Міністерство освіти і науки України
Харківський національний університет імені В. Н. Каразіна
Навчально-науковий інститут комп'ютерних наук та штучного інтелекту

Кафедра кібербезпеки інформаційних систем, мереж і технологій

До захисту допущено
Кафедрою КІСМіТ протокол №____ від «___» грудня 2025 р.

завідувач кафедри _____ Марина ЄСІНА
(підпис) *(ім'я, прізвище)*
«___» грудня 2025 р.

Кваліфікаційна робота

здобувача другого (магістерського) рівня вищої освіти

Статистичний аналіз екстрактора QRNG на основі криптографічної геш-функції SHA-3

Спеціальність (спеціалізація) 125 «Кібербезпека та захист інформації»

Освітня програма «Безпека інформаційних і комунікаційних систем»

Виконавець _____ Дмитро ВОЛОТКОВСЬКИЙ
(підпис) *(ім'я, прізвище)*

Науковий керівник _____ Олексій НАРЄЖНІЙ
(підпис) *(ім'я, прізвище)*

Харків – 2025

РЕФЕРАТ

Пояснювальна записка до дипломної роботи містить 125 сторінок, 27 рисунків, 11 таблиць, 5 лістингів, 7 додатків, 46 посилань на джерела.

Метою даної роботи є проведення статистичного аналізу роботи екстрактора квантового генератора випадкових чисел (QRNG) на основі криптографічної геш-функції SHA-3, а також обґрунтування доцільності використання даного алгоритму для постобробки «сирих» квантових даних.

Предметом дослідження є статистичний аналіз ефективності екстрактора, включаючи оцінку зміщення (bias), автокореляцій та проходження статистичних тестів на випадковість (NIST STS, Dieharder).

Об’єктом дослідження є екстрактор випадковості QRNG на основі криптографічної геш-функції SHA-3.

У результаті проведених досліджень було виконано теоретичний аналіз архітектури SHA-3 (Keccak) та її губкової конструкції (sponge construction). Розроблено програмне забезпечення мовою Python для реалізації екстрактора та проведення експериментів. Виконано порівняльний статистичний аналіз «сирих» даних, отриманих з QRNG, та даних після обробки алгоритмом SHA-3. Результати тестування пакетом Dieharder підтвердили, що застосування SHA-3 дозволяє усунути статистичні дефекти, такі як зміщення ймовірності та локальні кореляції, перетворюючи дефектний вхідний потік на криптографічно стійку випадкову послідовність.

Результати дослідження можуть бути використані для підвищення безпеки криптографічних систем, генерації ключів шифрування та в системах захисту критичної інфраструктури.

Ключові слова: КВАНТОВИЙ ГЕНЕРАТОР ВИПАДКОВИХ ЧИСЕЛ, QRNG, SHA-3, KECCAK, ЕКСТРАКТОР ВИПАДКОВОСТІ, ЕНТРОПІЯ, СТАТИСТИЧНИЙ АНАЛІЗ, NIST STS, DIEHARDER, КІБЕРБЕЗПЕКА.

ABSTRACT

The explanatory note for the master's thesis consists of 77 pages, 27 figures, 11 tables, 5 listings, 7 appendices and 46 references.

The purpose of this work is to conduct a statistical analysis of the operation of a Quantum Random Number Generator (QRNG) extractor based on the SHA-3 cryptographic hash function, as well as to justify the feasibility of using this algorithm for post-processing «raw» quantum data.

The subject of the research includes the statistical analysis of the extractor's efficiency, including the assessment of bias, autocorrelations, and passing of statistical randomness tests (NIST STS, Dieharder).

The object of the research is the QRNG randomness extractor based on the SHA-3 cryptographic hash function.

As a result of the conducted research, a theoretical analysis of the SHA-3 (Keccak) architecture and its sponge construction was performed. Software was developed in Python to implement the extractor and conduct experiments. A comparative statistical analysis was carried out on «raw» data obtained from a QRNG and data post-processed by the SHA-3 algorithm. Testing results using the Dieharder suite confirmed that the application of SHA-3 effectively eliminates statistical defects, such as probability bias and local correlations, transforming a defective input stream into a cryptographically secure random sequence.

The research results can be used to improve the security of cryptographic systems, encryption key generation, and critical infrastructure protection systems.

Keywords: QUANTUM RANDOM NUMBER GENERATOR, QRNG, SHA-3, KECCAK, RANDOMNESS EXTRACTOR, ENTROPY, STATISTICAL ANALYSIS, NIST STS, DIEHARDER, CYBERSECURITY.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ

AES – Advanced Encryption Standard

ASIC – Application-Specific Integrated Circuit

DRBG – Deterministic Random Bit Generator

EMI – Electromagnetic Interference

FIPS – Federal Information Processing Standards

FPGA – Field-Programmable Gate Array

GF – Galois Field

GUI – Graphical User Interface

LCG – Linear Congruential Generator

MAC – Message Authentication Code

MD – Merkle-Damgård

MT – Mersenne Twister

NIST – National Institute of Standards and Technology

OPSO – Overlapping Pairs Sparse Occupancy

OQSO – Overlapping Quadruples Sparse Occupancy

PRNG – Pseudo-Random Number Generator

QES – Quantum Entropy Source

QRNG – Quantum Random Number Generator

SHA – Secure Hash Algorithm

SHAKE – Secure Hash Algorithm Keccak

STS – Statistical Test Suite

TRNG – True Random Number Generator

XOF – Extendable-Output Function

АЦП – Аналого-цифровий перетворювач

ГВЧ – Генератор випадкових чисел

КГВЧ – Квантовий генератор випадкових чисел

ЛКГ – Лінійний конгруентний генератор

ПЛІС – Програмована логічна інтегральна схема

ВСТУП

На сьогоднішній день, у сучасних інформаційно-комунікаційних системах питання безпеки даних є одним із ключових аспектів. Через зростання кількості кібератак, розвиток квантових технологій та необхідність захисту критичної інфраструктури вимагають використання надійних криптографічних методів. Базовим елементом криптографії є генерація випадкових чисел, які застосовуються у створенні ключів, ініціалізаційних векторів, одноразових паролів та інших механізмів захисту.

На сьогоднідній день частота та масштаб кібератак стрімко зростають, що робить криптографічні системи невід'ємною частиною захисту інформації. Глобальна тенденція кіберзагроз демонструє постійне погіршення [1]. Це зумовлює необхідність посилення криптографічних механізмів, які забезпечують конфіденційність, цілісність та автентичність даних. Для ілюстрації масштабу, лише за перший квартал 2025 року середня кількість кібератак на одну організацію зросла на 47% порівняно з аналогічним періодом 2024 року [1]. Загальні прогнозовані збитки від кіберзлочинності до кінця 2025 року сягнуть 10,5 трильйонів доларів щорічно [1].

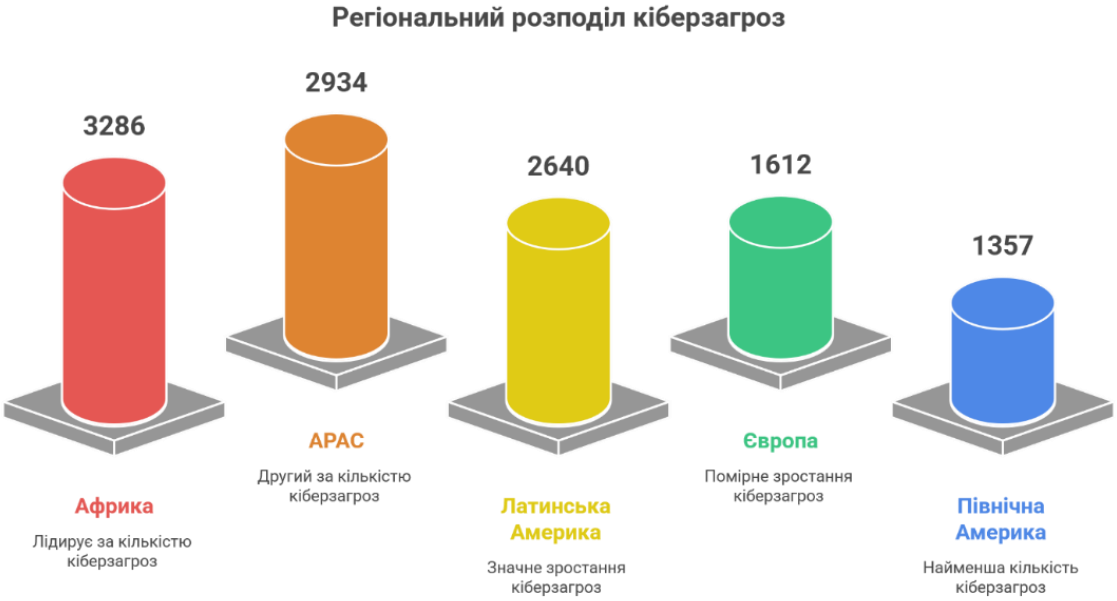


Рисунок 1 – Регіональний розподіл кіберзагроз

Рисунок 2 – Статистичний розподіл атак за галузями у Q1 2025

Ці тенденції радикально підвищують вимоги до криптографічної міцності систем захисту. Навіть незначні слабкості в генерації випадковості – ключового елемента для створення ключів та криптографічних протоколів – можуть стати критичною точкою прориву, призводячи до компрометації цілих систем [1].

Традиційні псевдовипадкові генератори чисел (PRNG) широко використовуються в сучасних комп’ютерних системах, однак їхня природа залишається детермінованою. Вони формують послідовності чисел на основі початкового значення та математичних алгоритмів, таких як лінійні конгруентні методи чи інші відомі схеми. Тобто у випадку відновлення або передбачення seed увесь подальший потік значень стає повністю відтворюваним, що знижує криптографічну стійкість системи та відкриває шлях до компрометації даних [2].

Проблема посилюється тим, що навіть сучасні генератори, здатні пройти базові статистичні перевірки, залишаються уразливими до практичних атак. Зокрема, приклади на кшталт Polynonce чи Randstorm продемонстрували, що слабкість у механізмах генерації випадкових чисел може існувати протягом багатьох років, залишаючись непоміченою. При цьому зловмисники, маючи доступ до частини вихідних даних, здатні відновити внутрішній стан генератора і прогнозувати подальші значення [2].

Таким чином успішне проходження наборів статистичних тестів, таких як NIST SP 800-22 [3], ще не є доказом криптографічної стійкості, адже такі перевірки

виявляють лише відхилення від статистичної випадковості, але не захищають від відновлення seed чи прогнозування виходу. У відповідь на ці ризики сучасні криптографічні стандарти, включаючи NIST FIPS 140-3 [4], встановлюють жорсткі вимоги до генерації випадкових чисел.

Квантові генератори випадкових чисел (QRNG) розглядаються сьогодні як стратегічна технологія для підвищення рівня криптографічної безпеки. На відміну від класичних псевдовипадкових генераторів (PRNG), які працюють на основі алгоритмів та початкового значення, QRNG використовують фундаментальні фізичні процеси, які неможливо відтворити чи передбачити класичними методами. Джерелом ентропії можуть бути квантові флуктуації вакууму, випадкове проходження фотонів через напівпрозоре дзеркало чи шум лазерного випромінювання [5]. Саме квантова невизначеність гарантує, що результати генерації не можна відтворити навіть у випадку повного контролю над системою, що робить QRNG принципово стійкими до криптоаналітичних атак.

У наукових дослідженнях розроблено кілька практичних схем QRNG: від гомодинної детекції для вимірювання вакуумного шуму до однофотонних експериментів, де реєструється факт проходження чи непроходження фотона через детектор [6]. Ці підходи дозволяють створювати «істинно випадкові» біти з високою швидкістю – сучасні реалізації досягають гігабітних темпів генерації, що робить технологію придатною не лише для наукових експериментів, а й для масового впровадження у мобільні пристрої та системи захисту критичної інфраструктури [6].

З точки зору безпеки, ключова відмінність QNRG полягає у відсутності залежності від початкового значення, що усуває одну з найслабших ланок у PRNG. Як підкреслює NHSJS, слабка випадковість у класичних генераторах вже неодноразово ставала причиною зламу криптографічних систем: від підбору ключів до компрометації протоколів аутентифікації [7]. Використання QRNG унеможливлює такі сценарії, оскільки випадковість визначається законами квантової фізики, а не детермінованим обчислювальним процесом.

Попри те, що QRNG гарантують істинну випадковість, їхні вихідні бітові послідовності («сирі» дані) часто містять певні статистичні відхилення: зсув імовірності (bias), автокореляції або залежності від фізичних параметрів системи. У реальних QRNG часто спостерігається зміщення, коли ймовірність «0» і «1» не дорівнює 0,5. Також можливі автокореляції між сусідніми бітами через характеристики обладнання.

Важливість постобробки є критичною частиною будь-якого практичного QRNG – вона перетворює сирий, ймовірно зміщений бітовий потік у вихід, який статистично близький до ідеально рівномірного. Одним із найефективніших класів екстракторів є криптографічні геш-функції. Зокрема, стандарт SHA-3, завдяки своїй губковій конструкції (sponge construction) та властивості лавинного ефекту, здатен виступати потужним екстрактором випадковості.

Таким чином, актуальність теми дослідження визначається зростаючою потребою у високоякісних генераторах випадкових чисел для криптографічних застосувань та необхідністю статистичного аналізу ефективності SHA-3 як екстрактора випадковості.

1 ТЕОРЕТИЧНІ ОСНОВИ ТА АНАЛІЗ АЛГОРИТМІВ ГЕНЕРАЦІЇ ВИПАДКОВИХ ЧИСЕЛ

1.1 Класифікація генераторів випадкових чисел

У науковій літературі та практиці існує кілька підходів до класифікації генераторів випадкових чисел (ГВЧ). Фундаментальним є поділ за природою джерела випадковості, який чітко розмежовує детерміновані та недетерміновані генератори [8]. Цю класифікацію можна представити у вигляді ієрархічної структури (рис 1.1), що відображає основні методи отримання випадкових послідовностей.

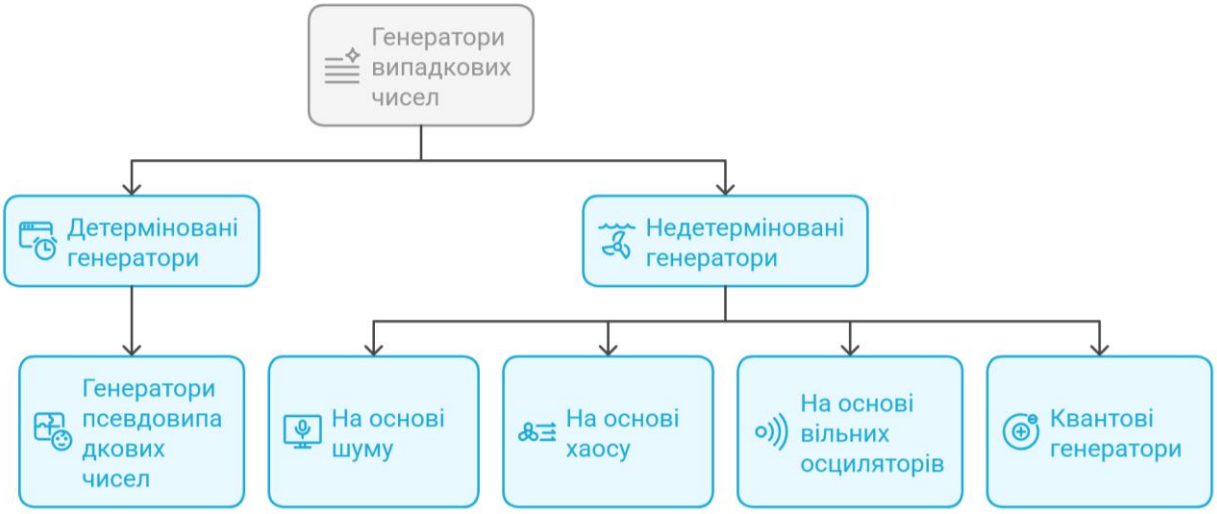


Рисунок 1.1 – Класифікація генераторів випадкових чисел

1) Математичні (алгоритмічні) генератори (детерміновані): Ця категорія, більш відома як генератори псевдовипадкових чисел (PRNG), базується на детермінованих алгоритмах. Отримавши початкове значення, відоме як «зерно» (seed), такий генератор продукує послідовність чисел за допомогою математичної формули. Якщо відомі алгоритм і зерно, вся послідовність може бути точно відтворена. Це робить їх передбачуваними, що є неприйнятним для криптографії, але корисним для відтворюваних симуляцій [9].

2) Фізичні генератори (недетерміновані): Ця категорія, відома як справжні генератори випадкових чисел (TRNG), отримує випадковість з непередбачуваних фізичних процесів. Їхній вихід є принципово невідтворюваним. Залежно від природи фізичного явища, їх можна поділити на кілька підтипів [10]:

- На основі шуму (noise): Використовують стохастичні флуктуації, такі як тепловий шум в електронних компонентах або атмосферний шум [10].
- На основі хаосу (chaos): Використовують вимірювання хаотичних, але детермінованих систем, непередбачуваність яких виникає через надзвичайну чутливість до початкових умов.
- На основі вільних осциляторів (FRO – Free-Running Oscillators): Використовують тремтіння (джитер) у фазі або частоті коливальних контурів, що є популярним методом для реалізації в цифрових мікросхемах [10].
- Квантові (quantum): Квантові генератори випадкових чисел (QRNG) є особливим підкласом TRNG. Вони використовують фундаментальну невизначеність, притаманну квантовій механіці (наприклад, поведінку фотона на дільнику променя). На відміну від інших фізичних генераторів, випадковість яких базується на складності та нашому незнанні системи, випадковість QRNG є доказовою властивістю самої природи [8].

Далі буде розглянуто детально кожен із цих типів генераторів, їхні принципи роботи, переваги та недоліки, приділяючи особливу увагу QRNG як еталону справжньої випадковості.

1.2 Класичні та некриптографічні генератори псевдовипадкових чисел (PRNG)

Генератори псевдовипадкових чисел (PRNG, від англ. Pseudorandom Number Generators) є фундаментальними інструментами в обчислювальній техніці, що застосовуються в широкому спектрі завдань – від наукових симуляцій та статистичного моделювання до відеоігор та процедурної генерації контенту. За своєю суттю, PRNG – це детермінований алгоритм, який, отримавши початкове значення, відоме як «зерно» (seed), генерує послідовність чисел, що імітує

властивості справді випадкової послідовності.[11] Архітектура більшості PRNG є рекурентною: початкове зерно ініціалізує внутрішній стан генератора, а на кожному наступному кроці спеціальна функція переходу оновлює цей стан і виводить число.

Якість класичних PRNG традиційно оцінюється за критеріями, важливими для моделювання: довжина періоду (кількість чисел до повторення), рівномірність розподілу, швидкість генерації та мінімальні вимоги до пам’яті. Однак ці метрики недостатні для криптографії, де ключовою є обчислювальна непередбачуваність. [3] Детермінована природа PRNG робить їх передбачуваними, що є фатальною вразливістю в контексті безпеки.

1.2.1 Лінійні конгруентні генератори (ЛКГ): алгоритмічна простота та криптографічна неспроможність

Лінійний конгруентний генератор (ЛКГ) є одним із найстаріших та найпростіших алгоритмів генерації псевдовипадкових чисел. Його типова апаратна реалізація зображена на рисунку 1.2. Алгоритм базується на рекурентному співвідношенні представлене у формулі 1.1 [12]:

$$X_{n+1} = (aX_n + c) \bmod m, \tag{1.1}$$

де:

X_n – поточне значення (і внутрішній стан),

X_0 – початкове зерно (seed),

m – модуль (визначає максимальний період),

a – множник,

c – приріст.

Через використання простих операцій (множення, додавання, модуль), ЛКГ є ефективними, бо вимагають мінімальної пам’яті (лише одне число) і працюють швидко. Вони популярні для вбудованих систем, простих симуляцій та як навчальний інструмент. Для досягнення максимального періоду параметри повинні задовольняти теоремі Халла-Добелла. [12]

Однак лінійність робить ЛКГ криптографічно неспроможними. З невеликої кількості послідовних значень зловмисник може відновити параметри через систему лінійних рівнянь і передбачити всю послідовність. Також є додаткові недоліки, саме чутливість до параметрів (поганий вибір призводить до коротких періодів) та «гратчаста структура» в багатовимірних тестах (спектральний тест), де точки лежать на гіперплощинах. Історичний приклад, такий як RANDU ($a = 65539,\ m = 2^{31}$), що мав сильні кореляції між трійками чисел, спотворюючи наукові дослідження.[13] ЛКГ непридатні для будь-яких завдань, де потрібна непередбачуваність.

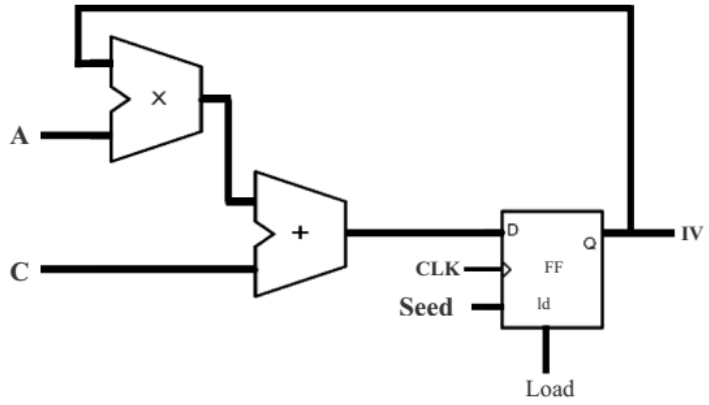


Рисунок 1.2 – Апаратна схема LCG [15]

1.2.2 Вихор Мерсенна: приклад високоякісної статистичної випадковості проти криптографічної незахищеності

Вихор Мерсенна (Mersenne Twister, MT), розроблений у 1997 році Макото Мацумото та Такудзі Нісімурою, є значним удосконаленням PRNG. Версія MT19937 має період $2^{19937} - 1$ (просте число Мерсенна) і внутрішній стан – масив з 624 32-бітних чисел (19968 біт). Процес включає [14]:

1) Ініціалізацію: Зерно заповнює масив стану.

2) Перетворення стану: Коли стан вичерпано, застосовується лінійне рекурентне співвідношення над полем $\mathrm{GF}(2)$ для нового набору. Саме цей процес архітектурно забезпечує оновлення внутрішнього стану, як показано на рисунку 1.3.

3) Видачу числа : Побітові операції (зсуви, XOR) покращують рівномірність.

Вихор Мерсенна (Mersenne Twister, MT), особливо у поширеній версії MT19937, демонструє видатні характеристики для статистичних застосувань. Його переваги включають наддовгий період ($2^{19937} - 1$) та рівномірний розподіл у високих розмірностях (до 623), що в поєднанні з високою продуктивністю зробило його стандартом у таких середовищах, як Python, R, MATLAB та Excel.[14]

Не зважаючи на ці переваги, MT не є криптографічно стійким. Його фундаментальний недолік – це лінійна рекурентність. Це означає, що, проаналізувавши 624 послідовних вихідних значення, можна повністю відновити внутрішній стан генератора, оскільки операція «загартовування» (tempering) є оборотною. Відновлення стану дозволяє зловмиснику передбачити всі майбутні значення. Також, серед інших проблем можна виділити повільну дифузію (схожі «зерна» дають корельовані послідовності) та провал деяких суворих статистичних тестів, наприклад, з набору TestU01.

Широке розповсюдження MT як генератора «за замовчуванням» створило небезпечну ситуацію. Розробники часто помилково використовують його у чутливих до безпеки контекстах, що призводить до вразливостей. Таким чином, MT є ефективним інструментом для симуляцій та моделювання, але абсолютно непридатним для завдань, що вимагають криптографічної непередбачуваності.

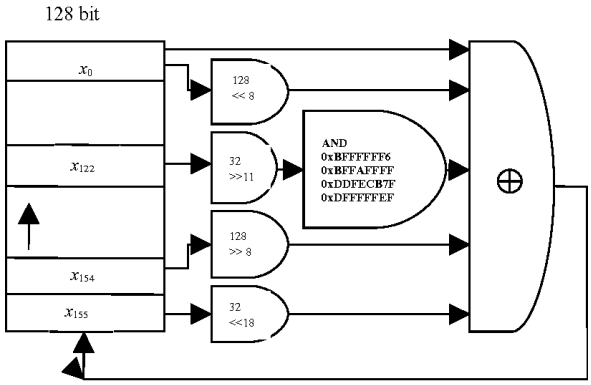


Рисунок 1.3 - Архітектура генерації нового стану у «Вихорі Мерсенна» (Mersenne Twister) [16]

Основні переваги та недоліки ЛКГ у порівнянні з вихором Менсона подано в таблиці 1.1, де продемонстровано їх відмінності за структурою, періодом та криптографічною стійкістю.

Таблиця 1.1 – Порівняльний аналіз класичних PRNG

Параметр	ЛКГ (наприклад, RANDU)	Вихор Мерсенна (MT19937)
Період	До m (короткий)	$2^{19937}-1$ (величезний)
Внутрішній стан	1 число	624 чисел (19968 біт)
Швидкість	Висока	Висока
Статистичні властивості	Погані (кореляції)	Добрі (рівномірність)
Криптографічна стійкість	Відсутня	Відсутня
Застосування	Прості симуляції	Наукові моделі, ігри

1.3. Справжні генератори випадкових чисел (TRNG)

Справжні генератори випадкових чисел (TRNG, від англ. True Random Number Generators) є недетермінованими системами, що генерують послідовності на основі непередбачуваних фізичних процесів.[8] На відміну від PRNG, TRNG не залежать від початкового зерна чи алгоритмів, а використовують ентропію з навколишнього середовища, забезпечуючи справжню випадковість. Через це вони застосовуються в криптографії та симуляціях, де потрібна непередбачуваність. Архітектура TRNG включає джерело ентропії в основі якого лежить фізичний процес, збір даних (сенсори, ампліфікатори), оцифровку (АЦП) та постобробку (для усунення дефектів, наприклад, через XOR або геш-функції).

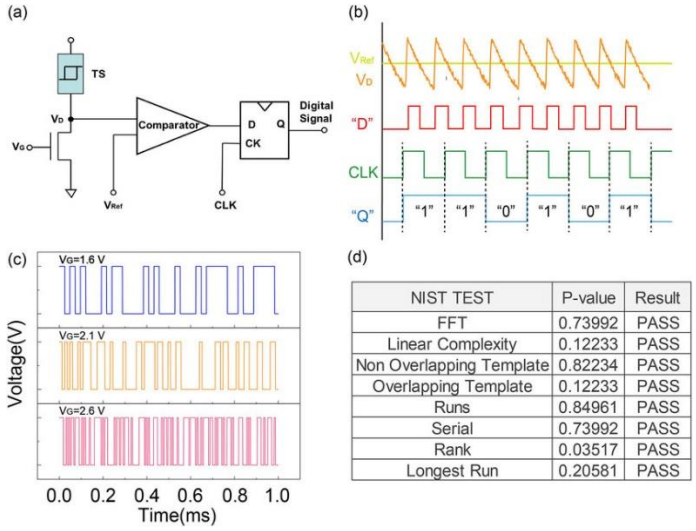
Якість TRNG оцінюється за рівнем ентропії (мін-ентропія як міра справжньої випадковості), швидкістю генерації та стійкістю до зовнішніх впливів.[17] Однак фізична природа робить їх чутливими до умов середовища, що може вводити кореляції чи зміщення. TRNG забезпечують вищу безпеку, ніж PRNG, але вимагають апаратної реалізації та перевірки на випадковість тестами NIST. Типову структурну схему TRNG, що ілюструє взаємодію фізичного джерела шуму та блоків обробки, наведено на рисунку 1.4.

1.3.1 Принцип роботи (фізичні процеси – тепловий шум, радіоактивний розпад)

Принцип роботи TRNG базується на стохастичних фізичних явищах, які неможливо передбачити. Схематичну структуру TRNG разом із характерними імпульсними сигналами на ключових етапах, а також приклад результатів

тестування NIST, наведено на рисунку 1.4. Основні джерела випадковості включають:

- Тепловий шум (thermal noise). Виникає через хаотичний рух електронів у провідниках при ненульовій температурі. У практичній реалізації шум від резистора або діода Зенера ампліфікується операційним підсилювачем, оцифровується АЦП (аналогово-цифровим перетворювачем) з високою роздільністю, а потім біти витягуються пороговим методом або через компаратор, де виконується порівняння з референсним рівнем для отримання 0/1. Це джерело дешеве та інтегрується в чіпи, але залежить від температури, що може зменшувати амплітуду шуму. [10]



NIST TEST	P-value	Result
FFT	0.73992	PASS
Linear Complexity	0.12233	PASS
Non Overlapping Template	0.82234	PASS
Overlapping Template	0.12233	PASS
Runs	0.84961	PASS
Serial	0.73992	PASS
Rank	0.03517	PASS
Longest Run	0.20581	PASS

Рисунок 1.4 – Структура та робота TRNG на основі порівняння аналогового сигналу [19]

- Радіоактивний розпад. Використовує випадкові події розпаду радіоактивних ізотопів (наприклад, америцій-241 або цезій-137). Детектор (счетчик Гейгера або сцинтилятор) реєструє альфа або бета частинки, фіксуючи час між подіями, який підпорядковується пуассонівському розподілу. Інтервали перетворюються на біти, наприклад коли парні/непарні значення лічильника або модуляція часу на бінарну послідовність через XOR з фіксованим шаблоном. Таке джерело забезпечує високу ентропію, незалежну від зовнішніх факторів, але, нажаль, вимагаєрадіоактивних матеріалів і може бути повільним (сотні

біт/с).[10] Типову архітектуру TRNG із постобробкою на основі 2-входових XOR-елементів, яка застосовується для усунення кореляцій та підвищення якості випадковості, наведено на рисунку 1.5.

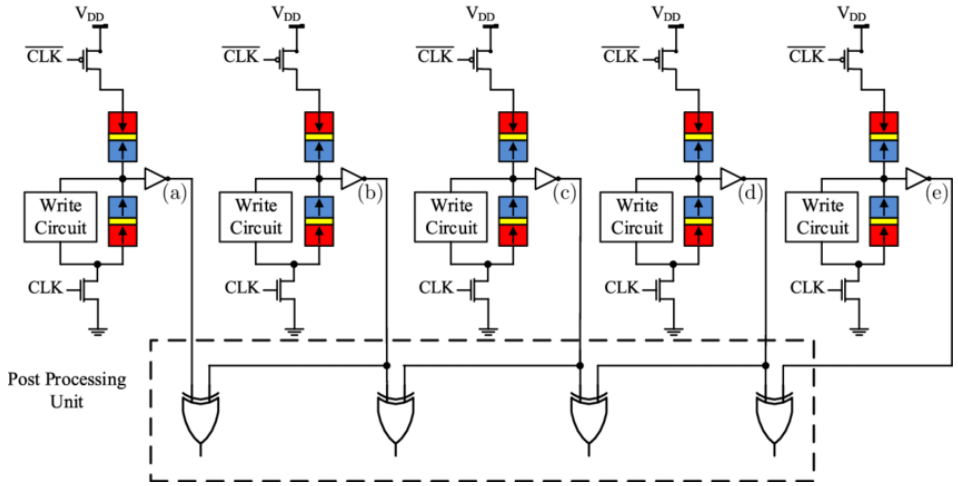


Рисунок 1.5 – Архітектура TRNG з постобробкою на основі 2-входових XOR-елементів [20]

- Інші джерела. атмосферний шум, jitter осциляторів (фазовий шум в кільцевих осциляторах) чи хаотичні системи (лазерний хаос з напівпровідниковими лазерами). TRNG часто комбінують кілька джерел для підвищення ентропії та стійкості. [8]

1.3.2 Недоліки: залежність від середовища, низька швидкість, можливі кореляції

TRNG мають суттєві недоліки, що обмежують їх використання, і вимагають ретельної постобробки для компенсації [17], [18]:

1) Залежність від середовища. Зовнішні фактори значно впливають на джерело ентропії, вводячи передбачуваність. Наприклад, в тепловому шумі при зниженні температури (нижче 0°C) амплітуда шуму падає, зменшуючи ентропію на 20–50%; радіоактивний розпад може бути порушений екрануванням або зовнішнім випромінюванням, як в космічних апаратах, де космічна радіація додає непередбачувані артефакти. Електромагнітні перешкоди (EMI) від сусідніх пристроїв можуть індукувати патерни в jitter осциляторів.

2) Низька швидкість обмежена швидкістю фізичних процесів, що робить TRNG непридатними для реального часу. Наприклад, шумові TRNG генерують 100–500 кбіт/с в типових чіпах як STM32, а радіоактивні – лише 10–100 біт/с через рідкість подій розпаду, що недостатньо для високошвидкісного шифрування.

3) Можливі кореляції. «Сирі» дані рідко бувають ідеальними, тому часто виникає статистичний перекіс, коли через недосконаліть обладнання ми отримуємо, наприклад, 60% одиниць замість рівномірного розподілу. Інша проблема – це коли біти «залипають» або повторюються через інерцію електроніки (так звана автокореляція). Це різко знижує якість випадковості (мін-ентропію), роблячи послідовність вразливою до статистичних атак.

