

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені Ігоря СІКОРСЬКОГО»
Навчально-науковий фізико-технічний інститут
Кафедра математичних методів захисту інформації

«На правах рукопису»
УДК УДК 004.056.5:004.8

«До захисту допущено»
Завідувач кафедри
_____ Сергій ЯКОВЛЄВ
«__» _____ 2025 р.

Дипломна робота
на здобуття ступеня бакалавра
за освітньо-професійною програмою
«Математичні методи криптографічного захисту інформації»

зі спеціальності: 113 Прикладна математика
на тему: «Дослідження застосування GAN мереж для
підбору паролів. Застосування GNPassGAN для зламу паролів
для неангломовних країн»

Виконав:
студент IV курсу, групи ФІ-13
Дідух Максим Андрійович _____

Керівник:
доц. каф. ММЗІ, к.т.н, доц.
Кучинська Наталія Вікторівна _____

Рецензент:
д-р філ., доц. каф. ММАД, ст. викл
Яйлимова Ганна Олексіївна _____

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

Студент _____

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені Ігоря СІКОРСЬКОГО»

Навчально-науковий фізико-технічний інститут
Кафедра математичних методів захисту інформації

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

ЗАТВЕРДЖУЮ

Завідувач кафедри

_____ Сергій ЯКОВЛЄВ

«__» _____ 2025 р.

ЗАВДАННЯ
на дипломну роботу

Студент: Дідух Максим Андрійович

1. Тема роботи: *«Дослідження застосування GAN мереж для підбору паролів. Застосування GNPassGAN для зламу паролів для неангломовних країн»*, науковий керівник дисертації: доц. каф. ММЗІ, к.т.н, доц. Кучинська Наталія Вікторівна,

затверджені наказом по університету №__ від «__» _____ 2025 р.

2. Термін подання студентом роботи: «__» _____ 2025 р.

3. Об'єкт дослідження: *генерація та підбір паролів з використанням генеративних змагальних мереж.*

4. Предмет дослідження: *особливості застосування GNPassGAN для генерування та аналізу кириличних паролів.*

5. Перелік завдань: *проаналізувати сучасні методи підбору паролів, зокрема імовірнісні моделі та моделі на основі глибокого навчання; дослідити архітектуру та принципи роботи PassGAN та GNPassGAN; перевірити роботу моделі GNPassGAN на кириличних паролях; перевірити згенеровані набори даних імовірнісними метриками, які не були наведені в оригінальному дослідженні; провести експериментальне дослідження ефективності GNPassGAN при генерації кириличних*

паролів; порівняти результати роботи GNPassGAN з кириличними паролями та латинськими паролями;

6. Орієнтовний перелік графічного (ілюстративного) матеріалу:
Презентація доповіді

7. Орієнтовний перелік публікацій: *планується доповідь на всеукраїнській конференції*)

8. Дата видачі завдання: 10 вересня 2024 р.

Календарний план

№ з/п	Назва етапів виконання магістерської дисертації	Термін виконання	Примітка
1	Узгодження теми роботи із науковим керівником	01-15 вересня 2024 р.	Виконано
2	Огляд опублікованих джерел за тематикою дослідження	Вересень-жовтень 2024 р.	Виконано
3	Аналіз структури PassGAN та GNPassGAN	Жовтень-листопад 2024 р.	Виконано
4	Підготовка наборів даних та тренування моделей	Листопад 2024 р. - січень 2025 р.	Виконано
5	Розробка метрик ефективності для натренованих моделей	Січень - лютий 2025 р.	Виконано
6	Проведення експериментів, обчислення метрик та аналіз результатів	Лютий - травень 2025 р.	Виконано
7	Оформлення дипломної роботи	Травень - червень 2025 р.	Виконано

Студент

_____ Максим ДІДУХ

Керівник

_____ Наталія КУЧИНСЬКА

РЕФЕРАТ

Кваліфікаційна робота містить: 102 стор., 47 рисунків, 3 таблиці, 25 джерел.

Дана робота присвячена дослідженню застосування генеративно-змагальних мереж (GAN) для підбору паролів, з особливим фокусом на кириличні паролі. Об'єктом дослідження є процес генерації та аналізу паролів за допомогою штучного інтелекту, а предметом – архітектура та ефективність GNPassGAN для роботи з кириличними паролями. Проаналізовано еволюцію методів від словникових атак до моделей глибинного навчання та досліджено архітектурні переваги GNPassGAN над PassGAN. Експериментальні результати демонструють високу ефективність GNPassGAN (до 13,56% точності вгадування з 25-кратним множителем для паролів фіксованої довжини). Встановлено, що транслітеровані кириличні паролі мають вдвічі вищу точність вгадування, ніж оригінальні кириличні, а також краще статистичне відтворення (дивергенція JS 0,09 проти 0,15). Виявлено певні обмеження моделі при роботі з кириличними паролями, що вказує на необхідність подальшого вдосконалення архітектури.

ГЕНЕРАТИВНІ ЗМАГАЛЬНІ МЕРЕЖІ, ПІДБІР ПАРОЛІВ,
GNPASSGAN, КИРИЛИЧНІ ПАРОЛІ, ТРАНСЛІТЕРАЦІЯ,
ДИВЕРГЕНЦІЯ ЄНСЕНА-ШЕННОНА, КРИПТОГРАФІЧНА БЕЗПЕКА

ABSTRACT

The qualification work contains: 102 pages, 47 figures, 3 tables, 25 sources.

This thesis investigates the application of Generative Adversarial Networks (GANs) for password cracking, with a special focus on Cyrillic passwords. The research object is the process of password generation and analysis using artificial intelligence, while the subject is the architecture and effectiveness of GNPassGAN for working with Cyrillic passwords. The evolution of methods from dictionary attacks to deep learning models is analyzed, and the architectural advantages of GNPassGAN over PassGAN are examined. Experimental results demonstrate high efficiency of GNPassGAN (up to 13.56% guessing accuracy with a 25x multiplier for fixed-length passwords). The study found that transliterated Cyrillic passwords have twice the guessing accuracy compared to original Cyrillic ones, as well as better statistical reproduction (JS divergence of 0.09 versus 0.15). Certain limitations of the model when working with Cyrillic passwords were identified, indicating the need for further architectural improvements.

GENERATIVE ADVERSARIAL NETWORKS, PASSWORD
GUESSING, GNPASSGAN, CYRILLIC PASSWORDS,
TRANSLITERATION, JENSEN-SHANNON DIVERGENCE,
CRYPTOGRAPHIC SECURITY

ЗМІСТ

Перелік умовних позначень, скорочень і термінів	9
Вступ.....	11
1 Застосування методів штучного інтелекту в криптографії.....	13
1.1 Використання штучного інтелекту в криптографії	13
1.1.1 Еволюційні обчислення для генерування алгоритмів шифрування.....	13
1.1.2 Нейронні мережі у криптографії	14
1.2 Геш-функції, їх застосування, сфера використання, сучасні стійкі геш-функції	16
1.2.1 Криптографічні геш-функції, їх основні властивості.....	16
1.2.2 Застосування геш-функцій	17
1.2.3 Сімейство SHA (Secure Hash Algorithm)	17
1.2.4 Геш-функції BLAKE2 і BLAKE3	18
1.2.5 Геш-функція Argon2.....	18
1.2.6 Демонстрація криптографічної стійкості сучасних геш-функцій.....	19
1.2.7 Атаки на геш-функції	20
1.2.8 Методи захисту від атак на геш-функції	21
1.3 Задача вгадування паролів, основні підходи до її вирішення.....	21
1.4 Моделі на основі правил (англ. Rule-based models)	23
1.5 Імовірнісні моделі	24
1.5.1 Марковська модель.....	24
1.5.2 Модель PCFG	25
1.5.3 Моделі глибинного навчання	25
Висновки до розділу 1.....	26
2 GAN мережі та приклади їх використання. Порівняльний аналіз PassGAN та GNPASSGAN	28
2.1 Огляд GAN мереж.....	28

2.1.1 Генератор	29
2.1.2 Дискримінатор	29
2.1.3 Двійковий класифікатор	30
2.1.4 Поширені випадки використання GAN мереж	30
2.1.5 Adobe Photoshop Neural Filters.....	33
2.2 GNPassGAN	35
2.2.1 Реалізації GAN мереж для вирішення задачі вгадування паролів, PassGAN та GNPassGAN	35
2.2.2 Архітектура PassGAN.....	36
2.2.3 Архітектура GNPassGAN.....	37
2.2.4 Ключові відмінності та переваги GNPassGAN	38
2.2.5 Опис експериментів	39
2.2.6 Порівняльні метрики	40
2.2.7 Аналіз результатів експерименту для PassGAN та GNPassGAN	40
2.2.8 Комбінації GAN-підходів з традиційними методами підбору паролів	41
Висновки до розділу 2.....	42
3 GNPassGAN для кириличних паролів: розширена методологія оцінки та експериментальні результати	44
3.1 Розгортання GNPassGAN в різних середовищах	44
3.1.1 Розгортання GNPassGAN в операційній системі Linux.....	45
3.1.2 Налаштування локального середовища	45
3.1.3 Розгортання GNPassGAN на хмарній платформі Vast.ai	46
3.2 Підготовка та аналіз наборів даних для тренування GNPassGAN	47
3.2.1 Джерела та характеристики наборів даних	48
3.2.2 Підготовка наборів даних.....	49
3.3 Аналіз процесу тренування	50
3.4 Опис експериментів та аналіз отриманих результатів	55
3.4.1 Методологія першого експерименту	56
3.4.2 Результати для наборів даних фіксованої довжини (Char10)	57

3.4.3 Результати для наборів даних змінної довжини (Char812) ..	8 58
3.4.4 Аналіз результатів	59
3.4.5 Методологія оцінки статистичних метрик	60
3.4.6 Дивергенція Єнсена-Шеннона	61
3.4.7 Розподіл довжини	62
3.5 Розподіл класів символів	62
3.6 Структурні патерни	63
3.7 Недоліки імплементації	64
Висновки до розділу 3	65
Висновки	68
Додаток А Тексти програм	70
A.1 preprocessing.py	70
A.2 probabilistic_models.py	75
A.3 transliterate_cyrillic.py	88
Додаток Б Отримані оцінки для статистичних метрик	93

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СКОРОЧЕНЬ І ТЕРМІНІВ

NIST — Національний інститут стандартів і технологій США (англ. U.S. National Institute of Standards and Technology)

RSA — асиметрична криптосистема запропонована Рівестом, Шаміром та Адлеманом (англ. Rivest-Shamir-Adelman) у 1977 р.

AES — покращений стандарт шифрування (англ. Advanced Encryption Standard) опублікований NIST у 2001 р.

SHA — сімейство геш-функцій розроблених NIST (англ. Secure Hash Algorithm)

ГПВЧ — генератор псевдовипадкових чисел

PCFG — імовірнісна контекстно-вільна граматика (англ. Probabilistic Context-Free Grammar)

LSTM — довга короткочасна пам'ять (англ. Long short-term memory)

LSTM-мережі — мережі з довгою короткочасною пам'яттю

GAN — генеративна змагальна мережа (англ. Generative Adversarial Network)

PassGAN — реалізація GAN мережі для задачі підбору паролів (англ. Password Generative Adversarial Network)

GNPassGAN — реалізація GAN мережі для задачі вгадування паролів з використанням градієнтної нормалізації (англ. Gradient Normalization Password Generative Adversarial Network)

Char10 — паролі довжиною до 10 символів (англ. Characters 10)

Char812 — паролі довжиною від 8 до 12 символів (англ. Characters 8-12)

FG — згенерований файл паролів (англ. generated password file)

FT — тестовий файл паролів (англ. testing password file)

JS — Єнсена-Шеннона (англ. Jensen-Shannon)

CPU — центральний процесор (англ. Central Processing Unit)

GPU — графічний процесор (англ. Graphics processing unit)

ВСТУП

Актуальність дослідження Паролі залишаються найпоширенішим методом автентифікації в сучасних інформаційних системах, незважаючи на розвиток альтернативних технологій. Проблема безпеки паролів набуває особливої актуальності в контексті зростання кількості кібератак та витоків даних. Генеративні змагальні мережі (GAN) продемонстрували значний потенціал у задачах генерації та аналізу паролів, проте їх ефективність щодо кириличних паролів залишається малодослідженою. Вивчення роботи GNPASSGAN з кириличними паролями дозволить розробити більш ефективні методи оцінки безпеки та захисту паролів для україномовних користувачів, що є надзвичайно актуальним в умовах зростаючих кіберзагроз.

Метою дослідження є вивчення роботи GNPASSGAN з кириличними паролями. Також дослідження спрямоване на аналіз ефективності генеративних змагальних мереж у задачі генерації та підбору кириличних паролів.

Задачі дослідження: проаналізувати сучасні методи підбору паролів, зокрема імовірнісні моделі та моделі на основі глибокого навчання; дослідити архітектуру та принципи роботи PassGAN та GNPASSGAN; перевірити роботу моделі GNPASSGAN на кириличних паролях; перевірити згенеровані набори даних імовірнісними метриками, які не були наведені в оригінальному дослідженні; провести експериментальне дослідження ефективності GNPASSGAN при генерації кириличних паролів; порівняти результати роботи GNPASSGAN з кириличними паролями та латинськими паролями;

Об'єктом дослідження є генерація та підбір паролів з використанням генеративних змагальних мереж.

Предметом дослідження є особливості застосування GNPASSGAN для генерування та аналізу кириличних паролів.

Наукова новизна полягає у проведеному дослідженні ефективності GNPassGAN для роботи з кириличними паролями. Запропоновано нові метрики оцінки якості генерації паролів, які дозволяють більш точно оцінювати ефективність моделей абстрагуючись від значень точності та унікальності.

Практичне значення отриманих результатів полягає у можливості їх використання для оцінки безпеки паролів у системах автентифікації.

1 ЗАСТОСУВАННЯ МЕТОДІВ ШТУЧНОГО ІНТЕЛЕКТУ В КРИПТОГРАФІЇ

У першому розділі даної роботи ми розглянемо застосування штучного інтелекту в криптографії. Дослідимо задачу вгадування паролів, розглянемо різні підходи до її вирішення — імовірнісні моделі (включаючи марковські моделі та PCFG), так і моделі на основі глибинного навчання.

1.1 Використання штучного інтелекту в криптографії

Взаємодія штучного інтелекту та криптографії є відносно новою областю досліджень, що має як перспективи, так і суттєві обмеження. Хоча алгоритми машинного навчання демонструють потенціал у певних криптографічних задачах, вони поки що виступають скоріше як допоміжний інструмент, а не як заміна традиційним методам. Дослідники активно вивчають можливості застосування штучного інтелекту в різних аспектах криптографії: від аналізу існуючих систем шифрування до оптимізації параметрів криптографічних примітивів. Розглянемо кілька напрямків, де методи штучного інтелекту починають знаходити практичне застосування, враховуючи як їхні переваги, так і наявні обмеження.

1.1.1 Еволюційні обчислення для генерування алгоритмів шифрування

Наукова робота [100] розглядає новий підхід до шифрування з використанням нейронних мереж та еволюційних обчислень. Автори

пропонують метод, який відходить від традиційних підходів до шифрування, генеруючи персоналізовані алгоритми шифрування, а не просто використовуючи персональні ключі з відомими алгоритмами.

