Міністерство освіти та науки

Прикарпатський національний університет

Фізико-технічний факультет

Кафедра комп’ютерної інженерії та електроніки

Самостійна робота з курсу

“Захист інформації у комп’ютерних мережах та системах”

Виконав студент групи КІ-41

Воробій Віталій

Івано-Франківськ 2020

ПЛАН

1. Вступ
2. Опис алгоритму BlowFish
3. Криптостійкість алгоритму BlowFish
4. Реалізація алгоритму BlowFish
5. Висновки

ВСТУП

***Blowfish*** — криптографічний алгоритм, який реалізує блочне симетричне шифрування. Розроблений Брюсом Шнайєром в 1993 році. Являє собою шифр на основі мережі Фейстеля. Виконано на простих і швидких операціях: XOR, підстановка, додавання. Не запатентований і вільно поширюваний.

До появи *Blowfish* алгоритми, що існували були або запатентованими, або ненадійними, а деякі і зовсім трималися в секреті (наприклад, *Skipjack*). Алгоритм був розроблений у 1993 році Брюсом Шнайєром як швидка й вільна альтернатива застарілому *DES* і запатентованому *IDEA*. За заявою автора, критерії проектування *Blowfish* були:

1. Швидкість (шифрування на 32-бітних процесорах відбувається за 26 тактів)
2. Простота (за рахунок використання простих операцій, які зменшують імовірність помилки реалізації алгоритму)
3. Компактність
4. Можливість налаштування стійкості.

Blowfish зарекомендував себе, як надійний алгоритм, тому реалізований у багатьох програмах, де не потрібна часта зміна ключа і необхідна висока швидкість шифрування / розшифрування.

1. Хешування паролів
2. Захист електронної пошти і файлів
   1. GnuPG (безпечне зберігання і передача)
3. В лініях зв'язку: зв'язка ElGamal (не запатентований) або RSA (дія патенту закінчилося в 2000 році) і Blowfish замість IDEA
   1. В маршрутизаторі Intel Express 8100 з ключем довжиною 144 біта
4. Забезпечення безпеки в протоколах мережного і транспортного рівня
   1. PuTTY (мережевий рівень)
   2. SSH (транспортний рівень)
   3. OpenVPN (створення зашифрованих каналів)

ОПИС АЛГОРИТМУ BLOWFISH

***Параметри***

1. Секретний ключ K (від 32 до 448 біт)
2. 32-бітові ключі шифрування P1 - P18
3. 32-бітові таблиці замін S1 - S4:

S1 [0] S1 [1] .. S1 [255]

S2 [0] S2 [1] .. S2 [255]

S3 [0] S3 [1] .. S3 [255]

S4 [0] S4 [1] .. S4 [255]

***Функція F(x) в Blowfish***

1. 32-бітний блок ділиться на чотири 8-бітних блоки (X1, X2, X3, X4), кожен з яких є індексом масиву таблиці замін S1-S4
2. Виконується обрахунок наступного виразу



1. Результат цих операцій — значення *F(x).*

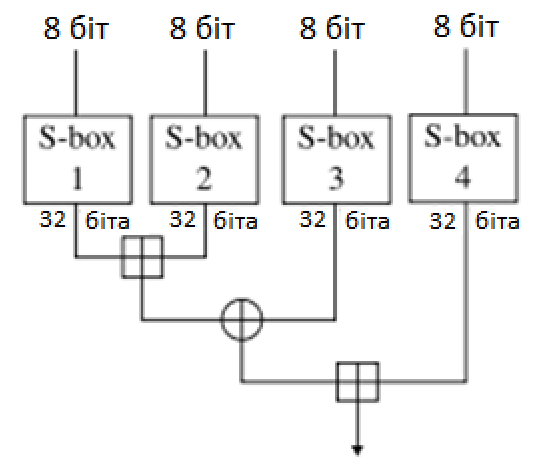
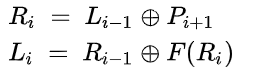


Рисунок 1. Візуальне представлення твірної функції у алгоритмі *BlowFish*

***Алгоритм шифрування 64-бітного блоку з відомим масивом P і F (x)***

1. Поділ на 32-бітові блоки *L0*і *R0*
2. Блоки *L0*і *R0* додаються за модулем 2 з *P0* і *P1*
3. Виконується наступне обчислення:



1. Блоки міняються місцями і передаються користувачу.

