**НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ**
**«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ**
**імені Ігоря СІКОРСЬКОГО»**
**НАВЧАЛЬНО-НАУКОВИЙ ФІЗИКО-ТЕХНІЧНИЙ ІНСТИТУТ**
**Кафедра математичних методів захисту інформації**

«До захисту допущено»
В.о. завідувача кафедри
_____ Сергій ЯКОВЛЄВ
«__» _____ 2022 р.

# Дипломна робота

**на здобуття ступеня бакалавра**

зі спеціальності: 113 Прикладна математика
на тему: **«Побудова нових тестів на випадковість коротких послідовностей із використанням теорії алгоритмічної інформації Колмогорова»**

Виконала: студентка 4 курсу, групи ФІ-84
Трет'якова Анна Олександрівна

Керівник: д.т.н., с.н.с. Кудін А.М. _____

Консультант: _ _____

Рецензент: звання, степінь, посада Прізвище І.П. _____

Засвідчую, що у цій дипломній роботі немає запозичень з праць інших авторів без відповідних посилань.

Студент _____

Київ — 2022

**НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ**
**«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ**
**імені Ігоря СІКОРСЬКОГО»**
**НАВЧАЛЬНО-НАУКОВИЙ ФІЗИКО-ТЕХНІЧНИЙ ІНСТИТУТ**
**Кафедра математичних методів захисту інформації**

Рівень вищої освіти — перший (бакалаврський)
Спеціальність (освітня програма) — 113 Прикладна математика,
ОПП «Математичні методи криптографічного захисту інформації»

ЗАТВЕРДЖУЮ
В.о. завідувача кафедри
_____ Сергій ЯКОВЛЄВ
«__» _____ 2022 р.

# ЗАВДАННЯ
## на дипломну роботу

Студент: Трет'якова Анна Олександрівна

1. Тема роботи: «*Побудова нових тестів на випадковість коротких послідовностей із використанням теорії алгоритмічної інформації Колмогорова*»,

керівник: д.т.н., с.н.с. Кудін А.М.,

затверджені наказом по університету №__ від «__» _____ 2022 р.

2. Термін подання студентом роботи: «__» _____ 2022 р.

3. Вихідні дані до роботи: *(впишіть вихідні дані до роботи)*

4. Зміст роботи: *Темою роботи є побудова нових тестів на випадковість коротких послідовностей із використанням теорії алгоритмічної інформації Колмогорова.*

*В роботі вирішуються наступні задачі:*

*1) провести аналіз актуальних теоретичних джерел пов'язаних з тематикою роботи;*

*2) виконати аналіз сучасних рішень схожих проблем;*

*3) обґрунтувати використання лінійного конгруентного генератора псевдовипадкових бітових послідовностей;*

*4) виконати програмну реалізацію лінійного конгруентного генератора;*

*5) експериментально перевірити можливість використання алгоритмів стискання даних для тестування псевдовипадкових послідовностей;*

*6) сформулювати формат тестування коротких бітових послідовностей.*

5. Перелік ілюстративного матеріалу (із зазначенням плакатів, презентацій тощо): *Презентація доповіді*

6. Дата видачі завдання: 10 вересня 2021 р.

Календарний план

| № з/п | Назва етапів виконання дипломної роботи | Термін виконання | Примітка |
|---|---|---|---|
| 1 | Узгодження теми роботи із науковим керівником | 01-15 вересня 2021 р. | Виконано |
| 2 | Огляд опублікованих джерел за тематикою дослідження | Вересень-жовтень 2021 р. | Виконано |
| 3 | Визначення мети роботи та основних задач дослідження | Грудень 2021 р. | Виконано |
| 4 | Вибір та обгрунтування двійкового генератора | Січень 2022 р. | Виконано |
| 5 | Реалізація генератора мовою C++ | Лютий 2022 р. | Виконано |
| 6 | Експериментальна перевірка можливості використання архіваторів для тестування ПВП | Березень 2022 р. | Виконано |
| 7 | Розробка власного формату тестування коротких ПВП | Квітень 2022 р. | Виконано |
| 8 | Оформлення дипломної роботи | 1 травня - 14 червня 2022 р. | Виконано |
| 9 | Підготовка до захисту дипломної роботи | 14-23 червня 2022 р. | Виконано |
| 10 | Захист дипломної роботи | 24 червня 2022 р. | Виконано |

Студент _____ Трет'якова А.О.

Керівник _____ Кудін А.М.

# РЕФЕРАТ

Кваліфікаційна робота містить: 49 стор., 4 рисунки, 4 таблиці, 26 джерел.

Об'єктом дослідження даної роботи є генерація криптографічно стійких псевдовипадкових послідовностей.

Предметом дослідження є методи тестування коротких двійкових послідовностей заснованих на теорії алгоритмічної інформації Колмогорова.

Метою роботи є визначення якості псевдовипадкових генераторів по коротких послідовностях отриманих від них.

В ході роботи проаналізовано сучасні системи тестування псевдовипадкових послідовностей та представлено можливість існування певної системи стискання даних, яку можна використовувати для такого тестування. Обґрунтовано використання лінійного конгруентного генератора для перевірки на ньому працездатності тестів псевдовипадкових послідовностей. Експериментально виділено споміж сучасних алгоритмів архівації даних алгоритм RAR та показано можливість його використання в якості універсального декомпресора. Запропоновано новий формат тестування коротких псевдовипадкових послідовностей.

ТЕСТУВАННЯ ПСЕВДОВИПАДКОВИХ ПОСЛІДОВНОСТЕЙ, АЛГОРИТМІЧНА ІНФОРМАЦІ, КОЛМОГОРІВСЬКА СКЛАДНІСТЬ, УНІВЕРСАЛЬНИЙ ДЕКОМПРЕСОР, RAR

# ABSTRACT

This work contains 49 pages, 4 figures, 4 tables, 26 sources.

The object of the research of this work is the generation of cryptographically stable random sequences.

The subject of research is methods of testing short binary sequences for cryptography.

The aim of this work is determination of quality of random generators on short sequences obtained from them.

In the course of the work the modern testing systems of pseudo-randomones are analyzed and the data of such a state of a certain compressing system that can be used for testing are presented. The use of a linear congruent generator to verify the current operation of pseudo-random sequence tests is substantiated. Among modern algorithms of data archiving the algorithm rar is experimentally allocated and the possibility of its use as a universal decompressor has been shown. A new format for testing short random sequences has been proposed.

PSEUDO-RANDOM SEQUENCE TESTING, ALGORITHMIC INFORMATION, KOLMOGOROV COMPLEXITY, UNIVERSAL DECOMPRESSOR, RAR

# ЗМІСТ

# ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СКОРОЧЕНЬ І ТЕРМІНІВ

ВП — випадкова послідовність

ПВГ — псевдовипадковий генератор

КВП — квазівипадкова послідовність

ПВП — псевдовипадкова послідовність

NIST — National Institute of Standards and Technology

NIST STS — NIST Statistical Test Suite

ЛКГ — лінійний конгруентний генератор

ЛКП — лінійна конгруентна послідовність

ЛКГ1 — перший програмно реалізований в ході дослідження лінійний конгруентний генератор

ЛКГ2 — другий програмно реалізований в ході дослідження лінійний конгруентний генератор

BWT — Burrows-Wheeler transform (перетворення Берроуза-Вілера)

PPM — Prediction by Partial Matching (передбачення по частковому збігу)

# ВСТУП

В сучасній криптографії все ще активно обговорюється питання стійкості псевдовипадкових послідовностей, які використовуються повсемісно. Поява нових генераторів таких послідовностей, або ж знаходження слабких місць у старих широко розповсюджених, це все є питаннями тестування послідовностей.

**Актуальність дослідження.** Особливо недовершеним наразі є напрямок перевірки коротких послідовностей, адже більшість статистичних тестів дають адекватні результати тільки на великих об'ємах.

Також нині досить реалістичним виглядає практичне застосування теорії алгоритмічної інформації Колмогорова, адже з часів її формулювання технології стискання даних дуже розвинулись.

Таким чином висвітлюється можливість використати потужні надбання напрямку зберігання даних для вирішення актуальних криптографічних проблем.

**Метою дослідження** є визначення якості псевдовипадкових генераторів по коротких послідовностях отриманих від них. Для досягнення мети необхідно розв'язати **задачу дослідження**, яка полягає у розробці тесту на випадковість коротких послідовностей із використанням теорії алгоритмічної інформації Колмогорова. Для розв'язання задачі необхідно вирішити такі завдання:

1) провести аналіз актуальних теоретичних джерел пов'язаних з тематикою роботи;

2) виконати аналіз сучасних рішень схожих проблем;

3) обґрунтувати використання лінійного конгруентного генератора псевдовипадкових бітових послідовностей;

4) виконати програмну реалізацію лінійного конгруентного генератора;

5) експериментально перевірити можливість використання алгоритмів стискання даних для тестування псевдовипадкових послідовностей;

6) сформулювати формат тестування коротких бітових послідовностей.

*Об'єктом дослідження* є створення криптографічно стійких псевдовипадкових послідовностей бітів.

*Предметом дослідження* є методи тестування коротких двійкових послідовностей заснованих на теорії алгоритмічної інформації Колмогорова.

При розв'язанні поставлених завдань використовувались такі *методи дослідження*: методи теорії імовірностей та математичної статистики, методи теорії алгоритмічної інформації та складності Колмогорова, програмування мовою C++ та програмні пакети статистичних тестів для послідовностей.

**Наукова новизна** отриманих результатів полягає у використаній теорії алгоритмічної інформації для практичного тестування коротких псевдовипадкових послідовностей, яке є ефективнішими за існуючі тести.

**Практичне значення** результатів полягає у тому, що з'являється можливість оцінювати якість джерела псевдовипадкових бітів в умовах, в яких отримання довгих зразків послідовності є ускладненим або зовсім неможливим. З іншої сторони можна скоротити тривалість самого тестування в першу чергу за рахунок використання коротших зразків послідовностей.

# 1 ОГЛЯД ЗАДАЧІ ТЕСТУВАННЯ ПОСЛІДОВНОСТЕЙ НА ВИПАДКОВІСТЬ

Одною із задач, які необхідно вирішувати при побудові стійких криптосистем, є генерація випадкових та псевдовипадкових чисел. Для цього існує багато наявних та з'являються нові генератори. Вихідні дані таких генераторів можуть використовуватись у багатьох криптографічних системах, наприклад, при генерації ключових даних. Послідовності, які є результатом роботи таких генераторів мають задовольняти низку жорстких вимог для того, щоб бути непрогнозованими.

## 1.1 Погляд на числові послідовності через призму тестів

Для перевірки виконання таких вимог у нових генераторів перед використанням, а також для криптоаналізу та побудови атак на вже впроваджених в криптосистемах, використовують різноманітні тести на випадковість. Будь-яку послідовність чисел чи символів можна розглядати як вибірку випадкової величини в термінах математичної статистики.

