CSE350: Network Security Assignment-1

Akash Kushwaha - 2021514

Suyash Kumar - 2021293

Project 2: Transposition Cipher

A **translational algorithm** in cryptography uses a matrix-based approach where text is arranged in rows and columns, and the order of extraction is determined by a predefined key. The key dictates the sequence in which columns are selected, effectively scrambling the original message.

Decryption requires the exact key to reconstruct the original message.

AIM

We were required to develop executable programs to encrypt, decrypt with a key, and launch a brute-force attack to discover the key given a cipher text. The encryption and decryption of the plaintext and ciphertext must be done using a Transposition Algorithm.

Hash Function

A hash function is a cryptographic tool that converts an input message into a fixed-size string of bytes, called a hash or digest. This output is unique for different inputs and is used primarily for data integrity, authentication, and digital signatures.

MD5 Hash Function Used

MD5 (Message-Digest Algorithm 5) is a widely-used hash function producing a 128-bit hash value. Although it is known to have vulnerabilities and is

susceptible to hash collisions, it is still used in various non-security applications and as an introductory algorithm in educational settings.

- **Function:** `calculate_md5`
- **Purpose:** To generate a fixed-length, 32-character hexadecimal hash from any given input string.
- **Input:** `input_string` - the string to hash.
- **Output:** A 32-character hexadecimal string representing the MD5 hash.
  We are using
  `hashlib` library to implement this hash function.
  Each hex digit represents 4 bits, and thus 128 bits create a 32-digit hexadecimal number.

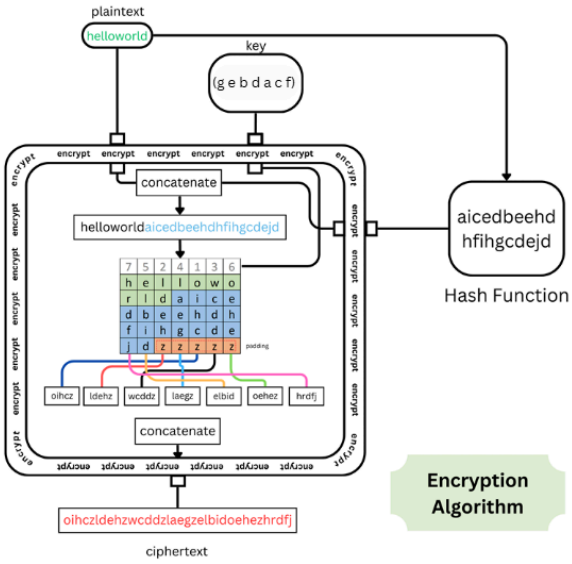
Plaintext

Function: `make_plain_text`

- **Purpose:** Creates a concatenated string consisting of the original text followed by its MD5 hash.
- **Input:** `original_text` - the text to be hashed and concatenated.
- **Output:** A new string that combines the original text with its 32-character MD5 hash.

This method is essential for ensuring the integrity of the plaintext in cryptographic processes, allowing for easy verification post-decryption.

Encryption



Function: `encrypt_transposition`

- **Purpose:** Encrypts plaintext using a columnar transposition method dictated by a key.
- **Input:**
  - `plaintext` - the text to be encrypted.
  - `key` - a string used to determine the order of transposition.
- **Output:** The ciphertext, where characters of the plaintext are rearranged according to the key.

How It Works:

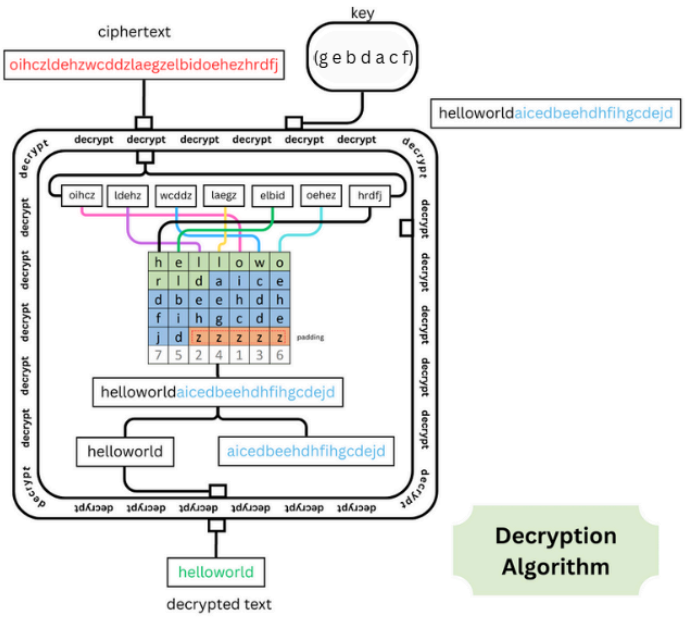
1. **Key Length and Mapping:**
   - Calculate the length of the key (`n`).
   - Create a `key_map` which maps each character in the key to its position in a sorted version of the key. This determines the new order of columns in the encryption grid.
2. **Padding:**
   - If the plaintext length isn't a multiple of `n`, pad it with `$` to make it fit perfectly into rows of length `n`.
3. **Creating Rows:**
   - Divide the padded plaintext into rows, each of length `n`.
4. **Columnar Rearrangement:**

