`

APPROVAL OF HONORS PROGRAM SENIOR PROJECT

Candidate

Delfina Conde Latini

Project Title

Simulating the World War II Cipher Machine, "Lorenz"

This Senior Project is approved as acceptable

Project Director

Dr. Bill Yankosky

Committee Member

Dr. Aaron Allen

Committee Member

Dr. Katie Beeman

Honors Program Director

Dr. Bill Yankosky

Honors Program Assistant Director

Dr. Fred Sanborn

April 29, 2025

Simulating the German World War II Cipher Machine, "Lorenz"

Delfina N. Conde Latini

Advised by: Dr. Bill Yankosky

Senior Honors Thesis

May 1, 2025

Table of Contents:

1. Abstract……...…….3

2. Definitions…….……...……4

3. Basics and History……...…6

4. How does The Lorenz Cipher Work?……...…...9

5. Mathematical Strength of The Lorenz Cipher……...……22

6. Programming……..23

7. Conclusion……...……..41

8. References…….……...…..42

9. Appendices ……...…..…..……...……43

Simulating the German World War II Cipher Machine, “Lorenz”

1. Abstract

This paper explores the history and importance of the World War II German cipher machine, Lorenz. It explains how the machine worked to secure military communications during the war and why it was a critical tool for the German forces. Using research about the machine’s design and impact, a program was created in C++ to recreate its encryption process. The paper breaks down the machine’s mechanics in easy-to-understand terms, helping readers appreciate its role in history and the field of cryptography.

2. Definitions

Before exploring the details of the Lorenz Cipher Machine, it is important to define key terms that form the foundation of cryptology. These terms will be referenced throughout this paper:

General Cryptography Terms *(Singh, 1999)*:

1. **Cryptology**: The science of creating and analyzing secret writing in all its forms.
2. **Plaintext**: The original, unencrypted form of a message.
3. **Ciphertext**: The encrypted, secret form of a message.
4. **Encryption**: The process of converting plaintext into ciphertext.
5. **Decryption**: The process of converting ciphertext back into plaintext.
6. **Key**: A piece of information, often a string of numbers or letters, that dictates how a cipher operates. Typically, the same key is used for both encryption and decryption, or one can be derived from the other..
7. **Cryptosystem**: A specific method used for encryption and decryption.
8. **Cryptanalysi**s: The science of deciphering plaintext from ciphertext without access to the encryption key.

Terms Related to the Lorenz Cipher Machine *(National Museum of Computing, n.d.)*:

9. **Lorenz Cipher Machine**: A German encryption device used during World War II to encode high-level military communications.
10. **Teleprinter**: A machine used to send messages over telephone or radio lines and print them on tape.

11. **Wheels**: The 12 rotating parts in the Lorenz machine that determine how the message is scrambled. Named as 5x"S," 2x"Mu," and 5x"K."
12. **Wheel Setting**: The starting position of each wheel when encrypting a message.
13. **Cams**: Small mechanical parts on each wheel that adjust its settings for encryption.
14. **Key (in Lorenz)**: A 501-bit sequence used to set the machine's encryption pattern.
15. **Wheel Breaking**: The process of figuring out the wheel settings to crack the encryption.
16. **Turingery**: Alan Turing's method for figuring out the settings of the Lorenz wheels manually.
17. **Modulo Two Arithmetic (XOR)**: A simple binary operation used in encryption.
18. **Cribs**: Guessing part of a message (e.g., a common phrase) to break the code.

3. Basics and History

The Lorenz Cipher Machine, developed by the German company C. Lorenz AG in the early 1940s, was a groundbreaking encryption device used to secure high-level military communications during World War II. Unlike the more widely known Enigma Machine, which was used on the battlefield by field units and naval commanders, the Lorenz Machine—codenamed "Tunny" by the Allies—was reserved for strategic messages sent between Adolf Hitler, his generals, and other top-ranking officials. This made it a vital tool for coordinating large-scale operations across Europe and North Africa.

The Machine first came into use in 1941, representing a significant advancement in cryptographic technology. It moved beyond simple substitution ciphers to a highly complex mechanized encryption method. Messages were typed into a teleprinter connected to the Lorenz Machine, which used twelve rotating wheels to generate a stream cipher. These wheels created a pseudo-random sequence of characters that hide the original message, transforming it into ciphertext that seemed random and undecipherable. To decrypt the message, the Lorenz Machine's design required precise synchronization between the sending and receiving Machines. Both devices needed identical initial settings for their 12 wheels. Any deviation resulted in garbled messages. This reliance on mechanical accuracy added an additional layer of operational complexity, underscoring the Machine's dependence on meticulous preparation and distribution of key settings.

The design of the Lorenz Machine ensured an enormous number of possible settings—over 16 million, million, million combinations, or 10^{19}. Each wheel had a unique

number of positions and was covered in pins that could be adjusted to create new encryption patterns. This staggering complexity gave German operators great confidence in the Machine's security, believing it to be unbreakable.

However, the Allies intercepted Lorenz-encrypted messages and quickly recognized their importance. The messages were transmitted using teleprinter signals, often over radio waves, and used binary code—a precursor to modern computer language. The British codebreakers at Bletchley Park[1] nicknamed these intercepted transmissions "Fish" and the cipher itself "Tunny."

Cracking the Lorenz Cipher was a big challenge due to its complexity, but it became a turning point in the war. The first major breakthrough came in 1941, when a German operator mistakenly re-sent a message with minor changes but without altering the Machine's settings—a breach of protocol that allowed Allied cryptographers to identify patterns in the encryption. This error gave the team at Bletchley Park, led by brilliant minds like John Tiltman and William Tutte, a critical foothold to study the Machine's inner workings.

To speed up the meticulous process of breaking Lorenz messages, engineer Tommy Flowers built *Colossus*, the world's first programmable electronic computer, in 1944. Colossus could analyze intercepted Lorenz-encrypted messages at unprecedented speeds, processing up to 1,000 characters per second. This innovation not only allowed the Allies to decrypt German messages rapidly but also marked the birth of modern computing.

[1] Bletchley Park was the British government's secret code breaking center during World War II. Located in Buckinghamshire, England, it became renowned for its crucial role in deciphering Axis communications—messages from the military and governments of Germany, Italy, Japan, and their allies.

The intelligence gathered from Lorenz-decrypted messages proved instrumental in shaping Allied strategies, including the planning of the D-Day invasion. The Machine's design and the efforts to break its code illustrate a pivotal moment in the history of cryptography and the war. It stands as a testament to both German engineering and the ingenuity of the Allied codebreakers who unraveled its secrets *(Singh, 1999)*.

4. How does The Lorenz Cipher work?

As mentioned before, the Lorenz Cipher Machine was a cryptographic device used by the German Army during World War II to secure high-level communications. It was a complex Machine designed to encode messages into a secure format (ciphertext) and decode ciphertext back into readable text (plaintext). Despite its complexity, British codebreakers at Bletchley Park famously succeeded in breaking its encryption, leading to significant advances in cryptography and computer science.

Key Concepts and Components

To understand how the Lorenz Cipher encrypted messages, it's crucial to grasp its core components and underlying principles. Each part of the Machine played a specific role in securing communications. Let’s explore the essential concepts that made this cipher work.

1. **International Telegraph Alphabet No. 2 (ITA2)**

   The International Telegraph Alphabet No. 2 (ITA2) is a 5-bit binary encoding system used for transmitting text over telegraph lines, as shown in *Figure 1*. Each character is represented by a unique 5-bit sequence, making it compatible with encryption Machines like the Lorenz Cipher. During encryption, plaintext is converted to ITA2 code, processed by the Machine, and transmitted securely.