Основна концепція полягає у використанні природних джерел шуму, таких як атмосферний шум від радіовипромінювання та радіоактивного розпаду, для навчання систем, які можуть генерувати унікальні алгоритми шифрування. Процес починається з подачі цих природних джерел шуму в систему, яка потім вчиться апроксимувати вхідний шум для створення нелінійних функцій. Ці функції згодом використовуються як ітератори і проходять ретельне тестування на криптографічну стійкість, включаючи перевірку на показники Ляпунова, рівні ентропії, довжину циклу і характеристики дифузії ключа.

Що робить цей підхід особливо цінним, так це його здатність генерувати необмежену кількість унікальних генераторів псевдовипадкових чисел (ГПВЧ), які можуть бути використані на індивідуальній основі. Це особливо актуально для сучасних додатків, де користувачі можуть отримати персоналізовані алгоритми шифрування, а не покладатися на стандартні алгоритми, які можуть бути вразливими до відомих алгоритмічних атак.

1.1.2 Нейронні мережі у криптографії

У статті[16], автор досліджує використання нейронних мереж в класичних криптографічних системах, зокрема RSA і AES. Оскільки дослідники продовжують вивчати нові підходи, штучний інтелект став цінним інструментом у розумінні та оцінці цих криптографічних стандартів.

У криптографічній системі RSA нейронні мережі продемонстрували неабиякий потенціал у кількох ключових сферах. Нейронні мережі можна навчити розпізнавати складні шаблони в процесах генерації ключів RSA, потенційно виявляючи слабкі місця, які традиційний аналіз може

пропустити. Наприклад моделі глибинного навчання можуть аналізувати великі масиви даних ключів RSA для виявлення статистичних закономірностей і потенційних вразливостей в алгоритмах генерації ключів. Алгоритми машинного навчання продемонстрували успіх в оптимізації вибору і реалізації параметрів RSA, що потенційно підвищує безпеку і продуктивність.

Ця сфера також відкриває нові підходи до криптоаналізу алгоритмів симетричного шифрування, тут будуть згадані тільки деякі результати для алгоритму AES, але слід зазначити, що аналогічні результати можна отримати і для інших алгоритмів. Нейронні мережі успішно застосовуються для аналізу процесів шифрування та дешифрування, намагаючись виявити тонкі закономірності або слабкі місця, які можуть бути використані. Методи машинного навчання виявилися особливо цінними при проведенні атак побічних каналів, коли системи штучного інтелекту аналізують найдрібніші зміни в енергоспоживанні, електромагнітних випромінюваннях і хронометражі під час операцій шифрування. Ці підходи показали багатообіцяючі результати в прогнозуванні ключових бітів AES. Дослідники також розробили моделі, здатні аналізувати статистичні властивості операцій AES, потенційно виявляючи раніше невідомі вразливості.

Хоча штучний інтелект продемонстрував значний потенціал у криптографічному аналізі, наразі він слугує скоріше додатковим інструментом, ніж заміною традиційним методам. Подальший розвиток його методів в криптографії відкриває нові можливості для майбутніх досліджень і розробок.

1.2 Геш-функції, їх застосування, сфера використання, сучасні стійкі геш-функції

Геш-функції є фундаментальним криптографічним примітивом, що широко використовуються в сучасних системах безпеки. Вони перетворюють вхідні дані довільної довжини у вихідне значення фіксованої довжини, яке називається геш-значенням.

Означення 1.1. Функція $H : \{0, 1\}^* \rightarrow \{0, 1\}^n$ - називається геш-функцією.

Означення 1.2. $\forall m \in \{0, 1\}^*, H(m) = h$ - називається геш-значенням функції H .

1.2.1 Криптографічні геш-функції, їх основні властивості

Серед множини усіх геш-функцій виділяють підмножину функцій, які можуть бути застосованими в криптографічних цілях. Такі геш-функції називають криптографічними. Криптографічні геш-функції повинні задовольняти наступним властивостям:

1) **Стійкість до пошуку прообразу (One-way):** маючи геш-значення h , має бути обчислювально неможливо знайти повідомлення m таке, що $H(m) = h$.

2) **Стійкість до колізій (Collision resistance):** має бути обчислювально неможливо знайти два різних повідомлення m_1 і m_2 такі, що $H(m_1) = H(m_2)$.

3) **Стійкість до пошуку другого прообразу (Second preimage resistance):** маючи повідомлення m_1 , має бути обчислювально неможливо знайти інше повідомлення m_2 таке, що $H(m_1) = H(m_2)$.

4) **Лавинний ефект (Avalanche effect):** незначна зміна вхідних даних повинна призводити до значної зміни вихідного геш-значення.

1.2.2 Застосування геш-функцій

Геш-функції мають широкий спектр застосувань:

- 1) **Зберігання паролів:** замість зберігання паролів у відкритому вигляді, системи зберігають їх геш-значення. При автентифікації обчислюється геш введеного пароля і порівнюється зі збереженим значенням.
- 2) **Цифрові підписи:** геш-функції використовуються для створення дайджесту повідомлення, який потім підписується за допомогою приватного ключа.
- 3) **Перевірка цілісності даних:** геш-значення файлу або повідомлення може використовуватися для перевірки, чи не були дані змінені.
- 4) **Структури даних:** геш-таблиці, Merkle дерева та інші структури даних використовують геш-функції для ефективного зберігання та пошуку інформації.
- 5) **Блокчейн-технології:** геш-функції є основою для створення блоків у блокчейні та забезпечення їх зв'язку.

1.2.3 Сімейство SHA (Secure Hash Algorithm)

SHA [19] є сімейством криптографічних геш-функцій, розроблених Національним інститутом стандартів і технологій США (NIST).

- 1) **SHA-1:** створює 160-бітний геш. Вважається застарілим через виявлені вразливості. У 2017 році було продемонстровано практичну атаку колізії.
- 2) **SHA-2:** включає SHA-224, SHA-256, SHA-384, SHA-512, SHA-512/224 і SHA-512/256. Ці функції вважаються безпечними і широко використовуються.

3) **SHA-3**: найновіше покоління, базується на алгоритмі Кессак¹. Включає SHA3-224, SHA3-256, SHA3-384 і SHA3-512. Розроблено як альтернатива SHA-2 з іншою внутрішньою структурою для забезпечення різноманітності криптографічних примітивів.

1.2.4 Геш-функції BLAKE2 і BLAKE3

BLAKE2 [20]— це геш-функція, розроблена як заміна MD5 і SHA-1, з акцентом на високу продуктивність. BLAKE3, представлений у 2020 році, є ще швидшою версією, що зберігає високий рівень безпеки.

1.2.5 Геш-функція Argon2

Argon2 [21] — це геш-функція, спеціально розроблена для гешування паролів. Вона перемогла в конкурсі Password Hashing Competition у 2015 році². Argon2 має три варіанти:

- 1) **Argon2d**: оптимізований для захисту від атак з використанням графічних процесорів.
- 2) **Argon2i**: оптимізований для захисту від атак за побічними каналами.
- 3) **Argon2id**: гібридний варіант, що поєднує властивості Argon2d і Argon2i.

Argon2 приймає кілька ключових параметрів, які дозволяють налаштувати баланс між безпекою та продуктивністю:

- 1) **Параметр часу (t)**: визначає кількість ітерацій алгоритму. Збільшення цього параметра підвищує обчислювальну складність і, відповідно, час, необхідний для обчислення геш-значення.
- 2) **Параметр пам'яті (m)**: визначає обсяг пам'яті (в кілобайтах), який використовується під час обчислення геш-значення. Цей параметр

¹<https://keccak.team/keccak.html>

²<https://eprint.iacr.org/2015/265>

особливо важливий для захисту від атак з використанням спеціалізованого обладнання, такого як ASIC або FPGA.

3) **Параметр паралелізму (p)**: визначає кількість паралельних потоків, які використовуються під час обчислення. Збільшення цього параметра дозволяє ефективніше використовувати багатоядерні процесори.

4) **Довжина вихідного значення**: може бути налаштована відповідно до вимог безпеки, зазвичай від 16 до 32 байтів.

5) Та інші параметри, які наразі не настільки важливі: сіль, секретний ключи та асоційовані дані.

Рекомендовані значення параметрів залежать від конкретного сценарію використання, але загальні рекомендації включають використання параметра пам'яті не менше 4096 КБ та параметра часу не менше 3 для забезпечення достатнього рівня безпеки в більшості застосувань.

1.2.6 Демонстрація криптографічної стійкості сучасних геш-функцій

Криптографічна стійкість геш-функцій може бути продемонстрована через кілька ключових властивостей:

1) **Обчислювальна складність пошуку прообразу**: для сучасних геш-функцій, таких як SHA-256, BLAKE2 та Argon2, не існує відомих методів знаходження прообразу, ефективніших за повний перебір. Для 256-бітного геш-значення це означає 2^{256} можливих варіантів, що робить атаку перебором практично неможливою з сучасними обчислювальними ресурсами.

2) **Стійкість до колізій**: для SHA-3 та BLAKE3 не виявлено практичних атак на пошук колізій. Для порівняння, SHA-1 вважається вразливим після демонстрації практичної атаки колізії у 2017 році

командою Google та CWI Amsterdam, яка потребувала приблизно 2^{63} ²⁰ операцій, що значно менше теоретичного порогу 2^{80} для 160-бітної геш-функції.

Для Argon2, як спеціалізованої функції гешування паролів, додатковим показником безпеки є параметризована обчислювальна складність. Наприклад, при налаштуванні Argon2id з параметрами часу $t=3$, пам'яті $m=4096$ КБ та паралелізму $p=1$, генерація одного геш-значення на сучасному процесорі займає приблизно 100 мс, що робить масовий перебір надзвичайно ресурсомістким. Практичні тести показують, що атакуючий з доступом до 1000 графічних процесорів зможе перевірити лише близько 10 мільйонів комбінацій на день, що робить перебір паролів з високою ентропією практично неможливим.

Ці властивості разом забезпечують високий рівень безпеки сучасних криптографічних геш-функцій для різноманітних застосувань, від зберігання паролів до забезпечення цілісності даних та цифрових підписів.

1.2.7 Атаки на геш-функції

1) **Атака грубою силою (Brute force attack):** перебір усіх можливих вхідних значень для знаходження прообразу.

2) **Атака за словником (Dictionary attack):** використання попередньо обчислених геш-значень для поширених вхідних даних.

3) **Атака райдужними таблицями (Rainbow table attack):** використання попередньо обчислених ланцюжків геш-значень для прискорення пошуку прообразу.

4) **Атака за побічними каналами (Side-channel attack):** використання інформації про фізичну реалізацію геш-функції.

5) **Криптоаналітичні атаки:** використання математичних властивостей геш-функції для знаходження колізій або прообразів.

1.2.8 Методи захисту від атак на геш-функції

1) **Солі (Salting)**: додавання випадкових даних до вхідного повідомлення перед гешуванням для захисту від атак за словником і райдужними таблицями.

2) **Перець (Peppering)**: додавання секретного значення до вхідного повідомлення перед гешуванням.

3) **Ключовий розтяг (Key stretching)**: багаторазове застосування геш-функції для збільшення обчислювальної складності атаки.

4) **Функції, що потребують значної пам'яті (Memory-hard functions)**: назва говорить сама за себе — використання функцій, які вимагають значного обсягу пам'яті для обчислення, що ускладнює паралельні атаки.

1.3 Задача вгадування паролів, основні підходи до її вирішення

Паролі є найбільш поширеним способом аутентифікації, імовірно за все, через те, що їх легко запам'ятати та реалізувати. Незважаючи на цінність даних, доступ до яких надає кожен конкретний пароль, середньостатистичний користувач все ще використовує відносно прості паролі, що мають семантичний зміст та легко запам'ятовуються. Все це робить користувацькі паролі все більш вразливими до загроз реального світу. Посилаючись на дослідження Пірмана [12], 40% користувачів використовують один і той же пароль на декількох різних платформах. Існує велика кількість різних технік для вгадування паролів, всі вони націлюються на взлам як можна більшої кількості паролів за найменшу кількість спроб. Виділяють два основних типи атак - онлайн та оффлайн.

Означення 1.3. Оффлайн атака - атака під час якої, атакуючий якимось чином отримати певну кількість криптографічних гешів паролів

користувачів та намагається відновити їх вгадуючи та тестуючи велику кількість паролів.

Означення 1.4. Онлайн атака - атака під час якої, атакуючий робить спроби вгадати пароль через web-інтерфейс або додаток.

Зауваження. Слід зазначити, що онлайн атаки є більше обмеженими тому, що більшість web-інтерфейсів та додатків, обмежують можливість надсилання запитів після певної кількості невдалих спроб. Таких обмежень не може бути у випадку оффлайн атак, де атакуючий ніяк не взаємодіє з іншими сервісам, виконуючи вгадування в ізольованому середовищі.

До того ж існують й інші властивості атак, які визначають кількість інформації про користувача, що має атакуючий. Тут також є два основних типи - тралення (англ. trawling) та націлене вгадування.

Означення 1.5. Націлене вгадування виникає тоді, коли атакуючий намагається дізнатись пароль користувача використовуючи будь-які дані, що мають відношення до конкретного користувача.

Ванг [13] запропонував класифікувати таку інформацію на дві категорії в залежності від ступеня конфіденційності. Перший тип - це особиста інформація користувача, що надає змогу ідентифікувати його, включаючи ім'я, електронну пошту, тощо. Другий тип - це ідентифікаційні дані користувача, які є частково публічними (наприклад, ім'я користувача) і частково приватними (наприклад, пароль).

Означення 1.6. Тралення (англ. trawling) - атака, під час якої, атакуючий намагається знайти користувача, що відповідає уже відомому паролю.

Зауваження. Більшість користувачів не заслуговують націлених атак, як правило кількість ресурсів які атакуючий витратить на націлену атаку перевищить кількість ресурсів, які він отримає після успішного взламу. Сконцентрованої уваги заслуговують лише спеціальні

користувачі, що відносяться до критичної інфраструктури, фінансових установ або документів.

Далі наведені деякі з відомих типів атак. Слід зазначити, що всі вони використовують атаки типу вичерпного офлайн перебору (англ. trawling offline guessing), коли атакуючий якимось чином здобув доступ до бази даних, що містить геші паролів.

1.4 Моделі на основі правил (англ. Rule-based models)

На просторах мережі Інтернет можна зайти величезну кількість викрадених паролів, ось найбільші з них:

Джерело	К-ть паролів
phpBB	3.0×10^5
Yahoo	4.4×10^5
Rock You	1.4×10^7
Myspace	5.5×10^4
SkullSecurityComp	6.7×10^6
LinkedIn	1.3×10^6

Таблиця 1.1 – Відомі відкриті бази паролів

Велика кількість викрадених паролів справжніх користувачів сильно спрощує вивчення та збір патернів паролів. Маючи таку велику вибірку можна створити нові паролі-кандидати, використовуючи наявні як шаблон. Прикладами таких реалізацій є John The Ripper³ та Hashcat⁴, ці програми реалізують велику варіативність методів взламу паролів, таких як прямий перебір, атаки з використанням словників та моделі на основі правил, яка є найшвидшою серед усіх атак.

