МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «ОДЕСЬКА ЮРИДИЧНА АКАДЕМІЯ»
ФАКУЛЬТЕТ КІБЕРБЕЗПЕКИ ТА ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ

Кафедра інформаційних технологій

КВАЛІФІКАЦІЙНА РОБОТА

на тему:

«СИСТЕМА ПОТОЧНОГО ШИФРУВАННЯ НА ОСНОВІ ЛІНІЙНОГО ІТЕРАЦІЙНОГО XORSHIFT–ГЕНЕРАТОРА»

Виконав: здобувач 4 курсу
Рівень бакалавр
Галузь знань 12 «Інформаційні технології»,
спеціальність 122 Комп’ютерні науки
Савенко Михайло Ігорович

Керівник: к.т.н., доцент
Щербина Юрій Володимирович

Рецензент: к.т.н., доцент
Чикунов Павло Олександрович

Одеса – 2024

НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «ОДЕСЬКА ЮРИДИЧНА АКАДЕМІЯ»
Факультет кібербезпеки та інформаційних технологій
Кафедра інформаційних технологій
Галузь знань: 12 Інформаційні технології
Спеціальність: 122 Комп'ютерні науки
Назва освітньої програми: Аналіз та безпека даних

ЗАТВЕРДЖУЮ
Завідувач кафедри інформаційних технологій
доцент _____ Н.І. Логінова
«____ »_____ 20__ року

З А В Д А Н Н Я
на кваліфікаційну (бакалаврську) роботу
здобувачу Савенку Михайлу Ігоровичу

1. Тема роботи: «Система поточного шифрування на основі лінійного ітераційного Xorshift-генератора»
Керівник роботи: к.т.н. доцент Щербина Юрій Володимирович
затверджені наказом НУ «ОЮА» від «02» квітня 2024 року № 809-06

2. Строк подання здобувачем роботи «20» травня 2024 року

3. Дата видачі завдання «15» вересня 2023 року

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів бакалаврської роботи	Строк виконання етапів роботи	Примітка
1.	Аналіз способів поточного шифрування та вимоги до їх програмної реалізації	02.10.2023	
2.	Обгрунтування вибору алгоритму поточного шифрування	14.02.2024	
3.	Програмна реалізація алгоритму шифрування	20.04.2024	
4.	Оформлення пояснювальної записки	17.05.2024	

Здобувач _____
(підпис) (прізвище та ініціали)

Керівник роботи _____
(підпис) (прізвище та ініціали)

АНОТАЦІЯ

Савенко М. І. Система поточного шифрування на основі лінійного ітераційного Xorshift-генератора. – Кваліфікаційна робота бакалавра за спеціальністю 122 Комп'ютерні науки.

Кваліфікаційна робота викладена на 54 сторінках основного тексту і 64 сторінках загального тексту, містить 3 розділи, 18 рисунків, 2 таблиці, 2 додатки, 12 використаних джерел.

Метою роботи є розроблення програмного продукту, який реалізує криптографічну систему поточного шифрування даних в реальному масштабі часу для їх подальшого передавання в каналах зв'язку автоматизованих комп'ютерних систем.

Об'єктом дослідження є оцінка криптографічної стійкості потокового шифрування даних з використанням критерію χ^2-Пірсона.

Предметом дослідження є способи рекурентного формування послідовностей псевдовипадкових чисел з рівномірним розподіленням ймовірностей бінарних символів, що використовуються для криптографічних потреб.

В процесі виконання роботи створено програмний застосунок, що дозволяє виконувати на заданому рівні криптографічний захист даних, що обробляються і передаються в автоматизованих системах. Виконано аналіз існуючим способів формування шифруючої гами для потокових шифрів і обрано найбільш ефективний його варіант. Дано обґрунтування способу оцінки запропонованого алгоритму шифрування, а саме використанню критерію χ^2-Пірсона для визначення ступіню рівномірності розподілення ймовірностей в шифруючій гамі та зашифрованому тексті. Проведено статистичні випробування криптографічної стійкості даних зашифрованих розробленим шифром.

Ключові слова: поточний шифр, псевдовипадкова послідовність, критерій Пірсона, статистичний аналіз.

ABSTRACT

Savenko M. I. A stream encryption sysbased on a linear iterative Xorshift generator. – Bachelor's qualifying work in the specialty 122 "Computer Science".

The qualification work is laid out on 54 pages of the main text and 64 pages of the general text, contains 3 chapters, 18 figures, 2 tables, 2 appendices, 12 used sources.

The purpose of the work is to develop a software product that implements a cryptographic system of current encryption of data in real time for their further transmission in the communication channels of automated computer systems.

The object of the study is to assess the cryptographic stability of streaming data encryption using the χ^2-Pearson test.

The subject of research is methods of recurrent formation of sequences of pseudo-random numbers with uniform distribution of probabilities of binary symbols used for cryptographic needs.

In the process of performing the work, a software application was created, which allows to perform cryptographic protection of data processed and transmitted in automated systems at a given level. An analysis of the existing methods of forming the cipher range for stream ciphers was performed and the most effective variant was selected. The rationale for the method of evaluating the proposed encryption algorithm is given, namely the use of the χ2-Pearson test to determine the degree of uniformity of the distribution of probabilities in the encryption scheme and the encrypted text. Statistical tests of cryptographic stability of data encrypted with the developed cipher were conducted.

Key words: current cipher, pseudorandom sequence, Pearson test, statistical analysis.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ 6
ВСТУП 7
РОЗДІЛ 1. АНАЛІЗ СПОСОБІВ ПОТОЧНОГО ШИФРУВАННЯ ТА ВИМОГИ ДО ЇХ ПРОГРАМНОЇ РЕАЛІЗАЦІЇ 10
1.1 Принцип поточного шифрування і вимоги до поточних шифрів 10
1.2 Математична модель поточного шифрування 13
1.3 Вибір інструментальних засобів реалізації поточного шифру 19
1.4 Висновки за розділом 1 21
РОЗДІЛ 2. ОБГРУНТУВАННЯ ВИБОРУ АЛГОРИТМУ ПОТОЧНОГО ШИФРУВАННЯ 22
2.1 Теоретичні підходи до вибору алгоритму поточного шифрування 22
2.2 Програмні способи формування потоку псевдовипадкових чисел 26
2.3 Обґрунтування вибору генератора псевдовипадкових чисел 33
2.4 Висновки за розділом 2 36
РОЗДІЛ 3. ПРОГРАМНА РЕАЛІЗАЦІЯ АЛГОРИТМУ ШИФРУВАННЯ 37
3.1 Вимоги до програмної реалізації поточного шифру 37
3.2 Програмна реалізація алгоритму генератора ПВЧ 39
3.3 Програмна реалізація алгоритму поточного шифрування 42
3.4 Дослідження і оцінка моделі побудованого поточного алгоритму 45
2.4 Висновки за розділом 3 50
ВИСНОВКИ 51
СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ 53
ДОДАТКИ 55
Додаток А. Програмний код функцій формування шифруючої гами та функцій для знищення тимчасового зовнішнього файлу 55
Додаток Б. Програмний код форми графічного інтерфейсу і функції шифрування 57

ПЕРЕЛІК СКОРОЧЕНЬ

ПВЧ – Псевдовипадкові числа

ПВП – Псевдовипадкова послідовність чисел

ЕОМ – Електронно-обчислювальна машина

ОБ – Одноразовий блокнот

CFB – Cipher Feedback Mode – (Режим зворотного зв’язку по шифртексту)

ЛКГ – Лінійний конгруентний генератор

ЛКП – Лінійна конгруентна послідовність

PRG – Pseudo Random Generator – (Генератор псевдовипадкових чисел)

LFSR – Linear feedback shift register – (Лінійний регістр зі зворотним зв’язком)

AES – Advanced Encryption Standard – (Конкурс на новий криптографічний) стандарт

NIST – National Institute of Standards and Technology (Національний інститут стандартів і технологій США)

ВСТУП

Зміни, що відбуваються у сфері сучасних Internet-технологій, висувають на передній план питання, пов’язані із удосконаленням засобів захисту даних, які передаються і обробляються в сучасних комп’ютерних мережах. Це пояснюється тим, що значна частина компаній, працюючих у різних сегментах сучасного ринку, переходять на роботу в онлайн-режимі, а це у свою чергу, висуває додаткові вимоги до систем захисту. Зокрема, виникає запит на сучасні поточні системи шифрування, що працюють у режимі реального часу.

Підвищення обчислювальних ресурсів сучасної комп’ютерної техніки дає порушникам інформаційної безпеки додаткові можливості для подолання систем комп’ютерного захисту, зокрема, зламу систем розподілення ключової інформації та криптографічних засобів захисту. Все це, відповідно, вимагає додаткових зусиль розробників направлених на підвищення стійкості засобів захисту даних, що зберігаються, передаються і обробляються в розподілених комп’ютерних системах. Із цього витікає, що подальше удосконалення систем шифрування, у тому числі, поточних криптографічних систем є актуальною задачею.

Проблема підвищення ефективності поточного шифрування може бути вирішеною шляхом підвищення швидкодії генератора шифруючого числового потоку за рахунок використання елементарних обчислювальних операцій, що виконуються за обмежену кількість тактів роботи мікропроцесора. На даний момент існує певна кількість напрямів створення таких генераторів та їх готових варіантів. Задача полягає в тому, щоб, на основі їх детального аналізу, вибрати найбільш придатний для використанні в сучасних умовах функціонування систем захисту.

Проведені у процесі дипломного проектування дослідження генераторів шифруючи числових послідовностей, базуються на комп’ютерному моделюванні роботи рекурентних генераторів псевдовипадкових послідовностей та їх подальшому статистичному аналізі з використанням критерію χ^2-Пірсона. В

роботі використовувалися наукові публікації таких авторів як: Bruce Schneier, Claude E. Shannon, Knuth, D. E., George Marsaglia та ін.

Метою роботи є розроблення програмного продукту, який реалізує криптографічну систему поточного шифрування даних в реальному масштабі часу для їх подальшого передавання в каналах зв'язку автоматизованих комп'ютерних систем.

Для досягнення поставленої мети в кваліфікаційній роботі необхідно вирішити наступні **завдання**:

– провести дослідження предметної області і надати математичне обґрунтування моделей поточного шифрування в розподілених комп'ютерних системах;

– проаналізувати теоретичні аспекти способів рекурентного формування послідовностей псевдовипадкових чисел, що використовуються для криптографічних потреб;

– провести аналіз та оцінку напрямів створення генераторів псевдовипадкових чисел, робити вибір найбільш ефективного варіанту для поточного шифрування;

– обґрунтувати вибір мови програмування та відповідного програмного середовища для виконання поставленої задачі;

– створити програмний застосунок, що реалізує обрану криптографічну модель шифрування;

– провести випробування створеного програмного за стосунку і виконати статистичну оцінку результатів шифрування.

Об'єктом дослідження є оцінка криптографічної стійкості потокового шифрування даних з використанням критерію χ^2-Пірсона.

Предметом дослідження є способи рекурентного формування послідовностей псевдовипадкових чисел з рівномірним розподіленням ймовірностей бінарних символів, що використовуються для криптографічних потреб.

В кваліфікаційній роботі використовувалися загальнонаукові та спеціальні методи наукового дослідження такі, як дискретна математика, моделювання, статистичний аналіз тощо.

Практичне значення отриманих результатів полягає у створені програмного за стосунку, який має зручний графічний інтерфейс, що дозволяє встановлювати початкові умови шифрування та контролювати процес його виконання.

Структура кваліфікаційної роботи відповідає меті та завданням дослідження і складається зі вступу, основної частини з трьох розділів, висновків та списку використаних джерел. Робота викладена на 64 сторінках, містить 18 рисунків, 2 таблиці. Список використаних джерел включає 12 джерел.

1. АНАЛІЗ СПОСОБІВ ПОТОЧНОГО ШИФРУВАННЯ ТА ВИМОГИ ДО ЇХ ПРОГРАМНОЇ РЕАЛІЗАЦІЇ

1.1 Принцип поточного шифрування і вимоги до поточних шифрів

Зростання об’ємів інформації, що передається в каналах сучасних інформаційних систем висуває на передній план задачу створення нових ефективних способів захисту інформаційних потоків. Рішення цієї актуальної задачі вимагає розроблення обчислювально стійких криптографічних систем.

Рівень безпеки, що забезпечується шифром в сучасних розподілених автоматизованих системах вирішальне значення, але особиста увага, так само, надається вартості обчислювальних ресурсів, що вимагає реалізація криптографічного алгоритму та швидкодії, що він забезпечує. Саме з цієї причини суттєву роль в організації обміну даними в комп’ютерних мережах відіграють поточні шифри.

Потокові шифри (stream cipher) уявляють собою окремий клас симетричних криптографічних алгоритмів, в яких потік елементів відкритого тексту послідовно перетворюється в елементи шифрованого тексту. В принципі, між блоковими і потоковими шифрами (ПШ) не існує чіткої межі розділення. Зазвичай, алгоритми блочного шифрування використовують в режимі гамування.

Поточний алгоритм шифрування дозволяє працювати в реальному часі за рахунок відсутності необхідності розбивання відкритого тексту на окремі блоки. Його робота основується на формуванні шифрувальних бітових послідовностей з близькими до «білого шуму» властивостями. Такі послідовності називають шифруючими гамами, а розподілення ймовірностей бітових символів у їх складі має бути рівномірним. Причому ступінь цієї рівномірності повинна бути вкрай високою [1].

Послідовності, що використовуються для потокового шифрування називають шифруючими гамами. Для їх створення використовують програмні генератори псевдовипадкових чисел (ПВЧ), які запускаються ключовою послідовністю. Загальну схему потокового шифрування даних наведено на

рисунку 1.1.

Генератор шифруючої гами створює потік псевдовипадкових символів g_i, що залежать від введеної користувачем ключової послідовності k_i. Під час шифрування ця послідовність складається побітно по модулю два з символами відкритого повідомлення x_i, що надходять із джерела даних.

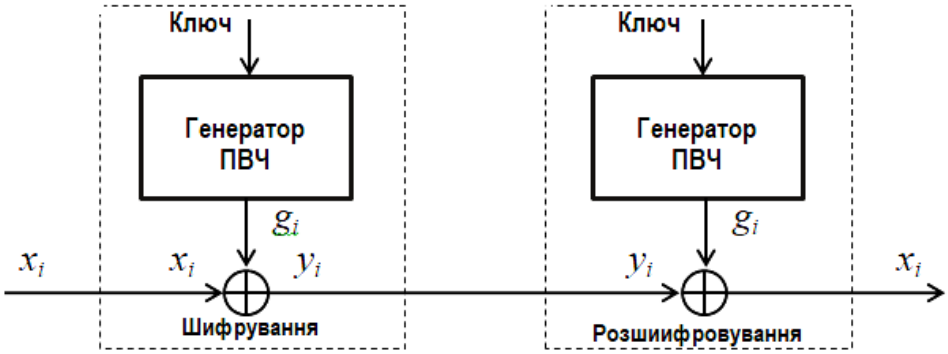

Рисунок 1.1 – Загальна схема потокового шифрування даних

$$y_i = x_i \oplus g_i, \quad i = 1, 2, 3, \ldots \tag{1.1}$$

Щоб на приймальній стороні із зашифрованого тексту y_1, y_2, ... отримати відкрите повідомлення x_1, x_2, ... , необхідно відтворити таку саму шифруючу гаму g_1, g_2, ... і повторити гамування зашифрованого повідомлення

$$x_i = y_i \oplus g_i, \quad i = 1, 2, 3, \ldots \tag{1.2}$$

Такий спосіб розшифрування ґрунтується на тому, що бінарні операції додавання і віднімання співпадають, а значить, такий спосіб криптографічного перетворення є зворотнім.