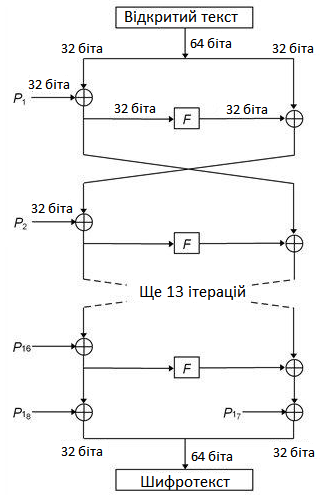


Рисунок 2. Мережа Фейстеля при шифруванні

***Алгоритм BlowFish***

Розділений на 2 етапи:

* Підготовчий — формування ключів шифрування по секретному ключу.
  + Ініціалізація масивів *P* і *S* за допомогою секретного ключа *K*
    - Ініціалізація *P1 - P18* фіксованим рядком, що складається з шістнадцяткових цифр мантиси числа пі.
    - Проводиться операція *XOR* над *P1* з першими 32 бітами ключа *K*, над *P2* з другими 32-бітами і так далі.
    - Якщо ключ *K* коротше, то він накладається циклічно.
  + Шифрування ключів і таблиць замін
    - Алгоритм шифрування 64-бітного блоку, використовуючи початкові ключі *P1 - P18* і таблицю замін *S1 - S4*, шифрує 64 бітну нульовий (0x0000000000000000) рядок. Результат записується в *P1, P2*.
    - *P1 і P2* шифруються зміненими значеннями ключів і таблиць замін. Результат записується в *P3 і P4*.
    - Шифрування триває до зміни всіх ключів *P1 - P18* і таблиць замін *S1 - S4*.
* Шифрування тексту отриманими ключами і *F(x)*, з попереднім розбиттям на блоки по 64 біти. Якщо неможливо розбити початковий текст точно на блоки по 64 біти, використовуються різні режими шифрування для побудови повідомлення, що складається з цілого числа блоків. Сумарні необхідна пам'ять 4168 байт: *P1 - P18* : 18 змінних по 32 біта; *S1-S4*: 4x256 змінних по 32 бита.

Розшифрування відбувається аналогічно, тільки *P1 - P18* застосовуються у зворотному порядку.

***Вибір початкового значення P-масиву і таблиці замін***

Немає нічого особливого в цифрах числа пі. Цей вибір полягає в ініціалізації послідовності, не пов'язаної з алгоритмом, яка могла б бути збережена як частина алгоритму або отримана при необхідності (Пі (число)).

Як вказує Брюс Шнайєр: «Підійде будь-який рядок з випадкових бітів цифр числа e, RAND-таблиці, або випадкові згенеровані цифри.»

КРИПТОСТІЙКІСТЬ АЛГОРИТМУ BLOWFISH

Криптостійкість головним чином залежить від *F(x)*. На це вказав *Serge Vaudenay*, кажучи про наявність невеликого класу слабких ключів (таких, що утворюють слабкі *S-box*): ймовірність появи слабкого *S-box* дорівнює *2-15*.

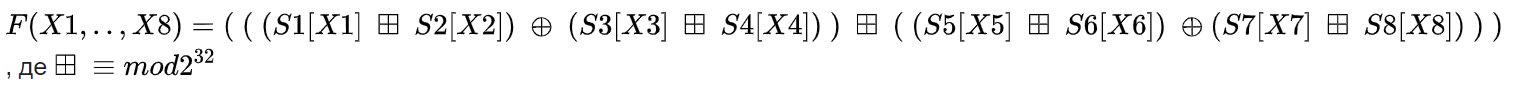
Він також розглянув спрощений варіант Blowfish, з відомою функцією *F(x)* і слабким ключем. Для цього варіанту потрібно обраних відкритих текстів (t - число раундів, а символи [] означають операцію отримання цілої частини числа). Ця атака може бути використана тільки для алгоритму з t <= 7. Для t = 7 потрібно 224 відкритих текстів, причому для варіанту з відомим *F (x)* і випадковим ключем потрібно 248 відкритих текстів. Але дана атака не ефективна для Blowfish з 16 раундами.

John Kelsey розробив атаку, яка дозволяла зламати 3-ітераційний Blowfish. Вона спирається на факт, що операції додавання по модулю 232 і XOR не комутативні.

Неможливо заздалегідь визначити чи є ключ слабким. Проводити перевірку можна тільки після генерації ключа.