Зауваження. Системи на основі правил генерують паролі виключно базуючись на вже відомих правилах, а створення нових правил

³<https://github.com/openwall/john>

⁴<https://github.com/hashcat/hashcat>

є складною задачею, що вимагає великого рівня експертизи.

Як наслідок, паролі, для яких складно побудувати правила будуть взламуватись набагато довше, якщо будуть взагалі.

1.5 Імовірнісні моделі

Імовірнісні моделі представляють собою клас методів для аналізу та генерації паролів, що базуються на статистичних закономірностях у навчальних даних. Ці моделі використовують математичний апарат теорії ймовірностей для оцінки вірогідності появи певних послідовностей символів у паролях. На відміну від простих словникових атак, імовірнісні моделі здатні генерувати нові паролі, які не містяться в навчальному наборі, але відповідають статистичним патернам реальних паролів. Далі ми розглянемо два основних підходи: марковські моделі, що базуються на послідовних залежностях між символами, та моделі на основі імовірнісних контекстно-вільних граматики (PCFG), які враховують структурні особливості паролів. Обидва підходи мають свої переваги та обмеження.

1.5.1 Марковська модель

Марковська модель являє собою підхід до підбору паролів, що походить від принципів мовного моделювання та була представлена Нараньяном [3]. Ця модель передбачає наступні символи в послідовності на основі попередніх символів, причому довжина контексту, відома як «порядок», відіграє вирішальну роль в її ефективності. Еволюція моделі зазнала значних покращень завдяки дослідженням Ма [2], які запровадили складні методи, такі як нормалізація кінцевих символів та згладжування Лапласа. Нормалізація кінцевих символів працює шляхом додавання певного символу до паролів, гарантуючи, що розподіли

ймовірностей зберігають математичну узгодженість. Згладжування за Лапласом вирішує проблему надмірної підгонки шляхом введення дельта-значення до підрахунку підрядків, що особливо корисно для марковських моделей вищих порядків, які обробляють більш обширну контекстну інформацію.

1.5.2 Модель PCFG

Модель на основі імовірнісних контекстно-вільних граматик (PCFG), являє собою складний підхід до аналізу паролів за допомогою структурної декомпозиції. Ця модель ґрунтується на фундаментальній передумові, що паролі можна розбити на незалежні шаблонні структури, кожна з яких містить окремі термінали. Модель обчислює ймовірності паролів шляхом множення ймовірностей структур і відповідних їм терміналів. Наприклад, пароль типу «goskyou123» аналізується шляхом розбиття його на окремі структури («L7» для семи літер і «D3» для трьох цифр) і відповідні їм закінчення («goskyou» і «123»). Можливості моделі було значно розширено завдяки кільком інноваціям, включаючи інтеграцію пінїнь⁵ для аналізу китайських паролів у роботі Лі [1], а також включення шаблонів клавіатури та багатослівних шаблонів [5]. Ма [2] продемонстрував підвищену точність вгадування, проаналізувавши шаблони кінцевих частот у навчальних даних.

1.5.3 Моделі глибинного навчання

На відміну від моделей на основі правил та імовірнісних моделей вгадування паролів, методи глибинного навчання не роблять жодних припущень щодо структури паролів. Множини паролів згенеровані таким методом не обмежуються лише певною підмножиною відомих паролів.

⁵Пінїнь, повна офіційна назва Ханьюй пінїнь — найпоширеніший стандарт латинізування китайської мови, тобто позначення звуків китайської за допомогою латинської абетки.[17]

Нейронні мережі, особливо мережі з довгою короткочасною пам'яттю (LSTM-мережі) [6], представляють собою нову технологію для вирішення задачі підбору паролів. Ці мережі є обчислювальними моделями, які імітують біологічні нейронні мережі, пропонуючи складні можливості апроксимації функцій. Нейромережевий підхід долає суттєве обмеження марковських моделей - обмеження контексту фіксованої довжини - шляхом збереження довгострокових залежностей у даних. Меліхер [4] вперше застосував цей підхід, використовуючи LSTM-мережі для вилучення та прогнозування ознак паролів, хоча їхня оцінка була обмежена обмеженими структурами та даними [7]. З тих пір ця область розвивалася за допомогою різних нових підходів, включаючи впровадження Хітай [8] генеративних змагальних мереж (GAN) для вгадування паролів, хоча це вимагало більше спроб вгадування, ніж моделі на основі LSTM. Лю [10] пішли далі, створивши гібридний підхід, який поєднав правила PCFG з LSTM-мережами, показавши значні покращення. Однак їхнє тестування було обмежене 10^{12} вгадуваннями, тоді як реальні сценарії тралення в автономному режимі часто вимагають до 10^{16} спроб. Сюй [11] також реалізував мережі LSTM, але обмежив їх оцінку до 10^{10} вгадувань, не проводячи комплексної оцінки.

Висновки до розділу 1

У даному розділі було розглянуто застосування штучного інтелекту в криптографії, з особливим фокусом на задачу підбору паролів. Проведений аналіз показав еволюцію методів від простих rule-based підходів до складних моделей глибокого навчання.

Було досліджено різні підходи до вирішення задачі вгадування паролів, включаючи традиційні імовірнісні моделі, такі як марковські ланцюги та PCFG. Ці методи вже показали свою ефективність, але мають обмеження, пов'язані з необхідністю явного визначення правил та структур паролів. Марковські моделі обмежені фіксованою довжиною

контексту, а PCFG вимагають попереднього визначення граматичних правил.

2 GAN МЕРЕЖІ ТА ПРИКЛАДИ ЇХ ВИКОРИСТАННЯ.

ПОРІВНЯЛЬНИЙ АНАЛІЗ PASSGAN ТА GNPASSGAN

В другому розділі даної роботи особливу увагу буде приділено огляду GAN мереж та їх типові випадки використання, після чого перейдемо до головного об'єкту дослідження — реалізацій GAN мереж, що були оптимізованою для задачі підбору паролів (PassGAN та її покращену версію GNPassGAN). Порівняємо структуру PassGAN та GNPassGAN, проаналізуємо експерименти, метрики для їх порівняння. Ознайомимось з гібридними підходами до розв'язку задачі підбору паролів, які поєднують в собі імовірнісні моделі та реалізації на основі GAN-мереж.

2.1 Огляд GAN мереж

Генеративні змагальні мережі (GAN), представлені Гудфеллоу у 2014 році [14], представляють новий підхід до генеративного моделювання з використанням глибинного навчання. Архітектура реалізує змагальну структуру, де дві нейронні мережі змагаються одна проти одної в грі на мінімакс¹. Перша мережа, відома як генератор (G), створює синтетичні дані, намагаючись імітувати реальний розподіл даних, тоді як друга мережа, дискримінатор (D), має на меті розрізняти реальні та згенеровані дані.

Ключова ідея GAN полягає в тому, що генератор приймає випадковий шум з латентного простору як вхідний сигнал і виробляє синтетичні зразки. Одночасно дискримінатор функціонує як двійковий класифікатор, виводячи ймовірність того, що будь-який вхідний сигнал є справжнім, а не згенерованим. Завдяки цьому змагальному процесу

¹Мінімакс (англ. minimax, нім. minimax n) — правило прийняття рішень, що використовується в теорії ігор та теорії прийняття рішень для мінімізації можливих втрат з тих, які особа, яка приймає рішення не може уникнути при розвитку подій за найгіршим для неї сценарієм.[18]

навчання, обидві мережі постійно вдосконалюють свої можливості: генератор покращує свою здатність створювати більш переконливі синтетичні дані, тоді як дискримінатор вдосконалюється у їх виявленні.

2.1.1 Генератор

Генератор (G) у GAN архітектурі виконує ключову роль створення синтетичних даних. Його основна функція полягає в перетворенні випадкового шуму з латентного простору у дані, що імітують реальний розподіл. У контексті паролів генератор намагається створювати послідовності символів, які виглядають як справжні паролі користувачів.

Математично функцію генератора можна представити як:

$$G(z) \rightarrow \hat{x} \quad (2.1)$$

де z — випадковий вектор шуму, а $\hat{x}$ — згенерований зразок (пароль).

З точки зору архітектури, генератор зазвичай реалізується як нейронна мережа з кількома прихованими шарами. У PassGAN та GNPASSGAN генератор використовує згорткові шари для обробки послідовностей символів [8]. Під час навчання генератор оптимізується для максимізації ймовірності того, що дискримінатор помилково класифікує згенеровані зразки як справжні.

2.1.2 Дискримінатор

Дискримінатор (D) функціонує як двійковий класифікатор, який оцінює, чи походить вхідний зразок з реального розподілу даних чи був створений генератором [22]. У контексті генерації паролів дискримінатор аналізує структуру, частоту символів та інші характеристики паролів, щоб відрізнити справжні паролі від згенерованих.

Математично дискримінатор можна представити як:

$$D(x) \rightarrow [0,1] \quad (2.2)$$

де x — вхідний зразок (пароль), а вихід — ймовірність того, що x є справжнім зразком.

У PassGAN та GNPASSGAN дискримінатор реалізований як згорткова нейронна мережа, що аналізує послідовності символів на різних рівнях абстракції [8]. Він навчається максимізувати правильну класифікацію як справжніх, так і згенерованих зразків.

2.1.3 Двійковий класифікатор

Термін "двійковий класифікатор" у контексті GAN мереж відноситься до функціональної ролі дискримінатора. Дискримінатор виконує бінарну класифікацію, розділяючи вхідні дані на два класи: справжні (клас 1) та згенеровані (клас 0) [22].

Ця класифікація є критичною для змагального процесу навчання GAN. Цільова функція для навчання може бути представлена як:

$$\min_G \max_D V(D, G) = \mathbb{E}_{x \sim p_{\text{data}}} [\log D(x)] + \mathbb{E}_{z \sim p_z} [\log(1 - D(G(z)))] \quad (2.3)$$

де перший член максимізує ймовірність правильної класифікації справжніх зразків, а другий член мінімізує ймовірність неправильної класифікації згенерованих зразків [14].

2.1.4 Поширені випадки використання GAN мереж

GAN мережі знайшли застосування у різних галузях. У сфері комп'ютерного зору такі архітектури, як StyleGAN [15], зробили революцію у створенні синтетичних зображень, створюючи

фотореалістичні людські обличчя, які майже неможливо відрізнити від справжніх фотографій. Медична галузь використовує GAN для створення синтетичних даних медичних зображень, вирішення проблем конфіденційності та доповнення обмежених наборів даних для рідкісних захворювань.

Ще одним значним проривом стало перетворення тексту в зображення: такі моделі, як DALL-E² і Stable Diffusion³, використовують архітектуру, натхненну GAN, для створення зображень з текстових описів. В аудіо області GAN дозволили синтезувати голос і створювати музику, в той час як в обробці відео вони сприяли прогнозуванню кадрів і поліпшенню якості відео [123].

²<https://openai.com/index/dall-e/>

³<https://stablediffusion.com/>

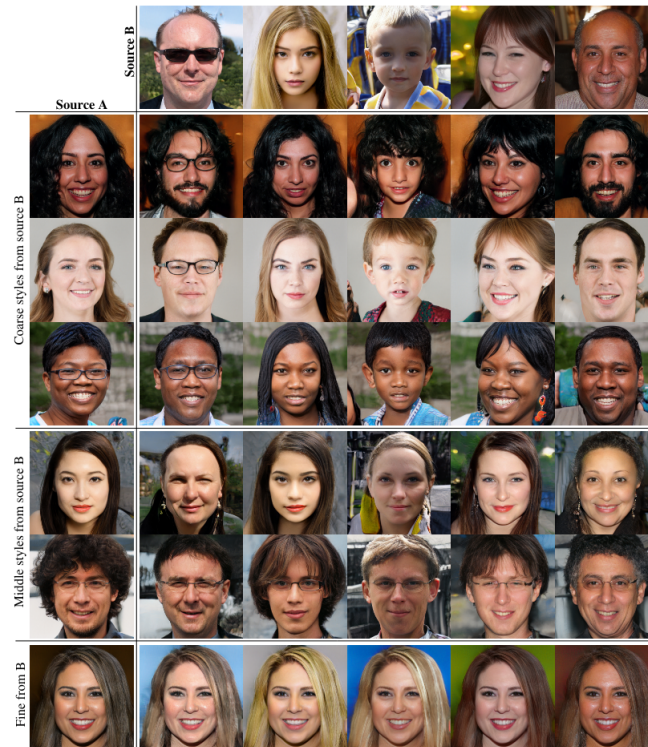


Рисунок 2.1 – Демонстрація керованого перенесення стилів у StyleGAN: зображення створені шляхом комбінування різних стилів із двох джерел (А і В). Копіювання грубих стилів (42 – 82) переносить позу та форму обличчя, середніх (162 – 322) — дрібніші риси, а тонких (642 – 10242) — кольорову схему та мікроструктуру [15]

2.1.5 Adobe Photoshop Neural Filters

Одним із практичних прикладів застосування генеративних змагальних мереж є Neural Filters (Нейронні фільтри) в Adobe Photoshop ⁴. Запроваджені у 2020 році, цей інструмент використовує GAN та інші нейронних мереж для виконання складних маніпуляцій із зображеннями, які раніше вимагали годин ручної роботи або були взагалі неможливими.

Нейронні фільтри поділяються на дві категорії: бета-фільтри, які все ще перебувають у розробці, та представлені фільтри, які пройшли повне тестування. Серед найпопулярніших можна виділити:

1) **Smart Portrait**: дозволяє змінювати вираз обличчя, вік, напрямок погляду, інтенсивність емоцій та навіть зачіску людини на фотографії. Цей фільтр використовує GAN для генерації реалістичних модифікацій обличчя, зберігаючи ідентичність людини ⁵.

2) **Style Transfer**: перетворює звичайні фотографії на зображення, що імітують стиль відомих художників або конкретні художні стилі, використовуючи архітектуру, подібну до StyleGAN ⁶.

3) **Skin Smoothing**: виявляє та згладжує недосконалості шкіри, зберігаючи природну текстуру та деталі обличчя ⁷.

4) **Depth Blur**: створює ефект глибини різкості, автоматично визначаючи різні плани зображення за допомогою нейронних мереж, що аналізують глибину сцени ⁸.

Технічно ці фільтри працюють завдяки різним типам генеративних мереж, включаючи Creative Adversarial Networks (CAN), які є надбудовою над GAN мережами. На відміну від класичних GAN, мережі CAN не лише намагаються створити реалістичні зображення, але й прагнуть генерувати творчий контент, що відхиляється від встановлених стилістичних норм. Як описано в дослідженні Ельгамалія [23], CAN

⁴<https://helpx.adobe.com/photoshop/using/neural-filters.html>

⁵<https://www.photoshopessentials.com/photo-editing/use-smart-portrait-in-photoshop-to-change-your-subjects-look/>

⁶<https://nerdschalk.com/how-to-transfer-styles-between-images-with-photoshop-style-transfer/>

⁷<https://www.photoshopessentials.com/photo-editing/smooth-skin/>

⁸<https://photoshop-tutorial.org/basics/depth-blur-filter/>

використовують подвійну систему зворотного зв'язку: одна спрямована на створення реалістичних зображень, а інша — на максимізацію стилістичної забарвленості та відхилення від відомих художніх стилів. Саме цей принцип лежить в основі фільтрів стилізації Adobe, які здатні створювати унікальні художні інтерпретації фотографій, а не просто імітувати існуючі стилі.