1.3.3 Обмеження використання у високобезпечних системах

У системах із підвищеними вимогами до безпеки (військова сфера, фінанси, криптографія) застосування TRNG має суттєві обмеження через низку ризиків, які вимагають додаткових заходів захисту [17], [18]:

1) Вразливість до фізичних атак. Апаратні генератори чутливі до маніпуляцій через сторонні канали (side-channel attacks). Зловмисник може штучно змінити умови роботи пристрою: наприклад, нагрівання чіпа (температурна атака) здатне зменшити тепловий шум і знизити ентропію. Інший метод – використання електромагнітного випромінювання (атаки типу TEMPEST) для створення кореляцій. Реальний приклад такої загрози – компрометація генераторів у смарт-картах через електромагнітні перешкоди від мобільних телефонів.

2) Складність сертифікації. TRNG потребують суворої валідації за стандартами, такими як NIST SP 800-90B (оцінка мін-ентропії) або європейський AIS-31. Проте навіть сертифіковані рішення (наприклад, Intel RDSEED, що відповідає FIPS 140-2) можуть мати динамічні дефекти, які лабораторні тести не виявляють. Це вимагає впровадження механізмів безперервного моніторингу якості у реальному часі.

3) Проблеми масштабованості та вартості. Апаратні генератори дорожчі та менш портативні порівняно з псевдовипадковими (PRNG). Наприклад, ціна TRNG на основі радіоактивного розпаду може перевищувати $1000 через регуляторні

вимоги. Тому оптимальним рішенням вважається гібридний підхід (DRBG), де TRNG забезпечує ентропію лише для ініціалізації швидкого криптографічного алгоритму, наприклад ChaCha20. Це дозволяє досягти швидкості передачі даних на рівні Гбіт/с, зберігаючи високий рівень безпеки.

Узагальнений порівняльний аналіз основних фізичних джерел випадковості, що застосовуються у сучасних генераторах істинно випадкових чисел, подано в таблиці 1.2.

Таблиця 1.2 – Порівняльний аналіз джерел TRNG

Джерело	Переваги	Недоліки	Швидкість генерації	Приклади реалізації	Рівень ентропії (біт/біт)	Вартість
Тепловий шум	Дешеве, інтегрується в чіпи	Залежить від температур, bias	Кілобіти/с – Мбіт/с	Intel DRNG, STM32 RNG	0.5–0.9	Низька
Радіоактивний розпад	Висока ентропія, незалежність	Радіоактивні матеріали, повільно, регуляції	Десятки – сотні біт/с	HotBits, Random.org (частково)	Близько 1.0	Висока
Осцилятор jitter	Швидке, апаратно просте	Вплив EMI, кореляції	Мегабити/с – Гбіт/с	Qualcomm TrustZone, ARM TRNG	0.7–0.95	Середня

1.4 Квантові генератори випадкових чисел (QRNG)

Квантові генератори випадкових чисел (QRNG) є вершиною технології генерації випадковості, оскільки вони використовують фундаментальні принципи квантової механіки для створення послідовностей, що є за своєю суттю непередбачуваними. На відміну від класичних TRNG, які покладаються на складні та хаотичні, але в принципі детерміновані фізичні процеси, QRNG використовують явища, що є істинно стохастичними згідно з сучасним розумінням фізики [26]. Ця унікальна властивість робить їх ідеальним джерелом ентропії для найвимогливіших криптографічних застосувань, де доказова непередбачуваність є ключовою вимогою [23].

1.4.1 Квантова перевага: фундаментальна невизначеність

Основна перевага QRNG над усіма іншими типами генераторів полягає в тому, що джерело їхньої випадковості є фундаментально непередбачуваним, а не просто складним для моделювання. Класичні TRNG, що використовують тепловий шум або джитер осциляторів, генерують випадковість, яка походить від надзвичайно складної системи з величезною кількістю змінних. Теоретично, якби можна було знати точний стан кожної частинки в системі, її майбутню поведінку можна було б передбачити. Таким чином, їхня випадковість базується на нашому «незнанні» повної інформації про систему [26].

QRNG, навпаки, використовують такі явища, як квантова суперпозиція та заплутаність. Згідно з принципом невизначеності Гейзенберга, результат вимірювання квантової системи, що перебуває в суперпозиції, є імовірнісним і не може бути передбачений заздалегідь, навіть за наявності повної інформації про систему до вимірювання [21]. Ця властивість забезпечує доказову випадковість, що не залежить від складності системи чи обчислювальних можливостей зловмисника, а є невід'ємною властивістю природи. Це робить QRNG значно надійнішими для генерації криптографічних ключів та інших параметрів безпеки, оскільки їхня випадковість не може бути скомпрометована шляхом кращого моделювання фізичного процесу.

1.4.2 Еталонна архітектура та принцип роботи QRNG

Хоча технічні деталі реалізації різних QRNG можуть відрізнятися, їхня побудова зазвичай підпорядковується єдиній еталонній моделі. Ця модель описує процес перетворення квантових явищ на криптографічно стійкий ключ і складається з чотирьох ключових етапів, функціональна схема яких наведена на рисунку 1.6 [21], [22]:

1) Джерело квантової ентропії (Quantum Entropy Source): Це центральний елемент генератора, де відбувається фундаментально випадковий квантовий процес. На цьому етапі готується квантовий стан (наприклад, фотон у суперпозиції) і проводиться його вимірювання, результат якого є непередбачуваним.

2) Перетворювач та оцифрування (Transducer and Digitization): Результат квантового вимірювання (наприклад, спрацювання одного з двох фотодетекторів) є фізичним, часто аналоговим, сигналом. Цей сигнал повинен бути перетворений на цифровий формат. Спеціальні перетворювачі та аналого-цифрові перетворювачі (АЦП) фіксують результат і генерують початковий потік «сирих» бітів.

3) Постобробка (Post-processing / Randomness Extraction): «сирі» біти майже ніколи не є ідеально випадковими через недосконаліть обладнання та зовнішні шуми. Тому вони пропускаються через алгоритм видобування випадковості. Цей алгоритм, часто реалізований на ПЛІС (FPGA) або комп'ютері, бере довгу, потенційно зміщену послідовність і «стискає» її в коротшу, але статистично бездоганну послідовність. Криптографічні геш-функції, як-от SHA-3, є поширеним вибором для цього етапу [25].

4) Моніторинг стану (Health Monitoring): Надійні QRNG включають вбудовані тести, які безперервно контролюють фізичні параметри джерела ентропії. Це дозволяє виявити аномалії, що можуть свідчити про несправність або спробу зовнішньої атаки на генератор (наприклад, спробу «засліплення» детекторів). Цей етап є критично важливим для забезпечення довіри до вихідних даних [26].

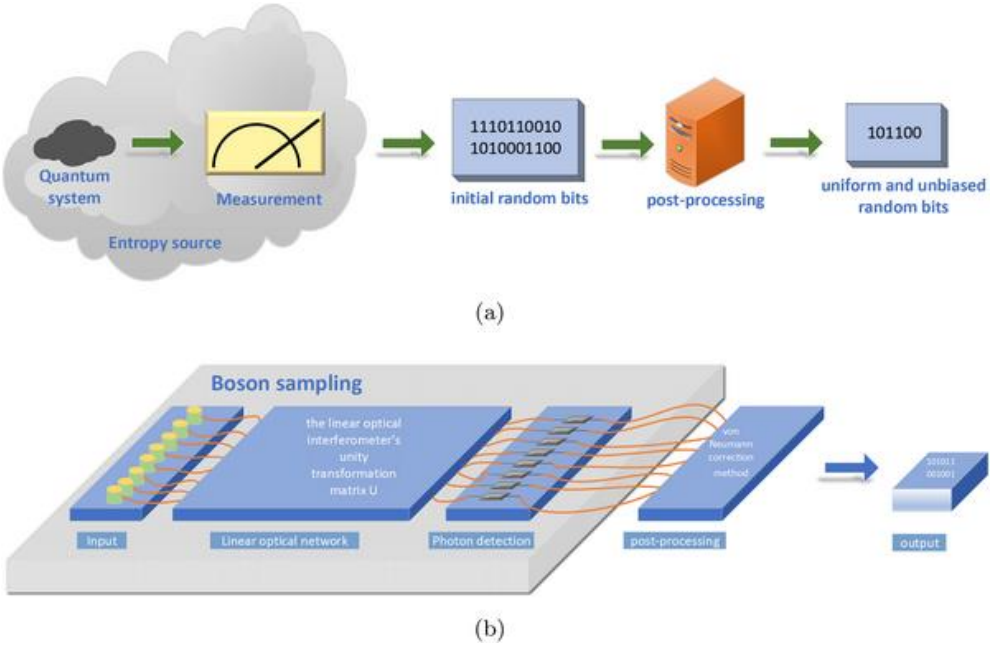


Рисунок 1.6 – Функціональні схеми QRNG: a) загальна блок-структура; b) приклад реалізації з мультифотонною інтерференцією. [27]

1.4.3 Джерела квантової випадковості та математична модель вимірювання

Існує декілька основних методів реалізації QRNG. Розглянемо два найпоширеніші підходи, включаючи їх математичний опис.

Перший – це фотонні події та дільник променя. Цей метод використовує непередбачувану поведінку окремих фотонів. У типовій схемі (рис 1.7) одиночний фотон спрямовується на напівпрозоре дзеркало (дільник променя), яке з однаковою ймовірністю (50/50) пропускає фотон або відбиває його. Математично, стан фотона можна описати як кубіт. Якщо позначити стан «пропускання» як $|0\rangle$ і стан «відбиття» як $|1\rangle$, то після проходження дільника променя фотон опиняється у стані суперпозиції [26], [23]:

$$|\psi\rangle = \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle) \tag{1.2}$$

Це означає, що фотон одночасно перебуває в обох станах. Процес вимірювання, що виконується двома детекторами, змушує цей стан «сколапсувати» до одного з базових станів. Згідно з правилом Борна, ймовірність отримати результат $|0\rangle$ або $|1\rangle$ дорівнює квадрату амплітуди відповідного стану:

$$P(0) = \left|\frac{1}{\sqrt{2}}\right|^2 = \frac{1}{2}, P(1) = \left|\frac{1}{\sqrt{2}}\right|^2 = \frac{1}{2} \tag{1.3}$$

Таким чином, результат кожного вимірювання є фундаментально випадковим з імовірністю 50%, що дозволяє генерувати послідовність непередбачуваних бітів.

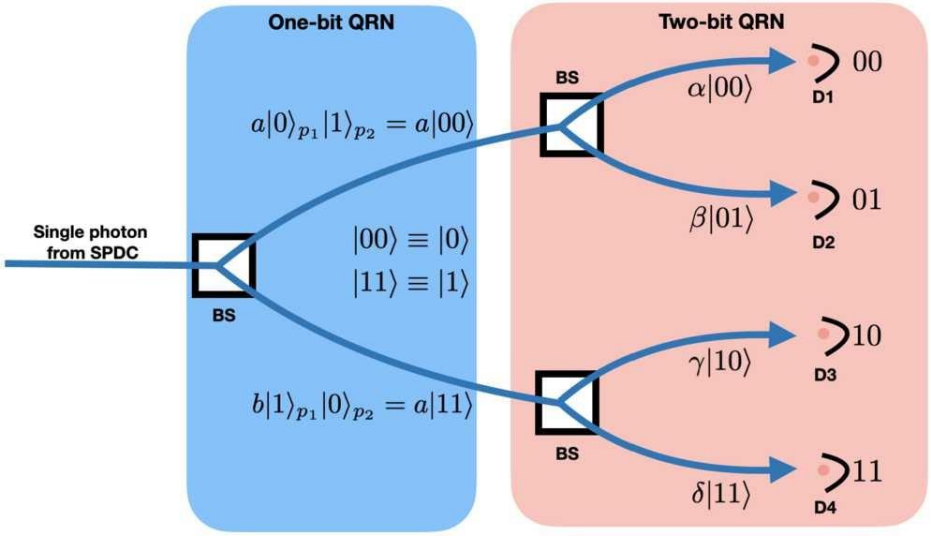


Рисунок 1.7 – Схема QRNG на основі одиночних фотонів та дільників променя [28]

Другий підхід опирається на флуктуації квантового вакууму. Згідно з класичною фізикою, вакуум – це порожній простір. Однак квантова механіка стверджує, що вакуум наповнений «нульовими коливаннями», що призводить до постійного виникнення та зникнення віртуальних частинок. Ці флуктуації електромагнітного поля є фундаментально випадковими. QRNG на основі вакуумних флуктуацій використовують високоточну техніку, що називається гомодинним детектуванням, для вимірювання цих надзвичайно слабких коливань. Виміряний сигнал потім оцифровується для створення послідовності випадкових бітів. Цей підхід є перспективним для створення високошвидкісних та інтегрованих QRNG, оскільки джерело ентропії (вакуум) є всюдисущим і не потребує спеціальних умов.[24]

1.4.4 Кількісна оцінка випадковості та ентропія

Для того, щоб оцінити якість «сирих» даних, отриманих з QRNG, та обґрунтувати необхідність постобробки, використовуються математичні поняття ентропії.

Ентропія Шеннона – це міра середньої невизначеності або інформаційного вмісту джерела. Для джерела X, що генерує символи x_i з імовірностями $p(x_i)$, ентропія Шеннона визначається як [26]:

$$H(X) = -\sum_i p(x_i)\log_2(p(x_i)) \tag{1.4}$$

Для ідеального генератора бітів, де $p(0) = p(1) = 0.5$, ентропія Шеннона дорівнює 1 біту на символ. Якщо ж є зміщення (bias), наприклад $p(0) = 0.6, p(1) = 0.4$, то ентропія буде меншою за 1, що вказує на зменшення невизначеності.

Не менш важливим є поняття мін-ентропія. У криптографії важливіша не середня, а найгірша невизначеність. Мін-ентропія H_∞ визначається через імовірність найбільш імовірного результату:

$$H_\infty(X) = -\log_2\left(\max_i p(x_i)\right) \tag{1.5}$$

Ця величина показує, скільки біт справжньої випадковості можна гарантовано видобути з одного символу. Вона є більш консервативною оцінкою і є ключовою для доведення безпеки криптографічних протоколів. [25]

Також слід врахувати умовну мін-ентропію. У реальному світі зловмисник (Єва, E) може мати доступ до побічної інформації, наприклад, через класичний шум в апаратурі. Умовна мін-ентропія $H_{\min}(X|E)$ кількісно визначає невизначеність результату X для зловмисника, який володіє інформацією E. Саме ця величина є вирішальною для оцінки безпеки QRNG.

2 КРИПТОГРАФІЧНА ГЕШ-ФУНКЦІЯ SHA-3 ЯК ІНСТРУМЕНТ ДЛЯ ОБРОБКИ ВИПАДКОВОСТІ

2.1 Еволюція криптографічних геш-функцій: від Меркла-Дамгарда до губок

Для повного розуміння архітектури геш-функції SHA-3 та її ролі як екстрактора випадковості в системах квантових генераторів випадкових чисел, необхідно розглянути еволюційний шлях розвитку криптографічних геш-функцій. Вибір SHA-3 не є випадковим, адже він є відповіддю на фундаментальні обмеження та вразливості попередніх стандартів, таких як MD5, SHA-1 та SHA-2, які базувалися на класичній конструкції Меркла-Дамгарда. [9]

Конструкція Меркла-Дамгарда (Merkle–Damgard), запропонована незалежно Ральфом Мерклом у 1979 році та Іваном Дамгардом у 1989 році, стала основою для багатьох поширених геш-функцій, включаючи MD5, SHA-1 та сімейство SHA-2.[9] Її загальна структура, зображена на рисунку 2.1, забезпечує побудову стійкої до колізій геш-функції на базі стійкої до колізій функції стиснення, з доведеною безпекою за умови правильного доповнення.

Принцип роботи конструкції полягає в наступних кроках:

1) Доповнення (Padding): Вхідне повідомлення M доповнюється, щоб його довжина стала кратною розміру блоку (наприклад, 512 біт). Цей крок включає доповлення біта «1» (наприклад, 0x80), нулів для вирівнювання та кодування довжини оригінального повідомлення в останньому блоці, що запобігає атакам на префікси.

2) Розбиття: Доповнене повідомлення розділяється на блоки фіксованого розміру $M_1, M_2, \dots, M_n$.

3) Ітеративне стиснення: Обробка починається з фіксованого ініціалізаційного вектора (IV). Функція стиснення f послідовно застосовується до кожного блоку :

$$H_0 = IV,$$
$$H_1 = f(H_0, M_1), \tag{2.1}$$

$$H_2 = f(H_1, M_2),$$
$$\vdots$$
$$H_n = f(H_{n-1}, M_n).$$

4) Результат: Фінальний геш – це H_n, який є повним внутрішнім станом після останньої ітерації. Також буває застосовується функція фіналізації для отримання гешу потрібної довжини.

Ця конструкція проста, ефективна та послідовна, що робить її зручною для реалізації. Однак ключова особливість – це повна відповідність геш-виходу внутрішньому стану – стає джерелом вразливостей. [30]

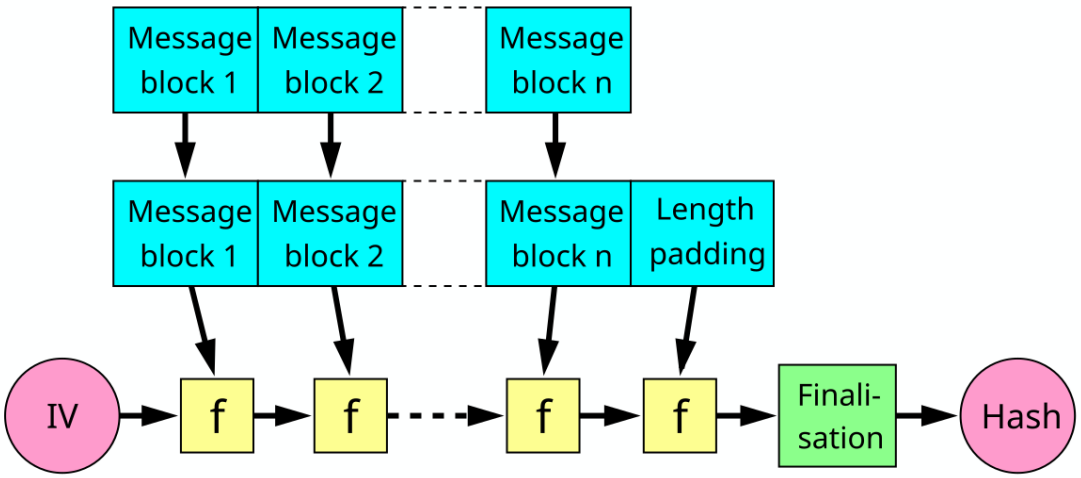


Рисунок 2.1 – конструкція Меркла-Дамгарда [32]

Хоча ця конструкція є простою та ефективною, її ключова особливість котра полягає у відповідності виходу внутрішньому стану – створює серйозні вразливості. Стійкість конструкції Меркла-Дамгарда похитнулася через два типи проблем:

1) Практичні атаки на колізії в MD5 та SHA-1. MD5 та SHA-1 визнані скомпрометованими через ефективні методи пошуку колізій (двох різних повідомлень з однаковим гешем) за допомогою диференціального криптоаналізу. Складність таких атак значно нижча за brute-force (наприклад, для MD5 колізії знаходять за секунди), що унеможливлює їх використання для перевірки цілісності чи цифрових підписів. NIST рекомендує відмовитися від цих алгоритмів з 2010 року. [29]

2) Теоретична вразливість SHA-2, а саме атака на розширення повідомлення (Length Extension Attack). SHA-2 стійкий до відомих колізій, але успадкував архітектурну ваду Меркла-Дамгарда. Оскільки вихідний геш є повним внутрішнім станом системи, зловмисник може використати його як новий вектор ініціалізації для продовження гешування, як це візуально продемонстровано на рисунку 2.2. Атака дозволяє зловмиснику, знаючи $H(M)$ та довжину M, обчислити $H(M \parallel \text{padding} \parallel M_2)$ без знання M. Це відбувається, бо $H(M)$ є повним внутрішнім станом, який можна використати як новий IV для продовження гешування. [30]

Особливо небезпечно для наївних MAC-схем типу $H(\text{secret} \parallel \text{data})$, де зловмисник може фальсифікувати повідомлення, додаючи власні дані (наприклад, дублюючи параметри в API-запитах). [9] Укорочені варіанти SHA-2 (SHA-384, SHA-512/256) менш вразливі завдяки ширшому внутрішньому стану.

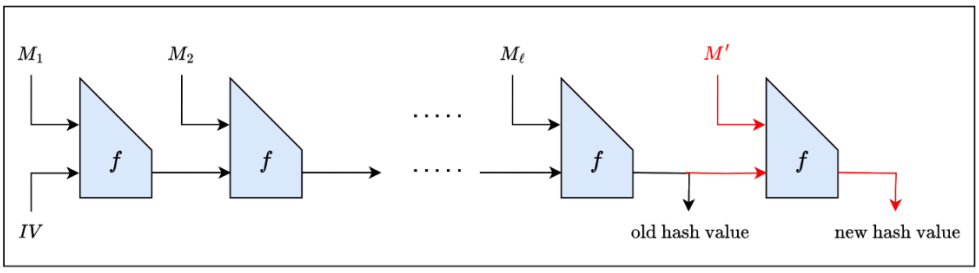


Рисунок 2.2 – Візуальне представлення атаки розширення повідомлення.[33]

Інші вади включають атаки на мультиколізії (пошук багатьох колізій з невеликими зусиллями) та herding-атаки, де зловмисник створює колізії для конкретних документів.

Компрометація SHA-1 та теоретичні проблеми SHA-2 створили потребу в принципово новій архітектурі, вільній від вад Меркла-Дамгарда. NIST ініціював конкурс на SHA-3 у 2007 році, щоб забезпечити більший вібір архітектур. Переможцем став Keccak (2012 рік), стандартизований як SHA-3 у 2015 році. [31]

SHA-3 базується на губковій конструкції (sponge construction), де внутрішній стан поділяється на «швидкість» (rate, r) для взаємодії з входом через XOR та «ємність» (capacity, c) котра є прихованою частиною для безпеки. Також ця конструкція включає фази «вбирання» (absorbing) даних з перестановкою Keccak-f та «вижимання» (squeezing) виходу. Це унеможливлює відновлення повного стану

з гешу, усуваючи length extension attack на архітектурному рівні.[31] Крім того, губка підтримує змінну довжину виходу (XOF, як SHAKE), що ідеально для екстракторів випадковості .

Таким чином, SHA-3 представляє нове покоління геш-функцій, стійке до відомих вад попередників, що робить її оптимальною для криптографічних завдань, включаючи «відбілювання» квантових даних у QRNG .

2.2 Конкурс NIST та вибір SHA-3 (Keccak)

Далі розглянемо процес відбору нового стандарту криптографічної геш-функції SHA-3, організований Національним інститутом стандартів і технологій США (NIST). Конкурс був ініційований через виявлені вразливості в MD5 та SHA-1, а також потенційні ризики для SHA-2, які базуються на конструкції Меркла-Дамгарда. Метою було створення альтернативного алгоритму з принципово іншою архітектурою для диверсифікації криптографічних стандартів і забезпечення стійкості до майбутніх загроз [34]. NIST діяв проактивно, не чекаючи компрометації SHA-2, а зосередившись на стратегічному хеджуванні ризиків, аби мати готовий стандарт на випадок класових атак.

Конкурс тривав з 2007 по 2012 (таб. 2.1) рік і став зразком відкритості, подібно до процесу вибору AES, з активним залученням глобальної криптографічної спільноти для аналізу кандидатів [34]. Він розпочався з 64 заявок, з яких відібрали 51 для першого раунду, 14 для другого та 5 фіналістів для третього. Процес включав публічні форуми, бенчмаркінг та глибокий криптоаналіз, що завершився конференцією в березні 2012 року. 2 жовтня 2012 року NIST оголосив переможцем алгоритм Keccak. Ця хронологія забезпечила ретельну перевірку, орієнтовану на безпеку, продуктивність та інновації.

Таблиця 2.1 – Ключові етапи конкурсу NIST SHA-3 (2007–2012)

Дата	Подія	Кількість кандидатів
2 листоп. 2007	Офіційне оголошення конкурсу SHA-3	N/A
31 жовт. 2008	Кінцевий термін подачі заявок	64
10 груд. 2008	Оголошення кандидатів 1-го раунду	51

Продовження таблиці 2.1

Дата	Подія	Кількість кандидатів
24 лип. 2009	Оголошення кандидатів 2-го раунду	14
9 груд. 2010	Оголошення 3-го раунду (фіналістів)	5
2 жовт. 2012	Оголошення переможця	1 (Keccak)

2.2.1 Фіналісти конкурсу: Огляд та порівняння

До фінального раунду дійшли п’ять алгоритмів, які пройшли роки інтенсивного криптоаналізу та вважалися найперспективнішими, а саме BLAKE, Grøstl, JH, Keccak та Skein. Кожен з них представляв унікальний підхід до дизайну геш-функцій, відходячи від традиційної конструкції Меркла-Дамгарда та пропонуючи інноваційні рішення для забезпечення безпеки [29]. BLAKE і Skein, наприклад, спиралися на ARX-операції (Addition-Rotation-XOR), що забезпечувало високу продуктивність у програмному забезпеченні, тоді як Grøstl використовував елементи, подібні до AES, для міцних пермутацій. JH і Keccak, у свою чергу, акцентували увагу на перестановкових структурах, що дозволило досягти високого запасу міцності та стійкості до відомих атак.

Ці алгоритми оцінювалися за комплексом критеріїв, включаючи відсоток неатакованих раундів, продуктивність у програмному та апаратному забезпеченні, гнучкість, а також індиференціабельність – ключовий показник стійкості до диференціального криптоаналізу [29]. Хоча всі фіналісти демонстрували оптимальні докази безпеки для колізій, передобразів та інших властивостей, вони відрізнялися за балансом цих параметрів. Детальне порівняння наведено в Таблиці 2.2, яка ілюструє сильні та слабкі сторони кожного кандидата на основі даних третього раунду конкурсу.

Таблиця 2.2 – Порівняння фіналістів SHA-3 за ключовими критеріями (на основі третього раунду)

Алгоритм	Запас міцності (неатаковані раунди, %)	Продуктивність у ПЗ (відносно SHA-2)	Продуктивність в апаратурі (throughput/area)	Дизайн та гнучкість	Індиференціабельність (біт)
BLAKE	71%	Висока (часто краща)	Середня	HAIFA, ARX, оптимальні докази	128/256 (256/512-біт варіанти)
Grøstl	40%	Низька (без AES-NI)	Низька	Chop-MD, AES-подібний, оптимальні докази	256/256
JH	38%	Низька	Низька	Sponge-подібний, субоптимальний для 512-біт	170/170
Keccak	79%	Середня (конкурентна)	Висока (2-4x SHA-2)	Sponge, гнучкий XOF, оптимальні докази	256/512
Skein	56%	Висока (лідер на ARM)	Середня	Chop-MD, tweakable, оптимальні докази	256/256

Як видно з таблиці, Keccak вирізнявся найбільшим запасом міцності та найкращою апаратною ефективністю, тоді як BLAKE і Skein лідирували в програмній швидкості, а Grøstl і JH мали менші маржі безпеки. Ця різноманітність підтвердила ефективність конкурсу в стимулюванні нових ідей, але також підкреслила, що вибір переможця залежав від комплексної оцінки, орієнтованої на довгострокові потреби криптографії.

2.2.2 Аналіз причин перемоги Keccak

NIST обрав Keccak як переможця, підкресливши його комплексну перевагу над іншими фіналістами за сукупністю ключових критеріїв: архітектура, елегантність дизайну, продуктивність та гнучкість [35], [36]. Алгоритм,

розроблений командою на чолі з Joan Daemen (співавтором AES), найкраще задовольняв ключову вимогу конкурсу, спрямовану на створення архітектурної альтернативи стандарту SHA-2. На відміну від BLAKE чи Skein, які зберігали елементи традиційних конструкцій, Keccak пропонував принципово інший підхід, що робило його стійким до структурних вразливостей, таких як атаки на розширення повідомлення [29].

Однією з ключових переваг став прозорий дизайн Keccak, який полегшував криптоаналіз і забезпечував найбільший запас міцності (79% неатакованих раундів), перевершуючи Grøstl (40%) та JH (38%) [36, 29]. NIST відзначив, що ця ясність конструкції дозволила досягти високого рівня впевненості в стійкості до диференціального та лінійного криптоаналізу, навіть при зменшенні кількості раундів, чого бракувало конкурентам з меншими маржами, як Skein (56%) чи BLAKE (71%). Безпека Keccak була не просто припущеною, а підкріпленою оптимальними доказами для всіх властивостей геш-функцій.

Щодо продуктивності, то в апаратних реалізаціях (ASIC та FPGA) Keccak показав найкращі результати. За критерієм пропускної здатності на одиницю площі він у 2–4 рази перевершив як SHA-2, так і інших конкурентів [29]. Це стало вирішальним фактором для майбутніх технологій, таких як IoT та вбудовані системи, де ресурси обмежені. Хоча в програмному забезпеченні Keccak поступався BLAKE та Skein за швидкістю на CPU, його баланс продуктивності переважив, особливо порівняно з повільнішими Grøstl і JH. NIST орієнтувався на довгострокову перспективу, де апаратна ефективність набувала все більшої ваги.

Нарешті, гнучкість Keccak виявилася унікальною перевагою, адже його конструкція дозволяла генерувати вихід довільної довжини через функції з розширюваним виходом XOF, такі як SHAKE128 та SHAKE256, включені до стандарту FIPS 202 [31]. Це робило алгоритм не просто геш-функцією, а універсальним інструментом для задач на кшталт ключової деривації та генерації псевдовипадкових бітів, що ідеально пасувало для екстракції випадковості в QRNG. У той час як інші алгоритми пропонували лише фіксовані вихідні значення, архітектурна гнучкість дозволила Keccak вийти за межі звичайної геш-функції. Він

став повноцінною криптографічною платформою з найвищим потенціалом адаптації.

Вибір Keccak як основи SHA-3 став стратегічним кроком для доповнення SHA-2, а не його заміни [36]. NIST зберіг упевненість у безпеці SHA-2, але забезпечив криптографічну гнучкість, дозволивши розробникам обирати між перевіреним часом SHA-2 та інноваційним SHA-3. Це посилює стійкість цифрової інфраструктури, особливо в контексті QRNG, де SHA-3 виступає ефективним екстрактором випадковості завдяки своїм перевагам.

2.3 Принцип роботи SHA-3: детальний розбір

В основі алгоритму SHA-3 лежить інноваційна губкова конструкція, яка кардинально відрізняється від ітеративного підходу Меркла-Дамгарда, використовуваного в попередніх стандартах, таких як SHA-1 та SHA-2. Ця конструкція забезпечує високий рівень безпеки, стійкість до відомих атак та гнучкість, зокрема можливість генерації виходу довільної довжини, що є ключовим для функцій з розширюваним виходом, таких як SHAKE128 та SHAKE256.[31] Губкова конструкція базується на перестановці Keccak-f, яка гарантує псевдовипадкове перемішування даних, роблячи SHA-3 ідеальним для екстракції ентропії в генераторах випадкових чисел, включаючи квантові (QRNG).