Криптостійкість можна налаштовувати за рахунок зміни кількості раундів шифрування (збільшуючи довжину масиву *P*) і кількості використовуваних *S-box*. При зменшенні використовуваних *S-box* зростає ймовірність появи слабких ключів, але зменшується використовувана пам'ять. Для адаптації Blowfish на 64-бітної архітектуру, можна збільшити кількість і розмір *S-box* (а отже і пам'ять для масивів *P* і *S*), а також ускладнити *F(x),* причому для алгоритму з такою функцією *F(x)* неможливі вищевказані атаки.

Модифікація F (x): на вхід подається 64-бітний блок, який ділиться на вісім 8-бітних блоків (X1-X8). Результат обчислюється за формулою:



На сьогоднішній день не існувало атак, які виконуються за розумний час. Успішні атаки можливі тільки через помилки реалізації.

РЕАЛІЗАЦІЯ АЛГОРИТМУ BLOWFISH

Алгоритм *BlowFish* я реалізував з допомогою мови програмування *Java*. Для автоматизованого тестування алгоритму і його компонентів я використав бібліотеку *JUnit 5* версії і для створення графічної демонстраційної програми я застосував програмний пакет *JavaFX*. У якості збірника проекту (build tool) я обрав Maven.

Всю логіку роботи алгоритму BlowFish я виніс у окремий клас:

package org.example.algorithm;  
  
import java.util.ArrayList;  
import java.util.List;  
  
public class BlowFish {  
  
 */\*\*  
 \* Максимальна кількість байтів у ключі  
 \*/* public static final int *KEY\_MAX\_BYTES* = 56;  
  
 */\*\*  
 \* Параметри блоку P  
 \*/* private static final int *P\_SIZE* = 18;  
  
 */\*\*  
 \* Параметри матриці S  
 \*/* private static final int *S\_ROWS* = 4, *S\_COLUMNS* = 256;  
  
 */\*\*  
 \* Блок P  
 \*/* private final int[] P = new int[*P\_SIZE*];  
  
 */\*\*  
 \* Матриця S  
 \*/* private final int[][] S = new int[*S\_ROWS*][*S\_COLUMNS*];  
  
 */\*\*  
 \* Конструювання об'єкту без ініціалізації  
 \*/* public BlowFish() {}  
  
 */\*\*  
 \* Ініціалізація об'єкту з допомогою ключа, поданого у текстовому вигляді  
 \*/* public BlowFish(String key) {  
 final byte[] rawKey = Algorithms.*stringToByteArray*(key);  
 init(rawKey);  
 }

*/\*\*  
 \* Ініціалізація об'єкту з допомогою ключа, поданого у вигляді масиву байт  
 \*/* public BlowFish(byte[] rawKey) {  
 init(rawKey);  
 }  
  
 */\*\*  
 \* Функція встановлення ключа  
 \*/* public void setKey(String key) {  
 final byte[] rawKey = Algorithms.*stringToByteArray*(key);  
 init(rawKey);  
 }  
  
 */\*\*  
 \* Ініціалізація конфігурації BlowFish  
 \* @param rawKey Ключ  
 \*/* private void init(byte[] rawKey) {  
 final int KEY\_LENGTH = rawKey.length;  
  
 if (KEY\_LENGTH > *KEY\_MAX\_BYTES*) {  
 throw new IllegalArgumentException("Розмір ключа занадто довгий (> 448 бітів)");  
 }  
  
 *// Ініціалізація P i S* System.*arraycopy*(*pi*, 0, P, 0, *P\_SIZE*);  
  
 for (int i = 0; i < *S\_ROWS*; i++) {  
 System.*arraycopy*(*pi*, *P\_SIZE* + *S\_COLUMNS* \* i, S[i], 0, *S\_COLUMNS*);  
 }  
  
 *// XOR P з вхідним ключем* int tmp = 0;  
 int nKeyPos = 0;  
 for (int i = 0; i < *P\_SIZE*; i++) {  
 for (int j = 0; j < 4; j++) {  
 tmp <<= 8;  
 tmp |= 0x0ff & rawKey[nKeyPos++ % KEY\_LENGTH];  
 }  
 P[i] ^= tmp;  
 }  
  
 *// Заміна P, S обчисленими значенням* int[] data = new int[2];  
  
 for (int i = 0; i < *P\_SIZE*; i += 2) {  
 cipherBlock(data);  
 P[i] = data[0];  
 P[i+1] = data[1];  
 }  
  
 for (int i = 0; i < *S\_ROWS*; i++) {  
 for (int j = 0; j < *S\_COLUMNS*; j += 2) {  
 cipherBlock(data);  
 S[i][j] = data[0];  
 S[i][j+1] = data[1];  
 }  
 }  
 }