Задача формулюється наступним чином: послідовність випадкових величин $\xi$, яка може набувати лише двох значень «0» та «1» має деякий розподіл ймовірностей P. Розмір вибірки для тестування складає n бінарних символів. Тоді необхідно визначити чи розглянута реалізація випадкової величини $xi$ має рівномірний розподіл на множині $X = \{0,1\}^n$.

При розв'язанні наведеної задачі випливає означення ідеальної випадкової послідовності, для якої визначають тільки такі властивості, які потрібні для подальшого зручного опису та дослідження послідовностей.

**Означення 1.1.** «ідеальної» [1](в криптографії) випадкова послідовність являє собою реалізацію послідовності незалежних випадкових величин $\xi_1,...,\xi_n$ , які мають рівномірний розподіл ймовірностей на заданому скінченному алфавіті символів $X = x_1,...,x_n$.

Ідеальна випадкова послідовність повинна задовольняти дві базові вимоги [3]:

1) для будь–якого $n$ та довільних значень індексів $1 \leqslant t_1 < ... < t_n$ випадкові величини $\xi_{t1},...,\xi_{tn}$ незалежні;

2) для будь-якого натурального $t$ випадкова величина $\xi_t$ рівномірно розподілена на множині $X$ іншими словами: $P(\xi_t = x) = 1/n, \forall x \in X$, $n$–кількість елементів у множині $X$.

При виконанні цих базових вимог виконується ряд інших властивостей:

1) для будь-якого n та довільних значень індексів $1 \leqslant t_1 < ... < t_n$ маємо рівномірний розподіл випадкових величин $(\xi_{t1},...,\xi_{tn})$ на множині $X^n$;

2) для будь-якої підпослідовності $1 \leqslant t_1 < ... < t_n$ натуральних чисел відповідна підпослідовність $\xi_{t1},...,\xi_{tn}$ послідовності $\xi_t$ також рівномірно розподілена випадкова послідовність, у наших визначеннях ідеальна;

3) якщо X – адитивна група та $pi_t$ – довільна детермінована послідовність або довільна випадкова послідовність над множиною X, яка не залежить від $\xi_t$, то випадкова послідовність $y_t$, де $y_t = xi_t + pi_t$, також рівномірно розподілена

4) при будь-якому натуральному t передбачення значення $\xi_t$, за знаками $\xi_1,...,\xi_t$ неможливо, тобто:

$$P(\xi_t = x_t | \xi_1 = x_1,\ldots,\xi_{t-1} = x_{t-1}) = P(\xi_t = x) = 1/n,$$

для будь-якого набору $(x_1,\ldots,x_t) \in X^t$.

Реальні випадкові послідовності в наш час генеруються за

допомогою кристалічних датчиків [4, 5], в яких за вихідний сигнал відповідають дійсно випадкові фізичні процеси, або, наприклад, на основі дезоксирибонуклеїнових кислот [6].

Умовно штучне отримання подібних послідовностей, за допомогою програмної реалізації на електронно обчислювальних машинах зветься псевдовипадковою [2]. Насправді вони є детермінованими, але їх задум імітувати випадкові.

**Означення 1.2.** Псевдовипадковим генератором назвемо функцію поліноміальної складності $G : \{0,1\}^l \to \{0,1\}^\infty$,таку що для деякого початкового значення довжини $n$ входу $G$ генерує послідовність довжини $g_k(n)$, де $g_k(n)$ деякий поліном степеня $k$.

Якість подібних ПВГ перевіряється різноманітними тестами, що певним чином кількісно або якісно оцінюють співвідношення між властивостями наявної вихідної послідовності з властивостями реальної випадкової послідовності. Такі тести бувають різними.

**Означення 1.3.** Функція виду $f : \{0,1\}^* \to [0,1]$ називаємо функціонально статистичним тестом, при цьому зауважимо, що $\mu_f(n) = 1/2^n \sum\limits_{|x|=n} f(x)$ – середнє значення функції $f$ при випадковому вході довжиною $n$, а $\mu_f{}^*(n) = \sum\limits_{|x|=n} p_n(x) f(x)$ – математичне очікування $f$ при вхідних значеннях з генератора довжиною n. Порівняння цих величин і буде тестуванням.

Якщо псевдовипадкову послідовність не можна ймовірнісно відрізнити від реальної випадкової, то її та її генератор називають квазівипадковими.

**Означення 1.4.** Генератор вважається квазівипадковим, якщо для всіх $t > 0$ існує достатньо велике n, що для будь-якого функціонального статистичного тесту f:

$$|\mu_f(n) - \mu_f{}^*(n)| \leqslant 1/n^t,$$

еквівалентним є виконування наступної властивості:

$$|\sum_{|x|=n} P_n(x) - (1/2)^n| < 1/n^t,$$

для будь-якого $t$ та для достатньо великого $n$.

Окрім функціональних тестів виділяються також ймовірнісні статистичні тести поліноміальної складності, адже не всі КВП проходять їх.

**Означення 1.5.** T буде ймовірнісним статистичним тестом поліноміального виду, коли $T : \{0,1\}^* \to \{0,1\}$, яка є обчислюваною на ймовірнісній поліноміальній машині Тьюринга. Тестуванням буде порівняння величин $\mu_T(g(n))$ та $\mu_T{}^*(G(n))$, де g - випадковий генератор, G - досліджуваний псевдовипадковий генератор.

**Означення 1.6.** Генератор називають «ідеальним» псевдовипадковим генератором, якщо для всіх $t > 0$ існує достатньо велике n, що для будь-якого ймовірнісного тесту поліноміальної складності $T$:

$$|\mu_T(g(n)) - \mu_T{}^*(G(n))| \leqslant 1/n^t$$

.

Тобто, якщо не існує такого ймовірнісного статистичного тесту, який відхилить даний псевдовипадковий генератор, то він є «ідеальним».

Квазівипадковий та «ідеальний» ПВГ відрізняються тим, що перший спирається на теорію інформації, а другий на теорію складності.

## 1.2 Сучасне тестування псевдовипадкових послідовностей

Можна виділити 2 узагальнені підходи аналізу ПВП [4]:

–статистичний — цей підхід орієнтований на пошук відхилень статистичних властивостей псевдовипадкової послідовності, на основі

яких можна передбачати наступні та попередні значення членів послідовності з ймовірністю більшою 0,5.

–криптографічний, ціль цього підходу — пошук таких закономірностей досліджуваної послідовності, щоб по її частині можна було відновити всю послідовність цілком.

Строгий доказ непередбачуваності генераторів псевдовипадкових послідовностей — складна проблема, яка досі не вирішена. В разі криптографічно стійких генераторів псевдовипадкових послідовностей зазвичай вважається, що визначення наступного біта вироблюваної послідовності з ймовірністю більше 0,5 у противника не буде достатньо ресурсів (таких, як часу, обчислювальних ресурсів, а також матеріальних коштів).

Існують теоретичні методи дослідження псевдовипадкових послідовностей, які використовують різні підходи до визначення випадковості. До них належать, наприклад, частотний підхід (вперше запропоновано фон Мізесом) [7], алгоритмічний підхід Мартін-Лефа, а також складнісний підхід Колмогорова [8, 9].

Популярними і потужними на даний час є наступні програмні пакети з тестами для перевірки ПВП [10]:

1) NIST Statistical Test Suite [11];

2) NESSIE [12];

2) TEST-U01 [13];

2) CRYPT-X [14];

3) The pLab Project [15];

4) DIEHARD [16];

5) ENT [17];

6) Dieharder [18].

Тести Crypt-X - це набір статистичних тестів розроблений дослідниками з науково-дослідного центру з інформаційної безпеки в технологічному університеті Квінсленду в Австралії і є комерційним пакетом програмного забезпечення. Тести застосовуються залежно від

типу алгоритму генератора, відповідно спрямовані на тестування генераторів псевдовипадкових чисел. Підтримуються потокові шифри, блокові шифри та генератори потоку ключів. У набір включені такі тести: частотний, на послідовність однакових бітів, лінійна складність, склдність послідовності, двійкова похідна, зміна точки.

Стандарти та тести NIST: у США було зроблено перший крок до стандартизації набору статистичних тестів шляхом прийняття у 1994 р. національного стандарту «Вимоги безпеки до криптографічних модулів» [19]. Проте вимоги та методика стандарту більше мають технологічний характер. Вони спрямовані на вирішення задачі статистичного контролю псевдовипадкових послідовностей, що використовуються в криптографічних модулях, та загалом малопридатні для вирішення завдання дослідження статистичних властивостей генераторів.

У 1999 р. спеціалістами NIST (Національний інститут стандартів та технологій (НІСТ), США), в рамках проєкту AES (Advanced Encryption Standard) був розроблений набір статистичних тестів «NIST STS» (NIST Statistical Test Suite) та запропонована методика проведення статистичного тестування ВП(ПВП).

**Твердження 1.1.** *Методика NIST STS є орієнтованою на використання в криптографічних задачах захисту інформації та на думку багатьох фахівців у цій галузі, на даний момент найкраще відповідає потребам усіх зацікавлених сторін.*

Пакет NIST STS включає 15 статистичних тестів, які розроблені для перевірки гіпотези про випадковість двійкових послідовностей довільної довжини, що породжуються генераторами ПВП. Усі тести спрямовані на виявлення різноманітних дефектів випадковості. Нижче наведено список тестів та визначається дефект:

1) частотний тест — занадто багато нулів або одиниць;

2) частотний тест у блоках — занадто багато нулів або одиниць у блоках;

3) перевірка кумулятивних сум — занадто багато нулів або одиниць на початку послідовності;

4) перевірка «дірок» у підпослідовності — відхилення у розподілі послідовностей одиниць;

5) перевірка «дірок» — велике (мале) число підпослідовностей нулів та одиниць свідчить, що коливання потоку бітів надто швидке (повільне);

6) перевірка рангів матриць — відхилення розподілу рангів матриць від відповідного розподілу для істинно випадкової послідовності, пов'язаний з періодичністю послідовностей;

7) перевірка шаблонів, що не перетинаються — неперіодичні шаблони зустрічаються дуже часто;

8) перевірка шаблонів, що перетинаються — занадто часто зустрічаються m-бітні послідовності одиниць;

9) універсальний статистичний тест Маурера — регулярність послідовності;

10) перевірка випадкових відхилень — відхилення від розподілу числа появ підпослідовностей певного виду;

11) різновид перевірки випадкових відхилень — відхилення від розподілу числа появ підпослідовностей певного виду;

12) перевірка апроксимованої ентропії — нерівномірність розподілу m-бітних слів. Малі значення означають високу повторюваність;

13) перевірка серій — нерівномірність розподілу m-бітних слів;

14) лінійна складність — відхилення від розподілу лінійної складності для кінцевої довжини підлаштування;

15) дискретне перетворення Фур'є — пошук повторюваних шаблонів.