2.3.1 Губкова конструкція (Sponge Construction)

Губкова конструкція – це загальний метод побудови криптографічної функції, що приймає на вхід дані довільної довжини та генерує вихід довільної довжини. Аналогія з губкою полягає в тому, що внутрішній стан алгоритму спочатку «вбирає» вхідні дані, а потім «вичавлює» вихідний геш. Ця модель забезпечує провірену безпеку проти загальних атак, таких як колізії чи передобразів, з теоретичним рівнем стійкості до $c/2$ біт, де c – ємність стану. [31]

Основними компонентами конструкції є:

1) Внутрішній стан (State, S): Масив бітів фіксованої довжини b. Для стандарту SHA-3 $b = 1600$ біт, що представляє собою тривимірний масив розміром $5 \times 5 \times 64$ (рис. 2.3) (5 шарів по 5 стовпців, кожен з 64 біт). [37]

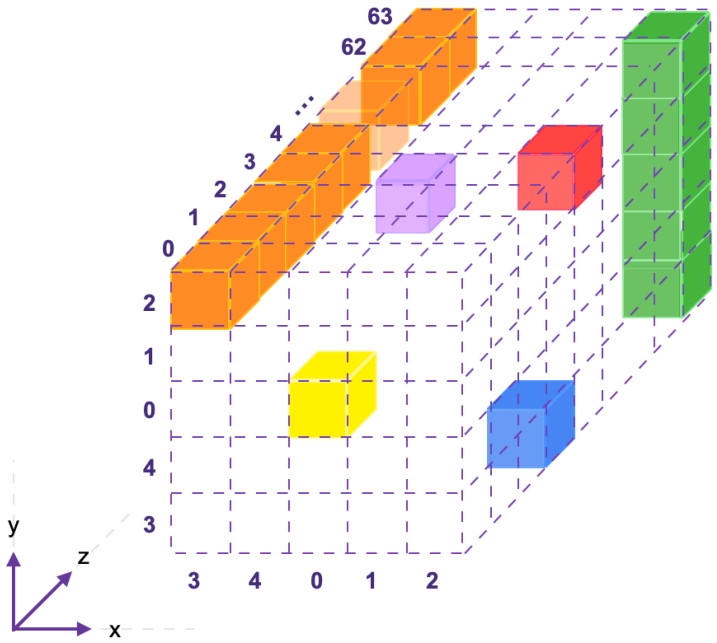


Рисунок 2.3 – Keccak-f 1600-бітний стан як 3D-масив [39]

Відображення між бітами стану s , записаного у вигляді лінійного масиву з 1600 біт, і бітами тривимірного масиву задається формулою: $s[64(5y + x) + z] \mapsto \text{Bit}[x, y, z]$

Тобто, біт з координатами (x, y, z) у 3D-масиві відповідає біту у лінійному масиві з індексом $64(5y + x) + z$. Наприклад, біт a = Bit[2,3,3] відповідає біту з індексом у лінійному масиві:

$$64(5 \cdot 3 + 2) + 3 = 64 \cdot 17 + 3 = 1088 + 3 = 1091.$$

Отже, a = Bit [2,3,3] – це 1092-й біт у послідовності s. Цей біт представлений на малюнку вище синім кольором. [39] Загальні характеристики та приклади внутрішнього стану S для різних координатних позицій узагальнено в таблиці 2.3, що дозволяє наочно простежити відповідність між лінійним поданням бітів та їхнім тривимірним відображенням у структурі.

Таблиця 2.3 – Приклади стану S [39]

x	y	z	s[64x + 320y + z]	Bit[x,y,z]	Колір куба (Рис.2.3)
0	0	0	s[64·0+320·0+0] = s[0]	Bit[0, 0, 0]	Жовтий
2	2	4	s[64·2+320·2+4] = s[772]	Bit[2, 2, 4]	Червоний

Продовження таблиця 2.3

2	3	3	s[64·2+320·3+3] = s[1091]	Bit[2, 3, 3]	Синій
0	2	3	s[64·0+320·2+3] = s[643]	Bit[0, 2, 3]	Фіолетовий
2	2	63	s[64·2+320·2+63] = s[831]	Bit[2, 2, 63]	Зелений
2	1	63	s[64·2+320·1+63] = s[511]	Bit[2, 1, 63]	Зелений
2	0	63	s[64·2+320·0+63] = s[191]	Bit[2, 0, 63]	Зелений
2	4	63	s[64·2+320·4+63] = s[1471]	Bit[2, 4, 63]	Зелений
2	3	63	s[64·2+320·3+63] = s[1151]	Bit[2, 3, 63]	Зелений

2) Функція перестановки (Permutation, f): Внутрішня функція, що псевдовипадково перемішує біти стану. В SHA-3 це функція Keccak-f, яка складається з 24 раундів. [31]

3) Швидкість (Rate, r): Частина стану (перші r біт), з якою безпосередньо взаємодіють блоки вхідного повідомлення через операцію XOR та з якої зчитуються блоки вихідного гешу. Значення r змінюється залежно від варіанту: наприклад, для SHA3-256 $r = 1088$ біт, для SHA3-512 $r = 576$ біт .

4) Ємність (Capacity, c): Частина стану (останні $c = b - r$ біт), яка ніколи не піддається прямому впливу вхідних даних і не зчитується напряму. Для SHA3-256 $c = 512$ біт, що забезпечує стійкість до атак на рівні 256 біт. Саме ємність c діє як «секретна» частина стану, яку зловмисник не може контролювати чи спостерігати, запобігаючи атакам на розширення повідомлення (length extension attacks), характерним для Меркла-Дамгарда. [38]

Процес роботи губки складається з двох фаз:

1) Фаза вбирання, де внутрішній стан S ініціалізується нулями. Вхідне повідомлення M доповнюється за спеціальним правилом: додається біт ‘1’, за ним нулі до кратності r, і в кінці спеціальний суфікс (наприклад, для SHA-3 – ‘01’, де середні біти – нулі). Потім M розбивається на блоки P_i розміром r біт. Для кожного блоку P_i він додається (XOR) до r-частини стану, після чого до всього стану застосовується функція f: $S = f(S \oplus (P_i \parallel 0^c))$. Цей процес повторюється для всіх блоків, забезпечуючи дифузію вхідних даних по всьому стану.

2) Фаза вичавлювання, де після обробки всіх вхідних блоків починається генерація виходу. Перші r біт (Rate) стану копіюються у вихідний потік Z_1. Якщо потрібен довший вихід, до стану S застосовується f, і наступні r біт копіюються як Z_2. Процес повторюється, доки не буде згенеровано вихід необхідної довжини L. Для фіксованих гешів, як SHA3-256, вихід обрізається до 256 біт .

Ця конструкція (рис. 2.4) робить SHA-3 гнучким для QRNG: «сирі» квантові дані вбираються в стан, а вичавлювання генерує рівномірний випадковий потік. Для ілюстрації губкової конструкції наведено схематичне зображення.

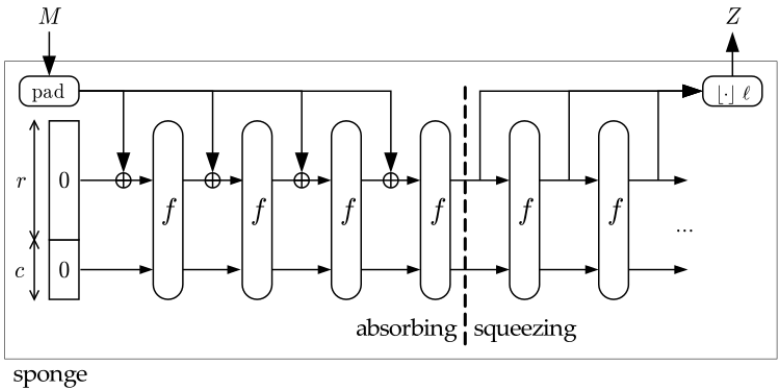


Рисунок 2.4 – Губкова конструкція SHA-3 [31]

2.3.2 Внутрішня перестановка Keccak-f та лавинний ефект

Криптографічна стійкість SHA-3 забезпечується властивостями внутрішньої функції перестановки Keccak-f, яка створює сильний лавинний ефект (avalanche effect): зміна одного біта на вході призводить до непередбачуваної зміни приблизно 50% бітів на виході після кількох раундів.[37] Keccak-f складається з $n_r = 24$ раундів, де стан представляється як тривимірний масив $A[5][5][64]$ (5×5 «планів» по 64 біти) .

Кожен раунд складається з п’яти кроків, які забезпечують поширення впливу, тобто дифузію та заплутування зв’язку, тобто конфузію:

1) θ (Theta): Лінійна операція змішування (рис. 2.5). Кожен біт $A[x, y, z]$ додається (XOR) з парністю (XOR-сумою) двох сусідніх стовпців: $A[x, y, z] \oplus \oplus_{i=0}^{4} A[x-1, i, z] \oplus \oplus_{i=0}^{4} A[x+1, i, z-1]$. Це забезпечує швидку дифузію змін по стовпцях .

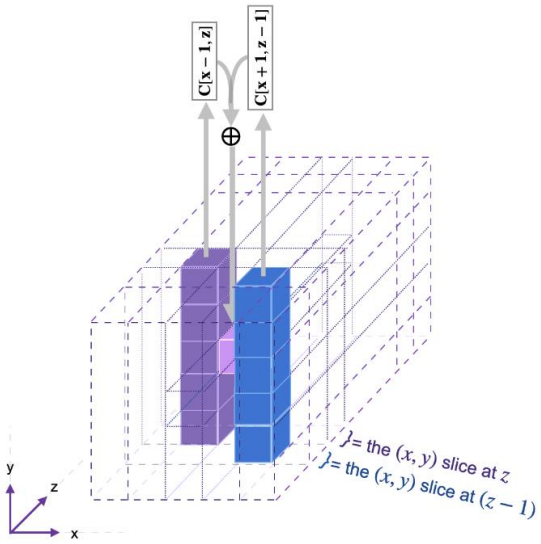


Рисунок 2.5 – відображення застосування до одного біта [39]

2) ρ (Rho): Лінійна операція циклічного зсуву. Кожна з 25 «доріжок» (64-бітних слів $A[x, y,*]$) зсувається на фіксовану величину (від 0 до 62 біт), визначену таблицею ротацій. Наприклад, доріжка не зсувається, – на 1 біт. Це руйнує симетрію та поширює зміни по бітах . [39]

3) π (Pi): Лінійна операція перестановки (рис. 2.6.). Доріжки переміщуються за шаблоном: $A'[x, y] = A[y, (2x + 3y) \bmod 5]$. Це додатково розсіює біти по масиву .

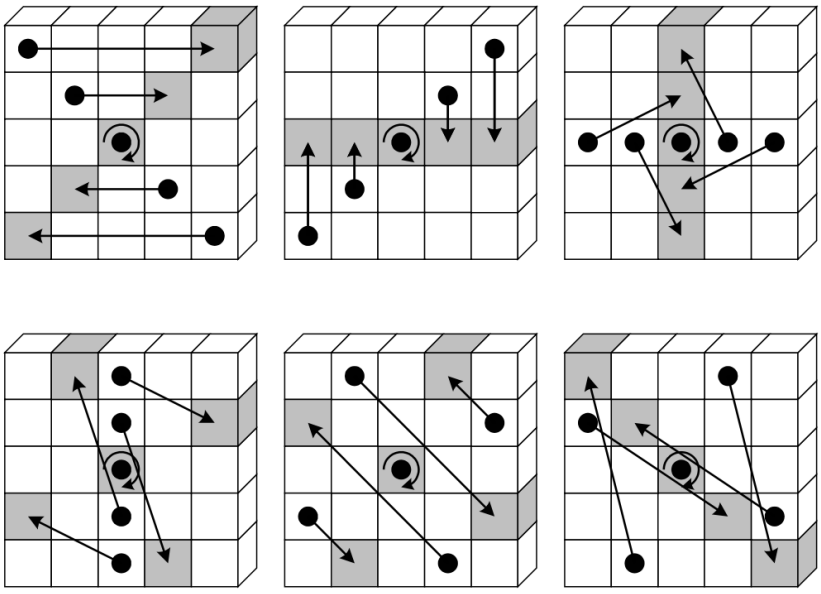


Рисунок 2.6 – Перемішування бітів у зрізі [39]

4) χ (Chi): Єдиний нелінійний крок (рис. 2.7). Застосовується до кожного 5-бітного рядка : $A'[x, y, z] = A[x, y, z] \oplus (\neg A[x + 1, y, z] \wedge A[x + 2, y, z])$. Це S-box-подібне перетворення руйнує лінійні залежності, забезпечуючи односторонність і стійкість до диференціальних атак.

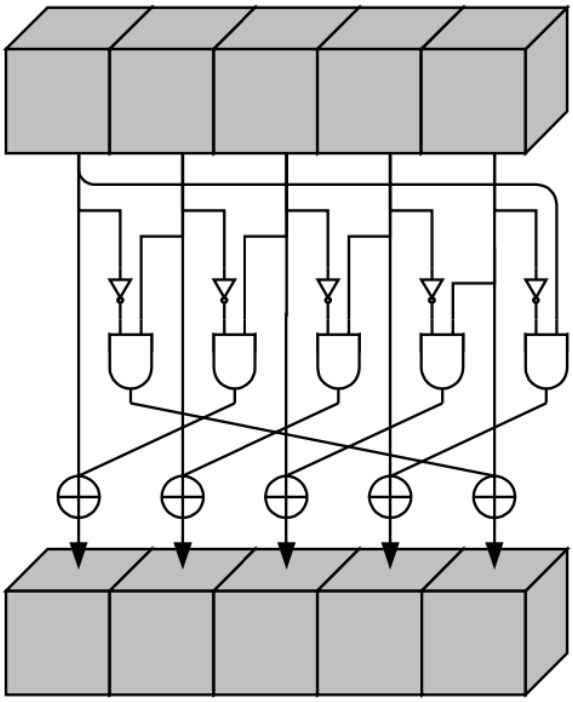


Рисунок 2.7 – Відображення кроку χ [39]

5) ι (Iota): Лінійна операція додавання раундової константи ($RC[i]$, унікальної для кожного раунду від 0 до 23) до однієї доріжки [0,0]: $A[0,0,z] \oplus RC[i][z]$. Це порушує симетрію між раундами та запобігає атакам на фіксовані точки.

Лавинний ефект досягається комбінацією цих кроків: лінійні операції (θ, ρ, π) поширюють зміни, а нелінійний χ робить їх непередбачуваними. Після 24 раундів будь-яка локальна зміна впливає на весь стан.

2.3.3 Концептуальний приклад «відбілювання» даних

Хоча повний числовий приклад для 1600-бітного стану складний, розглянемо концептуальний приклад «відбілювання» дефектних даних з QRNG, таких як зміщення (bias) чи автокореляція. [38]

Припустимо, «сирий» потік від QRNG має дефекти: зміщення (60% ‘0’ vs 40% ‘1’) та автокореляцію (довгі послідовності, наприклад, через oversampling). Приклад дефектного потоку: «0011000110000110...» (повторювані патерни). [38]

Процес перетворення в SHA-3:

1) Вбирання. Потік доповнюється та розбивається на блоки по 1088 біт. Кожен блок XOR-иться з rate-частиною стану (ініціалізованого нулями). Стан у такому випадку виглядає наступним чином: $S = (P_1 \parallel 0^{512}) \oplus S$. [31]

2) Лавинний ефект (24 раунди Keccak-f). Після XOR, застосовується функція f. У першому раунді θ поширює патерни по стовпцях, ρ зсуває їх, π переміщує, χ нелінійно комбінує (наприклад, «000» може стати «101» залежно від сусідів), ι додає константу. Після кількох раундів (наприклад, 5–10) патерни руйнуються, а це значить, що зміщення вирівнюється до ~50/50, кореляції зникають через дифузію. [31]

3) Вичавлювання. З rate зчитується Z_1 (1088 біт), застосовується f, зчитується Z_2 тощо. Як результат – поданий псевдовипадковий потік без дефектів, наприклад, «101010011101...» з рівномірним розподілом. [31]

SHA-3 діє як потужний «відбілювач», перетворюючи «грязні» квантові дані на криптостійку ентропію, ідеальну для QRNG. Для реальних тестів використовуються інструменти як NIST STS, де «відбілені» дані проходять тести на випадковість. [38]

2.4 Проблема «сирих» квантових даних і теоретична необхідність екстрактора випадковості

Квантові генератори випадкових чисел мають фундаментальну перевагу над псевдовипадковими – вони спираються на справжню непередбачуваність квантової механіки, що робить згенеровану випадковість доказово безпечною. [17], [8] Проте між цією теоретичною ідеальністю та практичною реалізацією лежить прірва: будь-яка система вимірювання квантового процесу є класичною, а отже, недосконалою. Вихідний бітовий потік, отриманий безпосередньо від квантового джерела ентропії до будь-якої постобробки (типова структура наведена на рисунку 2.8), називається «сирими» (raw) даними. Саме ці дані, попри квантовий походження, практично

завжди містять статистичні дефекти, що робить їх непридатними для прямого криптографічного використання. [40, 41]

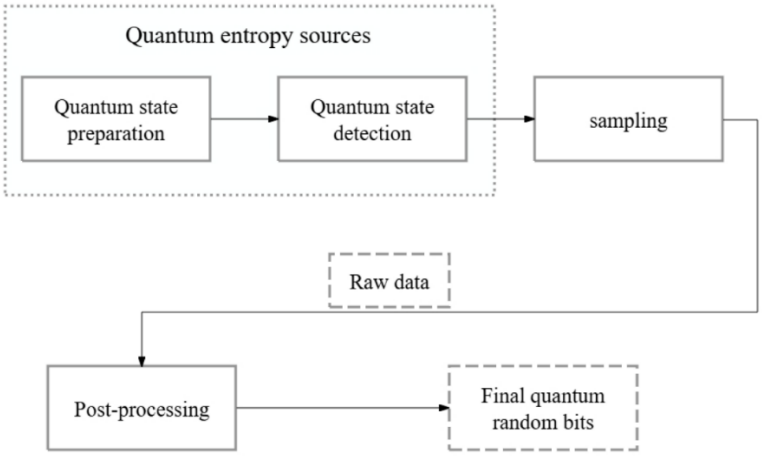


Рисунок 2.8 – Типова структура QRNG [25]

Ці дефекти мають виключно інженерне походження і проявляються у трьох основних формах.

Найпоширеніший – статистичне зміщення (bias), коли ймовірність появи 0 та 1 відхиляється від ідеальних 0.5. Воно виникає через асиметрію чутливості фотодетекторів, нелінійність аналого перетворювача чи навіть мінімальні виробничі відхилення оптичних компонентів (наприклад, дільник променя 49.9%/50.1% замість 50/50).[40] Приклад характерного зміщення у вигляді гістограми сирих даних для QRNG наведено на рисунку 2.9.

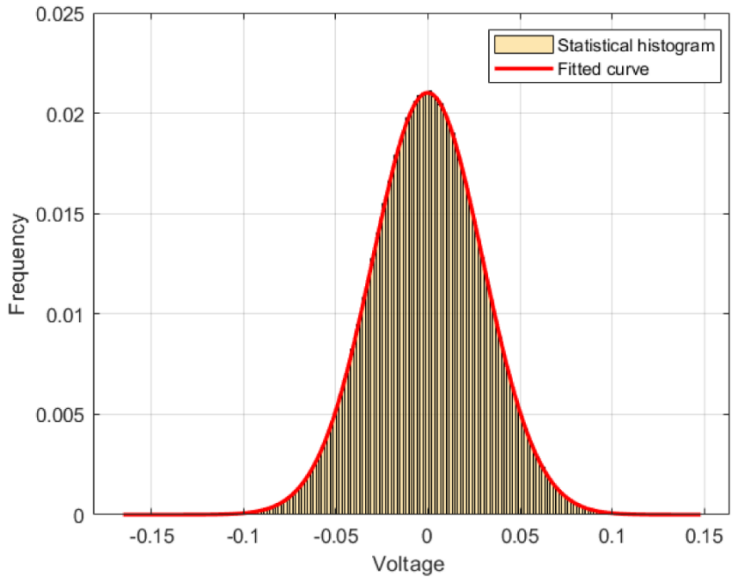


Рисунок 2.9 – Статистична гістограма сирих даних для QRNG на основі шуму ASE. [25]

Значно небезпечніша – це автокореляція, тобто залежність послідовних бітів один від одного. Найчастіше вона з'являється через надмірну дискретизацію, коли частота АЦП значно перевищує реальну смугу пропускання квантового шуму, система багаторазово фіксує один і той самий повільно змінюваний квантовий стан і як результат – довгі серії однакових бітів (десятки чи сотні нулів або одиниць поспіль), що легко виявляється статистичними тестами. [42]

Також, мабуть, найкритичніша проблема пов'язана з забрудненням класичним шумом. Сирий сигнал – це не чиста квантова ентропія, а суміш із детермінованими класичними шумами: тепловий шум резисторів, шум підсилювачів, флуктуації живлення, електромагнітні завади (EMI), перехресні перешкоди в чипі. На відміну від квантового шуму, класичний шум передбачуваний і потенційно керований зловмисником (наприклад, через цілеспрямовану інжекцію EMI). Якщо атакувач може змоделювати хоча б частину класичного шуму, він отримує часткову інформацію про біти – і вся криптографічна безпека системи руйнується. [41] Приклад характерного впливу класичного шуму на сирий сигнал наведено на рисунку 2.10.

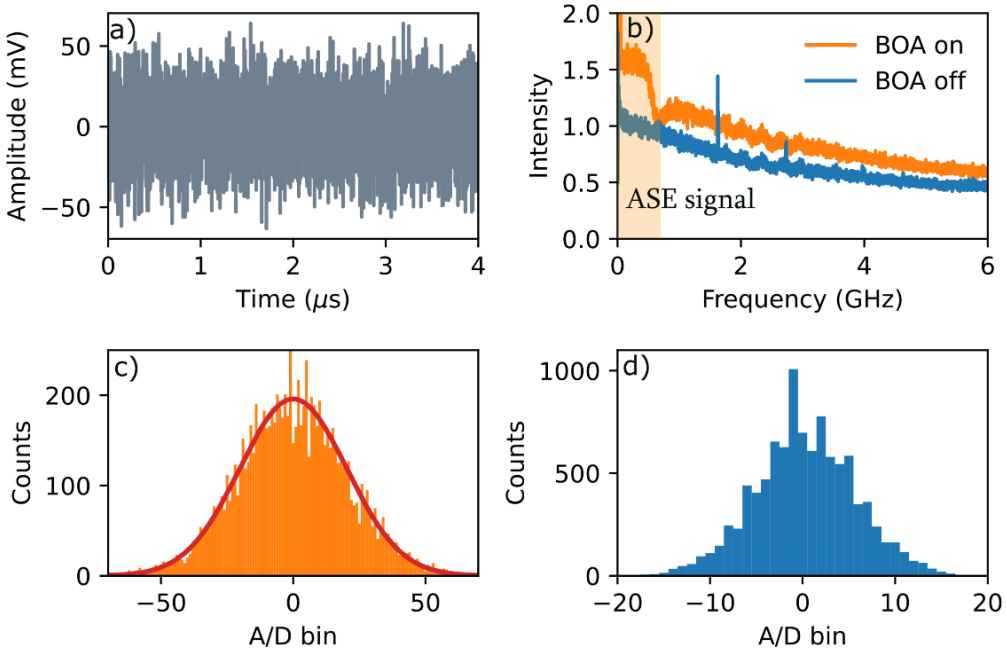


Рисунок 2.10 – Ілюстрація забруднення сирого сигналу класичним шумом [43]

Для безпечного використання сирих даних необхідно точно знати, скільки в них міститься справжньої ентропії. Ентропія Шеннона тут категорично

непридатна, адже джерело, яке в 99% випадків видає «0», матиме високу ентропію Шеннона (близько 0.08 біт на біт), але буде абсолютно марним криптографічно. [8]

Єдиною коректною метрикою є мін-ентропія:

$$H_\infty(X) = -\log_2(\max_v P(X = v)) \quad (2.2)$$

Вона вимірює непередбачуваність у найгіршому випадку – саме те, на що орієнтується атакувач, який знає всі апаратні недоліки системи. Якщо n-бітний блок сирих даних має мін-ентропію k біт, то з нього можна безпечно витягти щонайбільше k бітів справжньої випадковості. [40]

Процес перетворення дефектних сирих даних у статистично ідеальний потік називається екстракцією випадковості. Формально, (k, є)-екстрактор – це детермінована функція Ext: $\{0,1\}^n \to \{0,1\}^m$ (m ≤ k), яка для будь-якого k-джерела (з $H_\infty \geq k$) видає розподіл, є-близький до рівномірного (є зазвичай 2^{-100} ... 2^{-128}). [8]

Лема про залишкове гешування дає точний компроміс:

$$m \approx k - 2\log_2(1/\epsilon) \quad (2.3)$$

Тобто для безпеки рівня 2^{-128} ми втрачаємо приблизно 256 біт ентропії – це плата за доказову невідрізнюваність від ідеалу. [8] Існує три основні підходи до екстракції, але лише один придатний для реальних автономних КГВЧ:

1) Прості статистичні методи (фон Неймана, XOR-дебізери) ефективно прибирають лише зміщення і лише за умови незалежних бітів (IID). За наявності автокореляції (а вона є майже завжди) вони не працюють.

2) Інформаційно-теоретичні екстрактори на основі універсального гешування чи матриць Тепліца дають найвищу безпеку, але вимагають короткого ідеально випадкового зерна так званого «seed».

3) Обчислювальні детерміновані екстрактори (криптографічні геш-функції типу SHA-3, BLAKE3) не потребують зерна, працюють детерміновано і завдяки лавинному ефекту та сильній нелінійності повністю руйнують будь-які кореляції, зміщення та патерни навіть на сильно зашумлених даних. [42]

Саме тому в сучасних високошвидкісних квантових генераторах випадкових чисел (включно з комерційними рішеннями ID Quantique, Quantinuum, Toshiba) як

екстрактор використовується криптографічна геш-функція, а саме SHA-3 (Keccak) є стандартом де-факто, рекомендованим NIST SP 800-90B як «vetted conditioning component». [17], [41] Це не довільний вибір, а єдине рішення, яке одночасно вирішує всі три типи дефектів, не потребує зовнішньої випадковості та працює в реальному часі на швидкостях Гбіт/с.

2.5 SHA-3 як детермінований екстрактор для QRNG

Апробація криптографічної геш-функції SHA-3 як компонента постобробки (екстрактора) у квантових генераторах випадкових чисел (КГВЧ) є кульмінацією аналізу, що поєднує проблеми «сирих» квантових даних, викладені в 2.4, з унікальними архітектурними перевагами алгоритму Keccak, описаними в 2.3. Вибір SHA-3 не є довільним, а являє собою інженерне рішення, що одночасно вирішує три ключові проблеми КГВЧ: статистичне зміщення (bias), автокореляцію та забруднення класичним шумом. У той час як прості статистичні методи, на кшталт екстрактора фон Неймана, здатні коригувати лише зміщення за умови незалежних бітів (IID), вони є абсолютно неефективними проти автокореляції. Саме тому необхідний потужний детермінований (обчислювальний) екстрактор, рекомендований NIST [17], і Keccak є де-факто стандартом для цієї задачі.

Ключова перевага SHA-3 полягає в його здатності до «відбілювання» (whitening) даних, що досягається завдяки лавинному ефекту та нелінійності внутрішньої перестановки Keccak-f [44]. Як детально описано в 2.3.2, кожен із 24 раундів перестановки застосовує п’ять кроків, серед яких χ (Chi) є єдиним нелінійним перетворенням [31]. Саме ця нелінійність у поєднанні з операціями дифузії (θ, ρ, π) забезпечує експоненційне поширення та непередбачуване перетворення будь-яких змін. Коли потік «сирих» даних, що містить статистичні дефекти (наприклад, довгі серії ‘0’ через oversampling або зміщення 49.9%/50.1%), «вбирається» у стан геш-функції, 24-раундове перемішування повністю руйнує ці кореляції. Будь-яка статистична закономірність на вході перетворюється на статистично гомогенний, псевдовипадковий вихід, що концептуально ілюстровано в до цього

Однак найфундаментальніша перевага SHA-3 над попередніми стандартами (як SHA-2) лежить не лише у функції f, а в самій губковій архітектурі [44]. Як було зазнчено раніше, конструкція Меркла-Дамгарда (MD5, SHA-2) вразлива до атак на розширення повідомлення (length extension attack), оскільки її вихідний геш є повним відображенням внутрішнього стану. У контексті КГВЧ це створює ризик, оскільки детермінований класичний шум може бути частково відомий або навіть контрольований зловмисником, що потенціально компрометує стан. Губкова конструкція SHA-3, навпаки, розділяє стан на швидкість (rate, r) та ємність (capacity, c) [31]. Вхідні «сирі» дані та вихідний потік взаємодіють лише з r-частиною стану. c-частина ніколи напряму не спостерігається і не контролюється, але бере повну участь у перестановці Keccak-f. Таким чином, ємність діє як захищений буфер, що архітектурно унеможливлює атаки на розширення і руйнує будь-який зв’язок між «брудним» входом та «чистим» виходом.

Нарешті, практична реалізація SHA-3 як екстрактора стала можливою завдяки його гнучкості, що була однією з причин перемоги Keccak у конкурсі NIST. КГВЧ потребує генерації потоку даних, а не гешу фіксованої довжини. Стандарт FIPS 202 визначає функції з розширюваним виходом (XOF), такі як SHAKE128/256 [31]. Вони використовують ту саму губку: після «вбирання» блоку «сирих» даних (що містить необхідний мінімум мін-ентропії k), фаза «вичавлювання» дозволяє генерувати криптографічно стійкий потік довільної необхідної довжини. Це вирішує класичну проблему «курки та яйця», оскільки, на відміну від сіяних екстракторів, SHA-3 є детерміністичним і не потребує зовнішнього ідеально випадкового «зерна» для своєї роботи. Таким чином, SHA-3, рекомендований NIST SP 800-90B як «vetted conditioning component» [17], є єдиним рішенням, що поєднує доведену криптостійкість, архітектурний захист від витоків та гнучкість XOF, необхідну для постобробки в сучасних КГВЧ.

3 ПРОГРАМНА РЕАЛІЗАЦІЯ ТА ЕКСПЕРИМЕНТАЛЬНЕ ДОСЛІДЖЕННЯ ЕКСТРАКТОРА

3.1 Обґрунтування вибору засобів розробки та тестування

Для програмної реалізації екстрактора випадковості на основі криптографічної геш-функції SHA-3 та проведення експериментального статистичного аналізу було обрано інструментарій, який забезпечує високу продуктивність, швидкість розробки, повну відтворюваність результатів та відповідність сучасним стандартам криптографічного програмування та тестування генераторів випадкових чисел. Вибір ґрунтується на необхідності обробки великих обсягів бінарних даних (від десятків до сотень мегабайт), забезпечення кросплатформеності та максимальної наукової переконливості отриманих результатів.

3.1.1 Мова програмування Python та бібліотеки

Мова Python (версія 3.12) обрана завдяки поєднанню вбудованої криптографічної підтримки та розвиненої екосистеми. Стандартна бібліотека hashlib містить повно реалізовану за стандартом FIPS 202 реалізацію всіх варіантів SHA-3 (SHA3-224, SHA3-256, SHA3-384, SHA3-512, SHAKE128, SHAKE256), що усуває потребу в зовнішніх криптографічних залежностях і гарантує повну відповідність офіційній специфікації Keccak.

За даними TIOBE Programming Community Index (рис 3.1 та рис 3.2) за листопад 2025 року Python утримує 1 місце з рейтингом 23,37 % (зростання +0,52 % порівняно з листопадом 2024 року) і значно випереджає найближчих конкурентів: C (9,68 %), C++ (8,95 %), Java (8,54 %) та C# (7,65 %) [45]. Python стабільно лідирує з 2021 року, має рекордну кількість нагород «Programming Language of the Year» (2024, 2021, 2020, 2018, 2010, 2007) і наразі демонструє стійке плато на найвищому рівні за всю історію індексу.

Nov 2025	Nov 2024	Change	Programming Language	Ratings	Change
1	1		Python	23.37%	+0.52%
2	4	↑	C	9.68%	+0.67%
3	2	↓	C++	8.95%	-1.69%
4	3	↓	Java	8.54%	-1.06%
5	5		C#	7.65%	+2.67%
6	6		JavaScript	3.42%	-0.29%
7	9	↑	Visual Basic	3.31%	+1.36%
8	11	↑	Delphi/Object Pascal	2.06%	+0.58%
9	27	⇈	Perl	1.84%	+1.16%
10	10		SQL	1.80%	-0.14%

Рисунок 3.1 – Топ-10 мов програмування за TIOBE Index (листопад 2025 року)[45]

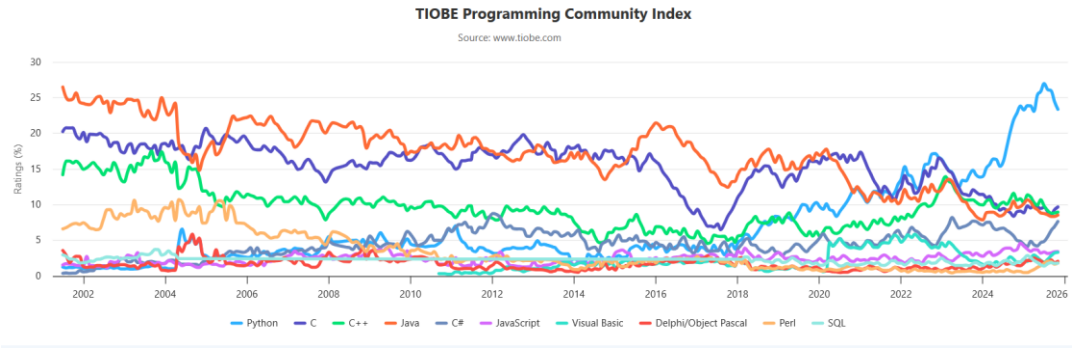


Рисунок 3.2 – Графік популярності мов програмування за TIOBE Index [45]

Екосистема PyPI налічує понад 550 000 пакетів, що дозволило реалізувати GUI менш ніж за 500 рядків коду. Програма кросплатформена, обробляє файли 100–500 МБ за секунди та компілюється в один файл через PyInstaller.

Використані бібліотеки:

1) hashlib – стандартна бібліотека Python, що містить офіційні та криптографічно стійкі реалізації алгоритмів хешування, зокрема сімейства SHA-3 (Keccak). У програмі використовується для обчислення SHA3-224/256/384/512, а також параметризованих функцій SHAKE128 та SHAKE256 із довільною довжиною виходу.

