|  |  |
| --- | --- |
| Child hiding in bushes  Encryption project  CMPG 215P – Group 40 | Documentation  Covering chilean’s tracks  Group 40  Anderson, Brian– 40851842  Bezuidenhout, Nicolette – 40513262  Coetzee, Christian – 40513262  Patel, Riya – 41914228  Ramsunar, Angelina - 41081269 |

**Table of Contents**

[**Why we chose to use blowfish as an encryption method:** 2](#_Toc135146060)

[**The Advantages and Disadvantages of Blowfish encryption method:** 3](#_Toc135146061)

[**Explanation of our own algorithm:** 4](#_Toc135146062)

[**Comparing own algorithm with others:** 11](#_Toc135146063)

[**References:** 12](#_Toc135146064)

## **Why we chose to use blowfish as an encryption method:**

Bruce Schneier created Blowfish, a symmetric-key block cipher method, in 1993. Blowfish works with fixed-size blocks of 64 bits and configurable key lengths ranging from 32 to 448 bits.

The Blowfish algorithm is basic and easy to grasp, making it accessible to developers and academics. Its ease of use contributes to its speed and efficiency.

Since Blowfish is not hampered by patents or license restrictions, it can be freely implemented and used. This makes it an appealing solution for open-source projects and applications that may raise intellectual property concerns.

As the amount of cyber-attacks rises, so should the adoption of encryption algorithms. It is critical to safeguard our smartphones, which include personal information, contact lists, and even movement logs. Our autos, which record our driving habits, require protection as well. In our case the protection of the Rottweilers, using Blowfish will ensure their safety as the Chihuahua assassin will be on the hunt for them.

Blowfish encryption techniques aid in the confidentiality of private data. They encrypt data transmission to protect contact lists and personal emails from being read by unauthorized parties, keep firmware upgrades out of devices you don't want to update, and verify that information senders are who they claim to be. We chose using blowfish as it will help us in the most effective manner to protect the Rottweilers.

Blowfish is a cryptographic algorithm that has been thoroughly researched throughout the years and therefore we chose to use it as it will produce reliable results.

## **The Advantages and Disadvantages of Blowfish encryption method:**

*Advantages:*

* protection: The encryption using Blowfish is incredibly hard to crack without the key due to its extremely high level of protection.
* Speed: The relatively speedy Blowfish technique makes it possible to encrypt and decrypt data quickly.
* Flexibility: Blowfish is a flexible algorithm that can be adapted to the unique requirements of a given application because it can be employed with a variety of block sizes and key lengths.
* Blowfish is free to use and can be implemented without any licensing limitations because it is in the public domain.

*Disadvantages:*

* The same key is utilized for both encryption and decryption because Blowfish uses a symmetric key method. This implies that keeping track of the keys can be difficult, especially if they are stolen.
* Vulnerabilities: Although Blowfish is thought to be relatively secure, certain flaws in the algorithm have been found throughout time. For instance, a related-key attack may be able to exploit the initial key configuration.
* Limited acceptance: Blowfish's limited use makes it less compatible with various systems and applications than some other encryption methods, including AES.
* Blowfish employs a 64-bit block size that is relatively tiny, hence it might not be appropriate for use in some applications that need bigger block sizes for security reasons.
* Blowfish is a reliable encryption technique that has been applied to a few applications with success. However, it might not be the greatest option in every circumstance, so careful thought should be taken into consideration for both of its advantages and disadvantages before selecting it for a certain use.

## **Explanation of our own algorithm:**

import java.io.\*;

import java.nio.file.\*;

import java.security.MessageDigest;

public class CaesarCipher {

public static void main(String[] args) throws IOException {

String inputFile = "original.txt";

String encryptedFile = "encrypted.txt";

String decryptedFile = "decrypted.txt";

int shift = 3;

String plainPassword = "password";

String hashedPassword = hashPassword(plainPassword);

// Encrypt file

byte[] inputData = readFileAsBinary(inputFile);

byte[] encryptedData = reverseCaesarCipher(inputData, shift);

writeBinaryFile(encryptedFile, encryptedData);

// Decrypt file

byte[] encryptedInputData = readFileAsBinary(encryptedFile);

byte[] decryptedData = reverseCaesarCipher(encryptedInputData, -shift);

writeBinaryFile(decryptedFile, decryptedData);

// Check password

String enteredPassword = "password";