A	11000	B	10011	C	01110	D	10010	E	10000	F	10110	G	01011
H	00101	I	01100	J	11010	K	11110	L	01001	M	00111	N	00110
O	00011	P	01101	Q	11101	R	01010	S	10100	T	00001	U	11100
V	01111	W	11001	X	10111	Y	10101	Z	10001				

3	00010	4	01000	8	11111	9	00100	+	11011	/	00000

Figure 1. Lorenz Cipher Machine Logical Layout. From the Center for Innovations in Mathematical Teaching. (n.d.).

2. **XOR Logic (Modulo-2 Arithmetic)**

   XOR logic (denoted as $\oplus$), also known as Modulo-2 arithmetic, is a reversible binary operation crucial to the encryption process. It flips binary values based on the rule: a bit changes to 1 only if the input bits are different. This allows messages to be encrypted and decrypted using the same key. The logic works as follows:

   - $0 \oplus 0 = 0$
   - $0 \oplus 1 = 1$
   - $1 \oplus 0 = 1$
   - $1 \oplus 1 = 0$

3. **Twelve Rotating Wheels**

   As we can see in *Figure 2*, the Lorenz Machine had 12 wheels, categorized into Key Wheels (K), Motor Wheels (M), and Control Wheels (S). Each wheel contained adjustable cams that flipped binary values based on their position.

   - **K Wheels (Key Wheels)**:
     - Five wheels (K1 to K5) with 41, 31, 29, 26, and 23 cams, respectively.
     - Rotate after each letter, XORing the plaintext with the current key.
   - **M Wheels (Motor Wheels)**:
     - Two wheels with 61 and 37 cams, respectively.

- The M wheels control whether the S wheels rotate or not, adding randomness.

- **S Wheels (Control Wheels)**:
  - Five wheels (S1 to S5) with 43, 47, 51, 53, and 59 cams, respectively.
  - Apply an additional XOR layer to the encrypted output.

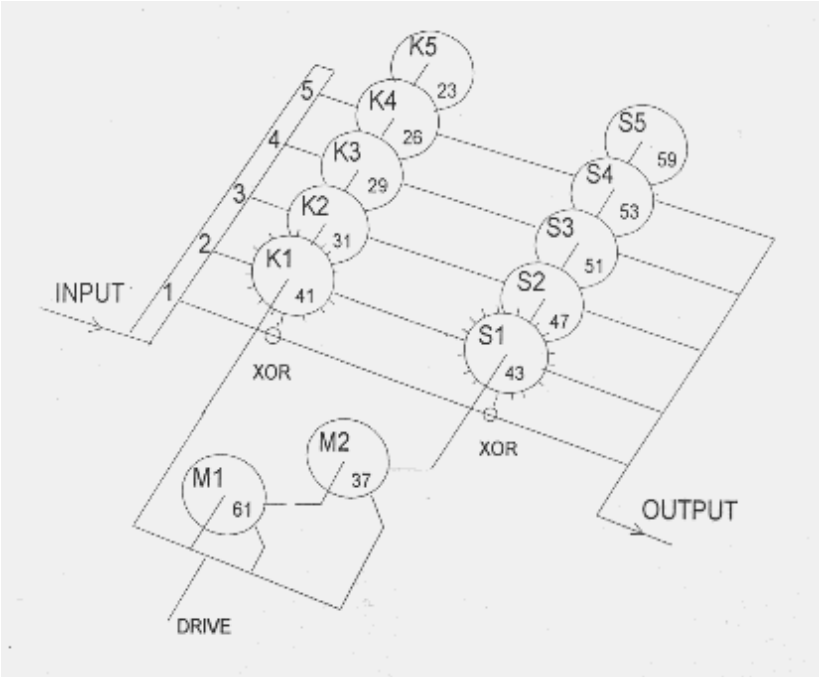


Figure 2. Lorenz Machine Logical Layout. From The National Museum of Computing, note by Charles Coultas (n.d.)

As shown in *Figure 3*, the S, M, and K wheels are displayed with their corresponding binary strings, based on the cam settings for each wheel. These strings consist of 1s and 0s, where a "+" symbol indicates a binary "1". These values will later be used in the C++ code to simulate the machine's behavior. The configuration in the figure represents an example of an initial setting, which determines the starting state of the machine before encryption begins. To

ensure the message can be properly decrypted, both the sending and receiving Lorenz machines must be configured with the exact same initial settings.

Figure 3 also shows additional details about the wheel structure. In the first column on the left, we can see the total number of cams on each wheel. The second column indicates how many of those cams were set to "+", meaning they are active and contribute a binary 1 during the encryption process. This setup helps visualize the internal configuration that affects how each wheel influences the final ciphertext.

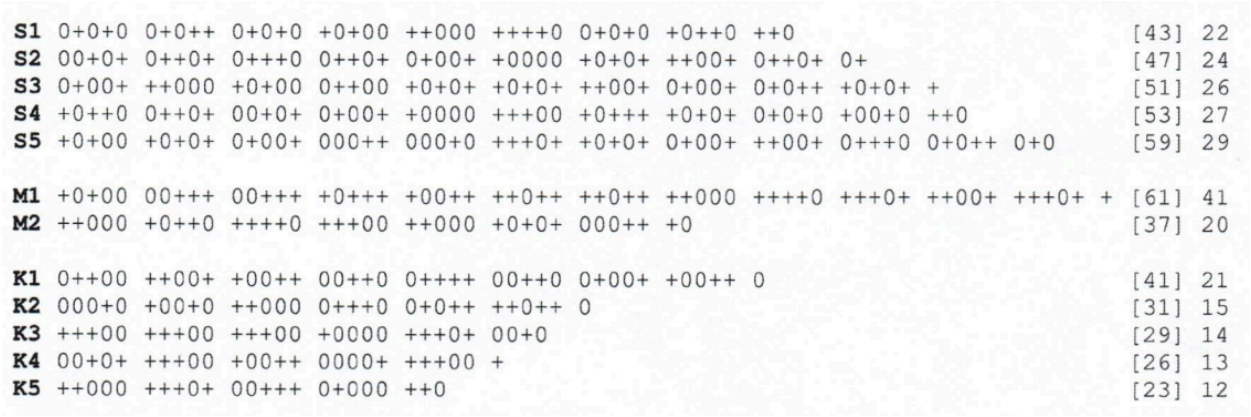


Figure 3. Lorenz Machine Logical Layout. From The National Museum of Computing, note by Charles Coultas (n.d.)

Encryption Process

With the components in place, let's dive into the step-by-step encryption process. This sequence ensured that each message was uniquely scrambled, making decryption impossible without the precise Machine settings.

1. **Encoding the Message**

   The plaintext message was converted into binary using the ITA2 system.

2. **Key Encryption (K Wheels)**

   Each letter's binary code was XORed with the current key generated by the K wheels bit by bit.

3. **Motor Wheel Control (M Wheels)**

   The M wheels dictated the movement of the S wheels, adding variability to the process:

   - M1 always advances by one step after each letter.
   - M2 moves only if the cam value of M1 is 1.
   - If M2 moves and the cam value is 1, the S wheels advance. Otherwise, they remain in their place.