Вхідне повідомлення x_1, x_2, ... та ключова послідовність g_1, g_2, ... уявляють собою незалежні потоки біт, а способи криптографічних перетворень відрізняються лише генераторами псевдовипадкової гами. Це означає, що безпека шифруючої системи залежить виключно від властивостей генератора ключового потоку, який уявляє собою рівномірну ПВЧ.

Сучасна наука використовує псевдовипадкові числа у самих різних сферах наукових досліджень – від методів математичної статистики та імітаційного

моделювання до криптографії. Від якості таких генераторів залежит і результат їх використання. Актуальною проблемою є створення такого генератора, щоб розподілення чисел на його виході максимально наближалось до рівномірного. Що ж стосується криптографічних генераторів, то тут ці вимоги особливо високі.

Генератор ПВЧ створює послідовність бітів, лише схожу на випадкову. В реальності вона обчислюються за певними правилами і не є випадковою. Це дозволяє абсолютно точно відтворити її на протилежній стороні каналу зв'язку, знаючи ключ шифрування.

Послідовність створеної шифруючої гами повинна мати великий період повторення T_n і має бути такою, щоб її аналіз криптоаналітиком не давав абсолютно ніякої апріорної інформації про спосіб її створення.

Побітне потокове шифрування, з точки зору програмної реалізації, є не зовсім зручним. Але сучасні мови програмування, такі, як, наприклад, C++ або C#, дозволяють виконувати одночасне складання по модулю два символів цілого байту, а це дозволяє просто вирішити цю проблему. У відповідності до принципів, сформованих Райнером Рюппелем [2], можна виділити чотири основних підходи проектування симетричних потокових шифрів:

– системно-теоритичний підхід, оснований на використанні складного криптографічного алгоритму формування ПВЧ, що на даний момент не досліджений;

– складно-теоритичний підхід, оснований на використанні, відомого на даний момент, складного криптографічного алгоритму формування ПВЧ;

– інформаційно-технічний підхід, оснований на спробі приховати від крипоаналітика і не можливості найти однозначне рішення;

– рандомізований підхід, оснований на створенні об'ємної задачі, що не може бути вирішена комп'ютером за прийнятний час.

Для поточних шифрувальних систем було сформовано наступні теоретичні критерії:

– великі періоди повторення псевдовипадкових чисел;

– велика лінійна складність;

– дифузія – розсіювання надлишковості у підструктурах;

– кожен біт шифруючої гами повинен бути складним перетворенням ключа;

– критерій нелінійності для логічних функцій.

Додатково, спроектований криптографічний алгоритм повинен забезпечувати високу швидкодію шифрування, і, крім того, він має бути побудований на простих, з точки зору реалізації, математичних перетвореннях.

Зазвичай, криптографічний алгоритм потокового шифрування може включати не один, а декілька генераторів, що утворюють шуфруючу гаму, як композицію кількох послідовностей, отриманих від різних генераторів ПВЧ.

Потокові шифри використовують в системах з невеликими обмеженими обчислювальними ресурсами, де актуальними є швидкість шифрування.

1.2 Математична модель поточного шифрування

Перший поточний шифр був запропонований Гілбертом Вернамом у 1917 році. Він побудував електромеханічну машину. Яка автоматично шифрувала текст, що надходив від телеграфного апарату. Це був перший випадок автоматизації передавання і шифрування в одній машині. В основу її роботи було закладено, описаний вище алгоритм простого гамування [4].

У 1949 році Клодом Шеноном було математично обґрунтовано, що, якщо у машині Вернама, використати абсолютно випадкову унікальну шифрувальну гаму, довжина якої співпадає з довжиною тексту і яка використовується не більше одного разу, то такий шифр зламати неможливо [5]. Правда тут виникає проблема з необхідністю мати велику кількість ключів на кожній стороні каналу зв’язку, але це плата за абсолютну стійкість. Саме це обумовлює популярність поточних шифрів, а також те, що вони не вимагають використання додаткової внутрішньої пам’яті.

Що стосується поточного шифру з абсолютною стійкістю, то вимоги до нього Шенона можна сформулювати так:

– кількість інформації у шифруючій гамі повинна бути не меншою. Ніж у відкритому тексті;

– шифруюча гама використовується лише один раз;

– шифруюча гама отримується від природнього датчика випадкових чисел.

На початку минулого сторіччя такі гами зберігались в одноразових блокнотах і, через це, шифрування такого типу так і називають принципом одноразового блокноту (ОБ). Принцип одноразового блокноту полягає в тому, що у такому блокноті міститься одна велика шифрувальна гама, з якої з початку “відрізається” необхідний фрагмент, такий, що за розміром співпадає з довжиною відкритого тексту. Після виконання шифрування використаний фрагмент видаляється з блокноту і більше не використовується. На приймальній стороні є такий самий шифрувальний блокнот і процедура шифрування використовується у зворотному порядку. Принцип одноразового блокноту полягає в тому, що ключем є вся шифруючи гама (рисунок 1.2) До цієї пори такі шифри використовують дипломатичні та військові організації.

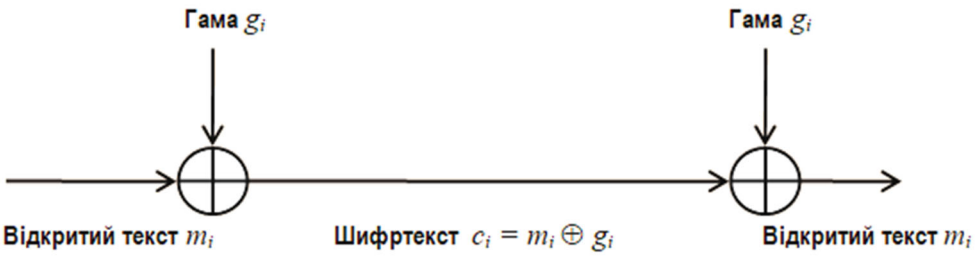

Рисунок 1.2 – Загальна схема потокового шифрування даних

З математичної точки зору, принцип поточного шифрування описується виразами (1) і (2), і пояснюється рисунком 1. З наведеного опису видно, потік вхідної послідовності шифрується послідовно, біт за бітом, що досягається складання по модулю два символів відкритого тексту з відповідними символами шифруючої гами. Сучасні поточні шифри відрізняються від одноразових блокнотів тим, що шифруючи гама не є одноразовою. Вона формується генератором ПВЧ на основі ключа і може бути зміненою шляхом зміни цього ключа. Тобто, доки шифрування повідомлень відбувається з одним ключем, шифруюча гама залишається не змінною. Але це не означає, що стійкість шифру буде низькою. Просто він перестане бути абсолютно стійким.

Абсолютно стійкі шифри в сучасних розподілених комп’ютерних системах не використовують через складність реалізації розподілення шифрувальних блокнотів. Проблема криптографічної стійкості вирішується шляхом організації регулярної централізованої зміни ключів для абонентів автоматизованої системи і, головне, за рахунок розроблення алгоритмів формування псевдовипадкових послідовностей, які майже не відрізняються від реально випадкових рівномірно розподілених чисел. Саме остання задача і є головним напрямом наукового пошуку ефективних алгоритмів шифрування.

Для потокових шифрів безпека зашифрованого текста визначається секретністю ключової гами, до якої поставлені наступні вимоги. Перша полягає в тому, щоб на основі аналізу її попередніх значень неможливо було б обчислити її наступні значення, за прийнятний для зловмисника термін. Друга – в тому, щоб канал. По якому передаються ключі до системи шифрування був абсолютно секретним, що означає, що тільки сторони інформаційного обміну мають доступ до ключів.

Іншими словами. перша вимога зводиться до того, що будь-які прогнози до наступних значень шифрувальної гами не кращі за звичайне вгадування. Друга вимога означає те, що має бути створена достатньо надійна система розподілу ключів між усіма парами суб’єктів системи інформаційного обміну.

Нажаль, розробники криптографічних систем зосереджують свої зусилля на виконанні першої вимоги. Вони намагаються створювати ефективні високонадійні генератори ПВЧ, стійкі до сучасних методів крипто аналізу. Але, як показує досвід, найбільш успішні атаки на симетричні криптосистеми пов’язані зі зламом системи зберігання і розподілення ключів.

Потокові шифри розділяють на синхронні та асинхронні.

У синхронних шифрах вихідний (зашифрований) потік символів залежить тільки від шифруючої гами і не залежить від попередніх символів зашифрованого тексту, а в асинхронних, у створенні шифруючої гами приймає участь ще й зашифрований текст, який подається на генератор ПВЧ через зворотний зв’язок (рисунок 1.3).

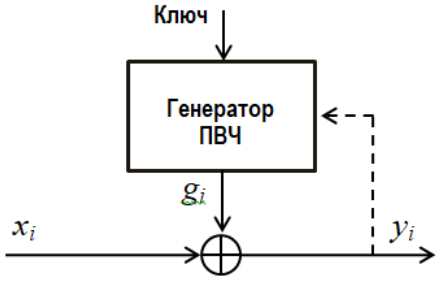

Рисунок 1.3 – Синхронні та асинхронні потокові шифри

Зворотний зв'язок, за допомогою якого забезпечується синхронний режим поточного шифрування називають режимом зворотного зв'язку по шифртексту (Cipher Feedback Mode – CFB).

Додатковою проблемою синхронного шифрування є те, що будь-яка помилка у вихідному потоці приводить до розмноження помилок у подальшому шифруванні і неможливості відновлення відкритого тексту на приймальній стороні. Виходячи із цього, переважна більшість потокових шифрів працює в синхронному режимі і використовуються в системах з невеликими обмеженими обчислювальними ресурсами, де актуальними є швидкість шифрування.

З іншого боку, активні атаки на синхронні шифри завжди приводять до втрати синхронізації, що повідомляє приймальній стороні про атаку на повідомлення.

Із сказаного витікає, що, все ж таки, головною частиною криптографічної системи є генератор шифрувальної гами. Зазвичай, він уявляє собою один або сукупність декількох кінцевих автоматів. Прикладом такого автомату може бути регістр зсуву з лінійним зворотним зв'язком та нелінійним фільтром.

Типовим прикладом поточного синхронного шифру є шифр, який використовує у якості генератора гами лінійний конгруентний генератор (ЛКГ) [6]. Лінійний конгруентний метод формування ПВЧ використовується у програмуванні для отримання випадкових чисел. Такий генератор уявляє собою кінцевий автомат, внутрішній поточний стан якого визначається виразом

$$x_{i+1} = (a * x_i + b) \bmod m, \tag{1.3}$$

де m – модуль ($0 < m$);

a – множник ($0 < a < m$);

b – додаток ($0 < b < m$);

mod (x, u) – операція, що повертає залишок від ділення першого аргументу на другого.

В цьому типі автомату поточний внутрішній стан визначається виключно його попереднім станом. Вихідна послідовність на виході такого автомату називається лінійною конгруентною послідовністю (ЛКП). Через конечність автомату вона буде циклічною з власним періодом повторення.

Параметри такого автомату обираються наступним способом. По-перше, число m має бути досить великим, так як період не може мати більше m елементів. Беручи до уваги той факт, що ділення – це ресурсно затратна операція, величина m має обиратись рівною 2^N, де N – це розрядність регістру мікропроцесора. В такому разі визначення залишку від ділення зводиться до подвійної логічної операції «AND». Іноді при $b = 0$, лінійну конгруентну послідовність називають мультиплікативним конгруентним методом, а при $b \neq 0$ – змішаним конгруентним методом. Довжина періоду залежить від величин m, a і b.

У потоковому шифруванні розглядається підклас так званих самосінхронізуючихся шифрів, які уявляють собою кінцеві автомати. Такий автомат f, який управляється секретним ключем k, вводить поточний стан

$$\delta_{i+1} = f(k, \delta_i). \tag{1.4}$$

Нелінійний фільтр F виводить кожний поточний стан автомату $k_i = F(\delta_i)$ у вихідний потік шифруючої гами g_i так, як це показано на рисунку 1.4.

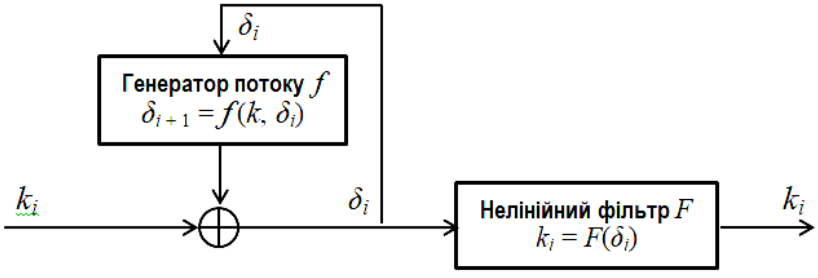

Рисунок 1.4 – Принцип роботи генератора ключового потоку

Враховуючи, виходи кожного кінцевого автомату є періодичними, будь-які генератори ключових потоків можуть бути реалізованими за допомогою лінійних регістрів зі зворотним зв’язком (Linear feedback shift register – LFSR).

Для самосінхронізучогося шифру повинен бути найденим хороший нелінійний фільтр F. Генератори на основі LFSR легко піддаються аналізу і є дуже популярними у поточному шифруванні. Конструкція такого шифру має відповідати наступним вимогам:

– забезпечення заданих статистичних властивостей періодичних шифруючої гами;

– забезпечення більшого періоду шифруючої гами у порівнянні з іншими варіантами нелінійних фільтрів;

– забезпечення максимальної складності ключового потоку;

– забезпечення простоти програмної комп’ютерної реалізації;

– забезпечення простоти обчислення з використанням обмежених ресурсів.

Таким чином, основна ідея під час проектування генераторів ключового потоку полягає в тому, щоб зробити його непередбаченим.

Загальні вимоги до симетричних поточних шифрів можна сформулювати так:

– стійкість шифру по відношенню до сучасного криптоаналізу має бути така, щоб зламати його можна було тільки повним перебором ключів;

– криптографічна стійкість алгоритму шифрування має визначатись тільки змістом ключа;

– об’єм зашифрованого тексту не повинен значно перевищувати величину відкритого тексту;

– помилки, що виникають під час шифрування не повинні викликати спотворення відкритого тексту;

– час, що витрачається на шифрування, має бути прийнятним;

– вартість шифрування має відповідати важливості даних, що підлягають захисту.

1.3 Вибір інструментальних засобів реалізації поточного шифру

Комп’ютерна реалізація алгоритмів шифрування передбачає подолання двох наступних проблем. Перша полягає в тому, що обчислювальні пристрої – це детерміновані автомати і вони не здатні генерувати випадкові числа. Вбудовані в різноманітні середовища програмувань датчики випадкових чисел утворюють так звані “псевдовипадкові” числа, які утворюються як результат виконання рекурентних процедур, де вигляд вихідної послідовності цілком визначається першим вхідним числом x_0 або групою початкових чисел x_0, x_1, x_2, … На даний момент існує багато варіантів програмних генераторів ПВЧ, які відрізняються законом розподілення вихідних чисел, періодом повторення T і, головне, кількістю обчислювальних ресурсів, необхідних для їх програмної реалізації.

