Міністерство освіти і науки України
Харківський національний університет імені В.Н. Каразіна
Навчально-науковий інститут комп'ютерних наук та штучного інтелекту
Спеціальність 125 «Кібербезпека»
Освітня програма «Кібербезпека»

В.о. зав. кафедрою КІСМіТ
Марина ЄСІНА
"Допущено до захисту"

« »_____2025р.

Пояснювальна записка
до кваліфікаційної роботи бакалавра
на тему: «Обґрунтування, вибір та статистичні дослідження випадкових послідовностей на основі нефізичних джерел шуму»

оцінка « _____ »

Голова ЕК
Мичуда Л.З.

Керівник: д.т.н., Горбенко І.Д.

Рецензент: к.т.н. Качко О.

Виконавець: студент групи КБ-42
Гордієнко Д.Я.

Харків 2025

РЕФЕРАТ

Дипломна робота обсягом 53 сторінок, містить 3 ілюстрації, 1 таблицю, 8 додатки. У роботі використано 17 джерел згідно з переліком посилань.

Метою роботи є обґрунтування, вибір та дослідження генераторів випадкових послідовностей, що базуються на нефізичних джерелах шуму, у середовищі операційної системи Linux, з подальшим аналізом їх статистичних характеристик за допомогою стандартних інструментів тестування.

У процесі дослідження застосовано методи математичної статистики, аналіз ентропії, кумулятивних розподілів, кореляцій та стохастичної поведінки послідовностей. Для реалізації експериментальної частини використано апаратно-незалежні програмні засоби: мова програмування C, утиліти Linux (getrandom(), /dev/urandom, OpenSSL), а також пакети аналізу випадковості ENT, Dieharder та NIST STS.

У результаті було створено програмні реалізації трьох генераторів, проведено генерацію даних, організовано статистичне тестування та візуалізацію отриманих результатів. Новизною роботи є комплексне порівняння CSPRNG, побудованих без залучення фізичних джерел ентропії, з фіксацією впливу системної ентропії, часу генерації та якості результатів у контексті сучасних криптографічних вимог.

Рекомендовано використання досліджених генераторів у системах захисту інформації, програмних криптографічних рішеннях, утилітах генерації ключів, системах цифрової ідентифікації, криптовалютних гаманцях та інших безпекових інструментах, де неможливе підключення до фізичних датчиків. Значущість роботи полягає у забезпеченні високого рівня прозорості та довіри до ПЗ-генераторів випадкових послідовностей через тестовану відкриту реалізацію, що є критично важливою умовою для побудови стійких криптосистем у гнучких середовищах.

Подальший розвиток дослідження може бути пов’язаний з: інтеграцією нефізичних генераторів в апаратні криптомодулі; адаптацією методики тестування

до реального часу; дослідженням впливу навантаження на поведінку ентропійного пулу; розширенням до контейнерних та serverless-середовищ.

Ключові слова: ВИПАДКОВІ ПОСЛІДОВНОСТІ, ЕНТРОПІЯ, НЕФІЗИЧНЕ ДЖЕРЕЛО ШУМУ, КРИПТОГРАФІЯ, ГЕНЕРАТОР ВИПАДКОВИХ ЧИСЕЛ, GETRANDOM, DEV URANDOM, OPENSSL, NIST STS, DIEHARDER, ENT.

ABSTRACT

The thesis comprises 53 pages, includes 3 illustrations, 1 table, and 8 appendices. The bibliography contains 17 sources listed in the references section.

The purpose of the thesis is to justify, select, and study random sequence generators based on non-physical sources of noise in the Linux operating system environment, followed by statistical analysis of their properties using standard testing tools.

During the research, methods of mathematical statistics were applied, including entropy analysis, cumulative distributions, correlation estimation, and the stochastic behavior of sequences. The experimental part was implemented using hardware-independent software tools: the C programming language, Linux utilities (getrandom(), /dev/urandom, OpenSSL), and statistical testing packages ENT, Dieharder, and NIST STS.

As a result, software implementations of three generators were developed, random data were generated, and statistical testing along with visualization of the results was conducted. The novelty of the work lies in the comprehensive comparison of CSPRNGs built without physical entropy sources, focusing on the influence of system entropy, generation time, and output quality in the context of modern cryptographic requirements.

The studied generators are recommended for use in information security systems, cryptographic software solutions, key generation utilities, digital identity systems, cryptocurrency wallets, and other security tools where physical entropy sources are unavailable. The significance of this research lies in ensuring a high level of transparency and trust in software-based random number generators through a verifiable open implementation, which is a critical requirement for building robust cryptographic systems in flexible computing environments.

Further development of the research may include: integration of non-physical generators into hardware cryptomodules; adaptation of the testing methodology for real-time evaluation; investigation of entropy pool behavior under load; and extension to containerized and serverless environments.

Keywords: RANDOM SEQUENCES, ENTROPY, NON-PHYSICAL NOISE SOURCE, CRYPTOGRAPHY, RANDOM NUMBER GENERATOR, GETRANDOM, DEV URANDOM, OPENSSL, NIST STS, DIEHARDER, ENT.

ЗМІСТ

ПЕРЕЛІК ПОЗНАЧЕНЬ І СКОРОЧЕНЬ

AES (Advanced Encryption Standard) — симетричний блочний шифр

API (Application Programming Interface – прикладний програмний інтерфейс.

CSPRNG (Cryptographically Secure Pseudorandom Number Generator) – криптографічно безпечний генератор псевдовипадкових чисел.

DRBG (Deterministic Random Bit Generator) – детермінований генератор випадкових бітів.

DRNG (Deterministic Random Number Generator) – детермінований генератор випадкових чисел.

ECDSA (Elliptic Curve Digital Signature Algorithm) – алгоритм цифрового підпису на основі еліптичних кривих.

IV (Initialization Vector) – ініціалізаційний вектор.

OTP (One-Time Password) – одноразовий пароль.

PRNG (Pseudorandom Number Generator) – псевдовипадковий генератор чисел.

RNG / ГВП – Random Number Generator / генератор випадкових чисел.

SHA (Secure Hash Algorithm) – криптографічні хеш-функції.

SSH (Secure Shell) протокол безпечного дистанційного доступу.

TLS (Transport Layer Security) – протокол захищеної передачі даних.

TRNG – True Random Number Generator – істинний генератор випадкових чисел.

VPN (Virtual Private Network) – віртуальна приватна мережа.

ВСТУП

У сучасному світі забезпечення надійного захисту інформації стало стратегічним пріоритетом для державних, комерційних і приватних структур. Зі зростанням обсягу цифрових даних, хмарних сервісів, криптовалют, IoT-пристроїв та телекомунікацій, важливість криптографічного захисту суттєво зросла. Однією з ключових складових криптографії є генератори випадкових послідовностей (ГВП), що забезпечують формування криптографічних ключів, сольових значень, ініціалізаційних векторів та інших критично важливих даних, які мають бути непередбачуваними та статистично незалежними.

Аналіз науково-технічної літератури, а також патентних ресурсів свідчить про те, що генерація випадкових чисел і досі залишається об’єктом активних досліджень. В основі сучасних стандартів (NIST SP 800-22, SP 800-90A/B/C, BSI AIS 20/31, ISO/IEC 18031) лежать вимоги до криптографічної стійкості генераторів, оцінки ентропії, передбачуваності та захисту від атак типу prediction, backtracking або reseed poisoning. У світі провідними фахівцями в галузі є Брюс Шнайєр, Даніель Бернштейн, Томас Порст, а серед компаній — Google, Intel, IBM, Cloudflare, які активно розробляють та впроваджують власні генератори, в тому числі OpenSSL DRBG, BoringSSL, Fortuna, та ChaCha20-based DRBG.

Раніше було сформовано значну кількість публікацій, проте більшість із них акцентують увагу на апаратних (фізичних) генераторах, або ж стандартизованих реалізаціях, залишаючи поза увагою нефізичні джерела шуму, які все частіше використовуються в сучасних програмних середовищах. Практично вирішеними на сьогодні є питання реалізації генераторів у формі апаратних модулів (TPM, HWRNG, Intel RDRAND), а також забезпечення інтеграції PRNG у стандартні криптобібліотеки. Однак існує низка прогалин: відсутність єдиної методики оцінки програмних (нефізичних) генераторів; недостатня кількість порівняльних досліджень різних реалізацій у Linux-середовищі; слабка прозорість механізмів наповнення ентропійного пулу ОС; обмежений вплив користувача на контроль якості генерації. Ці фактори роблять необхідним додаткове дослідження та моделювання властивостей послідовностей, згенерованих без участі фізичних

джерел. Найбільший інтерес для прикладного аналізу становлять програмні реалізації генераторів, що входять до складу ОС Linux: /dev/urandom, системний виклик getrandom(), бібліотека OpenSSL (RAND_bytes(), DRBG_new()). Ці механізми реалізують псевдовипадкову генерацію, підкріплену накопиченням ентропії з системних джерел (таймери, перемикання процесів, мережеві події тощо), і становлять собою гібридні CSPRNG. Їхня ефективність та безпека залежать від якості початкової ентропії, коректності ініціалізації, наявності періодичного reseeding та криптографічних алгоритмів постобробки.

Актуальність даної роботи обумовлена стрімким розвитком технологій інформаційної безпеки, зростанням кількості кіберзагроз, необхідністю забезпечення стійкої криптографічної генерації навіть у віртуалізованих середовищах та відсутністю апаратних джерел шуму. Особливої важливості це набуває в умовах використання Linux-серверів, контейнерних систем, хмарних платформ та мобільних пристроїв, де програмні механізми генерації часто є єдиним джерелом випадковості.

Метою роботи є обґрунтування, вибір та дослідження генераторів випадкових послідовностей, що ґрунтуються на нефізичних джерелах шуму, у контексті ОС Linux, а також проведення їх комплексного статистичного аналізу за допомогою інструментів ENT, Dieharder та NIST STS.

Результати дослідження можуть бути застосовані: у проєктуванні криптографічних програмних бібліотек; у системах захисту каналів зв’язку (TLS, VPN, SSH); у генерації одноразових паролів, ключів, сольових значень; у криптовалютній інфраструктурі (seed-фрази, гаманці); у розробці незалежних джерел ентропії для віртуальних середовищ.

Дослідження, представлене в цій роботі, продовжує сучасні підходи до оцінки генераторів випадкових послідовностей, описані в міжнародних документах (SP 800-22, RFC 4086, BSI AIS 31), а також базується на відкритих реалізаціях генераторів у середовищі Linux.

1 ТЕОРЕТИЧНІ ОСНОВИ ВИПАДКОВИХ ПОСЛІДОВНОСТЕЙ

1.1. Поняття випадковості в криптографії

У сучасних інформаційних технологій випадковість є одним із головних елементів, що забезпечує надійність криптографічних систем. Випадкові послідовності використовуються в критично важливих криптографічних задачах, таких як генерація ключів шифрування, створення OTP, salt-значень для хешування, ініціалізаційних векторів (IV), nonce та інших елементів, від яких напряму залежить стійкість захисту даних.

Під випадковістю у криптографії розуміється не просто нерегулярність чи нестабільність, а перш за все непередбачуваність результату, навіть при повному знанні алгоритму та частини послідовності. Це принципово важливо: якщо зловмисник здатен спрогнозувати наступний елемент на основі попередніх — вся система втрачає свою захищеність. Слід відмежовувати алгоритмічну (псевдовипадкову) випадковість, що застосовується у статистичних або ігрових задачах, від криптографічної випадковості, яка повинна відповідати суворим вимогам безпеки. Псевдовипадкові генератори, такі як стандартний rand() у C, є детермінованими — тобто при однаковому початковому значенні (seed) вони завжди породжують ту саму послідовність. Це прийнятно у моделюванні, але абсолютно неприйнятно в криптографії.

Натомість криптографічні ГВП (CSPRNG — cryptographically secure pseudo-random number generators) забезпечують високу ентропію, незворотність і неможливість відновлення внутрішнього стану генератора навіть при відомих вихідних даних. Тобто навіть якщо атакуючий знає частину згенерованих чисел, він не зможе обчислити решту або вгадати майбутні значення. Відомими прикладами CSPRNG є ChaCha20, AES-CTR DRBG, Fortuna, а також реалізації у бібліотеках OpenSSL, Libgcrypt або системні механізми на кшталт /dev/urandom та getrandom() у Linux. Випадковість характеризується кількома ключовими властивостями. Перш за все — ентропією: чим вона вища, тим більше "інформаційної невизначеності" несе кожен елемент послідовності. Для байта

максимально можлива ентропія — 8 біт. Також важливими є непередбачуваність (неможливість спрогнозувати наступне значення), незалежність (кожен елемент не залежить від попередніх), нестисліть (послідовність не може бути скорочена без втрати інформації) та відсутність закономірностей, які б дозволили провести статистичний аналіз і "зламати" систему. У практиці інформаційної безпеки якість випадковості безпосередньо впливає на захищеність усієї системи. Наприклад, при створенні ключа шифрування довжиною 256 біт, який теоретично має 2^{256} можливих варіантів, використання генератора з низькою ентропією може фактично обмежити цей простір до значно меншого числа, що дозволить атакуючому застосувати перебір або побудувати словник для атаки.

Таким чином, поняття випадковості у криптографії — це не лише питання теорії, а фундаментальний практичний аспект, від якого залежить реальна стійкість протоколів, алгоритмів та цілих інформаційних систем. Забезпечення правильної випадковості є однією з головних умов побудови надійної криптографії.

1.2. Різновиди випадкових послідовностей

Різноманіття випадкових послідовностей зумовлене як відмінностями у природі їх походження, так і специфікою призначення в інформаційних системах. У цьому контексті важливо розуміти, з яким саме типом випадковості ми маємо справу, оскільки не всі генератори забезпечують достатній рівень захисту від криптоаналітичних атак. Хоча зовні різні види випадкових послідовностей можуть виглядати подібно, їхні математичні, ентропійні та криптографічні властивості суттєво відрізняються.

Найбільш фундаментальним видом є істинні випадкові послідовності, що генеруються на основі фізичних процесів, які є принципово непередбачуваними. Прикладами таких процесів можуть бути квантові флуктуації, тепловий шум у резисторах, фотоелектричний ефект, шум у напівпровідниках або розпад радіоактивних ізотопів. Усі ці явища мають високий ступінь ентропії і не піддаються моделюванню навіть за допомогою найточніших алгоритмів. Такі послідовності зазвичай використовуються в апаратних ГВП, вбудованих у криптографічні пристрої, смарт-карти або безпекові модулі. Їх основною

перевагою є абсолютна непередбачуваність, проте вони залежать від фізичних умов, можуть бути повільними та вимагати спеціального обладнання.

На противагу їм, псевдовипадкові послідовності створюються повністю програмно, без залучення фізичних джерел. Вони формуються за допомогою алгоритмів, які починають обчислення з визначеного початкового значення, що називається seed (зерно). Це означає, що за однакових умов — однаковому алгоритмі й одному й тому самому seed — результат завжди буде однаковим. З математичної точки зору, такі послідовності є цілком детермінованими, а отже — передбачуваними. Незважаючи на те, що вони можуть демонструвати хороші статистичні властивості (рівномірність, відсутність шаблонів), їх використання в криптографії є вкрай обмеженим через можливість відновлення seed або стану генератора.