String enteredHash = hashPassword(enteredPassword);

boolean passwordCorrect = enteredHash.equals(hashedPassword);

System.out.println("Password is correct: " + passwordCorrect);

}

public static byte[] readFileAsBinary(String filename) throws IOException {

return Files.readAllBytes(Paths.get(filename));

}

public static void writeBinaryFile(String filename, byte[] data) throws IOException {

Files.write(Paths.get(filename), data);

}

public static byte[] reverseCaesarCipher(byte[] inputBytes, int shift) {

byte[] result = new byte[inputBytes.length];

for (int i = 0; i < inputBytes.length; i++) {

result[i] = (byte) ((inputBytes[i] - shift + 256) % 256);

}

return result;

}

public static String hashPassword(String password) {

try {

MessageDigest Md = MessageDigest.getInstance("SHA-256");

byte[] hashBytes = Md.digest(password.getBytes());

StringBuilder sbr = new StringBuilder();

for (byte bt : hashBytes) {

sbr.append(String.format("%02x", bt));

}

return sbr.toString();

} catch (Exception ex) {

ex.printStackTrace();

}

return null;

}

}

Working with input/output operations and file management in Java both require these imports. What each import performs is as follows:

* java.io.\*: Classes for conducting input and output operations are provided by this package. Classes like File, InputStream, OutputStream, and others are included in it.
* java.nio.file.\*: Classes for interacting with files and directories are included in this package. It has classes for Paths, Files, and other things.
* java.security.MessageDigest: Using several hashing algorithms like MD5, SHA-1, SHA-256, and others, this class offers capabilities for generating hash values or message digests of data. Hash functions are often employed in cryptography and may be used for things like digital signatures, data integrity checks, and password hashing. By importing the java.security.MessageDigest package, you gain access to the MessageDigest class and its methods, which allow you to perform hashing operations.

You may access a variety of classes and methods that make it easier to read from and write to files as well as change file paths by importing these packages.

There is a specified class called CaesarCipher. It has a main method where the program's execution begins.

* ‘public class CaesarCipher’ : CaesarCipher is a public class that is declared in the following line. The logic for utilizing the Caesar cipher technique to encrypt and decode text will be contained in this class.
* ‘public static void main(String[] args) throws IOException’: The main function is defined at this point, which is public static void main(String[] args) throws IOException. It states that it can throw an IOException, which is an exception that might happen during input/output operations, and it receives an array of String arguments (args).
* String inputFile = "original.txt"; : In this line, the String variable inputFile is declared and given the value "original.txt". The name of the input file that will be encrypted is what it stands for.
* String encryptedFile = "encrypted.txt";: In this line, the String variable encryptedFile is declared and given the value "encrypted.txt". It stands for the name of the file that will hold the encrypted text
* String decryptedFile = "decrypted.txt";: The String variable decryptedFile is declared and given the value "decrypted.txt" in this line. It stands for the name of the file that will hold the decrypted text.
* int shift = 3;: shift is an integer variable, and this line declares it and gives it the value 3. It shows how many spots will be moved during encryption and decryption for each character in the input text. The Caesar cipher will in this instance move each character three spaces to the right.

Moving on to the next section of the code.

* String plainPassword = "password";: In this line, a String variable named plainPassword is declared, and the word "password" is given as its value. It stands for the password in plain text that you wish to hash.
* String hashedPassword = hashPassword(plainPassword);: The String variable hashedPassword is declared on this line. It uses the plainPassword as an input and calls the hashPassword function. The password provided must be hashed by the hashPassword function. The hashed password that has been generated is then saved in the hashedPassword variable.

The next set of code is responsible for encrypting the file. The specified password is hashed in this area of the code, and the reverse Caesar cipher method is used to encrypt files. The data that has been encrypted is subsequently written to a file.

* “byte[] inputData = readFileAsBinary(inputFile);”: This line declares the inputData byte array variable. It uses the inputFile as a parameter when using the readFileAsBinary function. The inputFile method's readFileAsBinary method reads the contents of the file and returns them as a byte array. The outputData variable is then used to hold the resultant array.
* “byte[] encryptedData = reverseCaesarCipher(inputData, shift);”: This line declares the encryptedData byte array variable. Invoking the reverseCaesarCipher method, it sends the parameters inputData and shift. Each byte in the array shift locations is moved to the left by the reverseCaesarCipher method as it executes the reverse Caesar encryption technique to the inputData. The encryptedData variable is then used to hold the encrypted data.
* “writeBinaryFile(encryptedFile, encryptedData);”: The encryptedFile and encryptedData parameters are sent to the writeBinaryFile function when this line is called. The writeBinaryFile function creates a file if one doesn't already exist and writes the encryptedData to it.

