DAT510 - Assignment 1

Andreas Pedersen - 265200

September 13, 2024

Abstract

In this project, cryptographic techniques are implemented and the effect of changing a single character in the text to be encrypted and its effect on the ciphertext (the avalanche effect) is studied. One substitution cipher and one transposition cipher are implemented. This is Caesar cipher and row transposition cipher. Their cryptographic procedure is described and their weaknesses are accounted for. The result of encryption is that the ciphers used in combination do indeed encrypt the plaintext, but their weaknesses make the ciphertext vulnerable to different kinds of analyses and attacks. Further, the result of the avalanche effect is not that great, as there is no avalanche effect due to the lack of diffusion in the ciphers. Repeated encryption is also studied, but the avalanche effect is still unchanged. The implementation of the ECB block cipher was also done, which enhanced the security, however, the avalanche effect remained unchanged.

1 Introduction

The overall goal of this project is to obtain a better understanding of classical cryptographic techniques and the avalanche effect. This includes the algorithms for certain transposition and substitution ciphers. Further, the avalanche effect will be evaluated on these cipher implementations. The avalanche effect is considered one of the desirable properties of any encryption algorithm. A slight change in either the key or the plain text should result in a significant change in the cipher text [2]. The terms *diffusion* and *confusion* are also a central part of this project. These are the two main building blocks for cryptographic systems. Confusion involves making the relation between the cipher key and ciphertext as complex as possible. It aims to make the way the key was used to generate the ciphertext as confusing as possible for any potential attackers. So even if the attacker gets any insight into the statistics of the ciphertext, it would be very difficult to work out what the key could be. Diffusion is a general cryptographic principle where each letter of plaintext affects many letters of ciphertext, making it harder to detect patterns and therefore enhancing security [4]. The avalanche effect is a specific aspect of diffusion.

The ciphers implemented in this project are the well-known substitution cipher *Caesar cipher* and the transposition cipher *row transposition cipher*. The substitution technique works by replacing the letters in the plaintext with other letters or numbers and symbols. The Caesar cipher is the simplest and earliest substitution cipher used by Julius Caesar [5]. It is about replacing each letter of the alphabet with another one further down the alphabet. The decision about how far down is based on a so-called *Caesar shift key*. The transposition technique, also called permutation, works by transposing the plaintext, i.e. rearranging the letters so it looks more like gibberish. This is done without actually altering the letters themselves. A row transposition cipher is one transposition cipher that works by having the plaintext written out in a rectangle, row by row, then reading the text column by column, with a key determining the order of columns. Block ciphers will also be used in this project. Block ciphers split the plaintext into blocks and encrypt each block with a chosen encryption algorithm. This enhances the security of encryption.

2 Design and Implementation

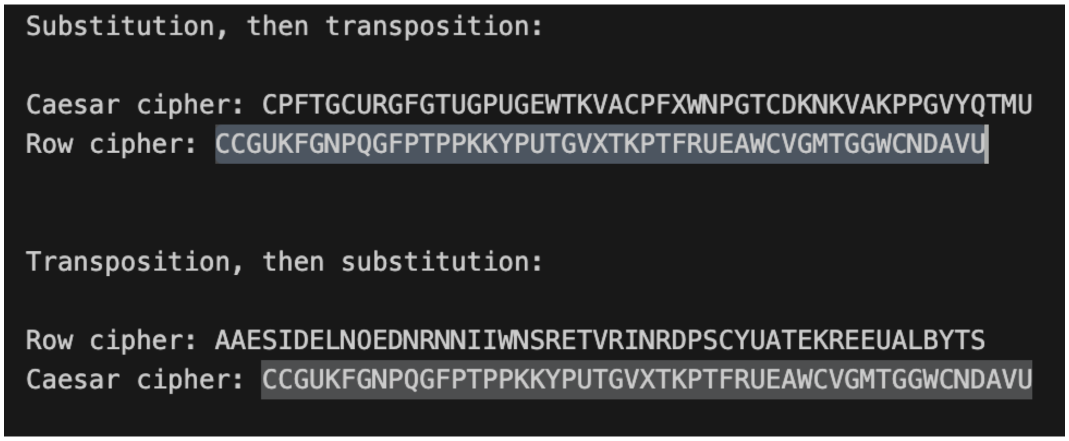
In this part, the implementation of the ciphers will be accounted for and the evaluation of the avalanche effect on these will be discussed. First off the plaintext to be encrypted and the keys used for the encryption in this project are shown in Figure 1.