У середовищах програмування більшості мов у якості датчика ПВЧ використовують лінійний конгруентний генератор, робота якого описується виразом (1.3). Такий генератор, нажаль дає послідовність псевдовипадкових чисел, розподілення яких, хоча і називають рівномірним, втім, якість цієї рівномірності далеко не відповідає вимогам, що висуваються до криптографічних генераторів. Це означає, що створення сучасного поточного шифру повинно починатись з розроблення власного генератора, виходячи із поставлених вимог.

Слід зауважити, що не зважаючи на те, що створена шифруючим пристроєм ключова послідовність детермінована, для зовнішнього спостерігача, який не знає ключа до шифру, вона має вигляд випадкової. Щоб така послідовність була придатною для криптографічних перетворень, необхідно, щоб для стороннього крипто аналітика вона виглядала як випадкова. Це означає, що що кожен наступний символ шифруючої гами повинен вгадуватись не з ймовірністю, що не відрізняється від величини 0,5, незалежно від результатів аналізу попередніх її символів.

Щоб обраний генератор псевдовипадкової послідовності чисел відповідав поставленим вимогам, він повинен формувати послідовність, період повторення якої має бути достатньо великим. Так, наприклад, лінійний конгруентний

генератор, має період повторення, обмежений величиною $2^m - 1$, де m = N, а N – визначається розрядністю мікропроцесора. Зазвичай, датчики, вбудовані у середовище програмування різних мов, мають N = 32768.

Переважна більшість відомих елементарних генераторів ПВЧ не використовують напряму для формування шифруючої гами. Вони добре вивчені, їх параметри можна знайти у відповідній літературі і вони входять до складу криптографічних алгоритмів у якості складових частин. Такими генераторами є описані лінійні конгруентні генератори, генератори, побудовані на основі чисел Фібоначчі, а також генератори Хаффмана, де використовуються регістри зі зворотним зв'язком.

Визначити стійкість побудованого генератора аналітичним способом неможливо. Для оцінки його ефективності використовують статистичні методи, що уявляють собою набори програмно організованих тестів [7]. Якщо тести виконуються, генератор вважають придатним, якщо ні, – його удосконалюють.

Задача криптоаналітика зводиться до перебору можливих варіантів застосовуваного генератора. Такий генератор повинен уявляти собою композицію елементарних генераторів, яка забезпечує максимально можливу, довжину періоду послідовності рівномірного розподілених символів.

Друга проблема створення алгоритму потокового шифрування полягає у створенні генератора ПВЧ, економічного з точки зору обчислювальних операцій. Як відомо, елементарними операціями, що виконує мікропроцесор, є складання бінарних чисел, їх порівняння та зсув в регістрі (вліво або вправо). Такі операції використовуються за один такт роботи мікропроцесора. Що ж стосується таких операцій як множення, ділення, підведення у ступінь та приведення по модулю, то вони зводяться до багатократного виконання елементарних операцій і є дуже затратними. У процесі створення алгоритму шифрування слід намагатись, по можливості, використовувати саме елементарні операції.

Що стосується вибору мови і середовища програмування, то це має бути мова, що найближче стоїть до асемблеру. Такою мовою є мова об'єктно-орієнтованого програмування C++, яка, у разі необхідності, дозволяє робити

асемблерні вставки, що дозволяє створювати компактний економічний код.

Зважаючи на те, що для зручного використання програмного продукту, який відповідає сучасним вимогам, необхідно створити графічний інтерфейс. Для створення такого інтерфейсу найбільше підходить таке середовище як MS Visual Studio. Воно дозволяє вирішувати обчислювальні задачі в широкому спектрі предметних областей різними мовами в тому числі і C++. Сьогодні це одне з небагатьох середовищ програмування, яке дозволяє якісно вирішувати графічні задачі.

1.4 Висновки за розділом 1

Сучасні поточні шифри, що використовуються для захисту даних, які передаються в розподілених комп'ютерних мережах, повинні забезпечувати заданий рівень безпеки. В той же час, вони мають бути ефективними з точки зору програмної реалізації, тобто забезпечувати достатню швидкодію при мінімальних ресурсних витратах.

В основу обраного типу генератора псевдовипадкових чисел по можливості повинні бути закладені елементарні операції на виконання яких витрачається мінімум часу і оперативної пам'яті.

Головною вимогою, що ставиться до генератора псевдовипадкових чисел, є рівномірність розподілення ймовірностей бінарних символів у складі вихідної послідовності.

Для програмної реалізації потокового шифру найкраще підходить мова C++, яка забезпечує економічну реалізацію математичних операцій, а у якості програмного середовища підходить середовище Visual Studio, через те, що воно містить ефективні засоби для створення графічного інтерфейсу.

2 ОБГРУНТУВАННЯ ВИБОРУ АЛГОРИТМУ ПОТОЧНОГО ШИФРУВАННЯ

2.1 Теоретичні підходи до вибору алгоритму поточного шифрування

Як було показано у попередньому розділі, ідеальним варіантом поточного шифрування є система з одноразовим блокнотом, запропонована Гілбертом Вернамом і вимоги до неї були сформульовані Клодом Шеноном. Безпека поточного шифрування такої системи повністю визначається властивостями шифрувальної гами g_0, g_1, … , а саме, рівномірністю розподілення ймовірностей символів у її складі. Це може бути математично обґрунтоване наступним чином.

Будемо вважати, що у каналі зв'язку криптограми передаються бінарними послідовностями. Нехай криптоаналітику відомо, що у відкритому тексті символ 0 передається із імовірністю p, а протилежний до нього символ 1 з імовірністю 1 – p. Відповідно, при спробі вгадати їх значення у перехоплених криптограмах, криптоаналітик буде мати успіх з імовірністю

$$P(p) = p * p + (1 - p)(1 - p). \quad (2.1)$$

Ця функція досягає свого мінімуму при p = 0,5. Якщо символи шифруючої гами g_0, g_1, … розподілені рівномірно, тобто $p(1) = p(0) = 0{,}5$, то і у складі криптограми вони теж будуть зустрічатися з однаковою імовірністю. З урахуванням (2.1), маємо

$$P(p) = 0{,}5 * p + 0{,}5 * (1 - p) = 0{,}5. \quad (2.2)$$

Таким чином, біти у криптограмі перетворюються на "білий шум". Це означає, що разі гамування, яке означає змішування символів відкритого тексту, незалежно від їх розподілення з символами рівномірно розподілених символів гами, у сумарній послідовності символи також будуть рівномірно розподілені.

Через складність вирішення проблеми розподілення ключі у разі використання одноразових блокнотів, розробники поточних шифрів

використовують гаму вальні послідовності, отримані від генераторів псевдовипадкових чисел. Побудувати ідеальний, з точки зору рівномірного розподілення символів, програмний генератор неможливо, але це зовсім не означає, що він обов'язково буде зламаний. Все залежить від інтелектуальних та обчислювальних ресурсів криптоаналітика. Якщо криптографічне перетворення передбачає виконання досить великої кількості складних операцій, у можливого зловмисника не вистачить можливостей для того щоб розшифрувати перехоплене повідомлення за розумний термін.

Із сказаного витікає, що сучасні криптографічні системи повинні будуватись з урахуванням можливостей імовірного супротивника, від якого захищається інформація, і розробники шифрів повинні їх враховувати.

В якості міри, за допомогою якої оцінюється складність обчислювального процесу під час дешифрування повідомлень, частіше за все використовують кількість елементарних операцій w деякого типу, що виконуються протягом часу дешифрування одного повідомлення, або визначення одного ключа. В кожному окремому випадку обговорюється, що саме означає термін "*елементарна операція*". Це може бути або сукупність операцій, що виконуються на конкретній апаратурі за один етап шифрування, або сукупність операцій, що виконується протягом часу однієї спроби підбору ключа, або щось інше. Головне – щоб вибір цієї міри був обґрунтований.

Складність процедур дешифрування залежить від кількості апріорної інформації, що є в розпорядженні у криптоаналітика. Найскладнішою вважається ситуація, коли крім перехопленої криптограми довжиною в N символів у нього нічого не має. Такий спосіб атаки на шифр називають "аналізом на основі перехоплених криптограм".

В цій ситуації, єдиним способом дешифрування криптограми є перевірка всіх можливих значень ключа. Допускаючи, що його довжина дорівнює довжині повідомлення, повна кількість ключів які треба буде перебрати складає величину 2^N. Цю величину називають потужністю ключового простору.

Підвищення потужності ключового простору, звичайно, збільшує

криптографічну стійкість шифру, але не завжди і необов'язково. Наприклад, для шифру простої заміни потужність ключового простору дорівнює величині $m!$, де m це кількість символів у алфавіті. Хоча при досить великій довжині тексту потужність ключового простору буде непомірно великою, шифри простої заміни ламаються статистичними методами досить просто.

На сьогоднішній день, нажаль, не існує універсальних методів, які можна було б застосувати для криптографічного аналізу будь-яких шифрів. Кожен з відомих шифрів потребує розробки індивідуального способу аналізу і дешифрування.

Під методом аналізу шифру розуміють сукупність правил та дій, що сформульовані на основі дослідження конкретного шифру і використання яких дає змогу виконати дешифрування криптограми з імовірністю, що відрізняється від нуля.

Ймовірність відгадування ключа до шифру, яка відрізняється від нуля означає, що, якщо ключ обирається не з усієї з множини K, а тільки з деякої її частини $K_l \subseteq K$, яка сформована випадковим вибором із K, імовірність дешифрування співпадає з імовірністю попадання ключа, за допомогою якого шифрується повідомлення, в множину K_l.

Під стійкістю шифру розуміють кількість операцій, що мають бути виконаними при умові використання найбільш ефективного алгоритму його крипто аналізу та найбільш потужного обчислювача.

Крім терміну стійкість, часто використовують такий термін як практична стійкість, який означає важкість реалізації найшвидшого з відомих алгоритмів, що відповідає найбільш ефективному з відомих методів на найшвидшому з доступних обчислювачів.

У процесі розроблення поточного шифру треба враховувати можливості, які має криптоаналітик супротивника. Побудова шифрів, стійкість яких значно перевершую необхідну не оправдана, оскільки вона досягається збільшенням часу, що витрачається на виконання процедур шифрування та розшифровування, а також використанням занадто великих обчислювальних ресурсів.

Переважна більшість генераторів псевдовипадкових чисел, які використовують для криптографічних потреб, уявляють собою програмну реалізацію функції від двох змінних $f(s,i)$, де i – номер поточного випадкового бінарного слова фіксованої довжини, а s – параметр або секретний ключ, з якого починається формування шифруючої гами. Ключ до шифру повинен мати випадкову природу, а потужність множини можливих ключів має бути достатньо великою.

Створена псевдовипадкова послідовність чисел розглядається як послідовність бінарних слів

$$f(s,1), f(s,2), f(s,3), \dots, f(s,k),$$

де k – необхідна довжина послідовності.

До генераторів ПВП ставляться наступні вимоги. Генератор ПВЧ повинен забезпечити, щоб:

– створена псевдовипадкова послідовність статистично виглядала як абсолютно випадкова;

– знання початкової частини ПВП (наприклад, $F(s, 1)$, $F(s, 2)$, $F(s, 3)$, ... , $F(s, k - 1)$), не повинно дозволяти передбачити наступний біт цієї послідовності з ймовірністю, яка відрізняється від величини 0,5.

Навіть при великому значенні s, такі вимоги не можливо задовольнити якщо не буде обмежена складність тестів та методів прогнозу.

В сучасній криптографії використовують тільки такі алгоритми формування ПВЧ, у яких довжина вихідної послідовності ПВЧ значно більше довжини ключа.

Будемо вважати, що послідовність бітів, породжена генератором ПВЧ z_1, z_2, ... , z_k, має довжину k, що більше довжини ключового слова s. В такому разі, для передбачення наступного символу z_{k+1} можна почергово подавати на вхід генератора всі можливі значення ключа s і генерувати перші k біт.

У разі спів падіння деякого значення u зі значенням s, буде отримана така сама послідовність z_1, z_2, ... , z_k, можна зробити висновок, що значення ключа знайдено. Це означає, що так само можна визначити наступні символи

шифруючої гами.

Метод прямого перебору ключів дозволяє передбачити наступний біт послідовності, породженої генератором псевдовипадкових чисел, і, це означає, що вона не є абсолютно випадковою. Термін прямого перебору ключів пропорційний величині $2^{|s|}$, де, $|s|$ – довжина ключового слова s. Вважається, що величина $|s|$ повинна бути не менше кількох сотень біт.

До генераторів псевдовипадкових чисел висуваються наступні вимоги:

– створена псевдовипадкова послідовність повинна статистично відрізнятись від абсолютно випадкової послідовності за прийнятний термін обчислень;

– наявність даних про знання якої-небудь початкової частини послідовності не повинно дозволяти передбачати наступний біт цієї послідовності за прийнятний термін обчислень.

Переважна більшість відомих генераторів ПВЧ відповідає переліченим умовам [8].

2.2 Програмні способи формування потоку псевдовипадкових чисел

Враховуючи той факт, що при симетричному шифруванні необхідно на обох сторонах каналу зв’язку формувати однакові шифруючи гами, для криптографічних потреб використовують алгоритмічні генератори, на виході яких кожен із вихідних символів є результатом обчислень над попереднім вихідним символом

$$g_{i+1} = f(g_i).$$

Одним із недоліків генераторів такого типу є можливість утворення “петлі”. Петля – це короткий внутрішній цикл, який може виникати у разі неправильного підбору параметрів генератора. У більшості таких випадків петля складається лише з двох чисел. Це явище, у більшості випадків, виникає в алгоритмах елементарних генераторв, до складу яких можна, в першу чергу, віднести лінійний конгруентний генератор та генератор Фібоначчі з запізненням [6].

Першими, розробленими математиками, методами отримання

псевдовипадкових чисел, закладеними в основу елементарних генераторів були:

– метод серединних квадратів;

– метод серединних добутків;

– метод перемішування;

– лінійний конгруентний метод.

Всі вони були добре описані Доналдом Кнутом в роботі [6], де було показана їх неефективність для потреб шифрування через низьку рівномірність розподілення сформованих ними вихідних чисел. Саме з цих причин вони можут використовуватись в шифрувальних алгоритмах лише у вигляді складових частин.

Щоб створити генератор псевдовипадкової гами, ми маємо взяти до уваги, що він, хоча і не може розглядатись, як абсолютно стійкий, він все ж таки повинен забезпечувати заданий рівень криптографічної стійкості.

Будемо вважати обчислювальну систему обчислювально безпечною, якщо для її злому необхідно виконання, що найменш t операций. Причому повинно бути достатнім, щоб злам генератора ПВЧ крипто аналітиком супротивника неможливо було виконати за прийнятний термін.

Для вирішення цієї задачі слід замінити абсолютно стійкий генератор на періодичний генератор ПВЧ з великим періодом повторення, вихідний потік якого утворюється із початкового числа, у якості якого використовується ключова комбінація, як це показано на рисунку 2.1.

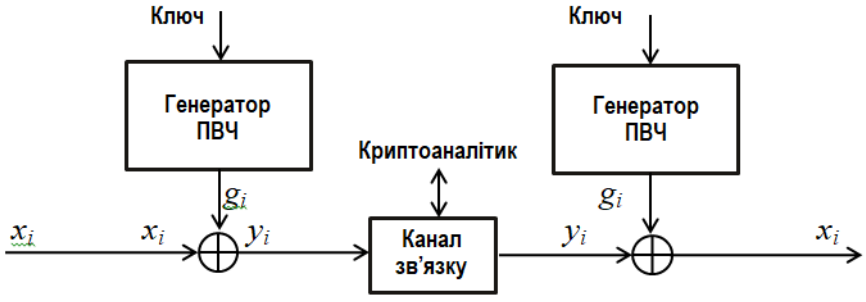

Рисунок 2.1 – Принцип роботи поточного шифрування

Будемо також вважати, що крипто аналітик супротивника намагається отримати ключ до шифру простим перебором всього простору можливих ключів.