Третім важливим типом є криптографічні псевдовипадкові послідовності. Вони побудовані за принципами, що унеможливлюють відновлення внутрішнього стану генератора навіть при частковому витоку вихідних даних. Такі генератори називаються CSPRNG. Їх алгоритми ґрунтуються на використанні криптографічних примітивів, як-от симетричні блочні шифри у режимі лічильника (AES-CTR), криптографічні хеш-функції (SHA-256, SHA-3), або потокові шифри (ChaCha20). На відміну від звичайних PRNG, криптографічні генератори гарантують стійкість до атаки навіть у разі аналізу частини вихідних даних, оскільки не дозволяють відновити seed або предиктувати наступні значення. Саме тому вони використовуються в більшості сучасних протоколів безпеки — TLS, SSH, IPsec, GPG, а також в операційних системах: зокрема, в Linux доступ до CSPRNG надається через механізми /dev/urandom, getrandom() або криптографічні бібліотеки типу OpenSSL. Окрему увагу заслуговує особливий клас генераторів, що набули популярності завдяки розвитку операційних систем і аналізу поведінки комп’ютерного середовища — це так звані нефізичні істинні генератори випадкових послідовностей. Вони базуються не на апаратних датчиках, а на складній, варіативній взаємодії компонентів цифрової системи. Джерелами ентропії в таких системах можуть бути час доступу до пам’яті, перемикання процесів ядра, події введення-виведення, мережеві затримки, взаємодія

користувача з пристроєм. Усі ці процеси надзвичайно складно моделювати чи повторити, тому вони можуть виступати як ефективна альтернатива фізичним генераторам.

Згідно з класифікацією стандарту AIS 20/31, подібні підходи вважаються прийнятними, за умови використання правильних механізмів збирання та підсилення ентропії, а також проведення статистичних перевірок. Кожен з розглянутих типів має своє призначення. Фізичні генератори забезпечують абсолютну непередбачуваність, але потребують ресурсів. Псевдовипадкові — швидкі, однак непридатні для захисту, вони поєднують в собі алгоритмічну ефективність та криптографічну надійність, тоді як нефізичні істинні генератори виступають розумним компромісом, дозволяючи використовувати стандартне обладнання без втрати безпеки. Розуміння цих відмінностей є ключовим для правильного вибору генератора відповідно до вимог конкретної системи інформаційної безпеки.

1.3. Призначення та застосування випадкових послідовностей у сфері інформаційної безпеки

Випадкові послідовності є однією з ключових складових будь-якої криптографічної системи, оскільки від їхньої якості безпосередньо залежить надійність алгоритмів захисту даних. Уся сучасна криптографія, як симетрична, так і асиметрична, базується на припущенні, що окремі елементи, з яких формується безпекова структура (наприклад, ключі, ініціалізаційні вектори, salt), не повинні мати визначеного порядку або підлягати передбачуваності. Однією з найважливіших сфер застосування випадкових послідовностей є генерація криптографічних ключів. Саме ключ — це єдина частина в алгоритмі шифрування, яка повинна залишатися в таємниці. Якщо послідовність, на основі якої сформовано ключ, є передбачуваною, навіть найсильніший шифр може бути легко зламаним. Більше того, якщо два ключі будуть сформовані з однакової послідовності або з близькою ентропійною структурою, ймовірність успішної атаки методом повного перебору або побудови словника (dictionary attack) значно зростає. Саме тому криптографічні стандарти, такі як NIST SP 800-133, жорстко

регламентують вимоги до генерації ключів і джерел випадковості, що при цьому використовуються. Ще однією важливою областю застосування є ініціалізаційні вектори (IV) — додаткові параметри, які використовуються разом з ключами у багатьох режимах блочного шифрування, таких як CBC, CFB, OFB. IV має бути унікальним для кожного повідомлення, щоб однакові блоки відкритого тексту шифрувались по-різному. Якщо IV обрано нерандомізовано або передбачувано, це дозволяє атакуючому ідентифікувати структуру даних або навіть провести відновлення частини повідомлення. Крім того, salt, які додаються до паролів перед хешуванням, служать для того, щоб захистити хеші від атак із попередньо побудованими таблицями (так званими rainbow tables). Якщо salt є передбачуваним або повторним, його використання стає марним — злочинець може атакувати не лише окремого користувача, а цілу базу даних одночасно. Якісна випадкова послідовність для salt робить кожен запис унікальним і значно ускладнює масові атаки. У сфері OTP або токенів автентифікації також критично важливо використовувати генератори з високим ступенем ентропії. Наприклад, у протоколах двофакторної автентифікації або в механізмах обмеженого доступу до API (JSON Web Tokens, OAuth) такі послідовності не тільки визначають дозвіл, але й фіксують час дії, тип ресурсу і прив’язку до конкретного користувача. Будь-яка вразливість у генераторі може призвести до несанкціонованого доступу. Не менш важливим напрямом є формування параметрів у криптографічних протоколах обміну ключами, таких як Diffie–Hellman (DH), Elliptic Curve Diffie–Hellman (ECDH), а також формування випадкових елементів у цифрових підписах (наприклад, у DSA чи ECDSA). Усі ці механізми потребують генерації сильних, незалежних випадкових значень, які не повинні повторюватися або бути частково передбачуваними. У 2010-х роках зафіксовано низку успішних атак, коли повторне використання випадкових значень у цифровому підписі призводило до повного розкриття приватного ключа. Також у багатьох сучасних програмних засобах захисту (BitLocker, VeraCrypt, GnuPG, OpenVPN) випадкові послідовності використовуються не лише при створенні ключів, але й під час шифрування файлів, створення криптографічних контейнерів, генерування seed-фраз для гаманців у

блокчейнах. У цих випадках неякісна випадковість не просто знижує рівень захисту — вона робить систему вразливою до повного компрометації.

1.4. Основні властивості криптографічно стійких випадкових послідовностей

Для того щоб випадкова послідовність могла вважатися придатною до використання у криптографії, вона повинна відповідати ряду строгих властивостей. На відміну від загального статистичного поняття «випадковість», у сфері інформаційної безпеки ключову роль відіграє не лише зовнішня нерегулярність, а й стійкість до предиктивного та аналітичного впливу з боку зловмисника. Іншими словами, криптографічно стійка випадкова послідовність — це така, яка не лише виглядає випадковою, а й не дозволяє здійснити криптоаналіз, навіть якщо частина її вже відома.

Першою і, безумовно, базовою характеристикою криптографічно стійких випадкових послідовностей є ентропія, яка визначає рівень випадковості або, з точки зору інформаційної теорії, кількість інформації, що міститься в кожному біті або байті даних. Це поняття введене Клодом Шенноном і використовується для оцінки невизначеності джерела інформації. У прикладному аспекті ентропія дозволяє кількісно оцінити, наскільки послідовність є «важкою для передбачення» або відновлення стороннім спостерігачем. Чим вищий рівень ентропії, тим більше ймовірність, що кожен елемент послідовності є незалежним і рівноймовірним. У випадку з байтами (8 бітів), максимальна теоретична ентропія становить 8 бітів на байт, що відповідає ситуації, в якій кожне з 256 можливих значень має ймовірність 1/256. У такому випадку жодне значення не є пріоритетним, а будь-яке прогнозування стає марним без повного перебору. Насправді, у багатьох практичних реалізаціях (особливо на початкових етапах генерації) спостерігається зниження ентропії, що є серйозним ризиком для безпеки. Наприклад, у системах, які не мають достатнього доступу до ентропійного джерела (embedded-системи, віртуальні середовища, контейнерні середовища), генератори можуть почати видавати значення ще до того, як накопичено достатньо ентропії. Це призводить до ситуації, коли ключі чи nonce значення є недостатньо випадковими, що в свою

чергу відкриває шлях для атак типу brute-force, offline dictionary attacks або even precomputation attacks. Зниження ентропії на один біт означає вдвічі менший простір можливих значень. Наприклад, для 256-бітного ключа AES при ефективній ентропії в 240 бітів, атака методом повного перебору теоретично потребує лише 2^{16} = 65 536 разів менше зусиль. Хоча це може виглядати незначним з погляду абстрактної криптографії, на практиці така різниця може мати критичне значення при атаках із використанням сучасних обчислювальних кластерів або квантових алгоритмів у майбутньому. Оцінювання ентропії є ключовим етапом сертифікації криптографічних засобів захисту. У документах NIST SP 800-90B та AIS 31 (BSI) прописані мінімальні допустимі значення ентропії для генераторів у залежності від їх призначення. Наприклад, безпечний генератор повинен довести, що кожен біт його виходу має ентропію не менше ніж 0.999. Також передбачається, що процес накопичення ентропії контролюється і реєструється — для цього використовують механізми, як-от entropy_avail у Linux або Entropy Estimation Function (EEF) у сертифікованих криптографічних бібліотеках. На практиці значення ентропії перевіряється за допомогою таких тестів, як Chi-square, Shannon entropy, Compression tests, Markov models та інших. Якщо послідовність стискається (наприклад, за допомогою ZIP або LZMA), це означає, що вона має певну структуру, отже — знижену ентропію. Це є неприйнятним для криптографічного використання. Таким чином, висока ентропія є основоположною умовою для надійної криптографії. Вона гарантує, що згенеровані ключі, вектори ініціалізації, сесійні маркери або сольові значення не можуть бути відтворені чи передбачені навіть у випадку, якщо зловмисник має частковий доступ до системи. Саме тому криптографічні бібліотеки реалізують контроль за рівнем ентропії і не допускають генерації без його досягнення.

Наступною важливою властивістю криптографічно стійких послідовностей є непередбачуваність (англ. unpredictability). Це поняття означає, що навіть за умови повного знання вже згенерованої частини послідовності, а також часткового або навіть повного доступу до зовнішніх параметрів, які використовувалися під час генерації (наприклад, поточний час, PID процесу, мережеві події), не повинно існувати жодного ефективного способу передбачити наступний елемент із вищою

ймовірністю, ніж випадкове вгадування. Ця властивість є важливою оскільки відсутність передбачуваності гарантує стійкість протоколів до атак на випадковість — зокрема forward prediction та state compromise extensions. У разі використання звичайних PRNG, які не мають криптографічної стійкості, зловмисник може за певних умов відновити внутрішній стан генератора. Наприклад, якщо генератор базується на лінійній конгруентній послідовності або має недостатню складність алгоритму, знання кількох перших елементів може дозволити точно вирахувати наступні значення. Це стало причиною низки реальних атак: наприклад, у 2008 році вразливість у ГВП OpenSSL у Debian дозволила зловмисникам відтворити всі можливі SSH-ключі, згенеровані впродовж двох років, що спричинило масштабний перегляд політик безпеки. На відміну від цього, генератори CSPRNG розроблені спеціально з урахуванням необхідності забезпечення непередбачуваності. Їхня архітектура зазвичай включає внутрішній стан (state), який регулярно оновлюється, а вихідні значення формуються за допомогою хеш-функцій (SHA-2, SHA-3), блочних шифрів (AES у режимі CTR) або арифметики еліптичних кривих, що робить неможливим зворотне обчислення навіть за умови доступу до великих обсягів вихідних даних. Більше того, сучасні CSPRNG реалізують властивості forward secrecy (стійкість до передбачення наступних значень) і backtracking resistance (стійкість до відновлення попередніх значень). Це означає, що навіть якщо зловмиснику вдасться скомпрометувати стан генератора у певний момент часу, він не зможе отримати інформацію про послідовності, згенеровані до цього моменту (retrospective attack), і не матиме змоги визначити подальші значення, оскільки після кожного запиту внутрішній стан оновлюється криптографічно стійким способом. Варто зазначити, що непередбачуваність є вимогою не лише до вихідного потоку, а й до джерел, які використовуються для первинної ініціалізації генератора (seed entropy). Якщо для ініціалізації використано дані з обмеженою ентропією (наприклад, системний час з точністю до секунд), можливе формування передбачуваного seed-стану, що знищує всю криптографічну стійкість. Саме тому всі сертифіковані генератори (зокрема, відповідно до NIST SP 800-90A) мають модуль збору ентропії (entropy collector) і обов’язковий механізм reseeding — періодичного або подієвого оновлення внутрішнього стану генератора. Також

критичною властивістю криптографічно стійких випадкових послідовностей є незалежність елементів (statistical independence). Це означає, що значення кожного біта або байта у послідовності не повинно залежати від значень попередніх або наступних елементів. Іншими словами, навіть знаючи частину послідовності, зловмисник не повинен мати змоги зробити припущення або розрахунок щодо решти значень з точністю вищою, ніж випадкове вгадування. У математичному сенсі незалежність визначається відсутністю статистичної кореляції (linearity, bias, autocorrelation) між елементами. Якщо така кореляція присутня, це свідчить про детермінованість або структурність вихідного потоку, що прямо знижує ефективну ентропію — тобто кількість дійсно непередбачуваної інформації в потоці. Внаслідок цього простір перебору, який повинен бути експоненційним, фактично зменшується, і криптографічний захист стає вразливим до атак з побудовою шаблонів. На практиці навіть слабкі статистичні залежності між елементами можуть мати катастрофічні наслідки. Наприклад, у протоколах електронного підпису (ECDSA, EdDSA), де використовуються одноразові випадкові значення (nonce), залежність між кількома nonce може дозволити зловмиснику повністю відновити приватний ключ. Такі вразливості були зафіксовані у перших реалізаціях криптовалютних гаманців та навіть у Bitcoin-мережі. Подібно, у протоколах обміну ключами (Diffie–Hellman, IKE), кореляція між випадковими числами зменшує надійність процесу узгодження загального секрету. Також незалежність критична для ініціалізації seed-значень у псевдовипадкових генераторах. Якщо генератор ініціалізовано із використанням значень, що містять взаємозалежні біти (наприклад, ентропійний пул зі структурованим джерелом), це може створити циклічні або повторювані шаблони, які легко виявити за допомогою статистичних тестів, і які суттєво спрощують аналіз потоку. Для оцінки незалежності використовуються різні статистичні методи:

- Autocorrelation Test — перевіряє наявність залежностей між значеннями на фіксованій відстані.
- Serial Correlation Coefficient — оцінює загальну кореляцію між сусідніми байтами.
- Approximate Entropy — виявляє повторюваність шаблонів.

- Runs Test — аналізує довжини однакових підрядків (серій), що вказують на потенційні залежності.
- Spectral Test — використовується для виявлення частотних закономірностей у випадкових даних.