Рисунок 2.2 – Приклад зміни емоції на фото за допомогою Adobe Neural Filters

Важливо зазначити, що Adobe використовує хмарні обчислення для обробки більшості нейронних фільтрів, оскільки вони вимагають значних обчислювальних ресурсів. Це дозволяє користувачам із менш потужними комп'ютерами отримати доступ до передових можливостей штучного інтелекту.

Нейронні фільтри Adobe Photoshop демонструють, як технології GAN переходять від дослідницьких лабораторій до комерційних

продуктів, роблячи складні маніпуляції із зображеннями доступними для широкого кола користувачів. Це також піднімає важливі етичні питання щодо автентичності зображень та потенційного зловживання такими технологіями, що спонукало Adobe впровадити систему Content Credentials для відстеження походження та модифікацій зображень.

2.2 GNPassGAN

Тепер розглянемо новий підхід до вгадування паролів з використанням вдосконалених генеративних змагальних мереж. GNPassGAN представляє собою значний крок вперед у застосуванні глибинного навчання для аналізу та генерації паролів, пропонуючи покращену архітектуру в порівнянні з попередніми підходами. Ми детально розглянемо теоретичні основи генеративних змагальних мереж (GANs), їх практичне застосування в різних областях, а також специфіку їх адаптації для задачі вгадування паролів. Особливу увагу буде приділено порівняльному аналізу PassGAN та його вдосконаленої версії GNPassGAN, включаючи детальний розбір експериментальних результатів та метрик ефективності. Цей розділ надасть комплексне розуміння того, як сучасні методи машинного навчання можуть бути ефективно застосовані для вирішення складних задач у сфері криптографії та кібербезпеки.

2.2.1 Реалізації GAN мереж для вирішення задачі вгадування паролів, PassGAN та GNPassGAN

Застосування генеративних змагальних мереж для вгадування паролів відкрило новий напрямок у розвитку методів тестування безпеки паролів. На відміну від традиційних підходів, таких як словникові та імовірнісні атаки, GAN мережі здатні вивчати та відтворювати складні

патерни в реальних паролях, генеруючи при цьому нові, потенційно валідні комбінації. Це особливо важливо в контексті сучасних вимог до безпеки, де паролі стають все складнішими та різноманітнішими. Розглянемо детально архітектури двох ключових реалізацій цього підходу.

2.2.2 Архітектура PassGAN

PassGAN, запропонована Хітаджем [8], базується на архітектурі SeqGAN [24] з певними модифікаціями для роботи з текстовими паролями. Основні компоненти архітектури PassGAN включають:

1) **Генератор**: використовує рекурентну нейронну мережу (RNN) з LSTM-блоками для генерації послідовностей символів. Структура генератора:

- вхідний шар для обробки випадкового шуму з латентного простору
- два прихованих LSTM-шари з 128 нейронами кожен
- вихідний шар з softmax-активацією для генерації символів
- розмір прихованого стану: 128

2) **Дискримінатор**: використовує згорткову нейронну мережу (CNN) для класифікації паролів. Структура дискримінатора:

- вхідний шар для обробки послідовностей символів
- три згорткові шари з фільтрами розміром 1, 2 і 3 відповідно
- шар об'єднання (max-pooling)
- повнозв'язний шар з 128 нейронами
- вихідний шар з сигмоїдною активацією

3) **Функція втрат**: використовує стандартну функцію втрат GAN:

$$\min_G \max_D V(D, G) = \mathbb{E}_{x \sim p_{\text{data}}} [\log D(x)] + \mathbb{E}_{z \sim p_z} [\log(1 - D(G(z)))] \quad (2.4)$$

4) **Навчання**: використовує політику градієнтного спуску з методом REINFORCE для навчання генератора, оскільки дискретна

природа текстових даних ускладнює пряме зворотнє поширення помилки.

PassGAN обробляє паролі як послідовності символів, де кожен символ представлений як one-hot вектор. Модель обмежена роботою з паролями фіксованої максимальної довжини (10 символів у початковій реалізації).

2.2.3 Архітектура GNPassGAN

GNPassGAN, представлена в роботі [9], є вдосконаленою версією PassGAN з кількома ключовими архітектурними модифікаціями:

1) **Генератор:** замість RNN використовує трансформерну архітектуру, що дозволяє краще обробляти довгострокові залежності в даних:

- вхідний шар для обробки випадкового шуму
- шар кодування для збереження інформації про порядок символів
- чотири блоки трансформера з механізмом самоуваги (self-attention)
- розмір прихованого стану: 256 (збільшено порівняно з PassGAN)
- вихідний шар з softmax-активацією

2) **Дискримінатор:** використовує гібридну архітектуру CNN-Transformer:

- вхідний шар для обробки послідовностей символів
- шар позиційного кодування
- паралельні згорткові шари з різними розмірами фільтрів (1, 2, 3, 4,

5)

- два блоки трансформера з механізмом самоуваги
- шар глобального усереднення (global average pooling)
- повнозв'язний шар з 256 нейронами
- вихідний шар з сигмоїдною активацією

3) **Функція втрат:** використовує модифіковану функцію втрат

Wasserstein GAN з градієнтним штрафом (WGAN-GP):

$$L = \mathbb{E}_{\tilde{x} \sim P_g} [D(\tilde{x})] - \mathbb{E}_{x \sim P_r} [D(x)] + \lambda \mathbb{E}_{\hat{x} \sim P_{\hat{x}}} [(||\nabla_{\hat{x}} D(\hat{x})||_2 - 1)^2] \quad (2.5)$$

де λ - коефіцієнт штрафу градієнта (зазвичай 10).

4) **Нормалізація**: використовує шари нормалізації спектра (spectral normalization) в дискримінаторі для стабілізації навчання.

5) **Обробка змінної довжини**: на відміну від PassGAN, GNPASSGAN здатна ефективно обробляти паролі змінної довжини (від 8 до 12 символів) завдяки використанню механізму маскування в трансформерній архітектурі.

6) **Розширений словник символів**: підтримує розширений набір символів, включаючи спеціальні символи та символи різних мовних груп.

2.2.4 Ключові відмінності та переваги GNPASSGAN

Порівняно з PassGAN, GNPASSGAN має кілька суттєвих переваг:

1) **Покращена здатність моделювання довгострокових залежностей**: механізм самоуваги в трансформерній архітектурі дозволяє моделі ефективніше вивчати взаємозв'язки між віддаленими символами в паролі, що особливо важливо для довгих паролів.

2) **Стабільніше навчання**: використання WGAN-GP та нормалізації спектра значно покращує стабільність процесу навчання, зменшуючи проблему колапсу режимів (mode collapse), яка часто спостерігається в стандартних GAN.

3) **Краща масштабованість**: паралельна природа трансформерів дозволяє ефективніше обробляти великі набори даних та генерувати більші обсяги паролів.

4) **Гнучкість щодо довжини паролів**: здатність обробляти паролі змінної довжини робить GNPASSGAN більш практичною для реальних сценаріїв, де паролі мають різну довжину.

5) **Зменшення дублікатів:** архітектурні вдосконалення та модифікована функція втрат сприяють генерації більш різноманітних паролів, зменшуючи кількість дублікатів на 31,69% для паролів фіксованої довжини та на 61,80% для паролів змінної довжини.

Ці архітектурні вдосконалення пояснюють значне покращення продуктивності GNPassGAN порівняно з PassGAN, що підтверджується експериментальними результатами, які будуть наведені нижче. Зокрема, використання трансформерної архітектури та механізму самоуваги дозволяє моделі краще вивчати складні патерни в паролях, а модифікована функція втрат WGAN-GP забезпечує стабільніше навчання та генерацію більш різноманітних паролів.

2.2.5 Опис експериментів

У дослідженні [9] модель GNPassGAN порівнюється виключно з PassGAN, яка вже продемонструвала перевагу над традиційними методами підбору паролів. Для порівняння використовується набір даних rockyou⁹ для навчання, а також набір даних phpbb¹⁰ і окрема частина rockyou для тестування. Вони провели експерименти у двох категоріях довжини паролів: паролі до 10 символів (Char10), що є поширеним явищем у дослідженнях паролів, та паролі від 8 до 12 символів (Char812), що краще відображає реальні вимоги у сучасному світі.

Набори даних були спочатку дедубліковані, а потім випадковим чином розділені на навчальні та тестові набори у співвідношенні 80:20, щоб забезпечити відсутність перетину між ними. І PassGAN, і GNPassGAN пройшли 200 000 навчальних ітерацій зі збереженням параметрів моделі кожні 10 000 ітерацій. У той час як початкова робота PassGAN [8] тестувала лише паролі довжиною до 10 символів, включення тестування на Char812 допомагає оцінити продуктивність на більш

⁹<http://downloads.skullsecurity.org/passwords/rockyou.txt.bz2>

¹⁰<https://github.com/danielmiessler/SecLists/blob/master/Passwords/Leaked-Databases/phpbb.txt>

складних паролів.

2.2.6 Порівняльні метрики

Формула точності збігу:

$$\text{Count}(\text{set}(FG) \cap \text{set}(FT)) / \text{Count}(\text{set}(FT)), \text{ де:}$$

FG — згенерований файл паролів (generated password file)
 FT — тестовий файл паролів (testing password file)

Ця метрика показує відсоток реальних паролів, успішно згенерованих при тестуванні на заданому наборі даних

2.2.7 Аналіз результатів експерименту для PassGAN та GNPASSGAN

Результати експерименту демонструють значну різницю в продуктивності між PassGAN та GNPASSGAN в різних сценаріях. Для набору даних Char10 GNPASSGAN показав значне покращення порівняно з PassGAN.

При генерації 10^8 паролів, GNPASSGAN досягнув:

- точність збігу 12.647% у порівнянні з 6.726% у PassGAN
- на 88,03% більше успішних збігів паролів, ніж PassGAN
- на 31,69% менше дублікатів паролів, що свідчить про кращу генеративну різноманітність

Масштабування продуктивності було особливо помітним при різних обсягах генерації. Зі збільшенням кількості згенерованих паролів зі 10^4 до 10^8 обидві моделі показали кращу точність збігів, але GNPASSGAN зберігав свою кращу продуктивність протягом усього часу. При 10^7 згенерованих паролів GNPASSGAN досягла точності збігу 4,258% порівняно з 2,038% у PassGAN.

Для сценарію Char812 (паролі від 8 до 12 символів) розрив між моделями став ще більш помітним. При генерації 10^8 паролів GNPassGAN продемонстрував:

- точність підбору 5,4927% проти 0,7777% у PassGAN
- у шість разів більше успішних збігів паролів ніж PassGAN
- на 61,80% менше дублікатів

2.2.8 Комбінації GAN-підходів з традиційними методами підбору паролів

Хоча GNPassGAN демонструє значні покращення порівняно з PassGAN, дослідники виявили, що комбінування нейромережових підходів з традиційними методами може дати ще кращі результати. У роботі [9] автори експериментували з гібридними підходами, поєднуючи GNPassGAN з Hashcat.

Hashcat, як один з найпотужніших інструментів для підбору паролів на основі правил, має свої сильні сторони у використанні відомих шаблонів та трансформацій. Водночас, GAN-моделі краще виявляють неявні патерни та генерують нові варіації паролів. Поєднання цих підходів дозволяє використати переваги обох методів.

Експерименти показали, що при послідовному застосуванні Hashcat та GNPassGAN можна досягти:

- збільшення точності збігу на 15-20% порівняно з використанням лише GNPassGAN
- розширення покриття різних типів паролів, включаючи як структуровані (які добре підбираються правилами Hashcat), так і нестандартні (які краще генеруються GAN-моделями)
- зменшення загального часу, необхідного для досягнення заданого рівня успішних збігів

Особливо ефективною виявилася стратегія, коли спочатку

застосовувався Hashcat для підбору паролів з чіткою структурою, а потім GNPassGAN для генерації більш складних та нестандартних варіантів. Такий підхід дозволив досягти точності збігу понад 18% при генерації 10^8 паролів для набору даних Char10, що значно перевищує результати окремого використання кожного з методів.

Ідея комбінування різних підходів до підбору паролів не є новою. Різні дослідники експериментували з гібридними методами, хоча більшість попередніх робіт фокусувалася на комбінуванні традиційних підходів або використанні ранніх версій GAN-моделей. Унікальність підходу з GNPassGAN полягає в тому, що він поєднує найсучаснішу GAN-архітектуру з перевіреними традиційними методами, демонструючи синергетичний ефект.

Ці результати підкреслюють важливість комплексного підходу до аналізу безпеки паролів, де різні методи можуть доповнювати один одного, компенсуючи індивідуальні обмеження кожного з них. Майбутні дослідження могли б зосередитися на розробці інтегрованих систем, які автоматично визначають оптимальну стратегію комбінування різних методів підбору паролів залежно від характеристик цільового набору даних.

Висновки до розділу 2

У даному розділі було проведено детальний огляд генеративних змагальних мереж (GAN) та їх застосування для вирішення задачі підбору паролів. Детальний аналіз архітектур PassGAN та GNPassGAN показав значний прогрес у цьому напрямку. GNPassGAN, завдяки своїм архітектурним вдосконаленням, демонструє кращі результати порівняно з попередником:

- підвищення точності збігу на 88,03% для паролів фіксованої довжини
- зменшення кількості дублікатів на 31,69%

– значне покращення результатів для паролів змінної довжини (від 8 до 12 символів)

Важливим спостереженням є те, що сучасні методи штучного інтелекту дозволяють не лише відтворювати існуючі патерни паролів, але й генерувати нові, потенційно валідні комбінації. Це має важливе значення як для оцінки безпеки існуючих систем аутентифікації, так і для розробки більш надійних методів захисту паролів.

Особливо показовим є еволюція GAN-підходів від PassGAN до GNPassGAN. Заміна рекурентної архітектури на трансформерну з механізмом самоуваги дозволила значно покращити здатність моделі вивчати довгострокові залежності в паролях. Використання модифікованої функції втрат WGAN-GP та нормалізації забезпечило стабільніше навчання та зменшення кількості дублікатів у згенерованих паролях.

Найбільш перспективним напрямком виявилися гібридні підходи, що поєднують переваги традиційних методів (Hashcat) з нейромережевими (GNPassGAN). Такі комбінації дозволяють досягти значно кращих результатів, ніж кожен метод окремо, забезпечуючи як широке покриття структурованих паролів, так і здатність виявляти нестандартні патерни. Експерименти показали, що послідовне застосування Hashcat та GNPassGAN може підвищити точність збігу на 15-20%, досягаючи показників понад 18% при генерації 10^8 паролів.

Результати досліджень підкреслюють необхідність постійного вдосконалення методів захисту паролів, оскільки технології їх зламу продовжують розвиватися. Водночас, розуміння цих методів допомагає розробляти більш ефективні рекомендації щодо створення надійних паролів та вдосконалювати системи аутентифікації.