Plaintext: “Andreas Pedersen Security and Vulnerability in Networks”
Numeric key: 13452
Caesar key: 2

Figure 1: The plaintext to be encrypted and keys used.

2.1 Apply Encryption

For the encryption part of this project, we were provided two options for applying encryption to an input text. **Option A** is first to apply a transposition cipher to the input text, then use a substitution cipher on the transposed ciphertext. **Option B** is to do the opposite order, namely first applying a substitution cipher to the input text, and then using a transposition cipher on the substituted ciphertext. The combination used for encryption in this project is **option B**. The cipher algorithms used are Caesar cipher for the substitution part, and row transposition cipher for the transposition part. Both implementations are described in Section 2.1.1 and Section 2.1.2 respectively. I chose the B option because, with Caesar substitution cipher first, you substitute the original letters in the input text with other letters. This differs from the row transposition cipher which just rearranges the letters (the original letters are still there after the first cipher). After implementing these ciphers I quickly saw that the resulting cipher text would be the same for both options, as shown in Figure 2. In my opinion, it is better to first substitute and then do transposition since the original letters of the input text can not be found in both ciphertexts.



```
Substitution, then transposition:

Caesar cipher: CPFTGCURGFGTUGPUGEWTKVACPFXWNPGTCDKNKVAKPPGVYQTMU
Row cipher: CCGUKFGNPQGFPTPPKKYPUTGVXTKPTFRUEAWCVGMTGGWCNDAVU


Transposition, then substitution:

Row cipher: AAESIDELNOEDNRNNIIWNSRETVRINRDPSCYUATEKREEUALBYTS
Caesar cipher: CCGUKFGNPQGFPTPPKKYPUTGVXTKPTFRUEAWCVGMTGGWCNDAVU
```

Figure 2: Both resulting ciphertexts of option B (top) and option A (bottom).

In terms of security, both options are equally good in this case since the resulting ciphertext will be the same either way. This is due to the fact that the ciphers run independently and the operations do not interfere with each other. They act on the input text in different ways. In the Caesar cipher, the same letter is always shifted to the same corresponding letter, as the cipher does a fixed shift to each letter in the alphabet. For instance, the letter ’A’ will always become a ’C’ if the key for shifting is 2. This makes the Caesar cipher vulnerable to frequency analysis, and also brute forcing. Frequency analysis is often done to see how often a certain letter occurs within the ciphertext. This can be further used in combination with the most commonly used letters in the alphabet to deduce the text. When it comes to brute force attacks, Caesar cipher is very vulnerable to this due to its limited number of possible keys. For the English alphabet, an attacker could easily try all 26 possible shifts and quickly find the correct one, and therefore obtain the plaintext. The transposition cipher only rearranges the letters in the input text in a certain way (in most cases based on a numeric key), so no change in the letters whatsoever other than their position in the string.

2.1.1 Caesar Cipher Implementation

The Caesar cipher is made as its own function. At the start, I remove the spaces and make the input text all uppercase to generalize it. Then I initialize the ciphertext as an empty string. Further, the function loops through the input text letter by letter, and checks if it is an actual letter, if not (i.e. a symbol) it is just appended to the ciphertext. Based on the Caesar key (which in this case is 2), the letters are shifted 2 letters down the alphabet. The actual procedure of this is done using Unicode (ASCII) with the Python functions `ord()` and `chr()`. The `ord()` function converts a character to its corresponding ASCII value, and `chr()` converts the resulting ASCII value back to a character. The line of code `chr((ord(letter) + key - 65) % 26 + 65)` does the actual shifting. The value 65 is used because for the letter `"A"`, `ord("A")` returns 65, and 66 for `ord("B")` etc. The subtraction of 65 is done so that `"A"` starts at 0. The modulo operation `% 26` makes sure that the shift wraps around if it exceeds the end of the (English) alphabet. After ensuring the shift result is within the range of 0 to 25 (A to Z), 65 is added back to get the correct ASCII value for the letter. Then the shifted letter is appended to the ciphertext string. When all the letters are looped through, the cipher is done, and the function returns the Caesar ciphertext.