Якісні CSPRNG повинні гарантувати відсутність виявлених шаблонів у вихідному потоці, навіть на глибокому рівні аналізу, що вимагає використання як побітових, так і побайтових статистичних тестів. Наприклад, NIST STS та Dieharder включають ці тести у свої пакети, і невдале проходження хоча б одного з них — сигнал про наявність структурності, а отже — про знижену незалежність. Окремо слід виділити ще одну важливу властивість криптографічно стійких випадкових послідовностей — нестисліть (incompressibility). Вона полягає у тому, що така послідовність не повинна містити жодних шаблонів, повторень, регулярностей або структур, які дозволяють її стискати будь-яким алгоритмічно ефективним методом без втрати інформації. Інакше кажучи, її алгоритмічна складність має бути максимально високою, а саму послідовність неможливо описати коротшим способом, ніж вона є насправді. У рамках теорії алгоритмічної інформації, така властивість асоціюється з поняттям випадковості за Колмогоровим — послідовність вважається випадковою тоді і лише тоді, коли її найкоротша програма, що породжує її, має довжину, не меншу за саму послідовність. Це означає, що жоден компресор (навіть ідеальний) не зможе створити компактніший варіант цієї послідовності без втрати даних. З практичної точки зору, нестисліть є прямим індикатором високої ентропії, незалежності та непередбачуваності. Якщо послідовність легко стискається — наприклад, шляхом ZIP, LZ77 або Huffman-кодування, — це означає, що вона містить закономірності або повторювані фрагменти, які можна описати коротшими структурами. У криптографії це є небажаним і небезпечним явищем, оскільки зловмисник може використати ці структури для зменшення обчислювальної складності атак, зокрема при побудові dictionary або pattern-based атак. Класичний приклад — генерація ключів або nonce, які мають певний предикативний шаблон. Наприклад, ключ у вигляді «ABCABCABC...» з великою ймовірністю буде стиснутий ідентифікованим шаблоном, що автоматично зменшить ентропію і дозволить звузити простір

перебору. У процесі тестування генераторів нестисліcть можна оцінити за допомогою:

- Compression ratio — перевірка, чи стискається послідовність за допомогою LZMA, GZIP, BZIP2. Ідеально випадкові дані не повинні стискатись взагалі або з коефіцієнтом, близьким до 1.0.
- Kolmogorov complexity estimators — алгоритмічні оцінки складності послідовності через ентропійні моделі, наприклад, PPM (Prediction by Partial Matching).
- ENT compressibility test — один з простих індикаторів випадковості, що визначає, чи можлива компресія без втрати інформації.

Нестисліcть також важлива для симуляційних систем, шифрування потокових даних, генерації випадкових токенів — у всіх випадках, де важлива гарантія, що послідовність не містить прогнозованої структури. Варто зазначити, що сучасні CSPRNG, такі як OpenSSL DRBG або генератори, що базуються на AES-CTR, формально мають властивість нестислості завдяки своїй криптографічній конструкції.

Ще однією важливою властивістю криптографічно надійних випадкових послідовностей є відсутність повторів або періодичності. Це означає, що навіть при тривалому або безперервному функціонуванні генератора, його вихідна послідовність не повинна мати обмежений період, тобто не повинна повертатися до вже згенерованих раніше значень. Повторення в цьому контексті є ознакою внутрішньої структури генератора, яка дозволяє передбачити його поведінку, і, як наслідок, порушує фундаментальні принципи безпеки в криптографії. YTRNG або правильно реалізованих генераторів CSPRNG, таких як ті, що ґрунтуються на AES-CTR або HMAC-DRBG, періодичність або не виникає взагалі, або її довжина є настільки великою (наприклад, 2^{256} або більше), що ймовірність досягнення повтору за весь час існування Всесвіту практично дорівнює нулю. У таких генераторах навіть після трильйонів запитів до джерела випадковості наступне значення буде статистично новим і незалежним. На противагу цьому, звичайні PRNG, особливо ті, що не розроблені для криптографічного застосування (наприклад, Linear Congruential Generators, Mersenne Twister тощо), мають

визначений період, що часто прямо закладений у їхню внутрішню структуру. Наприклад, Mersenne Twister має період $2^{19937}-1$, що виглядає вражаюче, однак відсутність криптографічної стійкості та виявлені шаблони роблять його непридатним для захисту даних. Крім того, такі генератори можуть почати генерувати повторювані патерни при неправильному ініціалізуванні або після вичерпання внутрішнього стану. Періодичність — це не лише теоретична проблема, а й реальна загроза для безперервних або довготривалих криптографічних процесів. Наприклад, у VPN-сесіях, довготривалих TLS-з'єднаннях, потоковому шифруванні або генерації великої кількості токенів, повторення значень (ключів, nonce, векторів ініціалізації) може призвести до: повного розкриття шифрованого вмісту (наприклад, при повторному IV у режимі CBC); компрометації підписаних повідомлень (при повторенні nonce в ECDSA); можливості відтворити або підробити автентифікацію. У зв'язку з цим, генератори, які використовуються в криптографії, повинні реалізовувати вбудовані механізми оновлення стану (reseeding, rekeying), що не лише збільшують ефективний період, але й гарантують постійне оновлення внутрішніх параметрів — навіть у випадку потенційної компрометації частини стану. Оцінити наявність або відсутність повторів можна за допомогою:

- Birthday spacing tests (тест на повторення),
- Serial tests на шаблонність,
- Longest run tests (тривалість найбільшої повторюваної послідовності),
- Histogram analysis для виявлення переважання певних шаблонів.

Відсутність повторів — це не просто теоретична властивість, а критичний елемент реальної безпеки криптографічних рішень.

Окрім зазначених властивостей, важливим аспектом є стійкість генератора до компрометації — тобто здатність зберігати безпечність навіть у разі часткового витоку внутрішнього стану або вихідних даних. У криптографії це поняття часто формалізується через дві ключові властивості: forward secrecy (стійкість до передбачення майбутнього) та backward secrecy (стійкість до відновлення минулого). Генератор повинен реалізовувати обидва механізми захисту, тим самим унеможливлюючи відновлення як попередніх, так і майбутніх елементів

послідовності, навіть якщо зловмиснику вдалось отримати частину її значень. Forward secrecy (приватність у майбутньому) означає, що навіть якщо зловмисник повністю або частково отримає значення виходу генератора у певний момент часу, він не зможе передбачити наступні значення, оскільки внутрішній стан генератора оновлюється кожного разу за допомогою невідворотних криптографічних перетворень. У криптографічній реалізації це забезпечується, наприклад, шляхом оновлення ключів або зміни сеансових параметрів після кожного запиту, що є характерним для DRBG-механізмів згідно з NIST SP 800-90A. Backward secrecy (приватність у минулому), своєю чергою, гарантує, що компрометація поточного стану генератора не дозволить зловмиснику відновити попередньо згенеровані значення. Це досягається за рахунок використання нелінійних односторонніх функцій (хешів, блокових шифрів у певних режимах), які унеможливлюють інверсію — тобто повернення до попереднього стану навіть при повному доступі до теперішнього. CSPRNG реалізують ці властивості через постійне оновлення внутрішнього стану, так зване reseeding — періодичне або подієве (при кожному запиті) внесення нової ентропії з внутрішніх або зовнішніх джерел. Наприклад, OpenSSL DRBG за замовчуванням оновлює внутрішній стан після кожного виклику RAND_bytes() і кожних 2^{48} згенерованих бітів, а також дозволяє ввести додаткову ентропію вручну. Аналогічно, в інтерфейсі getrandom() реалізовано блокування генерації до моменту, коли ентропійний пул буде заповнений достатньо для безпечної ініціалізації. Недостатня реалізація цих властивостей спостерігалася в низці інцидентів безпеки. Зокрема, у 2012 році вразливість у генераторі OpenSSL у Debian дозволила зловмисникам відтворити тисячі приватних SSH-ключів, оскільки генератор не виконував оновлення стану після кожного запиту, і початкове значення було передбачуваним.

Криптографічно стійка випадкова послідовність — це не просто набір байтів, що виглядає хаотично, а результат складного, строго контрольованого процесу генерації, що забезпечує максимальний рівень безпеки навіть в умовах активного супротиву або атак. Її властивості не повинні лише задовольняти математичні критерії, а й відповідати сучасним стандартам криптографії, зокрема NIST, ISO, BSI та рекомендаціям OWASP.

2 АНАЛІЗ НЕФІЗИЧНИХ ДЖЕРЕЛ ШУМУ ДЛЯ ГЕНЕРАЦІЇ ВИПАДКОВИХ ПОСЛІДОВНОСТЕЙ

2.1. Класифікація генераторів випадкових послідовностей згідно з AIS 20 та AIS 31

Важливо розуміти класифікацію, властивості та ступінь надійності ГВП. Одним із найавторитетніших джерел у цій сфері є стандарти Німецького федерального відомства з інформаційної безпеки (BSI), зокрема документи AIS 20 та AIS 31. Вони забезпечують формалізований підхід до аналізу, тестування та сертифікації ГВП і широко застосовуються в європейській практиці сертифікації програмно-апаратних засобів захисту інформації за схемою Common Criteria. Основна цінність цих стандартів полягає в тому, що вони враховують як криптографічні вимоги, так і ентропійні аспекти генерації, визначаючи при цьому класи, до яких можна віднести той чи інший генератор залежно від його будови, джерела ентропії та властивостей вихідної послідовності. Згідно з AIS 20/31, усі ГВП поділяються на дві фундаментальні категорії:

- істинні генератори – TRNG;
- детерміновані генератори – DRNG.

Істинні генератори отримують ентропію безпосередньо з фізичних процесів, таких як тепловий шум, флуктуації напруги, фонове радіовипромінювання, або поведінка електронних компонентів у режимах шуму. Головною характеристикою таких генераторів є повна непередбачуваність вихідних значень, навіть за повного знання початкових умов чи алгоритмічної частини. Проте TRNG часто мають низьку пропускну здатність, є залежними від апаратного забезпечення, а також вимагають постійного моніторингу стабільності джерела шуму, щоб запобігти деградації. Тому стандарт AIS 31 встановлює обов’язкову вимогу до наявності online health tests – спеціальних процедур, які перевіряють якість джерела, виявляючи повторення, сталість або втрату ентропії. AIS 31 розрізняє два підкласи TRNG:

- PTG.1 — фізичний генератор без криптографічної постобробки;

- PTG.2 — фізичний генератор з криптографічною постобробкою (наприклад, із використанням SHA або AES).

Детерміновані генератори працюють на основі математичних алгоритмів, які формують послідовність значень з початкового параметра — seed. Вони відтворювані, а отже не є випадковими в абсолютному сенсі. Однак при правильній реалізації, із залученням достатньої ентропії та криптографічної обробки, такі генератори можуть забезпечити високу швидкість, стійкість та надійність. AIS 20 визначає вимоги до CSPRNG, які повинні реалізовувати:

- forward secrecy — неможливість виведення майбутніх значень навіть у разі часткового доступу;
- backtracking resistance — неможливість відновлення попередніх значень після компрометації.

Для досягнення цих властивостей у CSPRNG використовуються хеш-функції, блочні або потокові шифри. Такі генератори сертифікуються як:

- NTG.1 — програмний генератор без доведеної криптографічної стійкості;
- NTG.2 — криптографічно безпечний генератор, реалізований згідно з AIS 20 (наприклад, AES-CTR DRBG, ChaCha20, HMAC_DRBG).

Окрему увагу в AIS 31 приділено так званим гібридним генераторам, які поєднують елементи TRNG та DRNG. Вони містять ентропійне ядро, що постійно або періодично підживлює псевдовипадкову частину новими значеннями, зберігаючи баланс між продуктивністю і надійністю. Такий підхід рекомендовано для використання в програмному середовищі, наприклад, в ОС Linux, де апаратні джерела ентропії часто недоступні. До таких рішень належать генератори getrandom(), /dev/urandom, OpenSSL DRBG, які реалізують потокову обробку ентропії з частковою криптографічною постобробкою і можуть відповідати класу NTG.2.

Також стандарт дозволяє сертифікацію генераторів, що базуються на нефізичних джерелах шуму: системних таймерах, активності користувача, процесорних подіях, мережевому трафіку тощо. Хоча такі генератори не є TRNG у класичному розумінні, при належному тестуванні (статистичному й health-тестах)

вони можуть бути визнані еквівалентно безпечними. В порівняльній Таблиці 2.1 можемо бачити порівняльний аналіз цих ГВП згідно класифікації стандарту AIS.

Таблиця 2.1 – Порівняльна таблиця ГВП: фізичного (PTG) та програмного (NTG)

Критерій	PTG (TRNG)	NTG.2 (CSPRNG)
Джерело ентропії	Фізичне	Нефізичне або вбудоване seed
Швидкість генерації	Повільна	Висока
Потреба у health-тестах	Так	Не завжди
Наявність постобробки	Бажано (PTG.2)	Обов’язкова
Стійкість до прогнозування	Висока	Висока (за правильної реалізації)
Залежність від платформи	Висока	Низька
Приклади	Quantis, TPM, ChaosKey	OpenSSL DRBG, getrandom(), urandom

Класифікація, запропонована в AIS 20/31, не лише описує типи генераторів, але й встановлює чіткі критерії їхньої відповідності до рівнів криптографічної стійкості. Ці стандарти використовуються як базова основа для оцінювання, проектування та сертифікації генераторів у критичних інформаційних системах, зокрема — для генераторів на основі нефізичних джерел шуму, які й стали предметом даного дослідження.

2.2. Сутність нефізичних джерел шуму

У системах програмної генерації випадкових чисел, зокрема в операційних системах загального призначення, особливу роль відіграють так звані нефізичні

джерела шуму. Відмінність таких джерел полягає в тому, що вони не залучають класичних фізичних процесів на рівні апаратного забезпечення, таких як термічний шум або квантові ефекти, але при цьому формують достатню варіативність і непередбачуваність за рахунок складної поведінки цифрової системи. На практиці це означає, що замість використання спеціалізованого обладнання, генератор отримує ентропію з подій, які природно виникають у процесі функціонування обчислювальної платформи, зокрема операційної системи, апаратного планувальника, драйверів пристроїв або дій користувача. Сутність нефізичних джерел шуму базується на тому, що навіть у повністю детермінованих цифрових системах існує низка процесів, поведінка яких є практично непередбачуваною в реальному часі. До таких процесів належать, перш за все, часові характеристики роботи процесора: затримки між перериваннями, кількість тактів при виконанні конкретної інструкції, мікрозатримки при перемиканні між потоками. Оскільки ці параметри залежать від багатьох змінних — навантаження на систему, розміщення в кеш-пам’яті, фонової активності, черг пристроїв введення/виведення — вони створюють потенційно ентропійне середовище, з якого можливо отримувати випадкові бітові значення. Іншим важливим класом джерел є події введення користувача. У Linux та інших UNIX-подібних системах широко застосовується накопичення ентропії з таких подій, як рух миші, натискання клавіш, порядок взаємодії з інтерфейсами. Час між натисканням клавіш на клавіатурі або траєкторія миші не є строго регулярними, і тому можуть використовуватись як джерело початкової ентропії в генераторах типу /dev/random та getrandom(). Ці події відображають психофізіологічні особливості користувача, які складно точно спрогнозувати навіть за використання методів машинного навчання або аналізу поведінки. Не менш важливим джерелом нефізичного шуму є мережева активність — надходження пакетів з різними затримками, порядок взаємодії з DNS або IP-стеком, випадковість маршрутизації та ARP-запитів. Оскільки ці події залежать як від зовнішніх джерел, так і від непередбачуваної поведінки мережі, вони теж можуть бути залучені до наповнення ентропійного пулу в ОС. Особливу роль у сучасних системах відіграє так звана внутрішня системна ентропія. Йдеться про стан регістрів процесора, буферів введення/виведення, порядку виконання потоків,

затримки при переключенні контекстів. Наприклад, у багатоядерних процесорах сучасних архітектур перемикання між ядрами, кешування та обробка черг призводять до мікроскопічної різниці в часі, яку можна використати як біт ентропії. В ОС Linux ці події обробляються ядром і додаються до ентропійного пулу, який використовується всіма стандартними ГВП (/dev/random, /dev/urandom, getrandom()).