The following section includes:

* “byte[] encryptedInputData = readFileAsBinary(encryptedFile);”: This line declares the encryptedInputData byte array variable. It uses the encryptedFile as an input and executes the readFileAsBinary function. The encryptedFile method specifies a file, and the readFileAsBinary method reads the contents of that file and returns them as a byte array. In the following step, the array is saved in the encryptedInputData variable.
* “byte[] decryptedData = reverseCaesarCipher(encryptedInputData, -shift);”: This code defines the decryptedData byte array variable. It uses the reverseCaesarCipher method and passes the options -shift and encryptedInputData. The encrypted input data is subjected to the reverse Caesar cipher technique using the reverseCaesarCipher method, which moves each byte in the array -shift places to the right. The data that has been successfully decrypted is then saved in the decryptedData variable.
* “writeBinaryFile(decryptedFile, decryptedData);”: This line uses the writeBinaryFile function and calls it with the parameters decryptedFile and decryptedData. The decryptedData is written via the writeBinaryFile method to the file given by decryptedFile, or to a new file if one doesn't already exist.

The next set of code checks the entered password against the stored hashed password. This portion of the code uses the reverse Caesar cipher technique to decode the encrypted file, writes the decrypted data to a file, and then verifies that the inputted password matches the password that was previously hashed and saved. In the end, it prints if the password entered was typed correctly or not.

* “String enteredPassword = "password";”: This line declares a String variable named enteredPassword and assigns it the value "password". It represents the password entered by the user.
* “String enteredHash = hashPassword(enteredPassword);”: This line declares a String variable named enteredHash. It calls the hashPassword method and passes enteredPassword as an argument. The hashPassword method hashes the entered password, and the resulting hashed password is stored in the enteredHash variable.
* “boolean passwordCorrect = enteredHash.equals(hashedPassword);”: This line declares a boolean variable named passwordCorrect. It compares the enteredHash with the stored hashedPassword using the equals method. If the entered password matches the stored password, passwordCorrect will be set to true; otherwise, it will be set to false.
* “System.out.println("Password is correct: " + passwordCorrect);”: This line prints the result of the password comparison. It displays whether the entered password is correct or not based on the comparison of the hashed passwords.

Two methods are specified in the next code section: writeBinaryFile and readFileAsBinary. These methods provide convenient functions for reading binary data from a file and writing binary data to a file, respectively.

* “public static byte[] readFileAsBinary(String filename) throws IOException”: The filename parameter of this function, which represents the name of the file to be read, is a String. It extracts the file's binary data contents and outputs them as a byte array. If a problem arises while reading the file, the function throws an IOException.
* “Files.readAllBytes(Paths.get(filename))”: To read all the bytes from the file indicated by filename, this line makes use of the Files class from the java.nio.file package. Files.readAllBytes() reads the contents of the file and returns them as a byte array. The Paths.get(filename) function produces a Path object representing the file path.
* “public static void writeBinaryFile(String filename, byte[] data) throws IOException”: This technique requires two inputs: a byte array data holding the binary data to be written and a String parameter filename giving the name of the file to be written. The binary data is written to the chosen file. If a problem arises when writing to the file, the function throws an IOException.
* “Files.write(Paths.get(filename), data)”: The Files.write() function sends the contents of the data array to the file after creating a Path object representing the file path using the Paths.get(filename) method.

Speaking about the next section. Each byte of the incoming data is subjected to the reverse Caesar cipher technique via the reverseCaesarCipher method. In the opposite direction (to the right for encryption or to the left for decryption), it shifts each byte by the set shift value. The output is the data that was obtained, which is returned and placed in the result array.

* “public static byte[] reverseCaesarCipher(byte[] inputBytes, int shift)”: The input data to be encrypted or decrypted is represented by the byte array inputBytes, and the number of places each byte to be moved is represented by the int variable shift.
* “byte[] result = new byte[inputBytes.length];”: This line defines a new result byte array with a length equal to the inputBytes array. The encrypted or decrypted data will be kept in this array.
* with regard to (int i = 0; i inputBytes.length; i++): Beginning with the first byte in the inputBytes array, this line initiates a for loop.
* “result[i] = (byte) ((inputBytes[i] - shift + 256) % 256)”;: Each byte in the inputBytes array is subjected to the reverse Caesar cipher procedure on this line. It then calculates the modulo 256 to circle the byte range (0-255) by deducting the shift value from the current byte value and adding 256 to achieve a positive result. The result value is kept in the result array's appropriate place.
* “return result;”: This line gives back an array of results that includes the data that has been encrypted or decrypted.