4. **Additional Encryption (S Wheels)**

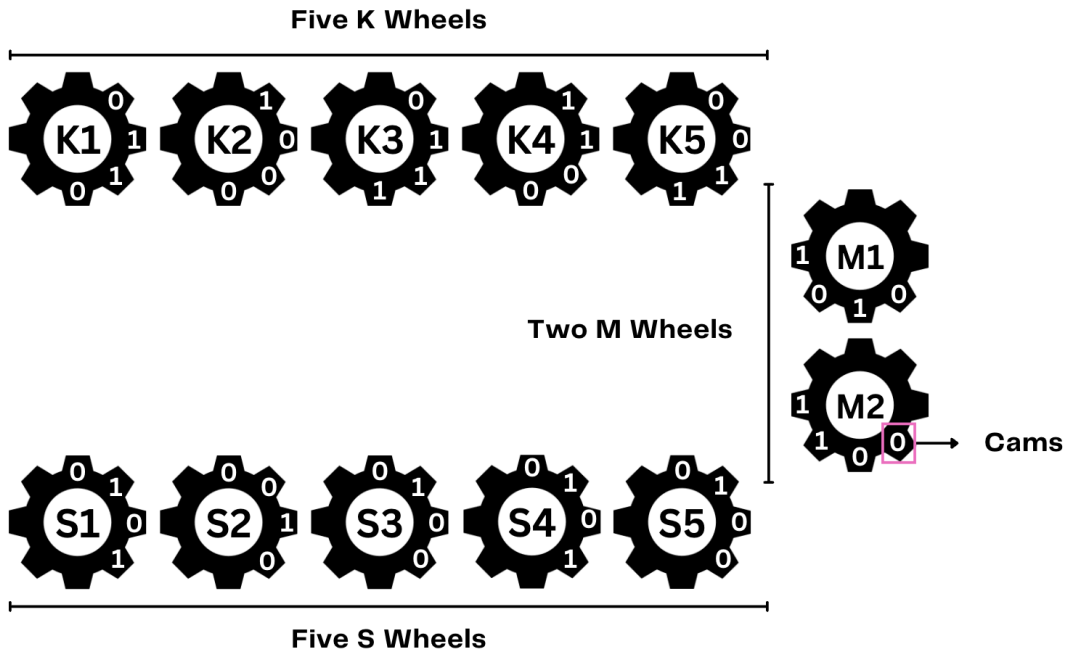
   - The S wheels only advance whenever the cam value of M2 is 1. When the S wheels advanced, they XORed their keys with the output from the K wheels bit by bit, adding another encryption layer.
   - If the S wheels do not advance, they continue using the same position until the M wheels trigger a change.

5. **Generating the Ciphertext**

   The final encrypted binary output was converted back into an ITA2 character, forming the ciphertext. This encrypted message was then ready for transmission.

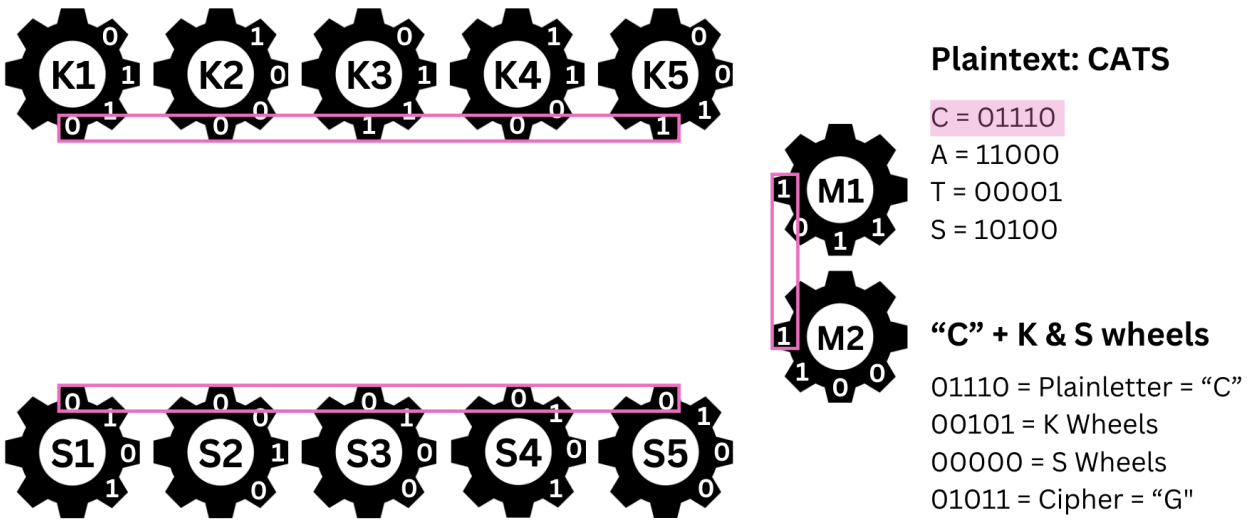
Now that we've broken down the components and explored the encryption process, let's examine how the Lorenz Cipher encrypted a message with a practical example. The image at the top of the next page illustrates the Lorenz Cipher Machine, showing the wheels (K, S, and M) in their initial starting positions. Each wheel consists of multiple cams, each represented by a binary

digit. A cam with a value of 1 indicates that it is active, while a cam with a value of 0 indicates that it is inactive.



For this demonstration, we'll encrypt the word "CATS" following the next steps:

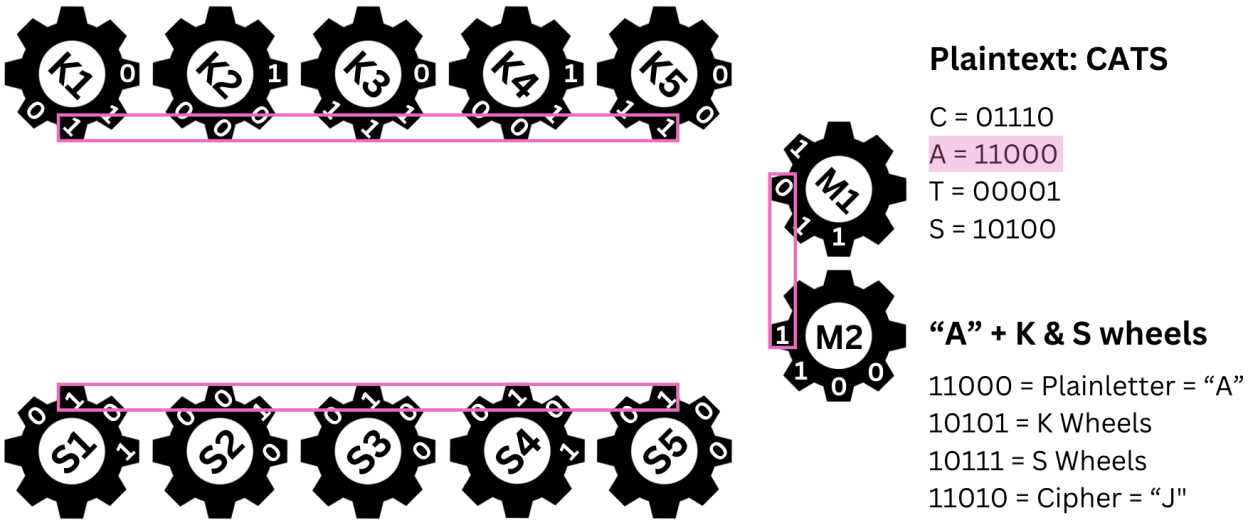
1. We determine the ITA2 binary code for each letter, as shown in the image.
2. To encrypt the message, apply the XOR operation by combining the plaintext binary, in this case corresponding to the letter C with the K wheels. As we learnt, this operation follows a simple rule: if two input bits are the same, the output is 0, and if the bits are different, the output is 1.
3. The resulting binary output is then further processed by combining it with the S wheels through another XOR operation.
4. Ultimately, this process outputs the cipher letter for the plaintext letter "C", which in this case is "G".