*/\*\*  
 \* Клас, що містить необхідні алгоритми для роботи BlowFish  
 \*/* public static class Algorithms {  
  
 */\*\*  
 \* Функція для створення бітової маски  
 \*/* public static Long createBitMask(int from, int to) {  
 long mask = 0L;  
 for (int i = from; i < to; i++) {  
 mask |= 1L << i;  
 }  
 return mask;  
 }  
  
 */\*\*  
 \* Функція перетворення тексту в масив байтів  
 \* @param string Вхідний текст  
 \*/* public static byte[] stringToByteArray(String string) {  
 final int N = string.length();  
 final byte[] rawKey = new byte[N \* 2];  
 for (int i = 0; i < N \* 2; i += 2) {  
 char c = string.charAt(i / 2);  
 rawKey[i] = (byte) (c << 8);  
 rawKey[i + 1] = (byte) c;  
 }  
 return rawKey;  
 }  
  
 */\*\*  
 \* Функція перетворення 64 бітного значення на два 32 бітних  
 \* @param v Вхідне 64 бітне значення  
 \*/* public static int[] splitLongToInts(long v) {  
 long mask = *createBitMask*(0, 32);  
 return new int[]{  
 (int) ((v >> 32) & mask),  
 (int) (v & mask)};  
 }  
  
 */\*\*  
 \* Функція поєднання двох 32 бітних чисел у одне 64 бітне  
 \* @param numbers Масив чисел  
 \* @throws IllegalArgumentException коли розмір масиву не рівний двійці  
 \*/* public static long mergeIntsToLong(int[] numbers) {  
 if (numbers.length != 2)  
 throw new IllegalArgumentException("Розмір масиву повинен бути рівний двійці");  
 return ((long) numbers[0] << 32) | numbers[1] & *createBitMask*(0, 32);  
 }  
  
 */\*\*  
 \* Функція перетворення тексту в список 64-розрядних чисел  
 \*/* public static List<Long> convertStringToList(String str) {  
 final List<Long> resultList = new ArrayList<>();  
 for (char c : str.toCharArray()) {  
 resultList.add((long) c);  
 }  
 return resultList;  
 }  
 */\*\*  
 \* Абстракція функції, що маніпулює блоком  
 \*/* @FunctionalInterface  
 interface BlockManipulationFunction {  
 void perform(int[] block);  
 }  
  
 */\*\*  
 \* Функція для застосування функції до кожного елементу списку  
 \* @param inputValues Список вхідних значень  
 \* @param func Функція маніпулювання блоком  
 \*/* private List<Long> apply(List<Long> inputValues, BlockManipulationFunction func) {  
 final List<Long> cipheredValues = new ArrayList<>();  
 for (Long v : inputValues) {  
 int[] block = Algorithms.*splitLongToInts*(v);  
 func.perform(block);  
 long resValue = Algorithms.*mergeIntsToLong*(block);  
 cipheredValues.add(resValue);  
 }  
 return cipheredValues;  
 }  
  
 */\*\*  
 \* Функція шифрування списку чисел  
 \*/* public List<Long> cipher(List<Long> inputValues) {  
 return apply(inputValues, this::cipherBlock);  
 }  
  
 */\*\*  
 \* Функція дешифрування списку чисел  
 \*/* public List<Long> decipher(List<Long> cipheredValues) {  
 return apply(cipheredValues, this::decipherBlock);  
 }  
  
 */\*\*  
 \* Функція шифрування розбитого 64 блоку  
 \*/* public void cipherBlock(int[] value) {  
 int left = value[0];  
 int right = value[1];  
  
 left ^= P[0];  
  
 for (int i = 1; i < 16; i += 2) {  
 right ^= F(left) ^ P[i];  
 left ^= F(right) ^ P[i + 1];  
 }  
  
 right ^= P[17];  
  
 value[0] = right;  
 value[1] = left;  
 }

*/\*\*  
 \* Функція дешифрування розбитого 64 бітного блоку.  
 \*/* public void decipherBlock(int[] value) {  
 int left = value[1];  
 int right = value[0];  
  
 right ^= P[17];  
  
 for (int i = 16; i > 1; i -= 2) {  
 left ^= P[i] ^ F(right);  
 right ^= P[i - 1] ^ F(left);  
 }  
  
 left ^= P[0];  
  
 value[0] = left;  
 value[1] = right;  
 }  
  
 */\*\*  
 \* Твірна функція Фейстеля для алгоритму BlowFish  
 \*/* private int F(int v) {  
 return (( S[0][ v >>> 24 ]  
 + S[1][(v >> 16) & 0xff])  
 ^ S[2][(v >> 8) & 0xff])  
 + S[3][ v & 0xff];  
 }  
  