2.1.2 Row transposition Cipher Implementation

Row transposition cipher works by having the plaintext written out in a grid, row by row, then reading the text column by column, with a numeric key determining the order and number of columns.

The row transposition cipher is also made as its own function. At the start, I remove the spaces and make the input text all uppercase to generalize it. Then I determine the number of columns based on the length of the key. Next, the number of rows is calculated by dividing the length of the text by the number of columns and rounding up to the nearest integer. Rounding up ensures that the text can fit in the grid needed for this cipher. If the text does not exactly fit in the grid, I pad the text with underscores so it fits. Then I convert the text into a list of characters and reshape this list into the 2D grid using the numbers of rows and columns calculated earlier. Further, I retrieve the indices that would sort the key, which is used to reorder the columns of the grid. Then I utilize the function `.T` in NumPy, which transposes a matrix, meaning rows become columns and columns become rows. Another function `.flatten()` is then used, which takes the 2D array and converts it into a 1D array. The next step is to then join all the characters in this array to a string, which becomes the ciphertext. Finally, I remove the padded underscores after the cipher is done and the function returns the ciphertext.

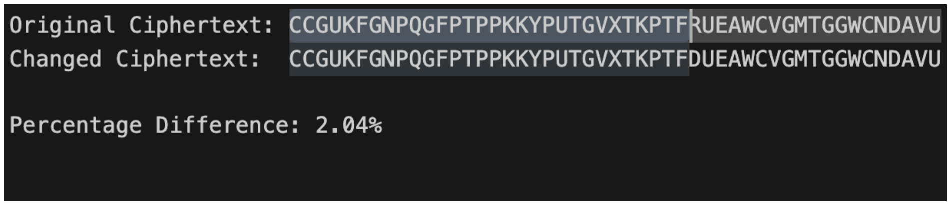
2.2 Evaluate the Avalanche Effect

This section is about the avalanche effect. What happens if I change something in the input text? Will the ciphertext remain unchanged, or will the change spread making the ciphertext more unpredictable and complex? Will additional rounds of encryption make the ciphertext more robust?

2.2.1 Initial Avalanche Effect Analysis

I changed the input text by changing one single character, and then re-encrypting the modified input using the same combination (option B) as described in Section 2.1. To evaluate the avalanche effect here I compare the original and modified ciphertexts by calculating the percentage of differing characters. This is done in a separate function. This function takes two ciphertexts as parameters, so I send in both the original ciphertext and the changed one. Then I make sure the shortest ciphertext is considered, but they should be equal in length. Next, I loop through each letter in the original ciphertext and check if it matches each letter in the changed ciphertext. If it does not match, a variable called `differences` is incremented by 1. Lastly, the percentage difference is returned by `(differences / length) * 100`.

As shown in Figure 3 there is a 2.04% difference with a single character change. This percentage represents one character and there is actually **no avalanche effect at all**. The changed character in the input text is changed in the ciphertext, but only that *one* character. The rest is untouched, which means that one character change had **no impact** on the rest of the ciphertext. Why there is no avalanche effect is discussed further in Section 2.3.



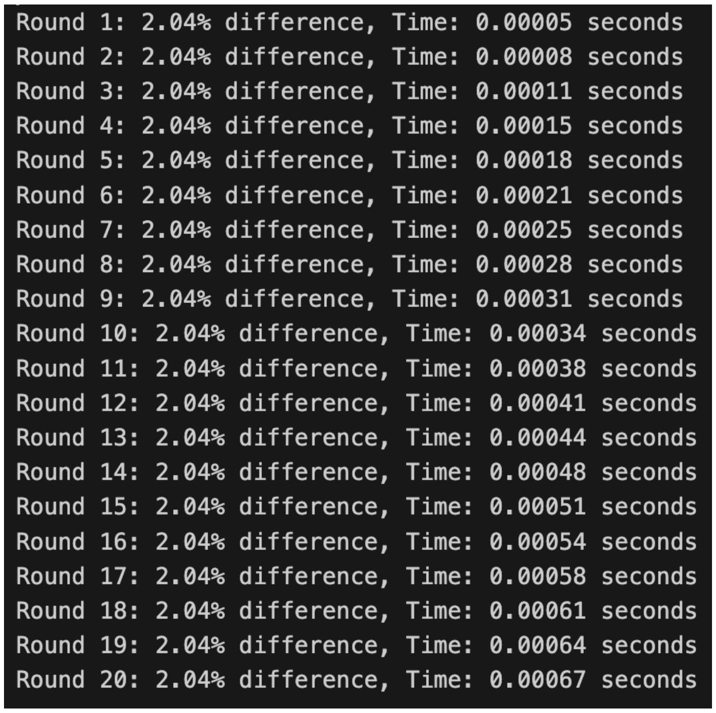
```
Original Ciphertext: CCGUKFGNPQGFPTPPKKYPUTGVXTKPTFRUEAWCVGMTGGWCNDAVU
Changed Ciphertext:  CCGUKFGNPQGFPTPPKKYPUTGVXTKPTFDUEAWCVGMTGGWCNDAVU

Percentage Difference: 2.04%
```