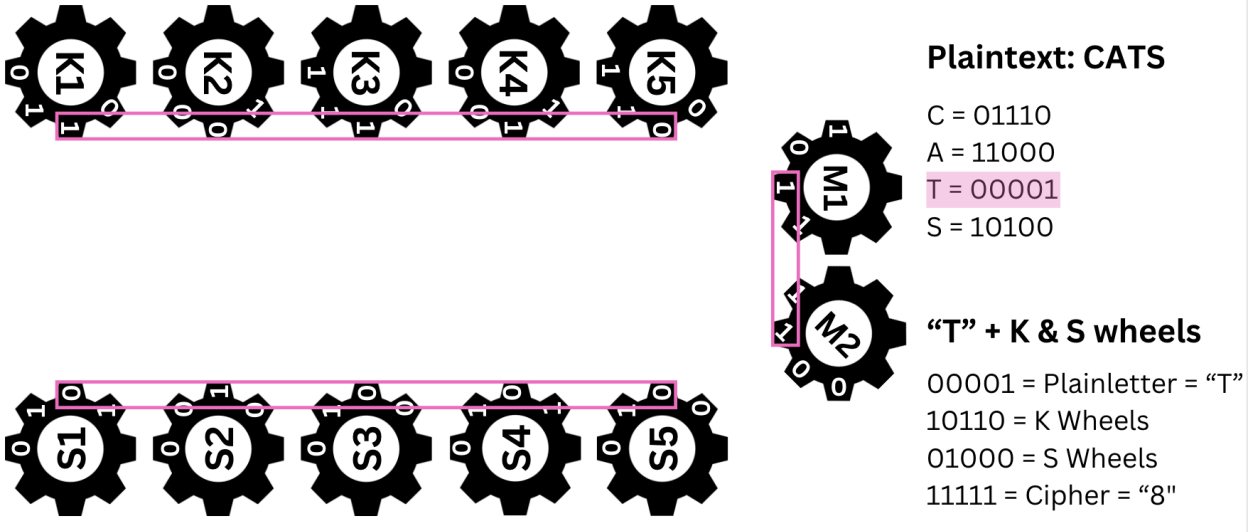
In the image above, we can see how encrypting the letter “C” (with its binary ITA2 code of 01110) results in the cipher letter “G”. It’s important to note that the M2 wheel begins with a cam value of 1 at its starting position. This configuration causes the S wheels to rotate before reaching the displayed starting position. Thus, the S wheels already rotated.

Continuing with the encryption process, we apply the same steps to the letter “A”. As explained earlier in the paper and as we can see in the image below, the K wheels and the M1 wheels rotated after each letter, advancing to the next cam positions. However, the M2 wheel remained stationary since the cam value of M1 was 0. The S wheels, on the other hand, advanced because the cam value of M2 was 1.

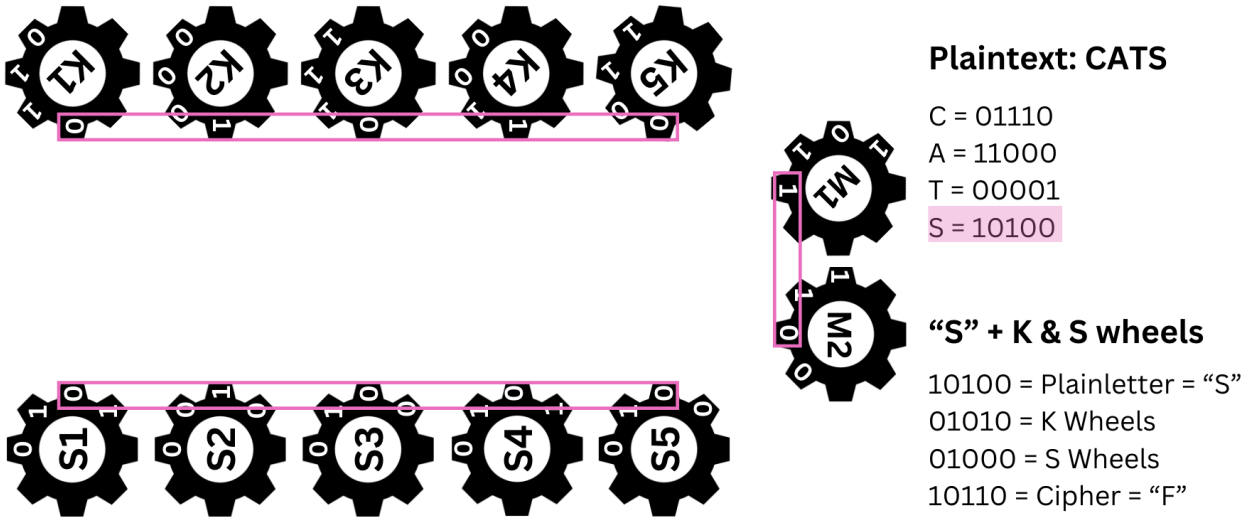
As we can see in the image at the top of the next page, in this step, the letter “A” (with its binary ITA2 code of 11000) was first XORed with the new K wheel values, resulting in an intermediate value. This intermediate value was then XORed with the S wheels' keys, ultimately producing the cipher letter “J”.



Continuing with the next letter, in the image below, we can observe that the K and M1 wheels have rotated. Additionally, the M2 wheel also advanced because the cam value of M1 was 1, and the S wheels rotated since the cam value of M2 was also 1. For the plaintext letter, “T”, its binary code (00001) was XORed with the K wheels' values. The result was then XORed with the S wheels' output, producing the cipher character, which in this case is the number “8”.



Finally, for the last letter, “S”, we observe that both the K and M1 wheels have rotated. The M2 wheel also advanced because the cam value at M1's current position was 1. The cam value of M2 is now 0, which prevented the S wheels from rotating. The binary code for the plaintext letter “S” (10100) was first XORed with the values from the K wheels. The result was then XORed with the S wheels' output, producing the cipher letter “F”. Below is an image illustrating this step:



Thus, by encrypting the word “CATS”, we obtained the final cipher text: “GJ8F”. By pressing this link[2], you can view a video demonstrating how the Lorenz Cipher’s wheels work, including their rotation and how they perform the encryption process in action.

This simple example illustrates the Lorenz Cipher’s complex encryption system, where rotating wheels and multiple XOR operations created a highly secure communication method. Each letter’s encryption depended on both the starting wheel positions and their rotations,

[2] https://drive.google.com/file/d/1lEYv6LtmEyCAeQqt8s7VHFW2wXZYGOsF/view

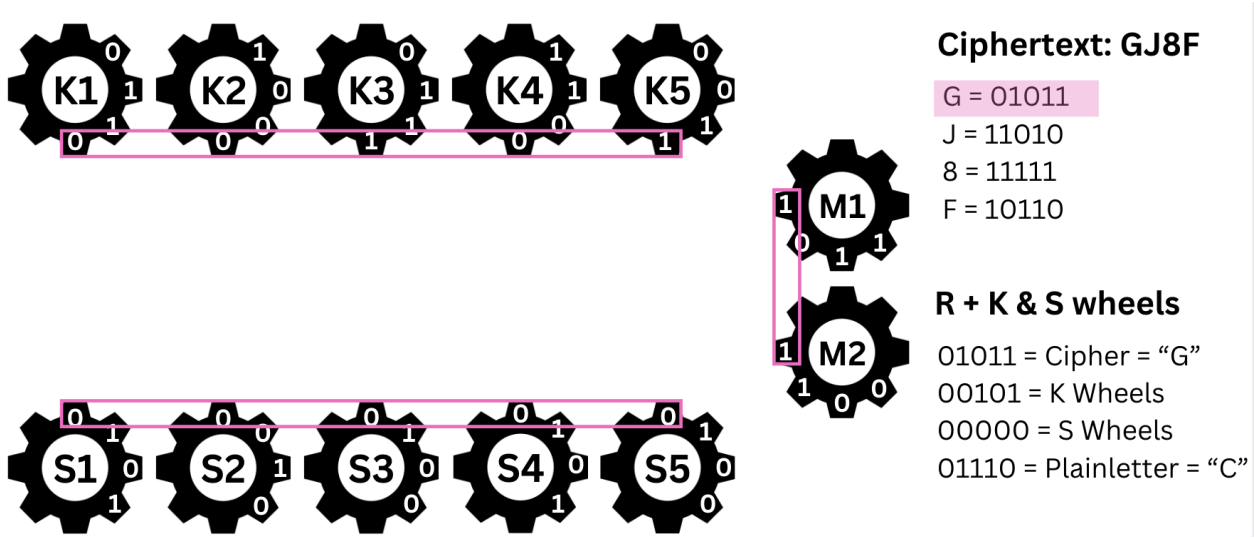
ensuring that even minor changes in alignment would produce entirely different results, making it an essential tool for secret wartime communications.