Як, вже було сказано, практично всі генератори розроблені не для

криптографічних цілей, не є криптографічно безпечними, і для поточного шифру слід або створити свій власний генератор, або скористатись одним або комбінацією відомих криптографічних генераторів.

Прикладом, генератора такого типу, що використовує прості операції, які легко програмно організувати є генератор, побудований на регістрі зсуву зі зворотним зв'язком (Linear feedback shift register – LFSR).

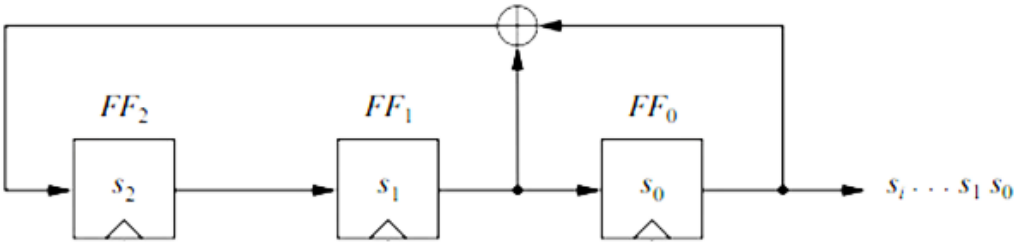

Рисунок 2.2 – Регістр зсуву з лінійним зворотним зв'язком 3-го ступіню і початковими величинами s_2, s_1, s_0

Такий генератор складається з бінарних елементів пам'яті (тригерів) і тракту зворотного зв'язку. На рисунку 2.2 показаний приклад реалізації LFSR генератора третього ступіню (m = 3). Ступінь генератора визначається кількістю тригерів (FF_2, FF_1, FF_0), що входять до його складу. Зворотний зв'язок визначається способом формування вхідного символу, що подається на старший тригер. В наведеному прикладі кожний вхідний символ визначається як сума по модулю два символів з виходів тригерів FF_0 і FF_1. Якщо у тригер FF_2 записати одиницю, а в FF_0 і FF_1 – нулі то отримаємо результат зсуву, наведений у таблиці 2.1.

Таблиця 2.1 – Послідовність перетворень у розрядах LFSR

№ п\п	FF_2	FF_1	$FF_0 = s_i$
0	1	0	0
1	0	1	0
2	1	0	1
3	1	1	0
4	1	1	1
5	0	1	1
6	0	0	1
7	0	1	0

В наведеному прикладі для реалізації зворотного зв’язку використовується економічна операція складання по модулю два (XOR).

В загальному вигляді LFSR-генератор показаний на рисунку 2.3.

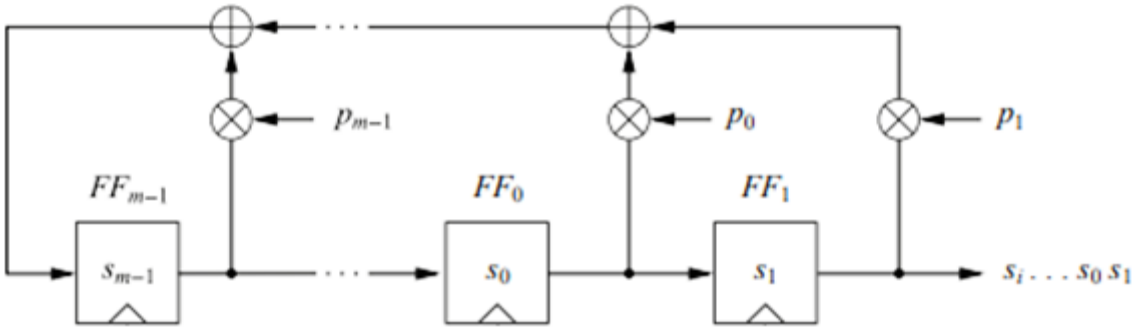

Рисунок 2.3 – LFSR-генератор з коефіцієнтами зворотного зв’язку p_i і початковими значеннями пам’яті s_{m-1}, … , s_0

Такий генератор містить m можливих тригерів та m можливих елементів зворотного зв’язку p_i, об’єднаних операцією XOR. Активація конкретного єлемента зв’язку визначається елементом p_0, p_1, ... , p_{m-1}. Якщо $p_i = 1$ (замкнутий перемикач) – зворотний зв’язок активний, якщо $p_i = 0$ (розімкнутий перемикач) – зворотний зв’язок не приймає участі у формування вхідного символу.

Описаний спосіб формального опису форми зворотного зв’язку є зручним і просто задає вигляд генератора ПВП. Саме вибір коефіцієнтів послідовності p_i та ступінь m, визначають криптографічну стійкість LFSR-генератора.

В наведеному вище прикладі LFSR-генератора третього ступіню ($m = 3$), біти внутрішнього стану s_i зміщуються на одиницю вправо на кожному такті роботи обчислювача. Крайній правий біт внутрішнього стану є вихідним бітом генератора. Він, разом з іншими попередніми бітами, безпосередньо впливає на значення всіх інших поточних вихідних символів.

Оскільки XOR – це лінійна операція, LFSR-генератори називають лінійними регістрами зсуву зі зворотним зв’язком. Якщо продовжувати лінійний зсув у такому регістрі, то в таблиці 2.1 буде записано увесь цикл повторення вихідної гами (останній стовпець таблиці). Далі послідовність буде повторюватись циклічно, а довжина періоду повторення дорівнює $2^3 – 1 = 7$ бінарних розрядів.

0010111 0010111 0010111 ...

В в загальному випадку $T = 2^m - 1$. Тут треба звернути увагу на той факт, що комбінація, яка утворюється з усіх нулів в тригерах регістру має бути відсутня, бо це приведе до нульового зациклення генератора. Із сказаного витікає що для того, щоб отримати великий період повторення вихідної послідовності, m слід обирати якомога більшим.

Далеко не всі комбінації коефіцієнтів дають максимально можливу довжину періоду T.

Для формального описання LFSR використовують поліноми Хафмана. Наприклад, поліном для описаного вище прикладу, має вигляд

$$P(x) = x^{m-1}p_{m-1} + x^1p_1 \ldots + p_0 = x^3 + x + 1.$$

Із теорії кодування відомо, що максимальну довжину періоду забезпечують примітивні (неприводимі) поліноми, тобто такі поліноми, які не розкладуються на простіші поліноми. У відповідній літературі можна знайти таблиці [1], у яких містяться поліноми вже досліджених LFSR-генераторів. Наприклад, m = 31, існує 69273666 різних примітивних поліномів, які забезпечують періоди різної довжини.

Проектування потокових шифрів на базі LFSR передбачає вибір одного або групи генераторів з різною довжиною періоду повторення та різними поліномами, що утворюють зворотний зв’язок. Якщо довжини їх періодів взаємно прості, а поліноми примітивні, то загальний період повторення буде максимально можливим. Вихідний біт комбінованого генератора буде функцією окремих розрядів LFSR. Її називають комбінаційною функцією.

У спеціальній літературі шифри на основі комбінованих генераторів описані дуже добре [9], але вони розглядаються більше з теоретичної точки зору і розроблювались для апаратної реалізації в докомп’ютерну епоху з виконанням операцій XOR, AND, OR, та NOT. Їх програмна реалізація викликає деякі труднощі через необхідність виконувати бітові операції, зміст яких визначається виключно виглядом утворюючих поліномів.

Майже одночасно зі створенням практично діючої доступної обчислювальної техніки її почали використовувати для потреб криптографії та статистичного моделювання, а це, у свою чергу, поставило проблему створення якісних генераторів ПВЧ, послідовності на виході яких, не відрізнялись від реальних випадкових процесів з рівномірним законом розподілення на виході. Оскільки лінійні конгруентні генератори дають вихідну послідовність, у якій кожний наступний символ залежить лише від попереднього символу, якість його невелика. Це наштовхнуло розробників на думку про необхідність його ускладнення.

Відомо [6], що період ЛКП максимальний, коли модуль m приблизно дорівнює величині кодового слова процесора. В такому разі його величина, зазвичай, перевищує 10^9, що для практичних задач перевищує необхідний об'єм випадкових чисел. Крім того, довжина періоду ПВП помітно впливає на ступінь "випадковості" чисел у її складі.

Одним із способів збільшення періоду полягає в тому, щоб зробити поточне вихідне число X_{n+1}, залежним від декількох попередніх чисел. Наприклад, від X_n та X_{n-1}. З урахуванням того, що вихідна послідовність не може повторюватись до тих пір, поки не виконається умова

$$(X_{n+\lambda} * X_{n+\lambda+1}) = (X_n * X_{n+1}).$$

Найпростіша послідовність такого типу – це послідовність Фібоначчі.

$$X_{n+1} = (X_n + X_{n-1}) \bmod m. \qquad (2.3)$$

Такий генератор (рисунок 2.4) було реалізовано і досліджено ще в минулому сторіччі, але, не дивлячись на те, що його період перевищував величину m, числа на його виході були недостатньо випадкові.

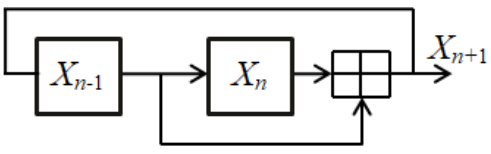

Рисунок 2.4 – Найпростіший генератор Фібоначчі

Так само, досліджувався генератор вигляду (рисунок 2.5)

$$X_{n+1} = (X_n + X_{n-k}) \bmod m. \quad (2.4)$$

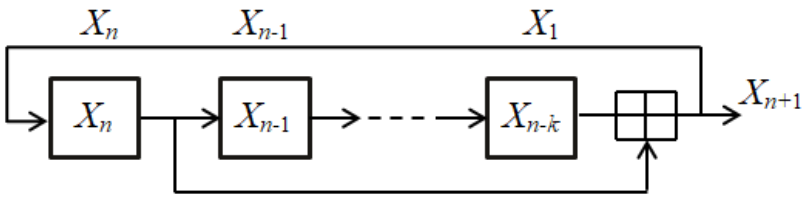

Рисунок 2.5 – Генератор Фібоначчі виду = ($X_{n+1} = X_n + X_{n-k}$) mod m

Якщо k – достатньо велике число, такий генератор ПВЧ дає великий період, такий, що перевищує m. Пізніше, експериментально було підібрано найбільш підходящі величини k_1 і k_2. Так генератор вигляду

$$X_n = (X_{n-24} + X_{n-55}) \bmod m, \quad n \geq 55, \quad (2.5)$$

Дав непогані, з точки зору рівномірності розподілення вихідних символів, результати. Його дослідження показали, що він дає найдовший, у порівнянні з іншими генераторами такого типу, період, швидко працює і його можна використовувати для генерації чисел з плаваючою точкою.

Числа 24 і 55, зазвичай, називають “запізненням”, а числа з виходу генератора – “послідовністю Фібоначчі з запізненням”. Оптимальні величини k_1 *і* k_2 приводяться в [6] і показані в таблиці 2.2.

Таблиця 2.2 – Оптимальні величини k_1 *і* k_2 *для генератора Фібоначчі з запізненням*

Оптимальні пари чисел k_1 *і* k_2			
24, 55	30, 127	37, 100	38, 89
83, 258	107, 378	273, 607	576, 3217
1029, 2281	4187, 9689	7083, 19937	9739, 23209

У 1984 Джордж Марсалья [10] запропонував генератор, що описується відношенням

$$X_n = (X_{n-24} * X_{n-55}) \bmod m, \quad n \geq 55, \quad (2.6)$$

де m кратно 4, а всі числа від X_0 до X_{54} непарні. В такому генераторі всі молодші розряди мають період повторення $2^{55} - 1$, в той час як старші бінарні розряди

перемішуються краще через те, що вони суттєво залежать від числових величин розрядів $X_{n\text{-}24}$ та $X_{n\text{-}55}$. Найкращий результат має місце, коли модуль m є простим числом. Вирішення проблеми такого генератора полягає у видаленні молодших розрядів з чисел вихідної послідовності, що суттєво знизить його продуктивність.

До 1991 року генератор, що описуються виразами (2.5) і (2.6), вважалися майже найкращих генераторів ПВЧ і використовувались здебільшого для потреб моделювання. Що ж стосується криптографічних засобів, то вимоги до них значно виросли і, згодом, було запропоновано більш досконалі алгоритми, складність яких, нажаль, значно перевищує наведені вище варіанти утворення випадкових чисел.

2.3 Обґрунтування вибору генератора псевдовипадкових чисел

Розглянуті у попередньому розділі LFSR-генератори та генератори Фібоначчі з запізненням, нажаль, мають недоліки, що обмежують їх використання. LFSR-генератори розроблялись в докомп'ютерну епоху і призначались для апаратної реалізації, а генератори Фібоначчі з запізненням не забезпечують рівномірності розподілення ймовірності у молодших розрядах сформованих чисел.

У 2003 році вийшла робота Джоржа Марсальї [11] де було запропоновано алгоритм програмного формування псевдовипадкових чисел, який забезпечував рівномірність розподілення одиниць і нулів усього кодового машинного слова. Позитивною стороною запропонованого алгоритму було те, що в ньому використовувались економічні, з точки зору витрат обчислювальних ресурсів, операції: складання по модулю два (xor) та "зсув вліво" або "зсув вправо" (shift). Саме через це, такий генератор був названий Xorshift.

З моменту появи Xorshift-генератора було створено декілька варіантів його модифікації. У загальному вигляді цей генератор уявляє собою декілька лінійних регістрів зі зворотним зв'язком, які забезпечують ефективність генерації чисел без використання надмірно розріджених поліномів.

Робота Xorshift генераторів основана на використанні 32-бітового цілого числа, як елемента векторного простору у двійковому полі по модулю 2. Вектори

таких чисел складаються за допомогою операції xor.

Алгоритм Xorshift розглядає набір усіх ненульових бінарних векторів 1 × 32 на Z, а f як лінійне перетворення над Z, представлене невиродженою бінарною матрицею T розміру 32 × 32.

Для випадкового числа $y \in Z$, вихідна послідовність генератора описується як yT, yT^2, yT^3, … тоді і тільки тоді, коли порядок матриці T дорівнює $2^{32} – 1$ у групі не вироджених бінарних матриць розміром 32 × 32 і послідовність має період $2^{32} – 1$.

Марсалья показав, що простий швидкий спосіб формування матричного добутку yT можна реалізувати, якщо порядок

$$T = (I + L^a)(I + R^b)(I + L^c)\,, \tag{2.7}$$

де L – це матриця, яка впливає на лівий зсув на одиницю $y << 1$. Відповідно матриця L^a реалізує зсув $y << a$. Оскільки матриця R – це транспонована матриця L, її використання реалізує правий зсув на одиницю $y << 1$.

Під час запуску генератора задаються чотири початкові 32-х бітних числа x, y, z та w. Саме вони визначають внутрішній стан генератора. Кожне наступне число $w = \{N_0, N_1, , \ldots , N_{31}\}$ формується як комбінація розрядів чисел x та попереднім значенням w так, як це показано у наведеному фрагменті коду, після чого відбувається зсув чисел $y \rightarrow x$, $z \rightarrow y$ та $w \rightarrow z$. Саме ці зсуви і підвищують “випадковість” молодших розрядів чисел.