## 1.3 Колмогорівський підхід до інформації та складності

Для опису теорії алгоритмічної інформації бітових послідовностей доцільно навести опорні теореми та визначення.

Основними поняттями в цій теорії є терміни декомпресор і складність

слова [20, 21], **??**. Словом називаємо двійкову послідовність.

**Означення 1.7.** Декомпресор (спосіб опису) – це певне відображення $D : 0{,}1^{->0{,}1}$ , яке є обчислюваним за допомогою алгоритму. Цей алгоритм можна застосувати до бітових слів, які входять в область визначення D і лише до них.

**Означення 1.8.** Колмогорівська складність бітового слова X відносно певного способу опису D – це така величина:

$$KS_D(x) = \min\{l(y) \mid D(x) = y\}$$

де $l(y)$ є довжиною слова $y$. Вважається, що $min(\varnothing) = +\infty$

Для того, щоб більше узагальнити поняття складності бітових слів вводиться термін оптимальний декомпресор, який є певним універсальним описовим способом.

**Означення 1.9.** Вважається, що декомпресор $D_1$ є не гіршим за декомпресор $D_2$, якщо існує константа  така, що:

$$KS_{D_1}(x) \leqslant KS_{D_2}(x) + c$$

для всіх X.

На основі можливості порівняння двох декомпресорів вводиться поняття оптимального декомпресора.

**Означення 1.10.** Декомпресор $D_0$ є оптимальним, якщо він є не гіршим за будь-який декомпресор $D_r$.

За допомогою наступної теореми вибудовується однозначне визначення Колмогорівської складності.

**Теорема 1.1.** . *(Соломонова-Колмогорова) Існує такий оптимальний декомпресор D, що для будь-якого іншого декомпресора D існує така константа c, що для будь-якого бітового слова  маємо:*

$$KS_D(x) \leqslant KS_{D^{\#}}(x) + c$$

Щоб позбутися впливу константи на результати у практиці вводиться наступне твердження: Нехай декомпресору $D^{\#}$ відповідає програма , тоді:

$$KS_D(x) \leqslant KS_{D^{\#}}(x) + l(p)$$

Тепер стає можливим зафіксувати деякий оптимальний декомпресор D та визначати складність бітового слова лише відносно $D$. При цьому позначаючи $KS(x)$.

Ознайомившись з опорними поняттями проводиться кількісне оцінювання складності Колмогорова слів певної фіксованої довжини.

**Означення 1.11.** Множина, яка перераховується – це множина така, що існує алгоритм, яким породжуються усі елементи цієї множини.

За останнім означенням можна сказати, що сімейство множин $V_n$ це сімейство множин, що перераховується, за умови, що множина $\{< n, x > |\ x \in V_n\}$ є множиною, що перераховується.

**Теорема 1.2.** *Множини* $S_n = \{x \mid KS(x) < n\}$ *утворюють сімейство, що перераховується, при цьому* $|S_n| < 2^n$ *при усіх* $n$.

**Доведення.** Нехай D це фіксований оптимальний декомпресор. Тоді при $l(y) < n$ всі слова вигляду $D(y)$ і лише вони будуть мати складність меншу ніж n. Кількість таких слів є не більшою, ніж кількість різних y, довжина яких менше n, яку можна порахувати:

$$1 + 2 + \cdots + 2^{n-1} = 2^n - 1$$

теорему доведено. □

З доведення цієї теореми 1.3 випливає те, що серед усіх можливих слів, ті що мають складність нижче $n - c$ складають частку меншу $2^{-}$. Це означає, що переважну більшість послідовностей суттєво стиснути не можна.

**Теорема 1.3.** *Нехай* $V_{n,n} = 0,1,2,\ldots$ - *сімейство, що перераховуються, при цьому* $|V_n| < 2^n$ *при всіх* $n$, *тоді знайдеться*

*константа  така, що* $KS(x) < n + c$ *при будь-яких* $n$ *та*  $\in V_n$

На основі двох останніх теорем можна говорити про існування слів зі складністю рівною їх довжині у множині усіх слів які мають довжину n.

Зробивши узагальнення всього вище наведеного стає можливим визначення Колмогорівської складності слова  як кількості алгоритмічної інформації слова . Це твердження основане на інтуїтивному розумінні кількості інформації, яке ґрунтується на тому, що описове подання певного об'єкта має деяку складність. Якщо слово можна стиснути, то кількість інформації, яку воно несе, не більша за кількість інформації, яку несе його стиснутий варіант. Звідси зрозуміло, що слова, які не можна значно стиснути або не можна стиснути взагалі, більш інформативні. Очевидно, що кількість інформації, яку несе слово, не може перевищувати довжину цього слова.

**Теорема 1.4.** *Існує константа с така, що при будь-якому слові маємо:*

$$KS(x) \leqslant l(x) + \mathrm{c}$$

До того, як Колмогоров визначив алгоритмічний підхід, вже була давно відкритою задача знаходження кількості інформації [22]. Колмогоров описав три різнопланові підходи в розв'язанні цієї задачі. Відмінність поняття алгоритмічної інформації від інформації Шеннона полягає у тому, що в його основі лежить поняття про складність об'єкта, а не про його ентропію. Кількість алгоритмічної інформації не може збільшуватись в результаті певного алгоритмічного перетворення.

**Теорема 1.5.** *Для кожного алгоритму A знайдеться константа с така, що при всіх , для яких визначено* () *виконується:*

$$KS(\ \mathrm{A}(x)) \leqslant KS(x) + \mathrm{c}$$

Для вирішення задачі розрізнення послідовностей різної складності наведено поняття простих слів.

**Означення 1.12.** Слово є простим, якщо $KS(x) < l(x)/2$ .

Для визначення поняття випадковості в теорії алгоритмічної інформації вводиться означення дефекту, що характеризує послідовність як невипадкову.

**Означення 1.13.** Дефект випадковості бітового слова – це величина: $d(x) = l(x) - KS(x)$

З теореми 1.13 та її доведення слідує, що величина ймовірності отримати дефект, більший за $h$, для довільно вибраної послідовності певної довжини, не перевищує $2^{-h}$.

**Теорема 1.6.** *Існує така константа c [21], що для кожної скінченної послідовності , яка має довжину n, та містить в собі k одиниць, справедлива наступна нерівність:*

$$\left|\frac{k}{n} - \frac{1}{2}\right| \leqslant c\sqrt{\frac{\log_2 n + d(x)}{n}}$$

З даної теореми випливає те, що частота одиниць в послідовностях, які ніяк не стискаються, близька до $1/2$, що справджує їх випадковість. З усього вищенаведеного випливає, що стиснення випадкової бітової послідовності неможливе.

## 1.4 Універсальний декомпресор

У рівномірно випадкових двійкових словах $a_0, a_1, ..., a_n$ мають задовольнятись певні статистичні ознаки [23]. Наприклад, одиниця має зустрічатися приблизно $n/2$ разів, а біграми 11, 01, 10, 00 приблизно n/4 разів і так далі. Очевидно, що у представленого означеного рівномірно випадкового слова виконаються всі статистичні ознаки одразу. Краще розглянути певний універсальний тест, який тестуватиме всі одразу статистичні ознаки слова, таким чином міститиме кожен з можливих тестів у собі. У разі існування такого тесту, можливим стане визначення

результатів основних питань теорії ймовірності, які йдуть з визначення випадкового слова. Прикладом можуть слугувати центральна гранична теорема та закон великих чисел.

Розглянемо формулювання статистичних тестів, яке з'явилось у матстатистиці. Статистичний тест перевіряє наявність певної ознаки, що обов'язково зустрічається у випадковому слові. Значущість показань кожної такої перевірки визначається ймовірністю вірності припущення про наявність такої ознаки. Формально тест визначається певною функцією $f(m,n)$.

Для такої функції визначається наступна нерівність:

$$|S_n - n| > f(m,n)$$

$f$ повинна задовольняти той факт, що не повинно існувати більше $2^{n-m}$ слів, у яких $n$ символів, та для яких ця нерівність не виконується. Надалі ця функція є не зменшуваною за константності цієї умови.

Підсумовуючи, тест приймає тільки ті двійкові послідовності, які влаштовують гіпотезу з певною значущістю. Візьмемо похибку $e = 2^{-m}, m = 1,2,3...$, тепер визначимо множину тестів наступним чином:

$$U \subseteq N \times X,$$

де $N$ — множина натуральних чисел, $X$ — множина двійкових слів, що задовольняють похибку.

$$U_m = \{x : (m,x) \in U\};$$
$$U_{m+1} \subseteq U_m, m = 1,2,...$$

Обмеження накладені на $U_m$ з похибкою $e = 2^{-m}$, забезпечується те, що у множині $X$ не більше $2^{n-m}$ слів, у яких $n$ символів.

U можна визначити в поняттях рекурентно перелічуваних множин, вона задається за допомогою рекурсивно перерахованого набору

послідовності ключових областей, визначених похибкою. Надалі, якщо мова йтиме про тестування, то слід розуміти означення шляхом рекурсивного перерахування наборів, заданих сімействами критичних областей, що входять у рівень значущості. Припускаючи подібне, узагальнюється концепцію тестування і будується універсальний формат тестування. При цьому виконуватиметься, якщо після універсального тестування, двійкове слово нарікається рівномірно випадковим, то будь-який інакший тест також покаже, що дане слово є рівномірно випадковим. тобто універсальний формат тестування. ігнорує рівень значущість.

**Теорема 1.7.** *Відповідно кожному тесту $V$ знайдеться такий тест $U$, що*

$$V_{m+c} \subseteq U_m, m = 1,2,...,$$

*де $C$ - залежна від $U$ і $V$ константа.*

Ця теорема доводиться через ефективну перелічуваність множини $U$.

В реальності тести на випадковість працюють з граничним значення, яке показує мінімальний рівень похибки, за якого відхиляється гіпотеза про рівномірну випадковість. При тому, що в вищенаведеному випадку йде мовна про використання параметра $m$, як альтернативи $e = 2^{-m}$, то представляється наступна формула:

$$M_U(x) = max_{x \in U_m}(m)$$

, де індексом $U$ зазначається залежність від реального тесту.

Відповідно до наведеного формулювання $m_U(x)$, яке є справедливим для будь-якого $x$, $U_0$ - це множина всіх послідовностей, що задовольняють умову, що є дійсною для будь-якого :

$$0 \leqslant m_U(x) \leqslant l(x),$$

де $l(x)$ - довжина послідовності $x$.