Decryption Process

Now that we have encrypted the plaintext “CATS” into the ciphertext “GJ8F”, let’s walk through the decryption process using the same initial wheel settings. Since the encryption process is reciprocal, decryption follows the exact same sequence as encryption—starting with the K wheels first, followed by the S wheels.

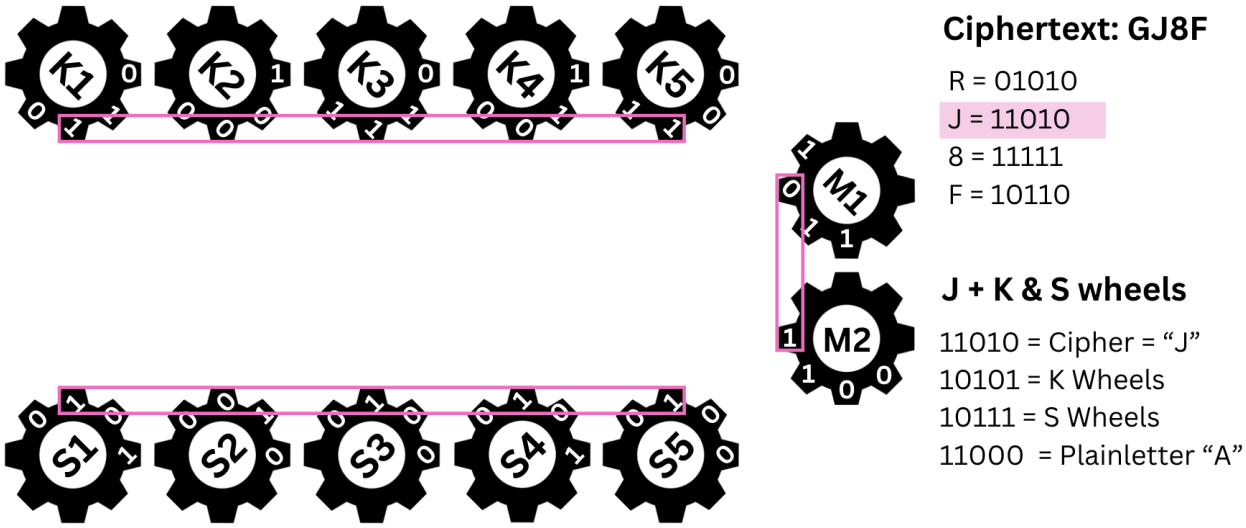
To begin the decryption process, we take the letter “G” from the ciphertext. First, we apply XOR with the K wheels, producing an intermediate binary value. Then, we apply XOR with the S wheels, which reveals the first letter of the original plaintext: “C”.

The image below illustrates this step, showing the initial wheel positions and how the XOR operations reverse the encryption.

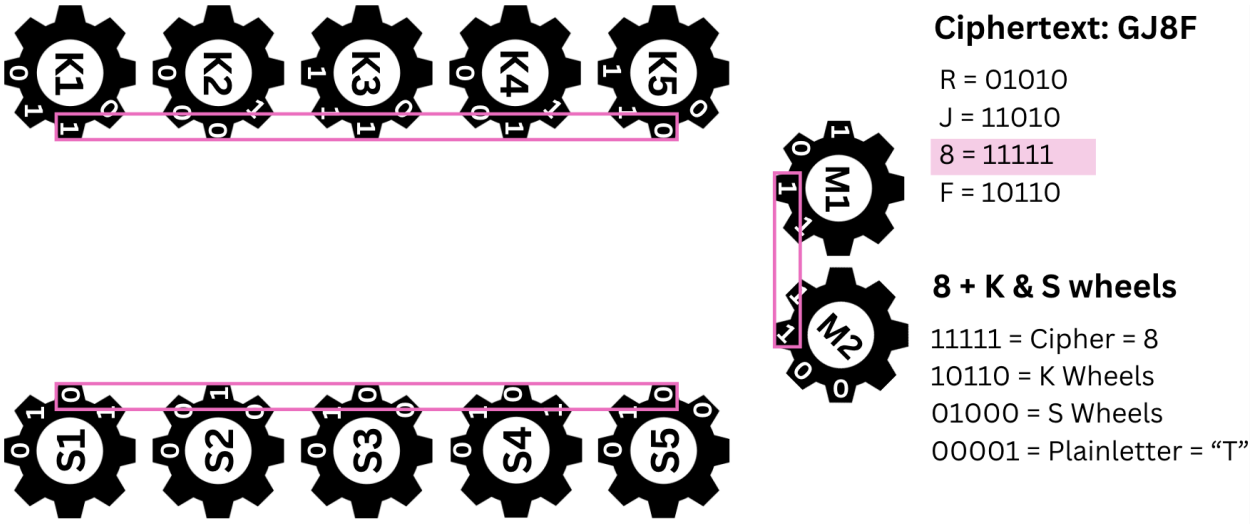


Next, we proceed with the ciphertext letter “J”, following the same decryption process. First, we apply XOR with the K wheels, producing an intermediate value. Then, we apply XOR with the S wheels, revealing the second letter of the original plaintext: “A”.

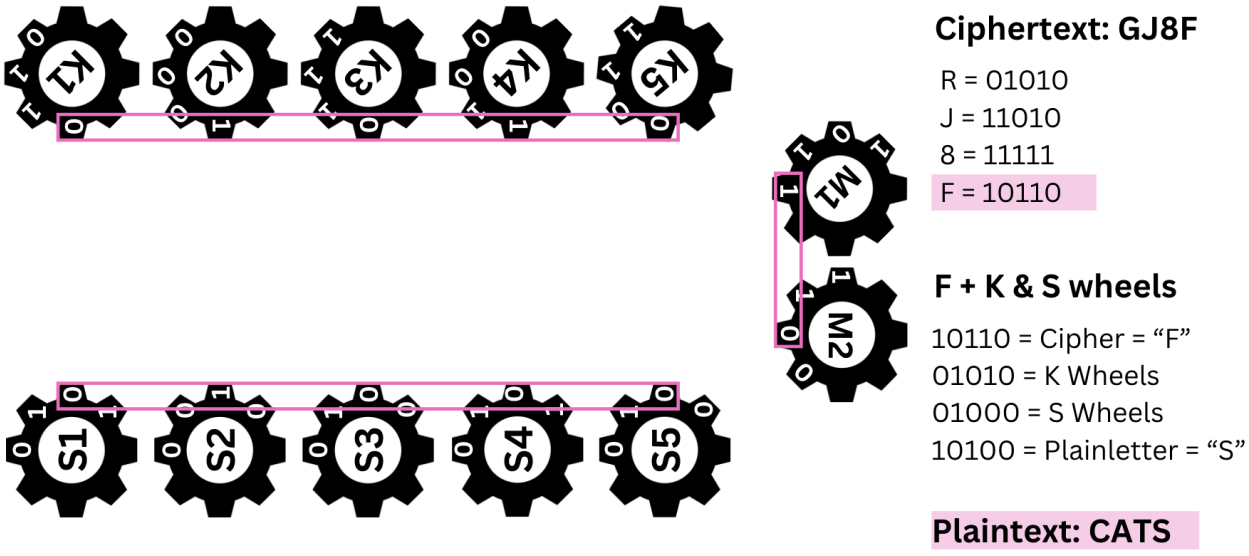
The image below illustrates this decryption step.



We repeat the same process with the next cipher character, “8”, applying XOR with the K wheels, followed by XOR with the S wheels, which reveals the next letter: “T”, as shown in the image at the top of the next page.