Фрагмент коду, що реалізує такий алгоритм, на рисунку 2.6, а спрощена його схема наведена на рисунку 2.7.

```
X = 123456789
Y = 362436069
Z = 521288629
W = 88675123
T = X ^ ((X << 11) & 0XFFFFFFFF);
X = Y;
Y = Z;
Z = W;
W = (W ^ (W >> 19)) ^ (T ^ (T >> 8));
```

Рисунок 2.6 – Фрагмент коду, що реалізує алгоритм генератора Xorshift

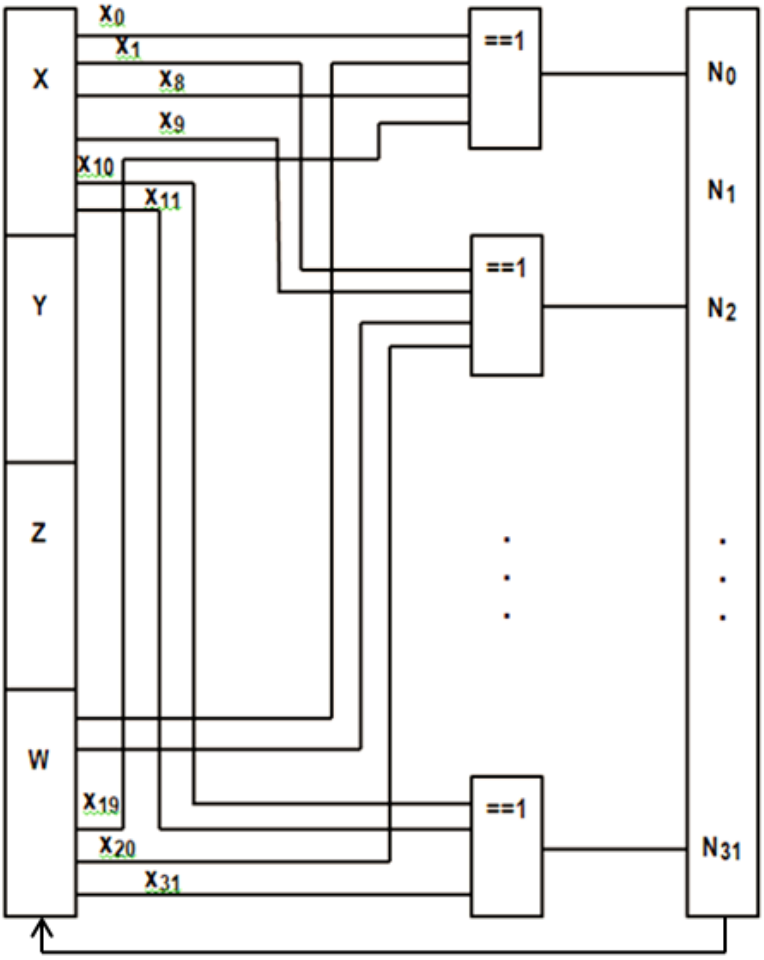

Рисунок 2.7 – Спрощена схема генератора Xorshift

У відповідності з цим алгоритмом, число утворюється зсув числа вправо на 11 позицій, складається з його попереднім значенням і до отриманого результату додається 32-х бітне число **`0xFFFFFFFF`**, в розрядах якого містяться всі одиниці.

Після виконання відповідних зсувів, вихідне число утворюється заскладним правилом

```
W = (W ^ (W >> 19)) ^ (T ^ (T >> 8)).
```

Потім воно видається на вихід генератора і перезаписується на звільнене числом місце. Далі процес циклічно повторюється до зупинки генератора.

Послідовності, що утворюються генератором Xorshift є найкращими з точки зору швидкодії і мінімуму ресурсів обчислювальної системи. Такі генератори успішно проходять тест BigCrush з пакету TestU01 [11].

Висновки за розділом 2

Із проведеного аналізу відомих програмних генераторів псевдовипадкових послідовностей для поточного шифрування найбільш придатними для формування шифрувальної гами є запропонований Джоржем Марсальєю Xorshift-генератор. Його переваги над іншими генераторами визначаються тим, що в основу його алгоритму закладені елементарні операції мікропроцесора, виконання яких відбувається за мінімальний час без зайвого використання пам’яті.

Додатковою перевагою Xorshift-генератора є успішне проходження тесту BigCrush з пакету TestU01, в той час, як окремі варіанти LFSR-генераторів, а також генераторів Фібоначчі з запізненням таких тестів не проходять і використовуються для криптографічних потреб лише у якості складових частин.

3. ПРОГРАМНА РЕАЛІЗАЦІЯ АЛГОРИТМУ ШИФРУВАННЯ

3.1 Вимоги до програмної реалізації поточного шифру

Враховуючи результати попереднього розділу, у якості генератора шифрувальної гами обираємо, запропонований Джорджем Марсальєю, Xorshift-генератор.

Програмна реалізація поточного шифру повинна забезпечувати ефективне шифрування комп'ютерних файлів будь-якого формату. Користувачеві готового програмного продукту має бути наданий зручний графічний інтерфейс, який дозволяє:

– вводити ключ до шифру;

– давати можливість приховувати символи введеного ключа до шифру;

– вказувати повний шлях до файлу, що має бути зашифрованим;

– вказувати повний шлях до зашифрованого файлу;

– контролювати виконання процесу шифрування.

У якості мови програмування обираємо мову C++, виходячи із того, що вона максимально наближена до мови асемблеру і дозволяє створювати економічний, з точки зору виконання арифметичних операцій, код, а у якості програмного середовища обираємо програмний пакет MS Visual Studio, який містить розвинені інструментальні графічні засоби для створення інтерфейсної частини програмного продукту.

Для забезпечення подальшої можливості удосконалення програмного продукту, він має бути багатофайловим.

В головному файлі мають бути розміщені:

– процедури оброблення подій графічного інтерфейсу;

– основна процедура шифрування.

Процедура створення шифруючої гами має бути винесена в окремий файл. Вона повинна, на основі введеного повного імені відкритого файлу і змісту ключової комбінації, визначати довжину необхідної шифруючої гами і зберігати її в окремому тимчасовому файлі, який знищується після завершення шифрування.

Для забезпечення можливості звернення до процедури шифрування, до складу програмного продукту має бути введений додатковий заголовочний фай, у якому зберігається список зовнішніх процедур, винесених у додатковий файл.

Рисунок 3.1 – Структурна схема програмного проекту

Розроблений шифр має бути перевіреним на предмет його стійкості до атак повного перебору ключового простору. Для цього має бути розробленою модель формування шифруючої гами та проведені випробування екземплярів вихідного потоку на предмет його рівномірності за допомогою статистичного критерію χ^2-Пірсона.

Для випробування змодельованого потоку шифруючої гами має бути створений програмний код, що дозволяє виконувати розділення комбінацій створеної послідовності довжиною N чисел на k підмножин, обчислювати їх потужність та значення критерію рівномірності розподілення χ^2-Пірсона.

Програмний продукт, призначений для оцінки моделі шифруючої гами, має бути виконаний як окремий проект. Він має оброблювати вихідну послідовність шифруючої гами довільної довжини, а також обчислювати критичне значення показника χ^2-Пірсона для конкретного екземпляру гами та його фактичне значення, отримане на основі статистичного аналізу. Цей додатковий програмний продукт, також повинен мати графічний інтерфейс з відображенням отриманих результатів дослідження.

3.2 Програмна реалізація алгоритму генератора ПВЧ

Програмна реалізація алгоритму формування потоку шифруючої гами передбачає реалізацію генератора ПВЧ у вигляді окремої функції, реалізованої в окремому файлі проекту.

Вхідними параметрами такої функції є символьний масив, в якому містяться 16 символів, що утворюють ключ до шифру, введений з клавіатури комп'ютера, а також, ім'я файлу, що має бути зашифрованим.

Результатом роботи програмного модулю є тимчасовий файл, що містить шифруючи гаму, довжина якої співпадає з розміром відкритого файлу. До складу модуля також входить функція, що знищує створений вихідний файл після завершення процесу шифрування.

Схема алгоритму формування шифруючої гами наведена на рисунку 3.2.

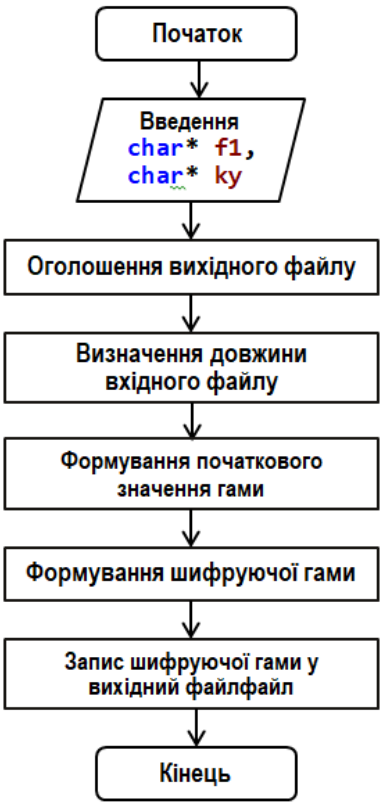

Рисунок 3.2 – Схема алгоритму формування шифруючої гами

Після виклику функції

```
void prng(char* f1, char* ky)
```

на диску D створюється вихідний файл **d:\\rng.txt**, в який буде записуватись послідовність створених псевдовипадкових чисел

```
ofstream out("d:\\rng.txt");
if (!out) { return; }
```

Далі, визначається довжина відкритого фалу **f1** файлу, щоб можна було сформувати шифруючу гаму відповідної довжини

```
                                        // Визначення довжини файлу
ifstream in(f1, ios::in | ios::binary);
if (!in.is_open()) { return; }

in.seekg(0, ios::end);      // Вказівник читання на кінець файлу
long fileSize = in.tellg(); // Повернення позиції вказівника
in.seekg(0, ios::beg);      // Вказівник читання на початок файлу

double ln = fileSize / 4;   // Визначення довжини файлу з гамою
long lg = (long)ln + 1;
```

Спочатку відкривається бінарний доступ до файлу **f1**. Потім вказівник файлу переставляється на його кінець, зчитується номер позиції файлу у змінну **long fileSize**, а потім отримана довжина файлу у байтах розділяється на групи по чотири байти та визначається кількість таких груп. Отримана величина записується у змінну **long lg**.

Наступним етапом обчислень у функції **prng** є перетворення 16 символів введених користувачем з клавіатури в чотири беззнакових чотирьохбайтових числа. Ці перетворення виконуються за чотири ітерації

```
unsigned int t, x, y, z, w;
/* ======================================== */
   x = x << 8;  x = x +  (int)ky[0]; //  1
   x = x << 8;  x = x +  (int)ky[1]; //  2
   x = x << 8;  x = x +  (int)ky[2]; //  3
   x = x << 8;  x = x +  (int)ky[3]; //  4
/* ======================================== */
   y = y << 8;  y = y +  (int)ky[4]; //  5
   y = y << 8;  y = y +  (int)ky[5]; //  6
   y = y << 8;  y = y +  (int)ky[6]; //  7
   y = y << 8;  y = y +  (int)ky[7]; //  8
/* ======================================== */
   z = z << 8;  z = z +  (int)ky[8]; //  9
   z = z << 8;  z = z +  (int)ky[9]; // 10
   z = z << 8;  z = z + (int)ky[10]; // 11
   z = z << 8;  z = z + (int)ky[11]; // 12
/* ======================================== */
   w = w << 8;  w = w + (int)ky[12]; // 13
```

```
    w = w << 8;  w = w + (int)ky[13]; // 14
    w = w << 8;  w = w + (int)ky[14]; // 15
    w = w << 8;  w = w + (int)ky[15]; // 16
/* ======================================== */
```

Із наведеного фрагменту коду видно, що у формуванні кожного числа типу **unsigned int** використовується чотири елементи символьного масиву **ky[i]**.

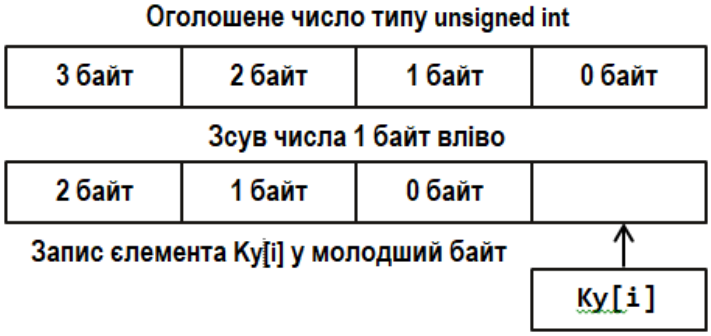

Рисунок 3.3 – Перетворення введеного ключа у 32-бітні числа

Спочатку оголошене число **x** зсувається на один байт вліво, звільняючи місце для молодшого байту числа, а потім, на очищене місце, у молодший байт записується поточний елемент ключового масиву. Далі, так само, у продовж чотирьох ітерацій, заповнюються наступні три байти. Наприкінці, всі чотири байти оголошеного числа **x** будуть заповнені першими чотирма символьними змінними введеними з клавіатури.

Аналогічно, формуються числа **y, z** і **w**.

Після створення початкових чисел **x, y, z** і **w** для Xorshift-генератора, відбувається формування псевдовипадкових чисел, що утворюють шифрувальну гаму. Кожне сформоване 32-х бітне число з виходу генератора одразу записується у вихідний файл **d:\\rng.txt**.

```
for (int i = 0; i < lg; i++)
{
   t = x ^ (x << 11);
   x = y; y = z; z = w;
   w = (w ^ (w >> 19)) ^ (t ^ (t >> 8));
   out.write((char*)&w, sizeof(unsigned int));
}  out.close();
```

Як видно із наведеного фрагменту коду, утворення відбувається ітераційному циклі, обмеженому довжиною відкритому файлу **lg**, вираженому у

чотирьох байтових одиницях і визначеною на початку функції. Файл з шифруючою гамою зберігається на диску D до завершення шифрування. Перед завершенням шифрувального процесу викликається функція delt(), яка знищує його, звільнюючи місце у пам'яті, яке він займає.

```
void delt()
{
   if(remove("d:\\rng.txt")) { return; }
}
```

Таким чином, зовнішня функція

```
prng(char* f1, char* ky);
```

утворює тимчасовий файл з шифрувальною гамою, яка викликається з функції шифрування

```
void cript(char* f1, char* f2)
```

і використовується для поточного шифрування. Повний код функції **prng()** наведений у додатку 1.

3.3 Програмна реалізація алгоритму поточного шифрування

Головна функція програмування отримує дані про розміщення відкритого і закритого файлів, що вводяться користувачем з головного інтерфейсу програми. Вона виконує наступні задачі:

– утворює відповідні вхідний і вихідний потоки до зовнішніх файлів;

– виконує гамування вхідного файлу і записує результат у вихідний файл;

– виконує управління відображенням процесу шифрування.

На початку роботи функції створюються відповідні потоки для роботи з вхідним та вихідним файлами, а також з файлом, в якому міститься шифруючи гама **d:\\rng.txt**.