Існування універсального формату тестування через пороговий рівень формулюється наступним чином: Існує універсальний тест U такий, що для кожного тесту $V$ знайдеться константа $c$, що задовольняє наступну умову для будь-якого двійкового слова $x$:

$$m_V(x) \leqslant m_U(x) + c,$$

Порогову величину для $x$ для певного універсального формату тестування називатимемо критичним значенням та записуватимемо $m(x)$.

Наступна теорема ілюструє зв'язок цієї порогової величини з величиною Колмогорівської складності слова.

**Теорема 1.8.** *Для будь-якого двійкового слова знайдеться константа c, що задовольнятиме умову:*

$$|l(x) - K(x|l(x)) - m(x)| \leqslant c,$$

Використовуючи означення тесту описане вище, для універасльного формату тестування здійснюватиметься умова:

$$|2S_n - n| \leqslant f(m(a_0,...,a_n) + c, n),$$

для будь-якого двійкового слова, що інакше:

$$|2S_n - n| \leqslant f(n - K(a_0,...,a_n|n) + c, n)$$

Використовуючи функцію Лапласа для приближення, величина виразу $|2S_n - n|$ набуватиме порядку $\sqrt{n}$, якщо $K(a_0,...,a_n|n) \approx n$

Відповідно до наведеного опису, універсальний формат тестування умовно об'єднуватиме всі можливі статистичні тестування та являтиметься не гіршим від них.

## Висновки до розділу 1

В хоті огляду наукових джерел щодо поставленої задачі в першу чергу сформульовано поняттєву базу для подальшого адекватного викладу дослідження.

На основі опрацьованої теоретичної інформації виводиться те, що існують такі псевдовипадкові послідовності, які можуть пройти всі статистичні тести, але при цьому будуть визначені як невипадкові тести, що ґрунтуються на теорії алгоритмічної інформації.

Також визначено, якими інструментами на даний момент вирішується задача тестування ПВП. В якості основного пакета тестів, на який слід рівнятись визначено NIST STS.

Виділено питання оптимального декомпресора за Колмогоровим теоретичне існування якого доведено, проте практичного повноцінного екземпляра поки немає. Таким чином задача перевірки можливості використання сучасних існуючих алгоритмів стискання даних для тестування ПВП залишається відкритою.

# 2 ГЕНЕРАЦІЯ ПСЕВДОВИПАДКОВИХ ПОСЛІДОВНОСТЕЙ

Для побудови тесту для ПВП є необхідною наявність опорного джерела цих послідовностей, аби перевіряти на ньому працездатність тесту. Детальна теоретична інформація про використовуваний в таких цілях двійковий генератор допомагає краще інтерпретувати показники розроблюваного тесту. Також подібне джерело спочатку оцінюється за допомогою існуючих авторитетних інструментів.

## 2.1 Вивчення вимог до досліджуваних послідовностей в сучасних пакетах тестів

Спираючись на твердження 1.1 для попереднього тестування опорної послідовності обрано пакет NIST STS. Він є найпотужнішим та найбільш популярним методом тестування ПВП та відповідно якості їх генераторів на даний момент.

В контексті досліджуваної теми слід проаналізувати обмеження довжин послідовностей, які можна ефективно тестувати за допомогою пакету NIST STS. В таблиці 2.1 наведені ці мінімальні параметри.

**Таблиця 2.1** – Обмеження на довжину послідовностей у NIST STS

| Назва тесту | К-сть потоків | Бітів у потоці | Додаткові параметри |
|---|---|---|---|
| 1 | 2 | 3 | 4 |
| Частотний тест | 100 | 100 | - |
| Частотний тест у блоках | 100 | 100 | M = 20 |
| перевірка «дірок» | 100 | 100 | M = 8 |

Продовження таблиці 2.1

| 1 | 2 | 3 | 4 |
|---|---|---|---|
| перевірка «дірок» у підпослідовності | 100 | 128 | M = Q = 32 |
| перевірка рангів матриць | 100 | 38912 | - |
| дискретне перетворення Фур'є | 100 | 1000000 | m = 9 |
| перевірка шаблонів, що не перетинаються | 100 | 1000000 | m = 10 |
| перевірка шаблонів, що перетинаються | 100 | 1000000 | L = 6, Q = 640 |
| універсальний статистичний тест Маурера | 100 | 387840 | M = 500, N = 200 |
| лінійна складність | 100 | 1000000 | m = 2 |
| перевірка серій | 100 | 1000000 | m = 2 |
| перевірка апроксимованої ентропії | 100 | 100 | - |
| Перевірка кумулятивних сум | 100 | 100 | - |
| перевірка випадкових відхилень | 100 | 1000000 | - |
| різновид перевірки випадкових відхилень | 100 | 1000000 | - |

Видно, що тільки шість з п'ятнадцяти методів є придатними для використання на коротких бітових послідовностях. З цього слідує наступне зауваження:

**Твердження 2.1.** *Комплекс тестів NIST STS не може дати повної картини про якість використовуваного генератора в умовах, коли генерація довгих послідовностей дуже затратна по часу, або їх отримання взагалі є неможливим.*

Попри це він все ще має бути достатньо ефективним для оцінки опорного генератора і саме з результатами тестування NIST STS слід порівнювати показник розроблюваного тесту, аби підкреслити його актуальність на даний час.

Також слід зазначити, що в огляді згадуються різні популярні і потужні групи тестів, які використовуються в наш час, такі як:

- NESSIE;
- Diehard;
- Dieharder.

Найменша послідовність, з якою можуть працювати тести NESSIE (при чому не всі) це 50 тисяч біт [23], що є хоч і далеко не найвищим обмеженням серед пакетів тестів, що розглядаються, але все рівно є потреба у тестуванні значно коротших ПВП.

Для тестів Diehard безпечним мінімумом для роботи є послідовності у декілька мільйонів двійкових символів (мінімум 2) [24]. А документація пакета Dieharder [18] взагалі акцентує велику увагу на тому, що даний пакет відрізняється від Diehard великою мірою тим, що обробляє значно довші зразки послідовностей і початковою величиною є послідовності у 10 мільйонів 32–бітних чисел, а нормою є послідовності у $10^{18}$ 32–бітних чисел.

Таким чином вищеперераховані тести погано співвідносяться з проблематикою тестування коротких двійкових послідовностей і є по суті непридатними для них.

## 2.2 Постановка задачі

В криптографії та криптоаналітиці зустрічається проблема швидкого тестування ПВГ в умовах, коли генерація довгих зразків послідовності є дуже затратною по часу, або взагалі є неможливою, як, наприклад, при атаці на короткі ключі в системах з багатофакторною аутентифікацією.

Таким чином виникає проблема тестування на рівномірну

випадковість коротких ПВП. Як зазначається вище, дана проблема вирішена сучасними система тестування лише частково.

З неповноцінності сучасних тестових систем випливає те, що є такі завідома непридатні до використання у криптографії псевдовипадкові генератори, які при проходженні сучасних тестових систем в умовах коротких досліджуваних послідовностей будуть прийматись такими системами, як адекватні та рівномірно незалежно випадкові.

Також опираючись на огляд, який наведено у цій роботі, а саме ту його частину, яка стосується теорії алгоритмічної інформації та Колмогорівської складності, можна стверджувати, що існує така система стискання даних, яка є не гіршою за інші і виступає універсальним декомпресором. Таку систему можна використовувати в якості універсального тесту на випадковість.

З моменту формулювання теорії універсального декомпресора до сьогодення технології систем стискання даних дуже розвинулись, з чого можна зробити припущення, що серед сучасних алгоритмів архівації даних вже може бути такий алгоритм, який можна з тим чи іншим припуском використовувати в якості наближення до ідеального декомпресора і відповідно тестувати на ньому ПВП.

З вищезазначеного формулюється задача, використовуючи певний опорний генератор псевдовипадкових чисел експериментально перевірити можливість використання сучасних алгоритмів архівування даних для стискання та тестування коротких ПВП. Знайшовши такий алгоритм, запропонувати новий формат перевірки якості ПВГ опираючись на згенеровані ними короткі ПВП.

## 2.3 Вибір та опис генератора

В якості опорного джерела послідовностей обрано лінійний конгруентний генератор, який визначається наступним чином:

**Означення 2.1.** Лінійний конгруентний генератор(ЛКГ) — це апаратний генератор псевдовипадкових послідовностей, що рекурентно обраховує та видає на вихід лінійну конгруентну послідовність.

**Означення 2.2.** Лінійна конгруентна послідовність(ЛКП) — це послідовність чисел, яка задається наступними параметрами:

$$m, \text{ модуль; } 0 < m;$$
$$a, \text{ множник; } 0 \leqslant a < m;$$
$$c, \text{ приріст; } 0 \leqslant c < m;$$
$$X_0, \text{ початкове значення; } 0 \leqslant X_0 < m$$

та обчислюється наступною рекурентною формулою:

$$X_{n+1} = (aX_n + c) \bmod m, \quad n \geqslant 0$$

Причиною вибору генератора саме цього виду є те, що як на достатніх об'ємах, так і на коротких послідовностях він може показувати гарні статистичні показники, але при цьому він є завідома неідеальним та передбачуваним і детекцію саме цієї неідеальності повинен виконувати розроблюваний тест, що ґрунтується на стисканні послідовності.

Максимальна довжина періоду послідовності для ЛКГ рівна модулю $m$. Для хороших статистичних показників є сенс використовувати саме генератор з максимальним періодом, адже в інакшому випадку за рахунок відсіяння частини елементів з'явиться додаткова нерівномірність послідовності.

Для зручності програмної реалізації за модуль обрано $2^{32}$.

Підбирати інші параметри генерації слід опираючись на наступну теорему:

**Теорема 2.1.** *Лінійна конгруентна послідовність, що визначається числами $m$, $a$, $c$ та $X_0$, має період довжиною $m$ тоді і тільки тоді, коли:*

*1) числа $c$ та $m$ взаємно прості;*

*2) $b = a - 1$ кратно $p$ для кожного простого $p$, який є дільником $m$;*

*3) $b$ кратно 4, якщо $m$ кратно 4.*

Доведення теореми 2.1 наведено у роботі [7].

Спираючись на цю теорему обрано параметри для двох генераторів:

перший генератор:

m = 4294967296, a = 65537, c = 119

другий генератор:

m = 4294967296, a = 9737, c = 23209

Виконано програмну реалізацію даних генераторів мовою C++ у вигляді консольного додатка з використанням середовища розробки Microsoft Visual Studio 2022. Код основних файлів програми наведено у додатку 3.2.

Надалі у роботі під словами перший генератор та другий генератор, а також під абревіатурами ЛКГ1 та ЛКГ2 слід розуміти саме реалізовані в ході дослідження ЛКГ-ри з наведеними тут параметрами.