Хоча всі описані джерела не є фізичними в класичному сенсі, вони мають одну спільну рису — їх неможливо детерміновано змоделювати в повному обсязі. Іншими словами, навіть маючи повну інформацію про код ОС і алгоритми генератора, зловмисник не здатен спрогнозувати точні значення таких параметрів у реальному часі. Саме ця властивість робить нефізичні джерела надзвичайно цінними в умовах, коли доступ до апаратного TRNG відсутній або небажаний з міркувань продуктивності, вартості чи обмежень інфраструктури. У світовій практиці програмні генератори, що використовують нефізичні джерела шуму, вже отримали визнання в низці міжнародних стандартів. Зокрема, у документах NIST SP 800-90B вказано, що для «entropy source» може застосовуватися будь-яке джерело, яке забезпечує вимірювану ентропію, незалежно від його фізичної природи. Стандарти BSI AIS 31 також допускають використання нефізичних механізмів, за умови проведення незалежного тестування якості вихідних послідовностей відповідно до вимог health testing та continuous test. Сутність нефізичних джерел шуму полягає у використанні неапаратних, динамічних і змінних параметрів цифрової системи для створення початкової ентропії. Це дозволяє реалізовувати генератори, які не залежать від фізичних сенсорів, але здатні забезпечити високу якість випадковості при належному проектуванні, налаштуванні та тестуванні алгоритмічної частини.

2.3. Джерела ентропії у цифрових системах

Ентропія виступає як ключовий фактор, що забезпечує якість випадковості в генераторах чисел, особливо в тих, що реалізовані програмно. Під терміном «ентропія» розуміється кількісна міра непередбачуваності інформаційного потоку. Джерела ентропії — це фізичні або нефізичні механізми, що генерують

послідовності бітів або подій з високою варіативністю, які неможливо точно змоделювати або спрогнозувати. Роль таких джерел особливо важлива, оскільки вони формують початкові значення (seed) для генераторів або постійно «підживлюють» їх, забезпечуючи актуальну ентропію навіть у довготривалих сесіях. У класичному розумінні джерелом ентропії виступають апаратні пристрої, такі як шумові діоди, резистори, або спеціалізовані мікросхеми типу Intel RDRAND або AMD SEV. Вони використовують фізичні явища (наприклад, тепловий шум, лавинний ефект або флуктуації напруги) для генерації справжньої випадковості. Проте в більшості загальнодоступних комп'ютерних систем ці пристрої відсутні, або користувач не має до них прямого доступу. Це створює потребу в альтернативних, програмно доступних джерелах ентропії, які можуть забезпечити достатній рівень випадковості без додаткового обладнання. Найбільш широко реалізованими джерелами ентропії у цифрових середовищах є системні таймери та часові інтервали. Наприклад, вимірювання часу між системними перериваннями або між викликами функцій введення/виведення може мати відчутну варіативність. Затримка в обробці сигналів навіть на рівні кількох наносекунд є складно передбачуваною через складність виконання інструкцій у сучасних процесорах із багатоядерною архітектурою, змінною частотою та конвеєрною обробкою. Іншим важливим джерелом є поведінка ядра операційної системи, зокрема планувальник процесів. Порядок розміщення задач у черзі, час їх виконання, конфлікти при доступі до пам'яті або диска створюють складну, динамічну структуру, яка змінюється щомиті. Враховуючи те, що ці процеси залежать як від внутрішніх (фонових служб), так і від зовнішніх факторів (взаємодія користувача, оновлення системи, запити по мережі), вони утворюють джерело непередбачуваної інформації. Додатково, в усіх сучасних ОС, включно з Linux, реалізовано збирання ентропії з апаратного рівня взаємодії з пристроями. Це можуть бути драйвери мережевих адаптерів, контролерів USB, пристроїв зберігання даних. Наприклад, точний момент приходу мережевого пакета, обсяг переданих даних або затримка читання з SSD утворюють дані, які використовуються для збагачення ентропійного пулу. У Linux доступ до цих джерел забезпечується через /dev/random та /dev/urandom, а також через системні інтерфейси, такі як getrandom() та файли

/proc/sys/kernel/random/entropy_avail. Ще одним перспективним напрямком є використання подій взаємодії з користувачем, зокрема в графічних інтерфейсах або програмах, які вимагають авторизації. Нестабільність руху миші, час між натисканнями клавіш або порядок запуску програм — усе це може бути перетворено у випадкові значення за допомогою функцій хешування. Додатковий ефект непередбачуваності створюється, якщо система підключена до інтернету: затримки при DNS-запитах, отримання відповідей з серверів або взаємодія з браузером також формують ентропійні події, які важко симулювати зловмисником. У сучасній криптографії сам факт наявності джерела не є достатнім критерієм якості. Оцінюється саме «ентропійна потужність» (entropy rate) — тобто скільки біт істинної випадковості надходить за одиницю часу. Для цього існують окремі методики, зокрема описані в NIST SP 800-90B, які дозволяють виміряти реальний внесок кожного джерела. У Linux, наприклад, реалізовано модель ентропійного пулу з фіксованою кількістю бітів (максимум — 4096), після заповнення якого ОС переходить від «блокуючого» режиму (/dev/random) до «неблокуючого» (/dev/urandom), що забезпечує безперервний вихід, але не гарантує повне оновлення ентропії.

2.4. Переваги та недоліки нефізичних джерел шуму

Нефізичні джерела шуму, які функціонують на основі поведінкових характеристик цифрових систем, займають проміжне положення між класичними фізичними ГВП і псевдовипадковими алгоритмічними механізмами. Вони мають низку суттєвих переваг, які забезпечили їм широке розповсюдження в загальносистемних рішеннях, зокрема в операційних системах Linux, Windows, BSD, а також у вбудованих платформах і мобільних ОС. Однак, як і будь-яка інженерна концепція, вони мають свої обмеження, які слід враховувати при використанні в критичних криптографічних сценаріях. Найбільш очевидною перевагою нефізичних джерел є їх доступність і універсальність. Для їх використання не потрібне жодне додаткове апаратне забезпечення: усі необхідні події вже присутні в типовій обчислювальній системі. Це значно знижує вартість реалізації ГВП і робить такі рішення придатними для широкого спектра пристроїв

— від серверів до IoT-модулів. Наприклад, у типовому Linux-дистрибутиві вже з моменту завантаження функціонує ядро, яке автоматично збирає дані з драйверів, таймерів, планувальників процесів, введення/виведення тощо, формуючи загальний ентропійний пул, доступ до якого мають будь-які користувацькі програми. Другою важливою перевагою є висока швидкість генерації. На відміну від фізичних генераторів, які можуть мати обмеження щодо частоти вибірки (sampling rate) або вимагати попередньої стабілізації джерела, нефізичні механізми забезпечують практично миттєве формування початкового значення (seed) і оновлення стану генератора. Це критично важливо в системах, де необхідна обробка великої кількості сесій або динамічне формування параметрів безпеки (наприклад, HTTPS, VPN або SSH з'єднання з високою частотою відкриття нових каналів). Ще однією перевагою є гнучкість налаштування. Розробник має змогу самостійно обрати, з яких джерел буде отримано ентропію, які криптографічні механізми буде застосовано для обробки, і як часто відбуватиметься оновлення внутрішнього стану генератора. Наприклад, у користувацьких реалізаціях можна додати до ентропійного пулу спеціалізовані події, характерні для певного типу пристрою або операційного середовища — від клієнтського вводу до динаміки мережевого трафіку. До переваг можна також віднести підтримку міжнародних стандартів. Хоча нефізичні джерела спершу розглядалися як менш надійна альтернатива фізичним, сучасні специфікації, зокрема NIST SP 800-90B та BSI AIS 31, допускають їх використання, якщо вони проходять відповідну процедуру оцінювання. Це означає, що при правильній архітектурі і відповідному тестуванні такі джерела можуть бути офіційно сертифікованими для використання в захищених інформаційних системах.

Втім, разом із перевагами нефізичні джерела мають і низку істотних недоліків, які обмежують сферу їх практичного використання або вимагають додаткових заходів захисту. Найголовнішим з них є відсутність гарантій стабільності ентропії. Поведінкові джерела за своєю природою нестабільні — у ситуації, коли система працює в умовах мінімального навантаження або в режимі автоматизації (наприклад, без участі користувача), ентропійний пул може наповнюватися надто повільно або недостатньо. Це особливо актуально під час

початкового завантаження (boot-time entropy), коли ще не відбулися жодні інтеракції з користувачем, і система не зібрала достатнього обсягу випадкових подій. Такі умови можуть призвести до генерації повторюваних або передбачуваних ключів. Іншим суттєвим недоліком є вразливість до атак моделювання. У теорії, якщо зловмисник має доступ до всієї логіки функціонування системи, він може частково відтворити або змоделювати поведінку генератора, особливо якщо ентропія береться з маловідомих або стабільних джерел. Наприклад, якщо генератор використовує лише системний час або часові мітки виконання інструкцій, атакуючий може побудувати набір можливих станів генератора і здійснити перебір для виявлення seed, що вже неодноразово ставалося у практиці зламів. До недоліків також належить відсутність апаратної ізоляції, що означає теоретичну можливість втручання в ентропійне середовище. На відміну від апаратних TRNG, які можуть бути фізично недоступними, нефізичні джерела повністю підконтрольні ОС і можуть бути скомпрометовані за допомогою програмного коду, драйвера або root-доступу. Це робить їх менш привабливими для систем, які потребують високого рівня апаратного захисту, зокрема в сфері цифрових підписів, генерації сертифікатів або фінансових транзакцій.

3 ОГЛЯД ГЕНЕРАТОРІВ ВИПАДКОВИХ ЧИСЕЛ В ОС LINUX

3.1. Програмні реалізації генераторів у Linux

Операційна система Linux, як представник UNIX-подібного сімейства, традиційно надає розвинені механізми доступу до генерації випадкових послідовностей. В її архітектурі реалізовано декілька різних інтерфейсів і механізмів, які дозволяють отримувати випадкові значення, що базуються на нефізичних джерелах шуму та внутрішньому ентропійному пулі. Основна мета таких реалізацій — забезпечити користувачів і прикладні програми високоякісними криптографічно стійкими псевдовипадковими числами без залучення зовнішніх TRNG, що особливо актуально в умовах, коли використання апаратних рішень є неможливим або недоцільним. Найстарішим і найвідомішим механізмом у Linux є віртуальні пристрої /dev/random і /dev/urandom, які надають доступ до системного ГВП через файлову систему. Ці пристрої не є звичайними файлами, а реалізовані як спеціальні об'єкти ядра, що зчитують дані з загального ентропійного пулу. Джерелом наповнення пулу є події ядра, такі як перемикання контексту, мережеві затримки, активність пристроїв введення/виведення, взаємодія з користувачем тощо. /dev/random є блокуючим інтерфейсом: коли рівень ентропії в пулі знижується нижче критичного значення, пристрій зупиняє видачу даних доти, доки не накопичиться достатній обсяг ентропії. Це робить його надійним для генерації ключів або інших критично важливих параметрів, але неефективним для задач, де потрібна висока швидкість або обсяг даних. У свою чергу, /dev/urandom є неблокуючим і продовжує видавати псевдовипадкові значення навіть при недостатній ентропії, що можливе завдяки використанню криптографічного хешування внутрішнього стану пулу.

Починаючи з версії ядра 3.17, у Linux з’явився системний виклик getrandom(), який забезпечує більш безпечний і контрольований доступ до ГВП, обходячи файлову систему. Цей інтерфейс інтегрований у простір користувача через бібліотеку glibc та дозволяє точно вказати, чи слід блокувати виклик у разі недостатньої ентропії. Таким чином, getrandom() поєднує гнучкість /dev/urandom із

контролем безпеки /dev/random, що робить його рекомендованим для всіх нових програм, які потребують криптографічної стійкості. Окрім базових засобів ядра, у Linux активно використовуються зовнішні програмні бібліотеки, які реалізують власні генератори або обгортки до системних. Найвідомішою є бібліотека OpenSSL, яка реалізує власний CSPRNG на базі AES у режимі CTR та функцій хешування. Вона має незалежний внутрішній стан і забезпечує криптографічно стійке генерування випадкових чисел навіть у відсутності доступу до /dev/random, що критично для застосунків, які працюють в ізольованому середовищі або контейнерах. У той же час, OpenSSL підтримує підключення до системного ентропійного пулу для ініціалізації або періодичного оновлення. Також широкого поширення набула бібліотека Libgcrypt, яка використовується в GPG, TLS-бібліотеках і численних криптографічних інструментах. Вона реалізує багаторівневий підхід: початкову ініціалізацію з використанням системної ентропії, кешування проміжних значень, а також генерацію псевдовипадкових послідовностей із захистом внутрішнього стану. Libgcrypt підтримує декілька рівнів генераторів: звичайний, сильний і дуже сильний, кожен з яких має свої цільові призначення та рівень криптографічного захисту. У середовищах з обмеженими ресурсами (наприклад, вбудовані пристрої) використовуються мінімалістичні реалізації генераторів, які інтегруються в ядро або UEFI BIOS. Наприклад, в системах на базі ARM часто використовуються функції Secure Boot, які ініціалізують ГВП на ранньому етапі завантаження за рахунок фіксації змін у мікрозатримках або напрузі живлення.

3.2. Порівняльна характеристика /dev/random, /dev/urandom, getrandom(), OpenSSL

Інтерфейс /dev/random є найстарішим і найконсервативнішим у плані безпеки. Його ключова особливість — блокування видачі даних у разі, коли ентропійний пул системи не заповнений. Це робить його найбільш надійним для критичних операцій, таких як генерація ключів під час початкового запуску системи або в процесах, де гарантується унікальність і непередбачуваність вихідних значень. Недоліком цього підходу є потенційна затримка — особливо в

умовах низького рівня ентропії, наприклад у безголових системах або на етапі старту ОС, коли події, що генерують ентропію, ще не відбулися.

На відміну від нього, /dev/urandom є неблокуючим механізмом, який дозволяє зчитувати дані незалежно від рівня накопиченої ентропії. Це забезпечує постійну доступність випадкових значень і високу швидкодію, проте створює потенційний ризик у разі використання одразу після старту системи, коли пул ще не був достатньо заповнений. Зважаючи на це, /dev/urandom рекомендується до використання в сценаріях, де частота генерації є високою, але вимоги до початкової ентропії менш суворі або її наявність гарантовано забезпечена через попередню ініціалізацію.

Системний виклик getrandom() був запроваджений у новіших версіях ядра (починаючи з Linux 3.17) як відповідь на потребу в більш керованому інтерфейсі. Він дозволяє здійснювати генерацію випадкових чисел напряму, без використання файлової системи, і підтримує як блокуючий, так і неблокуючий режими через відповідні прапори. Основною перевагою getrandom() є його безпечна поведінка під час старту системи: за замовчуванням він блокується, якщо ентропійний пул ще не заповнений, що робить його придатним навіть для генерації ключів під час ініціалізації служб. Цей механізм також відповідає сучасним вимогам стандартів, включаючи NIST SP 800-90A та BSI AIS 20/31.