The definition of the hashPassword method is found in this portion of the code. Using the SHA-256 algorithm, a password is hashed using this technique. The hashPassword function hashes the supplied password using the SHA-256 algorithm from the MessageDigest class. It returns the hashed password as a String after converting the hashed bytes to a hexadecimal representation.

* “public static String hashPassword(String password)”: This function accepts a String parameter password that represents the password to be hashed. Its return value is a String. The hashed password is sent back as a String.
* “MessageDigest md = MessageDigest.getInstance("SHA-256");”: This line uses the SHA-256 hashing technique to build an instance of the MessageDigest class. The algorithm name is sent as a parameter when using the getInstance function.
* “byte[] hashBytes = md.digest(password.getBytes())”;: the line hashBytes = md.digest(password.getBytes()); calculates the hash value of the password. The getBytes function is used to first transform the password into a byte array
* “StringBuilder sb = new StringBuilder();”: In order to save the hexadecimal representation of the hashed bytes, a StringBuilder object called sb is created with the statement StringBuilder sb = new StringBuilder().
* byte bt: hashBytes for String.format("%02x", bt); sb.append; The hashBytes array's individual bytes are iterated through in this for loop. It uses the String.format method with the %02x format specifier to translate each byte into a two-digit hexadecimal representation. The StringBuilder object is then supplemented with the outcome's hexadecimal representation.
* return sb.toString(); returns the StringBuilder's contents, which corresponds to the hashed password, as a String.
* Any exceptions that may arise while hashing are handled by the catch block, which also publishes the stack trace.
* The function returns null if there is an error or if the hashing operation fails.

In short summary :

The program reads a file, encrypts its content with the reverse Caesar cipher, writes the encrypted data to a file, reads the encrypted data, decrypts it, and then writes the decrypted data to a another file. Additionally, it uses SHA-256 for password hashing and verification.

## **Comparing own algorithm with others:**

|  |  |  |
| --- | --- | --- |
| *Own algorithm (Caesar Cipher)* | *Blowfish* | *Twofish(successor of Blowfish)* |
| **Substitution cipher** | Symmetric encryption scheme | Symmetric encryption scheme |
| **Lower level of security than Blowfish** | Higher level of protection than Caesar Cipher | Higher level of protection than Caesar Cipher |
| **Better suited for little amounts of data** | Can handle greater input sizes | Can handle greater input sizes |
| **Uses a fixed shift value** | Requires a secret key for encryption and decryption | Requires a secret key for encryption and decryption |
| **Easier to use than Blowfish and can be implemented using simple string manipulation** | More complicated than Caesar Cipher and necessitates the use of cryptographic libraries | More complicated than Caesar Cipher and necessitates the use of cryptographic libraries |
| **Only used for file encryption/ decryption in the standalone “CaesarCipher” class.** | Can be used for both passwords and file encryption/ decryption in code. | Can be used for both passwords and file encryption/ decryption in code. |

## **References:**

*What is Blowfish and how is it used in cryptography?* (n.d.). SearchSecurity.

<https://www.techtarget.com/searchsecurity/definition/Blowfish>

Puneet. (2020, September 23). *What is Blowfish? | Advantages and Disadvantages | Encryption Consulting*. Encryption Consulting. <https://www.encryptionconsulting.com/education-center/what-is-blowfish/>

Schneier, B. (2019). *Schneier on Security: The Blowfish Encryption Algorithm*. Schneier.com

<https://www.schneier.com/academic/blowfish/>

Awati, R. (2023). *Blowfish*. TechTarget. <https://www.techtarget.com/searchsecurity/definition/Blowfish>

Ivanovs, A. (April 30, 2023). *Blowfish (cipher).* StackDiary. Retrieved April 30, 2023, from <https://stackdiary.com/glossary/blowfish-cipher/>

Wikipedia. (2021, February 9). *Blowfish (cipher)*. Retrieved May 7, 2023, from <https://en.wikipedia.org/wiki/Blowfish_(cipher)>