ECE455: Enigma 1 Simulator

Darren Chau, Shion Mizuguchi

January 20, 2024

1 Abstract

The enigma machine is a cipher machine that was used by Germany during the Second World War. Enigma was considered so secure that it was used to encrypt the most top-secret messages. Enigma was used extensively throughout the entirety of the conflict, and the Allies devoted considerable resources into cracking Enigma, while the Germans worked to ensure that Enigma stayed out of enemy hands and updated the security of the system. The Enigma machine is well known for the large number of settings that can be used with the Machine, making decryption difficult. At the same time, flaws in its design and operational use allowed the Allies to eventually crack Enigma. We have decided to implement the Enigma 1 machine in Python.

Problem Statement: How can we implement the enigma machine in Python so it properly encrypts and decrypts messages?

2 Background

The Enigma machine was invented in Germany by engineer Arthur Scherbius at the end of World War 1. The machine was released on the commercial market in the early 1920s and was soon adopted by several governments and armies worldwide. Although the Germans changed machine settings daily and made a number of cryptographic improvements, Poland was able to crack Enigma in 1932. Such a pattern would continue throughout the war, with the Germans making improvements to the machine and to operating procedures, while the Allies would work to crack Enigma in order to read German communiques. Despite the best efforts of the Germans, Allied cryptologists were able to successfully crack Enigma and read a large number of classified messages, giving them the edge in the war.

The Enigma machine is made up of three primary encryption components: the plugboard, the rotors, and the reflector. Users input letters using a keyboard. The plugboard consisted of a set of cables that swapped letters with one another. By inputting cables, a user could correspond one letter to a different one. The rotors are a set of wheels with the 26 letters of the alphabet on them. Each rotor by itself acts as a substitution cipher, swapping one letter for another. The Enigma has three of these rotors in series, so the input is swapped as it passes through each one. In addition, the rotors shift with key presses. Most enigma machines had three rotors in series, while later models had four. The first rotor would shift with every key press, the second rotor would shift whenever the first rotor made a full cycle, and the third rotor would shift whenever the second rotor made a full cycle. The reflector redirects input back into the rotors, and was a reverse of the alphabet, so an input of 'A' would become an output of 'Z'. Whenever a user pressed a key, the input would go through the plugboard, then through all three rotors. It would go into the reflector and go back through the rotors in the reverse direction, and then back through the plugboard again to get the encrypted letter. Enigma was designed so that it could both encrypt and decrypt messages. If an encrypted message was inputted with the same settings as when the message was encrypted, the machine would decrypt the message and return the original message.

For the threat model of the Enigma, there are several things that the Germans wanted to accomplish. They wanted Enigma to be capable of both decrypting and encrypting. This made it so units would only need one enigma machine, instead of needing both an encrypting and decrypting machine.

Another goal was that Enigma be difficult to decipher. To this end, they worked to increase the number of ways that a machine could be set. Additional rotor slots were added to the machines to increase the number of total settings, and additional plugboard connections were added. Another goal was to prevent attackers from determining machine settings. The Germans would give out codebooks with settings for certain days, so users would know how to properly set the machine for decryption. Operators were given strict instructions to destroy the machine and these books if there was a risk of capture.

The thousands of ways to change the settings of Enigma made it difficult to crack Enigma. However, Enigma has several flaws, both in design and in operation. First, the presence of the reflector meant that a letter could not encrypted to itself. In addition, the plugboard connections were reciprocal, so if 'A' was mapped to 'B', then 'B' would be mapped to 'A'. Besides mechanical flaws, flaws in the operating procedure made it easier for Enigma to be cracked. For example, before World War 2 the Germans only used three rotors and changed rotor position infrequently. The Germans addressed this by introducing more rotors and changing rotor position every day. In addition, the capture of an Enigma machine and codebook would allow an attacker to know the internal wirings of the machine, as well as the codes that would be used in the future. The British were able to capture intact Enigma machines and codebooks, which let them decrypt all messages for a certain period. Besides that, the Germans often dismissed concerns that Enigma had been cracked, and often failed to make operational changes.