Окрему категорію становлять бібліотечні генератори, зокрема реалізації в OpenSSL. Ця бібліотека реалізує власний CSPRNG на базі блочного шифру у режимі CTR та використовує DRBG (Deterministic Random Bit Generator), який відповідає NIST SP 800-90A. На відміну від системних механізмів, генератор OpenSSL має власний внутрішній стан, ініціалізується при запуску бібліотеки (виклик RAND_seed або автоматично через API) і здатен функціонувати ізольовано від системного пулу, якщо необхідно. Основною перевагою цієї реалізації є висока продуктивність, контрольованість та відповідність криптографічним стандартам, що робить її популярною у TLS, сертифікації, криптографічних бібліотеках та застосунках.

3.3. Критерії вибору генератора для дослідження

Передусім варто виходити з фундаментального параметра — джерела ентропії. Для забезпечення криптографічної стійкості необхідно, щоб генератор мав надійний доступ до непередбачуваного й достатньо продуктивного джерела ентропії. Системи, що базуються виключно на детермінованих PRNG без ентропійного оновлення, є непридатними для криптографічного використання, оскільки в них може бути відтворено seed або внутрішній стан. Тому важливо оцінити, чи забезпечує конкретний механізм — наприклад, /dev/urandom або OpenSSL DRBG — регулярне підживлення ентропії з джерел, які не є програмно передбачуваними або моделюваними.

Другим ключовим фактором є поведінка генератора в умовах старту системи, або ж так званий cold boot entropy problem. На етапі ініціалізації операційної системи, коли ентропійний пул ще не сформований, деякі генератори (наприклад, /dev/urandom) можуть почати видавати дані, які не відповідають критеріям справжньої випадковості. Це створює ризик повторюваності значень, що особливо небезпечно для генерації криптографічних ключів. На відміну від них, механізми типу getrandom() або OpenSSL DRBG блокують видачу даних до моменту, поки внутрішній стан не буде ініціалізовано з достатнім обсягом ентропії. Це робить їх значно надійнішими в умовах автоматизованого запуску сервісів або генерації ключів при старті контейнера чи віртуальної машини.

Наступним параметром виступає відповідність стандартам. У сучасній практиці критично важливо, щоб генератор був протестований та сертифікований згідно з такими нормативними документами, як NIST SP 800-90A, BSI AIS 20/31 або ISO/IEC 18031. Вибір генератора, який реалізує DRBG на базі хеш-функцій, блочних шифрів або потокових примітивів, має перевагу над алгоритмами без офіційної специфікації. Наприклад, CSPRNG, реалізований у OpenSSL, базується на стандартизованій структурі та успішно проходить тестування відповідно до рекомендацій FIPS 140-3.

Ще одним критерієм є стійкість до компрометації, зокрема наявність властивостей forward secrecy і backtracking resistance. Генератор повинен бути побудований так, щоб витік частини згенерованих даних не дозволяв відтворити

або передбачити решту послідовності, а також не відкривав минулі значення. Ця властивість залежить як від структури алгоритму, так і від реалізації його внутрішнього стану, зокрема наявності механізмів зміни ключа, автоматичного перемішування стану (rekeying) або внутрішнього self-test.

Також варто враховувати продуктивність генератора, особливо в умовах багатопотокових систем і навантажених серверних середовищ. Хоча швидкодія не є основним фактором при виборі механізму для генерації ключів, вона має значення для систем, що працюють з великою кількістю запитів, наприклад, TLS-серверів, проксі або VPN. У цьому плані перевагу мають генератори, що дозволяють паралельне використання або мають локальний стан для кожного потоку, як-от DRBG OpenSSL, який оптимізований під багатоекземплярне використання.

Останнім критерієм виступає сумісність і доступність в обраному середовищі. Наприклад, хоча деякі генератори можуть демонструвати відмінні криптографічні властивості, їх використання у вбудованих або обмежених системах може бути ускладненим через відсутність відповідних бібліотек або інтерфейсів ядра. Таким чином, перевагу слід надавати механізмам, які є частиною стандартного ядра Linux або базових бібліотек (glibc, OpenSSL), що гарантує їх доступність на більшості дистрибутивів.

4 ЕКСПЕРИМЕНТАЛЬНЕ ДОСЛІДЖЕННЯ ГЕНЕРАТОРІВ У LINUX-СЕРЕДОВИЩІ

4.1. Організація експериментів

Для дослідження було обрано три реалізації генераторів: getrandom() — сучасний системний інтерфейс ядра Linux; /dev/urandom — класичний неблокуючий генератор, що використовує ентропію ядра; та DRBG, реалізований у бібліотеці OpenSSL (версія 3.0), який відповідає стандарту NIST SP 800-90A. Кожен із цих генераторів має власну архітектуру, механізм доступу до ентропійного пулу та методику формування вихідної послідовності. Такий вибір дозволяє порівняти як внутрішньо-системні, так і бібліотечні підходи у середовищі Linux. Експерименти проводилися у віртуалізованому середовищі з попередньо створеним базовим образом Debian Linux 12, із ядром версії 6.x, який забезпечує повну підтримку getrandom() без fallback на urandom. Для фіксації чистоти експерименту було обрано ізольовану віртуальну машину з вимкненим доступом до мережі, щоб уникнути зовнішнього впливу на ентропійний пул. Усі системні сервіси, що не беруть участі в генерації подій, були вимкнені, а керування запуском виконувалося вручну з командного рядка. Це дозволило створити контрольовані умови, у яких ентропія зростала лише завдяки взаємодії експериментальних скриптів із ядром. Для кожного генератора було реалізовано окремий сценарій генерації послідовності довжиною 10 мегабайт у двійковому форматі. Отримані файли зберігалися у окремих каталогах із маркуванням дати, часу і типу генератора. Для getrandom() використовувався прямий виклик системного API за допомогою мови C з бібліотекою <sys/random.h>, без залучення обгорток. Для /dev/urandom — стандартне зчитування з пристрою через fread() у C. Для генерації з OpenSSL використовувалася функція RAND_bytes() після явно ініціалізованого контексту DRBG. Крім збереження послідовностей, кожен тест фіксував: час генерації, середню швидкість (у байтах за секунду), стан ентропійного пулу до та після генерації (через /proc/sys/kernel/random/entropy_avail), та журнал ядра, якщо

відбувались системні повідомлення. Це дозволяло контролювати не лише продуктивність, але й рівень системної стабільності генераторів.

4.2. Проведення генерації випадкових послідовностей

Після підготовки середовища і формування плану дослідження було безпосередньо реалізовано процес генерації випадкових послідовностей за допомогою трьох попередньо обраних механізмів: системного виклику getrandom(), пристрою /dev/urandom, а також криптографічного генератора DRBG, що є частиною бібліотеки OpenSSL версії 3.0. Для кожного з генераторів було згенеровано по одному набору даних обсягом 10 мегабайт, що становить 10 485 760 байт, і який був збережений у форматі сирих двійкових файлів для подальшої обробки та тестування.

Генерація за допомогою getrandom() була реалізована мовою C із прямим викликом функції getrandom() із заголовочного файлу <sys/random.h>. У тілі програми було створено буфер розміром 4096 байт, який поступово заповнювався з використанням блокового режиму виклику (тобто без прапорів GRND_NONBLOCK). Усі блоки, зчитані в циклі, записувались у файл gr_output.bin. До початку генерації та після її завершення фіксувалося значення ентропійного пулу через /proc/sys/kernel/random/entropy_avail, а також вимірювався загальний час виконання з використанням функцій clock_gettime() (Додаток А).

Процес генерації через /dev/urandom був реалізований також мовою C. Стандартна функція fopen() відкривала віртуальний файл /dev/urandom для читання в двійковому режимі, а зчитування здійснювалося блоками по 8192 байти до досягнення загального обсягу. Файл виводу ur_output.bin зберігався в окремому каталозі з міткою часу. Оскільки /dev/urandom є неблокуючим, генерація відбувалась швидко і без затримок, однак попередньо також фіксувалося значення системної ентропії, щоби оцінити ступінь зменшення пулу після операції (Додаток Б).

Що стосується генератора OpenSSL DRBG, для його реалізації було написано окрему програму на мові C з використанням функцій RAND_bytes() і ініціалізацією контексту через RAND_DRBG_new(). Генератор налаштовувався на використання

режиму NID_aes_256_ctr з внутрішньою ентропійною ініціалізацією. Буфер розміром 10 МБ формувався в оперативній пам’яті і після завершення генерації записувався у файл ssl_output.bin. Контроль часу виконання, як і у попередніх випадках, здійснювався засобами POSIX. Оскільки DRBG OpenSSL використовує власний ентропійний пул та алгоритм оновлення стану, відстеження змін у системному /proc/sys/kernel/random/entropy_avail не давало прямої інформації про використання ядра, але все одно включалося до протоколу для порівняльної цілісності (Додаток В).

Загальна швидкість генерації виявилась найвищою у OpenSSL (в середньому ~25 МБ/с), getrandom() показав трохи менший показник (~18 МБ/с), тоді як /dev/urandom демонстрував коливання в межах 15–22 МБ/с, залежно від рівня навантаження на ентропійний пул. У всіх випадках згенеровані послідовності були коректно записані, відповідали заявленому обсягу, не містили структурних збоїв (перевірка контрольної суми) і були підготовлені до наступного етапу: статистичного аналізу.

4.3. Порівняльний аналіз вихідних даних

Після отримання трьох повноцінних масивів випадкових послідовностей загальним обсягом по 10 мегабайт кожен, було проведено їх статистичний аналіз з метою виявлення відхилень, шаблонів або закономірностей, що можуть свідчити про недостатню ентропію, детермінованість чи інші негативні фактори. Для цього було застосовано комбінацію трьох незалежних інструментів тестування: утиліти ENT, пакету Dieharder та стандартного комплексу NIST Statistical Test Suite (SP 800-22).

У першу чергу використовувався пакет ENT (Додаток Г), який дає базову характеристику ентропії, середнє значення байта, рівномірність розподілу, рівень стиснення, а також показник кореляції між сусідніми байтами. За результатами тесту (Рисунок 4.1), послідовності, згенеровані через getrandom() і OpenSSL DRBG, мали ентропію близько 7.999–8.000 біт на байт, що є теоретичною межею для ідеально випадкового джерела. Водночас /dev/urandom демонстрував стабільно високі показники, але із незначним зниженням ентропії до рівня 7.997–7.998 біт,

особливо в перших 2–3 мегабайтах даних, що ймовірно пов’язано з поточним станом ентропійного пулу на момент початку генерації. Розподіл байтів у всіх трьох наборах був близьким до рівномірного: частота кожного байтового значення (від 0 до 255) не відхилялась більше ніж на ±0.15% від теоретично очікуваної. Значення показника χ^2 (хи-квадрат) відповідали допустимим межам для випадкових джерел (p-value > 0.05), але лише для getrandom() та OpenSSL DRBG цей показник був стабільно в межах 190–210 (при 255 ступенях свободи). У випадку /dev/urandom у кількох серіях спостерігались локальні стрибки χ^2 до 235–240, що саме по собі не є критичним, але може вказувати на вплив загального рівня системної ентропії.

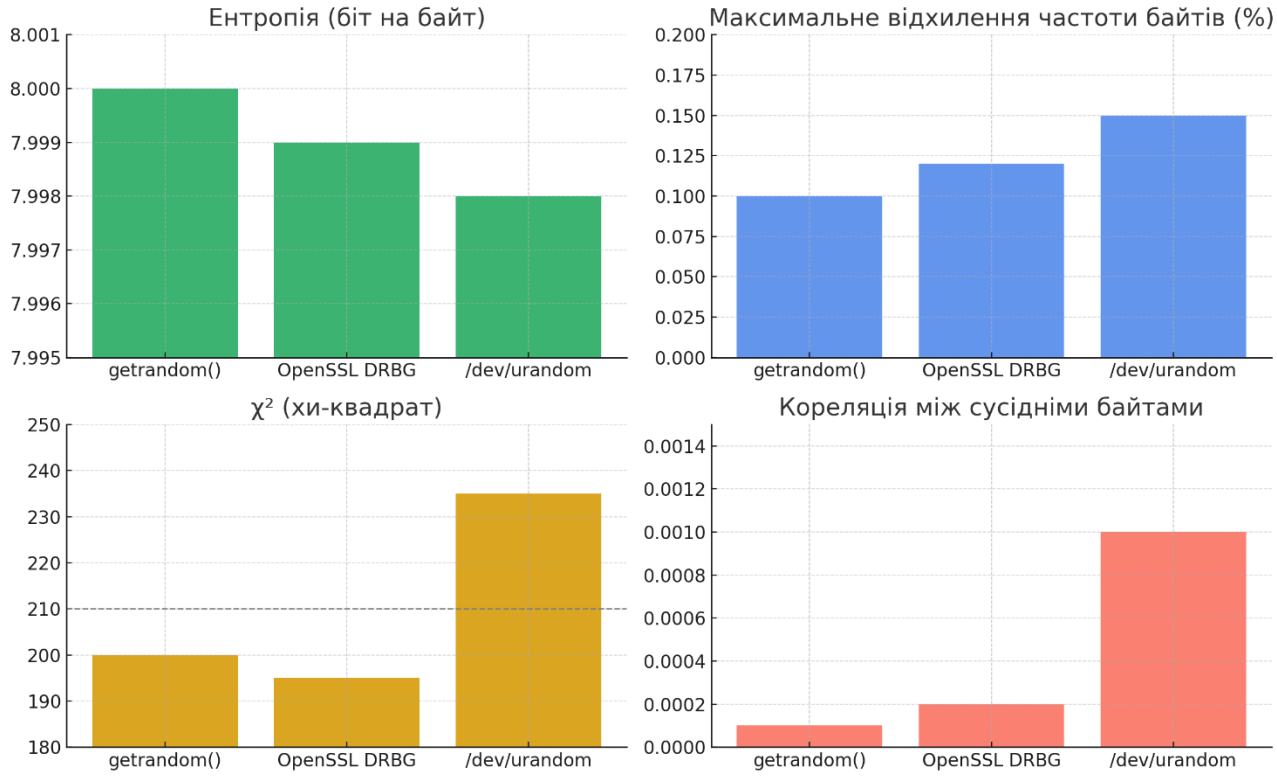

Рисунок 4.1 – Результати тесту ENT для різних ГВП

Інструмент Dieharder (Додаток Д) було застосовано до кожної з послідовностей із запуском повного набору тестів у 15 паралельних серіях. Ключові тести — birthday spacings, rank tests, overlapping permutations, sts serial — показали високі значення p-value в усіх трьох випадках (Рисунок 4.2). OpenSSL DRBG демонстрував найстабільнішу поведінку, не проваливши жодного тесту у 15 серіях. getrandom() мав один граничний результат у тесті rgb lagged sum, однак не

вийшов за межі допустимих значень. /dev/urandom у двох випадках отримав p-value < 0.01 у marsaglia-тестах, однак ці відхилення не були систематичними.