Згенеровані послідовності виводяться до консолі у шістнадцятковій формі, що показано у тестовому запуску програми на рисунку 2.1.

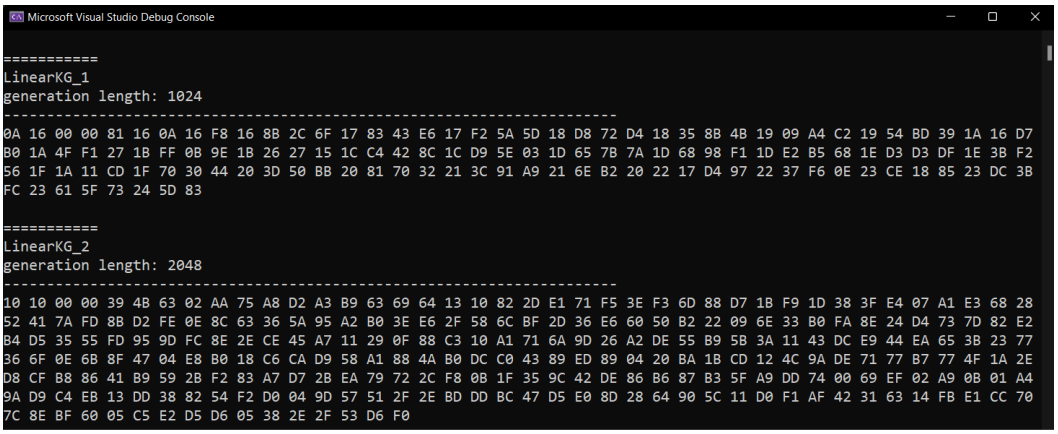


**Рисунок 2.1** – Ілюстрація тестового запуску реалізованих генераторів

Також результуючі послідовності записуються у файли, що дозволяє подальші маніпуляції з ними за допомогою архіваторного ПЗ та інших сторонніх програм. Обсяги та параметри генерації, а також адреса для зберігання результатів встановлюються шляхом редагування коду програми. Обсяги генерації необмежені, час генерації 400 мільйонів бітів на комп'ютері з процесором Intel(R) Core(TM) i5-1135G7 та операційною

системо Windows 11 менше 4-х секунд.

## 2.4 Тестування опорного генератора

Перед використанням в якості опорних генераторів для розробки тесту проведено тестування та аналіз ЛКГ1 та ЛКГ2.

Для цього використовується пакет NIST STS. Цей пакет тестів для запуску на операційній системи Linux є у вільному доступі на офіційному сайті NIST [11], також там знаходиться детальна документація з інструкціями по використанню [25].

Консольний інтерфейс, який наданий NIST є достатньо зручним у використанні, першим кроком вказується кількість бітів одного потоку досліджуваної послідовності. Далі вказується шлях до досліджуваного файлу та визначається, які саме з 15ти тестів треба застосувати до послідовності. Підтримується два формати файлів: файли закодовані форматом ASCII та прості бінарні файли, в яких наступний потік записується одразу без розділителів за попереднім. Реалізовний генератор записує послідовності саме у простому бінарному вигляді одну за одною.

Проаналізувавши вимоги до довжин послідовностей у NIST STS та для всеохоплюючої оцінки даним пакетом було проведено багато варіантів запусків з різними довжинами послідовностей. Найнаочніші з них винесені до таблиць 2.2 та 2.3.

В таблиці 2.2 показані результати тестування 100 потоків бітів по 2097152 біт кожним доступним тестом.

Тобто загалом для даного тестування необхідно було згенерувати більше 400та мільйона бітів. Саме тестування на комп'ютері з процесором Intel(R) Core(TM) i7-4510U та операційною системою Ubuntu 20.04.4 LTS тривало аж 23 хвилини.

В другому та третьому стовпчиках таблиці 2.2 вказані частки потоків відповідно першого та другого генераторів, які пройшли відповідний вказаний у рядку тест успішно.

**Таблиця 2.2** – Результати тестування довгих ПВП з NIST STS

| Назва тесту | Перший генератор | Другий генератор |
|---|---|---|
| Довжина послідовностей | 2097152 біт | 2097152 біт |
| Кількість потоків перевірки | 100 | 100 |
| Частотний тест | 100/100 | 100/100 |
| Частотний тест у блоках | 0/100 | 100/100 |
| перевірка «дірок» | 100/100 | 100/100 |
| перевірка «дірок» у підпослідовності | 99/100 | 99/100 |
| перевірка рангів матриць | 0/100 | 100/100 |
| дискретне перетворення Фур'є | 0/100 | 0/100 |
| перевірка шаблонів, що не перетинаються | 100/100 | 98/100 |
| перевірка шаблонів, що перетинаються | 99/100 | 99/100 |
| універсальний статистичний тест Маурера | 0/100 | 0/100 |
| лінійна складність | 99/100 | 99/100 |
| перевірка серій | 100/100 | 100/100 |
| перевірка апроксимованої ентропії | 100/100 | 100/100 |
| Перевірка кумулятивних сум | 80/100 | 100/100 |
| перевірка випадкових відхилень | 84/100 | 84/100 |
| різновид перевірки випадкових відхилень | 83/100 | 83/100 |

В інструкції [25] зазначено, що для прийняття гіпотези про рівномірну випадковість джерела бітів необхідно, щоб по кожному тесту не менше як 96 потоків його пройшли.

Можна стверджувати, що другий генератор є статистично кращим, ніж перший, адже він показав кращі результати по трьох тестам, попри це все одно знайшлись такі тести, які він зовсім не пройшов, чого достатньо для того, щоб говорити про його неякісність і непридатність для криптографії так, як і першого генератора.

В таблиці 2.3 наведений результат перевірки того, чи здатен пакет

**Таблиця 2.3** – Результати тестування коротких ПВП з NIST STS

| Назва тесту | Перший генератор | Другий генератор |
|---|---|---|
| Довжина послідовностей | 1536 біт | 1536 біт |
| Кількість потоків перевірки | 100 | 100 |
| Частотний тест | 83/100 | 100/100 |
| Частотний тест у блоках | 87/100 | 100/100 |
| Перевірка «дірок» | 78/100 | 98/100 |
| Перевірка «дірок» у підпослідовності | 80/100 | 100/100 |
| Перевірка апроксимованої ентропії | 93/100 | 99/100 |
| Перевірка кумулятивних сум | 80/100 | 100/100 |

тестів NIST STS виявити недосконаліть ЛКГ використовуючи тільки ті свої тести, які працюють на коротких зразках послідовностей. Відбувся запуск шести робочих тестів на 100 потоках по 1536 біти з обох генераторів. Слід зазначити, що результат отримано майже миттєво.

Враховуючи поріг прийняття гіпотези про випадковість на рівні 96 потоків з 100 маємо те, що перший генератор відкидається цим пакетом навіть в режимі коротких послідовностей.

При цьому другий генератор пройшов усі доступні тести з максимальним промахом у 2 потоки на перевірці «дірок», в результаті цього, за умов коротких послідовностей пакет NIST STS приймає його, тобто він не здатен ідентифікувати його неідеальність.

Було підтверджено, що при маленьких довжинах послідовностей пакет NIST STS не є цілком позбавленим свого сенсу, аже ідентифікує недоліки гіршого з двох представлених варіантів ЛКГ, але при цьому і не є достатньо ефективним для реального криптографічного застосування в

подібних умовах, адже завідома неідеальний генератор, працюючий на тій же формулі, що і відкинутий, але з трохи кращими параметрами, був прийнятий цим тестом.

## Висновки до розділу 2

В ході аналізу пакета NIST STS виявлено недолік цього пакету тестів, а саме неповноту його інформативності при використанні для перевірки коротких бітових послідовностей.

При цьому даний програмний пакет визнано придатним для попередньої оцінки якості генератора, який буде використовуватись для формування тесту на випадковість.

В якості опорного джерела обрано лінійний конгруентний генератор. Він проявляє достатньо непогані статистичні характеристики, але при цьому є завідома неідеальним, передбачуваним. Саме такі характеристики є зручними для перевірки тесту, який повинен шукати серед певних статистично ненайгірших послідовностей ті, які є надлишковими.

Було обрано 2 комбінації параметрів ЛКП для реалізації. Розроблено консольний додаток мовою C++ з використанням середовища Microsoft Visual Studio, що здатен генерувати та зберігати до файлів такі ЛКП.

За допомогою пакету NIST STS було перевірено статистичні характеристики реалізованих генераторів. Виявилось, що вони відрізняються за якістю. При повноцінному тестуванні обидва успішно відкидаються тестами NIST STS, але при тестуванні коротких послідовностей один з генераторів відкидається, а другий помилково приймається пакетом тестів, що підтверджує припущення про слабку чутливість в таких умовах.

# 3 РОЗРОБКА ТЕСТУ ДЛЯ КОРОТКИХ ПОСЛІДОВНОСТЕЙ

Розроблюваний тест задуманий на теорії Колмогорівської складності. Припускається можливість того, що якийсь або навіть декілька з існуючих на даний момент та поширених у вжитку алгоритмів стискання даних можна використовувати у якості практичного наближення до універсального декомпресора.

Експериментальним шляхом знайшовши подібний алгоритм з'являється можливість побудувати новий формат тестування для ПВП з його використанням.

## 3.1 Використання архіваторів для тестування псевдовипадкових послідовностей

Згідно з теорією алгоритмічної інформації Колмогорова, інформація, яку містить у собі повідомлення, дорівнює довжині найкоротшого можливого альтернативного запису цього повідомлення. При цьому елементи послідовності рівномірно розподілених випадкових величин не несуть в собі жодну інформацію про наступні елементи. Отже, реальну рівномірно незалежно розподілену послідовність не можна записати коротше, тобто не можна її стиснути.

Таким чином, якщо вдасться стиснути ПВП за допомогою певного існуючого алгоритму, то можна судити про її неідеальність і вдаватись до глибшого аналізу, якщо це потрібно (наприклад при криптоатаці), або одразу ж відмовитись від використання такої послідовності у випадку підбору послідовності для нового шифру.

Серед існуючих і популярних в наш час систем стискання даних можна виділити три напрямки відповідно до трьох підходів до роботи

алгоритму стискання.

Перший, найстаріший і досі найпопулярніший та конкурентноздатний підхід це словникове стискання. Всі сучасні алгоритми словникового стискання є модернізаціями перших 2-х алгоритмів запропонованих Абрахамом Лемпелем і Якобом Зівом у 1977 та 1978 роках, які відповідно носять назви LZ77 та LZ78. В алгоритмах такого типу спочатку відбувається накопичення словника з оброблюваного файлу, а далі так звані «слова», які повторюються, кодуються посиланнями на словник.