3 GNPASSGAN ДЛЯ КИРИЛИЧНИХ ПАРОЛІВ: РОЗШИРЕНА МЕТОДОЛОГІЯ ОЦІНКИ ТА ЕКСПЕРИМЕНТАЛЬНІ РЕЗУЛЬТАТИ

У даному розділі описано підхід до розгортання, тренування та оцінки ефективності GNPassGAN з особливим акцентом на роботі моделі з кириличними паролями. Розглянуто методологію налаштування середовища як на локальних системах, так і на орендованих хмарних ресурсах, що дозволило провести масштабні експерименти з різними наборами даних. Запропоновано розширену систему статистичних метрик для комплексної оцінки якості генерованих паролів, включаючи аналіз структурних патернів та дивергенцію Єнсена-Шеннона для різних характеристик паролів. Особливу увагу приділено порівнянню ефективності моделі при роботі з оригінальними кириличними паролями та їх транслітерованими варіантами, що дозволило виявити важливі закономірності та обмеження у застосуванні генеративних моделей для різних алфавітів.

3.1 Розгортання GNPassGAN в різних середовищах

Важливим етапом дослідження було розгортання та тестування архітектури GNPassGAN для розуміння її можливостей та обмежень. Процес включав налаштування середовища як на локальній машині, так і на орендованому сервері, що дозволило провести оцінку продуктивності моделі в різних умовах, оптимізувати процес тренування та виявити певні проблеми в реалізації.

3.1.1 Розгортання GNPassGAN в операційній системі Linux

Процес розгортання GNPassGAN на локальній машині з операційною системою Linux вимагав налаштування програмного середовища та залежностей. Хоча описані нижче кроки специфічні для Linux, аналогічний підхід може бути адаптований для інших операційних систем з відповідними модифікаціями. Локальне розгортання дозволило провести перше тренування моделі та оцінити базову продуктивність перед переходом до більш масштабних експериментів на орендованих системах.

3.1.2 Налаштування локального середовища

Першим етапом було клонування репозиторію GNPassGAN з GitHub¹ та створення ізольованого віртуального середовища для уникнення конфліктів між залежностями. Особливу складність представляла необхідність використання TensorFlow 1.x, який має суттєві відмінності від сучасної версії TensorFlow 2.x. Для вирішення цієї проблеми було застосовано Anaconda², що дозволило створити окреме середовище з необхідними версіями бібліотек.

Процес налаштування середовища включав наступні кроки:

```
1 conda create -n gn_pass_gan python=3.8
2 conda activate gn_pass_gan
3 conda install pip
4 conda install cudatoolkit=10.0
5 pip3 install -r requirements.txt
```

Після успішного налаштування середовища було проведено тренування базової моделі на локальній машині, що дозволило оцінити вимоги до системи та часові затрати на тренування. Для запуску процесу

¹<https://github.com/fangyiyu/GNPassGAN>

²<https://www.anaconda.com/download>

тренування слід виконати наступну команду:

```
1 python3 models.py --training-data [path_to_training_data] --output-dir [
    path_to_output_dir] --iters=[num_of_iterations] --save-every=[value]
```

Цей етап зайняв декілька днів, включаючи два дні безпосередньо на тренування моделі використовуючи NVIDIA QUADRO P1000 4GB³, що підтвердило необхідність пошуку більш ефективних обчислювальних ресурсів для подальших експериментів.

3.1.3 Розгортання GNPassGAN на хмарній платформі Vast.ai

Враховуючи значні обчислювальні вимоги для тренування GNPassGAN, було прийнято рішення дослідити можливості використання хмарних платформ для оренди обчислювальних ресурсів. Після аналізу доступних варіантів вибір зупинився на платформі Vast.ai⁴, яка пропонує доступ до серверів з відеокартами за доступними цінами та без проблем з доступом до ресурсів.

На відміну від великих хмарних провайдерів, таких як Google Cloud, AWS та Microsoft Azure, які часто мають обмежену доступність відеокарт та вимагають тривалого очікування або значних авансових платежів, Vast.ai забезпечує швидкий доступ до необхідних ресурсів. Це особливо актуально в контексті поточного неймовірно високого попиту на обчислювальні ресурси для задач штучного інтелекту.

Для підключення до Vast.ai необхідно виконати наступні кроки:

- 1) Зареєструватися на платформі та поповнити баланс
- 2) Згенерувати SSH-ключ на локальній машині та додати публічний ключ в розділі 'Keys' на платформі
- 3) Обрати відповідний сервер з необхідними характеристиками в розділі 'Search'

³<https://www.nvidia.com/content/dam/en-zz/Solutions/design-visualization/productspage/quadro/quadro-desktop/quadro-pascal-p1000-data-sheet-us-nv-704475-r1.pdf>

⁴<https://vast.ai/>

4) Орендувати сервер та підключитися до нього за допомогою SSH-команди, яка надається в розділі 'Instances'

Типова команда для підключення має наступний формат:

```
1 ssh -p 62433 root@156.238.224.242 -L 8080:localhost:8080
```

Для експериментів було обрано сервер з графічним прискорювачем NVIDIA RTX 4090 24GB⁵, через оптимальне співвідношення ціни та продуктивності для моєї задачі тренування мереж середнього масштабу. Після підключення до сервера процес налаштування середовища аналогічний описаному в розділі 3.1.2.

Використання хмарної платформи дозволило прискорити процес тренування моделей, вивільнити персональну систему під інші задачі та провести більшу кількість експериментів з різними конфігураціями та наборами даних, що було зайняло в рази більше часу при використанні лише локальних ресурсів. Для порівняння, на орендованій системі мені вдалось паралельно тренувати 8 моделей одночасно. Все це через те, що поточна реалізація процесу тренування не використовує усієї обчислювальної потужності наявної відеокарти. У випадку NVIDIA RTX 4090, підчас тренування однієї моделі, графічний процесор навантажується лише на 5-10%. Ця думка буде розвинена нижче в окремому розділі, де будуть наведені ще деякі проблеми з оригінальною реалізацією GNPassGAN.

3.2 Підготовка та аналіз наборів даних для тренування GNPassGAN

Після того, як середовище було налаштовано, слід було знайти та обробити набори тренувальних даних. Для забезпечення різноманітності було обрано декілька відомих наборів паролів з різних джерел, що дозволило оцінити ефективність моделі на різних типах даних, зокрема

⁵<https://www.nvidia.com/en-us/geforce/graphics-cards/40-series/rtx-4090/>

один з наборів містить переважно паролі кирилицею.

3.2.1 Джерела та характеристики наборів даних

Для тренування моделей було обрано п'ять наборів даних з різними характеристиками⁶:

1) **RockYou** — приблизно 14 мільйонів паролів, отриманих внаслідок витоку даних з популярного сервісу RockYou у 2009 році. Цей набір широко використовується в дослідженнях безпеки паролів та містить переважно англійські паролі.

2) **Gmail** — близько 5 мільйонів паролів, пов'язаних з обліковими записами Gmail. Цей набір представляє паролі користувачів популярного поштового сервісу та може містити більш різноманітні паттерни.

3) **Yahoo** — майже півмільйона паролів з витоку даних Yahoo. Цей набір менший за обсягом, але представляє іншу користувацьку базу.

4) **S** — близько 5 мільйонів паролів користувачів поштового сервісу, що активно використовується в країні-агресорці.

5) **Cyrillic** - 182 мільйони паролів, переважна більшість яких є паролі кирилицею.

Набір даних S представляє особливий інтерес, оскільки дозволяє дослідити, як GNPassGAN справляється з паролями, що містять транслітерацію неанглійських слів. Враховуючи тезу з розділу 1.3 про те, що більшість користувачів створюють паролі з семантичним змістом, можна очікувати, що значна частина паролів у цьому наборі буде містити транслітерацію слів мови країни агресора латиницею. З аналогічних причин, набір Cyrillic також є надзвичайно цікавим. Результати роботи моделей натренованих на цих наборах дозволять оцінити здатність GNPassGAN адаптуватися до різних мовних патернів.

З таблиці 3.1 видно, що більшість паролів із справжніх наборів даних мають довжину 8-12 символів.

⁶<https://downloads.skullsecurity.org/passwords/passwords/>

Джерело	К-ть паролів	< 8 символів	8-12 символів	= 10 символів	> 12 символів
RockYou	14×10^6	33.03%	59.91%	14.04%	7.06%
gmail	4.9×10^6	36.34%	61.60%	13.85%	2.05%
yahoo	4.4×10^5	34.73%	63.86%	12.37%	1.41%
S	4.6×10^6	39.65%	53.66%	8.70%	6.69%
cyrillic	182.2×10^6	41.42%	52.59%	13.42%	5.99%

Таблиця 3.1 – Порівняння розподілу даних в необроблених наборах базуючись на довжині паролів

3.2.2 Підготовка наборів даних

Підготовка даних для тренування моделей включала декілька етапів обробки:

1) **Дедуплікація** — видалення повторюваних паролів для уникнення зміщення в розподілі даних та підвищення ефективності тренування.

2) **Фільтрація за довжиною** — відбір паролів відповідно до вимог експериментів:

– для експериментів з паролями фіксованої довжини (Char10) — паролі довжиною до 10 символів

– для експериментів з паролями змінної довжини (Char812) — паролі довжиною від 8 до 12 символів

3) **Нормалізація** — приведення паролів до єдиного формату, включаючи видалення невідтримуванних символів та обробку спеціальних випадків.

Програмний код для обробки наборів даних реалізовано на мові Python та наведено в додатку А.1.

Обробка набору Cyrillic дещо відрізнялась, після кроків дедуплікації та фільтрації за довжиною, було відфільтровано лише паролі, що містять кирилицю, цифри та спецсимволи. Далі всі паролі було транслітеровано латиницею. В результаті було отримано 4 набори даних - char10 та char812

паролів лише з кирилицею, та char10 і char812 тих же паролів, але вже транслітерованих латиницею. Функцію для транслітерації також наведено в додатку А.3.

Далі кожен набір даних було випадковим чином розділено на дві частини - тренувальний та тестовий набори, у співвідношенні 80 : 20 відповідно.

Таблиця 3.2 – Статистика наборів даних фіксованої довжини (Char10)

Набір даних	Загальна кількість	Унікальних	Відсоток унікальних
RockYou (трен.)	1,611,238	1,611,238	100%
RockYou (тест)	402,810	402,810	100%
yahoo (трен.)	39,280	39,280	100%
yahoo (тест)	9,820	9,820	100%
S (трен.)	220,195	220,195	100%
S (тест)	55,048	55,048	100%
gmail (трен.)	444,910	444,910	100%
gmail (тест)	111,227	111,227	100%
Cyrillic (трен.)	19,638,339	19,638,337	100%
Cyrillic (тест)	4,903,645	4,903,645	100%
Cyrillic трансліт. (трен.)	19,638,335	19,638,335	100%
Cyrillic трансліт. (тест)	4,903,644	4,903,644	100%

3.3 Аналіз процесу тренування

Процес тренування GNPassGAN був відстежений, на рисунках 3.1, 3.2, 3.3, 3.4, 3.5, 3.6 та 3.7 представлені ключові метрики, отримані протягом 200 000 ітерацій тренування. Для кожної моделі було отримано схожі графіки, тому ми розглянемо їх лише для однієї моделі.

Метрики JS1 та JS2 (рис. 3.1 та 3.2), які представляють собою альтернативні варіанти обчислення дивергенції Єнсена-Шеннона для оцінки якості генеративної моделі. Ці метрики вимірюють статистичну

Таблиця 3.3 – Статистика наборів даних змінної довжини (Char812)

Набір даних	Загальна кількість	Унікальних	Відсоток унікальних
RockYou (трен.)	6,874,744	6,874,744	100%
RockYou (тест)	1,718,686	1,718,686	100%
Yahoo (трен.)	190,043	190,043	100%
Yahoo (тест)	47,510	47,510	100%
S (трен.)	1,344,445	1,344,445	100%
S (тест)	336,111	336,111	100%
gmail (трен.)	1,718,299	1,718,299	100%
gmail (тест)	429,574	429,574	100%
Cyrillic (трен.)	76,922,275	76,922,261	100%
Cyrillic (тест)	19,224,689	19,224,688	100%
Cyrillic трансліт. (трен.)	76,922,264	76,922,264	100%
Cyrillic трансліт. (тест)	19,224,685	19,224,685	100%

відстань між розподілами реальних та згенерованих паролів, де нижчі значення вказують на кращу здатність генератора створювати паролі, подібні до справжніх.

Метрика JS1 (рис. 3.1) демонструє значну варіативність протягом усього процесу тренування, коливаючись переважно в діапазоні від 0.01 до 0.06, з окремими піками до 0.10. Така поведінка свідчить про постійну динамічну адаптацію генератора до розпізнавальних здібностей дискримінатора. Незважаючи на коливання, загальний тренд залишається стабільним, що вказує на відсутність деградації моделі протягом тренування.

Метрика JS2 (рис. 3.2) демонструє більш виражену динаміку зміни. На початкових етапах тренування (до 25000 ітерацій) спостерігається різке зниження значень від пікових 0.30 до приблизно 0.15, після чого відбувається поступове зниження до рівня 0.08-0.10 з періодичними коливаннями. Така поведінка є типовою для успішного процесу навчання GAN-моделей і свідчить про поступове покращення якості генерованих паролів та наближення їх розподілу до розподілу реальних даних.

Графіки JS3 та JS4 (рис. 3.3 та 3.4) відображають значення

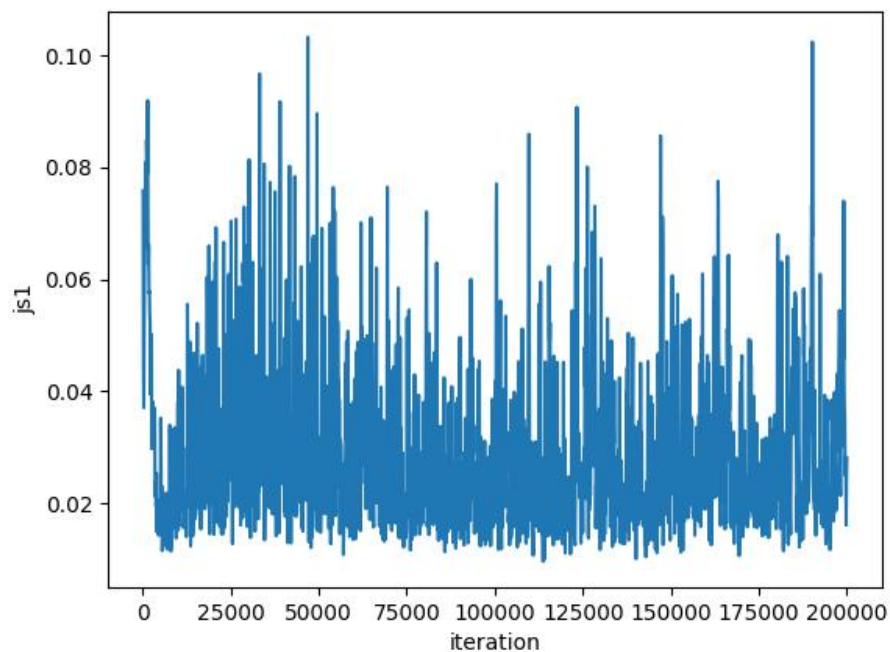


Рисунок 3.1 – Значення метрики JS1 під час тренування

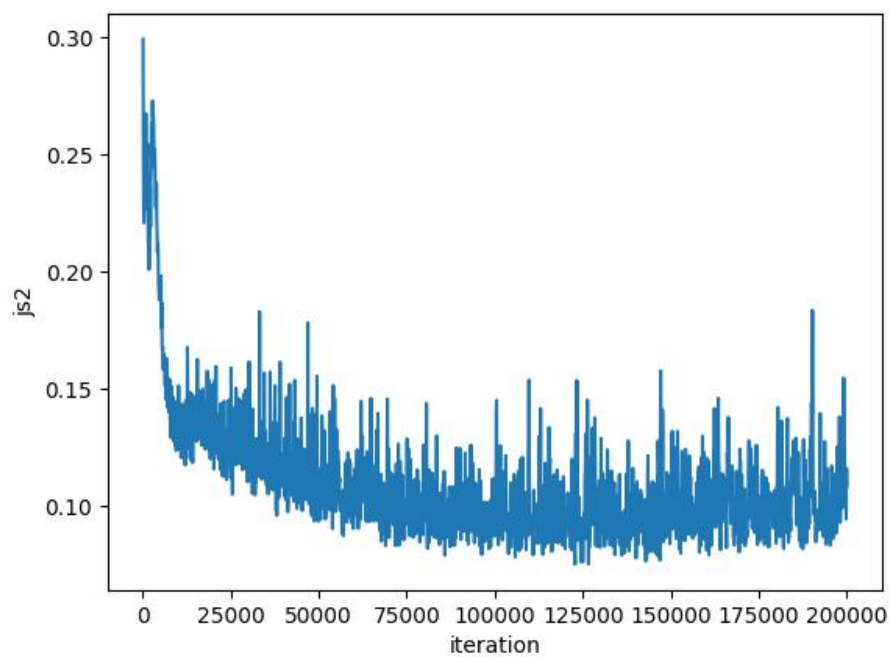


Рисунок 3.2 – Значення метрики JS2 під час тренування

дивергенції Єнсена-Шеннона, що є важливим показником якості навчання GAN-моделей. Обидва показники демонструють характерну поведінку: спочатку спостерігається різке зростання (до значень 0.59 для JS3 та 0.74 для JS4), після чого відбувається стабілізація. JS3 стабілізується приблизно на рівні 0.30, а JS4 - на рівні 0.55-0.60. Така динаміка свідчить про успішне досягнення балансу між генератором та дискримінатором.