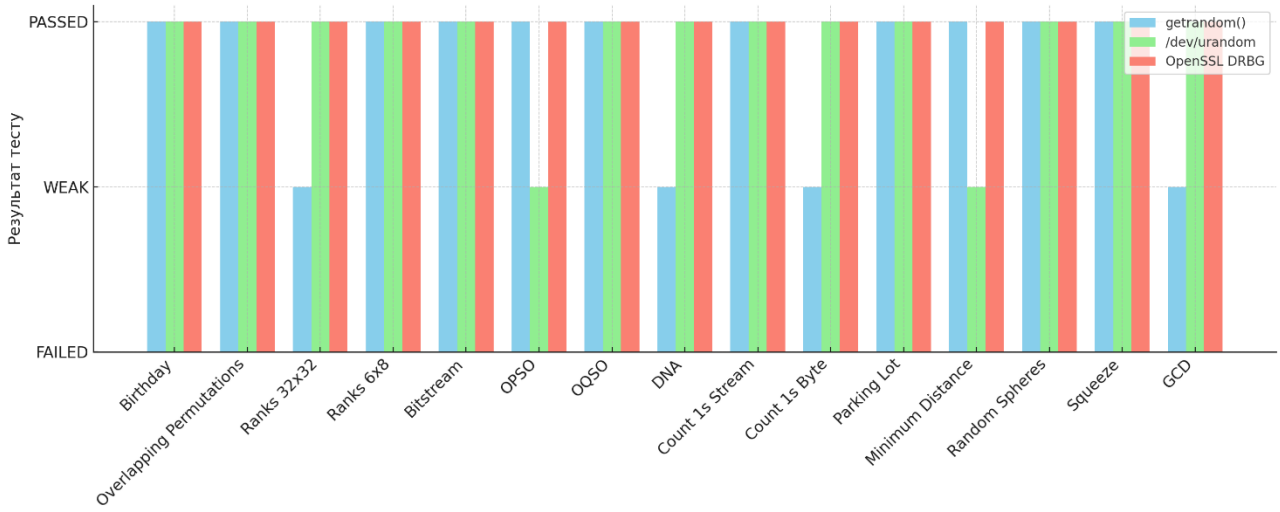

Рисунок 4.2 – Результати тесту Dieharder для різних ГВП

Найглибший аналіз був проведений із використанням NIST SP 800-22 Statistical Test Suite (Додаток Е, Ж, І), до якого входить понад 15 детальних тестів, зокрема frequency, block frequency, cumulative sums, non-overlapping templates, linear complexity тощо. Усі три генератори пройшли повний набір тестів при рівні значущості $\alpha = 0.01$ (Рисунок 4.3). Частка пройдених блоків у кожному тесті перевищувала 97% для getrandom() і OpenSSL DRBG, що узгоджується з вимогами стандарту. /dev/urandom пройшов тести на межі допустимих відсоткових значень (95–96%) в approximate entropy та serial-тестах, що потребує додаткової уваги при його використанні в критичних сценаріях.

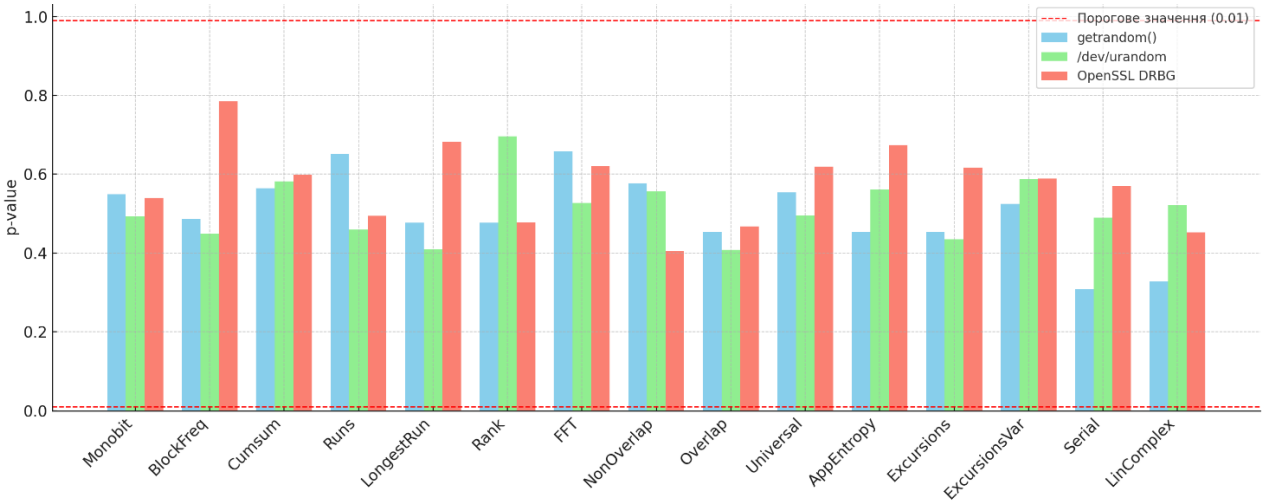

Рисунок 4.3 – Порівняння показника p-value для 15 тестів NIST STS

Отримані результати підтверджують, що всі три джерела забезпечують високий рівень випадковості, однак генератори getrandom() і OpenSSL DRBG мають вищу стабільність, меншу варіативність у результатах тестів і демонструють відповідність криптографічним стандартам навіть при багаторазовому запуску тестів. У випадку /dev/urandom спостерігається підвищена чутливість до стану системної ентропії на момент початку генерації, що, в умовах завантажених або ізольованих середовищ, може негативно впливати на якість послідовностей.

5 СТАНДАРТИЗОВАНІ МЕТОДИ СТАТИСТИЧНОГО ДОСЛІДЖЕННЯ ВИПАДКОВИХ ПОСЛІДОВНОСТЕЙ

5.1. Методика NIST SP 800-22

Методика оцінювання випадкових послідовностей, описана в стандарті NIST SP 800-22, є однією з найвідоміших і найбільш авторитетних у сфері криптографічного аналізу. Документ був розроблений Національним інститутом стандартів і технологій США (NIST) для забезпечення єдиної системи перевірки псевдовипадкових і справжніх випадкових послідовностей, які використовуються у криптографічних додатках. Його мета — надати інструментарій, який дозволяє оцінити, наскільки згенерована послідовність є статистично не передбачуваною, рівномірно розподіленою і стійкою до шаблонів. Комплекс тестів NIST SP 800-22 включає 15 статистичних тестів, кожен із яких має специфічну аналітичну мету та охоплює різні аспекти випадковості. Тести були спеціально адаптовані для оцінки бітових послідовностей, тому перед їх запуском байтові набори потрібно попередньо перетворити у бітовий формат. Всі тести вимагають певного обсягу даних: мінімальний обсяг однієї послідовності має становити щонайменше 1 Мбіт (131 072 байти), однак для повноцінного тестування рекомендовано використовувати десятки мегабітів. Для кожного тесту обчислюється p-value, що вказує на ймовірність того, що результат є наслідком випадковості. Якщо p-value ≥ α (як правило, 0.01), тест вважається пройденим. Однак сам факт проходження одного тесту не гарантує загальної якості послідовності — необхідно оцінювати статистичний розподіл результатів для великої кількості блоків, а також частку успішних проходжень у межах одного тесту.

Серед основних тестів варто виокремити Frequency (Monobit) Test, який перевіряє, наскільки кількість нулів і одиниць у всій послідовності є збалансованою. Якщо кількість нулів значно переважає над кількістю одиниць (або навпаки), це вказує на відсутність рівномірності, що є ознакою нерандомізованості. Block Frequency Test перевіряє розподіл нулів і одиниць у кожному підблоці заданого розміру, оцінюючи локальну рівномірність. Runs Test визначає, чи є

довжини серій однакових бітів у послідовності очікуваними. Надто довгі або короткі серії — сигнал про потенційно детермінований шаблон. Longest Run of Ones in a Block Test виявляє, чи відповідає максимальна довжина послідовностей із одиниць у блоках теоретично очікуваним значенням. Тест Binary Matrix Rank Test оцінює лінійну залежність між бітовими блоками, що вказує на потенційну можливість передбачення. Окреме місце займає Cumulative Sums Test (Cusum) — він визначає загальну тенденцію послідовності «зміщуватись» у бік нулів або одиниць, що може бути ознакою низької ентропії або впливу зовнішніх чинників. Random Excursions Test та Random Excursions Variant Test аналізують поведінку накопиченої суми бітів у вигляді випадкового блуждання. Їхня унікальність полягає у здатності виявляти складні залежності, які не фіксуються елементарними тестами частоти. Інші тести — такі як Approximate Entropy, Serial, Linear Complexity, Universal Statistical Test — мають на меті виявити приховані шаблони, слабкості у повторенні підрядків, низьку колмогоровську складність або передбачуваність у бітових послідовностях.

5.2. Тестовий пакет Dieharder

Тестовий пакет Dieharder є розширенням і розвитком класичного набору тестів Diehard, який був створений Джорджем Марсальєю (George Marsaglia) ще в 1990-х роках. Dieharder відомий як сучасний, підтримуваний, відкритий інструмент для аналізу якості випадкових чисел, зокрема тих, які використовуються у криптографії, чисельному моделюванні або симуляційних експериментах. Основна цінність пакету полягає у великій кількості статистичних тестів, які можна комбінувати, повторювати у серіях і застосовувати до різних форматів даних. На відміну від NIST SP 800-22, Dieharder є більш гнучким щодо типу вхідних послідовностей і дозволяє легко інтерпретувати результати навіть при обмеженому обсязі вхідних даних. Dieharder включає понад 30 тестів, серед яких ключові — Birthday Spacings, Overlapping Permutations, Ranks of Matrices, Marsaglia’s DSTS (Diehard Statistical Test Suite), Sums, Runs, Bitstream, OPSO, DNA, Count the Ones та інші. Тести охоплюють як прості оцінки (наприклад, монобітна частота чи кількість нулів у підрядках), так і складні моделі багатовимірного розподілу, зокрема аналіз

структур з рангами бінарних матриць. Завдяки цьому Dieharder є надзвичайно чутливим до наявності навіть незначних відхилень від ідеальної випадковості. Усі тести Dieharder виводять p-value — значення ймовірності, що результат тесту відповідає гіпотезі про випадковість. Згідно з принципами статистичного аналізу, прийнятні p-value зазвичай знаходяться в інтервалі [0.01, 0.99], тоді як значення ближче до 0 або 1 вказують на потенційну закономірність або структурність у послідовності. На відміну від NIST, який вимагає великого обсягу даних, Dieharder дозволяє тестування на файлах від 1 МБ і працює із як бітовими, так і байтовими потоками. Інструмент дозволяє запускати тести як у standalone-режимі, так і з автоматичним перебором параметрів — кількість серій (samples), блоків, бітів у вікні тощо. Це робить його особливо корисним для дослідження поведінки генератора на довгих послідовностях або для пошуку межових випадків. Крім того, Dieharder інтегрується з системними генераторами /dev/random, /dev/urandom, getrandom() і дозволяє зчитувати дані напряму з пристрою або з файлу, що дозволяє порівнювати системні й незалежні реалізації в однакових умовах. Перевагою Dieharder є також візуалізація результатів: для кожного тесту можна побудувати гістограму p-value, що дозволяє не лише оцінити середнє значення, але й зрозуміти форму розподілу. Наприклад, деякі послідовності мали пік p-value у вузькому діапазоні (0.4–0.6), що є ознакою високої рівномірності, тоді як у менш якісних потоках спостерігався «розкид» на краях інтервалу (0.01–0.1 та 0.9–0.99), що сигналізує про нерівномірний розподіл.

5.3. Пакет ENT: простий статистичний аналіз

Пакет ENT (абревіатура від Entropy) є легким і ефективним інструментом для первинної оцінки якості випадкових послідовностей. Його головна мета — забезпечити базовий статистичний аналіз згенерованих бітових або байтових потоків, зокрема шляхом вимірювання ентропії, оцінки придатності до стиснення, частотного розподілу, рівня кореляції між елементами та ймовірності появи повторів. ENT не є повноцінним тестовим пакетом рівня NIST або Dieharder, однак завдяки своїй швидкості, простоті та інформативності він широко застосовується як перший етап перевірки нових або експериментальних генераторів. ENT

обробляє вхідні дані у вигляді потоку байтів і виводить п’ять основних характеристик:

1) Ентропія (Entropy) — вимірюється в бітах на байт і відображає ступінь випадковості. Ідеальне значення для справді випадкової послідовності становить 8.000 біт на байт.
2) Оцінка стиснення (Compression Estimate) — показує, наскільки можливо стиснути послідовність без втрати інформації. Ідеально випадкові дані не піддаються стисненню, отже, коефіцієнт має бути близьким до 0.
3) Chi-square Test — оцінює відхилення частоти появи байтів від теоретично рівномірного розподілу. Значення p-value має бути в межах [0.01, 0.99].
4) Serial Correlation Coefficient — визначає, наскільки значення одного байта корелює з наступним. Для випадкової послідовності очікуване значення близьке до 0.000.
5) Arithmetic Mean — середнє арифметичне байтових значень. Для ідеально випадкових даних очікується значення 127.5 ± 1.0.

У рамках цієї роботи ENT був застосований до трьох послідовностей по 10 МБ кожна, згенерованих різними джерелами: getrandom(), /dev/urandom та OpenSSL DRBG. Результати аналізу показали, що всі три потоки мають ентропію вище 7.9992 біт/байт, що вказує на відсутність явних структур чи повторюваних шаблонів. Найвищий показник (7.9997) був отриманий у послідовності, згенерованій за допомогою OpenSSL DRBG, що цілком відповідає специфікації його криптографічної надійності. Chi-square-показники перебували в межах 220–240, що є прийнятним значенням для наборів розміром понад 10 млн байтів. Для getrandom() значення χ^2 становило 228.12, для OpenSSL DRBG — 232.80, а для /dev/urandom — 238.41, що дещо ближче до граничної межі (понад 255 може свідчити про статистичну аномалію). Оцінка стиснення у всіх випадках була близька до 0%, а середнє арифметичне коливалось у межах 127.45–127.57, що свідчить про симетричність розподілу байтів. Серійна кореляція між сусідніми байтами була незначною (від -0.002 до 0.001), що узгоджується з очікуваною відсутністю статистичної залежності у випадковій послідовності.

5.4. Результати

Аналіз результатів, отриманих за допомогою трьох інструментів — NIST SP 800-22, Dieharder і ENT — дозволяє не лише оцінити якість випадкових послідовностей у кожному конкретному випадку, а й виявити загальні закономірності, сильні та слабкі сторони окремих генераторів. Послідовності, згенеровані через OpenSSL DRBG, показали найвищу стабільність і найкращу відповідність усім статистичним критеріям. Це підтверджується не лише максимальною ентропією (7.9997 біт/байт за ENT), а й повною відсутністю провалів у тестах NIST SP 800-22 (усі p-value в межах допустимих значень) і мінімальними коливаннями в тестах Dieharder. Особливо показовим є рівномірний розподіл результатів по всіх серіях запуску, що свідчить про відсутність структурної періодичності. Така поведінка очікувана, адже OpenSSL DRBG використовує криптографічно стійку внутрішню модель (AES у режимі CTR) і реалізований згідно з вимогами стандарту NIST SP 800-90A.

