = encrypted_data['seedsToMessages']
    cipher = encrypted_data['cipher']

    # Function to decode the message from binary
    def decode(binary_string):
        return ''.join(chr(int(binary_string[i:i+8], 2)) for i in range(0, len(binary_string), 8))

    # Check if the password is in the dictionary
    if password in passwordsToSeeds:
        seed = passwordsToSeeds[password]
        # Decrypt the message (XOR each bit with the seed)
        ms = ''.join(str(int(bit) ^ (seed % 2)) for bit in cipher)
        # Decode the binary message
        message = decode(ms)
        if seedsToMessages[seed] == message:
            return message, "Correct password"
        else:
            return seedsToMessages[seed], "Honeyword detected"
    else:
        return None, "Wrong password"
```

VII. The evaluation of the technique is calculated using Avalanche Effect. The code to calculate Avalanche effect is in bits_by_bits.py (Kyriaskidis, 2018).

```
def pad(data, block_size):
    """Apply PKCS#7 padding to the data."""
    padding_len = block_size - (len(data) % block_size)
    padding = bytes([padding_len] * padding_len)
    return data + padding

def unpad(data):
    """Remove PKCS#7 padding from the data."""
    padding_len = data[-1]
    if padding_len > len(data):
        raise ValueError("Padding length is greater than the data length.")
    return data[:-padding_len]

def blowfish_encrypt(key, iv, plaintext):
    # Setup Blowfish in CBC mode with the given key and IV
    cipher = Cipher(algorithms.Blowfish(key), modes.CBC(iv), backend=default_backend())
    encryptor = cipher.encryptor()
    # Ensure plaintext is padded to a multiple of the block size
    padded_plaintext = pad(plaintext, algorithms.Blowfish.block_size)
    ciphertext = encryptor.update(padded_plaintext) + encryptor.finalize()
    return ciphertext

def calculate_avalanche_effect(data1, data2):
    """Calculate the Avalanche Effect as a percentage of bits changed."""
    total_bits = len(data1) * 8  # Total number of bits in one of the data blobs
    differing_bits = sum(bin(byte).count('1') for byte in (a ^ b for a, b in zip(data1, data2)))
    percentage_change = (differing_bits / total_bits) * 100
    return percentage_change

# Generate a random key and IV
key = os.urandom(8)  # Blowfish key size requirement
iv = os.urandom(8)   # IV size for Blowfish is the same as its block size (8 bytes)
```

VIII. Also, to evaluate and compare the model between AES and blowfish. AES was implemented for the same(Campos, 2021).

```python
from os import urandom
from Crypto.Cipher import AES
from Crypto.Util.Padding import pad, unpad

def enc_aes(password, key, iv):
    # Generate cipher object
    cipher = AES.new(key, AES.MODE_CBC, iv)
    # Encrypt the message with padding
    encrypted_text = cipher.encrypt(pad(password.encode('utf-8'), AES.block_size))
    return encrypted_text

def dec_aes(enc_password, key, iv):
    # Decrypt the message
    cipher = AES.new(key, AES.MODE_CBC, iv)
    decrypted_text = unpad(cipher.decrypt(enc_password), AES.block_size)
    return decrypted_text.decode('utf-8')

# Example usage:
key = urandom(16)  # AES-128 requires a 16-byte key
iv = urandom(16)   # IV should also be 16 bytes
password = "mysecretpassword"

print("AES Encryption Started.....")

encrypted = enc_aes(password, key, iv)
print("Encrypted:", encrypted)

decrypted = dec_aes(encrypted, key, iv)
print("Decrypted:", decrypted)
```

References

Campos, P.T. (2021) 'AES Implementation in Python', *Quick Code*, 13 June. Available at: https://medium.com/quick-code/aes-implementation-in-python-a82f582f51c2 (Accessed: 6 August 2024).

Jordan, K. (2021) 'Honey Encryption', *smucs*, 7 April. Available at: https://medium.com/smucs/honey-encryption-e56737af081c (Accessed: 6 August 2024).

Kyriaskidis, P. (2018) *avalanche-effect-on-Blowfish-/blowfish.py at master · PantelisKyriakidis/avalanche-effect-on-Blowfish- · GitHub*. Available at: https://github.com/PantelisKyriakidis/avalanche-effect-on-Blowfish-/blob/master/blowfish.py (Accessed: 10 August 2024).

Nguyen, V. (2016) *honey-encryption/main.py at master · victornguyen75/honey-encryption · GitHub*. Available at: https://github.com/victornguyen75/honey-encryption/blob/master/main.py (Accessed: 10 August 2024).

O, K. (2024) 'Python Blowfish Encryption Example', *DevRescue*, 14 January. Available at: https://devrescue.com/python-blowfish-encryption-example/ (Accessed: 31 July 2024).

Yin, W., Indulska, J. and Zhou, H. (2017) 'Protecting Private Data by Honey Encryption', *Security and Communication Networks*, 2017(1), p. 6760532. Available at: https://doi.org/10.1155/2017/6760532.