 */\*\*  
 \* Шістнадцяткові цифри мантиси числа пі  
 \*/* private static final int[] *pi* = {  
  
 0x243f6a88, 0x85a308d3, 0x13198a2e, 0x03707344,  
 0xa4093822, 0x299f31d0, 0x082efa98, 0xec4e6c89,  
 0x452821e6, 0x38d01377, 0xbe5466cf, 0x34e90c6c,  
 0xc0ac29b7, 0xc97c50dd, 0x3f84d5b5, 0xb5470917,  
 0x9216d5d9, 0x8979fb1b,

…   
 };  
  
}

І користуючись, зазделегідь встановленою бібліотекою Junit я написав тест для вищенаведеного класу:

package org.example.algorithm;  
  
import org.junit.jupiter.api.Test;  
import org.junit.jupiter.api.DisplayName;  
  
import java.util.List;  
  
import static org.junit.jupiter.api.Assertions.\*;  
public class BlowFishTest {  
  
 @Test  
 @DisplayName("Коли довжина ключа більша 448 біт стається помилка")  
 public void throwsIllegalArgumentExceptionWhenKeyIsTooLong() {  
 final int TOO\_BIG\_SIZE = 500;  
  
 *// Ініціалізація списку байтів занадто великого розміру* final byte[] bytes = new byte[TOO\_BIG\_SIZE];  
  
 *assertThrows*(IllegalArgumentException.class, () ->  
 new BlowFish(bytes)  
 );  
 }  
  
 @Test  
 @DisplayName("Перевірка розкладення 64 бітного числа на два 32 бітних")  
 public void testSplittingLongWorksFine() {  
 final long v = Long.*MAX\_VALUE*;  
 final int[] retrieved = BlowFish.Algorithms.*splitLongToInts*(v);  
  
 *assertEquals*(2, retrieved.length);  
 *assertEquals*(31, Integer.*toBinaryString*(retrieved[0]).length());  
 *assertEquals*(32, Integer.*toBinaryString*(retrieved[1]).length());  
 }  
  
 @Test  
 @DisplayName("Перевірка шифрування/розшифрування")  
 public void testCipherDecipherCombination() {  
 final BlowFish blowFish = new BlowFish("Деякий ключ");  
 final List<String> inputList = List.*of*(  
 "Тестування алгоритму",  
 "Testing of the algorithm",  
 "TestTest123...!",  
 "Валідація Blowfish алгоритму",  
 "Validation of this algorithm"  
 );  
 for (String str : inputList) {  
 final List<Long> inputValues = BlowFish.Algorithms.*convertStringToList*(str);  
 final List<Long> cipheredValues = blowFish.cipher(inputValues);  
 final List<Long> decipheredValues = blowFish.decipher(cipheredValues);  
 *assertEquals*(inputValues, decipheredValues);  
 }  
 }

@Test  
 @DisplayName("Перевірка зчеплення двох 32 - бітних чисел у одне 64 бітне")  
 public void checkMerge32bitsNumbers() {  
 int a = 0b000000000000000000000000000001;  
 int b = 0b000000000000000000000000000001;  
 long merged = BlowFish.Algorithms.*mergeIntsToLong*(new int[] {a, b});  
 *assertEquals*("100000000000000000000000000000001", Long.*toBinaryString*(merged));  
 }

@Test  
 @DisplayName("Коли у функції переданий масив розмірністю не рівною 2 - отримуємо помилку")  
 public void assertMergeThrowsExceptionWhenInputArraysSizeMismatch() {  
 *assertThrows*(IllegalArgumentException.class, () ->  
 BlowFish.Algorithms.*mergeIntsToLong*(new int[] {}));  
 *assertThrows*(IllegalArgumentException.class, () ->  
 BlowFish.Algorithms.*mergeIntsToLong*(new int[] {1, 2, 3}));  
 }  
  
}

Провів тестування у середовищі *Intellij IDEA*:

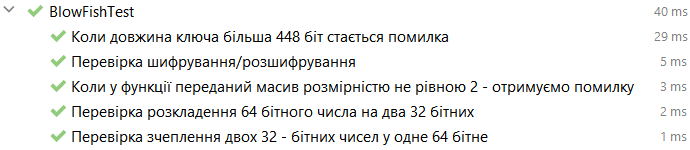


Рисунок 3. Результат проведення тестування

Користуючись програмою Scene Builder я розробив графічний інтерфейс.