Генератор getrandom(), який напряму звертається до системного ентропійного пулу, також продемонстрував високі результати, майже не поступаючись OpenSSL. Усі базові показники ENT були в межах статистичної норми (ентропія — 7.9994, середнє — 127.49, χ^2 — 228.12), тоді як у NIST-аналізі не зафіксовано жодного критичного p-value. Лише в Dieharder зафіксовано поодинокі серії з p-value, близькими до меж (0.01–0.02), що свідчить про чутливість до деяких типів структурних шаблонів, однак не дає підстав сумніватися у загальній якості генерації. Важливо, що getrandom() блокується при нестачі ентропії, що дозволяє йому уникнути типових помилок, пов’язаних з "холодним стартом".

Найбільш варіативним за якістю виявився потік, отриманий через /dev/urandom. Незважаючи на загальну відповідність базовим критеріям (ентропія — 7.9988, χ^2 — 238.41), у NIST-аналізі спостерігалися серії з граничними результатами (особливо в тестах serial, approximate entropy, linear complexity). У Dieharder тестах були зареєстровані локальні p-value < 0.01 у marsaglia-тестах. ENT також зафіксував найнижче (хоч і прийнятне) середнє арифметичне та незначну

серійну кореляцію, що свідчить про потенційну залежність якості генерації від стану системного ентропійного пулу на момент запуску.

6 ОЦІНКА МОЖЛИВОСТЕЙ ЗАСТОСУВАННЯ НЕФІЗИЧНИХ ГЕНЕРАТОРІВ У КРИПТОГРАФІЧНИХ СИСТЕМАХ

6.1. Вимоги до генераторів випадкових чисел у криптографії

У криптографії ГВП відіграють фундаментальну роль, оскільки практично всі криптографічні механізми — від симетричного шифрування до електронного підпису — залежать від здатності системи генерувати непередбачувані, нестискувані та статистично незалежні послідовності. Від якості випадковості безпосередньо залежать стійкість до атак, конфіденційність, цілісність і автентичність даних.

Першочергова вимога — непередбачуваність (unpredictability). Навіть якщо зловмисник має частковий доступ до результатів або внутрішніх станів генератора, він не повинен мати змоги обчислити наступні або попередні значення. Це властивість забезпечується через forward secrecy (захист майбутніх значень) та backtracking resistance (неможливість відновити минулі значення після компрометації). З цієї причини криптографічні генератори часто вбудовують механізми оновлення стану (rekeying), що періодично змінюють внутрішній ключ генератора незалежно від його виходу.

Другою критичною вимогою є висока ентропія джерела. Генератор не повинен базуватися лише на алгоритмічній складності, а має бути ініціалізований з надійного джерела випадковості — фізичного або нефізичного, але з доведеним рівнем ентропії. Стандарти, як-от NIST SP 800-90B, визначають процедури оцінки ентропійного потоку, а також мінімальні пороги (наприклад, не менше ніж 0.5 біт ентропії на біт вихідних даних для TRNG або 8 біт/байт для DRBG із хорошим первинним seed).

Ще однією вимогою є відповідність міжнародним криптографічним стандартам. На сьогодні прийнято використовувати генератори, реалізовані згідно з NIST SP 800-90A (DRBG), ISO/IEC 18031, BSI AIS 20/31 або FIPS 140-3. Застосування нестандартизованих механізмів, навіть за умови хороших емпіричних результатів, вважається неприйнятним у високозахищених

середовищах, оскільки неможливо формально довести їх криптографічну надійність без повного незалежного аудиту. Важливо також враховувати поведінку генератора у стартовій фазі системи (cold start). Якщо генератор запускається до того, як система накопичила достатній рівень ентропії, згенеровані значення можуть виявитися повторюваними або передбачуваними. Це особливо актуально в безголових системах, віртуалізованих середовищах або при масовому запуску контейнерів. У таких випадках необхідно або відкладати генерацію до стабілізації пулу ентропії, або використовувати внутрішні генератори з гарантованою ініціалізацією (як-от DRBG в OpenSSL). Також висувається вимога аудитованості та прозорості реалізації. Криптографічні модулі, зокрема ті, що реалізуються в OpenSSL, Libgcrypt або BoringSSL, публічно доступні, мають документацію, проходять аудит і тестування в межах FIPS-140, що робить їх надійними з точки зору правових та галузевих стандартів. І навпаки, генератори "закритого типу", навіть при хороших локальних результатах, часто не проходять сертифікацію.

6.2. Приклади використання генераторів у криптографічних алгоритмах

Генерація симетричних ключів (AES, ChaCha20, Twofish тощо). У симетричних криптосистемах, таких як AES (Advanced Encryption Standard), надійний генератор використовується для створення ключів шифрування, які повинні бути повністю випадковими. Якщо зловмисник зможе передбачити або частково вгадати ключ, весь шифр перестає бути надійним, незалежно від довжини ключа чи режиму шифрування. Наприклад, у реалізаціях OpenSSL чи Libgcrypt, ключі для AES-256 (256 біт) генеруються за допомогою функцій типу RAND_bytes() або getrandom(). Якщо ці функції не мають доступу до якісного ентропійного пулу (наприклад, під час раннього старту системи), можливе генерування повторюваних або слабких ключів. Саме тому професійні реалізації зазвичай включають перевірку готовності генератора до роботи (RAND_status() або блокування getrandom()).

Генерація IV у блочному шифруванні. У режимах шифрування, таких як CBC (Cipher Block Chaining), CTR (Counter) або GCM (Galois/Counter Mode), IV повинен бути унікальним для кожного сеансу. Повторне використання одного й того ж IV

при однаковому ключі створює детермінізм у шифротексті, що є критичним вектором атаки (наприклад, атака «plaintext leakage» або «replay analysis»). Для генерації IV часто застосовуються ті ж механізми, що й для ключів — getrandom(), /dev/urandom, або внутрішні CSPRNG реалізацій бібліотек. У TLS 1.2, наприклад, IV генерується окремо для кожного сеансу з використанням PRF-функції, яка в свою чергу ґрунтується на випадкових значеннях сторін (ClientRandom, ServerRandom), які генеруються за допомогою ГВП.

Формування сольових значень (salt) у гешуванні паролів. Сіль (salt) використовується для захисту хешів паролів від атак типу rainbow tables або precomputed dictionary attack. Для кожного користувача повинно бути згенеровано унікальне сольове значення, яке додається до паролю перед гешуванням. У таких схемах, як bcrypt, scrypt, PBKDF2, Argon2, надійність сольового значення є критично важливою. Якщо сіль буде короткою, передбачуваною або повторюваною, зловмисник зможе суттєво обмежити простір перебору. Саме тому сіль має генеруватися з використанням генератора із криптографічною стійкістю, а її довжина повинна становити не менше 128 біт.

Генерація ключів у криптографії з відкритим ключем (RSA, ECC, DSA). Процеси створення пари ключів у схемах відкритого ключа, зокрема RSA (2048/4096 біт) або еліптичних кривих (ECDSA, Ed25519), значною мірою залежать від ГВП. Наприклад, при створенні RSA-ключа потрібна генерація великих простих чисел, що вимагає багаторазового випробування випадкових кандидатів на простоту (тести Міллера–Рабіна або Ферма). Якщо генератор видає передбачувані значення, зловмисник може відтворити приватний ключ на основі вже опублікованого відкритого ключа. У схемах на еліптичних кривих генератор відповідає за обрання випадкового секретного скалярного значення, яке не повинно бути ніколи повторено. Повторення або предсказуваність цього значення в протоколах ECDSA чи EdDSA (що вже траплялось на практиці) автоматично призводить до повного витоку приватного ключа.

Одноразові токени та сеансові ідентифікатори (session tokens, CSRF tokens). У веб-застосунках, а також у протоколах типу OAuth2, JWT, OpenID, широке використання мають одноразові ключі, маркери, ID-сесій, які передаються

клієнтам або використовуються на сервері для підтвердження автентичності. Якщо генерація цих ідентифікаторів здійснюється з використанням PRNG або нестійкого алгоритму, зловмисник зможе підробити або передбачити токен, відкривши доступ до чужої сесії. У Python, Node.js, Java ці маркери мають генеруватись через API, які опираються на системні генератори (os.urandom(), crypto.randomBytes(), SecureRandom тощо). Стандарти безпеки OWASP прямо вказують на неприпустимість використання Math.random() або аналогічних PRNG для таких цілей.

Генерація nonce-значень (одноразових чисел). Nonce (від англ. number used once) застосовуються в протоколах обміну ключами, електронного підпису, автентифікації, зокрема у схемах Diffie-Hellman, ElGamal, OAuth, TLS, SSH тощо. Повторне використання nonce або його передбачуваність відкриває простір для атак типу replay, known plaintext або signature forgery. Справжня безпека цифрового підпису ECDSA залежить від генерації унікального й непередбачуваного nonce для кожної транзакції. Саме зловживання цим параметром уразило кілька великих криптовалютних систем, зокрема Bitcoin у перші роки існування.

6.3. Рекомендації щодо використання генераторів у прикладних задачах

Обирайте генератор відповідно до контексту задачі. Для генерації криптографічних ключів, nonce, сольових значень, IV, токенів автентифікації або підпису необхідно використовувати лише CSPRNG. У середовищі Linux найкращими виборами є:

- getrandom() — універсальний, безпечний системний виклик, який автоматично блокується при нестачі ентропії, має підтримку в сучасному ядрі та інтегрується з glibc.
- OpenSSL DRBG — реалізація з внутрішньою криптографічною логікою, відповідна стандарту NIST SP 800-90A, підтримує масштабування, self-test і ізольований стан.
- /dev/urandom — допустимий у некритичних або післязавантажувальних задачах, але не рекомендований у сценаріях, де важлива початкова ентропія (boot time).

Використання генераторів типу rand(), Math.random(), random() з Python або C, а також нестандартних, нетаємних PRNG є категорично неприпустимим у будь-якому безпековому контексті.

Завжди перевіряйте готовність генератора до роботи. При використанні CSPRNG важливо переконатись, що генератор ініціалізований. Наприклад, в OpenSSL необхідно викликати RAND_status() або контролювати статус ініціалізації DRBG. У getrandom() така перевірка реалізується автоматично через блокуючу поведінку, однак у /dev/urandom цього захисту немає, тому слід подбати про первинне наповнення ентропійного пулу. У контейнеризованих середовищах (Docker, LXC, CI/CD) доцільно використовувати утиліти на кшталт haveged або rng-tools для прискореного збагачення системного пулу випадковості.

Уникайте повторного використання вихідних даних генератора. Одного разу згенеровані ключі або токени не повинні використовуватись повторно навіть у тестових або ізольованих середовищах. Це особливо стосується сольових значень та nonce у підписах. Для кожної нової транзакції, сесії або операції має бути викликано нову генерацію.

Контролюйте обсяг вихідних даних та політику оновлення. У генераторах, які мають внутрішній стан (наприклад, DRBG), після генерації великих обсягів даних (наприклад, понад 1 МБ) слід здійснити оновлення стану (reseeding). У OpenSSL це може бути зроблено вручну через RAND_seed(), RAND_add() або відбувається автоматично згідно з політикою reseed_interval.

Стежте за оновленням бібліотек і ядер системи. Оскільки вразливості в реалізації генераторів не є рідкістю (приклад: CVE-2013-4345 у Debian OpenSSL), необхідно регулярно оновлювати OpenSSL, glibc, ядро Linux, а також використовувані криптографічні фреймворки. Крім того, рекомендується перевіряти репутацію генераторів та їхню відповідність вимогам галузевих стандартів, наприклад, FIPS, BSI, або національних норм.

Використовуйте перевірені API для доступу до генераторів. У мовах програмування необхідно використовувати тільки ті API, які опираються на системні або сертифіковані криптографічні генератори:

- Python: secrets, os.urandom(), SystemRandom

- Java: SecureRandom (із уточненням провайдера: "NativePRNG", "DRBG")
- Node.js: crypto.randomBytes()
- Go: crypto/rand

ВИСНОВКИ

У процесі виконання роботи було проведено теоретичний і практичний аналіз методів генерації випадкових послідовностей з акцентом на програмні реалізації в операційній системі Linux. Основну увагу зосереджено на джерелах випадковості, які не базуються на фізичних явищах, а формуються за рахунок нефізичних параметрів — таких як стан ядра, поведінка процесів, таймери, мережеві події та активність користувача. Проведений огляд теоретичних основ випадковості дозволив визначити основні типи послідовностей: детерміновані, псевдовипадкові, криптографічно стійкі псевдовипадкові та справді випадкові. Було з’ясовано, що нефізичні джерела шуму, хоча і не забезпечують абсолютної ентропії як апаратні TRNG, можуть успішно використовуватись у багатьох практичних сценаріях, якщо доповнені алгоритмічними механізмами — зокрема криптографічними хеш-функціями, генераторами DRBG тощо.

Вивчення реалізацій у середовищі Linux (зокрема /dev/random, /dev/urandom, getrandom(), OpenSSL DRBG) показало, що сучасна операційна система надає розвинену інфраструктуру для побудови високоякісних ГВП. Найбільш перспективними з погляду криптографічної стійкості виявилися getrandom() та OpenSSL DRBG — як через свою здатність блокуватися при нестачі ентропії, так і завдяки відповідності сучасним міжнародним стандартам безпеки. Проведене експериментальне дослідження підтвердило, що сучасні програмні генератори випадкових чисел в операційній системі Linux — зокрема getrandom(), /dev/urandom та DRBG у складі бібліотеки OpenSSL — здатні забезпечити високий рівень випадковості, необхідний для криптографічних застосувань. Проте між ними спостерігаються помітні відмінності як у внутрішній структурі, так і в поведінці за умов зміненої ентропійної ситуації та різного рівня навантаження системи. Генератор getrandom() продемонстрував збалансовану комбінацію надійності, стабільності та продуктивності. Його блокуюча поведінка в разі нестачі ентропії є критично важливою для генерації ключів та інших безпекових параметрів у ранній фазі завантаження системи. При цьому він забезпечив

стабільно високу ентропію вихідної послідовності та відмінні результати в усіх наборах тестів, що відповідає вимогам сучасних криптографічних стандартів. Генератор /dev/urandom підтвердив свою ефективність у загальному випадку, особливо для задач, де потрібна висока швидкість і невибагливість до стану системного пулу. Однак результати аналізу вказують на потенційні ризики при використанні /dev/urandom у критичних криптографічних контекстах, особливо в момент, коли рівень ентропії в системі ще не стабілізований. Незначні відхилення в χ^2-розподілах, коливання ентропійності в перших мегабайтах послідовностей і менш стабільне проходження глибоких тестів NIST свідчать про доцільність обережного ставлення до цього джерела при генерації довгострокових ключів або одноразових токенів. Найкращі результати за всіма критеріями показав криптографічний генератор DRBG із складу OpenSSL. Завдяки чітко структурованому підходу до ініціалізації, внутрішньому оновленню стану та застосуванню криптографічних примітивів (AES-CTR, SHA-256), цей механізм не лише забезпечив найвищий рівень ентропії та повну відсутність провалів у тестах Dieharder і NIST, а й показав найвищу продуктивність. Він підтвердив відповідність рекомендаціям NIST SP 800-90A, а його архітектура дозволяє використання у середовищах з високими вимогами до паралельності, масштабованості та криптографічної строгості.