Другим напрямком стиснення даних є алгоритми, що базуються на універсальних моделюваннях та кодуваннях контекстів. Найпоширенішим підходом у цьому напрямку є техніка передбачення по частковій поведінці(PPM) вперше запропонована в 1984 році.

Останньою течією є системи, які мають у собі попереднє перетворення даних для подальшого більш ефективного стиснення. Найпопулярнішим таким перетворенням є перетворення Барроуза-Уіллера(BWT) запропоноване у 1994 році. Хоч і започаткованою така попередня обробка була для покращення роботи двох вищеописаних підходів, але в ході свого розвитку напрям майже відокремився, адже склалось так, що після специфічних початкових перетворень даних більш ефективними є спеціально розроблені методи стискання, які в свою чергу без попередньої обробки дуже поступаються словниковим та контекстно моделюючим.

Також зустрічаються алгоритми, які комбінують вищеописані підходи, але для їх спільної продуктивної діяльності необхідна розробка алгоритму саме під сумісність. Не можна просто архів отриманий одним алгоритмом запустити на вхід іншому і отримати покращений результат. Прийнято рішення провести порівняння найпопулярніших представників різних алгоритмічних підходів з точки зору ефективності використання для поставленої задачі тестування на випадковість коротких послідовностей.

Таким чином йдеться про пошук алгоритму стискання даних, який можна використовувати в якості наближення до універсального декомпресора.

Обрані наступні архіватори для перевірки:

- 7-ZIP 21.07 — алгоритм та відповідний формат архіву 7z є високоефективною реалізацією алгоритму словникового алгоритму LZMA2, який в свою чергу походить від LZ77 і є дуже популярним через свої компресійні та швидкісні характеристики;

- XZ 1.0 — альтернативна менш популярна реалізація LZMA2;

- LZMA 2.1 — хороша реалізація LZMA, який є попередником LZMA2;

- Алгоритм Хаффмана (далі в таблиці AH) — класичний алгоритм стискання з універсальним кодуванням

- HA 1.1.0 — старий, але досі подекуди вживаний алгоритм заснований на контекстному моделюванні;

- PPMd 1.0 — один з найкращих на даний час архіваторів на методі передбачення по частковій поведінці PPM;

- RAR 5 — пропрієтарний алгоритм стискання даних, який є комбінацією PPM та LZSS, який в свою чергу походить від LZ77;

- BZIP2 1.5 — найпопулярніший алгоритм, що базується на перетворенні Барроуза-Уіллера (BWT);

- ABC-TC 2013 — більш новий хороший алгоритм на BWT.

Першою і найважливішою характеристикою архіваторів відповідно до поставленої задачі є мінімальна довжина вхідної неідеальної послідовності, для якої алгоритм буде чутливим. Тобто порогова довжина ПВП при якій почнеться стискання. Результати перевірок цієї границі наведені у таблиці 3.1

Відповідно для обох генераторів та усіх архіваторів вказано довжину послідовності в бітах, за якої починається стиснення, та дробом вказана частка, на яку зменшилась така порогова послідовність після обробки.

По суті, ця частка і є надлишковою інформацією і її величина є кількісною оцінкою невипадковості представлених послідовностей. Чим

чутливіший до неї архіватор, тим краще.

**Таблиця 3.1** – Порогові довжини чутливості

| | Перший генератор | Другий генератор |
|---|---|---|
| 7-zip | 6144 біт | 1048576 біт |
| | 0.026 | 0.099 |
| XZ | 256768 біт | 1048576 біт |
| | 0.012 | 0.008 |
| LZMA | 6144 біт | 1048576 біт |
| | 0.086 | 0.023 |
| AH | - | - |
| | - | - |
| HA | - | - |
| | - | - |
| PPMd | - | - |
| | - | - |
| RAR 5 | 768 біт | 5120 біт |
| | 0.41 | 0.273 |
| BZIP2 | - | - |
| | - | - |
| GZIP | - | - |
| | - | - |
| ABC | - | - |
| | - | - |

Експеримент продовжувався до довжини послідовності у 100 000 000 бітів. Власне алгоритми Хаффмана, HA, PPMd, BZIP2 та ABC взагалі не стиснули послідовність ЛКГ-ра в межах цієї довжини генерації. Тобто усі алгоритми, що не містять у собі словникового стискання можна одразу відкидати і не розглядати у якості перевірочних для ПВП. На даному етапі видно, що RAR відреагував раніше і на багато чутливіше, ніж чисті словникові алгоритми.

Наступним експериментом була побудова залежності степеня стискання від довжини вхідної послідовності. Результати обрахунків показано на Рисунках 3.1, 3.2 та 3.3

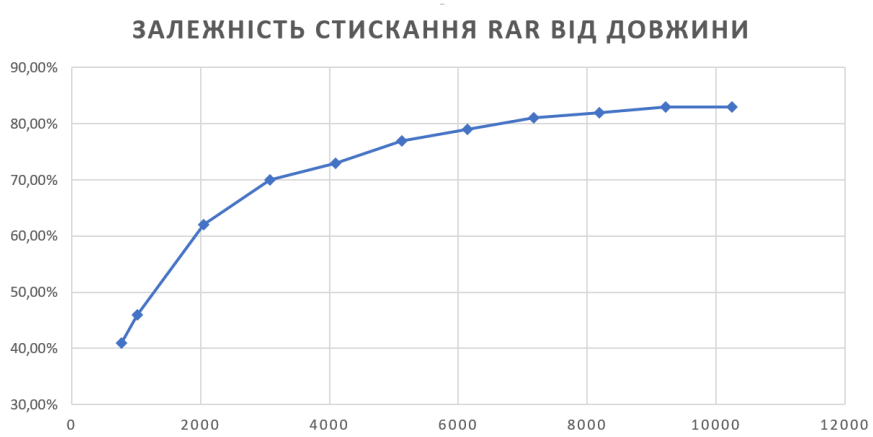


**Рисунок 3.1** – Залежність стискання послідовності першого генератору від величини послідовності для RAR 5

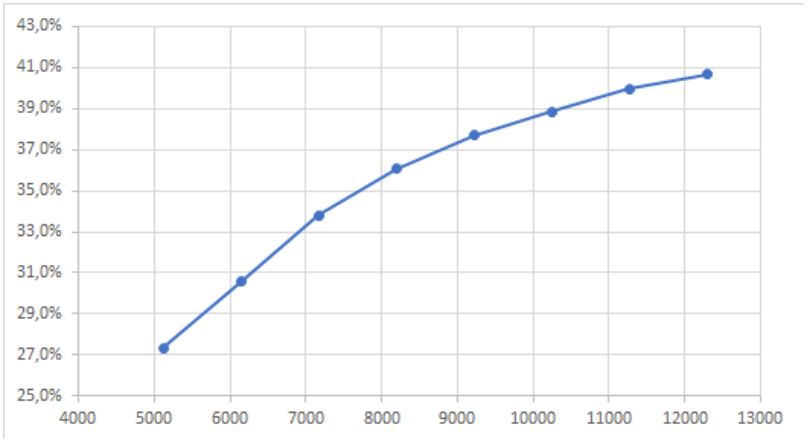


**Рисунок 3.2** – Залежність стискання послідовності другого генератору від величини послідовності для RAR 5

Видно, що степінь стискання у RAR 5 росте з довжиною послідовності, але залежність схожа за формою на логарифмічну і згодом зі збільшенням послідовності посилення стискання є незначним. Але величина стискання при цьому дійшла до відмітки у 83%, що відповідає

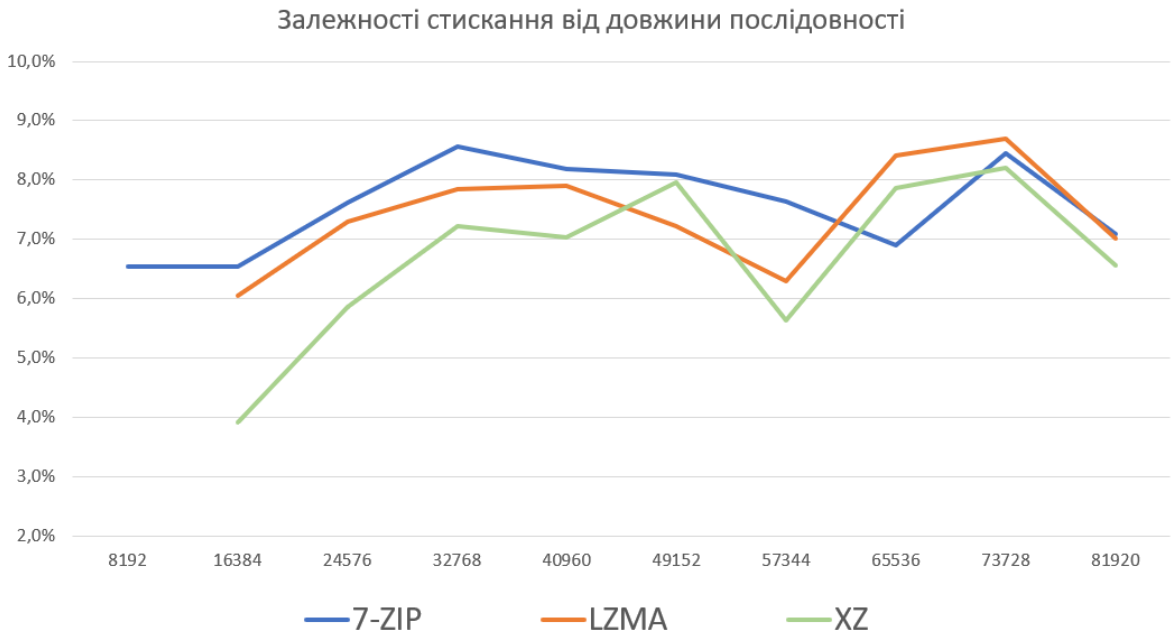


**Рисунок 3.3** – Залежність стискання від величини послідовності для 7-ZIP, LZMA та XZ

83% надлишкової інформації у послідовності, що говорить про сильну її передбачуваність.

В той же час у словникових методів 7-ZIP, LZMA та XZ не спостерігається нарощення ступеня стискання з величиною послідовності, як тільки словник зібрався вони з певним вірогіднісним відхиленням, яке відрізняється від генерації до генерації, вийшли орієнтовно на свою робочу величину стиснення для даного генератора.

Поєднання словникового архівування та методу передбачення по частковій поведінці дозволило алгоритму RAR 5 показати кращі від усіх інших показники на короткій неідеальній ПВП-ті, при цьому у багатьох опублікованих результатах тестів з великими файлами різних специфік даний архіватор також не поступається іншим кандидатам.