3 Implementation

Our Python code seeks to simulate the Enigma 1, the first Enigma machine used by the German army. Enigma 1 had five rotors and room for three, with the operator swapping the rotors, location, and starting position. Our rotors have the same sequences as those used for Enigma 1. Our code allows a user to input a sequence of letters, choose three rotors out of five provided, receive an encrypted sequence, and then decrypt it.

The plugboard is represented by two sequences of 10 letters. The plugboard does not swap every letter, and generally only 10 plugboard connections were made in Enigma 1 operations. Letters in the same position in both sequences are swapped, so B is swapped with Q and vice versa.

```
// Hello.java
plugboard_a = "BCDEKMOPUG"
plugboard_b = "QRIJWTSXZH"
```

The reflector is represented by a string of the 26 letters of the alphabet. Letters are simply swapped based on position.

```
reflector = "YRUHQSLDPXNGOKMIEBFZCWVJAT" #Reflector B
```

The rotors are represented by a rotor class. They are initialized by inputting the letter sequence of the rotor. To simulate the rotor and its shifting functionality, the class has a letter sequence that maps to the alphabet (A-Z, ordered) and its corresponding alphabet sequence are set for each rotor. In other words, the two sequences mirror each other and represent a mapping. The class also includes functions to shift the position of the rotor, get the position of a letter in the letter sequence, return the letter at a position, gets the position of a letter in the alphabet sequence, and returns the letter at a position in the alphabet sequence.

```
class Rotor:
  def __init__(self, letters, alphabet):
    self.letters = letters #Letters that the rotor maps to the alphabet (A-Z, ordered)
    self.alphabet = alphabet #The alphabet (A-Z, ordered) of the particular rotor

  def shiftRotor(self):
    """Function to shift the rotor by one letter"""
    #Shifts the mapped letters of the rotor by one position
    self.letters = self.letters[1:] + self.letters[0]
    #Shifts the corresponding alphabet (A-Z, ordered) of the rotor by one position
    self.alphabet = self.alphabet[1:] + self.alphabet[0]

  def getLetterAtPos(self, position):
    """Function to return the letter from self.letters given a position"""
    return self.letters[position]

  def getAlphabetAtPos(self, position):
    """Function to return the letter from self.alphabet given a position"""
    return self.alphabet[position]

  def getLetterPos(self, letter):
    """Function to return the position of a letter from self.letters"""
    return self.letters.index(letter)

  def getAlphabetPos(self, letter):
    """Function to return the position of a letter from self.alphabet"""
    return self.alphabet.index(letter)
```

Our implementation uses one function to both encrypt and decrypt messages. On the user side, the user is prompted to enter the message they want to encrypt, and then chooses the rotors they wish to use. It then calls the encrypt function to create the encrypted message. The encrypted message is outputted, and the user is then asked if they would like to decrypt the message. If they answer yes, the decrypted message, which is the same as the original, is outputted.

```
#Loop to ask user for input message, and check if input is valid
while(True):
  message = input('Enter Message: ')
  if len(message) == 10000:
    print("Your message exceeded the character limit of 10000 characters")
  else: break

#Prints out options in rotorset to user
print("Enigma 1 uses three rotors. You can choose from rotors 1 to 5.")
print("[1] Rotor 1: EKMFLGDQVZNTOWYHXUSPAIBRCJ")
print("[2] Rotor 2: AJDKSIRUXBLHWTMCQGZNPYFVOE")
print("[3] Rotor 3: BDFHJLCPRTXVZNYEIWGAKMUSQO")
print("[4] Rotor 4: ESOVPZJAYQUIRHXLNFTGKDCMWB")
print("[5] Rotor 5: VZBRGITYUPSDNHLXAWMJQOFECK")
print("Enter number corresponding to rotor choice.")

#Loop to ask user's choice of the three rotors, and check if each input is valid
while(True):
  rotorRnum = input("Choose right rotor (1-5): ")
  if rotorRnum.isnumeric()==False or int(rotorRnum) < 1 or int(rotorRnum) > 5:
    print("Right rotor number invalid, enter number between 1 to 5." + "\n")
  else: break
while(True):
  rotorMnum = input("Choose middle rotor (1-5): ")
  if rotorMnum.isnumeric()==False or int(rotorMnum) < 1 or int(rotorMnum) > 5:
    print("Middle rotor number invalid, enter number between 1 to 5." + "\n")
  else: break
while(True):
  rotorLnum = input("Choose left rotor (1-5): ")
  if rotorLnum.isnumeric()==False or int(rotorLnum) < 1 or int(rotorLnum) > 5:
    print("Left rotor number invalid, enter number between 1 to 5" + "\n")
  else: break
```

```
#Encrypts message with user’s chosen rotorset
encrypted_message = encrypt(message, rotorRnum, rotorMnum, rotorLnum)
print(f"Encrypted message: {encrypted_message}")

#If user wishes to decrypt message, loop takes user’s choice and checks if valid
while(True):
  answer = input(’Would you like to decrypt the message? (Y/N): ’).upper()
  if answer=="Y" or answer=="YES":
    decrypted_message = encrypt(encrypted_message, rotorRnum, rotorMnum, rotorLnum)
    print(f"Decrypted message: {decrypted_message}")
    break
  if answer=="N" or answer=="NO":
    print("Goodbye")
    break
  else:
    print("Invalid input, try again")
```