Finally, decrypting “F” using the same steps reveals the last letter: “S”. With this, the full plaintext is restored, “CATS”. The image below shows how this final step completes the decryption process.



By reversing the encryption step by step, we gradually reconstruct the original plaintext, just as it was before encryption. By pressing this link[3], you can view a video demonstrating how the Lorenz Cipher performs the decryption process in action.

Through this demonstration, we have seen both the encryption and decryption processes of the Lorenz Cipher in action. By starting with a plaintext message, applying multiple layers of XOR operations while advancing the cipher wheels, we transformed "CATS" into the seemingly unrecognizable ciphertext "GJ8F". Then, by carefully reversing the process using the same starting wheel positions, we uncovered the original message, letter by letter, proving the symmetric nature of the encryption system.

This example highlights the complexity and security of the Lorenz Cipher, which played a crucial role in secure wartime communications. The combination of rotating wheels, conditional movements, and XOR operations ensured that even small variations in the wheel positions would generate entirely different ciphertexts, making decryption without the correct settings nearly impossible. Understanding this process not only sheds light on historical cryptographic methods but also emphasizes the foundational principles of modern encryption techniques.

[3] https://drive.google.com/file/d/19hcqCA4SEgqVUCfPcKrgQL6JjCVtwK-C/view

5. Mathematical Strength of the Lorenz Cipher

The Lorenz Cipher's remarkable strength stems from its advanced use of binary operations, modular arithmetic, and its intricate design incorporating 12 code wheels. At its core, the cipher relies on the XOR operation, which is self-inverse—applying the same key twice (once during encryption and once during decryption) restores the original plaintext. This property ensures robust and efficient encryption.

The 12 wheels, each with a unique number of cams (pins or switches), create an immense key space. Notably, the cams are set to numbers that are co-prime (share no common factors other than 1), ensuring an exceptionally long period before the same key sequence repeats. This design significantly enhances security. Thus, the total number of key settings is the product of the cams on all 12 wheels:

$$41 \times 31 \times 29 \times 26 \times 23 \times 43 \times 47 \times 51 \times 53 \times 59 \times 37 \times 61 \approx 1.6 \times 10^{19}$$

This astronomical figure—approximately 16 quintillion—renders brute-force attacks computationally infeasible, even with modern computing power.

6. Programming

This section of the paper explores how the C++ program, developed in Visual Studio, simulates the encryption process, replicating the behavior of the Lorenz Cipher Machine. The program is structured into multiple components, each handling a specific aspect of encryption. Let's take a closer look.

1. Include Libraries

```C/C++
#include <iostream>

#include <string>

#include <vector>

#include <unordered_map>

#include <bitset>
```

These libraries are used for:

- `#include <iostream>` Allows the program to use input (cin) and output (cout) for user interaction.
- `#include <string>` Enables string manipulation, which is essential for handling the encoded messages.

- `#include <vector>` Provides dynamic arrays (std::vector), which we use for handling the encryption wheels.
- `#include <unordered_map>` Implements a hash table, which allows fast lookups for the ITA2 encoding table.
- `#include <bitset>` (Not actively used in this version, but typically helpful for working with binary representations.

2. Define the ITA2 Encoding Table

The program uses a data structure called an **unordered map** to store and organize information for quick access. In this case, it holds the ITA2 encoding, a system that, as we previously mentioned, assigns a unique 5-bit binary code to each uppercase letter and certain symbols.

Here is the part of the program that is responsible for converting each character in the input message to its corresponding binary code.

```
C/C++
// ITA2 encoding table

unordered_map<char, string> ita2 = {

    {'A', "11000"}, {'B', "10011"}, {'C', "01110"}, {'D',
"10010"}, {'E', "10000"}, {'F', "10110"}, {'G', "01011"},
```

```
    {'H', "00101"}, {'I', "01100"}, {'J', "11010"}, {'K',
"11110"}, {'L', "01001"}, {'M', "00111"}, {'N', "00110"},

...
```

3. Cam Organization

The cam organization for the K, M, and S wheels is hardcoded in the program, meaning the data is directly written into the code rather than being loaded from an external source. Each wheel is represented by a string of binary values.

For the K and S wheels, we use a `vector<string>`, which is a collection of multiple strings. This structure allows the program to store and organize several strings, with each string representing a separate wheel.

In contrast, for the M wheels, we use individual string variables (`string m1_wheel` and `string m2_wheel`). Since there are only two M wheels, using separate string variables simplifies their representation. This structure enables the program to efficiently manage and access both M wheels' data independently.

```C/C++
// Cam organization for K wheels

vector<string> k_wheels = {
```

```
    "01100110011001100110011111001100101100l",

    "00010100101100001110010111100110",

    "11100111001110010000111010010",

    "00101111001001100001111001",

    "1100011101001110100011O"

};



// Cam organization for M wheels

string m1_wheel =

"10100001110011110111100111101110001111011101100111101111";

string m2_wheel = "11000101101111011100110001010100011O";



// Cam organization for S wheels

vector<string> s_wheels = {
```

```
    "0101001011010101010001100011111001010110110",
    "0010101101011100110101001100001010111001011010",
    "0100111000101000110010101101011100101001010111101011",
    "1011001101001010100110000111001011110101010100101110",
    "1010010101010001100010111011010100101100101110010100"
};
```

4. Wheel Advancement

The `advance_wheel` function is responsible for simulating the mechanical rotation of the Lorenz Cipher Machine wheels by shifting the binary string by one position.

Here's the breakdown of the function:

```
C/C++
// Function to advance a wheel by one position
void advance_wheel(string &wheel) {
    char first = wheel[0]; // Step 1: Store the first character
of the string
```

```
    wheel = wheel.substr(1) + first; // Step 2: Remove the first
character and add it to the end

}
```

5. Function to Set Initial Wheel Positions

The `set_wheel_position(string &wheel, int position)` function is responsible for setting the initial position of a given encryption wheel based on user input. This function takes two parameters: a reference to the wheel string (`wheel`) and the user-defined starting position (`position`). It then shifts the wheel left `(position - 1)` times by repeatedly calling the `advance_wheel()` function. This effectively moves the first character of the string to the end, simulating a wheel rotation.

C/C++

```
void set_wheel_position(string &wheel, int position) {

    for (int i = 0; i < position - 1; ++i) {

        advance_wheel(wheel);

    }

}
```

6. XOR Operations

The program encrypts the message using XOR (exclusive OR) operations. The `xor_strings` function performs a bitwise XOR between the ITA2 binary code of the input message and binary keys derived from the K and S wheels. In this process, each corresponding pair of bits from the two strings is compared, and the result is either 0 or 1 according to the XOR rule.

```C/C++
// Function to perform XOR on two binary strings

string xor_strings(const string &a, const string &b) {

    string result;

    for (size_t i = 0; i < a.size(); ++i) {

        result += (a[i] == b[i]) ? '0' : '1';

    }

    return result;

}
```

7. Convert Encrypted Binary to Readable Characters

The function `binary_to_char(const string &binary)` is responsible for converting a single 5-bit binary sequence into its corresponding ITA2-encoded character. It does this by iterating through the predefined `ita2` encoding table, which maps each uppercase letter and certain symbols to a unique 5-bit binary value. When a match is found, the function returns the associated character; otherwise, it returns `'?'` as a fallback to indicate an invalid or unrecognized binary sequence.

```C/C++
char binary_to_char(const string &binary) {

    for (const auto &pair : ita2) {

        if (pair.second == binary) {

            return pair.first;

        }

    }

    return '?'; // Default if no match found

}
```

8. Message Encryption Process

The core encryption process is handled by the `encrypt_message` function. This function encrypts each character of the input message using a series of XOR operations and wheel advancements that simulate the behavior of the Lorenz Cipher Machine. Here's a step-by-step breakdown of how it works:

1. **Convert the character to ITA2 binary**: Each character from the input message is mapped to its corresponding ITA2 binary code.
2. **Generate a key from the K wheels**: The function constructs a 5-bit K key by taking the first bit from each of the five K wheels. After retrieving each bit, the corresponding wheel is advanced to simulate rotation.
3. **XOR the binary code with the K key**: The function XORs the ITA2 binary code with the K key.
4. **Advance the M wheels and determine if the S wheels should advance**:
    a. The M1 wheel is advanced.
    b. If the first bit of the M1 wheel is 1, the M2 wheel is also advanced.
    c. If the first bit of the M2 wheel is 1, all S wheels are advanced.
5. **Generate a key from the S wheels**: The function constructs a 5-bit S key by taking the first bit from each of the five S wheels.
6. **Apply a second XOR operation with the S key**: The function performs another XOR, this time between the result of the first XOR operation and the S key.