Figure 3: Percentage difference with one character change.

2.2.2 Repeated Avalanche Effect Analysis

In this section, I repeat the encryption 20 times and check if the avalanche effect changes with each additional round of encryption. This is done in a function that takes the number of rounds, original text, changed text, and the keys for encryption as parameters. The function then does the same encryption combination as before for the number of rounds set, where each new round encrypts the ciphertexts returned from the previous round. When the encryption is done, the percentage difference function is run on both ciphertexts. I store the round number, percentage difference and the time taken in a list where each entry is a dictionary. Finally, the repetition function returns this list, and I can easily index the dictionary entries in the list to get the results, where each entry is a round. Figure 4 shows this.

The output shows that even with 20 rounds of encryption, there is **still no avalanche effect**. The percentage difference is the same in each round. I also tried 250 rounds, in which I got the same result; 2.04% each time. This implies that it does not matter if you encrypt 1 round, 20 rounds, or even more. Why there is no avalanche effect with additional rounds is also discussed further in Section 2.3.



```
Round 1: 2.04% difference, Time: 0.00005 seconds
Round 2: 2.04% difference, Time: 0.00008 seconds
Round 3: 2.04% difference, Time: 0.00011 seconds
Round 4: 2.04% difference, Time: 0.00015 seconds
Round 5: 2.04% difference, Time: 0.00018 seconds
Round 6: 2.04% difference, Time: 0.00021 seconds
Round 7: 2.04% difference, Time: 0.00025 seconds
Round 8: 2.04% difference, Time: 0.00028 seconds
Round 9: 2.04% difference, Time: 0.00031 seconds
Round 10: 2.04% difference, Time: 0.00034 seconds
Round 11: 2.04% difference, Time: 0.00038 seconds
Round 12: 2.04% difference, Time: 0.00041 seconds
Round 13: 2.04% difference, Time: 0.00044 seconds
Round 14: 2.04% difference, Time: 0.00048 seconds
Round 15: 2.04% difference, Time: 0.00051 seconds
Round 16: 2.04% difference, Time: 0.00054 seconds
Round 17: 2.04% difference, Time: 0.00058 seconds
Round 18: 2.04% difference, Time: 0.00061 seconds
Round 19: 2.04% difference, Time: 0.00064 seconds
Round 20: 2.04% difference, Time: 0.00067 seconds
```

Figure 4: 20 rounds of encryption with percentage difference and computational time.

2.3 Analyze the Avalanche Effect

The avalanche effect describes how a small change in the input text results in a large change to the output, ideally affecting a significant portion of the output. However, that is not the case with the simple ciphers implemented in this project. As seen in previous sections the avalanche effect is unchanged even with additional rounds of encryption. The primary reason why substitution and transposition ciphers have no avalanche effect is because they lack effective *diffusion* mechanisms. Substitution ciphers change individual characters, but the changes remain localized. Transposition ciphers only rearrange characters without any spread of the effects of a change.

When using these simple classical ciphers, there is no avalanche effect because these ciphers operate in a character-by-character or position-based manner, without spreading changes to any other characters in the ciphertext. As mentioned before, the Caesar cipher has fixed shifting of the letters, so the changed letter will be shifted to a new one, but the others will be shifted to the same letter as they did before. This is present in Figure 3, where it is easy to see that the letters highlighted remain the same, but the one changed character can be found where the highlighting stops, the rest of the characters following that one changed character also remains the same. The row transposition cipher only rearranges the letters using its grid-transposition algorithm, so therefore there will be no change in the other unchanged letters a.k.a no avalanche effect.