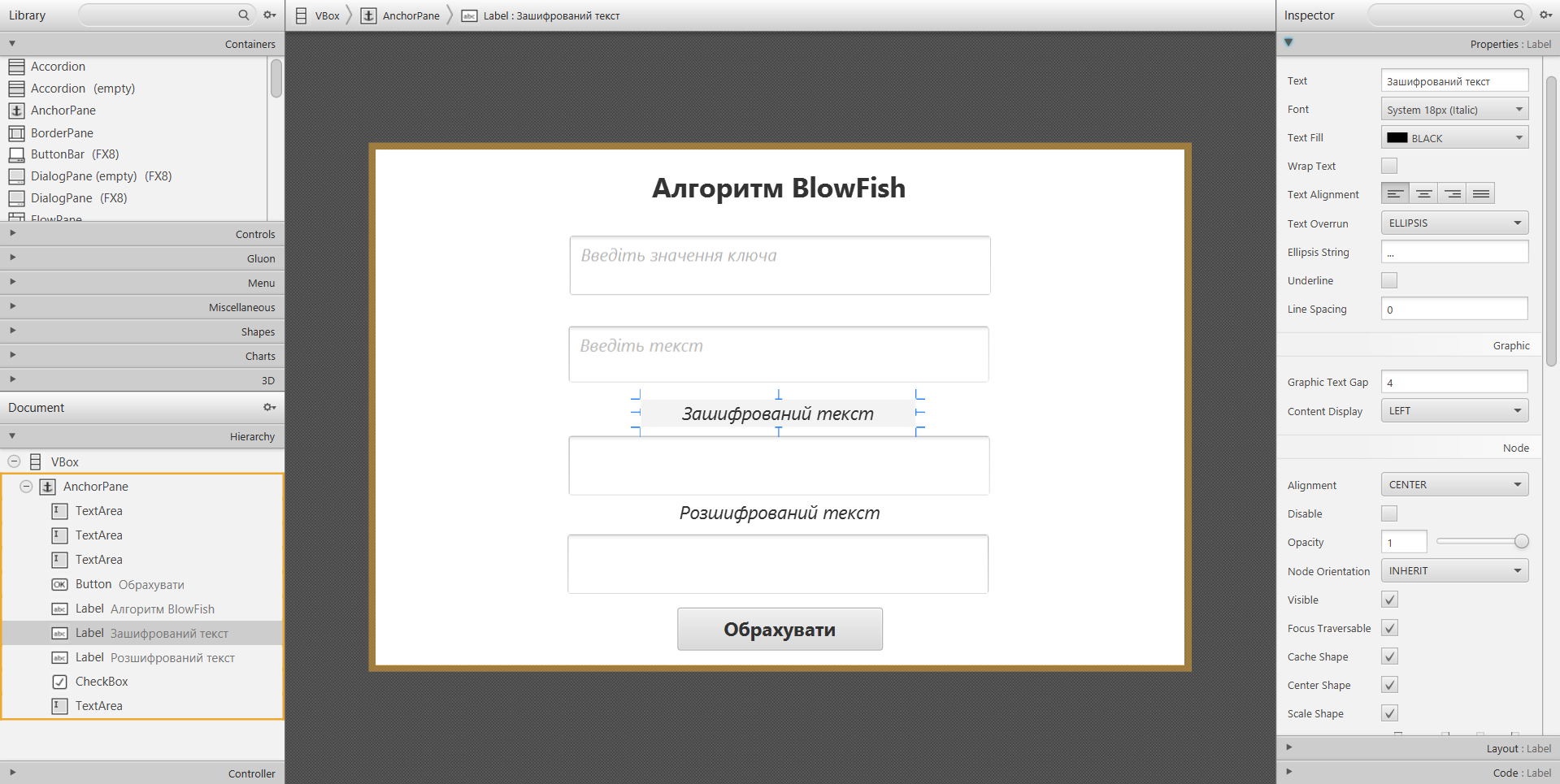


Рисунок 4. Розроблений графічний інтерфейс у програмі Scene Builder

Написав клас, що відповідає за завантаження файлу, що описує розмітку графічного інтерфейсу і відображає його у вікні.