7. **Append the encrypted binary to the final output**: The encrypted binary string for the current character is added to the final encrypted message.
8. **Convert the encrypted binary message into a readable text format**: This section is responsible for processing the full encrypted binary message by breaking it into 5-bit chunks and converting each chunk into a character using the `binary_to_char()` function. The loop initializes an empty string called `readable_output`, which will store the final decrypted text. It then iterates over the `encrypted_message` in steps of five, ensuring that each segment corresponds to a valid ITA2 character.

   Within the loop, `substr(i, 5)` extracts a 5-bit binary sequence from the encrypted string. This sequence is passed to `binary_to_char()`, which converts it into a character. The resulting character is then appended to `readable_output`. This process continues until all binary chunks have been processed, ultimately forming a fully readable version of the encrypted message.

```C/C++
// Function to encrypt a message

string encrypt_message(const string &message) {

    string encrypted_message;

    for (char c : message) {
```

```
        // Convert character to ITA2 binary
        string ita2_code = ita2[c];


        // Generate key from K wheels
        string k_key;
        for (int i = 0; i < 5; ++i) {
            k_key += k_wheels[i][0];
            advance_wheel(k_wheels[i]);
        }


        // XOR plaintext with K key
        string xor1 = xor_strings(ita2_code, k_key);


        // Advance M wheels and possibly S wheels
```

```
        advance_wheel(m1_wheel);
        if (m1_wheel[0] == '1') {
            advance_wheel(m2_wheel);
            if (m2_wheel[0] == '1') {
                for (int i = 0; i < 5; ++i) {
                    advance_wheel(s_wheels[i]);
                }
            }
        }


        // Generate key from S wheels
        string s_key;
        for (int i = 0; i < 5; ++i) {
            s_key += s_wheels[i][0];
```

```
        }


        // XOR with S key
        string final_xor = xor_strings(xor1, s_key);


        // Append the encrypted character
        encrypted_message += final_xor;
    }


        // Convert the encrypted binary message into a readable
text format
    string readable_output;
    for (size_t i = 0; i < encrypted_message.size(); i += 5) {
        string binary_chunk = encrypted_message.substr(i, 5);
```

```
        readable_output += binary_to_char(binary_chunk);
    }
    return readable_output;
}
```

9. Initialize Wheels with User Input

The `initialize_wheels()` function is responsible for allowing the user to set the starting positions of the encryption wheels before any messages are encrypted.

This function begins by defining a vector of integers (`max_positions`), which holds the maximum allowable positions for each wheel. Each wheel has a different length, so the user must select a position within its valid range. Once a valid starting position is chosen, the function calls `set_wheel_position(*wheels[i], position)`, which rotates the wheel to the specified position.

C/C++

```
// Function to get user-defined wheel positions

void initialize_wheels() {
```

```
    vector<int> max_positions = {41, 31, 29, 26, 23, 61, 37, 43,
47, 51, 53, 59};

    vector<string*> wheels = {&k_wheels[0], &k_wheels[1],
&k_wheels[2], &k_wheels[3], &k_wheels[4], &m1_wheel, &m2_wheel,

                              &s_wheels[0], &s_wheels[1],
&s_wheels[2], &s_wheels[3], &s_wheels[4]};

    vector<string> wheel_names = {"k1", "k2", "k3", "k4", "k5",
"m1", "m2", "s1", "s2", "s3", "s4", "s5"};



    for (size_t i = 0; i < wheels.size(); ++i) {

        int position;

        do {

            cout << \"Choose a number from 1 to \" <<
max_positions[i] << \" for \" << wheel_names[i] << \": \";

            cin >> position;

        } while (position < 1 || position > max_positions[i]);
```

```
        set_wheel_position(*wheels[i], position);
    }
    cin.ignore();
}
```

10. Main Function

The `main` function handles the user interaction for encrypting messages. It allows users to enter multiple messages in a single session and displays both the encrypted binary output and a readable version. The program runs in a loop until the user types “EXIT” to terminate. Here's a step-by-step breakdown of how the function works:

```
C/C++
int main() {
    initialize_wheels();
    while (true) {
        string message;
```

```
        cout << "Enter message to encrypt: "; // The program
prompts the user to enter a message to encrypt.

        getline(cin, message);  // The program reads the full
input line using getline().

        if (message == "EXIT") break; // If the user enters
"EXIT", the program breaks out of the loop and ends.

        }

        string encrypted_message = encrypt_message(message);

        cout << "Encrypted message: " << encrypted_message <<
endl;

    }

    return 0;

}
```

To demonstrate the program in action, *Figure 4* shows a screenshot of the interface where the starting wheel positions are manually selected to encrypt the word “CATS.” Once the encryption process is run, the program outputs the ciphertext “4WO4.”

```
Choose a number from 1 to 41 for starting position of wheel k1: 1
Choose a number from 1 to 31 for starting position of wheel k2: 1
Choose a number from 1 to 29 for starting position of wheel k3: 1
Choose a number from 1 to 26 for starting position of wheel k4: 1
Choose a number from 1 to 23 for starting position of wheel k5: 1
Choose a number from 1 to 61 for starting position of wheel m1: 1
Choose a number from 1 to 37 for starting position of wheel m2: 1
Choose a number from 1 to 43 for starting position of wheel s1: 1
Choose a number from 1 to 47 for starting position of wheel s2: 1
Choose a number from 1 to 51 for starting position of wheel s3: 1
Choose a number from 1 to 53 for starting position of wheel s4: 1
Choose a number from 1 to 59 for starting position of wheel s5: 1
Enter message to encrypt (uppercase letters only, or type 'EXIT' to quit): CATS
Encrypted message: 4WO4
```

Figure 4. Encryption of the plaintext "CATS" using manually selected starting wheel positions in the C++ program.

Then, as shown in *Figure 5*, the same starting wheel positions are used to decrypt "4WO4," successfully recovering the original message letter by letter. This demonstration emphasizes the symmetric nature of the Lorenz Cipher: both encryption and decryption require identical initial settings to work correctly.

```
Choose a number from 1 to 41 for starting position of wheel k1: 1
Choose a number from 1 to 31 for starting position of wheel k2: 1
Choose a number from 1 to 29 for starting position of wheel k3: 1
Choose a number from 1 to 26 for starting position of wheel k4: 1
Choose a number from 1 to 23 for starting position of wheel k5: 1
Choose a number from 1 to 61 for starting position of wheel m1: 1
Choose a number from 1 to 37 for starting position of wheel m2: 1
Choose a number from 1 to 43 for starting position of wheel s1: 1
Choose a number from 1 to 47 for starting position of wheel s2: 1
Choose a number from 1 to 51 for starting position of wheel s3: 1
Choose a number from 1 to 53 for starting position of wheel s4: 1
Choose a number from 1 to 59 for starting position of wheel s5: 1
Enter message to encrypt (uppercase letters only, or type 'EXIT' to quit): 4WO4
Decrypted message: CATS
```

Figure 5. Decryption of the ciphertext "4WO4" using the same starting wheel positions to recover the original message.