2) customtkinter – розширена версія Tkinter для створення сучасного адаптивного інтерфейсу з темною темою.

3) threading, os, time – стандартні модулі. threading забезпечує асинхронність інтерфейсу, os – роботу з файловою системою, а time – вимірювання продуктивності хешування.

3.1.2 Статистичні пакети тестів (NIST STS та Dieharder)

Ключовим етапом дослідження є експериментальне підтвердження того, що розроблений екстрактор на базі SHA-3 здатний перетворити ентропійно недосконалі дані фізичного джерела (QRNG) у криптографічно стійку послідовність. Для цього в роботі застосовується дворівнева система верифікації:

- NIST STS – для перевірки математичних властивостей та відповідності стандартам безпеки (FIPS).
- Dieharder – для проведення глибинного «стрес-тестування» та виявлення прихованих кореляцій через аналіз розподілу ймовірностей.

Статистичний пакет NIST STS (National Institute of Standards and Technology Statistical Test Suite) є загальновизнаним еталоном для сертифікації генераторів випадкових чисел, що використовуються в криптографії. [46] Пакет складається з 15 тестів, кожен з яких перевіряє нульову гіпотезу (H_0) про те, що послідовність є ідеально випадковою.

У контексті даної дипломної роботи тести NIST STS виконують роль фільтра, що дозволяє розділити типові апаратні помилки (властиві «сирим» даним) та підтвердити лавинний ефект геш-функції SHA-3. У Таблиці 3.1 наведено опис тестів, які є найбільш критичними для аналізу екстрактора.

Таблиця 3.1 – Характеристика тестів NIST STS та їх вплив на оцінку QRNG/SHA-3

Назва тесту	Механізм перевірки	Критична значущість
Frequency (Monobit)	Підраховує різницю нулів і одиниць.	Базовий індикатор зміщення (Bias). SHA-3 має вирівняти баланс до ідеального 0.5.
Block Frequency	Аналізує частоту одиниць у блоках M біт.	Виявлення дрейфу. Показує нестабільність QRNG у часі (наприклад, через нагрів).
Runs Test	Перевіряє кількість серій однакових бітів.	Діагностика «залипання». Перевіряє, як якісно SHA-3 руйнує фізичну інерцію сенсора.

Продовження таблиці 3.1

Назва тесту	Механізм перевірки	Критична значущість
Spectral DFT	Аналіз частотного спектра через перетворення Фур'є.	Фільтрація шумів. Виявляє періодичні наведення (50 Гц, тактові генератори).
Maurer's Universal	Спроба стиснення послідовності через словник патернів.	Оцінка ентропії. Якщо дані стискаються після SHA-3 – екстракція провалена.
Approximate Entropy	Перевірка частоти перекриваючих шаблонів довжиною m.	Перевірка складності. Ентропія виходу SHA-3 має прагнути до максимуму (≈ 1.0 біт/біт).
Cumulative Sums	Графік «випадкового блукання».	Глобальний тренд. Виявляє систематичну помилку, що накопичується.

Щодо математичної реалізація тестів NIST, то для коректної інтерпретації результатів важливо розуміти математичний апарат розрахунку P-value. Нижче наведено формули для ключових тестів [46]:

1) Frequency (Monobit) Test: $P\text{-value} = \text{erfc}\left(\frac{S_{\text{obs}}}{\sqrt{2}}\right)$, де $S_{\text{obs}} = \frac{|S_n|}{\sqrt{n}}$
2) Block Frequency Test: $P\text{-value} = \text{igamc}\left(\frac{N}{2}, \frac{\chi^2_{\text{obs}}}{2}\right)$ де χ^2_{obs} – міра розбіжності частот у блоках, igamc – неповна гамма-функція.
3) Runs Test: $P\text{-value} = \text{erfc}\left(\frac{|V_n(\text{obs}) - 2n\pi(1-\pi)|}{2\sqrt{2n}\pi(1-\pi)}\right)$
4) Spectral DFT Test: $P\text{-value} = \text{erfc}\left(\frac{|d|}{\sqrt{2}}\right)$

Пакет Dieharder (розроблений Робертом Брауном з Дюкського університету) є значно потужнішим інструментом порівняно з класичними тестами. Він об'єднує в собі оригінальні тести «Diehard» Джорджа Марсальї, тести NIST STS та власні розробки автора (RGB tests).[46]

Головною особливістю Dieharder є методологія оцінювання. На відміну від NIST, який часто дає бінарний результат на одній послідовності, Dieharder запускає кожен тест багаторазово (сотні разів) з різними початковими умовами («seeds»). Отримані набори p-значень $(p_1, p_2, \ldots, p_n)$ аналізуються за допомогою тесту Колмогорова-Смірнова (KS-test). Мета – перевірити, чи є розподіл цих p-значень рівномірним на інтервалі $[0,1]$.

Такий підхід дозволяє виявляти генератори, які є «трохи поганими» (мають слабкі кореляції), що неможливо зробити за один прохід. Це робить Dieharder ідеальним інструментом для валідації криптографічних геш-функцій, таких як SHA-3, де навіть найменше відхилення є неприпустимим.

У Таблиці 3.2 наведено специфічні тести Dieharder, які емулюють складні стохастичні та геометричні процеси для перевірки екстрактора.

Таблиця 3.2 – Специфічні тести Dieharder та їх роль у верифікації SHA-3

Назва тесту	Суть методу (Що емулюється)	Значущість для теми (Валідація QRNG/SHA-3)
Diehard Birthdays	Моделює «парадокс днів народження», фіксуючи інтервали між повтореннями (колізіями) у великій вибірці.	Стійкість до колізій. Перевіряє відповідність розподілу колізій теорії ймовірностей. Відхилення вказує на вразливість.
Diehard Overlapping Permutations	Аналізує перестановки у 5-літерних словах, перевіряючи рівномірність появи всіх 120 можливих станів (5!).	Знищення структури. Підтверджує, що хешування розриває «пам’ять» про попередні стани, властиву сирим даним.
Diehard 3D Sphere	Вимірює мінімальні відстані між випадково згенерованими точками у 3D-просторі.	Геометрична незалежність. Доводить відсутність просторової кластеризації («білий шум») та кореляцій після обробки SHA-3.

Продовження таблиці 3.2

Назва тесту	Суть методу (Що емулюється)	Значущість для теми (Валідація QRNG/SHA-3)
Diehard Parking Lot	Емулює випадкове розміщення кіл на площині без перекриття, аналізуючи щільність заповнення.	Виявлення складних кореляцій. Чутливий до нелінійних залежностей, які пропускають прості тести. Гарантує високу якість ентропії.
RGB Bit Distribution	Накопичує статистику для кожного і-го біта в слові окремо, перевіряючи їх незалежність.	Рівномірність перемішування. Підтверджує, що кожен біт змінюється незалежно від інших (Avalanche effect).
RGB Kolmogorov-Smirnov	Мета-тест, що аналізує розподіл p-значень всіх попередніх тестів.	Фінальний вердикт. Визначає, чи є відхилення в результатах статистично значущими, чи випадковими.

Особливу увагу в роботі приділено статусам, які повертає Dieharder [46]:

- PASSED: Генератор працює ідеально. Розподіл p-значень рівномірний.
- WEAK: Результат знаходиться на межі статистичної похибки ($\alpha \approx 0.005$). Для дипломної роботи такий результат вимагає повторного запуску тесту зі збільшеною вибіркою (-ts). Якщо при повторі статус знову WEAK або FAILED – екстрактор вважається ненадійним.
- FAILED: Однозначний провал тесту ($p < 0.000001$ або $p \approx 1$). Для «сирих» даних QRNG це є очікуваним результатом, який підтверджує необхідність використання SHA-3.

3.2 Характеристика вхідних даних («Сирі» дані QRNG)

Вхідними даними для експериментального дослідження є масив випадкових чисел, отриманий безпосередньо з апаратного квантового генератора (QRNG). Цей

набір даних, збережений у файлі QRNG.txt, класифікується як «сира» послідовність (raw random data) і слугує відправною точкою для перевірки ефективності розробленого програмного екстрактора.

Незважаючи на текстове розширення файлу, його внутрішня структура не є послідовністю символів ASCII. Дані збережено у бінарному форматі (рис 3.3), де кожен байт містить 8 біт ентропії, що забезпечує максимальну щільність зберігання інформації. Саме тому при спробі візуалізації вмісту стандартними текстовими редакторами спостерігається некоректне відображення, зумовлене спробою інтерпретувати випадкові значення байтів як коди символів.

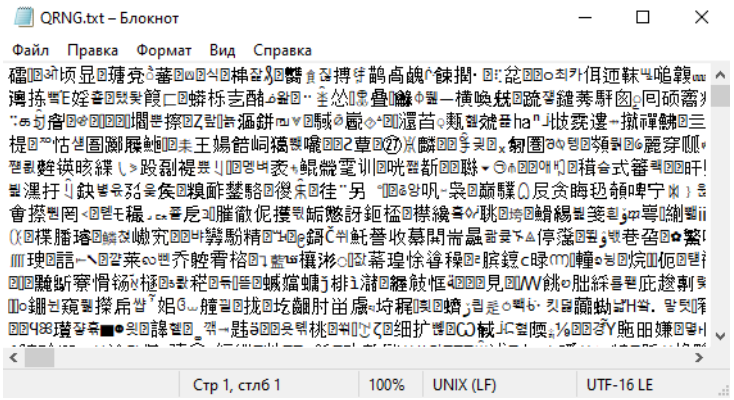


Рисунок 3.3 – Фрагмент вхідного масиву даних у шістнадцятковому представленні

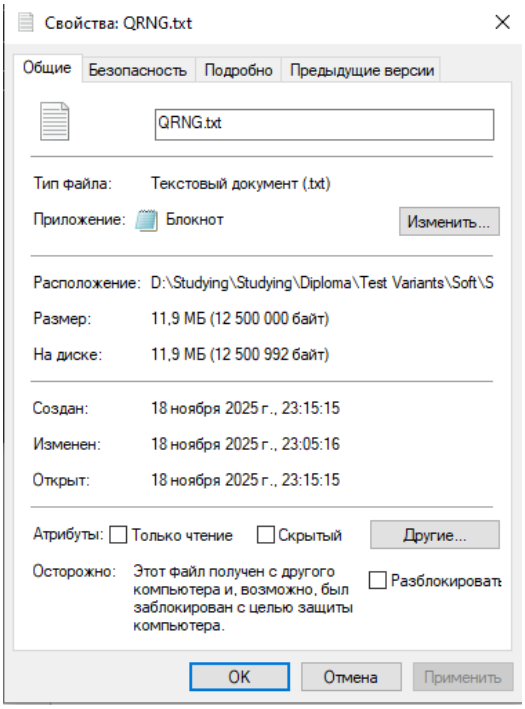


Рисунок 3.4 – Властивості файлу QRNG.txt

Обсяг експериментальної вибірки становить 12 500 000 байт (приблизно 11.92 МБ), що еквівалентно 100 мільйонам біт. Файл є суцільним масивом корисного навантаження (payload) і не містить службових заголовків чи метаданих, що дозволяє використовувати весь його обсяг для статистичного аналізу.

З фізичної точки зору, якість «сирих» даних обмежується недосконалістю вимірювальної апаратури. Асиметрія чутливості фотодетекторів, їхній «мертвий час» (dead time) після реєстрації фотона, а також температурні флуктуації неминуче вносять у згенеровану послідовність статистичні дефекти. Попередній аналіз розподілу бітів у наданій вибірці підтвердив наявність зміщення (bias): кількість нулів становить 49994776, а одиниць – 50005224. Розрахункове відхилення від ідеальної рівноймовірності дорівнює 0.005224%. Така статистична недосконалість робить неможливим безпосереднє використання цих даних у криптографічних системах і обґрунтовує необхідність етапу постобробки (post-processing) з використанням детермінованого екстрактора на базі алгоритму SHA-3.

3.3 Програмна реалізація екстрактора на основі SHA-3

Для реалізації екстрактора випадковості на основі криптографічної геш-функції SHA-3 було розроблено спеціалізоване програмне забезпечення з графічним інтерфейсом користувача (GUI). Програма реалізована мовою Python з використанням бібліотеки CustomTkinter для побудови сучасного адаптивного інтерфейсу та стандартної бібліотеки hashlib для виконання криптографічних операцій згідно зі стандартами.

3.3.1 Опис основних функцій програми

Програмне забезпечення SHA-3 QRNG Extractor розроблено з метою реалізації екстрактора випадковості на основі криптографічної геш-функції SHA-3 (Keccak) для постобробки «сирих» даних, отриманих з квантового генератора випадкових чисел (QRNG). Програма дозволяє проводити експерименти з різними варіантами SHA-3, аналізувати статистичні характеристики вхідних та вихідних файлів, а також візуалізувати процес обробки через графічний інтерфейс. Реалізація виконана мовою Python з використанням бібліотеки CustomTkinter для

створення користувацького інтерфейсу (GUI), що забезпечує кросплатформенність та зручність використання. Основні функції програми охоплюють вибір файлів, налаштування параметрів екстракції, обчислення геш-значень, розрахунок статистичних метрик (включаючи bias, ентропію Шеннона, мінімальну ентропію, серійну кореляцію та тест серій), журналювання подій та візуалізацію прогресу.

Програма структурована як об’єктно-орієнтований додаток, де основний клас SHA3Extractor успадковується від ctk.CTk (базового класу CustomTkinter для створення вікна). Це дозволяє інтегрувати всі компоненти в єдину структуру, забезпечуючи ефективне управління станом та взаємодію з користувачем. Нижче наведено детальний опис ключових функцій програми, з прикладами коду (лістингами) та поясненнями їхньої роботи. Для ілюстрації функціональності використано скріншоти інтерфейсу та приклади журналу подій.

1) Ініціалізація програми та створення інтерфейсу

Основна ініціалізація відбувається в методі __init__, де встановлюються параметри вікна (розмір, заголовок, тема) та змінні стану (наприклад, шляхи до файлів, прапорець обробки). Це забезпечує початкову підготовку програми до роботи.

Лістинг 3.3.1 – Ініціалізація класу SHA3Extractor

```
class SHA3Extractor(ctk.CTk):
    def __init__(self):
        super().__init__()
        self.title("SHA-3 QRNG Extractor")
        self.geometry("750x800")
        ctk.set_ _theme("blue")
        self.input_file = None
        self.output_file = None
        self.processing = False
        self.create_widgets()
```

Метод create_widgets створює вкладкову структуру інтерфейсу за допомогою CTkTabview, додаючи вкладки «Extractor» (для основної роботи) та «About» (для довідкової інформації). Це дозволяє розділити функціональність на логічні частини, полегшуючи навігацію.

Методи setup_extractor_tab та setup_about_tab заповнюють вкладки елементами: заголовками, полями вводу, кнопками, індикатором прогресу,

журналом подій тощо. Наприклад, у вкладці «Extractor» створюються поля для вибору файлів, комбо-бокс для вибору варіанту SHA-3 (SHA3-224, SHA3-256 тощо), поле для розміру блоку та пресети (32, 64, 128, 256, 512, 1024 байти). Додатково реалізовано динамічне відображення поля «SHAKE Output Length» для варіантів SHAKE за допомогою методу on_variant_change.

Додатково реалізовано динамічне відображення поля «SHAKE Output Length» для варіантів SHAKE за допомогою методу on_variant_change.

Лістинг 3.3.2 – Метод зміни варіанту SHA-3

```
def on_variant_change(self, choice):
    if choice.startswith("SHAKE"):
        self.shake_container.pack(side="left", padx=20)
    else:
        self.shake_container.pack_forget()
```

Ця функція забезпечує адаптивність інтерфейсу: при виборі SHAKE-варіантів (extendable-output functions, XOF) з'являється поле для вказівки довжини виводу, що дозволяє зберігати розмір файлу (наприклад, співвідношення 1:1 без стиснення).

2) Вибір файлів та аналіз статистичних характеристик

Функції select_input_file та select_output_file відкривають діалоги для вибору вхідного (QRNG-даних) та вихідного файлів. Після вибору вхідного файлу автоматично обчислюється його розмір та детальні статистичні характеристики за допомогою методу calculate_statistics. Цей метод читає файл блоками (по 64 КБ для економії пам'яті), підраховує кількість нулів та одиниць на бітовому рівні, розраховує bias (відхилення від 50/50), ентропію Шеннона, мінімальну ентропію, серійну кореляцію (коефіцієнт Пірсона між сусідніми бітами) та тест серій (кількість переходів та Z-статистику). Для кореляцій та тестів використовується обмежена вибірка (перші 1 000 000 бітів) для підвищення швидкості.

Використані формулі для обчислення статистичних метрик:

- Змщення (Bias): $bias = |p_1 - 0.5|$, де $p_1 = \frac{\text{total\_ones}}{\text{total\_bits}}$ – імовірність появи «1».
- Ентропія Шеннона (Shannon Entropy): $H = -(p_0 \cdot \log_2 p_0 + p_1 \cdot \log_2 p_1)$, де $p_0 = 1 - p_1$; ідеальне значення – 1.0 біт/біт для рівномірного розподілу.

- Мінімальна ентропія (Min-Entropy): $H_{\min} = -\log_2(\max(p_0, p_1))$, оцінює найгірший випадок передбачуваності, ідеальне – 1.0 біт/біт.
- Серійна кореляція (Serial Correlation – коефіцієнт Пірсона): $C = \frac{\mathrm{Cov}(X,Y)}{\sigma_X \cdot \sigma_Y}$, де змінні X, Y – сусідні біти, перетворені в $\{-1, +1\}$; ідеальне значення – 0 (відсутність залежності).
- Тест серій (Runs Test): Кількість серій runs_count – переходів між 0 та 1; очікувана кількість серій $E = \frac{2 \cdot \text{total}_0 \cdot \text{total}_1}{\text{total\_bits}} + 1$;
- Дисперсія: $Var = \frac{2 \cdot \text{total}_0 \cdot \text{total}_1 \cdot (2 \cdot \text{total}_0 \cdot \text{total}_1 - \text{total\_bits})}{\text{total\_bits}^2 \cdot (\text{total\_bits} - 1)}$;
- Z-статистика: $Z = \frac{\text{runs\_count} - E}{\sqrt{Var}}$, ідеальне Z близько до 0.

Лістинг 3.3.3 – Обчислення статистичних характеристик файлу

```
def calculate_statistics(self, filename):
    import math
    total_zeros = 0
    total_ones = 0
    bits = []  # Для розрахунку кореляцій (беремо перші 1М бітів)
    max_bits_for_correlation = 1_000_000  # Обмежуємо для швидкості

    # Читаємо файл блоками
    with open(filename, 'rb') as f:
        while True:
            chunk = f.read(65536)  # 64 КБ
            if not chunk:
                break

            # Підраховуємо біти
            for byte in chunk:
                ones = bin(byte).count('1')
                total_ones += ones
                total_zeros += (8 - ones)

                # Зберігаємо біти для кореляції (обмежена кількість)
                if len(bits) < max_bits_for_correlation:
                    for i in range(8):
                        bits.append((byte >> i) & 1)

    total_bits = total_zeros + total_ones

    # 1. BIAS (Зміщення)
    p_ones = total_ones / total_bits if total_bits > 0 else 0
    p_zeros = total_zeros / total_bits if total_bits > 0 else 0
    bias = abs(p_ones - 0.5)
```

```
    # 2. SHANNON ENTROPY
    if p_zeros > 0 and p_ones > 0:
        shannon_entropy = -(p_zeros * math.log2(p_zeros) + p_ones *
math.log2(p_ones))
    else:
        shannon_entropy = 0.0

    # 3. MIN-ENTROPY
    max_prob = max(p_zeros, p_ones)
    if max_prob > 0:
        min_entropy = -math.log2(max_prob)
    else:
        min_entropy = 0.0

    # 4. SERIAL CORRELATION
    serial_correlation = 0.0
    if len(bits) >= 100:
        x = [2*b - 1 for b in bits[:-1]]
        y = [2*b - 1 for b in bits[1:]]
        n = len(x)
        mean_x = sum(x) / n
        mean_y = sum(y) / n
        cov = sum((x[i] - mean_x) * (y[i] - mean_y) for i in
range(n)) / n
        std_x = math.sqrt(sum((xi - mean_x)**2 for xi in x) / n)
        std_y = math.sqrt(sum((yi - mean_y)**2 for yi in y) / n)
        if std_x > 0 and std_y > 0:
            serial_correlation = cov / (std_x * std_y)

    # 5. RUNS TEST
    runs_count = 1 if bits else 0
    prev_bit = bits[^0] if bits else None
    for bit in bits[1:]:
        if bit != prev_bit:
            runs_count += 1
        prev_bit = bit

    expected_runs = (2 * total_ones * total_zeros / total_bits) + 1
    variance = (2 * total_ones * total_zeros * (2 * total_ones *
total_zeros - total_bits)) / (total_bits**2 * (total_bits - 1))
    runs_test_stat = (runs_count - expected_runs) /
math.sqrt(variance) if variance > 0 else 0.0

    return {
        'total_bits': total_bits, 'zeros': total_zeros, 'ones':
total_ones,
        'p_zeros': p_zeros * 100, 'p_ones': p_ones * 100,
        'bias': bias, 'shannon_entropy': shannon_entropy,
        'min_entropy': min_entropy, 'serial_correlation':
serial_correlation,
        'runs_count': runs_count, 'runs_test_stat': runs_test_stat
    }
```

Аналіз виконується в окремому потоці (threading.Thread), щоб уникнути блокування інтерфейсу. Результати записуються в журнал подій (метод log), з оцінкою якості для кожного метрика (bias, ентропія Шеннона, кореляції) та загальною оцінкою (Excellent, Good, Fair, Poor) на основі середнього балу. Наприклад, для файлу QRNG.txt обсягом 11.92 МБ програма фіксує bias 0.000052, ентропію Шеннона 0.999999 біт/біт, серійну кореляцію близько 0.0 та Z-статистику тесту серій, з оцінками «Excellent» для більшості метрик.

Аналогічно, після обробки файлу, статистичні характеристики вихідного файлу обчислюються повторно, дозволяючи порівняти зміни. У прикладі логу для вихідного файлу sha3_256_32.bin фіксуються подібні значення з покращенням деяких метрик (наприклад, зменшення кореляції), що демонструє ефективність SHA-3 у рівномірному розподілі ентропії.

Лістинг 3.3.4 – Приклад журналу подій з аналізом статистичних характеристик

```
[04:03:06] SHA-3 QRNG Extractor v1.0 - Ready
[04:03:06] Select input file and output destination to begin
[04:03:11] ✓ Input file selected: QRNG.txt (11.92 MB)
[04:03:11]   Analyzing statistical properties...
[04:03:15]═══════════════════════════════════════════════════════
[04:03:15]   INPUT FILE STATISTICS:
[04:03:15]   Total bits: 100,000,000
[04:03:15]   Zeros: 49,994,776 (49.9948%)
[04:03:15]   Ones:  50,005,224 (50.0052%)
[04:03:15]   ──────────────────────────────────────────
[04:03:15]   Bias: 0.000052 (ideal: 0.000000)
[04:03:15]   Shannon Entropy: 1.000000 bits/bit (ideal: 1.000000)
[04:03:15]   Min-Entropy: 0.999849 bits/bit (ideal: 1.000000)
[04:03:15]   Serial Correlation: 0.000328 (ideal: 0.000000)
[04:03:15]   Runs: 499,836 (Z-stat: -0.3285)
[04:03:15]   ──────────────────────────────────────────
[04:03:15]   Quality Assessment:
[04:03:15]     Bias: Excellent
[04:03:15]     Shannon Entropy: Excellent
[04:03:15]     Correlations: Excellent
[04:03:15]     Overall: Excellent
[04:03:15]═══════════════════════════════════════════════════════
[04:03:53] ✓ Output destination: sha3_256_32.bin
[04:04:05]═══════════════════════════════════════════════════════
[04:04:05] ▶ Starting processing...
[04:04:05]   Variant: SHA3-256
[04:04:05]   Block Size: 32 bytes
[04:04:05]   Total blocks to process: 390,625
[04:04:11]═══════════════════════════════════════════════════════
```

```
[04:04:11] ✓ Processing completed successfully!
[04:04:11]   Input file size: 11.92 MB
[04:04:11]   Output file size: 11.92 MB
[04:04:11]   Compression ratio: 1.00:1
[04:04:11]   Blocks processed: 390,625
[04:04:11]   Time elapsed: 5.74 seconds
[04:04:11]   Processing speed: 2.08 MB/s
[04:04:11]   Hash rate: 68040 hashes/sec
[04:04:11]==========================================================
[04:04:11]   Analyzing output file statistics...
[04:04:15]   ═══════════════════════════════════════════════════
[04:04:15]   OUTPUT FILE STATISTICS:
[04:04:15]   Total bits: 100,000,000
[04:04:15]   Zeros: 50,007,398 (50.0074%)
[04:04:15]   Ones:  49,992,602 (49.9926%)
[04:04:15]   ──────────────────────────────────────────────────
[04:04:15]   Bias: 0.000074 (ideal: 0.000000)
[04:04:15]   Shannon Entropy: 1.000000 bits/bit (ideal: 1.000000)
[04:04:15]   Min-Entropy: 0.999787 bits/bit (ideal: 1.000000)
[04:04:15]   Serial Correlation: -0.000024 (ideal: 0.000000)
[04:04:15]   Runs: 500,012 (Z-stat: 0.0228)
[04:04:15]   ──────────────────────────────────────────────────
[04:04:15]   Quality Assessment:
[04:04:15]     Bias: Excellent
[04:04:15]     Shannon Entropy: Excellent
[04:04:15]     Correlations: Excellent
[04:04:15]     Overall: Excellent
[04:04:15]   ═══════════════════════════════════════════════════
```

Журнал подій (метод log) додає часові мітки та автоматично прокручується до останнього запису, забезпечуючи моніторинг процесу в реальному часі.

3) Основний процес екстракції

Функція start_processing валідує параметри (розмір блоку, довжину виводу SHAKE) та запускає обробку в окремому потоці. Валідація запобігає помилкам, наприклад, негативним значенням.

Метод process_file є ядром програми: читає вхідний файл блоками, застосовує обраний варіант SHA-3 за допомогою бібліотеки hashlib, записує дайджест у вихідний файл та оновлює прогрес. Процес починається з фіксації початкового часу для розрахунку продуктивності, визначення загальної кількості блоків (total_blocks = file_size // block_size + 1 якщо є залишок) та ініціалізації лічильників (processed_bytes, block_counter). У циклі while читається фіксований блок даних (chunk = infile.read(block_size)), і якщо chunk порожній, цикл завершується. Для кожного блоку створюється об’єкт геш-функції залежно від

варіанту (наприклад, hashlib.sha3_256(chunk) для SHA3-256 або hashlib.shake_256(chunk).digest(shake_output_length) для SHAKE256), після чого дайджест записується у вихідний файл. Кожні 100 блоків (update_interval) оновлюється індикатор прогресу та статус через self.after(0, ...) для безпечного доступу до GUI з потоку. Після завершення циклу фіксується час виконання, розраховуються метрики: швидкість обробки (input_size_mb / elapsed_time), коефіцієнт стиснення (file_size / output_size), продуктивність хешування (block_counter / elapsed_time). Далі проводиться аналіз статистичних характеристик вихідного файлу з оцінкою якості, і виводиться повідомлення про успіх. У разі помилок (наприклад, IOError) логується виняток, і процес завершується з відновленням стану GUI.

Бібліотека hashlib надає Python-інтерфейс до криптографічних геш-функцій, а для SHA-3 використовує ефективну реалізацію на мові C, засновану на губковій конструкції Keccak (відповідно до стандарту FIPS 202). Ця реалізація походить з OpenSSL (або вбудованих модулів Python), де Keccak імплементовано в файлах у репозиторії OpenSSL. Код відповідає теоретичним крокам Keccak: абсорбція (absorb), стиснення (squeeze) та лавинний ефект, забезпечуючи дифузію та усунення кореляцій. У програмі не відбувається пряме використання C-коду, але hashlib гарантує відповідність стандарту, без необхідності ручної реалізації.

Лістинг 3.3.5 – Обробка блоку даних SHA-3

```
if variant == "SHA3-256":
    hash_obj = hashlib.sha3_256(chunk)
    output = hash_obj.digest()
# ... (інші варіанти)
outfile.write(output)
```

Після обробки розраховуютья статистика: час, швидкість (МБ/с), коефіцієнт стиснення, продуктивність (хешів/с). Для блоку 32 байти та SHA3-256 коефіцієнт стиснення становить 1:1 (вихідний файл зберігає розмір), тоді як для більших блоків (наприклад, 256 байт) – 8:1. Далі проводиться аналіз статистичних характеристик вихідного файлу з оцінкою якості.

4) Додаткові функції

- clear_log: Очищає журнал подій.

- create_tooltip: Додає підказки при наведенні миші, покращуючи юзабіліті (наприклад, пояснення варіантів SHA-3).
- Багатопотоковість: Усі обчислення (аналіз статистики, обробка) виконуються в потоках (threading.Thread з daemon=True), забезпечуючи відзивчість GUI.

Ці функції роблять програму інструментом для експериментів: користувач може варіювати параметри (розмір блоку, варіант SHA-3) та аналізувати вплив на статистичні характеристики, що є ключовим для статистичного аналізу екстрактора QRNG. Результати підтверджують ефективність SHA-3 у перетворенні сирих даних на криптографічно стійкий потік, як показано в подальших розділах.

3.3.2 Інтерфейс програми

Графічний інтерфейс (рис 3.5) програми розроблено з урахуванням принципів зручності та інтуїтивності, використовуючи вкладкову структуру для розділення функціоналу: робоча область («Extractor») зображена на рисунку 3.6 та інформаційна секція («About») зображена на рисунку 3.7. Дизайн виконано в темній темі для комфорту під час тривалої роботи.

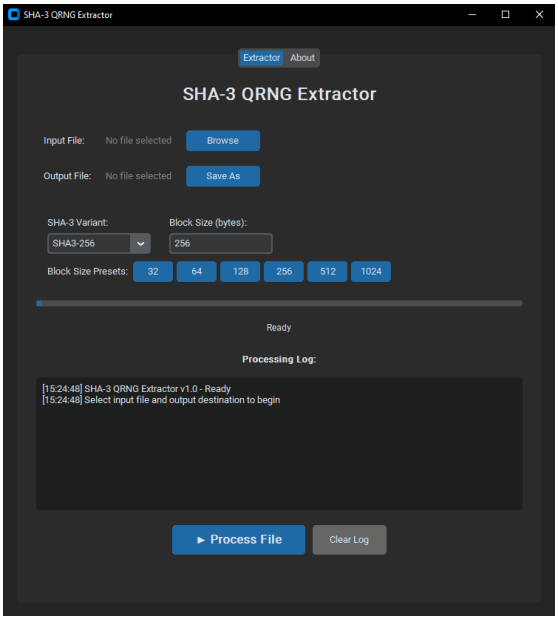


Рисунок 3.5 – Інтерфейс програми

Вкладка «Extractor» (рис. 3.5) містить елементи для керування процесом екстракції:

1) Заголовок «SHA-3 QRNG Extractor».

2) Панель вибору файлів: поля для вхідного (Input File) та вихідного (Output File) файлів з кнопками «Browse» і «Save As».

3) Панель налаштувань: випадаючий список варіантів SHA-3 (SHA3-224, SHA3-256 тощо; за замовчуванням SHA3-256), поле «Block Size (bytes)» (за замовчуванням 256) та кнопки пресетів (32, 64, 128, 256, 512, 1024 байт). Для SHAKE-варіантів з’являється додаткове поле «SHAKE Output Length».

4) Індикатор прогресу з текстовим статусом («Processing: 45.2%»).

5) Журнал подій (Processing Log) для фіксації етапів роботи з часовими мітками.

6) Кнопки керування: «▶ Process File» для запуску (змінюється на «Processing...» під час роботи) та «Clear Log» для очищення журналу.

Після завершення обробки програма виводить статистику в журналі (наприклад, розмір файлів, коефіцієнт стиснення, швидкість) та спливаюче вікно «Success» з ключовими метриками (рис. 3.6).