## 3.2 Побудова тесту коротких псевдовипадкових послідовностей

Вище вже зазначалось, що згідно з твердженням 1.1 на пакет NIST STS та його результативність можна рівнятись в сучасних умовах. При цьому твердження 2.1 підмічає такий явний недолік цього пакету, як неповноцінність та неточність результатів тестування в умовах коротких послідовностей. Цей недолік важко оспорити. Втрата чутливості відбувається через неможливість застосування найкрутіших тестів пакету, а саме: дискретного перетворення Фур'є, перевірки рангів матриць, універсального статистичного тесту Маурера та перевірки випадкових відхилень. Пропонується використовувати стискання алгоритмом наближеним до ідеального декомпресора замінити ці тести пакета.

За результатами проведених експериментальних перевірок спроможності сучасних архіваторів стискати послідовність отриману лінійним конгруентним генератором виділяється явний фаворит — алгоритм RAR 5. Мінімальна довжина вхідної послідовності, на якій він починає стискати знаходиться в районі семиста бітів, що є доволі таки маленьким значенням порівнюючи з мінімумом у 1000000 бітів для більшості тестів пакету NIST STS.

Для підтвердження відсутності фактів, які заборонили би використання RAR 5 у якості наближення до ідеального декомпозитору, джерело [26] надає декілька великих таблиць з порівняннями продуктивності різних архіваторів в залежності від походження архівованих файлів. RAR 5 не займає там перші позиції, але в кожному списку він відхилений від лідера не більше, ніж на 4 відсотки стискання. Також слід відзначити, що лідери майже всіх списків різні, тобто вони спеціалізуються на даних видах файлів, а RAR 5 при цьому не відрізняється від них більше, ніж на певну константу.

Власне для перевірки гіпотези рівномірної випадковості джерела

ПВП, в умовах, коли немає можливості отримати довгі зразки послідовності джерела, або коли генерація довгих послідовностей є дуже затратною по часу, що є актуальним для деяких сучасних статистично хороших генераторів, пропонується наступний розроблений формат тестування:

1) короткі послідовності згенерованих бітів спочатку проходять тестування доступними тестами з пакета NIST STS:

– частотний тест;

– частотний тест у блоках;

– перевірка кумулятивних сум;

– перевірка «дірок»;

– перевірка «дірок» у підпослідовності;

– перевірка апроксимованої ентропії;

2) короткі послідовності піддаються стисканню алгоритмом RAR.

Кожен з тестів NIST STS в якості результату видає так зване значення P-value. Ця величина показує імовірність генерації ідеальним джерелом рівномірної незалежної випадкової величини такої послідовності, яка є гіршою, тобто менш рівноймовірною, за досліджувану. Критерієм визнання досліджуваної послідовності рівномірною випадковою є виконання наступної нерівності:

$$P - value \geqslant \alpha,$$

де $\alpha$ це рівень значущості. Якщо критерій виконується, то з імовірністю $1 - \alpha$ приймається гіпотеза про рівномірну випадковість джерела досліджуваної послідовності.

Результатом другого етапу розробленого тесту буде стиснена вихідна послідовність. Якщо об'єм вихідного файлу не є меншим за вхідний досліджуваний, то цей етап приймає гіпотезу про рівномірну випадковість джерела. Якщо ж RAR 5 зміг стиснути послідовність, що досліджується, то гіпотеза відхиляється.

Також у разі відхилення гіпотези додатковим отриманим показником буде степінь стискання. Цей параметр характеризуватиме міру надлишковості піддослідного генератора. За її величиною можна визначити кількість необхідних відомих бітів для передбачення наступного.

Насправді є припущення, що стискання алгоритмом RAR 5 може цілком замінити весь пакет NIST STS в умовах коротких послідовностей і не буде таких коротких послідовностей, які прийме RAR, але відхилить NIST STS. Але перевірка цього є питанням вже майбутніх досліджень, а на даному етапі пропонується не відкидати доступні тести з NIST STS та вважати два різних підходи взаємодоповнюючими.

Єдиним суттєвим мінусом даного формату тестування є те, що RAR 5 є пропрієритарним алгоритмом, в вільному доступі знаходиться тільки поверхнева інформація про принципи його роботи. Це майже унеможливлює побудову подальших досліджень з відштовхуванням від даного тесту, тобто не можна покращити роботу того, що невідомо, як працює. Подальші дослідження слід перенести у русло пошуку кращих архіваторних алгоритмів серед тих, які є відкритими.

## Висновки до розділу 3

За результатами експериментального пошуку серед сучасних алгоритмів стискання даних такого, який можна було б вважати наближенням до ідеального декомпресора, тобто такого, який є придатним для тестування коротких ПВП виділено тільки один алгоритм, який є суттєво кращим інших, а саме RAR 5.

Визначено, що чутливість RAR 5 на коротких послідовностях є кращою за пакет NIST STS. Таким чином підтверджено можливість RAR 5 виступати заміною тої частини тестів NIST STS, яка є непрацездатною при дослідженні коротких послідовностей.

Запропоновано свою двоетапну схему тестування для коротких ПВП.

# ВИСНОВКИ

Проведено огляд опублікованих джерел пов'язаних з темою тестування коротких псевдовипадкових послідовностей. Цей огляд показав, що наявні інструменти вирішують задачу подібного тестування недостатньо ефективно та подекуди неінформативні.

Також в результаті огляду доведено теоретичну можливість алгоритмів стискання даних показати кращі результати на полі перевірки якості генераторів по коротких послідовностях.

Для перевірки працездатності розроблюваного тесту обрано та реалізовано мовою C++ два варіанти лінійного конгруентного генератора. Цей алгоритм генерації за рахунок своєї передбачуваності, та в той же час не найгірших статистичних показників є дуже хорошим кандидатом, щоб на ньому перевіряти працездатність нового тесту.

Також перед подальшим використанням реалізовних генераторів їх було протестовано популярним потужним статистичним пакетом тестів NIST STS. Виявлено, що по відношенню до одного з обраних генераторів даним пакетом тестів помилково приймається гіпотеза про його рівномірну випадковість у разі тестування послідовностей короткої довжини. Таким чином виявлено практичний екземпляр недосконалості системи NIST STS.

В результаті експериментальної перевірки можливості використання сучасних алгоритмів стискання даних для тестування псевдовипадкових послідовностей споміж інших виділено алгоритм RAR 5 та визнано можливість використання його у якості ідеального декомпресора. RAR 5 проявив свою чутливість починаючи з 700 бітів послідовності.

Інші досліджувані алгоритми в більшості своїй взагалі не впорались з задачею стискання лінійної конгруентної послідовності. Тільки алгоритми типу LZMA та LZMA2 впорались з гіршим варіантом з двох обраних, але не впорались з тим, який був проблемним також і для NIST STS.

Запропоновано власний двоетапний формат тестування коротких бітових послідовностей, які об'єднує у собі частину тестів NIST STS та стискання алгоритмом RAR 5. Наведений тест успішно відхиляє гіпотезу про рівномірну випадковість вищезгаданого реалізованого генератора навіть на коротких послідовностях.

Пропрієритарність алгоритму RAR 5 є суттєвим недоліком для подальших глибоких оцінок якості роботи подібного тесту, тому подальші дослідження вбачаються у пошуку або розробці не менш хороших алгоритмів, які матимуть відкритий код, для того, щоб можна було досліджувати також і теоретичну ефективність, а не тільки емпіричну, а також більша кількість дослідників матиме змогу поставити під сумнів якість подібного тестування.

# ПЕРЕЛІК ПОСИЛАНЬ

1. А.В. Виноградов, Тесты случайности и псевдослучайные числа: теория и практика. Москва : Московский государственный университет имени М.В. Ломоносова., 2018. 21 с.

2. Х.Х. Альнаджар, Модель и программный комплекс генератора псевдослучайных чисел, основанного на нечеткой логикею. Казань : Казанский национальный исследовательский технический университет им. А.Н. Тупол., 2018. 192 с.

3. D. M'Raihi. TOTP: Time-Based One-Time Password Algorithm // D.M'Raihi,S. Machani, M. Pei, J. Rydell / Internet Engineering Task Force (IETF), 2011

4. Методы генерации и тестирования случайных последовательностей / М.Б. Будько, М.Ю. Будько, А.В. Гирик, В.А. Грозов. / Университет ИТМО, Санкт-Петербург. – 2019. – С. 73

5. A Crystallization Robot for Generating True Random Numbers Based on Stochastic Chemical Processes / Edward C. Lee, Juan M. Parrilla-Gutierrez, Alon Henson та ін.]. / Matter. – 2020. – С. 649–657.

6. DNA synthesis for true random number generation / L. C. Meiser et al. Nature Communications. 2020. Vol. 11, no. 1.

7. Д. Кнут. Искусство программирования, том 2. Получисленные методы – 3-е изд. / М.: «Вильямс». –2007. – С. 832

8. Иванов М.А., Чугунков И.В. Криптографические методы защиты информации в компьютерных системах и сетях. / М.: НИЯУ МИФИ, 2012. – С.400

9. Слеповичев И.И. Генераторы псевдослучайных чисел. / Саратов: СГУ, 2017. – С. 118

10. Григорьев А. Ю. Методы тестирования генераторов случайных и псевдослучайных последовательностей. Ученые записки УлГУ. Сер. Математика и информационные технологии. УлГУ. Электрон. 2017. № 1.

С. 22–28.

11. NIST Statistical Test Suite. [Електронний ресурс]. — Режим доступу: http://csrc.nist.gov/groups/ST/toolkit/rng/documentation_software.html.

12. Raddum H. The Statistical Evaluation of the NESSIE Submission. 5020th ed. Bergen : Department of Informatics, The University of Bergen, 2002. 28 p.

13. TEST-U01. [Електронний ресурс]. — Режим доступу: http://www.iro.umontreal.ca/~simardr/testu01/tu01.html

14. CRYPT-X. [Електронний ресурс]. — Режим доступу: http://www.isi.qut.edu.au/resources/cryptx.

15. The pLab Project. [Електронний ресурс]. — Режим доступу: http://random.mat.sbg.ac.at.

16. George Marsaglia, DIEHARD Statistical Tests. [Електронний ресурс]. — Режим доступу: http://stat.fsu.edu/~geo/diehard.html.

17. ENT. [Електронний ресурс]. — Режим доступу: http://www.fourmilab.ch/random/.

18. Robert G. Brown's General Tools Page. [Електронний ресурс]. — Режим доступу: http://www.phy.duke.edu/~rgb/General/dieharder.php

19. FIPS PUB 140-1. Security requirements for cryptographic modules. Effective from 1994-01-11. Official edition. 1994. 56 p.

20. В.В. Вьюгин. Колмогоровская сложность и алгоритмическая случайность. Москва : МФТИ, 2012. 132 с.

21. В.А. Успенский. Колмогоровская сложность и алгоритмическая случайность. Москва : МЦНМО, 2010. 556 с.