Спочатку оголошуються відповідні потоки

```
ifstream kl  ("d:\\rng.txt", ios::in | ios::binary);
ifstream in  (f1, ios::in  | ios::binary);
ofstream out (f2, ios::out | ios::binary);
```

Потім, перевіряється, чи відкриті відповідні файли

```
if (!in.is_open())  { return; }
```

```
if (!kl.is_open())  { return; }
if (!out.is_open()) { return; }
```

Далі, вказівник вхідного файлу переводиться у кінець цього файлу і зчитується його положення. Після цього вказівник повертається на початок файлу.

```
in.seekg(0, ios::end);  // Вказівник читання на кінець файлу
fileSize = in.tellg();  // Повернення позиції вказівника
in.seekg(0, ios::beg);  // Вказівник читання на початок файлу
```

Після підготовки файлів до роботи, створюється три динамічні масиви для тимчасового зберігання даних вхідного файлу, шифруючої гами та результату шифрування

```
char* readBuffer  = new char[fileSize];
char* gammBuffer  = new char[fileSize];
char* writeBuffer = new char[fileSize];
```

і заповнюються відповідними даними з файлових потоків

```
in.read(readBuffer, fileSize);  // Запис у буфер
kl.read(gammBuffer, fileSize);  // Запис у буфер
```

Саме шифрування виконується як циклічне сумування по модулю два відповідних байтів із буферів з текстом та шифруючою гамою

```
for (unsigned int i = 0; i < fileSize; ++i)
{
  writeBuffer[i] = readBuffer[i] ^ gammBuffer[i];
  PrBar->Value = i; // Управління елементом Progress Bar
}
```

Причому, паралельно в циклі відображується стан процесу шифрування. По закінчені процесу шифрування виконується запис результатів шифрування з вихідного буфера у вихідний файл

```
out.write(writeBuffer, fileSize);
```

і звільнюється пам’ять від всіх трьох оголошених динамічних масивів та потоків для зв’язку з зовнішніми файлами.

```
delete[] readBuffer;    // Звільнення пам’яті
delete[] gammBuffer;    // Звільнення пам’яті
delete[] writeBuffer;   // Звільнення пам’яті

in.close();             // Закриваємо потік
kl.close();             // Закриваємо потік
out.close();            // Закриваємо потік
```

Для того, щоб запущена процедура шифрування виконала шифрування

вхідного коду, необхідно спочатку створити зовнішній файл з шифруючою гамою, викликавши процедуру

```
void prng (char* f1, char* ky);
```

і, тільки потім, вже викликати процедуру

```
void cript(char* f1, char* f2)
```

Послідовність цих дій задається в обробнику події натискання кнопки **RUN** на формі інтерфейсу програми

```
  private: System::Void Butt1_Click(System::Object^ sender,
                                                System::EventArgs^ e) {
  Lab4->Text = "";
  String^ ins = TBox1->Text;  char* fin = StrToChar(ins);
  String^ ots = TBox2->Text;  char* fot = StrToChar(ots);
  String^ key = TBox3->Text;  char* kch = StrToChar(key);
  prng(fin, kch);
  cript(fin, fot);
}
```

Тут з відповідних текстових вікон зчитуються повні імена відповідних файлів та символи ключа, які передаються у відповідні функції у якості аргументів.

Зовнішній вигляд програмного інтерфейсу наведений на рисунку 3.4.

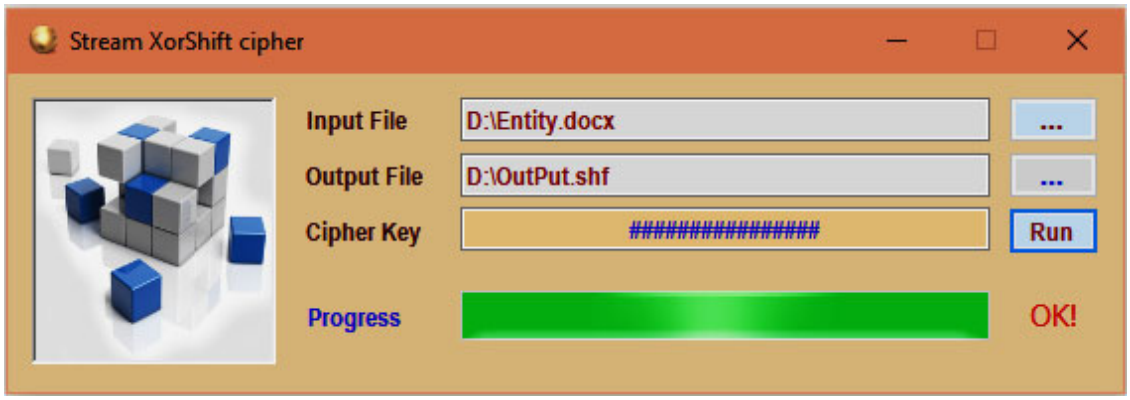

Рисунок 3.4 – Інтерфейсна частина програмного продукту

На форму проекту винесено п’ять міток для пояснюючих надписів, три текстових вікна TBox1, TBox2 і TBox3, для вибору імен відкритого і закритого файлів, а також ключа до шифру відповідно. За допомогою кнопок Butt1 і Butt2 відкриваються провідники для вибору відповідних файлів. Обробники цих подій

описані в основному файлі програми і мають наступний вигляд.

```
private: System::Void Butt2_Click(System::Object^ sender,
                                                  System::EventArgs^ e) {
  OpenFileDialog^ OpnDlg = gcnew OpenFileDialog();
  OpnDlg->Filter = "All files (*.*)|*.*";
  OpnDlg->Title = "Выбор файла";
  if (OpnDlg->ShowDialog() ==
                System::Windows::Forms::DialogResult::OK)
  { TBox1->Text = OpnDlg->FileName; }
  Butt1->Enabled = true;
}

private: System::Void Butt3_Click(System::Object^ sender,
                                                  System::EventArgs^ e) {
  OpenFileDialog^ OpnDlg = gcnew OpenFileDialog();
  OpnDlg->Filter = "All files (*.*)|*.*";
  OpnDlg->Title = "Выбор файла";
  if (OpnDlg->ShowDialog() ==
System::Windows::Forms::DialogResult::OK)
  { TBox2->Text = OpnDlg->FileName; }
}
```

Інтерфейс програми містить засоби автоматизації, що дозволяють ввести ключ до шифру, вибрати файл, який має бути зашифрованим, вказати ім’я вихідного зашифрованого файлу і виконати шифрування.

Обробник натискання кнопки Butt3 викликає процедури створення шифруючої гами prng(fin, kch) і шифрування cript(fin, fot). Його вигляд наведено вище.

Через елемент Progress Bar відстежується процес шифрування.

3.4 Дослідження і оцінка моделі побудованого поточного алгоритму

Криптографічна стійкість алгоритму шифрування визначається рівномірністю розподілення символів у шифруючій гамі. Щоб оцінити таку стійкість використовуються, спеціально розроблені, статистичні тести. Національним інститутом стандартів і технологій США (National Institute of Standards and Technology – NIST) [12] свого часу було запропоновано пакет статистичних тестів та методику оцінювання генераторів для криптографічних потреб. Кожен із запропонованих цим документом тестів уявляє реалізацію критерію χ^2-Пірсона, де у якості предмету дослідження використовується та або

інша конфігурація бінарних символів.

З урахуванням сказаного для оцінки створеної програмної реалізації потокового шифру так само використаємо критерій χ^2-Пірсона, а у якості об'єкту дослідження обираємо бінарні комбінації. З яких складаються цілі числа деякого розміру. Це дозволить виконати:

– перевірки на рівномірність розподілу;

– перевірки на статистичну незалежність.

Проведення випробувань з використанням критерію χ^2-Пірсона, передбачає сортування чисел з виходу генератора ПВЧ за їх значенням по k однаковим інтервалам. Для цього обирається розмір числа у байтах. Наприклад, нехай це будуть двобаштові числа. Тоді максимальним числом буде число $2^{16} - 1$. Обираємо також кількість інтервалів $k = 16$. Тоді межі таких інтервалів будуть мати значення, наведені в наступному фрагменті коду

```
for (int i = 0; i < ln; ++i) {          // Calculating the number of hits in each interval
if (msv[i] >=     0 && msv[i] <  4095) { hkv[0]  = hkv[0]  + 1; }   //  1
if (msv[i] >=  4096 && msv[i] <  8191) { hkv[1]  = hkv[1]  + 1; }   //  2
if (msv[i] >=  8192 && msv[i] < 12287) { hkv[2]  = hkv[2]  + 1; }   //  3
if (msv[i] >= 12288 && msv[i] < 16383) { hkv[3]  = hkv[3]  + 1; }   //  4
if (msv[i] >= 16384 && msv[i] < 20479) { hkv[4]  = hkv[4]  + 1; }   //  5
if (msv[i] >= 20480 && msv[i] < 24575) { hkv[5]  = hkv[5]  + 1; }   //  6
if (msv[i] >= 24576 && msv[i] < 28671) { hkv[6]  = hkv[6]  + 1; }   //  7
if (msv[i] >= 28672 && msv[i] < 32767) { hkv[7]  = hkv[7]  + 1; }   //  8
if (msv[i] >= 32768 && msv[i] < 36863) { hkv[8]  = hkv[8]  + 1; }   //  9
if (msv[i] >= 36864 && msv[i] < 40959) { hkv[9]  = hkv[9]  + 1; }   // 10
if (msv[i] >= 40960 && msv[i] < 45055) { hkv[10] = hkv[10] + 1; }   // 11
if (msv[i] >= 45056 && msv[i] < 49151) { hkv[11] = hkv[11] + 1; }   // 12
if (msv[i] >= 49152 && msv[i] < 53247) { hkv[12] = hkv[12] + 1; }   // 13
if (msv[i] >= 53248 && msv[i] < 57343) { hkv[13] = hkv[13] + 1; }   // 14
if (msv[i] >= 57344 && msv[i] < 61439) { hkv[14] = hkv[14] + 1; }   // 15
if (msv[i] >= 61440 && msv[i] < 65535) { hkv[15] = hkv[15] + 1; } } // 16
```

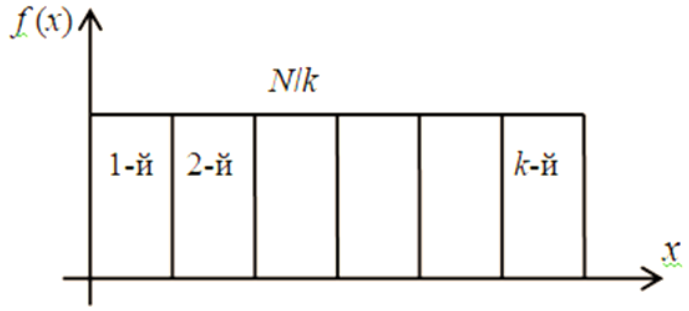

Рисунок 3.5 – Розподілення послідовності шифруючої гами на k інтервалів

З наведеного фрагменту коду видно, що в розмір кожного інтервалу має величину (2^{16} – 1)/16 = 4096. Кожне число з виходу генератора шифруючої гами перевіряється на предмет сегменту, до якого воно належить, а потім число, записане у відповідну ячейку масиву **`hkv[i]`** збільшується на одиницю. Таким чином, після аналізу всіх **`ln`** чисел в ячейках масиву **`hkv[i]`** буде записано кількість чисел, що попадають у відповідний інтервал.

Якщо у кожен інтервал попадає n_i чисел, то їх сума дорівнює $n_1 + n_2 + \ldots + n_k = N$. Визначення експериментального значення показника χ^2 оцінюється через суму квадратів різниць між отриманими числами n_i та їх еталонним значенням $p_i \cdot N$.

$$\chi^2 = (n_1 - p_1 N)^2 + (n_2 - p_2 N)^2 + \cdots + (n_k - p_k N)^2 \tag{3.1}$$

Для рівномірного розподілення ймовірностей $p_1 = p_2, = \ldots = p_k$. З наведеного виразу видно, що чим менше різниця в кожному із складових частин, тим менше значення показника χ^2. В теорії статистики [6], кожна складова частина формули приводиться до величини $p_i \cdot N$ і, тоді, експериментальне значення показника рівномірності показник виглядає як

$$\chi^2 = \frac{(n_1 - p_1 N)^2}{p_1 N} + \frac{(n_2 - p_2 N)^2}{p_1 N} + \cdots + \frac{(n_k - p_k N)^2}{p_k N} \tag{3.2}$$

або

$$\chi^2_{Exp} = \sum_{i=1}^{k} \frac{(n_i - p_i N)^2}{p_i N} \tag{3.3}$$

Теоретичне значення цього показника $\chi^2_{\text{теор}}$ можна отримати, користуючись функціями, вбудованими у більшість спеціальних математичних проектів, або скористатись відомими способами їх обчислень. У випадку рівномірного розподілу ймовірностей, вхідними параметрами до них є α – ймовірність того, наскільки генератор ПВЧ повинен відповідати вимогам рівномірного розподілу, а також, зменшена на одиницю, кількість інтервалів k – 1, на які розбивається числова послідовність шифруючої гами.

У разі, якщо $\chi^2_{\text{эксп}}$ значно більше $\chi^2_{\text{теор}}$ (α – велике), то генератор гами не відповідає вимогам рівномірності, через те, що отримані значення n_i значно відходять від теоретичних значень $p_i \cdot N$, і не є випадковими.

Методика перевірки за критерієм χ^2 виглядає так.

1. Діапазон чисел на виході генератора гами розбивається на k рівних інтервалів.

2. Формується послідовність вихідних символів формату 2^m.

3. Підраховується кількість ПВЧ, що попали в кожен інтервал:

$$n_i, i = 1, \dots, k.$$

4. Обчислюється значення $\chi^2_{\text{эксп}}$ за формулою (3.3).

5. На основі порівняння $\chi^2_{\text{эксп}}$ та $\chi^2_{\text{теор}}$ робиться висновок про придатність генератора для його використання.

Експеримент повторюється не менше 5 разів і отримані результати усереднюються.

В процесі дипломного проектування розроблено програму, що дозволяє виконувати кількісну оцінку запропонованого алгоритму шифрування. Його графічний інтерфейс наведено на рисунку 3.6.

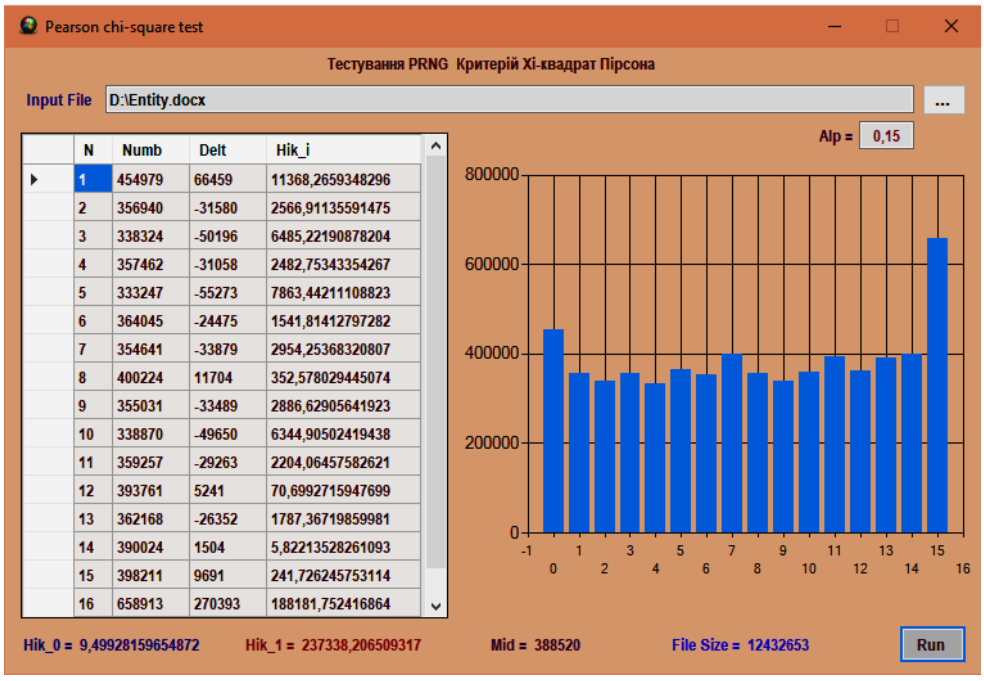

Рисунок 3.6 – Розподілення двобаштових чисел у відкритому файлі

Вхідними параметрами до функції проведення випробувань є шлях до зашифрованого файлу та показник α, що визначає вимоги до рівномірності розподілення вихідних чисел. Кількість інтервалів є фіксованою і дорівнює 16. Термінальні засоби інтерфейсу дозволяють вводити ці параметри перед початком випробувань, а також відображати розподілення 16-розрядних чисел по 16 інтервалах у вигляді таблиці та гістограми.