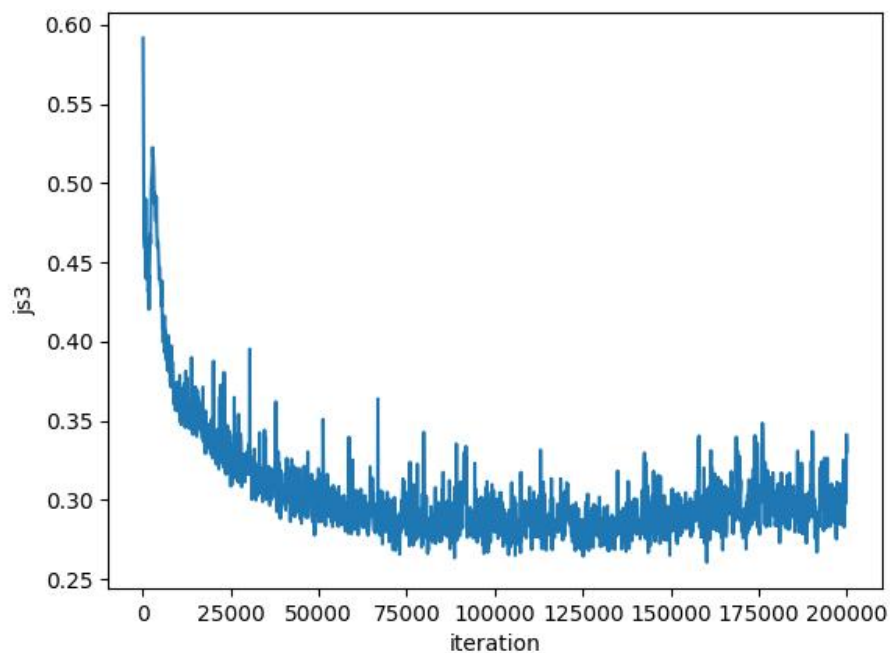


Рисунок 3.3 – Значення дивергенції Єнсена-Шеннона (JS3) під час тренування

Функції втрат дискримінатора та генератора (рис. 3.5 та 3.6) також демонструють характерну поведінку для стабільного навчання GAN. Втрати дискримінатора після початкових коливань стабілізуються в діапазоні 0.95-1.15, що вказує на здатність дискримінатора ефективно розрізняти справжні та згенеровані паролі. Втрати генератора стабілізуються в діапазоні 0.80-1.00, що свідчить про постійне вдосконалення здатності генератора створювати правдоподібні паролі.

Графік часу виконання ітерацій (рис. 3.7) демонструє періодичні

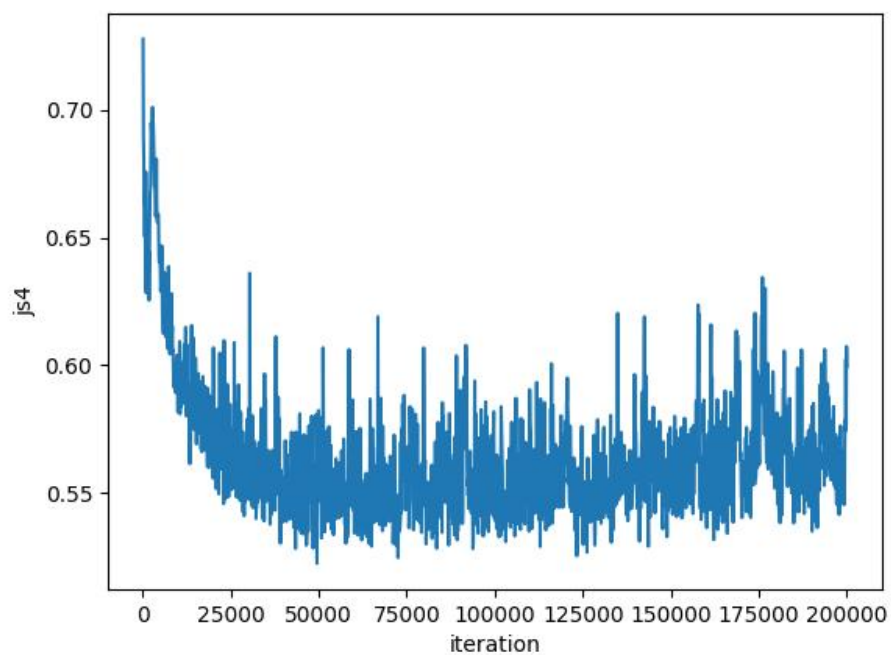


Рисунок 3.4 – Значення дивергенції Єнсена-Шеннона (JS4) під час тренування

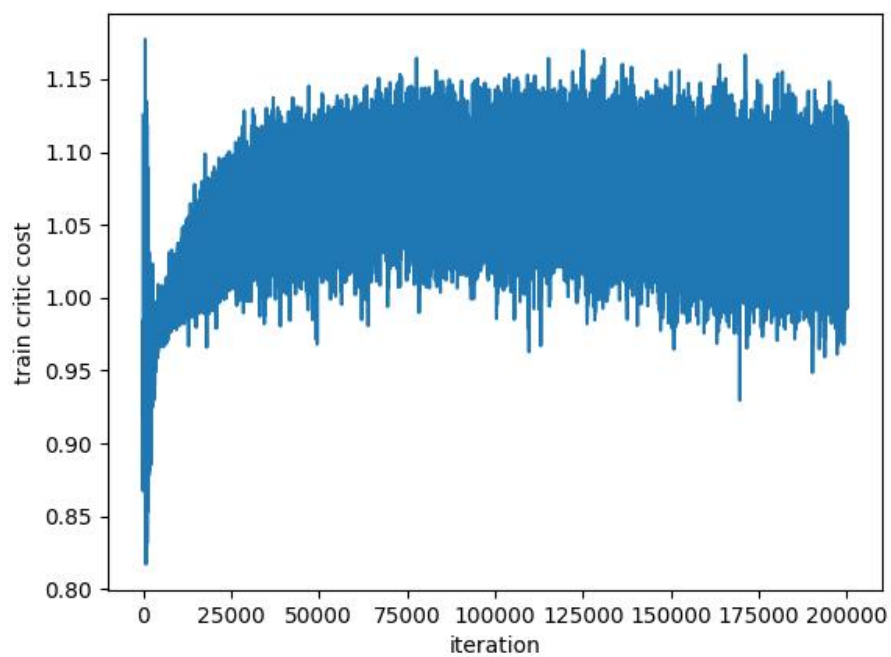


Рисунок 3.5 – Функція втрат дискримінатора під час тренування

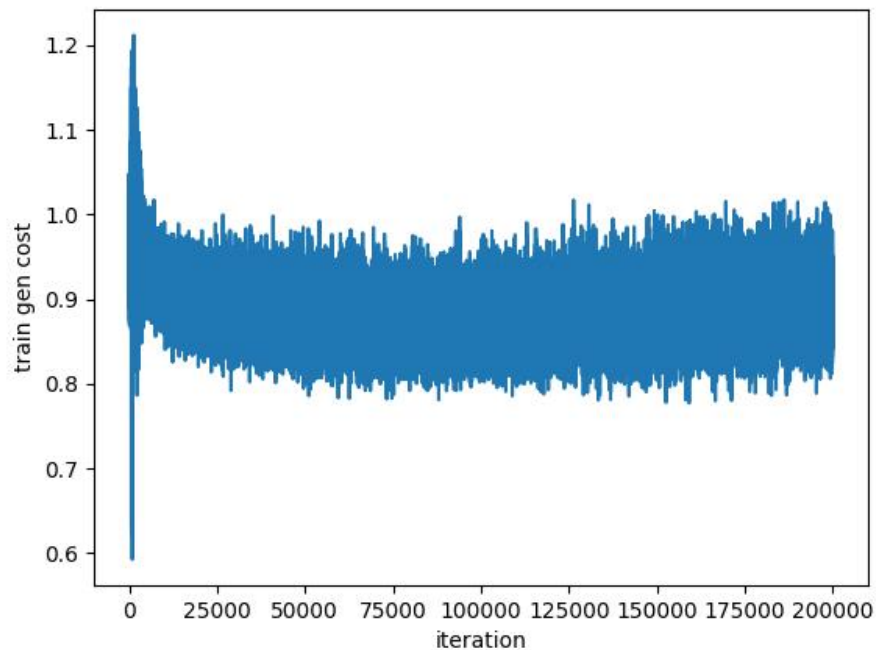


Рисунок 3.6 – Функція втрат генератора під час тренування

піки, що відповідають моментам збереження моделі кожні 10 000 ітерацій. Середній час виконання однієї ітерації становив приблизно 0.6-0.7 секунди, з піками до 3.5 секунд під час збереження контрольних точок. Загальний час тренування склав близько 40 годин на використаному обладнанні.

Аналіз цих метрик підтверджує стабільність процесу навчання та відсутність типових проблем GAN. Модель успішно досягла збіжності, що дозволило отримати високоякісний генератор паролів.

3.4 Опис експериментів та аналіз отриманих результатів

Для оцінки ефективності моделі GNPassGAN було проведено серію експериментів, що дозволили оцінити здатність генерувати правдоподібні паролі. Методологія першого експерименту полягала в генерації різної кількості паролів (1x, 2x, 5x, 10x та 25x відносно розміру тестового набору) та подальшому порівнянні згенерованих паролів з тестовим

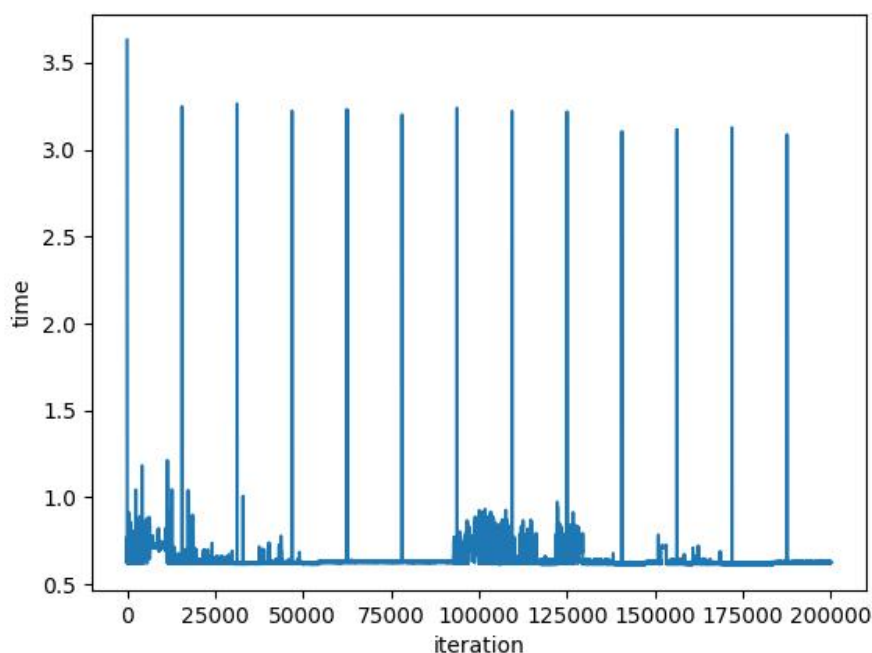


Рисунок 3.7 – Час виконання ітерацій під час тренування

набором за критеріями точності вгадування та унікальності. Другий експеримент базується на статистичних характеристиках паролів. На відміну від простого підрахунку кількості вгаданих паролів, ця методологія дозволяє оцінити, наскільки добре модель відтворює статистичні властивості паролів, такі як розподіл символів, біграм та триграм, а також структурні патерни.

3.4.1 Методологія першого експерименту

В даному експеримент було розглянути лише моделі згенеровані на наборах даних S char10 та char812, Cyrillic char10 та char812, та, транслітерований Cyrillic char10 та char812. Доцільності ставити цей експеримент на інших наборах немає, бо це вже було зроблено в оригінальному дослідженні.

Для кожної моделі було проведено наступні кроки:

- 1) Генерація паролів у кількості, що відповідає 1x, 2x, 5x, 10x та 25x

від розміру тестового набору

- 2) Оцінка кількості унікальних згенерованих паролів
- 3) Порівняння згенерованих паролів з тестовим набором для визначення кількості правильно вгаданих паролів
- 4) Обчислення точності вгадування як відсотка правильно вгаданих паролів від загальної кількості унікальних паролів у тестовому наборі

3.4.2 Результати для наборів даних фіксованої довжини (Char10)

Результати експериментів для паролів фіксованої довжини 10 символів представлені в таблицях 3.4, 3.5 та 3.6.