2.4 Optimize Avalanche Effect with Reasonable Computation

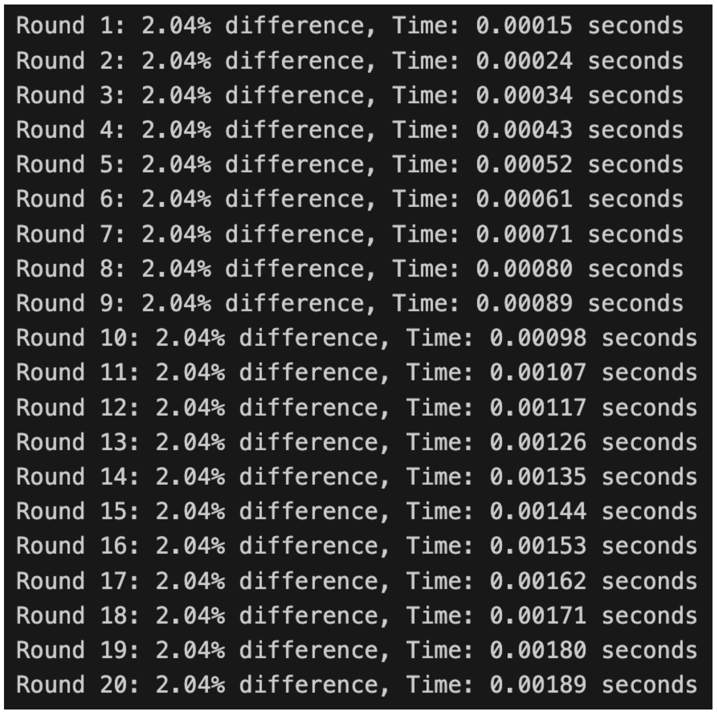
For the scenario in this project, there is no avalanche effect in the case of the classical substitution and transposition ciphers, and thus, the idea of optimal balance between a strong avalanche effect and efficient computation requires reevaluation. There is really nothing to optimize in this scenario. The computational cost is negligible, and the lack of an avalanche effect is the result of the simple ciphers. As seen from the output in previous sections, the avalanche effect is not present. Because of these limitations, there is no meaningful trade-off between computational efficiency and strength in these ciphers. Optimizing for the avalanche effect is important for modern encryption algorithms. With substitution and transposition ciphers, you do not get this effect at all, and trying to optimize for it would not make a difference. Any attempt to optimize for a stronger avalanche effect would mean altering the fundamental structure of the algorithms, making them more complex. However, this would go against the simplicity of the ciphers without providing significant benefits in terms of security.

2.5 Enhance Security with Block Ciphers

Both substitution and transposition ciphers have their weaknesses. As mentioned earlier, they are vulnerable to frequency analysis (pattern recognition) and brute forcing. Their lack of diffusion also makes them weak as the avalanche effect is not present. Substitution ciphers have a predictable structure and are vulnerable to statistical attacks if enough ciphertext is available. Both substitution and transposition ciphers suffer from insufficient confusion and diffusion, concepts central to more secure modern cryptographic methods like block ciphers. A block cipher is an encryption algorithm that takes a fixed size of input say b bits and produces a ciphertext of b bits again. If the input is larger than b bits it can be divided further [1].

To enhance the security of the encryption combination, I will implement the block cipher Electronic Code Book (ECB). It is the easiest block cipher mode of functioning. ECB is easier because of direct encryption of each block of input plaintext and output is in the form of blocks of encrypted ciphertext [1][3]. In my implementation, I have a function that does the ECB encryption and one function for padding. The padding function pads the plaintext to make sure that the blocks are of equal length. In this function, I first calculate the amount of padding needed by taking the remainder when dividing the length of the plaintext by the set block size. Then it returns the plaintext with the added padding. In the ECB function, I first remove any spaces for simplicity. Then I utilize the padding function on the plaintext, which sets up the text to be divided into the set block size. Then, the plaintext is split into the blocks, and the encryption combination as earlier (Caesar cipher, then row transposition) is applied for each block. Each encrypted block is appended to a ciphertext string (which starts as empty) and the ciphertext is then returned when all the blocks are encrypted.