Крім того користувачеві видаються дані про кількість байтів у файлі, очікувану кількість чисел в кожному сегменті та величини $\chi^2_{эксп}$ і $\chi^2_{теор}$. На рисунку 3.7 наведено результати розподілення ймовірностей чисел у вихідному (зашифрованому) файлі.

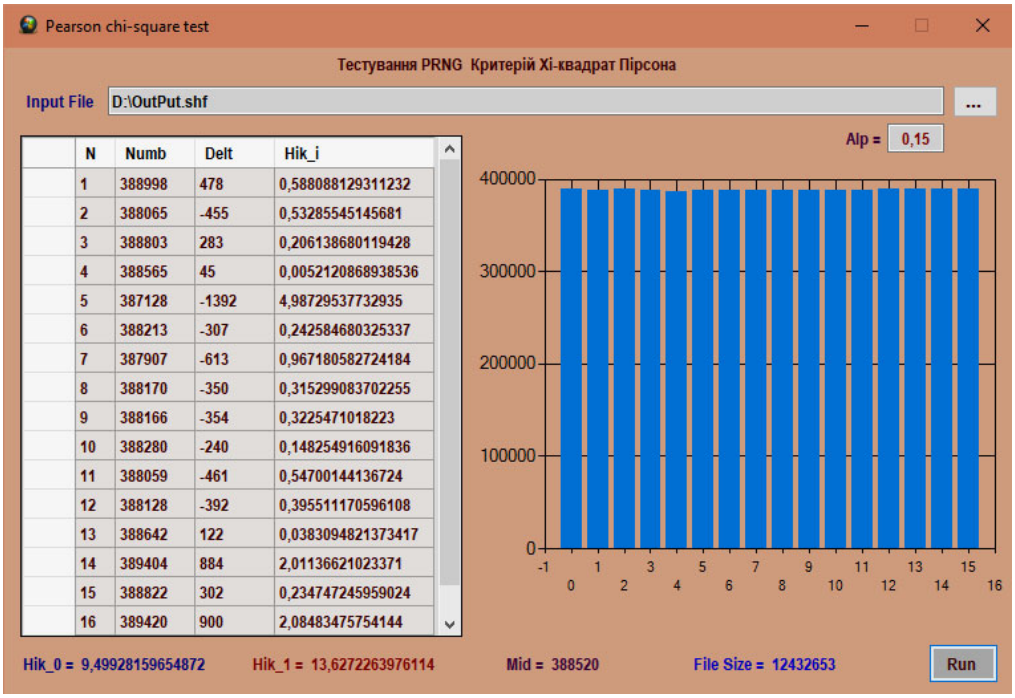

Рисунок 3.7 – Розподілення двобаштових чисел у відкритому файлі

Результати проведених досліджень показують, що у переважній більшості проведених випробувань отриманий результат наближається до очікуваного теоретичного значення $\chi^2_{теор}$, що свідчить про достатню стійкість обраного алгоритму шифрування.

Експеримент проводився багаторазово з різними типами фалів. Отримані результати показали, що тип файлу суттєво не впливає на стійкість шифрування. В параграфі 2.2 (вирази 2.1 і 2.2) було показано, що розподілення ймовірності

символів у зашифрованому файлі повністю визначається розподіленням символів у шифруючий гамі і це повністю підтверджується проведеними випробуваннями.

Висновки за розділом 3

Програмна реалізація поточного шифру повинна забезпечувати швидкісне шифрування файлів будь-якого формату і надавати користувачеві зручний зрозумілий інтерфейс. Органи управління шифрування повинні забезпечувати введення ключа до шифру та повних імен відкритого і закритого файлів, а також дозволяти відстежувати процес шифрування.

Найкращим варіантом мови програмування логічно обрати мову C++, оскільки вона є найближчою до мови низького рівню Асемблера, що дає можливість створювати найбільш простий і ефективний, з точки зору використання обчислювальних ресурсів, код. Найбільш придатним середовищем програмування слід обрати середовище Visual Studio, яке забезпечує широкі можливості у створенні графічних додатків.

Програмний проект має бути багато файловим, щоб забезпечувалась можливість подальшого його удосконалення. Зокрема у додатковий файл мають бути винесені процедури створення шифруючої гами і звільнення дискового простору після завершення шифрування.

Враховуючи, що стійкість шифрів пропонується оцінювати статистичним тестуванням, для оцінки стійкості запропонованого алгоритму використано критерій χ^2-Пірсона. Проведені випробування показали, що створений шифр відповідає вимогам поставленим на дипломне проектування.

ВИСНОВКИ

Створення та удосконалення ефективних швидкодіючих потокових шифрів, що працюють в реальному масштабі часу, є перспективним напрямом розвитку сучасної криптографії.

Підвищення ефективності потокового шифрування, у першу чергу, має зосереджуватись на створенні швидкодіючого генератора, який вимагає невеликої кількості ресурсів обчислювальної системи. Економія таких ресурсів досягається використанням елементарних операцій мікропроцесора.

Найкращою мовою програмування для економічної реалізації потокового шифрування є мова С++, оскільки вона стоїть найближче до мов низького рівню.

Головною вимогою до генераторів шифруючої гами є рівномірність розподілення бінарних символів у складі вихідного потоку шифруючої гами. Саме на цьому мають бути зосереджені зусилля розробників під час ствоення або вибору генератора гами.

Для ефективної програмної реалізації потокового шифру у різні часи пропонувались генератори зі зворотним зв’язком (LFSR-генератори) та генератори Фібоначчі з запізненням. Пізніше був запропонований Xorshift-генератор, який, на відміну від попередніх варіантів генераторів, відрізняється використанням лише найпростіших елементарних обчислювальних операцій. З урахуванням обмеженості обчислювальних та часових ресурсів у можливого супротивника і відомих результатів дослідження Xorshift-генератора, слід саме його обрати у якості генератора шифруючої гами.

Програмна реалізація розробленого алгоритму шифрування, по-перше, повинна відповідати вимогам ефективності захисту і програмної реалізації, і, по-друге, надавати користувачеві зручний, простий і зрозумілий інтерфейс для користування. Для рішення цієї задачі найбільше підходить середовище програмування Visual Studio, оскільки воно надає можливість створення графічного інтерфейсу з елементами введення і виведення інформації, що використовується у процесі шифрування, а також дозволяє створювати багато

файлові програмні проекти мовою C++.

Для попередньої оцінки стійкості використовується статистичне тестування потоку шифруючої гами або вихідної послідовності зашифрованого тексту. Враховуючи, що в основу переважної більшості тестових пакетів закладений тест χ^2-Пірсона, слід саме його використати для виконання статистичної оцінки рівномірності розподілення ймовірностей у потоках бінарних символів.

Проведені статистичні випробування запропонованого в роботі алгоритму шифрування показали, що створений шифр відповідає вимогам поставленим на дипломне проектування.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Bruce Schneier. Applied Cryptography: Protocols, Algorithms, and Source Code in C, 2nd Edition 662. Pages 1995.
2. Rueppel R.A. Good stream ciphers are hard to design. Conference: Security Technology, 1989. Proceedings. 1989 International Carnahan Conference. February 1989. DOI:10.1109/CCST.1989.751974. URL : https://www.researchgate.net/publication/3790837_Good_stream_ciphers_are_hard_to_design.
3. Christof Paar. Understanding Cryptograph. Springer Heidelberg Dordrecht London New York. DOI 10.1007/978-3-642-04101-3. 2010. 372 p. URL : https://moodlearchive.epfl.ch/2022-2023/pluginfile.php/2319743/mod_resource/content/1/Understanding-Cryptography.pdf.
4. Hu Qi. Stream Ciphers and Linear Complexity. 8-Jan-2008. – 84 p. URL : https://www.cs.umd.edu/~huqi/MasterThesisR.pdf.
5. Claude E. Shannon Communication theory of secrecy systems 01 Oct 1949-Bell System Technical Journal (Alcatel-Lucent)-Vol. 28, Iss: 4, pp 656-715.
6. Knuth, D. E. The Art of Computer Programming. Volume 2. Seminumerical Algorithms. 3rd edition / D. E. Knuth. – Boston, Mass, USA : Addison-Wesley, Longman Publishing, 1997. – 762 p
7. A Statistical Test Suite for Random and Pseudorandom Number Generators for Cryptographic Applications. Special Publication 800-22. URL : https://nvlpubs. nist .gov/nistpubs/Legacy/SP/nistspecialpublication800-22r1a.pdf.
8. Ferguson N., Schneier B. Practical cryptography Computer Science, Mathematics / B. 2003. 539 p.
9. S. Halevi and H. Krawczyk. MMH: message authentication in software in the Gbit/second rates. In Proceedings of the 4th Workshop on Fast Software Encryption, volume 1267, pages 172–189. Springer, 1997. URL : https://www. semanticscholar. org/paper/MMH%3A-Software-Message-Authentication-in-the-Gbit-Halevi-Krawczyk/4cbe9fa3f41e0a1070218ef26644b545214be953
10. George Marsaglia, Liang-Huei Tsay. Matrices and the structure of random number sequences. Linear Algebra Appl. 67 (1985), 147–156. https://doi.org/10.1016/0024-3795(85)90192-2 URL: https://www.sciencedirect.com/science/article/pii/0024379585901922
11. George Marsaglia. Random Number Generators. 2003. DOI 10.22237/jmasm/

1051747320 URL: https://digitalcommons.wayne.edu/jmasm/vol2/iss1/2/

12. NIST Special Publication 800-22: A Statistical Test Suite for the Validation of Random Number Generators and Pseudo Random Number Generators for Cryptographic Applications. April 15, 2010. URL : https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-22r1a.pdf

ДОДАТКИ

Додаток А. Програмний код функцій формування шифруючої гами та функцій для знищення тимчасового зовнішнього файлу

```
#include "header.h"
#include <iostream>
#include <fstream>          // for ofstream, ifstream
#include <random>
#include <ctime>            // for function time()

using namespace std;

void prng(char* f1, char* ky)
{
   ofstream out("d:\\rng.txt");
   if (!out) { return ; }
                                                  // Визначення довжини
файлу
   ifstream in(f1, ios::in | ios::binary); // Створення вхідного
потоку
   if (!in.is_open()) { return; }

   in.seekg(0, ios::end);         // Вказівник читання на кінець файлу
   long fileSize = in.tellg();    // Повернення позиції вказівника
   in.seekg(0, ios::beg);         // Вказівник читання на початок
файлу

   double ln = fileSize / 4;      // Визначення довжини файлу з гамою
   long lg = (long)ln + 1;
                            // Формування початкових чисел генератора
   unsigned int t, x, y, z, w;
   /* ========================================== */
      x = x << 8;  x = x +  (int)ky[0]; //  1
      x = x << 8;  x = x +  (int)ky[1]; //  2
      x = x << 8;  x = x +  (int)ky[2]; //  3
      x = x << 8;  x = x +  (int)ky[3]; //  4
   /* ========================================== */
      y = y << 8;  y = y +  (int)ky[4]; //  5
      y = y << 8;  y = y +  (int)ky[5]; //  6
      y = y << 8;  y = y +  (int)ky[6]; //  7
      y = y << 8;  y = y +  (int)ky[7]; //  8
   /* ========================================== */
      z = z << 8;  z = z +  (int)ky[8]; //  9
      z = z << 8;  z = z +  (int)ky[9]; // 10
      z = z << 8;  z = z + (int)ky[10]; // 11
      z = z << 8;  z = z + (int)ky[11]; // 12
   /* ========================================== */
      w = w << 8;  w = w + (int)ky[12]; // 13
      w = w << 8;  w = w + (int)ky[13]; // 14
      w = w << 8;  w = w + (int)ky[14]; // 15
```

```
      w = w << 8;  w = w + (int)ky[15]; // 16
   /* ========================================== */
      for (int i = 0; i < lg; i++)
      {
         t = x ^ (x << 11);
         x = y; y = z; z = w;
         w = (w ^ (w >> 19)) ^ (t ^ (t >> 8));
         out.write((char*)&w, sizeof(unsigned int));
      }  out.close();
} /* ============================================== */
void delt()
{
   if (remove("d:\\rng.txt")) { return; }
}
```