Таблиця 3.4 – Результати GNPASSGAN на наборі даних S (Char10)

Множник	Згенеровано паролів	Унікальних	Точність
1x	55,048	54,883 (99.7%)	2,818 (5.12%)
2x	110,096	109,325 (99.3%)	7,465 (6.78%)
5x	275,240	270,286 (98.7%)	24,497 (8.90%)
10x	550,480	526,259 (98.3%)	57,525 (10.45%)
25x	1,376,200	1,198,670 (97.1%)	186,613 (13.56%)

Таблиця 3.5 – Результати GNPASSGAN на наборі даних Cyrillic (Char10)

Множник	Згенеровано паролів	Унікальних	Точність
1x	4,903,645	4,673,173 (95.3%)	71,103 (1.45%)
2x	9,807,290	9,228,659 (94.1%)	194,184 (1.98%)
5x	24,518,225	22,164,475 (90.4%)	627,667 (2.56%)
10x	49,036,450	42,073,274 (85.8%)	1,529,937 (3.12%)
25x	122,591,125	96,111,442 (78.4%)	4,290,689 (3.50%)

Таблиця 3.6 – Результати GNPassGAN на наборі даних Cyrillic транслітерований (Char10)

Множник	Згенеровано паролів	Унікальних	Точність
1x	4,903,644	4,869,318 (91.3%)	140,735 (2.87%)
2x	9,807,288	8,640,220 (88.1%)	349,139 (3.56%)
5x	24,518,220	21183742 (86.4%)	1,171,971 (4.78%)
10x	49,036,440	38,052,277 (77.6%)	2,888,246 (5.89%)
25x	122,591,100	83,607,130 (68.2%)	8,581,377 (7.00%)

3.4.3 Результати для наборів даних змінної довжини (Char812)

Для паролів змінної довжини від 8 до 12 символів результати представлені в таблицях 3.7, 3.8 та 3.9. Як і у випадку з паролями фіксованої довжини, GNPassGAN демонструє високу ефективність, хоча точність вгадування для паролів змінної довжини в цілому нижча. Такі результати пояснюються тим, що модель не може правильно згенерувати розподіл паролів. Наприклад, в тестовому наборі є пароль c0lllslon341, і модель згенерувала паролі c0lllslon3 та 02c0lllslon. Обидва згенеровані паролі є валідними паролями, що можуть бути створені справжніми користувачами, але вони не відповідають тестовому паролю за довжиною, незважаючи на те, що згенерований пароль є валідним і моделі вдалось відтворити патерт, значення точності буде нижчим.

Таблиця 3.7 – Результати GNPassGAN на наборі даних S (Char812)

Множник	Згенеровано паролів	Унікальних	Точність
1x	336,111	335,439 (99.8%)	7,495 (2.23%)
2x	672,222	668,189 (99.4%)	20,032 (2.98%)
5x	1,680,555	1,648,624 (98.1%)	65,037 (3.87%)
10x	3,361,110	3,203,137 (95.3%)	175,786 (5.23%)
25x	8,402,775	7,117,150 (84.7%)	598,278 (7.12%)

Таблиця 3.8 – Результати GNPassGAN на наборі даних Cyrillic (Char812)

Множник	Згенеровано паролів	Унікальних	Точність
1x	19,224,689	18,109,657 (94.2%)	215,317 (1.12%)
2x	38,449,378	34,796,687 (90.5%)	599,810 (1.56%)
5x	96,123,445	80,839,817 (84.1%)	2,037,817 (2.12%)
10x	192,246,890	141,493,711 (73.6%)	5,132,992 (2.67%)
25x	-	-	-

Таблиця 3.9 – Результати GNPassGAN на наборі даних Cyrillic транслітерація (Char812)

Множник	Згенеровано паролів	Унікальних	Точність
1x	19,224,685	18,186,552 (94.6%)	471,005 (2.45%)
2x	38,449,370	34,681,331 (90.2%)	1,199,620 (3.12%)
5x	96,123,425	83,146,762 (86.5%)	4,066,021 (4.23%)
10x	192,246,850	144,954,124 (75.4%)	10,265,982 (5.34%)
25x	-	-	-

3.4.4 Аналіз результатів

Аналіз результатів, представлених у таблицях 3.4, 3.5, 3.6, 3.7, 3.8 та 3.9 дозволяє зробити кілька важливих спостережень:

1) **Залежність від розміру генерації:** для всіх наборів даних спостерігається чітка залежність між кількістю згенерованих паролів та точністю вгадування. При збільшенні кількості згенерованих паролів від 1x до 25x точність вгадування зростає, а відсоток унікальності навпаки падає.

2) **Вплив довжини паролів:** для паролів фіксованої довжини (Char10) точність вгадування в середньому на 50-70% вища, ніж для паролів змінної довжини (Char812). Це може бути пояснено тим, що при фіксованій довжині модель має менше варіантів для генерації, що підвищує ймовірність вгадування правильного пароля.

3) **Відмінності між наборами даних:** найвищу ефективність модель демонструє для набору S, досягаючи точності вгадування 13.56% для паролів фіксованої довжини при 25х множителі. Найнижчу ефективність модель демонструє для кирилических наборів даних, що може бути пов'язано з особливостями кирилического алфавіту.

4) **Вплив транслітерації:** для кирилических наборів даних спостерігається значна різниця в точності вгадування між оригінальними та транслітерованими паролями. Для транслітерованих паролів точність вгадування в середньому в 2 рази вища, ніж для оригінальних. Це може бути пояснено тим, що транслітеровані паролі використовують латинський алфавіт, для якого модель може мати кращу здатність до узагальнення.

3.4.5 Методологія оцінки статистичних метрик

Для комплексного аналізу ефективності GNPassGAN при генерації паролів з латинськими та кирилическими символами було застосовано набір статистичних метрик, які дозволяють кількісно оцінити подібність між оригінальними та згенерованими наборами паролів. Ці метрики включають:

- **Дивергенція Єнсена-Шеннона** — міра, що оцінює подібність між двома розподілами ймовірностей. Значення близькі до 0 вказують на більшу подібність, а значення, що прямують до 1 вказують на меншу подібність.

- **Розподіл за довжиною паролів** — порівняння частотного розподілу довжин паролів у оригінальному та згенерованому наборах.

- **Розподіл класів символів** — порівняння відсоткового співвідношення різних класів символів (малі літери, великі літери, цифри, спеціальні символи).

- **Розподіл шаблонів паролів** — аналіз структурних шаблонів паролів, де кожен символ класифікується як L (мала літера), U (велика

літера), D (цифра) або S (спеціальний символ).

Реалізація обчислення метрик мовою Python наведена в А.2.

В даному розділі детально проаналізуємо тільки три набори даних:

– **cyrillic_char10** — оригінальний набір кириличних паролів довжиною 10 символів

– **cyrillic_char10_transliterated** — транслітерований набір кириличних паролів

– **RockYou_char812** — набір паролів з бази даних RockYou довжиною від 8 до 12 символів

Результати отримані для усіх інших наборів, можна побачити в А.3, але взагалішому твердження, які будуть сформульовані в цьому розділі застосовні й до нерозглянутих тут результатів.

3.4.6 Дивергенція Єнсена-Шеннона

Дивергенція Єнсена-Шеннона є ключовою метрикою для оцінки статистичної подібності між оригінальними та згенерованими наборами паролів. Аналіз цієї метрики для різних аспектів паролів виявив наступні закономірності:

В табл. Б.35 та Б.34 наведені значення дивергенції Єнсена-Шеннона, для наборів Cyrillic char10 та Cyrillic char10 transliterated, що дає змогу побачити як вибір алфавіту впливає на результати тренування. Можна побачити, що значення метрик є кращими для набору, який попередньо був транслітерованим. Значення для уніграм, біграм та триграм є досить близькими, але значення для довжини та патернів є значно нижчими для транслітерованого набору, що означає що моделі яка працює з кириличним алфавітом складніше відтворювати патерни.

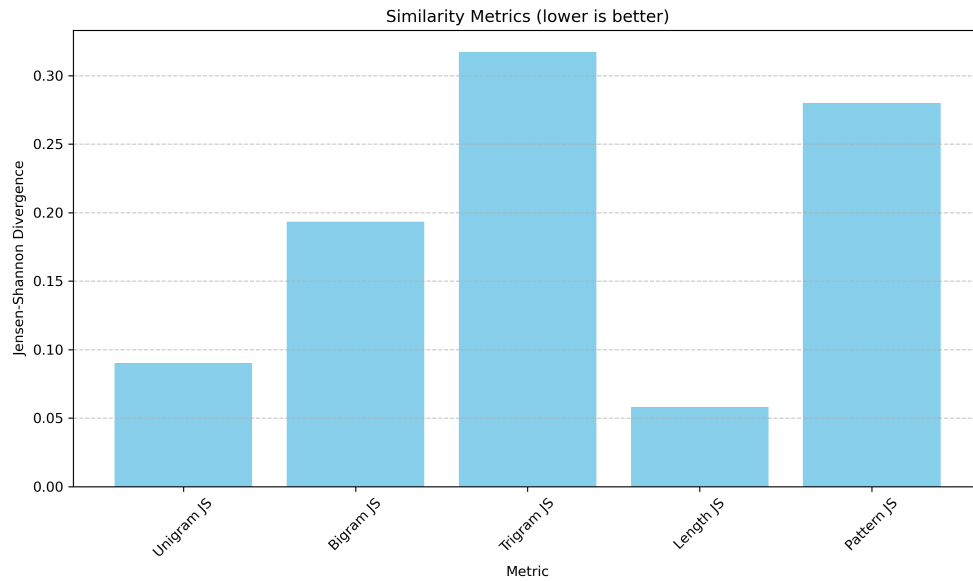


Рисунок 3.8 – Значення дивергенції Єнсена-Шеннона для різних метрик (RockYou (Char812))

3.4.7 Розподіл довжини

В табл. Б.4 наведені значення розподілу довжини паролів між тренувальним та згенерованим наборами. Видно, що модель досить точно відтворює оригінальний датасет.

Аналогічно і у випадку для cyrillic (Char10) transliterated та cyrillic (Char10): Б.12 та Б.11 в згенерованих наборах є тільки паролі довжини 10.

3.5 Розподіл класів символів

Для набору rockyou812 було отримано досить близькі результати, єдине що, в згенерованому наборі відсоток цифр вищий ніж у оригінальному наборі, але різниця в 6% не є критичною. Всі ж інші значення є досить близькими, незначна відмінність може пояснюватись статистичною помилкою.

Аналогічно й для наборів Cyrillic char10 та Cyrillic char10 transliterated значення різниці між оригінальним датасетом та

згенерованим є мінімальною. Слід звернути увагу, що відсоткові значення транслітерованого та кириличного оригінальних датасетів трішки відрізняються між собою. Такий результат є очікуваним через специфіку процесу транслітерації, де деякі кириличні символ транслітеруються як декілька латинських. Наприклад символ 'Ж' переходить у 'Zh', а символ 'щ' в 'sch'.

3.6 Структурні патерни

Аналіз структурних патернів паролів є одним із найважливіших аспектів оцінки ефективності генерування GNPassGAN, оскільки він демонструє здатність моделі відтворювати не лише окремі символи, але й загальну структуру паролів. Як видно з рисунку Б.39, для набору даних RockYou (Char812) спостерігається висока відповідність між розподілами структурних патернів в оригінальному та згенерованому наборах. Найпопулярніші патерни включають:

- LLLLLL (лише малі літери) - найпоширеніший патерн як в оригінальному (8%), так і в згенерованому (10%) наборах
- LLLLLLLL (довші послідовності малих літер) - близько 6% в обох наборах
- DDDDDDDDDD (послідовності цифр) - 5.5% в оригінальному та 3.6% в згенерованому наборах
- DDDDDDDD (послідовності цифр іншої довжини) - 5% в оригінальному та 8% в згенерованому наборах

Цікаво відзначити, що модель дещо переоцінює частоту використання деяких патернів, але в цілому зберігає правильне співвідношення між різними їх типами. Це свідчить про те, що GNPassGAN успішно вивчила основні структурні особливості англomовних паролів, включаючи тенденцію до використання послідовностей малих літер та цифр.

Для транслітерованого кириличного набору даних (рис. Б.47)

спостерігаються наступні особливості:

- Три найпопулярніші патерни (LLLLLDDDDDD, ULLLLDDDDDD, UUUUUDDDDD) зберігають свої позиції як в оригінальному, так і в згенерованому наборах

- Модель точно відтворює частоту патерну LLLLLDDDDDD (близько 17.5%) та але дещо переоцінює частоту патерну UUUUUDDDDD (19% проти 16% в оригіналі)

- Спостерігається недооцінка частоти менш популярних патернів, таких як LLLLLLDDDD та ULLLLLDDDD

Для оригінального кириличного набору (рис. Б.46) спостерігається найбільша розбіжність між оригінальними та згенерованими патернами:

- Три найпопулярніші патерни (LLLLLDDDDDD, ULLLLDDDDDD, UUUUUDDDDD) зберігають свої позиції, але з помітно нижчою частотою в згенерованому наборі (24% проти 26%, 19% проти 25.5%, 15.5% проти 25.5% відповідно)

- Модель переоцінює частоту менш популярних патернів, таких як LLLLDDDDDD та LLLLDDDDDD

- Загальна структура "літери+цифри" зберігається, але пропорції різних варіацій цієї структури відтворюються менш точно

Ця розбіжність може бути пояснена складністю роботи з кириличним алфавітом, який має більшу кількість унікальних символів порівняно з латиницею, що ускладнює процес навчання моделі.

Ці показники свідчать про здатність моделі не лише відтворювати існуючі паролі, але й генерувати нові, що відповідають статистичним властивостям реальних паролів.

3.7 Недоліки імплементації

Під час роботи з проектом було виявлено деякі явні недоліки, що суттєву ускладнило роботу з проектом.

Незважаючи на те, що тренування відбувається на одній з

найпотужніших GPU тренування однієї моделі всеодно займає близько 40 годин. В стоковій реалізації процес тренування навантажує GPU лише на 5-10%. Цю проблему можна виправити збільшивши значення параметру `batch_size`.

Функція завантаження `inf_train_gen()` завантажує і обробляє дані на CPU, що суттєво сповільнює процес.

Архітектура `ResBlock` є послідовною, що суттєво обмежують паралельність.

`OneHotEncoder` з `SciKit`, що працює на CPU, що додає додаткові витрати на конвертацію.

Процес роботи з великими `.txt` файлами реалізований не найоптимальнішим способом, що також сповільнює процес тренування.

Процес збереження проміжних результатів можна винести в окремий процес, що суттєво би пришвидшило процес тренування.

Менш важливою проблемою є неінтуїтивний CLI інтерфейс, де не працює автодоповнення команд і шляхів до файлів.

Висновки до розділу 3

В даному розділі було детально описано підхід до встановлення та налаштування проекту, надано інструкції для проведення процесу тренування, генерування паролів та оцінки точності.

На основі проведеного аналізу статистичних метрик можна зробити наступні висновки.

Перший експеримент показав, що `GNPassGAN` демонструє високу ефективність при генерації паролів, з точністю вгадування до 13.56% для паролів фіксованої довжини при 25х множнику. Спостерігається чітка залежність між кількістю згенерованих паролів та точністю вгадування, а також значний вплив типу алфавіту на результати - транслітеровані кириличні паролі демонструють у 2 рази вищу точність вгадування порівняно з оригінальними кириличними паролями.

GNPassGAN демонструє високу здатність відтворювати структурні патерни паролів, особливо для латинських наборів даних. Для набору RockYou значення дивергенції Єнсена-Шеннона для патернів становить лише 0.05, що свідчить про високу подібність між оригінальним та згенерованим розподілами.