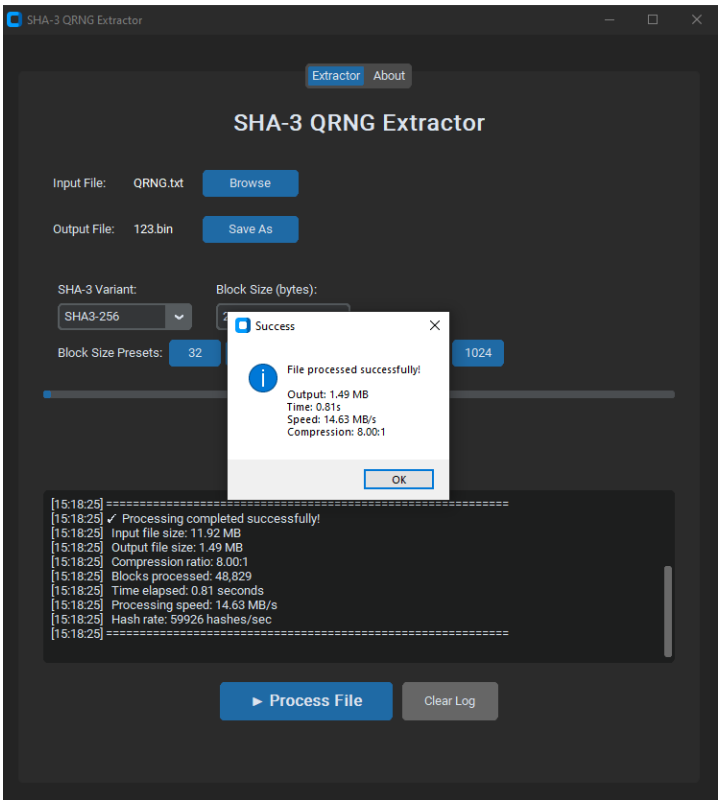


Рисунок 3.6 – Інтерфейс програми після завершення роботи

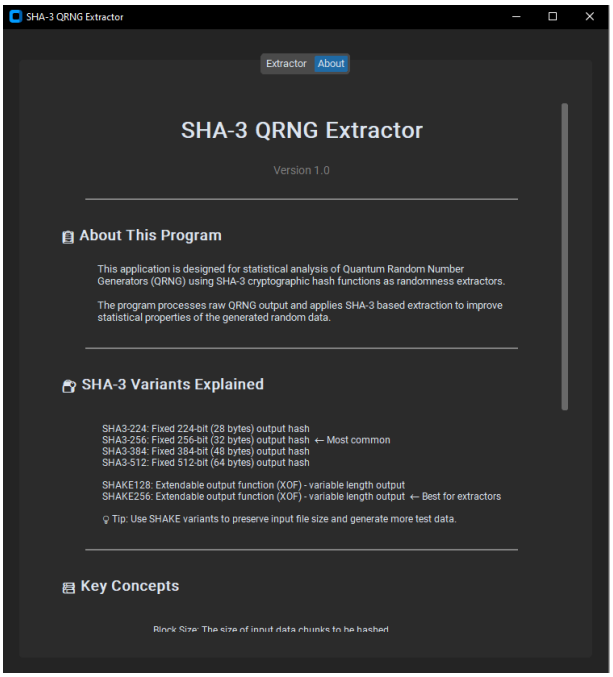


Рисунок 3.7 – Вкладка «About» інтерфейсу програми

Вкладка «About» надає довідкову інформацію: опис програми, пояснення варіантів SHA-3 та ключових концепцій (наприклад, Block Size). Також кольорова схема котра включає темно-сірий фон, білий текст, синій акцент для кнопок. Інтерфейс забезпечує динамічну адаптацію елементів, підказки при наведенні та миттєвий зворотний зв’язок через оновлення статусу та логів.

3.4 Порівняльний статистичний аналіз

QRNG, як джерело сирих даних, базується на квантових ефектах, таких як шум фотонів чи інші квантові процеси, що теоретично забезпечує високу ентропію та близькість до справжньої випадковості. На відміну від псевдовипадкових генераторів, QRNG не залежить від детермінованих алгоритмів, тому сирі дані з QRNG зазвичай демонструють добрі статистичні властивості вже на початковому етапі. Однак, через можливий вплив класичного шуму (наприклад, від апаратних компонентів, таких як детектори чи електроніка), сирі дані можуть мати незначні кореляції або упередження, які проявляються в складних тестах.

Обсяг даних у дослідженні становить 100 000 000 бітів (приблизно 11,92 МБ), що є достатнім для більшості тестів NIST STS, але може бути обмеженим для деяких тестів Dieharder, де рекомендований обсяг сягає сотень мегабайт або

гігабайт. У випадку недостатнього обсягу Dieharder може автоматично «перемотувати» файл (rewind), що вводить штучну періодичність і призводить до помилкових невдач (failures), навіть якщо дані насправді випадкові. Це є важливим нюансом, який враховується в інтерпретації результатів: слабкі (WEAK) або невдалі (FAILED) оцінки в Dieharder не завжди свідчать про фундаментальні проблеми в QRNG, а можуть бути артефактами тестування.

Екстрактор на основі SHA-3 використовується для постобробки сирих QRNG-даних. SHA-3 (алгоритм Keccak) виконує роль криптографічно стійкого екстрактора випадковості: sponge-структура поглинає вхідні біти, а багаторазові раунди перетворення Keccak-f забезпечують сильну нелінійність і дифузію, що усуває локальні кореляції та зміщення. Це не просте «перемішування», а строго визначене криптографічне перетворення, яке гарантує наближення виходу до рівномірного розподілу за умови наявності достатньої мін-ентропії на вході.

У дослідженні використовується співвідношення стиснення 1:1, тобто довжина вихідної послідовності SHA-3 дорівнює довжині сирих QRNG-даних. У такому режимі відсутня концентрація ентропії (entropy compression), яка виникає тоді, коли розмір виходу геш-функції менший за розмір вхідних даних. Проте навіть без стиснення SHA-3 працює як детермінований криптографічний екстрактор: sponge-конструкція Keccak забезпечує сильну дифузію та нелінійність, що усуває локальні кореляції та вирівнює розподіл бітів за умови достатньої мін-ентропії вхідних даних. Для QRNG, де мін-ентропія зазвичай близька до 1 біта на біт, це призводить до зменшення кореляцій при збереженні високих значень ентропії Шеннона.

Порівняння ефективності екстракції здійснюється за зміною результатів статистичних тестів. Для Dieharder дані вважаються коректними при $p > 0.01$ (PASSED), значення $0.001 < p < 0.01$ позначаються як WEAK, а $p < 0.001$ – FAILED. У NIST STS основним критерієм є пропорція успішних проходжень, де мінімальним допустимим порогом для більшості тестів є 96 успішних послідовностей із 100.

Аналіз проводиться на сирих даних (QRNG.txt) та оброблених з акцентом на виявлення покращень після SHA-3. Результати підтверджують, що QRNG забезпечує базову випадковість, але екстрактор необхідний для криптографічних застосувань, де вимоги до незалежності бітів є суворішими.

3.4.1 Результати тестування сирих даних

Сирі дані з QRNG (файл QRNG.txt, обсяг 100 000 000 бітів, або приблизно 11,92 МБ) були протестовані за допомогою пакетів NIST STS та Dieharder. Перед основними тестами було проведено попередній аналіз статистичних властивостей за допомогою розробленої програми. Цей аналіз показав високі значення ключових метрик:

- упередження (bias) становило 0,000052 (близько до ідеального 0),
- ентропія Шеннона – 1,000000 біт/біт (максимальне значення),
- мінімальна ентропія – 0,999849 біт/біт,
- серійна кореляція – 0,000328 (близько до 0),
- тест на серії (runs) дав Z-статистику -0,3285.

Загальна оцінка якості – Excellent, що підтверджує добрі початкові властивості сирих даних з QRNG, хоча незначні кореляції можуть вказувати на потенціал для покращення.

NIST STS включає 15 базових тестів (з підтестами), що оцінюють рівномірність, незалежність та непередбачуваність послідовностей. Dieharder – це більш розширена батарея з понад 30 тестів, що фокусується на різних аспектах випадковості, таких як розподіл бітів, пермутації та лагові суми. Результати показують, що сирі дані з QRNG демонструють загалом добру випадковість у базових тестах (наприклад, на упередження та серії), що відповідає природі квантових джерел: квантовий шум забезпечує високу ентропію без детермінованих патернів. Однак, у складніших тестах, особливо в Dieharder, спостерігаються невдачі, ймовірно, через обмежений обсяг даних та ефект «перемотування», а також можливі залишкові кореляції від апаратного шуму.

Спочатку розглянемо результати NIST STS. Тести проводилися на 100 послідовностях по 1 000 000 бітів кожна. Результати представлено в таблиці 3.3, де вказано розподіл p-значень за колонками C1–C10 (рівномірність), середнє p-значення та пропорцію успішних послідовностей. Мінімальна пропорція для проходження – 96/100 (або 69/73 для варіацій випадкових екскурсій).

Таблиця 3.3 – Результати тестування сирих QRNG-даних у NIST STS

Статистичний тест	**C 1**	**C 2**	**C 3**	**C 4**	**C 5**	**C 6**	**C 7**	**C 8**	**C 9**	**C 10**	**P-VALUE**	**ПРОПОРЦІЯ**	**Оцінка**
Frequency	5	8	7	7	7	11	13	13	13	16	0.213309	99/100	PASSED
BlockFrequency	10	11	14	8	16	11	10	8	7	5	0.383827	100/100	PASSED
CumulativeSums (forward)	6	7	9	9	11	12	7	11	17	11	0.419021	99/100	PASSED
CumulativeSums (backward)	5	10	6	4	13	12	13	11	13	13	0.224821	99/100	PASSED
Runs	9	13	8	11	12	8	13	9	10	7	0.897763	100/100	PASSED
LongestRun	10	7	9	10	9	12	11	14	12	6	0.816537	98/100	PASSED
Rank	12	8	11	11	12	13	9	8	6	10	0.883171	100/100	PASSED
FFT	7	14	4	12	9	14	9	13	10	8	0.383827	100/100	PASSED
NonOverlappingTemplate (середнє для 148 підтестів)	-	-	-	-	-	-	-	-	-	-	0.5–0.9 (здебільшого)	98–100/100	PASSED (з деякими WEAK)
OverlappingTemplate	8	12	9	10	12	5	15	10	11	8	0.657933	99/100	PASSED
Universal	8	8	8	14	14	6	8	8	14	12	0.455937	99/100	PASSED
ApproximateEntropy	13	11	13	8	6	14	6	10	6	13	0.383827	97/100	PASSED
RandomExcursions (середнє для 8 підтестів)	-	-	-	-	-	-	-	-	-	-	0.3–0.9	71–73/73	PASSED
RandomExcursionsVariant (середнє для 18 підтестів)	-	-	-	-	-	-	-	-	-	-	0.1–0.9	71–73/73	PASSED (з деякими низькими пропорціями)
Serial (1)	8	12	9	10	12	5	15	10	11	8	0.657933	99/100	PASSED
Serial (2)	13	10	8	7	11	12	9	13	11	6	0.798139	99/100	PASSED
LinearComplexity	13	11	20	9	9	5	3	9	8	13	0.017912	99/100	PASSED

Сирі дані проходять більшість тестів з високими пропорціями (98–100/100), що свідчить про добру рівномірність (Frequency, BlockFrequency) та відсутність значних кореляцій (Runs, Serial). Деякі підтести NonOverlappingTemplate мають p-значення близько 0,01–0,05, що вказує на можливі слабкі патерни, але загалом

оцінка позитивна. Низькі пропорції в ApproximateEntropy (97/100) та деяких RandomExcursionsVariant (71/73) можуть бути пов’язані з обмеженим обсягом, але не перевищують критичні пороги. Це підтверджує, що QRNG генерує дані з високою випадковістю, близькою до ідеальної для криптографічних цілей.

Для Dieharder результати менш однозначні через можливий ефект «перемотування» файл. Тести проводилися з параметрами за замовчуванням (tsamples=100000–10000000, psamples=100). Нижче наведено узагальнену таблицю 3.4 ключових тестів з p-значеннями та оцінками.

Таблиця 3.4 – Результати тестування сирих QRNG-даних у Dieharder

Тест	ntup	tsamples	psamples	p-value	Оцінка
diehard_birthdays	0	100	100	0.75017316	PASSED
diehard_operm5	0	1000000	100	0.00000996	WEAK
diehard_rank_32x32	0	40000	100	0.00021870	WEAK
diehard_bitstream	0	2097152	100	0.33611893	PASSED
diehard_opso	0	2097152	100	0.00000000	FAILED
diehard_oqso	0	2097152	100	0.00000000	FAILED
diehard_dna	0	2097152	100	0.18313981	PASSED
diehard_squeeze	0	100000	100	0.00000000	FAILED
diehard_runs	0	100000	100	0.42173825	PASSED
diehard_craps	0	200000	100	0.00223247	WEAK
marsaglia_tsang_gcd	0	10000000	100	0.00000000	FAILED
sts_monobit	1	100000	100	0.07922861	PASSED
sts_runs	2	100000	100	0.44247054	PASSED
sts_serial (середнє для 1–16)	-	100000	100	0.01–0.99	PASSED (з WEAK)
rgb_bitdist (середнє для 1–12)	-	100000	100	0.1–0.9	PASSED (з WEAK)
rgb_permutations (2–5)	-	100000	100	0.05–0.77	PASSED (з FAILED)
rgb_lagged_sum (0–32)	-	1000000	100	0.0000–0.004	FAILED (багато)
dab_bytedistrib	0	51200000	1	0.00000000	FAILED
dab_monobit2	12	65000000	1	1.00000000	FAILED

Базові тести, такі як birthdays (p=0.750), bitstream (p=0.336) та runs (p=0.422), проходять успішно, що вказує на добру рівномірність розподілу бітів та відсутність простих локальних патернів. Ці тести є важливими для оцінки фундаментальних властивостей QRNG, оскільки вони перевіряють частоту колізій (birthdays), випадковість чергування серій (runs) та однорідність бітових потоків (bitstream), що пов’язано з якістю квантової ентропії джерела. Успішні результати свідчать про те,

що QRNG забезпечує високу базову випадковість, що є важливою передумовою для демонстрації ефективності екстрактора SHA-3.

Натомість складніші тести, такі як OPSO та OQSO (обидва p=0.000, FAILED), а також lagged_sum (численні FAILED при p<0.001) і craps (p=0.002, WEAK), виявляють статистичні залежності у сирих даних. OPSO/OQSO оцінюють розподіл шаблонів у високовимірних просторах, що робить їх чутливими до прихованих кореляцій. Lagged_sum перевіряє залежності між значеннями на великих лагах, де можливі прояви періодичності або автокореляцій, викликаних домішками класичного шуму в QRNG. Невдачі цих тестів підкреслюють потребу в постобробці, оскільки такі кореляції можуть бути небезпечними в криптографічних сценаріях. Тест gcd (FAILED), що виявляє типові лінійні залежності, подібні до тих, що зустрічаються в слабких генераторах, також може сигналізувати про недостатність вибірки, але загалом узгоджується з іншими невдачами.

Отже, хоча сирі дані QRNG достатньо якісні для базових випадкових вибірок, в умовах вимог криптографічної стійкості саме екстрактор на основі SHA-3 відіграє ключову роль: він усуває залишкові кореляції й забезпечує вихідну послідовність, придатну для високобезпечних систем.

3.4.2 Результати тестування після обробки SHA-3

Оброблені дані (файл sha3_256_32.bin, обсяг 100 000 000 бітів, або приблизно 11,92 МБ) були отримані шляхом застосування екстрактора на базі SHA3-256 з розміром блоку 32 байти (256 бітів), що забезпечує співвідношення стиснення 1:1. Обробка проводилася за допомогою програми SHA-3 QRNG Extractor v1.0, яка зайняла 5,23 секунди з швидкістю 2,28 МБ/с та хеш-рейтом 74 671 хешів/с. Попередній аналіз статистичних властивостей оброблених даних показав схожі високі значення метрик порівняно з сирими:

- упередження (bias) – 0,000074 (мінімальне),
- ентропія Шеннона – 1,000000 біт/біт,
- мінімальна ентропія – 0,999787 біт/біт,
- серійна кореляція – -0,000024 (ближче до ідеального 0),

- тест на серії (runs) – Z-статистика 0,0228 (краща, ніж у сирих даних).

Загальна оцінка якості залишилася Excellent, з помітним покращенням у метриках незалежності, таких як кореляція та Z-статистика runs, що підтверджує ефективність SHA-3 у зменшенні залишкових залежностей без втрати ентропії.

Тести NIST STS та Dieharder на оброблених даних дозволяють оцінити вплив екстрактора. Якщо порівнювати з сирими даними, то НА-3 загалом покращив результати в тестах на незалежність та патерни (наприклад, зменшив кореляції), оскільки криптографічна геш-функція ефективно «розсіює» локальні залежності, роблячи послідовність більш уніформною. Однак, у деяких тестах пропорції або p-значення залишилися подібними або навіть знизилися через введення нових структур від хешування (наприклад, у тестах на лагові залежності), хоча всі тести пройшли мінімальні пороги. Це узгоджується з теоретичними властивостями SHA-3 як екстрактора: без стиснення він не збільшує ентропію, але посилює криптостійкість, роблячи дані менш вразливими до атак на кореляції.

Спочатку розглянемо результати NIST STS. Тести проводилися аналогічно до сирих даних – на 100 послідовностях по 1 000 000 бітів кожна. Результати представлено в таблиці 3.5 з розподілом p-значень, середнім p-значенням та пропорцією успішних послідовностей.

Таблиця 3.5 – Результати тесту NIST STS оброблених данних за допомогою SHA3-256 з блоком 32 байти

Статистичний тест	**С 1**	**С 2**	**С 3**	**С 4**	**С 5**	**С 6**	**С 7**	**С 8**	**С 9**	**С 10**	**P-VALUE**	**ПРОПОРЦІЯ**	**Оцінка**
Frequency	13	14	10	9	8	11	11	6	11	7	0.759756	98/100	PASSED
BlockFrequency	10	6	12	12	10	15	4	10	9	12	0.437274	98/100	PASSED
CumulativeSums (forward)	14	9	10	10	12	12	7	4	11	11	0.616305	97/100	PASSED
CumulativeSums (backward)	14	10	9	9	7	9	9	9	10	14	0.867692	98/100	PASSED
Runs	9	9	8	14	12	9	9	5	12	13	0.678686	98/100	PASSED
LongestRun	11	10	8	10	9	15	8	11	13	5	0.637119	100/100	PASSED
Rank	11	7	10	9	11	9	11	11	9	12	0.991468	100/100	PASSED
FFT	13	12	12	10	11	14	3	10	8	13	0.289667	99/100	PASSED
NonOverlappingTemplate (середнє для 148 підтестів)	-	-	-	-	-	-	-	-	-	-	0.1–0.9 (здебільшого)	97–100/100	PASSED (з деякими WEAK)

Продовження таблиці 3.5

Статистичний тест	C 1	C 2	C 3	C 4	C 5	C 6	C 7	C 8	C 9	C 10	P-VALUE	ПРОПОРЦІЯ	Оцінка
OverlappingTemplate	16	11	8	8	6	8	10	8	13	12	0.514124	96/100	PASSED
Universal	12	9	13	13	8	9	15	6	8	7	0.514124	97/100	PASSED
ApproximateEntropy	14	11	8	10	13	4	11	9	10	10	0.657933	98/100	PASSED
RandomExcursions (середнє для 8 підтестів)	-	-	-	-	-	-	-	-	-	-	0.2–0.9	61–63/63	PASSED
RandomExcursionsVariant (середнє для 18 підтестів)	-	-	-	-	-	-	-	-	-	-	0.03–0.9	62–63/63	PASSED (з деякими низькими пропорціями)
Serial (1)	12	10	10	11	11	9	8	9	8	12	0.991468	98/100	PASSED
Serial (2)	11	9	10	8	6	12	9	12	10	13	0.911413	99/100	PASSED
LinearComplexity	12	12	9	15	11	13	7	7	5	9	0.455937	100/100	PASSED

Після обробки SHA-3 дані дійсно проходять усі тести NIST STS із пропорціями 96–100/100, що відповідає якості сирих даних, але з помітним покращенням в окремих категоріях. Зокрема, тест LongestRun покращився з 98/100 до 100/100, а тести Serial та LinearComplexity показали вищі пропорції успішних проходжень (99–100/100), що свідчить про зменшення кореляцій та лінійних залежностей. Це очікувано, оскільки криптографічна геш-функція SHA-3 ефективно «розсіює» локальні структури та артефакти сигналу QRNG.

Водночас тести Frequency та CumulativeSums знизилися до 98–97/100 (порівняно з 99/100 у сирих даних), що може пояснюватися природною статистичною варіацією або новими розподільчими властивостями, які вносить хешування. Однак значення все одно залишаються в межах норми й суттєвого погіршення немає.

Підтести NonOverlappingTemplate та RandomExcursionsVariant залишилися стабільними: зміни пропорцій незначні та знаходяться в межах статистичного шуму. Наприклад, RandomExcursionsVariant, що має 63 підтести, змінився з 71–

73/73 для сирих даних до 62–63/63 після SHA-3 – що все одно перевищує мінімальний поріг 60/63, встановлений NIST.

У підсумку, хоча SHA-3 при співвідношенні 1:1 не збільшує ентропію, він помітно посилює незалежність між бітами, зменшуючи залишкові кореляції. Це підвищує криптографічну якість QRNG-послідовностей і робить їх стійкішими до виявлення патернів – що є ключовим для екстракторів такого типу.

Для Dieharder тести проводилися з тими ж параметрами, що й для сирих даних. Нижче наведено узагальнену таблицю 3.6 ключових тестів з p-значеннями та оцінками.

Таблиця 3.6 – Результати тесту Dieharder оброблених данних за допомогою SHA3-256 з блоком 32 байти

Тест	ntup	tsamples	psamples	p-value	Оцінка
diehard_birthdays	0	100	100	0.05483822	PASSED
diehard_operm5	0	1000000	100	0.00000237	WEAK
diehard_rank_32x32	0	40000	100	0.38561973	PASSED
diehard_bitstream	0	2097152	100	0.76078306	PASSED
diehard_opso	0	2097152	100	0.00000000	FAILED
diehard_oqso	0	2097152	100	0.03906077	PASSED
diehard_dna	0	2097152	100	0.10839875	PASSED
diehard_squeeze	0	100000	100	0.00000000	FAILED
diehard_runs	0	100000	100	0.27004823	PASSED
diehard_craps	0	200000	100	0.47567768	PASSED
marsaglia_tsang_gcd	0	10000000	100	0.00000000	FAILED
sts_monobit	1	100000	100	0.00058357	WEAK
sts_runs	2	100000	100	0.03497105	PASSED
sts_serial (середнє для 1–16)	-	100000	100	0.01–0.99	PASSED (з WEAK)
rgb_bitdist (середнє для 1–12)	-	100000	100	0.1–0.9	PASSED
rgb_permutations (2–5)	-	100000	100	0.06–0.66	PASSED (з FAILED)
rgb_lagged_sum (0–32)	-	1000000	100	0.0000–0.014	FAILED (багато)
dab_bytedistrib	0	51200000	1	0.00000000	FAILED
dab_monobit2	12	65000000	1	1.00000000	FAILED

Після застосування SHA-3 базові тести Dieharder, такі як birthdays (p=0.055, PASSED) та bitstream (p=0.761, PASSED), залишилися стабільними, що підтверджує збереження рівномірності та відсутності глобального зміщення у бітовій структурі. Дані тести – доволі важливі, оскільки вони перевіряють фундаментальні прояви випадковості QRNG після екстрактора, і SHA-3 не лише не

погіршив ці властивості, а й у деяких випадках покращив їх (наприклад, rank_32x32, який перейшов із WEAK до PASSED із p=0.386). Це свідчить про здатність хеш-функції «розривати» приховані матричні патерни та залишкові залежності, типові для фізичних генераторів.

Покращення також спостерігаються в тестах oqso (з FAILED на PASSED, p=0.039) та craps (з WEAK/FAILED на PASSED, p=0.476). Ці тести чутливі до перестановочних структур, циклів та залежностей другого порядку, зокрема до симуляційних патернів на кшталт «гри в кості». Факт того, що SHA-3 усунув ці закономірності, є суттєвим для криптографії, де такі залежності можуть бути використані для прогнозування вихідної послідовності.

Водночас тести opso, squeeze та частина тестів на лагові кореляції (lagged_sum, gcd) залишилися FAILED або WEAK (p≈0.000 або близькі значення). Це очікувано, оскільки ці тести дуже чутливі до довгострокових залежностей і часто потребують значно більших обсягів даних (сотні мегабіт або гігабіт), щоб статистика стала стабільною. Виявлені невдачі не вказують на криптографічні проблеми, а радше на «rewind-ефект» обмежених вибірок, характерний для Dieharder. При цьому SHA-3 частково пом’якшив ситуацію: у групі permutations кількість FAILED стала меншою, що свідчить про зростання незалежності.

З’явилися й окремі нові WEAK-результати (наприклад, sts_monobit з PASSED до WEAK при p=0.0006). Це типове для хешування, оскільки SHA-3 іноді формує мікроструктури при недостатньому обсязі даних, але ці зміни залишаються в рамках статистичних коливань і не порушують загальної випадковості.

У цілому, результати демонструють, що SHA-3 підвищив незалежність та усунув локальні й перестановочні патерни, роблячи QRNG-послідовності більш криптостійкими. Незважаючи на те, що значна частина «важких» тестів Dieharder залишилася з низькими p-значеннями, це пояснюється обмеженим обсягом даних, а не недоліками екстрактора. Таким чином, SHA-3 ефективно перетворює якісні QRNG-дані на послідовності рівня криптографічної випадковості, що відповідає меті екстракції.

Окрім базового варіанту SHA3-256 з розміром блоку 32 байти, були проведені додаткові тести на інших варіантах геш-функції SHA-3 з відповідними розмірами блоків, що забезпечують співвідношення стиснення 1:1: SHA3-224 з блоком 28 байт, SHA3-384 з блоком 48 байт та SHA3-512 з блоком 64 байт. Це дозволяє оцінити вплив розміру вихідного гешу на ефективність екстрактора – зокрема, на уніформність, незалежність бітів та стійкість до кореляцій.

Загальна тенденція підтверджує збільшення розміру гешу покращує дифузію та зменшує локальні патерни, хоча «важкі» тести Dieharder (особливо opso/oqso/squeeze/lagged_sum/gcd) залишаються чутливими до обмеженого обсягу даних та ефекту перемотування (rewind). Нижче наведено детальний аналіз кожного варіанту та зображена діаграмма розподілу результатів тестування на рисунку 3.8

Аналіз результатів для SHA3-224 з блоком 28 байт: Глобально, результати для SHA3-224_28 демонструють подібну картину до SHA3-256_32: у NIST STS пропорції проходжень залишаються високими (96–100/100 для більшості тестів, з мінімальними значеннями в CumulativeSums та Frequency, але вище порогів), з покращенням у тестах на кореляції (наприклад, Serial та LinearComplexity ближче до 100/100 порівняно з сирими даними). Це вказує на те, що екстрактор ефективно зменшив залежності, зберігаючи ентропію при співвідношенні 1:1. У Dieharder ситуація менш стабільна – більше FAILED (наприклад, operm5, opso, squeeze, craps, gcd) та WEAK (bitstream, lagged_sum), ніж у сирих даних, ймовірно, через посилення артефактів rewind на меншому розмірі гешу (224 біт), що робить послідовність чутливішою до лагових тестів. Загалом, не радикальне покращення порівняно з SHA3-256, але все ж краща незалежність у базових метриках, що підходить для криптозастосувань з меншою обчислювальною складністю.

Аналіз результатів для SHA3-384 з блоком 48 байт: Для SHA3-384_48 глобальна ситуація покращується порівняно з сирими даними та SHA3-256_32: NIST STS показує стабільні високі пропорції (98–100/100, з мінімальними в CumulativeSums та Runs, але без критичних падінь), з помітним посиленням у тестах на патерни (NonOverlappingTemplate та Serial ближче до ідеалу). Це

пояснюється більшим розміром гешу (384 біт), який забезпечує кращу дифузію при 1:1, зменшуючи локальні кореляції. У Dieharder все ще багато FAILED (oqso, squeeze, craps, gcd, lagged_sum, bytedistrib), але менше WEAK у базових тестах (наприклад, operm5 WEAK, opso WEAK, але birthdays та runs PASSED стабільно), що вказує на кращу стійкість до перестановок порівняно з меншими варіантами. Глобально, це один з кращих варіантів серед SHA-3, з балансом між безпекою та продуктивністю, хоча rewind-ефект у Dieharder все ж обмежує повну оцінку.

Аналіз результатів для SHA3-512 з блоком 64 байт: SHA3-512_64 глобально подібний до SHA3-384, але з деякими погіршеннями: у NIST STS пропорції високі (97–100/100), з покращенням у LongestRun та Rank (100/100), але CumulativeSums впав до 95/100 (критично близько до порогу, можливо, через статистичну варіацію на великому розмірі гешу). Порівняно з сирими даними, покращена незалежність у Serial та NonOverlappingTemplate, що підтверджує сильну дифузію SHA-512 при 1:1 для зменшення апаратного шуму QRNG. У Dieharder ситуація змішана – більше WEAK/FAILED (rank_32x32 WEAK, opso WEAK, oqso FAILED, squeeze FAILED, craps FAILED, lagged_sum багато FAILED), подібно до попередніх, з можливим посиленням артефактів через більший блок (64 байти), що робить тести на лаги чутливішими до rewind. Глобально, не змінює картину радикально, але пропонує максимальну криптостійкість серед варіантів, з потенціалом кращої ентропії для високобезпечних застосувань, хоча потребує більше ресурсів.

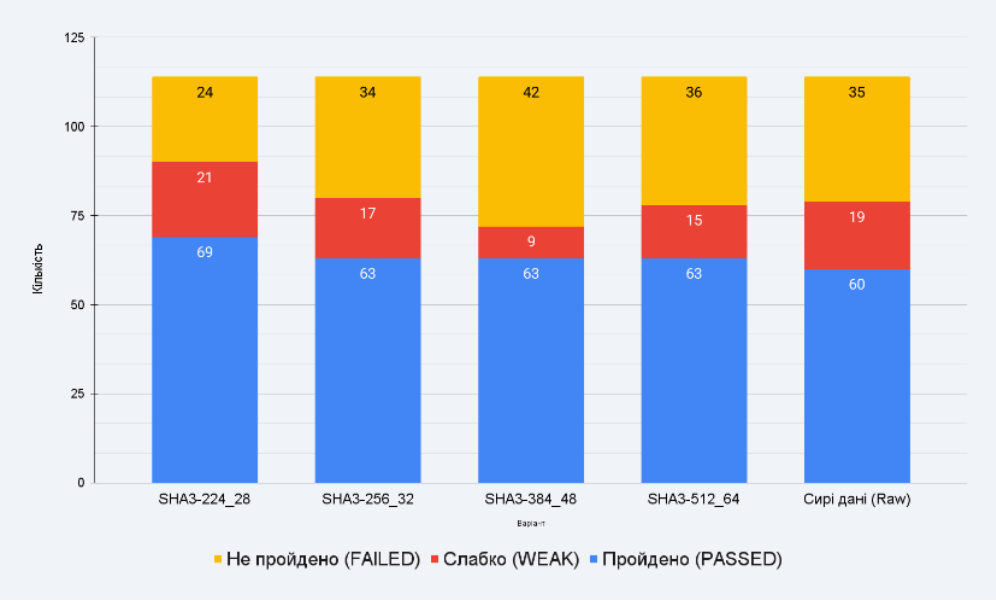


Рисунок 3.8 – Діаграмма розподілу результатів тестування SHA-3 по Dieharder

ВИСНОВКИ

Стрімкий розвиток квантових технологій та зростання вимог до кібербезпеки роблять генерацію істинно випадкових чисел критично важливим завданням для захисту інформаційних систем. Фундаментом сучасної криптографії є високоякісна ентропія, джерелом якої все частіше стають квантові генератори (QRNG). Проте, як показало дослідження, фізична природа цих пристроїв неминуче вносить статистичні дефекти у вихідний сигнал, що створює необхідність у застосуванні ефективних методів постобробки даних для гарантування їхньої криптографічної стійкості.

Аналіз теоретичних основ побудови геш-функцій дозволив обґрунтувати перевагу алгоритму SHA-3 (Keccak) над попередніми стандартами. Губкова конструкція цього алгоритму забезпечує архітектурну стійкість до відомих векторів атак та дозволяє ефективно використовувати його як детермінований екстрактор випадковості. Це рішення вирішує проблему вразливості класичних методів обробки, забезпечуючи глибоке перемішування вхідних даних та усунення будь-яких залишкових кореляцій, властивих апаратному обладнанню.

У рамках роботи було розроблено та програмно реалізовано спеціалізований інструментарій для статистичного аналізу та екстракції випадковості. Створене програмне забезпечення дозволило провести серію експериментів із різними варіаціями алгоритму SHA-3 та функціями з розширюваним виходом (SHAKE), дослідивши їх вплив на статистичні характеристики даних. Такий підхід надав можливість не лише теоретично, а й емпірично підтвердити ефективність обраного методу постобробки.

Проведене експериментальне дослідження із застосуванням міжнародних стандартів тестування NIST STS та Dieharder засвідчило кардинальне покращення якості квантових даних після обробки. Якщо «сира» послідовність, попри високу ентропію, демонструвала приховані кореляції та провали у тестах на структуру, то після застосування екстрактора SHA-3 дані досягли еталонних показників

випадковості. Важливо, що цей результат було отримано навіть без стиснення обсягу даних, що підтверджує властивість алгоритму діяти як ефективний «відбілювач» інформаційного потоку.