Додаток Б. Програмний код форми графічного інтерфейсу та функції шифрування

```
#pragma once
#include "header.h"
#include <iostream>
#include <fstream>
#include <stdio.h>          // fof  remove(fname)

namespace DipMike04 {

using namespace std;
using namespace System;
using namespace System::IO;
using namespace System::Data;
using namespace System::Drawing;
using namespace System::Threading;
using namespace System::Collections;
using namespace System::Windows::Forms;
using namespace System::ComponentModel;
using namespace System::Runtime::InteropServices;

  /// <summary>
  /// Сводка для Form1
  /// </summary>
  public ref class Form1 : public System::Windows::Forms::Form
  {
  public:
    Form1(void)
    {
      InitializeComponent();
      //
      //TODO: добавте код конструктора
      //
    }

  protected:
    /// <summary>
    /// Звільнити всі використані ресурси.
    /// </summary>
    ~Form1()
    {
      if (components)
      {
        delete components;
      }
    }
  private: System::Windows::Forms::Label^ Lab1;
  private: System::Windows::Forms::Label^ Lab2;
  private: System::Windows::Forms::Label^ Lab3;
  private: System::Windows::Forms::Label^ Lab4;
  private: System::Windows::Forms::Label^ Lab5;
  private: System::Windows::Forms::Button^ Butt1;
  private: System::Windows::Forms::Button^ Butt2;
```

```
  private: System::Windows::Forms::Button^ Butt3;
  private: System::Windows::Forms::TextBox^ TBox1;
  private: System::Windows::Forms::TextBox^ TBox2;
  private: System::Windows::Forms::TextBox^ TBox3;
  private: System::Windows::Forms::PictureBox^ PcBox;
  private: System::Windows::Forms::ProgressBar^ PrBar;
  private: System::Windows::Forms::OpenFileDialog^ OpnDlg;
  protected:

  private:
    /// <summary>
    /// Обов'язкова змінна конструктора.
  /* ############################################################ */
     char* StrToChar(String^ str)
     {
       IntPtr ptr = Marshal::StringToHGlobalAnsi(str);
       char* ch = (char*)ptr.ToPointer();
       return(ch);
     }
  /* ############################################################ */
    /// </summary>
    System::ComponentModel::Container ^components;

#pragma region Windows Form Designer generated code
    /// <summary>
    /// Необхідний метод для підтримки конструктора — не измюйте
    /// зміст метода за допомогою редактора кода.
    /// </summary>
    void InitializeComponent(void)
    {
      System::ComponentModel::ComponentResourceManager^ resources =
(gcnew
System::ComponentModel::ComponentResourceManager(Form1::typeid));
   this->Butt1 = (gcnew System::Windows::Forms::Button());
   this->Lab1 = (gcnew System::Windows::Forms::Label());
   this->TBox1 = (gcnew System::Windows::Forms::TextBox());
   this->TBox2 = (gcnew System::Windows::Forms::TextBox());
   this->TBox3 = (gcnew System::Windows::Forms::TextBox());
   this->Lab2 = (gcnew System::Windows::Forms::Label());
   this->Lab3 = (gcnew System::Windows::Forms::Label());
   this->Butt2 = (gcnew System::Windows::Forms::Button());
   this->Butt3 = (gcnew System::Windows::Forms::Button());
   this->OpnDlg = (gcnew System::Windows::Forms::OpenFileDialog());
   this->Lab4 = (gcnew System::Windows::Forms::Label());
   this->PrBar = (gcnew System::Windows::Forms::ProgressBar());
   this->Lab5 = (gcnew System::Windows::Forms::Label());
   this->PcBox = (gcnew System::Windows::Forms::PictureBox());
(cli::safe_cast<System::ComponentModel::ISupportInitialize^>(this-
>PcBox))->BeginInit();
      this->SuspendLayout();
      //
      // Butt1
      //
      this->Butt1->BackColor =
System::Drawing::SystemColors::GradientActiveCaption;
      this->Butt1->ForeColor = System::Drawing::Color::Maroon;
```

```
        this->Butt1->Location = System::Drawing::Point(504, 67);
        this->Butt1->Name = L"Butt1";
        this->Butt1->Size = System::Drawing::Size(46, 24);
        this->Butt1->TabIndex = 0;
        this->Butt1->Text = L"Run";
        this->Butt1->UseVisualStyleBackColor = false;
        this->Butt1->Click += gcnew System::EventHandler(this,
&Form1::Butt1_Click);
        //
        // Lab1
        //
        this->Lab1->AutoSize = true;
        this->Lab1->Location = System::Drawing::Point(147, 15);
        this->Lab1->Name = L"Lab1";
        this->Lab1->Size = System::Drawing::Size(59, 16);
        this->Lab1->TabIndex = 1;
        this->Lab1->Text = L"Input File";
        //
        // TBox1
        //
        this->TBox1->BackColor = System::Drawing::Color::LightGray;
        this->TBox1->ForeColor = System::Drawing::Color::Maroon;
        this->TBox1->Location = System::Drawing::Point(228, 12);
        this->TBox1->Name = L"TBox1";
        this->TBox1->Size = System::Drawing::Size(267, 22);
        this->TBox1->TabIndex = 2;
        this->TBox1->Text = L"D:\\";
        //
        // TBox2
        //
        this->TBox2->BackColor = System::Drawing::Color::LightGray;
        this->TBox2->ForeColor = System::Drawing::Color::Maroon;
        this->TBox2->Location = System::Drawing::Point(228, 40);
        this->TBox2->Name = L"TBox2";
        this->TBox2->Size = System::Drawing::Size(267, 22);
        this->TBox2->TabIndex = 2;
        this->TBox2->Text = L"D:\\OutPut.shf";
        //
        // TBox3
        //
        this->TBox3->BackColor = System::Drawing::Color::BurlyWood;
        this->TBox3->ForeColor =
System::Drawing::Color::FromArgb(static_cast<System::Int32>(static_c
ast<System::Byte>(0)),
static_cast<System::Int32>(static_cast<System::Byte>(0)),
          static_cast<System::Int32>(static_cast<System::Byte>(192)));
        this->TBox3->Location = System::Drawing::Point(228, 67);
        this->TBox3->Name = L"TBox3";
        this->TBox3->Size = System::Drawing::Size(267, 22);
        this->TBox3->TabIndex = 2;
        this->TBox3->Text = L"****************";
        this->TBox3->TextAlign =
System::Windows::Forms::HorizontalAlignment::Center;
        //
        // Lab2
        //
```

```
        this->Lab2->AutoSize = true;
        this->Lab2->Location = System::Drawing::Point(147, 43);
        this->Lab2->Name = L"Lab2";
        this->Lab2->Size = System::Drawing::Size(67, 16);
        this->Lab2->TabIndex = 1;
        this->Lab2->Text = L"Output File";
        //
        // Lab3
        //
        this->Lab3->AutoSize = true;
        this->Lab3->Location = System::Drawing::Point(147, 71);
        this->Lab3->Name = L"Lab3";
        this->Lab3->Size = System::Drawing::Size(66, 16);
        this->Lab3->TabIndex = 1;
        this->Lab3->Text = L"Cipher Key";
        //
        // Butt2
        //
        this->Butt2->BackColor =
System::Drawing::SystemColors::GradientActiveCaption;
        this->Butt2->Font = (gcnew System::Drawing::Font(L"Tahoma",
9.75F, System::Drawing::FontStyle::Bold,
System::Drawing::GraphicsUnit::Point,
          static_cast<System::Byte>(204)));
        this->Butt2->ForeColor = System::Drawing::Color::Maroon;
        this->Butt2->Location = System::Drawing::Point(504, 11);
        this->Butt2->Name = L"Butt2";
        this->Butt2->Size = System::Drawing::Size(46, 24);
        this->Butt2->TabIndex = 3;
        this->Butt2->Text = L"...";
        this->Butt2->UseVisualStyleBackColor = false;
        this->Butt2->Click += gcnew System::EventHandler(this,
&Form1::Butt2_Click);
        //
        // Butt3
        //
        this->Butt3->BackColor =
System::Drawing::SystemColors::ScrollBar;
        this->Butt3->Font = (gcnew System::Drawing::Font(L"Tahoma",
9.75F, System::Drawing::FontStyle::Bold,
System::Drawing::GraphicsUnit::Point,
          static_cast<System::Byte>(204)));
        this->Butt3->ForeColor =
System::Drawing::Color::FromArgb(static_cast<System::Int32>(static_c
ast<System::Byte>(0)),
static_cast<System::Int32>(static_cast<System::Byte>(0)),
          static_cast<System::Int32>(static_cast<System::Byte>(192)));
        this->Butt3->Location = System::Drawing::Point(504, 39);
        this->Butt3->Name = L"Butt3";
        this->Butt3->Size = System::Drawing::Size(46, 24);
        this->Butt3->TabIndex = 4;
        this->Butt3->Text = L"...";
        this->Butt3->UseVisualStyleBackColor = false;
        this->Butt3->Click += gcnew System::EventHandler(this,
&Form1::Butt3_Click);
        //
```

```
        // OpnDlg
        //
        this->OpnDlg->FileName = L"openFileDialog1";
        //
        // Lab4
        //
        this->Lab4->AutoSize = true;
        this->Lab4->Font = (gcnew System::Drawing::Font(L"Tahoma",
11.25F, System::Drawing::FontStyle::Regular,
System::Drawing::GraphicsUnit::Point,
          static_cast<System::Byte>(204)));
        this->Lab4->ForeColor =
System::Drawing::Color::FromArgb(static_cast<System::Int32>(static_c
ast<System::Byte>(192)),
static_cast<System::Int32>(static_cast<System::Byte>(0)),
          static_cast<System::Int32>(static_cast<System::Byte>(0)));
        this->Lab4->Location = System::Drawing::Point(512, 112);
        this->Lab4->Name = L"Lab4";
        this->Lab4->Size = System::Drawing::Size(0, 18);
        this->Lab4->TabIndex = 1;
        //
        // PrBar
        //
        this->PrBar->BackColor =
System::Drawing::SystemColors::GradientActiveCaption;
        this->PrBar->ForeColor =
System::Drawing::SystemColors::InactiveCaption;
        this->PrBar->Location = System::Drawing::Point(228, 109);
        this->PrBar->Name = L"PrBar";
        this->PrBar->Size = System::Drawing::Size(267, 25);
        this->PrBar->TabIndex = 5;
        //
        // Lab5
        //
        this->Lab5->AutoSize = true;
        this->Lab5->ForeColor =
System::Drawing::Color::FromArgb(static_cast<System::Int32>(static_c
ast<System::Byte>(0)),
static_cast<System::Int32>(static_cast<System::Byte>(0)),
          static_cast<System::Int32>(static_cast<System::Byte>(192)));
        this->Lab5->Location = System::Drawing::Point(148, 114);
        this->Lab5->Name = L"Lab5";
        this->Lab5->Size = System::Drawing::Size(55, 16);
        this->Lab5->TabIndex = 6;
        this->Lab5->Text = L"Progress";
        //
        // PcBox
        //
        this->PcBox->BackColor = System::Drawing::Color::Gainsboro;
        this->PcBox->BorderStyle =
System::Windows::Forms::BorderStyle::Fixed3D;
        this->PcBox->Image =
(cli::safe_cast<System::Drawing::Image^>(resources-
>GetObject(L"PcBox.Image")));
        this->PcBox->Location = System::Drawing::Point(12, 12);
        this->PcBox->Name = L"PcBox";
```

```
        this->PcBox->Size = System::Drawing::Size(123, 134);
        this->PcBox->SizeMode =
System::Windows::Forms::PictureBoxSizeMode::StretchImage;
        this->PcBox->TabIndex = 7;
        this->PcBox->TabStop = false;
        this->PcBox->Click += gcnew System::EventHandler(this,
&Form1::PcBox_Click);
        //
        // Form1
        //
        this->AutoScaleDimensions = System::Drawing::SizeF(7, 16);
        this->AutoScaleMode =
System::Windows::Forms::AutoScaleMode::Font;
        this->BackColor = System::Drawing::Color::Tan;
        this->ClientSize = System::Drawing::Size(562, 160);
        this->Controls->Add(this->PcBox);
        this->Controls->Add(this->Lab5);
        this->Controls->Add(this->PrBar);
        this->Controls->Add(this->Butt3);
        this->Controls->Add(this->Butt2);
        this->Controls->Add(this->TBox3);
        this->Controls->Add(this->TBox2);
        this->Controls->Add(this->TBox1);
        this->Controls->Add(this->Lab4);
        this->Controls->Add(this->Lab3);
        this->Controls->Add(this->Lab2);
        this->Controls->Add(this->Lab1);
        this->Controls->Add(this->Butt1);
        this->Font = (gcnew System::Drawing::Font(L"Arial Narrow",
9.75F, System::Drawing::FontStyle::Bold,
System::Drawing::GraphicsUnit::Point,
          static_cast<System::Byte>(204)));
        this->ForeColor =
System::Drawing::Color::FromArgb(static_cast<System::Int32>(static_c
ast<System::Byte>(64)),
static_cast<System::Int32>(static_cast<System::Byte>(0)),
          static_cast<System::Int32>(static_cast<System::Byte>(0)));
        this->FormBorderStyle =
System::Windows::Forms::FormBorderStyle::Fixed3D;
        this->Icon =
(cli::safe_cast<System::Drawing::Icon^>(resources-
>GetObject(L"$this.Icon")));
        this->Location = System::Drawing::Point(300, 150);
        this->Margin = System::Windows::Forms::Padding(3, 4, 3, 4);
        this->MaximizeBox = false;
        this->Name = L"Form1";
        this->StartPosition =
System::Windows::Forms::FormStartPosition::Manual;
        this->Text = L"Програма для симетричного поточного
шифрування";
        this->FormClosing += gcnew
System::Windows::Forms::FormClosingEventHandler(this,
&Form1::Form1_FormClosing);
        this->Load += gcnew System::EventHandler(this,
&Form1::Form1_Load);
```

```
(cli::safe_cast<System::ComponentModel::ISupportInitialize^>(this-
>PcBox))->EndInit();
      this->ResumeLayout(false);
      this->PerformLayout();
    }
#pragma endregion
//#################################################################
  void cript(char* f1, char* f2)
  {
   long  fileSize;
   PrBar->Value = 0;            // PrBar->Initial value
   PrBar->Minimum = 0;                 // PrBar->Minimum

   ifstream kl  ("d:\\rng.txt", ios::in | ios::binary);
   ifstream in  (f1, ios::in  | ios::binary);
   ofstream out (f2, ios::out | ios::binary);

   if (!in.is_open())  { return; }
   if (!kl.is_open())  { return; }
   if (!out.is_open()) { return; }

   in.seekg(0, ios::end);  // Вказівник читання на кінець файлу
   fileSize = in.tellg();  // Повернення позиції вказівника
   in.seekg(0, ios::beg);  // Вказівник читання на початок файлу
   PrBar->Maximum = fileSize;                          // Maximum

   char* readBuffer  = new char[fileSize];
   char* gammBuffer  = new char[fileSize];
   char* writeBuffer = new char[fileSize];

   in.read(readBuffer, fileSize);  // Запис у буфер
   kl.read(gammBuffer, fileSize);  // Запис у буфер

   for (unsigned int i = 0; i < fileSize; ++i)
   {
     writeBuffer[i] = readBuffer[i] ^ gammBuffer[i];
     PrBar->Value = i;                    // Maximum
   }
     Lab4->Text = "OK!";
     out.write(writeBuffer, fileSize); // Запис з буферу у out-файл

    in.close();                  // Закриваємо потік
    kl.close();                  // Закриваємо потік
    out.close();                 // Закриваємо потік

    delete[] readBuffer;            // Запис у буфер
    delete[] gammBuffer;            // Запис у буфер
    delete[] writeBuffer;           // Запис у буфер
  }
//#################################################################
  private:     System::Void     Form1_Load(System::Object^     sender,
System::EventArgs^ e) {
    Butt1->Enabled = false;
}//#################################################################
    private:    System::Void    Butt2_Click(System::Object^    sender,
System::EventArgs^ e) {
```

```
  OpenFileDialog^ OpnDlg = gcnew OpenFileDialog();
  OpnDlg->Filter = "All files (*.*)|*.*";
  OpnDlg->Title = "Выбор файла";
  if                    (OpnDlg->ShowDialog()                    ==
System::Windows::Forms::DialogResult::OK)
  { TBox1->Text = OpnDlg->FileName; } Butt1->Enabled = true;
}//###############################################################
   private: System::Void Butt3_Click(System::Object^ sender,
   System::EventArgs^ e) {
   OpenFileDialog^ OpnDlg = gcnew OpenFileDialog();
   OpnDlg->Filter = "All files (*.*)|*.*";
   OpnDlg->Title = "Вібір файла";
   if (OpnDlg->ShowDialog() ==
   System::Windows::Forms::DialogResult::OK)
   { TBox2->Text = OpnDlg->FileName; }
}//###############################################################
   private: System::Void Butt1_Click(System::Object^ sender,
   System::EventArgs^ e) {
   Lab4->Text = "";
   String^ ins = TBox1->Text;    char* fin = StrToChar(ins);
   String^ ots = TBox2->Text;    char* fot = StrToChar(ots);
   String^ key = TBox3->Text;    char* kch = StrToChar(key);
   prng(fin, kch);
   cript(fin, fot);
}//###############################################################
   private: System::Void Form1_FormClosing(System::Object^ sender,
   System::Windows::Forms::FormClosingEventArgs^ e) {
   delt();
}//###############################################################
   private: System::Void PcBox_Click(System::Object^ sender,
   System::EventArgs^ e) {
   if(TBox3->PasswordChar == false) { TBox3->PasswordChar = '#'; }
   else { TBox3->PasswordChar = false; }
}//###############################################################
};
}
```