package org.example;  
  
import javafx.application.Application;  
import javafx.scene.Parent;  
import javafx.scene.Scene;  
import javafx.fxml.FXMLLoader;  
import javafx.stage.Stage;  
  
import java.io.IOException;  
import java.net.URL;  
  
*/\*\*  
 \* JavaFX App  
 \*/*public class App extends Application {  
  
 @Override  
 public void start(Stage primaryStage) throws IOException {  
 FXMLLoader loader = new FXMLLoader();  
 URL xmlUrl = getClass().getResource("/gui.fxml");  
 loader.setLocation(xmlUrl);  
 Parent root = loader.load();  
  
 primaryStage.setScene(new Scene(root));  
 primaryStage.setTitle("BlowFish");  
 primaryStage.show();  
  
 }  
  
 public static void main(String[] args) {  
 *launch*();  
 }

}

Написав клас-контроллер, що відповідає за обробку подій під час виконання програми і зміну вмісту сцени.

package org.example;  
  
import javafx.event.ActionEvent;  
import javafx.fxml.FXML;  
import javafx.scene.control.Button;  
import javafx.scene.control.CheckBox;  
import javafx.scene.control.TextArea;  
import org.example.algorithm.BlowFish;  
  
import java.util.List;  
  
public class Controller {  
  
 */\*\*  
 \* Функція для перетворення списку 64-розрядних чисел у текст  
 \*/* public static String convertNumberListToString(List<Long> inputList) {  
 final StringBuilder builder = new StringBuilder();  
 for (long v : inputList) {  
 builder.append((char) v);  
 }  
 return builder.toString();  
 }

private final BlowFish blowFish = new BlowFish();

@FXML  
 public TextArea inputText;

@FXML  
 public TextArea cipheredText;

@FXML  
 public TextArea decipheredText;

@FXML  
 public Button calcButton;

@FXML  
 public CheckBox isDecipheredTextCorrect;

@FXML  
 public TextArea keyArea;

*/\*\*  
 \* Попереднє значення ключа  
 \*/* private String prevKey = null;

@FXML  
 public void onCalcButtonClicked(ActionEvent ignored) {  
 final String keyAreaText = keyArea.getText();  
  
 *// Якщо попереднє значення ключа рівне поточному - не робимо повторну ініціалізацію* if (!keyAreaText.equals(prevKey)) {  
  
 if (keyAreaText.length() > BlowFish.*KEY\_MAX\_BYTES* / 2) {  
 blowFish.setKey(keyAreaText.substring(0, BlowFish.*KEY\_MAX\_BYTES* / 2));  
 } else {  
 blowFish.setKey(keyAreaText);  
 }  
  
 prevKey = keyAreaText;  
 }  
  
 isDecipheredTextCorrect.setVisible(true);  
  
 *// Робимо текстове поле не активним* inputText.setDisable(true);  
  
 final String input = inputText.getText();  
  
 *// Перетворення вхідного тексту в список чисел* final List<Long> inputList = BlowFish.Algorithms.*convertStringToList*(input);  
  
 *// Обрахунок шифрування і дешифрування* final List<Long> cipheredList = blowFish.cipher(inputList);  
 final List<Long> decipheredList = blowFish.decipher(cipheredList);  
  
 final String cipheredValue = *convertNumberListToString*(cipheredList);  
 final String decipheredValue = *convertNumberListToString*(decipheredList);  
  
 cipheredText.setText(cipheredValue);  
 decipheredText.setText(decipheredValue);  
  
 *// Встановлюємо значення прапорця у вікні в залежності від правильності розшифрування* isDecipheredTextCorrect.setSelected(input.equals(decipheredValue));  
  
 *// Розблоковуємо текстове поле* inputText.setDisable(false);  
 }  
  
}

Під час запуску програми я отримав наступне:

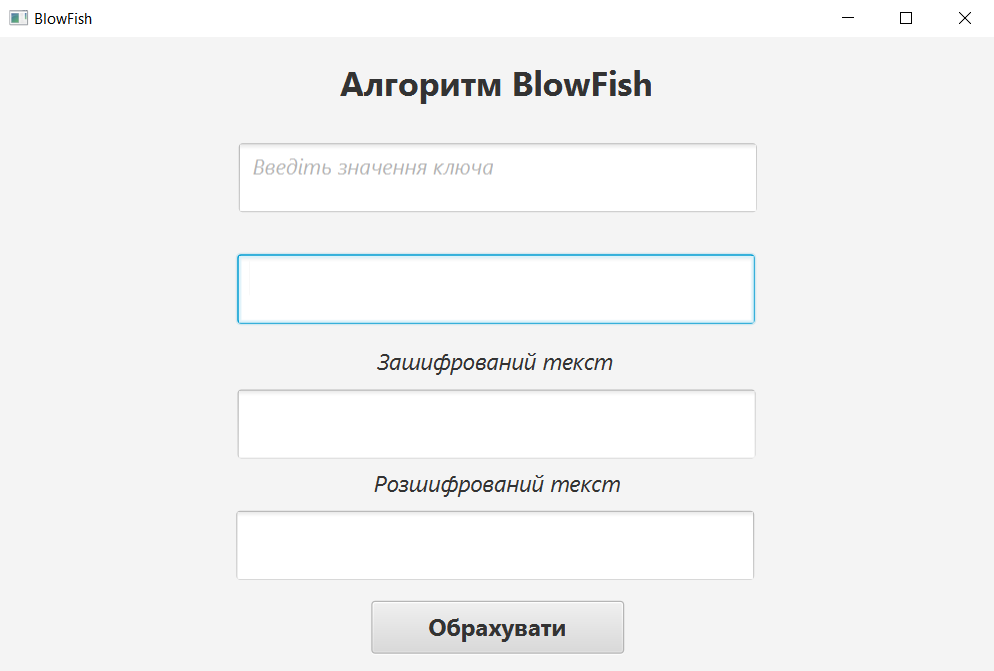


Рисунок 5. Початковий вигляд програми

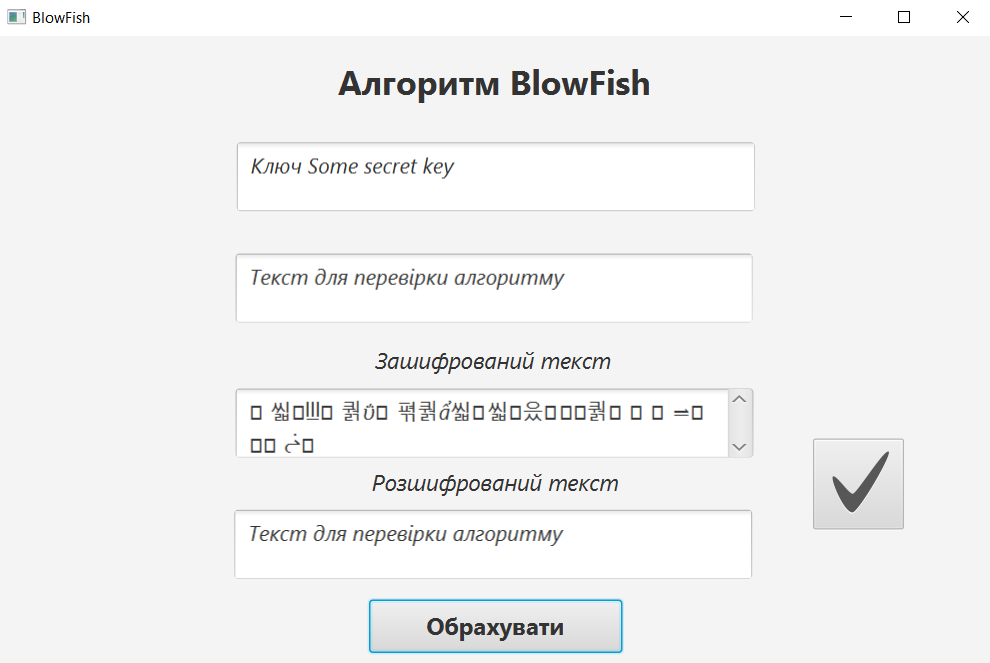


Рисунок 6. Вигляд програми після вводу ключа і тексту для шифрування

ВИСНОВКИ

***Blowfish*** — криптографічний алгоритм, який реалізує блочне симетричне шифрування. Як було показано, цей алгоритм є швидким, широко розповсюдженим і в той же час надійним. У цій роботі було представлено опис алгоритму, його криптостійкість

У цій роботі я також успішно реалізував алгоритм BlowFish, протестував його і написав програму з графічним інтерфейсом, що шифрує і розшифровує текст за цим алгоритмом.