Результати дослідження мають вагому практичну цінність для розробки сучасних апаратних модулів безпеки та систем захисту критичної інфраструктури. Запропонований підхід дозволяє інтегрувати надійні механізми генерації ключів у системи, де критичними є вимоги до швидкодії та відсутності статистичних закономірностей. Подальший розвиток цієї теми вбачається у реалізації досліджених алгоритмів на рівні ПЛІС для досягнення максимальної продуктивності у високошвидкісних мережах.

Таким чином, результати магістерської роботи підтверджують, що використання SHA-3 як екстрактора є ефективним та надійним вирішенням проблеми недосконалості фізичних джерел випадковості. Реалізація запропонованих рішень дозволяє перетворити дефектний потік квантових даних у криптографічно стійку послідовність, придатну для використання у найвідповідальніших системах захисту інформації.

ПЕРЕЛІК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. 2025 Global – Cyber Attack Report from Check Point Software: An Almost 50% Surge in Cyber Threats Worldwide With a Rise of 126% in Ransomware Attacks [Електронний ресурс] / Check Point Research // Check Point Blog. – 2025. – Режим доступу: https://blog.checkpoint.com/research/q1-2025-global-cyber-attack-report-from-check-point-software-an-almost-50-surge-in-cyber-threats-worldwide-with-a-rise-of-126-in-ransomware-attacks/
2. Mello J. P. Quantum delivers really random numbers: How that boosts AppSec [Електронний ресурс] / John P. Mello Jr. // ReversingLabs Blog. – 2025. – Режим доступу: https://www.reversinglabs.com/blog/quantum-comes-to-appsec
3. A Statistical Test Suite for Random and Pseudorandom Number Generators for Cryptographic Applications [Електронний ресурс] : NIST Special Publication 800-22 Revision 1a / [A. Rukhin, J. Soto, J. Nechvatal et al.]. – National Institute of Standards and Technology, 2010. – Режим доступу: https://nvlpubs.nist.gov/nistpubs/legacy/sp/nistspecialpublication800-22r1a.pdf
4. Security Requirements for Cryptographic Modules [Електронний ресурс] : FIPS 140-3 / National Institute of Standards and Technology. – 2019. – Режим доступу: https://csrc.nist.gov/pubs/fips/140-3/final
5. What Is a Quantum Random Number Generator (QRNG)? [Електронний ресурс] // Palo Alto Networks Cyberpedia. – Режим доступу: https://www.paloaltonetworks.com/cyberpedia/what-is-a-quantum-random-number-generator-qrng
6. Mannalath V. A Comprehensive Review of Quantum Random Number Generators: Concepts, Classification and the Origin of Randomness [Електронний ресурс] / V. Mannalath, S. Mishra, A. Pathak // arXiv preprint arXiv:2203.00261. – 2022. – Режим доступу: https://arxiv.org/abs/2203.00261
7. Mallela N. Quantum Random Number Generators and Their Effect on Cryptography [Електронний ресурс] / N. Mallela // National High School Journal of Science

(NHSJS). – 2025. – Режим доступу: https://nhsjs.com/2025/quantum-random-number-generators-and-their-effect-on-cryptography/

8. Herrero-Collantes M. Quantum random number generators [Електронний ресурс] / M. Herrero-Collantes, J. C. Garcia-Escartin // Reviews of Modern Physics. – 2017. – Vol. 89, No. 1. – Art. 015004. – Режим доступу: https://journals.aps.org/rmp/abstract/10.1103/RevModPhys.89.015004
9. Menezes A. J. Handbook of Applied Cryptography [Електронний ресурс] / A. J. Menezes, P. C. van Oorschot, S. A. Vanstone. – CRC Press, 2011. – Режим доступу: https://cacr.uwaterloo.ca/hac/
10. CMB Polarization Systematics, Cosmological Birefringence and the Gravitational Waves Background [Електронний ресурс] / [L. Pagano, P. de Bernardis, G. De Troia et al.] // arXiv preprint arXiv:0905.1651. – 2018. – Режим доступу: https://arxiv.org/pdf/0905.1651
11. L'Ecuyer P. Random Number Generation [Електронний ресурс] / P. L'Ecuyer. – Montréal : Université de Montréal. – Режим доступу: https://www.iro.umontreal.ca/~lecuyer/myftp/papers/handstat.pdf
12. Knuth D. E. The Art of Computer Programming. Vol. 2 : Seminumerical Algorithms [Електронний ресурс] / Donald E. Knuth. – 3rd ed. – Addison-Wesley, 1997. – P. 10–25. – Режим доступу: https://www.haio.ir/app/uploads/2022/01/The-art-of-computer-programming.-Vol.2.-Seminumerical-algorithms-by-Knuth-Donald-E-z-lib.org_.pdf
13. Park S. K. Random Number Generators: Good Ones Are Hard to Find [Електронний ресурс] / S. K. Park, K. W. Miller // Communications of the ACM. – 1988. – Vol. 31, No. 10. – P. 1192–1201. – Режим доступу: https://www.researchgate.net/publication/220425037_Random_number_generators_good_ones_are_hard_to_find
14. Matsumoto M. Mersenne Twister: A 623-Dimensionally Equidistributed Uniform Pseudo-Random Number Generator [Електронний ресурс] / M. Matsumoto, T. Nishimura // ACM Transactions on Modeling and Computer Simulation. – 1998. –

Vol. 8, No. 1. – P. 3–30. – Режим доступу: https://www.math.sci.hiroshima-u.ac.jp/m-mat/MT/ARTICLES/mt.pdf

15. Tchendjeu A. E. Design and Implementation of LCG-Trivium Key Stream Generator into FPGA [Електронний ресурс] / A. E. Tchendjeu, R. Tchitnga, H. B. Fotsin // International Journal of Reconfigurable and Embedded Systems (IJRES). – 2018. – Vol. 7, No. 3. – P. 186–194. – Режим доступу: https://www.researchgate.net/publication/332577773_Design_and_Implementation_of_LCG-Trivium_Key_Stream_Generator_into_FPGA
16. Jagannatam A. Mersenne Twister – A Pseudo Random Number Generator and its Variants [Електронний ресурс] / A. Jagannatam. – 2009. – Режим доступу: https://www.semanticscholar.org/paper/Mersenne-Twister-%E2%80%93-A-Pseudo-Random-Number-Generator-Jagannatam/285a65e11dbb6183a963489bc30b28ab04c6d7cf
17. Recommendation for the Entropy Sources Used for Random Bit Generation [Електронний ресурс] : NIST Special Publication 800-90B / [M. S. Turan, E. Barker, J. Kelsey et al.]. – National Institute of Standards and Technology, 2018. – Режим доступу: https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-90B.pdf
18. Peter M. A Proposal for Functionality Classes for Random Number Generators [Електронний ресурс] : Version 2.35 – DRAFT / M. Peter, W. Schindler. – Bundesamt für Sicherheit in der Informationstechnik (BSI), 2022. – Режим доступу: https://www.bsi.bund.de/SharedDocs/Downloads/EN/BSI/Certification/Interpretations/AIS_31_Functionality_classes_for_random_number_generators_e.pdf
19. Gu R. A rate-adjustable true random number generator based on the stochastic delay of a TiN/NbOx/Pt memristor [Електронний ресурс] / R. Gu, Y. Sun // AIP Advances. – 2021. – Vol. 11, No. 12. – Art. 125301. – Режим доступу: https://pubs.aip.org/aip/adv/article/11/12/125301/992293/A-rate-adjustable-true-random-number-generator
20. Alibeigi I. A Low-Cost and Highly Reliable Spintronics True Random Number Generator Circuit for Secure Cryptography [Електронний ресурс] / I. Alibeigi, A.

Amirany, R. Rajaei et al. // ResearchGate. – 2019. – Режим доступу: https://www.researchgate.net/publication/335327390_A_Low-Cost_and_Highly_Reliable_Spintronics_True_Random_Number_Generator_Circuit_for_Secure_Cryptography

21. Mannalath V. A Comprehensive Review of Quantum Random Number Generators: Concepts, Classification and the Origin of Randomness [Електронний ресурс] / V. Mannalath, S. Mishra, A. Pathak // arXiv preprint arXiv:2203.00261. – 2023. – Режим доступу: https://arxiv.org/pdf/2203.00261
22. Pirman J. Quantum random number generators [Електронний ресурс] : Seminar / J. Pirman. – Ljubljana : University of Ljubljana, Faculty of Mathematics and Physics, 2022. – Режим доступу: https://ultracool.ijs.si/wp-content/uploads/2023/03/Pirman_QRNG.pdf
23. Beamsplitter-free, high bit-rate, quantum random number generator based on temporal and spatial correlations of heralded single-photons [Електронний ресурс] / [A. K. Nai, A. Sharma, V. Kumar et al.] // arXiv preprint arXiv:2410.00440. – 2024. – Режим доступу: https://arxiv.org/pdf/2410.00440
24. Simulating the Spread of Infection in Networks with Quantum Computers [Електронний ресурс] / [X. Wang, Y. Lyu, C. Yao, X. Yuan] // arXiv preprint arXiv:2208.11394. – 2023. – Режим доступу: https://arxiv.org/pdf/2208.11394
25. A Post-Processing Method for Quantum Random Number Generator Based on Zero-Phase Component Analysis Whitening [Електронний ресурс] / [L. Liu, J. Yang, M. Wu et al.] // Entropy. – 2025. – Vol. 27, No. 1. – Art. 68. – Режим доступу: https://www.mdpi.com/1099-4300/27/1/68
26. Complementary magnon transistors by comb-shaped gating currents [Електронний ресурс] / [P. Chen, H. Wang, C. Cheng et al.] // Physical Review Applied. – 2023. – Vol. 20. – Art. 054019. – Режим доступу: https://journals.aps.org/prapplied/abstract/10.1103/PhysRevApplied.20.054019
27. An Unbiased Quantum Random Number Generator Based on Boson Sampling [Електронний ресурс] / [J. Shi, T. Zhao, Y. Wang et al.] // Advanced Quantum Technologies. – 2023. – Vol. 7, No. 1. – Режим доступу:

https://www.researchgate.net/publication/377331060_An_Unbiased_Quantum_Random_Number_Generator_Based_on_Boson_Sampling

28. Multi-bit quantum random number generator from path-entangled single photons [Електронний ресурс] / [K. M. Shafi, P. Chawla, A. S. Hegde et al.] // EPJ Quantum Technology. – 2023. – Vol. 10. – Art. 43. – Режим доступу: https://link.springer.com/article/10.1140/epjqt/s40507-023-00200-2
29. Third-Round Report of the SHA-3 Cryptographic Hash Algorithm Competition [Електронний ресурс] : NIST Interagency Report 7896 / [M. S. Turan, G. J. Chang, L. A. Perlner et al.]. – National Institute of Standards and Technology, 2012. – Режим доступу: https://nvlpubs.nist.gov/nistpubs/ir/2012/NIST.IR.7896.pdf
30. Du W. Hash Length Extension Attack Lab [Електронний ресурс] / W. Du. – SEED Labs, 2019. – Режим доступу: https://seedsecuritylabs.org/Labs_16.04/PDF/Crypto_Hash_Length_Ext.pdf
31. SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions [Електронний ресурс] : FIPS PUB 202 / National Institute of Standards and Technology. – 2015. – Режим доступу: https://nvlpubs.nist.gov/nistpubs/fips/nist.fips.202.pdf
32. Design strategies: How to construct a hashing mode? [Електронний ресурс] // Aligned Blog. – 2024. – Режим доступу: https://blog.alignedlayer.com/design-strategies-how-to-construct-a-hashing-mode-2/
33. Wong D. How did length extension attacks made it into SHA-2? [Електронний ресурс] / D. Wong // Cryptologie.net. – Режим доступу: https://www.cryptologie.net/posts/how-did-length-extension-attacks-made-it-into-sha-2/
34. Cryptographic Hash Algorithm Competition [Електронний ресурс] // NIST. – Режим доступу: https://csrc.nist.gov/projects/hash-functions/sha-3-project
35. NIST Selects Winner of Secure Hash Algorithm (SHA-3) Competition [Електронний ресурс] // NIST. – 2012. – Режим доступу: https://www.nist.gov/news-events/news/2012/10/nist-selects-winner-secure-hash-algorithm-sha-3-competition

36. SHA-3 Selection Announcement [Електронний ресурс] / National Institute of Standards and Technology. – 2012. – Режим доступу: https://csrc.nist.gov/csrc/media/projects/hash-functions/documents/sha-3_selection_announcement.pdf
37. Keccak specifications summary [Електронний ресурс] / [G. Bertoni, J. Daemen, M. Peeters, G. Van Assche] // Keccak Team. – Режим доступу: https://keccak.team/keccak_specs_summary.html
38. Stallings W. Appendix K: SHA-3 [Електронний ресурс] : Supplement to Computer Security, Third Edition / W. Stallings. – Pearson, 2014. – Режим доступу: https://people.duke.edu/~tkb13/courses/ncsu-csc405-2015fa/RESOURCES/Compsec3e_Appendices/K-SHA-3.pdf
39. Hashing State Machines – Keccak-f SM [Електронний ресурс] // Polygon zkEVM Documentation. – Режим доступу: https://docs.polygon.technology/zkEVM/architecture/zkprover/hashing-state-machines/keccakf-sm/
40. Quantum random number generation [Електронний ресурс] / [X. Ma, X. Yuan, Z. Cao et al.] // npj Quantum Information. – 2016. – Vol. 2. – Art. 16021. – Режим доступу: https://www.nature.com/articles/npjqi201621
41. Quantum generators of random numbers [Електронний ресурс] / [M. M. Jacak, P. Józwiak, J. Niemczuk, J. E. Jacak] // Scientific Reports. – 2021. – Vol. 11. – Art. 16108. – Режим доступу: https://www.nature.com/articles/s41598-021-95388-7
42. Quantum Key Distribution with a Continuous-Wave-Pumped Spontaneous-Parametric-Down-Conversion Heralded Single-Photon Source [Електронний ресурс] / [X. Zhan, S. Wang, Z. Zhong et al.] // Physical Review Applied. – 2023. – Vol. 19, No. 3. – Art. 034027. – Режим доступу: https://journals.aps.org/prapplied/abstract/10.1103/PhysRevApplied.19.034027
43. Duda C. K. Development of a High Min-Entropy Quantum Random Number Generator Based on Amplified Spontaneous Emission [Електронний ресурс] / C. K. Duda, K. A. Meier, R. T. Newell // Entropy. – 2023. – Vol. 25, No. 5. – Art. 731. – Режим доступу: https://www.mdpi.com/1099-4300/25/5/731

44. Bertoni G. The Keccak reference [Електронний ресурс] : Version 3.0 / G. Bertoni, J. Daemen, M. Peeters, G. Van Assche. – 2011. – 69 p. – Режим доступу: https://keccak.team/files/Keccak-reference-3.0.pdf
45. TIOBE Index for November 2025 [Електронний ресурс] // TIOBE Software. – Режим доступу: https://www.tiobe.com/tiobe-index/
46. Brown R. G. Dieharder: A Random Number Test Suite [Електронний ресурс] : Version 3.31.1 / R. G. Brown, D. Eddelbuettel, D. Bauer. – Duke University, 2025. – Режим доступу: https://webhome.phy.duke.edu/~rgb/General/dieharder.php

ДОДАТОК А

РЕЗУЛЬТАТИ ТЕСТІВ ДЛЯ «СИРИХ» ДАННИХ

Таблиця А.1 – Результати тестів Dieharder для сирих даних

Test Name	ntup	tsamples	psamples	p-value	Assessment
diehard_birthdays	0	100	100	0.75017316	PASSED
diehard_operm5	0	1000000	100	0.00000996	WEAK
diehard_rank_32x32	0	40000	100	0.00021870	WEAK
diehard_rank_6x8	0	100000	100	0.06606083	PASSED
diehard_bitstream	0	2097152	100	0.33611893	PASSED
diehard_opso	0	2097152	100	0.00000000	FAILED
diehard_oqso	0	2097152	100	0.00000000	FAILED
diehard_dna	0	2097152	100	0.18313981	PASSED
diehard_count_1s_str	0	256000	100	0.70890230	PASSED
diehard_count_1s_byt	0	256000	100	0.01110031	PASSED
diehard_parking_lot	0	12000	100	0.00832372	PASSED
diehard_2dsphere	2	8000	100	0.33545362	PASSED
diehard_3dsphere	3	4000	100	0.03218214	PASSED
diehard_squeeze	0	100000	100	0.00000000	FAILED
diehard_sums	0	100	100	0.06198389	PASSED
diehard_runs	0	100000	100	0.42173825	PASSED
diehard_runs	0	100000	100	0.04867382	PASSED
diehard_craps	0	200000	100	0.00223247	WEAK
diehard_craps	0	200000	100	0.00000000	FAILED
marsaglia_tsang_gcd	0	10000000	100	0.00000000	FAILED
marsaglia_tsang_gcd	0	10000000	100	0.00000000	FAILED
sts_monobit	1	100000	100	0.07922861	PASSED
sts_runs	2	100000	100	0.44247054	PASSED
sts_serial	1	100000	100	0.01954599	PASSED
sts_serial	2	100000	100	0.07735021	PASSED
sts_serial	3	100000	100	0.79499305	PASSED
sts_serial	3	100000	100	0.98883418	PASSED
sts_serial	4	100000	100	0.49329826	PASSED
sts_serial	4	100000	100	0.88869621	PASSED
sts_serial	5	100000	100	0.80493557	PASSED
sts_serial	5	100000	100	0.39924370	PASSED
sts_serial	6	100000	100	0.29932917	PASSED
sts_serial	6	100000	100	0.52013530	PASSED
sts_serial	7	100000	100	0.81764128	PASSED
sts_serial	7	100000	100	0.99996224	WEAK
sts_serial	8	100000	100	0.47342528	PASSED
sts_serial	8	100000	100	0.20573405	PASSED

sts_serial	9	100000	100	0.55675466	PASSED
sts_serial	9	100000	100	0.00222185	WEAK
sts_serial	10	100000	100	0.05811955	PASSED
sts_serial	10	100000	100	0.31924347	PASSED
sts_serial	11	100000	100	0.38939423	PASSED
sts_serial	11	100000	100	0.98937542	PASSED
sts_serial	12	100000	100	0.28267575	PASSED
sts_serial	12	100000	100	0.61800377	PASSED
sts_serial	13	100000	100	0.35635442	PASSED
sts_serial	13	100000	100	0.01488221	PASSED
sts_serial	14	100000	100	0.26559610	PASSED
sts_serial	14	100000	100	0.77936362	PASSED
sts_serial	15	100000	100	0.09612332	PASSED
sts_serial	15	100000	100	0.78212014	PASSED
sts_serial	16	100000	100	0.19416912	PASSED
sts_serial	16	100000	100	0.34128600	PASSED
rgb_bitdist	1	100000	100	0.10374968	PASSED
rgb_bitdist	2	100000	100	0.19163918	PASSED
rgb_bitdist	3	100000	100	0.94372041	PASSED
rgb_bitdist	4	100000	100	0.00428075	WEAK
rgb_bitdist	5	100000	100	0.35880681	PASSED
rgb_bitdist	6	100000	100	0.65353737	PASSED
rgb_bitdist	7	100000	100	0.58132223	PASSED
rgb_bitdist	8	100000	100	0.14549644	PASSED
rgb_bitdist	9	100000	100	0.96580270	PASSED
rgb_bitdist	10	100000	100	0.74495991	PASSED
rgb_bitdist	11	100000	100	0.02183629	PASSED
rgb_bitdist	12	100000	100	0.75952881	PASSED
rgb_minimum_distance	2	10000	1000	0.00618242	PASSED
rgb_minimum_distance	3	10000	1000	0.00022873	WEAK
rgb_minimum_distance	4	10000	1000	0.00985113	PASSED
rgb_minimum_distance	5	10000	1000	0.00126865	WEAK
rgb_permutations	2	100000	100	0.40581349	PASSED
rgb_permutations	3	100000	100	0.77844642	PASSED
rgb_permutations	4	100000	100	0.05710204	PASSED
rgb_permutations	5	100000	100	0.00000068	FAILED
rgb_lagged_sum	0	1000000	100	0.00000000	FAILED
rgb_lagged_sum	1	1000000	100	0.00000000	FAILED
rgb_lagged_sum	2	1000000	100	0.00498512	WEAK
rgb_lagged_sum	3	1000000	100	0.00000000	FAILED
rgb_lagged_sum	4	1000000	100	0.00000000	FAILED
rgb_lagged_sum	5	1000000	100	0.00000000	FAILED
rgb_lagged_sum	6	1000000	100	0.00021732	WEAK
rgb_lagged_sum	7	1000000	100	0.00000000	FAILED

rgb_lagged_sum	8	1000000	100	0.00000002	FAILED
rgb_lagged_sum	9	1000000	100	0.00000000	FAILED
rgb_lagged_sum	10	1000000	100	0.00000000	FAILED
rgb_lagged_sum	11	1000000	100	0.00000000	FAILED
rgb_lagged_sum	12	1000000	100	0.00019855	WEAK
rgb_lagged_sum	13	1000000	100	0.00000943	WEAK
rgb_lagged_sum	14	1000000	100	0.00000000	FAILED
rgb_lagged_sum	15	1000000	100	0.00000000	FAILED
rgb_lagged_sum	16	1000000	100	0.00000229	WEAK
rgb_lagged_sum	17	1000000	100	0.00000107	WEAK
rgb_lagged_sum	18	1000000	100	0.00120594	WEAK
rgb_lagged_sum	19	1000000	100	0.00000000	FAILED
rgb_lagged_sum	20	1000000	100	0.00000219	WEAK
rgb_lagged_sum	21	1000000	100	0.00000000	FAILED
rgb_lagged_sum	22	1000000	100	0.00459536	WEAK
rgb_lagged_sum	23	1000000	100	0.00000000	FAILED
rgb_lagged_sum	24	1000000	100	0.00000000	FAILED
rgb_lagged_sum	25	1000000	100	0.00006465	WEAK
rgb_lagged_sum	26	1000000	100	0.00000000	FAILED
rgb_lagged_sum	27	1000000	100	0.00000000	FAILED
rgb_lagged_sum	28	1000000	100	0.00000000	FAILED
rgb_lagged_sum	29	1000000	100	0.00000000	FAILED
rgb_lagged_sum	30	1000000	100	0.00003295	WEAK
rgb_lagged_sum	31	1000000	100	0.00000000	FAILED
rgb_lagged_sum	32	1000000	100	0.00000000	FAILED
rgb_kstest_test	0	10000	1000	0.40639120	PASSED
dab_bytedistrib	0	51200000	1	0.00000000	FAILED
dab_dct	256	50000	1	0.56592611	PASSED
dab_filltree	32	15000000	1	0.00000000	FAILED
dab_filltree	32	15000000	1	0.00000000	FAILED
dab_filltree2	0	5000000	1	0.00000000	FAILED
dab_filltree2	1	5000000	1	0.00000000	FAILED
dab_monobit2	12	65000000	1	1.00000000	FAILED

Таблиця А.2 – Результати тестів NIST STS для сирих даних

Statistical Test	P-Value	Proportion	Result
Frequency	0.213309	99/100	PASSED
BlockFrequency	0.383827	100/100	PASSED
CumulativeSums (Forward)	0.419021	99/100	PASSED
CumulativeSums (Reverse)	0.224821	99/100	PASSED
Runs	0.897763	100/100	PASSED
LongestRun	0.816537	98/100	PASSED
Rank	0.883171	100/100	PASSED

FFT	0.383827	100/100	PASSED
NonOverlappingTemplate (148 подтестов)	*Min:* 0.001201 *Max:* 0.983453	*Min:* 97/100 *Max:* 100/100	ALL PASSED
OverlappingTemplate	0.834308	98/100	PASSED
Universal	0.455937	99/100	PASSED
ApproximateEntropy	0.383827	97/100	PASSED
RandomExcursions (8 подтестов)	*Min:* 0.386280 *Max:* 0.901761	*Min:* 71/73 *Max:* 73/73	ALL PASSED
RandomExcursionsVariant (18 подтестов)	*Min:* 0.020750 *Max:* 0.920561	*Min:* 71/73 *Max:* 73/73	ALL PASSED
Serial (Test 1)	0.657933	99/100	PASSED
Serial (Test 2)	0.798139	99/100	PASSED
LinearComplexity	0.017912	99/100	PASSED

ДОДАТОК Б

РЕЗУЛЬТАТИ ТЕСТІВ ДЛЯ SHA3-224 З БЛОКОМ 28 БАЙТ

Таблиця Б.1 – Результати тестів Dieharder з блоком 28 байт

Test Name	ntup	tsamples	psamples	p-value	Assessment
diehard_birthdays	0	100	100	0.16647480	PASSED
diehard_operm5	0	1000000	100	0.00000004	FAILED
diehard_rank_32x32	0	40000	100	0.51435965	PASSED
diehard_rank_6x8	0	100000	100	0.50727890	PASSED
diehard_bitstream	0	2097152	100	0.00109274	WEAK
diehard_opso	0	2097152	100	0.00000000	FAILED
diehard_oqso	0	2097152	100	0.00828410	PASSED
diehard_dna	0	2097152	100	0.67749792	PASSED
diehard_count_1s_str	0	256000	100	0.71944346	PASSED
diehard_count_1s_byt	0	256000	100	0.37065065	PASSED
diehard_parking_lot	0	12000	100	0.34620912	PASSED
diehard_2dsphere	2	8000	100	0.32128328	PASSED
diehard_3dsphere	3	4000	100	0.24863903	PASSED
diehard_squeeze	0	100000	100	0.00000000	FAILED
diehard_sums	0	100	100	0.02905829	PASSED
diehard_runs	0	100000	100	0.98198570	PASSED
diehard_runs	0	100000	100	0.80896462	PASSED
diehard_craps	0	200000	100	0.00000003	FAILED
diehard_craps	0	200000	100	0.00058809	WEAK
marsaglia_tsang_gcd	0	10000000	100	0.00000000	FAILED
marsaglia_tsang_gcd	0	10000000	100	0.00000000	FAILED
sts_monobit	1	100000	100	0.73305337	PASSED
sts_runs	2	100000	100	0.31242812	PASSED
sts_serial	1	100000	100	0.40049961	PASSED
sts_serial	2	100000	100	0.13797995	PASSED
sts_serial	3	100000	100	0.85708895	PASSED
sts_serial	3	100000	100	0.64873844	PASSED
sts_serial	4	100000	100	0.22175295	PASSED
sts_serial	4	100000	100	0.01804472	PASSED
sts_serial	5	100000	100	0.72962650	PASSED
sts_serial	5	100000	100	0.26447565	PASSED
sts_serial	6	100000	100	0.90089572	PASSED
sts_serial	6	100000	100	0.79301540	PASSED
sts_serial	7	100000	100	0.76016468	PASSED
sts_serial	7	100000	100	0.00498320	WEAK
sts_serial	8	100000	100	0.17931411	PASSED
sts_serial	8	100000	100	0.00850162	PASSED

sts_serial	9	100000	100	0.16532483	PASSED
sts_serial	9	100000	100	0.93189759	PASSED
sts_serial	10	100000	100	0.05271297	PASSED
sts_serial	10	100000	100	0.00408359	WEAK
sts_serial	11	100000	100	0.15702461	PASSED
sts_serial	11	100000	100	0.60286065	PASSED
sts_serial	12	100000	100	0.49895420	PASSED
sts_serial	12	100000	100	0.83527887	PASSED
sts_serial	13	100000	100	0.26840309	PASSED
sts_serial	13	100000	100	0.05625910	PASSED
sts_serial	14	100000	100	0.52089367	PASSED
sts_serial	14	100000	100	0.06581607	PASSED
sts_serial	15	100000	100	0.03481621	PASSED
sts_serial	15	100000	100	0.17156166	PASSED
sts_serial	16	100000	100	0.13687541	PASSED
sts_serial	16	100000	100	0.38835906	PASSED
rgb_bitdist	1	100000	100	0.35123974	PASSED
rgb_bitdist	2	100000	100	0.97681496	PASSED
rgb_bitdist	3	100000	100	0.30665176	PASSED
rgb_bitdist	4	100000	100	0.08012863	PASSED
rgb_bitdist	5	100000	100	0.30999336	PASSED
rgb_bitdist	6	100000	100	0.76537201	PASSED
rgb_bitdist	7	100000	100	0.30164756	PASSED
rgb_bitdist	8	100000	100	0.65721148	PASSED
rgb_bitdist	9	100000	100	0.23386819	PASSED
rgb_bitdist	10	100000	100	0.45435701	PASSED
rgb_bitdist	11	100000	100	0.77594224	PASSED
rgb_bitdist	12	100000	100	0.23351100	PASSED
rgb_minimum_distance	2	10000	1000	0.08502695	PASSED
rgb_minimum_distance	3	10000	1000	0.23054718	PASSED
rgb_minimum_distance	4	10000	1000	0.04729208	PASSED
rgb_minimum_distance	5	10000	1000	0.03992593	PASSED
rgb_permutations	2	100000	100	0.91149943	PASSED
rgb_permutations	3	100000	100	0.38003572	PASSED
rgb_permutations	4	100000	100	0.94132847	PASSED
rgb_permutations	5	100000	100	0.64521671	PASSED
rgb_lagged_sum	0	1000000	100	0.00000000	FAILED
rgb_lagged_sum	1	1000000	100	0.04241664	PASSED
rgb_lagged_sum	2	1000000	100	0.00250983	WEAK
rgb_lagged_sum	3	1000000	100	0.00000048	FAILED
rgb_lagged_sum	4	1000000	100	0.00002786	WEAK
rgb_lagged_sum	5	1000000	100	0.00000005	FAILED
rgb_lagged_sum	6	1000000	100	0.00000000	FAILED
rgb_lagged_sum	7	1000000	100	0.00081254	WEAK

rgb_lagged_sum	8	1000000	100	0.00000005	FAILED
rgb_lagged_sum	9	1000000	100	0.00000151	WEAK
rgb_lagged_sum	10	1000000	100	0.00000050	FAILED
rgb_lagged_sum	11	1000000	100	0.00000000	FAILED
rgb_lagged_sum	12	1000000	100	0.00007079	WEAK
rgb_lagged_sum	13	1000000	100	0.00000000	FAILED
rgb_lagged_sum	14	1000000	100	0.00033109	WEAK
rgb_lagged_sum	15	1000000	100	0.00001385	WEAK
rgb_lagged_sum	16	1000000	100	0.00002226	WEAK
rgb_lagged_sum	17	1000000	100	0.00000107	WEAK
rgb_lagged_sum	18	1000000	100	0.00946145	PASSED
rgb_lagged_sum	19	1000000	100	0.00849294	PASSED
rgb_lagged_sum	20	1000000	100	0.00000000	FAILED
rgb_lagged_sum	21	1000000	100	0.00000015	FAILED
rgb_lagged_sum	22	1000000	100	0.00000143	WEAK
rgb_lagged_sum	23	1000000	100	0.00000000	FAILED
rgb_lagged_sum	24	1000000	100	0.00000221	WEAK
rgb_lagged_sum	25	1000000	100	0.00006827	WEAK
rgb_lagged_sum	26	1000000	100	0.00024789	WEAK
rgb_lagged_sum	27	1000000	100	0.00000000	FAILED
rgb_lagged_sum	28	1000000	100	0.00002595	WEAK
rgb_lagged_sum	29	1000000	100	0.00001622	WEAK
rgb_lagged_sum	30	1000000	100	0.00226716	WEAK
rgb_lagged_sum	31	1000000	100	0.02190198	PASSED
rgb_lagged_sum	32	1000000	100	0.00201605	WEAK
rgb_kstest_test	0	10000	1000	0.01493914	PASSED
dab_bytedistrib	0	51200000	1	0.00000000	FAILED
dab_dct	256	50000	1	0.83018411	PASSED
dab_filltree	32	15000000	1	0.00000000	FAILED
dab_filltree	32	15000000	1	0.00000000	FAILED
dab_filltree2	0	5000000	1	0.00000000	FAILED
dab_filltree2	1	5000000	1	0.00000000	FAILED
dab_monobit2	12	65000000	1	1.00000000	FAILED

Таблиця Б.2 – Результати тестів NIST STS з блоком 28 байт

Statistical Test	P-Value	Proportion	Result
Frequency	0.616305	96/100	PASSED
BlockFrequency	0.759756	100/100	PASSED
CumulativeSums (Forward)	0.964295	98/100	PASSED
CumulativeSums (Reverse)	0.867692	98/100	PASSED
Runs	0.554420	99/100	PASSED
LongestRun	0.964295	100/100	PASSED
Rank	0.213309	99/100	PASSED

FFT	0.037566	98/100	PASSED
NonOverlappingTemplate (148 подтестов)	*Min:* 0.001030 *Max:* 0.987896	*Min:* 96/100 *Max:* 100/100	ALL PASSED
OverlappingTemplate	0.236810	99/100	PASSED
Universal	0.383827	99/100	PASSED
ApproximateEntropy	0.616305	99/100	PASSED
RandomExcursions (8 подтестов)	*Min:* 0.005762 *Max:* 0.883171	*Min:* 57/58 *Max:* 58/58	ALL PASSED
RandomExcursionsVariant (18 подтестов)	*Min:* 0.006661 *Max:* 0.851383	*Min:* 57/58 *Max:* 58/58	ALL PASSED
Serial (Test 1)	0.129620	99/100	PASSED
Serial (Test 2)	0.935716	99/100	PASSED
LinearComplexity	0.350485	100/100	PASSED