The start function is used to start a session of encryption and decryption by choosing the rotors. It is used to reset the rotors back to their initial configuration, which is most useful for decryption since the rotors need to be in the same position as when the message was encrypted. This function initializes the rotor objects, and then assigns the positions of the chosen rotors based on the user input.

```
def start(rotorRnum, rotorMnum, rotorLnum):
  """Function to start a new session of encryption/decryption by choosing and resetting
      rotors"""
  #Creates all five rotors
  rotor1 = Rotor(str_rotor1, alphabet)
  rotor2 = Rotor(str_rotor2, alphabet)
  rotor3 = Rotor(str_rotor3, alphabet)
  rotor4 = Rotor(str_rotor4, alphabet)
  rotor5 = Rotor(str_rotor5, alphabet)

  #default three rotorset
  rotorR = rotor1
  rotorM = rotor2
  rotorL = rotor3

  #Assigns the right rotor, middle rotor, and left rotor based on user’s choice
  #rotorRnum: right rotor choice, rotorMnum: middle rotor choice, rotorLnum: left rotor choice
  if rotorRnum == "1": rotorR = rotor1
  elif rotorRnum == "2": rotorR = rotor2
  elif rotorRnum == "3": rotorR = rotor3
  elif rotorRnum == "4": rotorR = rotor4
  elif rotorRnum == "5": rotorR = rotor5
  if rotorMnum == "1": rotorM = rotor1
  elif rotorMnum == "2": rotorM = rotor2
  elif rotorMnum == "3": rotorM = rotor3
  elif rotorMnum == "4": rotorM = rotor4
  elif rotorMnum == "5": rotorM = rotor5
  if rotorLnum == "1": rotorL = rotor1
  elif rotorLnum == "2": rotorL = rotor2
  elif rotorLnum == "3": rotorL = rotor3
  elif rotorLnum == "4": rotorL = rotor4
  elif rotorLnum == "5": rotorL = rotor5

  return rotorR, rotorM, rotorL
```

The encrypt function is the primary function that encrypts or decrypts a message. It receives the user input of the message and chosen rotors. Our function first converts all letters in the message to uppercase, to ensure consistency and for simplicity of implementation. It then initializes the rotors.