22. А.Н. Колмогоров. Три подхода к определению понятия "количество информации". Проблемы передачи информации. 1965. Т. 1, № 1. С. 3–11.

23. Т.Ю. Прокопенко. Методи оцінки стійкості систем багатофакторної автентифікації, які використовують одноразові паролі. Київ, 2016. 96 с.

24. The Diehard Tests – the urban engine. The urban engine. [Електронний ресурс]. — Режим доступу: http://theurbanengine.com/blog//the-diehard-tests.

25. NIST SP 800-22 Revision 1a. A Statistical Test Suite for Random and Pseudorandom Number Generators for Cryptographic Applications // National Institute of Standards and Technology Special Publication 800-22, Revision 1a, 131 pages, April 2010.

26. Всё о сжатии данных, изображений и видео. [Електронний ресурс]. — Режим доступу: https://www.compression.ru/arctest/menu/cmptests.htm

# ДОДАТОК А ТЕКСТИ ПРОГРАМ

## А.1 main.cpp

```
#include <iostream>
#include <fstream>
#include <string>
#include <chrono>
#include <algorithm>
#include "Rabin.h"
#include <windows.h>

using namespace std;

/*ФУНКЦИЯ СОЗДАНИЯ НОВОГО ФАЙЛА*/
void file_create(const char* PATH,char flag1, u32 length32) {
    ofstream f1(PATH, ios::binary);
    //srand(time(0));
    //LehmerLow lahmerl = LehmerLow(u32(rand()), 1003);
    //lahmerl.MakeNBloks(1000);
    //lahmerl.printSeq();
    if (flag1 == 0) {
        cout << "\no\n";
    }
    else if(flag1 == 1) {
        cout << "\LinearKG_1\n";
        cout << "generation length: " << length32 * 32;
        LinearKG gener1 = LinearKG(u32(rand()), length32 + u32(3));
        gener1.MakeNBloks(length32);
        //cout << "\nTesting LKG_1\n";
        Tester alpha00(gener1, 8);
        //alpha00.print_results();
```

```
        //alpha00.print_results2();
        //alpha00.print_results3();
        cout << "\n------------------\n";

        int tmp = 0;
        for (int i = 0; i < length32; i++) {
            for (int j = 0; j < 4; j++) {
                //tmp = (rand() % 256);
                tmp = gener1.Byte(i, j);
                f1.write((char*)&tmp, sizeof(char));
                //f1.write((char*)&i, sizeof(char));
            }
        }
    }
    else {
        cout << "\LinearKG_2\n";
        cout << "generation length: " << length32 * 32;
        LinearKG_2 gener2 = LinearKG_2(u32(rand()),
        length32 + u32(3));
        gener2.MakeNBloks(length32);
        //cout << "\nTesting LKG_2\n";
        Tester alpha00(gener2, 8);
        //alpha00.print_results();
        //alpha00.print_results2();
        //alpha00.print_results3();
        cout << "\n------------------\n";

        int tmp = 0;
        for (int i = 0; i < length32; i++) {
            for (int j = 0; j < 4; j++) {
                //tmp = (rand() % 256);
```

```
                tmp = gener2.Byte(i, j);
                f1.write((char*)&tmp, sizeof(char));
                //f1.write((char*)&i, sizeof(char));
            }
        }
        //gener2.printSeq();
    }




    f1.close();
}

/*ФУНКЦИЯ ОБХОДА ФАЙЛА*/
void file_read(const char* PATH) {
    ifstream f1(PATH, ios::binary);
    int value = 0;
    string str = "";
    int i = 0;
    while (!f1.eof()) {
        i++;
        f1.read((char*)&value, sizeof(char));
        if (!f1) break;
        //str += to_string(value);
        str += "0123456789ABCDEF"[value / 16];
        str += "0123456789ABCDEF"[value % 16];
        str += " ";
        if (i % 40 == 0) {
            str += "\n";
```

```
        }
    }
    cout << str << '\n';
    f1.close();
}

int main() {

    srand(time(0));

    const char* PATH = "C:\\LKG_1_test";

    cout << "\n===========\n";

    file_create(PATH, 1, 6553600);
    //file_read(PATH);

    PATH = "C:\\LKG_2_test";

    cout << "\n===========\n";


    file_create(PATH, 2, 6553600);
    //file_read(PATH);

    /*
    PATH = "C:\\LKG_768";

    cout << "\n===========\n";

    file_create(PATH, 2, 768);
```

```
file_read(PATH);

PATH = "C:\\LKG_1024";

cout << "\n==========\n";

file_create(PATH, 2, 1024);
file_read(PATH);

PATH = "C:\\LKG_1280";

cout << "\n==========\n";

file_create(PATH, 2, 1280);
file_read(PATH);

PATH = "C:\\LKG_1536";

cout << "\n==========\n";

file_create(PATH, 2, 1536);
file_read(PATH);

PATH = "C:\\LKG_1792";

cout << "\n==========\n";

file_create(PATH, 2, 1792);
file_read(PATH);

PATH = "C:\\LKG_2048";
```

```
cout << "\n===========\n";

file_create(PATH, 2, 2048);
file_read(PATH);

PATH = "C:\\LKG_2304";

cout << "\n===========\n";

file_create(PATH, 2, 2304);
file_read(PATH);


PATH = "C:\\LKG_2560";

cout << "\n===========\n";

file_create(PATH, 2, 2560);
file_read(PATH);

PATH = "C:\\LKG_2816";

cout << "\n===========\n";

file_create(PATH, 2, 2816);
file_read(PATH);

PATH = "C:\\LKG_3072";

cout << "\n===========\n";
```

```
    file_create(PATH, 2, 3072);
    file_read(PATH);
    */


    //PATH = "C:\\LKG_1_512.zip";
    //file_read(PATH);

    //system("\"Rar\" a arhive_r2.rar C:\\bm_32k.txt");
    //system("pause");



    //file_create(PATH);
    //file_read(PATH);


    cin.get();
    cin.get();

}
```

## A.2 Generators.cpp

```
#include <iostream>
#include <string>
#include <bitset>
#include <algorithm>
#include <ctime>
#include <fstream>
```

```
#include <io.h>
#include <fcntl.h>
#include <stdio.h>
#include <NTL/ZZ.h>


constexpr auto BOX_SIZE = 32UL;

using namespace std;
using namespace NTL;

typedef unsigned char u8;
typedef unsigned short u16;
typedef unsigned long u32;
typedef unsigned long long u64;


class Generator {

protected:

        u32 size;
        u32 global_I;
        u32 local_I;
        u32* sequence;

        virtual void Step() = 0;

        static u32 getGlobIndex(u32 n) {
                return n / u32(32);
        }
```

```
        static u32 getLocalIndex(u32 n) {
                return n % u32(32);
        }

//        const u32 Byte(u32 sequence_index, u32 byte_index)
    {
//                return (sequence[sequence_index] >>
8 * byte_index) & 255UL;
//        }

public:

        Generator(u32 _size = 65) {
                size = _size;
                global_I = 0;
                local_I = 0;
                sequence = new u32[size];
                for (u32 i = 1; i < size; i++) {
                        sequence[i] = 0;
                }
        };

        Generator(const Generator& a) {
                size = a.size;
                global_I = a.global_I;
                local_I = a.local_I;
                sequence = new u32[size];
                for (u32 i = 1; i < size; i++) {
                        sequence[i] = a.sequence[i];
                }
```

```
}
const u32 Byte(u32 sequence_index, u32 byte_index)
{
        return (sequence[sequence_index]
        >> 8 * byte_index) & 255UL;
}
const u32 Boxe(u32 sequence_index) {
        return sequence[sequence_index];
}

virtual Generator& operator = (const Generator& a)
{
        size = a.size;
        global_I = a.global_I;
        local_I = a.local_I;
        delete[] sequence;
        sequence = new u32[size];
        for (u32 i = 0; i < size; i++) {
                sequence[i] = a.sequence[i];
        }
        return *this;
}

u32 Length() {
        return global_I * 32UL + local_I;
}

void MakeNBloks(u32 n) {
        while ((global_I < n)) {
                (*this).Step();
        }
```

```
        sequence[n] = 0;
        global_I = n - 1;
        local_I = 31;
}

void printSeq() {
        bitset<32> temp;
        if (global_I > 20) {
                for (u32 i = 0;
                i < min(global_I + 1UL, 10UL); i++) {
                        temp =
                        bitset<32>(sequence[global_I - i]);
                        cout << temp << " ";
                }
        }
        cout << "\n.    .    .    .    .\n";
        if (global_I < 11) {
                for (u32 i = 0;
                i < global_I + 1UL; i++) {
                        temp =
                        bitset<32>(sequence[global_I - i]);
                        cout << temp << " ";
                }
        }
        else {
                for (u32 i = global_I - 10;
                i < global_I + 1UL; i++) {
                        temp =
                        bitset<32>(sequence[global_I - i]);
                        cout << temp << " ";
                }
```

```
                }
                cout << "\n";
        }

        ~Generator() {
                delete[] sequence;
        }

        friend class Geffe;

        friend class Tester;

        friend class PtimeGenerator;

};

class Lehmer :public Generator {

protected:

        static const u64 koef_a = 65537ULL;
        static const u64 koef_c = 119ULL;
        u32 xn;

        virtual u32 NewValue(u32 x) = 0;

        virtual void Step() {
                xn = u32(koef_a * u64(xn) + koef_c);
                local_I += 8UL;
                if (local_I == 32UL) {
                        local_I = 0;
```

```
                global_I++;
        }
        sequence[global_I] |= NewValue(xn) << local_I;
    }

public:

    Lehmer(u32 x0 = 1UL, u32 _size = 46880UL) :Generator(_size) {
        xn = x0;
    }

};

class LinearKG :public Lehmer {

protected:

    void Step() {
        xn = u32(koef_a * u64(xn) + koef_c);
        global_I++;
        sequence[global_I] = NewValue(xn);
    }

    u32 NewValue(u32 x0) {
        return x0;
    }

public:

    LinearKG(u32 x0 = 1, u32 _size = 46880) :Lehmer(x0, _size) {
        sequence[0] = NewValue(x0);
```

```
        }

};

class LinearKG_2 :public Lehmer {

protected:

        static const u64 koef_a1 = 9737ULL; //0x10001; // 2^16 + 1
        static const u64 koef_c1 = 23209ULL; //0x77;

        void Step() {
                xn = u32(koef_a1 * u64(xn) + koef_c1);
                global_I++;
                sequence[global_I] = NewValue(xn);
        }

        u32 NewValue(u32 x0) {
                return x0;
        }

public:

        LinearKG_2(u32 x0 = 1, u32 _size = 46880):
        Lehmer(x0, _size) {
                sequence[0] = NewValue(x0);
        }

};
```