ДОДАТОК В

РЕЗУЛЬТАТИ ТЕСТІВ ДЛЯ SHA3-256 З БЛОКОМ 32 БАЙТ

Таблиця В.1 – Результати тестів Dieharder з блоком 32 байт

Test Name	ntup	tsamples	psamples	p-value	Assessment
diehard_birthdays	0	100	100	0.05483822	PASSED
diehard_operm5	0	1000000	100	0.00000237	WEAK
diehard_rank_32x32	0	40000	100	0.38561973	PASSED
diehard_rank_6x8	0	100000	100	0.02314292	PASSED
diehard_bitstream	0	2097152	100	0.76078306	PASSED
diehard_opso	0	2097152	100	0.00000000	FAILED
diehard_oqso	0	2097152	100	0.03906077	PASSED
diehard_dna	0	2097152	100	0.10839875	PASSED
diehard_count_1s_str	0	256000	100	0.78642827	PASSED
diehard_count_1s_byt	0	256000	100	0.71654334	PASSED
diehard_parking_lot	0	12000	100	0.25399711	PASSED
diehard_2dsphere	2	8000	100	0.70017843	PASSED
diehard_3dsphere	3	4000	100	0.99904214	WEAK
diehard_squeeze	0	100000	100	0.00000000	FAILED
diehard_sums	0	100	100	0.08767954	PASSED
diehard_runs	0	100000	100	0.27004823	PASSED
diehard_runs	0	100000	100	0.55205276	PASSED
diehard_craps	0	200000	100	0.47567768	PASSED
diehard_craps	0	200000	100	0.00009879	WEAK
marsaglia_tsang_gcd	0	10000000	100	0.00000000	FAILED
marsaglia_tsang_gcd	0	10000000	100	0.00000000	FAILED
sts_monobit	1	100000	100	0.00058357	WEAK
sts_runs	2	100000	100	0.03497105	PASSED
sts_serial	1	100000	100	0.00271013	WEAK
sts_serial	2	100000	100	0.22779934	PASSED
sts_serial	3	100000	100	0.40408698	PASSED
sts_serial	3	100000	100	0.52444699	PASSED
sts_serial	4	100000	100	0.02617516	PASSED
sts_serial	4	100000	100	0.22096092	PASSED
sts_serial	5	100000	100	0.23891512	PASSED
sts_serial	5	100000	100	0.22278332	PASSED
sts_serial	6	100000	100	0.64135630	PASSED
sts_serial	6	100000	100	0.74526233	PASSED
sts_serial	7	100000	100	0.73284363	PASSED
sts_serial	7	100000	100	0.59919562	PASSED
sts_serial	8	100000	100	0.53375871	PASSED
sts_serial	8	100000	100	0.90129476	PASSED

sts_serial	9	100000	100	0.02052625	PASSED
sts_serial	9	100000	100	0.00393354	WEAK
sts_serial	10	100000	100	0.77470699	PASSED
sts_serial	10	100000	100	0.48830578	PASSED
sts_serial	11	100000	100	0.29299380	PASSED
sts_serial	11	100000	100	0.12370506	PASSED
sts_serial	12	100000	100	0.45878401	PASSED
sts_serial	12	100000	100	0.74451812	PASSED
sts_serial	13	100000	100	0.79137400	PASSED
sts_serial	13	100000	100	0.55359181	PASSED
sts_serial	14	100000	100	0.09932261	PASSED
sts_serial	14	100000	100	0.52253308	PASSED
sts_serial	15	100000	100	0.01532365	PASSED
sts_serial	15	100000	100	0.36477042	PASSED
sts_serial	16	100000	100	0.98886734	PASSED
sts_serial	16	100000	100	0.89662947	PASSED
rgb_bitdist	1	100000	100	0.45424627	PASSED
rgb_bitdist	2	100000	100	0.16987688	PASSED
rgb_bitdist	3	100000	100	0.15520719	PASSED
rgb_bitdist	4	100000	100	0.47173261	PASSED
rgb_bitdist	5	100000	100	0.11940287	PASSED
rgb_bitdist	6	100000	100	0.92440002	PASSED
rgb_bitdist	7	100000	100	0.33874707	PASSED
rgb_bitdist	8	100000	100	0.44542465	PASSED
rgb_bitdist	9	100000	100	0.95503009	PASSED
rgb_bitdist	10	100000	100	0.67226282	PASSED
rgb_bitdist	11	100000	100	0.76741083	PASSED
rgb_bitdist	12	100000	100	0.85013335	PASSED
rgb_minimum_distance	2	10000	1000	0.06887893	PASSED
rgb_minimum_distance	3	10000	1000	0.12377673	PASSED
rgb_minimum_distance	4	10000	1000	0.00000000	FAILED
rgb_minimum_distance	5	10000	1000	0.00000217	WEAK
rgb_permutations	2	100000	100	0.66314492	PASSED
rgb_permutations	3	100000	100	0.06916983	PASSED
rgb_permutations	4	100000	100	0.18705232	PASSED
rgb_permutations	5	100000	100	0.00000001	FAILED
rgb_lagged_sum	0	1000000	100	0.00020526	WEAK
rgb_lagged_sum	1	1000000	100	0.00000005	FAILED
rgb_lagged_sum	2	1000000	100	0.00000001	FAILED
rgb_lagged_sum	3	1000000	100	0.00000000	FAILED
rgb_lagged_sum	4	1000000	100	0.00000000	FAILED
rgb_lagged_sum	5	1000000	100	0.00000003	FAILED
rgb_lagged_sum	6	1000000	100	0.00001119	WEAK
rgb_lagged_sum	7	1000000	100	0.00000000	FAILED

rgb_lagged_sum	8	1000000	100	0.00004262	WEAK
rgb_lagged_sum	9	1000000	100	0.00000000	FAILED
rgb_lagged_sum	10	1000000	100	0.00055541	WEAK
rgb_lagged_sum	11	1000000	100	0.00000000	FAILED
rgb_lagged_sum	12	1000000	100	0.00000693	WEAK
rgb_lagged_sum	13	1000000	100	0.00000057	FAILED
rgb_lagged_sum	14	1000000	100	0.00000000	FAILED
rgb_lagged_sum	15	1000000	100	0.00000000	FAILED
rgb_lagged_sum	16	1000000	100	0.00000001	FAILED
rgb_lagged_sum	17	1000000	100	0.00000000	FAILED
rgb_lagged_sum	18	1000000	100	0.00000001	FAILED
rgb_lagged_sum	19	1000000	100	0.00000000	FAILED
rgb_lagged_sum	20	1000000	100	0.00006519	WEAK
rgb_lagged_sum	21	1000000	100	0.00020072	WEAK
rgb_lagged_sum	22	1000000	100	0.00003842	WEAK
rgb_lagged_sum	23	1000000	100	0.00000000	FAILED
rgb_lagged_sum	24	1000000	100	0.00000000	FAILED
rgb_lagged_sum	25	1000000	100	0.00000671	WEAK
rgb_lagged_sum	26	1000000	100	0.00002523	WEAK
rgb_lagged_sum	27	1000000	100	0.00000000	FAILED
rgb_lagged_sum	28	1000000	100	0.00000000	FAILED
rgb_lagged_sum	29	1000000	100	0.00000000	FAILED
rgb_lagged_sum	30	1000000	100	0.01414256	PASSED
rgb_lagged_sum	31	1000000	100	0.00000000	FAILED
rgb_lagged_sum	32	1000000	100	0.00000000	FAILED
rgb_kstest_test	0	10000	1000	0.10242335	PASSED
dab_bytedistrib	0	51200000	1	0.00000000	FAILED
dab_dct	256	50000	1	0.22092543	PASSED
dab_filltree	32	15000000	1	0.00000000	FAILED
dab_filltree	32	15000000	1	0.00000000	FAILED
dab_filltree2	0	5000000	1	0.00000000	FAILED
dab_filltree2	1	5000000	1	0.00000000	FAILED
dab_monobit2	12	65000000	1	1.00000000	FAILED

Таблиця В.2 – Результати тестів NIST STS з блоком 32 байт

Statistical Test	P-Value	Proportion	Result
Frequency	0.759756	98/100	PASSED
BlockFrequency	0.437274	98/100	PASSED
CumulativeSums (Forward)	0.616305	97/100	PASSED
CumulativeSums (Reverse)	0.867692	98/100	PASSED
Runs	0.678686	98/100	PASSED
LongestRun	0.637119	100/100	PASSED
Rank	0.991468	100/100	PASSED

FFT	0.289667	99/100	PASSED
NonOverlappingTemplate (148 подтестов)	Min: 0.004629 Max: 0.998821	Min: 96/100 Max: 100/100	ALL PASSED
OverlappingTemplate	0.514124	96/100	PASSED
Universal	0.514124	97/100	PASSED
ApproximateEntropy	0.657933	98/100	PASSED
RandomExcursions (8 подтестов)	Min: 0.264458 Max: 0.988549	Min: 61/63 Max: 63/63	ALL PASSED
RandomExcursionsVariant (18 подтестов)	Min: 0.033288 Max: 0.970538	Min: 62/63 Max: 63/63	ALL PASSED
Serial (Test 1)	0.991468	98/100	PASSED
Serial (Test 2)	0.911413	99/100	PASSED
LinearComplexity	0.455937	100/100	PASSED

ДОДАТОК Г

РЕЗУЛЬТАТИ ТЕСТІВ ДЛЯ SHA3-384 З БЛОКОМ 48 БАЙТ

Таблиця Г.1 – Результати тестів Dieharder з блоком 48 байт

Test Name	ntup	tsamples	psamples	p-value	Assessment
diehard_birthdays	0	100	100	0.88166438	PASSED
diehard_operm5	0	1000000	100	0.00411855	WEAK
diehard_rank_32x32	0	40000	100	0.04695429	PASSED
diehard_rank_6x8	0	100000	100	0.00541366	PASSED
diehard_bitstream	0	2097152	100	0.08472316	PASSED
diehard_opso	0	2097152	100	0.00036851	WEAK
diehard_oqso	0	2097152	100	0.00000000	FAILED
diehard_dna	0	2097152	100	0.09926963	PASSED
diehard_count_1s_str	0	256000	100	0.00941820	PASSED
diehard_count_1s_byt	0	256000	100	0.92558201	PASSED
diehard_parking_lot	0	12000	100	0.71038532	PASSED
diehard_2dsphere	2	8000	100	0.91146975	PASSED
diehard_3dsphere	3	4000	100	0.96106028	PASSED
diehard_squeeze	0	100000	100	0.00000207	WEAK
diehard_sums	0	100	100	0.26400271	PASSED
diehard_runs	0	100000	100	0.98566515	PASSED
diehard_runs	0	100000	100	0.86156968	PASSED
diehard_craps	0	200000	100	0.00000000	FAILED
diehard_craps	0	200000	100	0.01215006	PASSED
marsaglia_tsang_gcd	0	10000000	100	0.00000000	FAILED
marsaglia_tsang_gcd	0	10000000	100	0.00000000	FAILED
sts_monobit	1	100000	100	0.01701602	PASSED
sts_runs	2	100000	100	0.29056370	PASSED
sts_serial	1	100000	100	0.09854455	PASSED
sts_serial	2	100000	100	0.13071987	PASSED
sts_serial	3	100000	100	0.01078505	PASSED
sts_serial	3	100000	100	0.15151961	PASSED
sts_serial	4	100000	100	0.04119458	PASSED
sts_serial	4	100000	100	0.55546618	PASSED
sts_serial	5	100000	100	0.07904706	PASSED
sts_serial	5	100000	100	0.68152502	PASSED
sts_serial	6	100000	100	0.06439760	PASSED
sts_serial	6	100000	100	0.67400798	PASSED
sts_serial	7	100000	100	0.36772277	PASSED
sts_serial	7	100000	100	0.56767359	PASSED
sts_serial	8	100000	100	0.00042669	WEAK
sts_serial	8	100000	100	0.00001813	WEAK

sts_serial	9	100000	100	0.00861746	PASSED
sts_serial	9	100000	100	0.62198658	PASSED
sts_serial	10	100000	100	0.10702299	PASSED
sts_serial	10	100000	100	0.14524742	PASSED
sts_serial	11	100000	100	0.50507019	PASSED
sts_serial	11	100000	100	0.38010747	PASSED
sts_serial	12	100000	100	0.31911539	PASSED
sts_serial	12	100000	100	0.15314378	PASSED
sts_serial	13	100000	100	0.42362016	PASSED
sts_serial	13	100000	100	0.98013113	PASSED
sts_serial	14	100000	100	0.35632235	PASSED
sts_serial	14	100000	100	0.19662051	PASSED
sts_serial	15	100000	100	0.21317167	PASSED
sts_serial	15	100000	100	0.03792279	PASSED
sts_serial	16	100000	100	0.70426514	PASSED
sts_serial	16	100000	100	0.23704397	PASSED
rgb_bitdist	1	100000	100	0.00193135	WEAK
rgb_bitdist	2	100000	100	0.28467280	PASSED
rgb_bitdist	3	100000	100	0.82557329	PASSED
rgb_bitdist	4	100000	100	0.84582019	PASSED
rgb_bitdist	5	100000	100	0.07396732	PASSED
rgb_bitdist	6	100000	100	0.01254484	PASSED
rgb_bitdist	7	100000	100	0.93918033	PASSED
rgb_bitdist	8	100000	100	0.06045992	PASSED
rgb_bitdist	9	100000	100	0.64645831	PASSED
rgb_bitdist	10	100000	100	0.05037400	PASSED
rgb_bitdist	11	100000	100	0.96483074	PASSED
rgb_bitdist	12	100000	100	0.85538946	PASSED
rgb_minimum_distance	2	10000	1000	0.01660608	PASSED
rgb_minimum_distance	3	10000	1000	0.03468271	PASSED
rgb_minimum_distance	4	10000	1000	0.02322980	PASSED
rgb_minimum_distance	5	10000	1000	0.47618808	PASSED
rgb_permutations	2	100000	100	0.00049749	WEAK
rgb_permutations	3	100000	100	0.11257480	PASSED
rgb_permutations	4	100000	100	0.02593817	PASSED
rgb_permutations	5	100000	100	0.27759622	PASSED
rgb_lagged_sum	0	1000000	100	0.00000000	FAILED
rgb_lagged_sum	1	1000000	100	0.00000000	FAILED
rgb_lagged_sum	2	1000000	100	0.00000000	FAILED
rgb_lagged_sum	3	1000000	100	0.00000000	FAILED
rgb_lagged_sum	4	1000000	100	0.00000000	FAILED
rgb_lagged_sum	5	1000000	100	0.00000000	FAILED
rgb_lagged_sum	6	1000000	100	0.00000000	FAILED
rgb_lagged_sum	7	1000000	100	0.00000000	FAILED

rgb_lagged_sum	8	1000000	100	0.00000000	FAILED
rgb_lagged_sum	9	1000000	100	0.00000000	FAILED
rgb_lagged_sum	10	1000000	100	0.00000000	FAILED
rgb_lagged_sum	11	1000000	100	0.00000000	FAILED
rgb_lagged_sum	12	1000000	100	0.00000000	FAILED
rgb_lagged_sum	13	1000000	100	0.00000000	FAILED
rgb_lagged_sum	14	1000000	100	0.00000000	FAILED
rgb_lagged_sum	15	1000000	100	0.00000000	FAILED
rgb_lagged_sum	16	1000000	100	0.00000000	FAILED
rgb_lagged_sum	17	1000000	100	0.00000000	FAILED
rgb_lagged_sum	18	1000000	100	0.00000000	FAILED
rgb_lagged_sum	19	1000000	100	0.00000000	FAILED
rgb_lagged_sum	20	1000000	100	0.00000000	FAILED
rgb_lagged_sum	21	1000000	100	0.00000000	FAILED
rgb_lagged_sum	22	1000000	100	0.00000000	FAILED
rgb_lagged_sum	23	1000000	100	0.00000000	FAILED
rgb_lagged_sum	24	1000000	100	0.00000000	FAILED
rgb_lagged_sum	25	1000000	100	0.00000000	FAILED
rgb_lagged_sum	26	1000000	100	0.00000000	FAILED
rgb_lagged_sum	27	1000000	100	0.00000000	FAILED
rgb_lagged_sum	28	1000000	100	0.00000000	FAILED
rgb_lagged_sum	29	1000000	100	0.00000000	FAILED
rgb_lagged_sum	30	1000000	100	0.00000000	FAILED
rgb_lagged_sum	31	1000000	100	0.00000000	FAILED
rgb_lagged_sum	32	1000000	100	0.00000000	FAILED
rgb_kstest_test	0	10000	1000	0.00124596	WEAK
dab_bytedistrib	0	51200000	1	0.00000000	FAILED
dab_dct	256	50000	1	0.41020996	PASSED
dab_filltree	32	15000000	1	0.00000736	WEAK
dab_filltree	32	15000000	1	0.00000000	FAILED
dab_filltree2	0	5000000	1	0.00000000	FAILED
dab_filltree2	1	5000000	1	0.00000000	FAILED
dab_monobit2	12	65000000	1	1.00000000	FAILED

Таблиця Г.2 – Результати тестів NIST STS з блоком 48 байт

Statistical Test	P-Value	Proportion	Result
Frequency	0.867692	98/100	PASSED
BlockFrequency	0.455937	99/100	PASSED
CumulativeSums (Forward)	0.719747	99/100	PASSED
CumulativeSums (Reverse)	0.834308	99/100	PASSED
Runs	0.401199	99/100	PASSED
LongestRun	0.437274	100/100	PASSED
Rank	0.514124	99/100	PASSED

FFT	0.262249	100/100	PASSED
NonOverlappingTemplate (148 подтестов)	Min: 0.011791 Max: 0.998821	Min: 96/100 Max: 100/100	ALL PASSED
OverlappingTemplate	0.759756	99/100	PASSED
Universal	0.075719	100/100	PASSED
ApproximateEntropy	0.102526	99/100	PASSED
RandomExcursions (8 подтестов)	Min: 0.023812 Max: 0.988549	Min: 60/63 Max: 63/63	ALL PASSED
RandomExcursionsVariant (18 подтестов)	Min: 0.013411 Max: 0.970538	Min: 62/63 Max: 63/63	ALL PASSED
Serial (Test 1)	0.719747	99/100	PASSED
Serial (Test 2)	0.759756	97/100	PASSED
LinearComplexity	0.289667	100/100	PASSED

ДОДАТОК Д

РЕЗУЛЬТАТИ ТЕСТІВ ДЛЯ SHA3-512 З БЛОКОМ 64 БАЙТ

Таблиця Д.1 – Результати тестів Dieharder з блоком 64 байт

Test Name	ntup	tsamples	psamples	p-value	Assessment
diehard_birthdays	0	100	100	0.33294321	PASSED
diehard_operm5	0	1000000	100	0.09215772	PASSED
diehard_rank_32x32	0	40000	100	0.00004647	WEAK
diehard_rank_6x8	0	100000	100	0.56358337	PASSED
diehard_bitstream	0	2097152	100	0.90966433	PASSED
diehard_opso	0	2097152	100	0.00062332	WEAK
diehard_oqso	0	2097152	100	0.00000028	FAILED
diehard_dna	0	2097152	100	0.15434581	PASSED
diehard_count_1s_str	0	256000	100	0.92727265	PASSED
diehard_count_1s_byt	0	256000	100	0.41206919	PASSED
diehard_parking_lot	0	12000	100	0.07295537	PASSED
diehard_2dsphere	2	8000	100	0.26744873	PASSED
diehard_3dsphere	3	4000	100	0.67765116	PASSED
diehard_squeeze	0	100000	100	0.00000000	FAILED
diehard_sums	0	100	100	0.70170347	PASSED
diehard_runs	0	100000	100	0.04461644	PASSED
diehard_runs	0	100000	100	0.47745058	PASSED
diehard_craps	0	200000	100	0.00000002	FAILED
diehard_craps	0	200000	100	0.31886151	PASSED
marsaglia_tsang_gcd	0	10000000	100	0.00000000	FAILED
marsaglia_tsang_gcd	0	10000000	100	0.00000000	FAILED
sts_monobit	1	100000	100	0.00828909	PASSED
sts_runs	2	100000	100	0.35339562	PASSED
sts_serial	1	100000	100	0.00253522	WEAK
sts_serial	2	100000	100	0.47243636	PASSED
sts_serial	3	100000	100	0.83210056	PASSED
sts_serial	3	100000	100	0.78210360	PASSED
sts_serial	4	100000	100	0.99234149	PASSED
sts_serial	4	100000	100	0.54099357	PASSED
sts_serial	5	100000	100	0.77804643	PASSED
sts_serial	5	100000	100	0.68155506	PASSED
sts_serial	6	100000	100	0.35786175	PASSED
sts_serial	6	100000	100	0.44370519	PASSED
sts_serial	7	100000	100	0.17917387	PASSED
sts_serial	7	100000	100	0.87564840	PASSED
sts_serial	8	100000	100	0.98288824	PASSED
sts_serial	8	100000	100	0.48090285	PASSED

sts_serial	9	100000	100	0.84384273	PASSED
sts_serial	9	100000	100	0.51680332	PASSED
sts_serial	10	100000	100	0.14042985	PASSED
sts_serial	10	100000	100	0.13106744	PASSED
sts_serial	11	100000	100	0.15397564	PASSED
sts_serial	11	100000	100	0.85406867	PASSED
sts_serial	12	100000	100	0.72817667	PASSED
sts_serial	12	100000	100	0.09371183	PASSED
sts_serial	13	100000	100	0.04088708	PASSED
sts_serial	13	100000	100	0.22871592	PASSED
sts_serial	14	100000	100	0.21354815	PASSED
sts_serial	14	100000	100	0.08892353	PASSED
sts_serial	15	100000	100	0.60295317	PASSED
sts_serial	15	100000	100	0.46779796	PASSED
sts_serial	16	100000	100	0.25308763	PASSED
sts_serial	16	100000	100	0.00083307	WEAK
rgb_bitdist	1	100000	100	0.82987392	PASSED
rgb_bitdist	2	100000	100	0.79410059	PASSED
rgb_bitdist	3	100000	100	0.21998723	PASSED
rgb_bitdist	4	100000	100	0.16666230	PASSED
rgb_bitdist	5	100000	100	0.99636090	WEAK
rgb_bitdist	6	100000	100	0.04441818	PASSED
rgb_bitdist	7	100000	100	0.75373644	PASSED
rgb_bitdist	8	100000	100	0.47397910	PASSED
rgb_bitdist	9	100000	100	0.24706085	PASSED
rgb_bitdist	10	100000	100	0.70067709	PASSED
rgb_bitdist	11	100000	100	0.56752318	PASSED
rgb_bitdist	12	100000	100	0.64695869	PASSED
rgb_minimum_distance	2	10000	1000	0.03672297	PASSED
rgb_minimum_distance	3	10000	1000	0.06366278	PASSED
rgb_minimum_distance	4	10000	1000	0.00055416	WEAK
rgb_minimum_distance	5	10000	1000	0.09482853	PASSED
rgb_permutations	2	100000	100	0.02923637	PASSED
rgb_permutations	3	100000	100	0.25615365	PASSED
rgb_permutations	4	100000	100	0.00047733	WEAK
rgb_permutations	5	100000	100	0.75546592	PASSED
rgb_lagged_sum	0	1000000	100	0.00000241	WEAK
rgb_lagged_sum	1	1000000	100	0.00000000	FAILED
rgb_lagged_sum	2	1000000	100	0.00000000	FAILED
rgb_lagged_sum	3	1000000	100	0.00000000	FAILED
rgb_lagged_sum	4	1000000	100	0.00000852	WEAK
rgb_lagged_sum	5	1000000	100	0.00000000	FAILED
rgb_lagged_sum	6	1000000	100	0.00027331	WEAK
rgb_lagged_sum	7	1000000	100	0.00000000	FAILED

rgb_lagged_sum	8	1000000	100	0.00000013	FAILED
rgb_lagged_sum	9	1000000	100	0.00000000	FAILED
rgb_lagged_sum	10	1000000	100	0.00149166	WEAK
rgb_lagged_sum	11	1000000	100	0.00000000	FAILED
rgb_lagged_sum	12	1000000	100	0.00000000	FAILED
rgb_lagged_sum	13	1000000	100	0.00000000	FAILED
rgb_lagged_sum	14	1000000	100	0.00000002	FAILED
rgb_lagged_sum	15	1000000	100	0.00000000	FAILED
rgb_lagged_sum	16	1000000	100	0.00000000	FAILED
rgb_lagged_sum	17	1000000	100	0.00000000	FAILED
rgb_lagged_sum	18	1000000	100	0.00063551	WEAK
rgb_lagged_sum	19	1000000	100	0.00000000	FAILED
rgb_lagged_sum	20	1000000	100	0.00000000	FAILED
rgb_lagged_sum	21	1000000	100	0.00000000	FAILED
rgb_lagged_sum	22	1000000	100	0.00000000	FAILED
rgb_lagged_sum	23	1000000	100	0.00000000	FAILED
rgb_lagged_sum	24	1000000	100	0.00000186	WEAK
rgb_lagged_sum	25	1000000	100	0.00000000	FAILED
rgb_lagged_sum	26	1000000	100	0.00000132	WEAK
rgb_lagged_sum	27	1000000	100	0.00000000	FAILED
rgb_lagged_sum	28	1000000	100	0.00000000	FAILED
rgb_lagged_sum	29	1000000	100	0.00000000	FAILED
rgb_lagged_sum	30	1000000	100	0.00012389	WEAK
rgb_lagged_sum	31	1000000	100	0.00000000	FAILED
rgb_lagged_sum	32	1000000	100	0.00000000	FAILED
rgb_kstest_test	0	10000	1000	0.26734715	PASSED
dab_bytedistrib	0	51200000	1	0.00000000	FAILED
dab_dct	256	50000	1	0.01658460	PASSED
dab_filltree	32	15000000	1	0.00000000	FAILED
dab_filltree	32	15000000	1	0.00000000	FAILED
dab_filltree2	0	5000000	1	0.00000000	FAILED
dab_filltree2	1	5000000	1	0.00000000	FAILED
dab_monobit2	12	65000000	1	1.00000000	FAILED

Таблиця Д.2 – Результати тестів NIST STS з блоком 64 байт

Statistical Test	P-Value	Proportion	Result
Frequency	0.055361	97/100	PASSED
BlockFrequency	0.304126	100/100	PASSED
CumulativeSums (Forward)	0.262249	96/100	PASSED
CumulativeSums (Reverse)	0.145326	98/100	PASSED
Runs	0.236810	99/100	PASSED
LongestRun	0.595549	100/100	PASSED
Rank	0.514124	100/100	PASSED

FFT	0.924076	97/100	PASSED
NonOverlappingTemplate (148 подтестов)	Min: 0.012650 Max: 0.999777	Min: 96/100 Max: 100/100	ALL PASSED
OverlappingTemplate	0.946308	100/100	PASSED
Universal	0.678686	100/100	PASSED
ApproximateEntropy	0.739918	99/100	PASSED
RandomExcursions (8 подтестов)	Min: 0.178278 Max: 0.848588	Min: 60/62 Max: 62/62	ALL PASSED
RandomExcursionsVariant (18 подтестов)	Min: 0.013411 Max: 0.931952	Min: 60/62 Max: 63/63	ALL PASSED
Serial (Test 1)	0.115387	100/100	PASSED
Serial (Test 2)	0.657933	98/100	PASSED
LinearComplexity	0.437274	99/100	PASSED