- Construct the ciphertext by collecting characters column by column based on the order defined in the `key_map`.
- For each column index from 0 to `n-1`, find its actual position in the key and append the corresponding character from each row to the ciphertext.

5. **Concatenation:**
   - Combine the collected characters to form the final ciphertext string.

This encryption method scrambles the plaintext in a way that can only be reversed if the key is known, ensuring that the encrypted message is secure against unauthorized decryption without the key.

Decryption



Transposition decryption is the reverse process of transposition encryption. It rearranges the characters back to their original positions in the plaintext, using the same key that was used for encryption. This ensures that the original message can be accurately reconstructed from the ciphertext.

Function: `decrypt_transposition`

- **Purpose:** Decrypts ciphertext that was encrypted using a columnar transposition method.
- **Input:**
  - `ciphertext` - the text to be decrypted.
  - `key` - the string used during the encryption process.
- **Output:** The decrypted plaintext, reconstructed to its original form.

How It Works:

1. **Key Length and Mapping:**
   - Determine the length of the key (`n`).
   - Create a `key_map` that identifies the position of each character in the key relative to its sorted order, similar to the encryption process.
2. **Column Setup:**
   - Calculate the number of rows in the transposition grid, which is the total length of the ciphertext divided by `n`.
   - Create an array to store the columns, which will initially be empty strings.
3. **Distribute Ciphertext into Columns:**
   - Populate the columns with characters from the ciphertext based on the original column positions determined by the `key_map`.
   - Characters are extracted sequentially from the ciphertext and filled into their respective columns as per the positions outlined in the `key_map`.
4. **Reconstruct Rows:**
   - Build the plaintext by reading the characters row-wise across the columns, effectively reversing the columnar rearrangement process done during encryption.
5. **Remove Padding:**
   - Strip any padding (`$`) that was added during the encryption process to align the plaintext with the key length.

Brute Force Attack

A brute force attack is a straightforward strategy to crack encrypted data. It involves systematically checking all possible keys until the correct one is found. This method relies on trial and error and is generally considered as the simplest, yet often the most time-consuming, way to overcome cryptographic security measures.

The brute force attack is designed to discover the correct key that can decrypt a set of ciphertexts encrypted using a transposition cipher.

Function: `brute_force_attack`

- **Purpose:** To find the key that successfully decrypts given ciphertexts to their correct plaintext forms that satisfy the predefined property π.
- **Input:**
  - `ciphertexts` - a list of encrypted texts.
  - `alphabet` - a sorted sequence of characters used in the key.
  - `max_key_length` - the maximum permissible length of the key, based on computational feasibility.
- **Output:** The key that correctly decrypts all the provided ciphertexts or None if no suitable key is found.

How It Works:

1. **Generate Potential Keys:**
   - Use permutations of the given alphabet up to the `max_key_length` to generate all possible key combinations.
   - Iterate through each generated key, testing it against all the ciphertexts.
2. **Key Validation:**
   - For each key, decrypt each ciphertext and verify the result using the property π (e.g., checking if the decrypted text concatenated with its hash matches the original format).
   - The key validation function `check_key_with_ciphertext` performs the decryption and checks the property π.
3. **Determine the Correct Key:**

- If a key successfully decrypts a ciphertext and the resultant plaintext satisfies the property π for all given ciphertexts, this key is considered correct.
- If a key fails for any ciphertext, it is discarded, and the next key permutation is tested.

4. **Return Result:**
   - If a valid key is discovered, return this key. Otherwise, return `None` indicating that no valid key could be found within the tested range.