The input letter is first checked to see if it goes through the plugboard, swapping if the letter is found in the plugboard. It is then fed through the rotors, swapping letters as it goes through. For each letter it shifts the rotors, shifting the right rotor with each letter, and shifting the middle and left rotor based on the rotations of the other rotors – after the right rotor completes a full rotation, the middle rotor shifts once, and when the middle rotor completes a full rotation, the left rotor shifts once. When it reaches the reflector, the letter is swapped and then fed through the rotors in reverse. Finally, it reaches the plugboard again, and is then checked if it goes through, and then the encrypted letter is added to the encrypted message. Once it goes through the entire input message, it returns the full encrypted sequence. The user can optionally choose to decrypt the message once their input message is fully encrypted, which will return their original message.

```
def encrypt(message, rotorRnum, rotorMnum, rotorLnum):
  """Function to encrypt (or decrypt) a given message"""
  message = message.upper()
  encrypted_message = ""

  #Creates right, middle, and left rotors based on user input
  rotorR, rotorM, rotorL = start(rotorRnum, rotorMnum, rotorLnum)
  #Counts if the right rotor and middle rotor made a full rotation
  count_rotorR_shifts = 9 #Initial position/offset of right rotor
  count_rotorM_shifts = 22 #Initial position/offset of middle rotor

  for letter in message:
    #Checks if letter in message string is an alphabet
    #If a user inputs a character that is not an
    #alphabet, it will be kept as is (e.g. spaces)
    if letter in alphabet:
      #Shift right rotor with every key entered
      rotorR.shiftRotor()
      count_rotorR_shifts += 1
      #Shift middle rotor when right rotor fully rotates
      if count_rotorR_shifts == 26:
        rotorM.shiftRotor()
        count_rotorM_shifts += 1
        count_rotorR_shifts = 0 #Reset right rotor's shifts
      #Shifts middle and left rotor when middle rotor fully rotates
      elif count_rotorM_shifts == 26:
        rotorM.shiftRotor()
        rotorL.shiftRotor()
        count_rotorM_shifts = 1 #Since middle rotor shifts, set to 1

      #First checks if inputted letter can be swapped with the plugboard letters
      if letter in plugboard_a:
        letter = plugboard_b[plugboard_a.index(letter)]
      elif letter in plugboard_b:
        letter = plugboard_a[plugboard_b.index(letter)]

      position = alphabet.index(letter)

      #Letter through the three rotors, from the right rotor to left rotor
      letter = rotorR.getLetterAtPos(position)
      position = rotorR.getAlphabetPos(letter)
      letter = rotorM.getAlphabetAtPos(position)
      position = rotorM.getAlphabetPos(letter)
      letter = rotorM.getLetterAtPos(position)
      position = rotorM.getAlphabetPos(letter)
      letter = rotorL.getAlphabetAtPos(position)
      position = rotorL.getAlphabetPos(letter)
      letter = rotorL.getLetterAtPos(position)

      #Letter through the reflector
      position = rotorL.getAlphabetPos(letter)
      letter = reflector[position]

      #Letter through the three rotors, now in the opposite direction from left
      #to right rotor
      position = alphabet.index(letter)
      letter = rotorL.getAlphabetAtPos(position)
      position = rotorL.getLetterPos(letter)
      letter = rotorL.getAlphabetAtPos(position)
      position = rotorL.getAlphabetPos(letter)
      letter = rotorM.getAlphabetAtPos(position)
      position = rotorM.getLetterPos(letter)
      letter = rotorM.getAlphabetAtPos(position)
      position = rotorM.getAlphabetPos(letter)
      letter = rotorR.getAlphabetAtPos(position)
      position = rotorR.getLetterPos(letter)
      letter = rotorR.getAlphabetAtPos(position)
      position = rotorR.getAlphabetPos(letter)
      letter = alphabet[position]

      #Checks if outputted letter from right rotor can be swapped with the
      #plugboard letters
      if letter in plugboard_a:
        letter = plugboard_b[plugboard_a.index(letter)]
      elif letter in plugboard_b:
        letter = plugboard_a[plugboard_b.index(letter)]

      encrypted_message += letter
    return encrypted_message
```

4 Conclusion

Our implementation of Enigma 1 was a success, and we were able to get great insight into the operations of the machine. We were also able to simulate both the security advantages and disadvantages that the Enigma 1 imposed. For future projects, we could simulate more advanced enigma machines such as the four-rotor variant. However, a more interesting project could be simulating the British “Bombe” computer, which was used to decipher Enigma messages. The “Bombe” was the culmination of years of effort, and it will be interesting to get more insight into how the British worked to decipher Enigma messages.