ДОДАТОК Е

ПРОГРАМНА РЕАЛІЗАЦІЯ ЄКСТРАТОРА SHA3

Лістринг Е.1 – Повний код програми

```
import customtkinter as ctk
from tkinter import filedialog, messagebox
import hashlib
import threading
import time
import os

class SHA3Extractor(ctk.CTk):
    """Головний клас програми SHA-3 QRNG Extractor"""

    def __init__(self):
        """Ініціалізація головного вікна та параметрів програми"""
        super().__init__()

        # Налаштування вікна
        self.title("SHA-3 QRNG Extractor")
        self.geometry("750x800")

        # Налаштування теми
        ctk.set_appearance_mode("dark")
        ctk.set_default_color_theme("blue")

        # Змінні стану
        self.input_file = None
        self.output_file = None
        self.processing = False

        self.create_widgets()

    def create_widgets(self):
        """Створення елементів інтерфейсу"""
        # Створюємо TabView
        self.tabview = ctk.CTkTabview(self, width=710, height=750)
        self.tabview.pack(padx=20, pady=20, fill="both",
expand=True)

        # Додаємо вкладки
        self.tabview.add("Extractor")
        self.tabview.add("About")

        # Налаштовуємо вкладки
        self.setup_extractor_tab()
        self.setup_about_tab()

    def setup_extractor_tab(self):
        """Налаштування робочої вкладки Extractor"""
```

```
        tab = self.tabview.tab("Extractor")

        # Заголовок
        title_label = ctk.CTkLabel(
            tab,
            text="SHA-3 QRNG Extractor",
            font=ctk.CTkFont(size=24, weight="bold")
        )
        title_label.pack(pady=15)

        # Фрейм для файлів
        file_frame = ctk.CTkFrame(tab)
        file_frame.pack(padx=20, pady=10, fill="x")

        # Вхідний файл
        ctk.CTkLabel(file_frame, text="Input File:").grid(row=0,
column=0, padx=10, pady=10, sticky="w")
        self.input_label = ctk.CTkLabel(file_frame, text="No file
selected", text_color="gray")
        self.input_label.grid(row=0, column=1, padx=10, pady=10,
sticky="w")
        input_btn = ctk.CTkButton(file_frame, text="Browse",
command=self.select_input_file, width=100)
        input_btn.grid(row=0, column=2, padx=10, pady=10)
        self.create_tooltip(input_btn, "Select your QRNG raw data
file")

        # Вихідний файл
        ctk.CTkLabel(file_frame, text="Output File:").grid(row=1,
column=0, padx=10, pady=10, sticky="w")
        self.output_label = ctk.CTkLabel(file_frame, text="No file
selected", text_color="gray")
        self.output_label.grid(row=1, column=1, padx=10, pady=10,
sticky="w")
        output_btn = ctk.CTkButton(file_frame, text="Save As",
command=self.select_output_file, width=100)
        output_btn.grid(row=1, column=2, padx=10, pady=10)
        self.create_tooltip(output_btn, "Choose where to save
processed file")

        # Фрейм налаштувань
        settings_frame = ctk.CTkFrame(tab)
        settings_frame.pack(padx=20, pady=10, fill="x")

        # Перша строка: SHA-3 Variant і Block Size
        row1_frame = ctk.CTkFrame(settings_frame,
fg_color="transparent")
        row1_frame.pack(padx=10, pady=5, fill="x")

        # SHA-3 Variant (зліва)
        variant_container = ctk.CTkFrame(row1_frame,
fg_color="transparent")
        variant_container.pack(side="left", padx=5)
```

```
        variant_label = ctk.CTkLabel(variant_container, text="SHA-3
Variant:")
        variant_label.pack(anchor="w")
        self.sha3_variant = ctk.CTkComboBox(
            variant_container,
            values=["SHA3-224", "SHA3-256", "SHA3-384", "SHA3-512",
"SHAKE128", "SHAKE256"],
            command=self.on_variant_change,
            width=140
        )
        self.sha3_variant.set("SHA3-256")
        self.sha3_variant.pack(anchor="w")
        self.create_tooltip(self.sha3_variant, "Select hash function
variant\nSHA3: fixed output\nSHAKE: extendable output")

        # Block Size (справа)
        block_container = ctk.CTkFrame(row1_frame,
fg_color="transparent")
        block_container.pack(side="left", padx=20)
        block_label = ctk.CTkLabel(block_container, text="Block Size
(bytes):")
        block_label.pack(anchor="w")
        self.block_size_entry = ctk.CTkEntry(block_container,
width=140)
        self.block_size_entry.insert(0, "256")
        self.block_size_entry.pack(anchor="w")
        self.create_tooltip(self.block_size_entry, "Size of input
chunks to hash\nLarger blocks = smaller output file")

        # SHAKE Output (справа, показується тільки для SHAKE)
        self.shake_container = ctk.CTkFrame(row1_frame,
fg_color="transparent")
        shake_label = ctk.CTkLabel(self.shake_container, text="SHAKE
Output (bytes):")
        shake_label.pack(anchor="w")
        self.shake_output_entry = ctk.CTkEntry(self.shake_container,
width=140)
        self.shake_output_entry.insert(0, "256")
        self.shake_output_entry.pack(anchor="w")
        self.create_tooltip(self.shake_output_entry, "Output size
per block for SHAKE\nSet equal to block size to preserve file size")

        # Друга строка: Presets для розміру блоку
        preset_frame = ctk.CTkFrame(settings_frame,
fg_color="transparent")
        preset_frame.pack(padx=10, pady=5, fill="x")
        ctk.CTkLabel(preset_frame, text="Block Size
Presets:").pack(side="left", padx=5)
        for size in [32, 64, 128, 256, 512, 1024]:
            btn = ctk.CTkButton(
                preset_frame,
                text=str(size),
```

```
                command=lambda s=size:
self.block_size_entry.delete(0, "end") or
self.block_size_entry.insert(0, str(s)),
                width=55,
                height=28
            )
            btn.pack(side="left", padx=2)

        # Progress bar
        self.progress_bar = ctk.CTkProgressBar(tab, width=680)
        self.progress_bar.pack(padx=20, pady=10)
        self.progress_bar.set(0)

        # Статус
        self.status_label = ctk.CTkLabel(tab, text="Ready",
font=ctk.CTkFont(size=12))
        self.status_label.pack(pady=5)

        # Лог
        log_label = ctk.CTkLabel(tab, text="Processing Log:",
font=ctk.CTkFont(size=12, weight="bold"))
        log_label.pack(pady=(10, 5))

        self.log_text = ctk.CTkTextbox(tab, width=680, height=180)
        self.log_text.pack(padx=20, pady=5)
        self.log("SHA-3 QRNG Extractor v1.0 - Ready")
        self.log("Select input file and output destination to
begin")

        # Кнопки управління
        button_frame = ctk.CTkFrame(tab)
        button_frame.pack(pady=15)

        self.process_button = ctk.CTkButton(
            button_frame,
            text="▶ Process File",
            command=self.start_processing,
            width=180,
            height=40,
            font=ctk.CTkFont(size=16, weight="bold"),
            fg_color="#1f6aa5",
            hover_color="#144870"
        )
        self.process_button.pack(side="left", padx=5)

        clear_log_btn = ctk.CTkButton(
            button_frame,
            text="Clear Log",
            command=self.clear_log,
            width=100,
            height=40,
            fg_color="#666666",
            hover_color="#444444"
```

```
        )
        clear_log_btn.pack(side="left", padx=5)

    def setup_about_tab(self):
        """Налаштування інформаційної вкладки About"""
        tab = self.tabview.tab("About")

        # Прокручуваний фрейм для контенту
        scroll_frame = ctk.CTkScrollableFrame(tab, width=680,
height=680)
        scroll_frame.pack(padx=20, pady=20, fill="both",
expand=True)

        # Заголовок
        title = ctk.CTkLabel(
            scroll_frame,
            text="SHA-3 QRNG Extractor",
            font=ctk.CTkFont(size=28, weight="bold")
        )
        title.pack(pady=20)

        # Версія
        version = ctk.CTkLabel(
            scroll_frame,
            text="Version 1.0",
            font=ctk.CTkFont(size=14),
            text_color="gray"
        )
        version.pack()

        # Розділювач
        separator1 = ctk.CTkFrame(scroll_frame, height=2,
fg_color="gray")
        separator1.pack(fill="x", padx=50, pady=20)

        # Опис програми
        desc_title = ctk.CTkLabel(
            scroll_frame,
            text="📦 About This Program",
            font=ctk.CTkFont(size=18, weight="bold")
        )
        desc_title.pack(pady=(10, 10), anchor="w", padx=20)

        description = """This application is designed for
statistical analysis of Quantum Random Number
Generators (QRNG) using SHA-3 cryptographic hash functions as
randomness extractors.

The program processes raw QRNG output and applies SHA-3 based
extraction to improve
statistical properties of the generated random data."""

        desc_text = ctk.CTkLabel(
```

```
            scroll_frame,
            text=description,
            font=ctk.CTkFont(size=13),
            justify="left",
            wraplength=640
        )
        desc_text.pack(pady=10, padx=20)

        # Роздiлювач
        separator2 = ctk.CTkFrame(scroll_frame, height=2,
fg_color="gray")
        separator2.pack(fill="x", padx=50, pady=20)

        # SHA-3 Variants
        sha3_title = ctk.CTkLabel(
            scroll_frame,
            text="🔐 SHA-3 Variants Explained",
            font=ctk.CTkFont(size=18, weight="bold")
        )
        sha3_title.pack(pady=(10, 10), anchor="w", padx=20)

        variants_info = """
SHA3-224: Fixed 224-bit (28 bytes) output hash
SHA3-256: Fixed 256-bit (32 bytes) output hash  ← Most common
SHA3-384: Fixed 384-bit (48 bytes) output hash
SHA3-512: Fixed 512-bit (64 bytes) output hash

SHAKE128: Extendable output function (XOF) - variable length output
SHAKE256: Extendable output function (XOF) - variable length
output  ← Best for extractors

💡 Tip: Use SHAKE variants to preserve input file size and generate
more test data."""

        variants_text = ctk.CTkLabel(
            scroll_frame,
            text=variants_info,
            font=ctk.CTkFont(size=12),
            justify="left",
            wraplength=640
        )
        variants_text.pack(pady=10, padx=20)

        # Роздiлювач
        separator3 = ctk.CTkFrame(scroll_frame, height=2,
fg_color="gray")
        separator3.pack(fill="x", padx=50, pady=20)

        # Key Concepts
        concepts_title = ctk.CTkLabel(
            scroll_frame,
            text="📚 Key Concepts",
            font=ctk.CTkFont(size=18, weight="bold")
```

```
        )
        concepts_title.pack(pady=(10, 10), anchor="w", padx=20)

        concepts_info = """
Block Size: The size of input data chunks to be hashed
• Larger blocks → Fewer output data
• Smaller blocks → More output data, but longer processing time

Output File: Binary file suitable for statistical testing
• NIST Statistical Test Suite (STS)
• Dieharder Random Number Test Suite
• Recommended minimum size: 10-50 MB for comprehensive testing

Compression Ratio: Input size / Output size
• SHA3-256 with 256-byte blocks: ~8:1 compression
• SHAKE256 with equal block/output: 1:1 (no compression)"""

        concepts_text = ctk.CTkLabel(
            scroll_frame,
            text=concepts_info,
            font=ctk.CTkFont(size=12),
            justify="left",
            wraplength=640
        )
        concepts_text.pack(pady=10, padx=20)

        # Роздiлювач
        separator4 = ctk.CTkFrame(scroll_frame, height=2,
fg_color="gray")
        separator4.pack(fill="x", padx=50, pady=20)

        # Author info
        author_title = ctk.CTkLabel(
            scroll_frame,
            text="👨‍🎓 Author Information",
            font=ctk.CTkFont(size=18, weight="bold")
        )
        author_title.pack(pady=(10, 10), anchor="w", padx=20)

        author_info = """
Diploma Thesis: Statistical Analysis of a QRNG Extractor Based on
the
SHA-3 Cryptographic Hash Function

Created for academic research and statistical analysis of quantum
random
number generators.

Developed with Python and CustomTkinter framework."""

        author_text = ctk.CTkLabel(
            scroll_frame,
            text=author_info,
```

```
            font=ctk.CTkFont(size=12),
            justify="left",
            wraplength=640
        )
        author_text.pack(pady=10, padx=20)

        # Footer
        footer = ctk.CTkLabel(
            scroll_frame,
            text="© 2025 - For Educational and Research Purposes",
            font=ctk.CTkFont(size=11),
            text_color="gray"
        )
        footer.pack(pady=30)

    def create_tooltip(self, widget, text):
        """Створює підказку для елемента інтерфейсу"""
        def on_enter(event):
            self.status_label.configure(text=text)

        def on_leave(event):
            if not self.processing:
                self.status_label.configure(text="Ready")

        widget.bind("<Enter>", on_enter)
        widget.bind("<Leave>", on_leave)

    def clear_log(self):
        """Очищає журнал подій"""
        self.log_text.delete("0.0", "end")
        self.log("Log cleared")

    def on_variant_change(self, choice):
        """Показує/приховує поле SHAKE Output залежно від обраного
варіанту"""
        if choice.startswith("SHAKE"):
            self.shake_container.pack(side="left", padx=20)
        else:
            self.shake_container.pack_forget()

    def calculate_statistics(self, filename):
        """
        Обчислює детальні статистичні характеристики файлу:
        1. Bias (зміщення) - відхилення від 50/50 розподілу
        2. Shannon Entropy - інформаційна ентропія
        3. Min-Entropy - мінімальна ентропія (передбачуваність)
        4. Serial Correlation - кореляція між сусідніми бітами
        5. Runs Test - тест серій (кількість переходів)
        """
        import math

        total_zeros = 0
        total_ones = 0
```

```
        bits = []  # Для розрахунку кореляцій (беремо перші 1M біт)
        max_bits_for_correlation = 1_000_000  # Обмежуємо для
швидкості

        # Читаємо файл блоками
        with open(filename, 'rb') as f:
            while True:
                chunk = f.read(65536)  # 64 КБ
                if not chunk:
                    break

                # Підраховуємо біти
                for byte in chunk:
                    ones = bin(byte).count('1')
                    total_ones += ones
                    total_zeros += (8 - ones)

                    # Зберігаємо біти для кореляції (обмежена
кількість)
                    if len(bits) < max_bits_for_correlation:
                        for i in range(8):
                            bits.append((byte >> i) & 1)

        total_bits = total_zeros + total_ones

        # ========================================================
        # 1. BIAS (Зміщення)
        # ========================================================
        p_ones = total_ones / total_bits if total_bits > 0 else 0
        p_zeros = total_zeros / total_bits if total_bits > 0 else 0
        bias = abs(p_ones - 0.5)

        # ========================================================
        # 2. SHANNON ENTROPY (Ентропія Шеннона)
        # ========================================================
        # H = -Σ p(x) * log₂(p(x))
        # Ідеальне значення = 1.0 біт/біт

        if p_zeros > 0 and p_ones > 0:
            shannon_entropy = -(p_zeros * math.log2(p_zeros) +
p_ones * math.log2(p_ones))
        else:
            shannon_entropy = 0.0

        # ========================================================
        # 3. MIN-ENTROPY (Мінімальна ентропія)
        # ========================================================
        # H_min = -log₂(max(P(0), P(1)))
        # Ідеальне значення = 1.0 біт/біт

        max_prob = max(p_zeros, p_ones)
        if max_prob > 0:
            min_entropy = -math.log2(max_prob)
```

```
        else:
            min_entropy = 0.0

        # ====================================================
        # 4. SERIAL CORRELATION (Серійна кореляція)
        # ====================================================
        # Кореляція Пірсона між x[i] і x[i+1]
        # C = Cov(X,Y) / (σₓ * σᵧ)
        # Значення близьке до 0 вказує на незалежність

        serial_correlation = 0.0
        if len(bits) >= 100:  # Мінімум для надійної оцінки
            # Перетворюємо біти в {-1, +1} для розрахунку кореляції
            x = [2*b - 1 for b in bits[:-1]]  # x[i]
            y = [2*b - 1 for b in bits[1:]]   # x[i+1]

            n = len(x)
            mean_x = sum(x) / n
            mean_y = sum(y) / n

            # Коваріація
            cov = sum((x[i] - mean_x) * (y[i] - mean_y) for i in
range(n)) / n

            # Стандартні відхилення
            std_x = math.sqrt(sum((xi - mean_x)**2 for xi in x) / n)
            std_y = math.sqrt(sum((yi - mean_y)**2 for yi in y) / n)

            # Кореляція Пірсона
            if std_x > 0 and std_y > 0:
                serial_correlation = cov / (std_x * std_y)

        # ====================================================
        # 5. RUNS TEST (Тест серій)
        # ====================================================
        # Підраховуємо кількість переходів (runs)
        # Run = безперервна послідовність однакових біт
        # Для випадкових даних очікується певна кількість runs

        runs_count = 0
        runs_test_statistic = 0.0

        if len(bits) >= 100:
            # Підраховуємо runs (серії)
            runs_count = 1
            for i in range(1, len(bits)):
                if bits[i] != bits[i-1]:
                    runs_count += 1

            # Очікувана кількість runs для випадкових даних
            n = len(bits)
            n1 = sum(bits)  # Кількість одиниць
            n0 = n - n1     # Кількість нулів
```

```
            if n1 > 0 and n0 > 0:
                # Очікуване значення runs
                expected_runs = (2 * n0 * n1) / n + 1

                # Дисперсія runs
                variance_runs = (2 * n0 * n1 * (2 * n0 * n1 - n)) /
(n * n * (n - 1))

                if variance_runs > 0:
                    # Z-статистика (стандартизоване відхилення)
                    runs_test_statistic = (runs_count -
expected_runs) / math.sqrt(variance_runs)
                else:
                    runs_test_statistic = 0.0

        return {
            # Базовые данные
            'total_bits': total_bits,
            'zeros': total_zeros,
            'ones': total_ones,
            'p_zeros': p_zeros * 100,  # В процентах
            'p_ones': p_ones * 100,     # В процентах

            # Статистические метрики
            'bias': bias,
            'shannon_entropy': shannon_entropy,
            'min_entropy': min_entropy,
            'serial_correlation': serial_correlation,
            'runs_count': runs_count,
            'runs_test_stat': runs_test_statistic,
        }

    def select_input_file(self):
        """
        Відкриває діалог вибору вхідного файлу.
        Визначає розмір файлу та обчислює статистичні метрики.
        """
        filename = filedialog.askopenfilename(
            title="Select Input File",
            filetypes=[("All Files", "*.*"), ("Binary Files",
"*.bin"), ("Text Files", "*.txt")]
        )
        if filename:
            self.input_file = filename
            self.input_label.configure(text=os.path.basename(filenam
e), text_color="white")
            file_size = os.path.getsize(filename) / (1024 * 1024)

            self.log(f"✓ Input file selected:
{os.path.basename(filename)} ({file_size:.2f} MB)")
            self.log("  Analyzing statistical properties...")
```

```
            # Обчислюємо статистику в окремому потоці
            self.status_label.configure(text="Analyzing file
statistics...")

            def analyze():
                try:
                    stats = self.calculate_statistics(filename)

                    # Записуємо результати в журнал
                    self.log("  " + "=" * 58)
                    self.log(f"  INPUT FILE STATISTICS:")
                    self.log(f"  Total bits:
{stats['total_bits']:,}")
                    self.log(f"  Zeros: {stats['zeros']:,}
({stats['p_zeros']:.4f}%)")
                    self.log(f"  Ones:  {stats['ones']:,}
({stats['p_ones']:.4f}%)")
                    self.log("  " + "-" * 41)
                    self.log(f"  Bias: {stats['bias']:.6f} (ideal:
0.000000)")
                    self.log(f"  Shannon Entropy:
{stats['shannon_entropy']:.6f} bits/bit (ideal: 1.000000)")
                    self.log(f"  Min-Entropy:
{stats['min_entropy']:.6f} bits/bit (ideal: 1.000000)")
                    self.log(f"  Serial Correlation:
{stats['serial_correlation']:.6f} (ideal: 0.000000)")
                    self.log(f"  Runs: {stats['runs_count']:,} (Z-
stat: {stats['runs_test_stat']:.4f})")

                    # Оцінка якості
                    quality_scores = []

                    # Bias quality
                    if stats['bias'] < 0.001:
                        bias_quality = "Excellent"
                        quality_scores.append(5)
                    elif stats['bias'] < 0.005:
                        bias_quality = "Good"
                        quality_scores.append(4)
                    elif stats['bias'] < 0.01:
                        bias_quality = "Fair"
                        quality_scores.append(3)
                    else:
                        bias_quality = "Poor"
                        quality_scores.append(2)

                    # Shannon Entropy quality
                    shannon_dev = abs(1.0 -
stats['shannon_entropy'])
                    if shannon_dev < 0.001:
                        shannon_quality = "Excellent"
                        quality_scores.append(5)
                    elif shannon_dev < 0.005:
```

```
                        shannon_quality = "Good"
                        quality_scores.append(4)
                    elif shannon_dev < 0.01:
                        shannon_quality = "Fair"
                        quality_scores.append(3)
                    else:
                        shannon_quality = "Poor"
                        quality_scores.append(2)

                    # Correlation quality
                    corr_abs = abs(stats['serial_correlation'])
                    if corr_abs < 0.01:
                        corr_quality = "Excellent"
                        quality_scores.append(5)
                    elif corr_abs < 0.05:
                        corr_quality = "Good"
                        quality_scores.append(4)
                    elif corr_abs < 0.1:
                        corr_quality = "Fair"
                        quality_scores.append(3)
                    else:
                        corr_quality = "Poor"
                        quality_scores.append(2)

                    avg_quality = sum(quality_scores) /
len(quality_scores)
                    if avg_quality >= 4.5:
                        overall_quality = "Excellent"
                    elif avg_quality >= 3.5:
                        overall_quality = "Good"
                    elif avg_quality >= 2.5:
                        overall_quality = "Fair"
                    else:
                        overall_quality = "Poor"

                    self.log("  " + "─" * 41)
                    self.log(f"  Quality Assessment:")
                    self.log(f"    Bias: {bias_quality}")
                    self.log(f"    Shannon Entropy:
{shannon_quality}")
                    self.log(f"    Correlations: {corr_quality}")
                    self.log(f"    Overall: {overall_quality}")
                    self.log("  " + "═" * 58)

                    self.after(0, lambda:
self.status_label.configure(text="Ready"))

                except Exception as e:
                    self.log(f"  ✗ Statistical analysis failed:
{str(e)}")
                    self.after(0, lambda:
self.status_label.configure(text="Ready"))
```

```
            # Запускаємо аналіз в окремому потоці
            threading.Thread(target=analyze, daemon=True).start()

    def select_output_file(self):
        """Відкриває діалог вибору місця збереження вихідного
файлу"""
        filename = filedialog.asksaveasfilename(
            title="Save Output File",
            defaultextension=".bin",
            filetypes=[("Binary Files", "*.bin"), ("All Files",
"*.*")]
        )
        if filename:
            self.output_file = filename
            self.output_label.configure(text=os.path.basename(filena
me), text_color="white")
            self.log(f"✓ Output destination:
{os.path.basename(filename)}")

    def log(self, message):
        """
        Додає повідомлення до журналу з часовою міткою.
        Автоматично прокручує до останнього запису.
        """
        timestamp = time.strftime("%H:%M:%S")
        self.log_text.insert("end", f"[{timestamp}] {message}\n")
        self.log_text.see("end")

    def start_processing(self):
        """
        Ініціює процес обробки файлу.
        Виконує валідацію параметрів та запускає обробку в окремому
потоці.
        """
        # Перевірка: чи вже виконується обробка
        if self.processing:
            messagebox.showwarning("Warning", "Processing already in
progress!")
            return

        # Перевірка: чи обрано вхідний файл
        if not self.input_file:
            messagebox.showerror("Error", "Please select an input
file!")
            return

        # Перевірка: чи обрано вихідний файл
        if not self.output_file:
            messagebox.showerror("Error", "Please select an output
file!")
            return

        # Валідація розміру блоку
```

```
        try:
            block_size = int(self.block_size_entry.get())
            if block_size <= 0:
                raise ValueError
        except ValueError:
            messagebox.showerror("Error", "Block size must be a
positive integer!")
            return

        # Отримання варіанту SHA-3 та параметрів SHAKE
        variant = self.sha3_variant.get()
        shake_output_length = None

        # Валідація довжини виводу SHAKE (якщо обрано SHAKE-варіант)
        if variant.startswith("SHAKE"):
            try:
                shake_output_length =
int(self.shake_output_entry.get())
                if shake_output_length <= 0:
                    raise ValueError
            except ValueError:
                messagebox.showerror("Error", "SHAKE output length
must be a positive integer!")
                return

        # Запуск обробки в окремому потоці
        self.processing = True
        self.process_button.configure(state="disabled", text="⏳
Processing...")

        thread = threading.Thread(
            target=self.process_file,
            args=(block_size, variant, shake_output_length),
            daemon=True
        )
        thread.start()

    def process_file(self, block_size, variant,
shake_output_length):
        """
        ОСНОВНА ФУНКЦІЯ ОБРОБКИ ФАЙЛУ

        Виконує поблочне хешування вхідного файлу з використанням
SHA-3.
        Кожен блок даних обробляється незалежно.

        Аргументи:
            block_size: Розмір блоку для читання та хешування (в
байтах)
            variant: Обраний варіант SHA-3
            shake_output_length: Довжина виводу для SHAKE
        """
        try:
```

```
            start_time = time.time()
            file_size = os.path.getsize(self.input_file)
            processed_bytes = 0
            block_counter = 0
            update_interval = 100

            self.log("=" * 60)
            self.log(f"▶ Starting processing...")
            self.log(f"  Variant: {variant}")
            self.log(f"  Block Size: {block_size} bytes")
            if shake_output_length:
                self.log(f"  SHAKE Output Length:
{shake_output_length} bytes")

            total_blocks = file_size // block_size + (1 if file_size
% block_size else 0)
            self.log(f"  Total blocks to process: {total_blocks:,}")

            with open(self.input_file, 'rb') as infile,
open(self.output_file, 'wb') as outfile:
                while True:
                    chunk = infile.read(block_size)
                    if not chunk:
                        break

                    # Застосування SHA-3
                    if variant == "SHA3-224":
                        hash_obj = hashlib.sha3_224(chunk)
                        output = hash_obj.digest()
                    elif variant == "SHA3-256":
                        hash_obj = hashlib.sha3_256(chunk)
                        output = hash_obj.digest()
                    elif variant == "SHA3-384":
                        hash_obj = hashlib.sha3_384(chunk)
                        output = hash_obj.digest()
                    elif variant == "SHA3-512":
                        hash_obj = hashlib.sha3_512(chunk)
                        output = hash_obj.digest()
                    elif variant == "SHAKE128":
                        hash_obj = hashlib.shake_128(chunk)
                        output =
hash_obj.digest(shake_output_length)
                    elif variant == "SHAKE256":
                        hash_obj = hashlib.shake_256(chunk)
                        output =
hash_obj.digest(shake_output_length)

                    outfile.write(output)
                    processed_bytes += len(chunk)
                    block_counter += 1

                    # Оновлення UI кожні 100 блоків
                    if block_counter % update_interval == 0:
```

```
                        progress = processed_bytes / file_size
                        self.after(0, lambda p=progress:
self.progress_bar.set(p))
                        self.after(0, lambda p=progress:
self.status_label.configure(text=f"Processing: {p*100:.1f}%"))

            # Фінальне оновлення UI
            self.after(0, lambda: self.progress_bar.set(1.0))
            self.after(0, lambda:
self.status_label.configure(text="Processing: 100.0%"))

            # Обчислення статистики
            elapsed_time = time.time() - start_time
            output_size = os.path.getsize(self.output_file) / (1024
* 1024)
            input_size_mb = file_size / (1024 * 1024)
            speed = input_size_mb / elapsed_time if elapsed_time > 0
else 0
            compression_ratio = file_size /
os.path.getsize(self.output_file) if
os.path.getsize(self.output_file) > 0 else 0
            hash_rate = block_counter / elapsed_time if elapsed_time
> 0 else 0

            # Запис статистики в журнал
            self.log("=" * 60)
            self.log("✓ Processing completed successfully!")
            self.log(f"  Input file size: {input_size_mb:.2f} MB")
            self.log(f"  Output file size: {output_size:.2f} MB")
            self.log(f"  Compression ratio:
{compression_ratio:.2f}:1")
            self.log(f"  Blocks processed: {block_counter:,}")
            self.log(f"  Time elapsed: {elapsed_time:.2f} seconds")
            self.log(f"  Processing speed: {speed:.2f} MB/s")
            self.log(f"  Hash rate: {hash_rate:.0f} hashes/sec")
            self.log("=" * 60)

            # Аналіз статистики вихідного файлу
            self.log("  Analyzing output file statistics...")
            self.after(0, lambda:
self.status_label.configure(text="Analyzing output statistics..."))

            try:
                output_stats =
self.calculate_statistics(self.output_file)

                self.log("  " + "═" * 50)
                self.log(f"  OUTPUT FILE STATISTICS:")
                self.log(f"  Total bits:
{output_stats['total_bits']:,}")
                self.log(f"  Zeros: {output_stats['zeros']:,}
({output_stats['p_zeros']:.4f}%)")
```

```
                self.log(f"  Ones:  {output_stats['ones']:,}
({output_stats['p_ones']:.4f}%)")
                self.log("  " + "—" * 50)
                self.log(f"  Bias: {output_stats['bias']:.6f}
(ideal: 0.000000)")
                self.log(f"  Shannon Entropy:
{output_stats['shannon_entropy']:.6f} bits/bit (ideal: 1.000000)")
                self.log(f"  Min-Entropy:
{output_stats['min_entropy']:.6f} bits/bit (ideal: 1.000000)")
                self.log(f"  Serial Correlation:
{output_stats['serial_correlation']:.6f} (ideal: 0.000000)")
                self.log(f"  Runs: {output_stats['runs_count']:,}
(Z-stat: {output_stats['runs_test_stat']:.4f})")

                # Оцінка якості вихідного файлу
                quality_scores = []

                if output_stats['bias'] < 0.001:
                    bias_quality = "Excellent"
                    quality_scores.append(5)
                elif output_stats['bias'] < 0.005:
                    bias_quality = "Good"
                    quality_scores.append(4)
                elif output_stats['bias'] < 0.01:
                    bias_quality = "Fair"
                    quality_scores.append(3)
                else:
                    bias_quality = "Poor"
                    quality_scores.append(2)

                shannon_dev = abs(1.0 -
output_stats['shannon_entropy'])
                if shannon_dev < 0.001:
                    shannon_quality = "Excellent"
                    quality_scores.append(5)
                elif shannon_dev < 0.005:
                    shannon_quality = "Good"
                    quality_scores.append(4)
                elif shannon_dev < 0.01:
                    shannon_quality = "Fair"
                    quality_scores.append(3)
                else:
                    shannon_quality = "Poor"
                    quality_scores.append(2)

                corr_abs = abs(output_stats['serial_correlation'])
                if corr_abs < 0.01:
                    corr_quality = "Excellent"
                    quality_scores.append(5)
                elif corr_abs < 0.05:
                    corr_quality = "Good"
                    quality_scores.append(4)
                elif corr_abs < 0.1:
```

```
                    corr_quality = "Fair"
                    quality_scores.append(3)
                else:
                    corr_quality = "Poor"
                    quality_scores.append(2)

                avg_quality = sum(quality_scores) /
len(quality_scores)
                if avg_quality >= 4.5:
                    overall_quality = "Excellent"
                elif avg_quality >= 3.5:
                    overall_quality = "Good"
                elif avg_quality >= 2.5:
                    overall_quality = "Fair"
                else:
                    overall_quality = "Poor"

                self.log("  " + "─" * 50)
                self.log(f"  Quality Assessment:")
                self.log(f"    Bias: {bias_quality}")
                self.log(f"    Shannon Entropy: {shannon_quality}")
                self.log(f"    Correlations: {corr_quality}")
                self.log(f"    Overall: {overall_quality}")
                self.log("  " + "═" * 50)

            except Exception as e:
                self.log(f"  ✗ Output statistics calculation failed:
{str(e)}")

            self.after(0, lambda:
self.status_label.configure(text="✓ Completed!"))
            self.after(0, lambda: messagebox.showinfo("Success",
                f"File processed successfully!\n\n"
                f"Output: {output_size:.2f} MB\n"
                f"Time: {elapsed_time:.2f}s\n"
                f"Speed: {speed:.2f} MB/s\n"
                f"Compression: {compression_ratio:.2f}:1"))

        except Exception as e:
            self.log(f"✗ Error: {str(e)}")
            self.after(0, lambda: messagebox.showerror("Error", f"An
error occurred:\n{str(e)}"))
        finally:
            self.processing = False
            self.after(0, lambda:
self.process_button.configure(state="normal", text="▶ Process
File"))
            self.after(0, lambda: self.progress_bar.set(0))

if __name__ == "__main__":
    app = SHA3Extractor()
    app.mainloop()
```

ДОДАТОК Ж

ПЕРЕЛІК ПУБЛІКАЦІЙ

НАЦІОНАЛЬНИЙ АЕРОКОСМІЧНИЙ УНІВЕРСИТЕТ
„ХАРКІВСЬКИЙ АВІАЦІЙНИЙ ІНСТИТУТ"
Кафедра комп'ютерних систем, мереж і кібербезпеки

СТУДЕНТСЬКА КОНФЕРЕНЦІЯ
ІНФОРМАЦІЙНА, ФУНКЦІЙНА
І КІБЕРБЕЗПЕКА
СКІФіК

Матеріали п'ятої
науково-технічної конференції

27-28 листопада 2025 року

ХАРКІВ - 2025

Секція 1

ЗАСТОСУВАННЯ КРИПТОГРАФІЧНОЇ ГЕШ-ФУНКЦІЇ SHA-3 ЯК ЕКСТРАКТОРА ВИПАДКОВОСТІ ДЛЯ QRNG

Волотковський Д. С.
Харківський національний університет ім. В. Н. Каразіна, м. Харків, Україна
Науковий керівник: Нарєжній О. П.

Актуальність. Квантові генератори випадкових чисел (КГВЧ, QRNG) є основою сучасної криптографії, оскільки використовують фундаментальну квантову невизначеність для генерації істинно випадкових послідовностей. Проте, «сирі» (raw) дані, отримані безпосередньо з КГВЧ, неминуче містять статистичні дефекти через вплив класичного шуму, теплові флуктуації детекторів та недосконаліть вимірювальної апаратури. До них належать зміщення (bias) у розподілі ймовірностей бітів та, що більш небезпечно, часові автокореляції – прихована статистична залежність між послідовними значеннями вибірки. Наявність цих дефектів критично знижує реальну непередбачуваність (мін-ентропію) вихідного потоку, створюючи вектори атак для прогнозування ключів шифрування. Це робить обов'язковою імплементацію процедур постобробки – алгоритмічної екстракції випадковості (randomness extraction) на базі універсальних хеш-функцій або екстракторів Тепліца, що дозволяє нівелювати вплив апаратних артефактів та гарантувати криптографічну стійкість генеруємої послідовності відповідно до стандартів NIST SP 800-90B [1].

Метою даної роботи є теоретичне обґрунтування доцільності використання криптографічної геш-функції SHA-3 як обчислювального екстрактора для постобробки «сирих» статистично дефектних даних, отриманих з КГВЧ.

Основні положення. Аналіз простих статистичних екстракторів, що не враховують автокореляції, продемонстрував їхню категоричну непридатність для постобробки квантових даних. Їхня теорія працює лише за критичного припущення, що вхідні біти є незалежними та однаково розподіленими (IID) [2]. «Сирі» дані КГВЧ, що містять автокореляції, прямо порушують це припущення, тому метод не усуває залежності [2]. SHA-3, на відміну від SHA-2 (з конструкцією Меркла-Дамгарда), використовує інноваційну губкову конструкцію (sponge construction). Її внутрішня функція Keccak-f забезпечує потужний лавинний ефект

(avalanche effect), а нелінійний крок χ (Chi) визначений як $a' = a \oplus ((\neg b) \wedge c)$, гарантовано руйнує кореляції. SHA-3 є детерміністичним екстрактором (не вимагає «зерна»), що вирішує «проблему курки та яйця» та надає XOF (SHAKE) для виходу довільної довжини [2,3]. Зокрема, застосування функцій SHAKE128/256 дозволяє гнучко адаптувати пропускну здатність (throughput) постобробки до швидкості фізичного джерела, виконуючи функцію конденсора ентропії. Це забезпечує ефективне згладжування (smoothing) вхідного розподілу з низькою мін-ентропією та стійкість до атак на розрізнення (distinguishing attacks).

Висновки. «Сирі» дані КГВЧ є статистично дефектними через наявність автокореляцій. Прості методи (напр., Фон Неймана) недієві, оскільки базуються на хибному для КГВЧ припущенні IID. Криптографічна геш-функція SHA-3 є прагматичним та надійним інженерним рішенням. Її губкова конструкція та нелінійні властивості гарантовано руйнують складні статистичні залежності, перетворюючи дефектний потік на криптографічно стійку випадковість, що успішно проходить верифікацію за стандартами NIST SP 800-22 та Dieharder..

Список літератури

1. Ma, X., et al. (2013). Postprocessing for quantum random-number generators: Entropy evaluation and randomness extraction. Physical Review A, 87(6). URL: https://arxiv.org/pdf/1207.1473 (дата звернення: 11.11.2025)
2. Vadhan, S. P. (2012). Pseudorandomness: Randomness Extractors. Foundations and Trends® in Theoretical Computer Science, 7(1–3). URL: https://people.seas.harvard.edu/~salil/pseudorandomness/extractors.pdf (дата звернення: 11.11.2025)
3. Dworkin, M. (2015). SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions (NIST FIPS 202). URL: https://nvlpubs.nist.gov/nistpubs/fips/nist.fips.202.pdf (дата звернення: 11.11.2025)

Відомості про авторів

Волотковський Дмитро Сергійович, магістрант кафедри кібербезпеки інформаційних систем, мереж і технологій, ХНУ ім. В. Н. Каразіна, volotkovskyi2020kb12@student.karazin.ua
Нарєжній Олексій Павлович, доцент кафедри кібербезпеки інформаційних систем, мереж і технологій, ХНУ ім. В. Н. Каразіна, к.т.н., o.nariezhnii@karazin.ua

АЛФАВІТНИЙ ВКАЗІВНИК / ALPHABETICAL POINTER