Key

A key in cryptography is a piece of information that determines the functional output of a cryptographic algorithm. For encryption and decryption, the key is used to scramble and unscramble data, respectively.

Key we have used:

The key is a sequence of unique letters used to decide the transposition in the encryption and decryption processes.

- While keys often consist of numbers, they can also be made up of letters or symbols as we have used.
- If the maximum key length is 9, similar to numeric keys ranging from 1 to 9, our keys are permutations of any 9 unique letters in lowercase from the alphabets.
- And during the brute force attack we provide the sorted key, so that the unique characters are known.
- During decryption, the same key used in encryption is necessary to accurately reorder the transposed characters and retrieve the original plaintext.

Usage and Code Flow Explanation

Step-by-Step Usage:

1. **Run the Script:** Initially, press the 'Run' button to execute the script. No manual input is needed for the first part of the execution, as it uses a predefined key.
2. **Initial Encryption and Decryption:**
    - **Encryption:** The script encrypts five predefined plaintexts using a hard-coded key. Each plaintext is first appended with its MD5 hash to form a new plaintext which is then encrypted.
    - **Decryption:** Each ciphertext is decrypted using the same key.
    - **Verification:** The script checks if the decrypted text matches the original plaintext concatenated with its hash (property π).
3. **Brute Force Attack:**
    - **Input Key:** After initial encryption and decryption, you will be prompted to input a key. This key is used to re-encrypt the same five texts.
    - **Brute Force Execution:** The script then attempts to discover this key by trying all possible permutations of a given set of characters up to the length of 9. It uses these permutations to decrypt the newly encrypted texts and checks if the decrypted texts satisfy the property π.
    - **Key Discovery:** If a permutation successfully decrypts all texts correctly, it is identified as the correct key.

Code Flow:

- `test_encrypt_decrypt` **Function:** Handles the initial encryption and decryption using a predetermined key. It prints the original text, the modified plaintext, the ciphertext, and the results of the decryption along with the verification status for each of the five texts.
- `run_brute_force` **Function:** Takes a user input for a new key, re-encrypts the texts, and launches a brute force attack to find this key by checking against all possible key permutations. The results of the brute force, including any discovered key and the decrypted texts, are printed.

Samples

```
Original text:  helloalgorithm
Plain_text:  helloalgorithmbf166fa043b9fd1b4b3ed9271db67922
Ciphertext:  ot693d$lr13472ahffeb$li6bb12eof4b29lmadd6$hgb0197
Decryptedtext:  helloalgorithmbf166fa043b9fd1b4b3ed9271db67922
Matches original plaintext?:  True

Original text:  securetransmission
Plain_text:  securetransmission23b2023154ac651c86b42282c0e2605d
Ciphertext:  rm21126$cno264e$ei35c80$usn3522$eai0cb0$tsb4825$srs2a6cd
Decryptedtext:  securetransmission23b2023154ac651c86b42282c0e2605d
Matches original plaintext?:  True

Original text:  plaintextmessage
Plain_text:  plaintextmessage5d46f828bd89a39deb827ea4e31338c8
Ciphertext:  ns4de4cam589e3ts68be8iedbda8lte2373eaf983$pxg8a21
Decryptedtext:  plaintextmessage5d46f828bd89a39deb827ea4e31338c8
Matches original plaintext?:  True

Original text:  cryptographyrules
Plain_text:  cryptographyrulesf9be7d1af5bb9e35908497a2abd96b2d
Ciphertext:  ty9f5abyps1e99orb5922phfa376raed94dgueb0adcrl7b8b
Decryptedtext:  cryptographyrulesf9be7d1af5bb9e35908497a2abd96b2d
Matches original plaintext?:  True

Original text:  federatedlearning
...
Ciphertext:  ra95b09dlg32ccarf0c8feee985eedn0818tn97d15fei53d3
Decryptedtext:  federatedlearninge9f950395073828bcdd1c508138ce9f5
Matches original plaintext?:  True
```

Computational Considerations:

- **Efficiency:** Brute force attacks are computationally expensive, especially as the key length increases. The number of potential keys grows exponentially with the key length, leading to longer decryption times.
- **Feasibility:** Practicality of brute force attacks depends on the key size and available computational power. Smaller key sizes or stronger computational resources can make brute force feasible.
- **Security Implications:** The feasibility of brute force attacks highlights the importance of using sufficiently long and complex keys in cryptographic systems to prevent unauthorized decryption.