To conclude, this C++ program successfully simulates the encryption process of the Lorenz Cipher Machine. By implementing ITA2 encoding, key generation through rotating

wheels, and XOR-based encryption, the program provides an accurate representation of how messages were encrypted during World War II. The use of functions to handle wheel advancement, XOR operations, and message processing ensures maintainability and clarity in the code structure.

7. Conclusion

This paper delves into the history and intricate workings of the Lorenz Cipher, highlighting its significance and complexity. To further explore and analyze its functionality, programs were initially developed using the C++ programming language, demonstrating how modern computational tools can be applied to decode historical cryptographic systems. This approach not only provides insight into Lorenz's mechanisms but also showcases the enduring relevance of cryptography in understanding both historical and contemporary encryption methods.

8. References

Center for Innovations in Mathematical Teaching. (n.d.). *Lorenz cipher machine*. Retrieved from https://www.cimt.org.uk/resources/topical/lorenz/codes_u19_text.pdf

Gillow, M. (n.d.). *The Lorenz Machine*. Virtual Lorenz. Retrieved from https://www.lorenz.virtualcolossus.co.uk/lorenz3d.html

Lewand, R. E. (2000). *Cryptological mathematics*. The Mathematical Association of America.

National Museum of Computing. (n.d.). *The Lorenz cipher machine: How it was broken at Bletchley Park*. Retrieved from https://static1.squarespace.com/static/5bf28ad6b98a7888bf3cdce5/t/6008b775401a704571ceca02/1611183992836/breaking+lorenz+iii.pdf

National Security Agency. (n.d.). *German cipher machines of World War II*. Center for Cryptologic History. Retrieved from https://www.nsa.gov/portals/75/documents/about/cryptologic-heritage/historical-figures-publications/publications/wwii/german_cipher.pdf

Singh, S. (1999). *The code book: The science of secrecy from ancient Egypt to quantum cryptography*. Anchor Books.

singingbanana. (n.d.). *Lorenz: Hitler's "Unbreakable" Cipher Machine*. YouTube. Retrieved from https://www.youtube.com/watch?v=GBsfWSQVtYA&t=574s

Virtual Colossus. (n.d.). *The Lorenz machine*. Retrieved from https://lorenz.virtualcolossus.co.uk/lorenz3d.html

9. Appendices

Here is the complete code:

```C/C++
#include <iostream>

#include <string>

#include <vector>

#include <unordered_map>

#include <bitset>



using namespace std;



// ITA2 encoding table

unordered_map<char, string> ita2 = {

    {'A', "11000"}, {'B', "10011"}, {'C', "01110"}, {'D',

"10010"}, {'E', "10000"}, {'F', "10110"}, {'G', "01011"},
```

```
    {'H', "00101"}, {'I', "01100"}, {'J', "11010"}, {'K',
"11110"}, {'L', "01001"}, {'M', "00111"}, {'N', "00110"},

    {'O', "00011"}, {'P', "01101"}, {'Q', "11101"}, {'R',
"01010"}, {'S', "10100"}, {'T', "00001"}, {'U', "11100"},

    {'V', "01111"}, {'W', "11001"}, {'X', "10111"}, {'Y',
"10101"}, {'Z', "10001"},

    {'3', "00010"}, {'4', "01000"}, {'8', "11111"}, {'9',
"00100"}, {'+', "11011"}, {'/', "00000"}

};



// Cam organization for K wheels

vector<string> k_wheels = {

    "011001100110011001100111100110010110011",

    "000101001011000011100101111001100",

    "111001110011100100001110100101",
```

```
    "00101111001001100001111001",
    "1100011101001110100011O"
};


// Cam organization for M wheels
string m1_wheel =
"10100001110011110111100111101101110001111011101100111101111";
string m2_wheel = "11000101101111011100110001010100011O";


// Cam organization for S wheels
vector<string> s_wheels = {
    "01010010110101010100011000111100101010110110",
    "00101011010111001101010011000010101110010110101",
    "01001110001010001100101011010111001010010101110101l",
```

```
    "10110011010010101001100001110010111101010101010010110",
    "10100101010100011000101110110101001011001011100010100"
};


// Function to advance a wheel by one position
void advance_wheel(string &wheel) {
    char first = wheel[0];
    wheel = wheel.substr(1) + first;
}


// Function to set wheel starting position
void set_wheel_position(string &wheel, int position) {
    for (int i = 0; i < position - 1; ++i) {
        advance_wheel(wheel);
```

```
    }
}


// Function to perform XOR on two binary strings
string xor_strings(const string &a, const string &b) {
    string result;
    for (size_t i = 0; i < a.size(); ++i) {
        result += (a[i] == b[i]) ? '0' : '1';
    }
    return result;
}


// Function to convert binary to readable character
char binary_to_char(const string &binary) {
```

```
    for (const auto &pair : ita2) {
        if (pair.second == binary) {
            return pair.first;
        }
    }
    return '?';
}


// Function to encrypt a message
string encrypt_message(const string &message) {
    string encrypted_message;
    for (char c : message) {
        // Convert character to ITA2 binary
        string ita2_code = ita2[c];
```

```
// Generate key from K wheels
string k_key;
for (int i = 0; i < 5; ++i) {
    k_key += k_wheels[i][0];
    advance_wheel(k_wheels[i]);
}

// XOR plaintext with K key
string xor1 = xor_strings(ita2_code, k_key);

// Advance M wheels and possibly S wheels
advance_wheel(m1_wheel);
if (m1_wheel[0] == '1') {
```

```
        advance_wheel(m2_wheel);
        if (m2_wheel[0] == '1') {
            for (int i = 0; i < 5; ++i) {
                advance_wheel(s_wheels[i]);
            }
        }
    }


    // Generate key from S wheels
    string s_key;
    for (int i = 0; i < 5; ++i) {
        s_key += s_wheels[i][0];
    }
```

```
        // XOR with S key
        string final_xor = xor_strings(xor1, s_key);

        // Append the encrypted character
        encrypted_message += final_xor;
    }

    string readable_output;
    for (size_t i = 0; i < encrypted_message.size(); i += 5) {
        string binary_chunk = encrypted_message.substr(i, 5);
        readable_output += binary_to_char(binary_chunk);
    }
    return readable_output;
}
```

```
// Function to get user-defined wheel positions
void initialize_wheels() {
    vector<int> max_positions = {41, 31, 29, 26, 23, 61, 37, 43,
47, 51, 53, 59};
    vector<string*> wheels = {&k_wheels[0], &k_wheels[1],
&k_wheels[2], &k_wheels[3], &k_wheels[4], &m1_wheel, &m2_wheel,
                              &s_wheels[0], &s_wheels[1],
&s_wheels[2], &s_wheels[3], &s_wheels[4]};
    vector<string> wheel_names = {"k1", "k2", "k3", "k4", "k5",
"m1", "m2", "s1", "s2", "s3", "s4", "s5"};

    for (size_t i = 0; i < wheels.size(); ++i) {
        int position;
        do {
```

```
            cout << "Choose a number from 1 to " <<
max_positions[i] << " for starting position of wheel " <<
wheel_names[i] << ": ";

            cin >> position;

        } while (position < 1 || position > max_positions[i]);

        set_wheel_position(*wheels[i], position);

    }

    cin.ignore();

}



int main() {

    initialize_wheels();

    while (true) {

        string message;
```

```
        cout << "Enter message to encrypt (uppercase letters
only, or type 'EXIT' to quit): ";
        getline(cin, message);
        if (message == "EXIT") {
            break;
        }
        string encrypted_message = encrypt_message(message);
        cout << "Encrypted message: " << encrypted_message <<
endl;
    }
    return 0;
}
```