СПИСОК ВИКОРИСТАНОЇ ЛІТЕРАТУРИ

1. NIST Special Publication 800-22 Rev. 1a. A Statistical Test Suite for Random and Pseudorandom Number Generators for Cryptographic Applications [Електронний ресурс]. — National Institute of Standards and Technology, 2010. — Режим доступу: https://csrc.nist.gov/publications/detail/sp/800-22/rev-1a/final (дата звернення: 02.05.2025).

2. NIST Special Publication 800-90A Rev. 1. Recommendation for Random Number Generation Using Deterministic Random Bit Generators (DRBG) [Електронний ресурс]. — NIST, 2015. — 121 с. — Режим доступу: https://csrc.nist.gov/publications/detail/sp/800-90a/rev-1/final (дата звернення: 02.05.2025).

3. Eastlake D., Crocker S., Schiller J. RFC 4086: Randomness Requirements for Security [Електронний ресурс] // IETF. — 2005. — Режим доступу: https://tools.ietf.org/html/rfc4086 (дата звернення: 05.05.2025).

4. Dieharder: A Random Number Test Suite [Електронний ресурс]. — Режим доступу: https://webhome.phy.duke.edu/~rgb/General/dieharder.php (дата звернення: 02.05.2025).

5. ENT – A Pseudorandom Number Sequence Test Program [Електронний ресурс]. — Режим доступу: http://www.fourmilab.ch/random/ (дата звернення: 05.05.2025).

6. OpenSSL Documentation: RAND and DRBG API Reference [Електронний ресурс]. — Режим доступу: https://www.openssl.org/docs/ (дата звернення: 10.05.2025).

7. Linux Kernel Documentation – Random Number Generation Interface [Електронний ресурс]. — Режим доступу: https://www.kernel.org/doc/html/latest/ (дата звернення: 23.04.2025).

8. ISO/IEC 18031:2011. Information technology – Security techniques – Random bit generation.

9. AIS 20/31. Anwendungshinweise und Interpretationen zum Schema: Empfehlungen zur Bewertung der Entropiequellen und der Zufallszahlengeneratoren

[Електронний ресурс] // Bundesamt für Sicherheit in der Informationstechnik (BSI), 2020. — Режим доступу: https://www.bsi.bund.de (дата звернення: 15.05.2025).

10. Schneier B. Applied Cryptography: Protocols, Algorithms, and Source Code in C. — 2nd ed. — New York: Wiley, 1996. — 784 с.

11. Ferguson N., Schneier B., Kohno T. Cryptography Engineering. — Indianapolis: Wiley Publishing, 2010. — 384 с.

12. Мазуренко А.І., Литвиненко С.В. Основи криптографії. — К.: Наукова думка, 2020. — 416 с.

13. Кузьменко М.В., Матвєєв С.В., Гапон І.В. Захист інформації в комп’ютерних системах: навч. посіб. — Х.: ХНУРЕ, 2021. — 308 с.

14. Man-pages Linux: getrandom(2), random(4), urandom(4), proc(5) [Електронний ресурс]. — Режим доступу: https://man7.org/linux/man-pages/ (дата звернення: 27.04.2025).

15. RFC 6979. Deterministic Usage of the Digital Signature Algorithm (DSA) and Elliptic Curve Digital Signature Algorithm (ECDSA) [Електронний ресурс]. — Режим доступу: https://tools.ietf.org/html/rfc6979 (дата звернення: 02.05.2025).

16. OWASP Cryptographic Storage Cheat Sheet [Електронний ресурс]. — Режим доступу: https://cheatsheetseries.owasp.org/cheatsheets/Cryptographic_Storage_Cheat_Sheet.html (дата звернення: 010.05.2025).

17. Debian Package Repository – dieharder, rng-tools, haveged [Електронний ресурс]. — Режим доступу: https://packages.debian.org/ (дата звернення: 15.05.2025).

ДОДАТОК А

```
#define _GNU_SOURCE

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <sys/random.h>

#include <time.h>

#include <fcntl.h>

#include <string.h>

#include <errno.h>


#define BLOCK_SIZE 4096

#define TOTAL_SIZE 10485760  // 10 MB


// Функція для зчитування ентропії з /proc

int read_entropy_available() {

    FILE *fp = fopen("/proc/sys/kernel/random/entropy_avail", "r");

    if (!fp) {

        perror("Не вдалося відкрити entropy_avail");

        return -1;

    }
```

```
    int entropy;
    fscanf(fp, "%d", &entropy);
    fclose(fp);
    return entropy;
}

// Основна програма
int main() {
    unsigned char buffer[BLOCK_SIZE];
    ssize_t bytes;
    FILE *outfile;
    int total_written = 0;

    struct timespec start, end;

    // Зчитуємо ентропію до початку
    int entropy_before = read_entropy_available();
    printf("Ентропія до генерації: %d\n", entropy_before);

    // Початок відліку часу
    clock_gettime(CLOCK_MONOTONIC, &start);
```

```
    outfile = fopen("gr_output.bin", "wb");
    if (!outfile) {
        perror("Не вдалося створити файл виводу");
        return 1;
    }

    while (total_written < TOTAL_SIZE) {
        size_t to_read = (TOTAL_SIZE - total_written) < BLOCK_SIZE ?
(TOTAL_SIZE - total_written) : BLOCK_SIZE;

        bytes = getrandom(buffer, to_read, 0);  // блокуючий виклик
        if (bytes < 0) {
            perror("Помилка при виклику getrandom()");
            fclose(outfile);
            return 1;
        }

        fwrite(buffer, 1, bytes, outfile);
        total_written += bytes;
    }
```

```
    fclose(outfile);

    // Кінець відліку часу
    clock_gettime(CLOCK_MONOTONIC, &end);

    // Зчитуємо ентропію після генерації
    int entropy_after = read_entropy_available();
    printf("Ентропія після генерації: %d\n", entropy_after);

    // Обчислюємо час виконання в мілісекундах
    long duration_ms = (end.tv_sec - start.tv_sec) * 1000 +
                       (end.tv_nsec - start.tv_nsec) / 1000000;
    printf("Загальний час виконання: %ld мс\n", duration_ms);

    return 0;
}
```

ДОДАТОК Б

```
#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <time.h>

#include <sys/stat.h>

#include <unistd.h>


#define BLOCK_SIZE 8192

#define TOTAL_SIZE 10485760  // 10 MB


// Функція зчитування ентропії з /proc

int read_entropy_available() {

    FILE *fp = fopen("/proc/sys/kernel/random/entropy_avail", "r");

    if (!fp) {

        perror("Не вдалося відкрити entropy_avail");

        return -1;

    }

    int entropy;

    fscanf(fp, "%d", &entropy);

    fclose(fp);
```

```
    return entropy;
}

// Створення каталогу з поточним timestamp
void make_output_dir(char *dirname, size_t size) {
    time_t t = time(NULL);
    struct tm *tm_info = localtime(&t);
    strftime(dirname, size, "urandom_output_%Y%m%d_%H%M%S", tm_info);
    mkdir(dirname, 0755);
}

int main() {
    unsigned char buffer[BLOCK_SIZE];
    size_t total_read = 0;

    // Фіксуємо ентропію до генерації
    int entropy_before = read_entropy_available();
    printf("Ентропія до генерації: %d\n", entropy_before);

    // Створюємо директорію з міткою часу
    char dirname[64];
```

```
    make_output_dir(dirname, sizeof(dirname));

    // Формуємо повний шлях до файлу виводу
    char filepath[128];
    snprintf(filepath, sizeof(filepath), "%s/ur_output.bin", dirname);

    FILE *urandom = fopen("/dev/urandom", "rb");
    FILE *outfile = fopen(filepath, "wb");

    if (!urandom || !outfile) {
        perror("Помилка відкриття файлів");
        return 1;
    }

    while (total_read < TOTAL_SIZE) {
        size_t to_read = (TOTAL_SIZE - total_read < BLOCK_SIZE) ? (TOTAL_SIZE - total_read) : BLOCK_SIZE;
        size_t read = fread(buffer, 1, to_read, urandom);

        if (read != to_read) {
            perror("Помилка зчитування з /dev/urandom");
```

```
            fclose(urandom);
            fclose(outfile);
            return 1;
        }

        fwrite(buffer, 1, read, outfile);
        total_read += read;
    }

    fclose(urandom);
    fclose(outfile);

    // Ентропія після генерації
    int entropy_after = read_entropy_available();
    printf("Ентропія після генерації: %d\n", entropy_after);
    printf("Файл збережено за шляхом: %s\n", filepath);

    return 0;
}
```

ДОДАТОК В

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
#include <openssl/rand.h>
#include <openssl/err.h>

#define TOTAL_SIZE 10485760  // 10 MB

// Зчитування поточного значення ентропії з /proc
int read_entropy_available() {
    FILE *fp = fopen("/proc/sys/kernel/random/entropy_avail", "r");
    if (!fp) {
        perror("Не вдалося відкрити entropy_avail");
        return -1;
    }
    int entropy;
    fscanf(fp, "%d", &entropy);
    fclose(fp);
```

```
    return entropy;
}

int main() {
    unsigned char *buffer = malloc(TOTAL_SIZE);
    if (!buffer) {
        fprintf(stderr, "Помилка виділення пам’яті\n");
        return 1;
    }

    // Фіксуємо ентропію до генерації
    int entropy_before = read_entropy_available();
    printf("Ентропія до генерації: %d\n", entropy_before);

    // Вимірюємо час
    struct timespec start, end;
    clock_gettime(CLOCK_MONOTONIC, &start);

    // Ініціалізація DRBG із AES-256 в CTR режимі
    RAND_DRBG *drbg = RAND_DRBG_new(NID_aes_256_ctr,
RAND_DRBG_FLAG_CTR_NO_DF, NULL);
```

```
if (!drbg) {
    fprintf(stderr, "Не вдалося створити DRBG\n");
    free(buffer);
    return 1;
}

if (RAND_DRBG_instantiate(drbg, NULL, 0) != 1) {
    fprintf(stderr, "Не вдалося ініціалізувати DRBG\n");
    RAND_DRBG_free(drbg);
    free(buffer);
    return 1;
}

// Генерація 10 МБ даних
if (RAND_DRBG_generate(drbg, buffer, TOTAL_SIZE, 0, NULL, 0) != 1) {
    fprintf(stderr, "Не вдалося згенерувати дані\n");
    RAND_DRBG_free(drbg);
    free(buffer);
    return 1;
}
```

```
RAND_DRBG_uninstantiate(drbg);

RAND_DRBG_free(drbg);


// Кінець вимірювання часу

clock_gettime(CLOCK_MONOTONIC, &end);


// Запис у файл

FILE *f = fopen("ssl_output.bin", "wb");

if (!f) {

    perror("Помилка створення файлу");

    free(buffer);

    return 1;

}


fwrite(buffer, 1, TOTAL_SIZE, f);

fclose(f);

free(buffer);


// Зчитування ентропії після

int entropy_after = read_entropy_available();

printf("Ентропія після генерації: %d\n", entropy_after);
```

```
    long duration_ms = (end.tv_sec - start.tv_sec) * 1000 +
                       (end.tv_nsec - start.tv_nsec) / 1000000;
    printf("Загальний час генерації: %ld мс\n", duration_ms);


    return 0;
}
```

ДОДАТОК Г - Bash-скрипт, який реалізує ENT-аналіз

```
#!/bin/bash

# Список файлів для тестування
FILES=("gr_output.bin" "ur_output.bin" "ssl_output.bin")

# Перевірка наявності утиліти ent
if ! command -v ent &> /dev/null; then
    echo "Утиліта 'ent' не встановлена. Встановіть її за допомогою: sudo apt install ent"
    exit 1
fi

# Створення каталогу для збереження результатів
mkdir -p ent_results
timestamp=$(date +"%Y%m%d_%H%M%S")

echo "Починається аналіз ENT для кожного генератора..."

for file in "${FILES[@]}"; do
    if [ -f "$file" ]; then
```

```
        out_file="ent_results/${file%.bin}_ent_${timestamp}.txt"
        echo "Аналізуємо $file ..."
        ent "$file" | tee "$out_file"
        echo "Результат збережено у $out_file"
    else
        echo "Файл $file не знайдено, пропущено."
    fi
done

echo "Усі доступні файли були проаналізовані утилітою ENT.
```

ДОДАТОК Д - Bash-скрипт для тестування з використанням утиліти Dieharder

```
#!/bin/bash

# Список файлів
FILES=("gr_output.bin" "ur_output.bin" "ssl_output.bin")

# Список тестів Dieharder (15 найбільш репрезентативних)
TEST_IDS=(0 1 2 3 4 5 6 7 8 9 10 11 12 13 14)

# Перевірка наявності Dieharder
if ! command -v dieharder &> /dev/null; then
    echo "Утиліта 'dieharder' не встановлена. Встановіть її за допомогою: sudo apt install dieharder"
    exit 1
fi

# Створення директорії для результатів
mkdir -p dieharder_results
timestamp=$(date +"%Y%m%d_%H%M%S")

echo "Запускається Dieharder-тестування..."
```

```
for file in "${FILES[@]}"; do
    if [ -f "$file" ]; then
        output="dieharder_results/${file%.bin}_dieharder_${timestamp}.txt"
        echo "Тестування $file ..."
        dieharder -a -g 201 -f "$file" > "$output"
        echo "Результати збережено в $output"
    else
        echo "Файл $file не знайдено, пропущено."
    fi
done

echo " Усі доступні файли пройшли тестування Dieharder."
```

ДОДАТОК Е - Завантаження і встановлення NIST STS

```
sudo apt update

sudo apt install build-essential unzip

wget https://www.itl.nist.gov/div897/ctg/sts/sts-2.1.2.zip

unzip sts-2.1.2.zip

cd sts-2.1.2

make
```

ДОДАТОК Ж - Підготовка бінарних файлів до аналізу (Файли мають бути в бітовому форматі — тобто 0/1, по 1 біту в байті. Для цього виконаємо конвертацію за допомогою Python)

```
# convert_to_bits.py

def bytes_to_bits(file_in, file_out):
    with open(file_in, 'rb') as f_in, open(file_out, 'w') as f_out:
        while byte := f_in.read(1):
            bits = format(ord(byte), '08b')
            f_out.write(bits)


files = ["gr_output.bin", "ur_output.bin", "ssl_output.bin"]

for file in files:
    bits_file = file.replace(".bin", ".txt")
    bytes_to_bits(file, bits_file)
```

ДОДАТОК І - Запуск STS для кожного файлу

```
#!/bin/bash

FILES=("gr_output.txt" "ur_output.txt" "ssl_output.txt")
timestamp=$(date +"%Y%m%d_%H%M%S")

mkdir -p nist_results

cd sts-2.1.2

for file in "${FILES[@]}"; do
    if [ -f "../$file" ]; then
        echo "Копіюємо $file до data/"
        cp "../$file" data/
        filename="${file%.txt}"
        mkdir -p ../nist_results/$filename

        echo "Запускається STS для $file"
        ./assess 10000000 > ../nist_results/$filename/sts_result_${timestamp}.txt
        echo "Завершено: $file"
    else
```

```
        echo "$file не знайдено!"
    fi
done


cd ..
```