This somewhat enhances the security of the ciphertext as it adds another layer of encryption, but when it comes to the avalanche effect, there is really no avalanche effect with ECB. I repeated the encryption 20 times, to see if it had any impact on the avalanche effect and the overall security. I utilized the same repeat function described in Section 2.2.2. Figure 5 shows the 20 rounds with ECB and that there is no avalanche effect. This is due to the same reasons as described in Section 2.3 and that ECB does not really do any more encryption than splitting into blocks and encrypting each block using the same encryption algorithm(s). The limitations of ECB are that it is still vulnerable to frequency analysis and has no diffusion.



```
Round 1: 2.04% difference, Time: 0.00015 seconds
Round 2: 2.04% difference, Time: 0.00024 seconds
Round 3: 2.04% difference, Time: 0.00034 seconds
Round 4: 2.04% difference, Time: 0.00043 seconds
Round 5: 2.04% difference, Time: 0.00052 seconds
Round 6: 2.04% difference, Time: 0.00061 seconds
Round 7: 2.04% difference, Time: 0.00071 seconds
Round 8: 2.04% difference, Time: 0.00080 seconds
Round 9: 2.04% difference, Time: 0.00089 seconds
Round 10: 2.04% difference, Time: 0.00098 seconds
Round 11: 2.04% difference, Time: 0.00107 seconds
Round 12: 2.04% difference, Time: 0.00117 seconds
Round 13: 2.04% difference, Time: 0.00126 seconds
Round 14: 2.04% difference, Time: 0.00135 seconds
Round 15: 2.04% difference, Time: 0.00144 seconds
Round 16: 2.04% difference, Time: 0.00153 seconds
Round 17: 2.04% difference, Time: 0.00162 seconds
Round 18: 2.04% difference, Time: 0.00171 seconds
Round 19: 2.04% difference, Time: 0.00180 seconds
Round 20: 2.04% difference, Time: 0.00189 seconds
```

Figure 5: 20 rounds of ECB encryption with percentage difference and computational time.

3 Discussion

The results of this project show that Caesar cipher and row transposition cipher have no avalanche effect. Both ciphers are vulnerable to frequency analysis and other attacks due to their lack of effective confusion and diffusion. The repeated encryption, trying up to 250 rounds, confirmed that additional rounds do not enhance the avalanche effect. Implementing the ECB block cipher did not enhance the avalanche effect issues present in the simple ciphers, but added minimal extra security. The repeated rounds of ECB encryption also showed that additional rounds do not enhance the avalanche effect. Optimizing for a better avalanche effect within the constraints of the simple ciphers proves impractical. Attempting to enhance diffusion would likely overwrite the fundamental structure of these ciphers. This limitation highlights the necessity for more advanced cryptographic algorithms that are specifically designed to provide diffusion and confusion, important for modern cryptographic security.

4 Conclusion

In conclusion, the implemented ciphers are not that complex. They are vulnerable to different kinds of attacks, and after the results gathered in this project, not that secure. The avalanche effect is not present when using these classical ciphers either, and considering this effect is commonly seen in more complex modern algorithms, it is not good to have no avalanche effect at all. The ciphers do indeed encrypt the plaintext and make it look like random nonsense, however, since they are simple in design and vulnerable, I would improve the encryption procedure with different ciphers that are more secure. I would not recommend these simple ciphers. The same for the ECB block cipher; I would try different block ciphers and then evaluate the avalanche effect. In short, the focus would be on the confusion and diffusion mechanisms as future improvements.

References

[1] Block Cipher modes of Operation. https://www.geeksforgeeks.org/block-cipher-modes-of-operation/, 2023. [Accessed 12-09-2024].

[2] Avalanche Effect in Cryptography. https://www.geeksforgeeks.org/avalanche-effect-in-cryptography, 2024. [Accessed 05-09-2024].

[3] Chunming Rong. Block cipher operation. Lecture 07, "Block Ops". [Accessed 12-09-2024].

[4] Chunming Rong. Block ciphers and the data encryption standard. Lecture 04, "Block DES". [Accessed 12-09-2024].

[5] Chunming Rong. Classical encryption techniques. Lecture 03, "Classical". [Accessed 12-09-2024].