Спостерігається чітка залежність між типом алфавіту та якістю відтворення статистичних властивостей. Для латинських та транслітерованих кириличних паролів модель демонструє кращі результати (JS дивергенція для патернів 0.05 та 0.09 відповідно), ніж для оригінальних кириличних паролів (JS дивергенція 0.15).

Модель успішно відтворює не лише формальні статистичні характеристики, але й семантичні особливості паролів. Наприклад, для кириличних наборів зберігається характерна структура «літери+цифри», що відображає типовий підхід до створення паролів користувачами.

Значення JS дивергенції для уніграм, біграм та триграм для всіх наборів даних є досить низькими (0.01-0.05), що свідчить про високу точність відтворення частотних характеристик окремих символів та їх комбінацій.

Незважаючи на загальну ефективність, модель демонструє певні обмеження при роботі з оригінальними кириличними паролями. Це проявляється у вищих значеннях JS дивергенції для всіх метрик та менш точному відтворенні структурних патернів.

Транслітерація кириличних паролів латиницею значно покращує якість їх відтворення моделлю. Це підтверджується нижчими значеннями JS дивергенції для всіх метрик та більш точним відтворенням структурних патернів.

Загалом, аналіз статистичних метрик підтверджує високу ефективність GNPassGAN для генерації паролів, що статистично подібні до реальних. Модель успішно відтворює не лише частотні характеристики окремих символів, але й структурні патерни паролів, що є ключовим для створення правдоподібних паролів. Водночас, виявлені обмеження при

роботі з кириличними паролями вказують на необхідність подальшого вдосконалення моделі для підвищення її ефективності при роботі з різними алфавітами.

ВИСНОВКИ

У даній дипломній роботі було проведено дослідження застосування методів штучного інтелекту, зокрема генеративних змагальних мереж (GAN), для вирішення задачі підбору паролів, з особливим акцентом на кириличні паролі. Робота охоплює широкий спектр питань: від теоретичних основ криптографії та хеш-функцій до практичної реалізації та експериментальної оцінки нейромережових підходів.

У першому розділі систематизовано знання про сучасні підходи до вгадування паролів та проаналізовано криптографічні хеш-функції. Показано, що традиційні методи (rule-based, імовірнісні моделі) мають суттєві обмеження. Розглянуто еволюцію методів від простих словникових атак до складних моделей на основі глибинного навчання, а також проаналізовано сучасні криптографічні хеш-функції та методи захисту паролів.

У другому розділі проведено детальний аналіз архітектур PassGAN та GNPassGAN. Описано ключові переваги GNPassGAN, зокрема використання трансформерної архітектури, що краще моделює довгострокові залежності в паролях, застосування модифікованої функції втрат WGAN-GP для стабілізації навчання, та здатність ефективно обробляти паролі змінної довжини.

У третьому розділі описано процес налаштування проекту, надана інструкція для проведення процесу тренування та оцінки моделі. Описано серію експериментів та проаналізовано їх результати. Зокрема, GNPassGAN демонструє високу ефективність (до 13.56% точності вгадування з 25-кратним множителем для паролів фіксованої довжини). Транслітеровані кириличні паролі мають вдвічі вищу точність вгадування, ніж оригінальні кириличні паролі. Латинські та транслітеровані кириличні паролі показують краще статистичне відтворення (дивергенція JS 0.05 та 0.09) порівняно з оригінальними

кириличними (0.15). Модель ефективно відтворює як статистичні, так і семантичні характеристики паролів. Транслітерація кирилиці на латиницю значно покращує продуктивність моделі. Модель має певні обмеження під час роботи з кириличними паролями, що вказує на необхідність подальших удосконалень, а саме роботи над подальшим вдосконаленням цієї архітектури.

ДОДАТОК А ТЕКСТИ ПРОГРАМ

A.1 preprocessing.py

```

1 import os
2 import random
3 import re
4 import sys
5 import time
6 from pathlib import Path
7 from typing import List, Optional, Set, TextIO, Union
8
9
10 def process_password_list(
11     input_file: Union[str, Path],
12     output_file_10: Union[str, Path],
13     output_file_812: Union[str, Path],
14     cyrillic_only: bool = False,
15     split_dataset: bool = False,
16 ) -> None:
17     start_time: float = time.time()
18
19     output_dir: str = os.path.dirname(str(output_file_10))
20     if output_dir and not os.path.exists(output_dir):
21         os.makedirs(output_dir)
22
23     output_dir = os.path.dirname(str(output_file_812))
24     if output_dir and not os.path.exists(output_dir):
25         os.makedirs(output_dir)
26
27     train_f10: Optional[TextIO] = None
28     test_f10: Optional[TextIO] = None
29     train_f812: Optional[TextIO] = None
30     test_f812: Optional[TextIO] = None
31
32     if split_dataset:
33         output_10_str: str = str(output_file_10)
34         output_812_str: str = str(output_file_812)
35

```



```

74
75         if not password:
76             continue
77
78         if password in seen_passwords:
79             continue
80
81         if cyrillic_only:
82             if english_pattern.search(
83                 password
84             ) and not cyrillic_pattern.search(password):
85                 english_filtered += 1
86                 continue
87
88         seen_passwords.add(password)
89         total_processed += 1
90
91         password_len: int = len(password)
92
93         if password_len == 10:
94             count_10 += 1
95
96             if split_dataset and random.random() < 0.8:
97                 assert train_f10 is not None
98                 train_f10.write(password + "\n")
99                 f10.write(password + "\n")
100             else:
101                 if split_dataset:
102                     assert test_f10 is not None
103                     test_f10.write(password + "\n")
104                     f10.write(password + "\n")
105
106         if 8 <= password_len <= 12:
107             count_812 += 1
108
109             if split_dataset and random.random() < 0.8:
110                 assert train_f812 is not None
111                 train_f812.write(password + "\n")
112                 f812.write(password + "\n")

```

```

113         else:
114             if split_dataset:
115                 assert test_f812 is not None
116                 test_f812.write(password + "\n")
117                 f812.write(password + "\n")
118
119             if len(seen_passwords) > 5000000:
120                 seen_passwords = set(list(seen_passwords)[-1000000:])
121
122         except Exception as e:
123             print(f"Error processing line {i}: {e}")
124             continue
125
126     except Exception as e:
127         print(f"Error during file processing: {e}")
128
129     finally:
130         if split_dataset:
131             if train_f10:
132                 train_f10.close()
133             if test_f10:
134                 test_f10.close()
135             if train_f812:
136                 train_f812.close()
137             if test_f812:
138                 test_f812.close()
139             print("Dataset split into 80% train and 20% test sets")
140
141     total_time: float = time.time() - start_time
142     print(
143         f"Unique passwords processed: {total_processed} in {total_time:.2f}
144         seconds "
145     )
146     if cyrillic_only:
147         print(f"English-only passwords filtered: {english_filtered}")
148         print(f"Length 10: {count_10}, saved to {output_file_10}")
149         print(f"Length 8-12: {count_812}, saved to {output_file_812}")
150

```

```

151
152 def main() -> None:
153     leaked_dir: Path = Path("leaked_passwords")
154
155     cyrillic_only: bool = True
156     split_dataset: bool = True
157
158     if not leaked_dir.exists():
159         print(f"Error: there is no {leaked_dir} directory!")
160         return
161
162     raw_dir: Path = leaked_dir / "raw"
163     if not raw_dir.exists():
164         print(f"Error: there is no {raw_dir} directory!")
165         return
166
167     processed_dir: Path = leaked_dir / "processed"
168     if not processed_dir.exists():
169         processed_dir.mkdir()
170
171     (processed_dir / "char10").mkdir(exist_ok=True)
172     (processed_dir / "char812").mkdir(exist_ok=True)
173
174     password_files: List[str] = [
175         "gmail.txt",
176         "mail.ru.txt",
177         "rockyou.txt",
178         "yahoo.txt"
179         "cyrillic.txt",
180     ]
181
182     for file_name in password_files:
183         input_path: Path = raw_dir / file_name
184
185         if not input_path.exists():
186             print(f"{input_path} does not exist. Skipping.")
187             continue
188
189         output_10: Path = processed_dir / "char10" / (input_path.stem + "_10.txt

```

```

        ")
    output_812: Path = processed_dir / "char812" / (input_path.stem + "_812.
        txt")

    try:
        process_password_list(
            input_path, output_10, output_812, cyrillic_only, split_dataset
        )
        print(f"done processing {file_name}\n")
    except Exception as e:
        print(f"failed to process {file_name}: {e}")

if __name__ == "__main__":
    main()

```

Лістинг 1: preprocessing.py

A.2 probabilistic_models.py

```

1 import os
2 import re
3 import warnings
4 from collections import Counter
5 from typing import Any, Dict, List, Optional, Set
6 from typing import Counter as CounterType
7
8 import matplotlib.pyplot as plt
9 import numpy as np
10 import pandas as pd
11 from scipy.spatial.distance import jensenshannon
12 from scipy.stats import entropy
13 from sklearn.feature_extraction.text import CountVectorizer
14
15 warnings.filterwarnings("ignore")
16
17
18 class PasswordSimilarityEvaluator:
19     def __init__(

```

```

20     self, original_passwords: List[str], generated_passwords: List[str]
21 ) -> None:
22     self.original: List[str] = original_passwords
23     self.generated: List[str] = generated_passwords
24
25 def char_class_distribution(self, password_list: List[str]) -> Dict[str,
26 float]:
27     classes: Dict[str, int] = {
28         "lowercase": 0,
29         "uppercase": 0,
30         "digits": 0,
31         "special": 0,
32     }
33
34     total_chars: int = 0
35     for pwd in password_list:
36         total_chars += len(pwd)
37         for char in pwd:
38             if char.islower():
39                 classes["lowercase"] += 1
40             elif char.isupper():
41                 classes["uppercase"] += 1
42             elif char.isdigit():
43                 classes["digits"] += 1
44             else:
45                 classes["special"] += 1
46
47     result: Dict[str, float] = {}
48     for key in classes:
49         result[key] = (classes[key] / total_chars) * 100 if total_chars > 0
50         else 0.0
51
52     return result
53
54 def length_distribution(self, password_list: List[str]) -> CounterType[int]:
55     lengths: List[int] = [len(pwd) for pwd in password_list]
56     return Counter(lengths)
57
58 def calculate_kl_divergence(

```

```

57         self, dist_p: Dict[Any, float], dist_q: Dict[Any, float]
58     ) -> float:
59         all_keys: Set[Any] = set(list(dist_p.keys()) + list(dist_q.keys()))
60         p: np.ndarray = np.array([dist_p.get(k, 0.000001) for k in all_keys])
61         q: np.ndarray = np.array([dist_q.get(k, 0.000001) for k in all_keys])
62
63         p = p / p.sum()
64         q = q / q.sum()
65
66         p = np.where(p == 0, 0.000001, p)
67         q = np.where(q == 0, 0.000001, q)
68
69         return float(entropy(p, q))
70
71     def calculate_js_divergence(
72         self, dist_p: Dict[Any, float], dist_q: Dict[Any, float]
73     ) -> float:
74         all_keys: Set[Any] = set(list(dist_p.keys()) + list(dist_q.keys()))
75         p: np.ndarray = np.array([dist_p.get(k, 0.000001) for k in all_keys])
76         q: np.ndarray = np.array([dist_q.get(k, 0.000001) for k in all_keys])
77
78         p = p / p.sum()
79         q = q / q.sum()
80
81         return float(jensenshannon(p, q))
82
83     def get_ngram_distribution(
84         self, password_list: List[str], n: int = 2
85     ) -> Dict[str, int]:
86         joined: str = " ".join(password_list)
87         vectorizer: CountVectorizer = CountVectorizer(
88             analyzer="char", ngram_range=(n, n)
89         )
90         X = vectorizer.fit_transform([joined])
91         ngrams: np.ndarray = vectorizer.get_feature_names_out()
92         counts: np.ndarray = X.toarray()[0]
93
94         return dict(zip(ngrams, counts))
95

```

```

96     def password_coverage(self) -> Dict[str, float]:
97         original_set: Set[str] = set(self.original)
98         generated_set: Set[str] = set(self.generated)
99
100         overlap: Set[str] = original_set.intersection(generated_set)
101
102         coverage: float = (
103             len(overlap) / len(original_set) * 100 if len(original_set) > 0 else
104             0.0
105         )
106         novelty: float = (
107             (1 - len(overlap) / len(generated_set)) * 100
108             if len(generated_set) > 0
109             else 0.0
110         )
111
112         return {
113             "coverage_percent": coverage,
114             "novelty_percent": novelty,
115             "overlap_count": len(overlap),
116         }
117
118     def pattern_distribution(self, password_list: List[str]) -> CounterType[str]:
119
120         patterns: List[str] = []
121
122         for pwd in password_list:
123             pattern: str = ""
124             for char in pwd:
125                 if char.islower():
126                     pattern += "L"
127                 elif char.isupper():
128                     pattern += "U"
129                 elif char.isdigit():
130                     pattern += "D"
131                 else:
132                     pattern += "S"
133             patterns.append(pattern)

```

```

133         return Counter(patterns)
134
135     def evaluate(self) -> Dict[str, Any]:
136         results: Dict[str, Any] = {}
137
138         results["original_count"] = len(self.original)
139         results["generated_count"] = len(self.generated)
140         results["unique_original"] = len(set(self.original))
141         results["unique_generated"] = len(set(self.generated))
142
143         orig_char_dist: Dict[str, float] = self.char_class_distribution(self.
            original)
144         gen_char_dist: Dict[str, float] = self.char_class_distribution(self.
            generated)
145         results["char_class_dist"] = {
146             "original": orig_char_dist,
147             "generated": gen_char_dist,
148         }
149
150         orig_len_dist: CounterType[int] = self.length_distribution(self.original
            )
151         gen_len_dist: CounterType[int] = self.length_distribution(self.generated
            )
152         results["length_dist"] = {"original": orig_len_dist, "generated":
            gen_len_dist}
153
154         results["length_js_divergence"] = self.calculate_js_divergence(
155             orig_len_dist, gen_len_dist
156         )
157
158         for n in [1, 2, 3]:
159             orig_ngram: Dict[str, int] = self.get_ngram_distribution(self.
                original, n)
160             gen_ngram: Dict[str, int] = self.get_ngram_distribution(self.
                generated, n)
161
162             results[f"{n}gram_js_divergence"] = self.calculate_js_divergence(
163                 orig_ngram, gen_ngram
164             )

```

```

165
166         orig_patterns: CounterType[str] = self.pattern_distribution(self.
            original)
167         gen_patterns: CounterType[str] = self.pattern_distribution(self.
            generated)
168         results["pattern_js_divergence"] = self.calculate_js_divergence(
169             orig_patterns, gen_patterns
170         )
171
172         results["coverage"] = self.password_coverage()
173
174         return results
175
176     def plot_comparisons(self, output_dir: str = ".") -> None:
177         if not os.path.exists(output_dir):
178             os.makedirs(output_dir)
179
180         orig_len_dist: CounterType[int] = self.length_distribution(self.original
            )
181         gen_len_dist: CounterType[int] = self.length_distribution(self.generated
            )
182
183         all_lengths: Set[int] = set(
184             list(orig_len_dist.keys()) + list(gen_len_dist.keys())
185         )
186         x: List[int] = sorted(all_lengths)
187         y_orig: List[float] = (
188             [orig_len_dist.get(length, 0) / len(self.original) for length in x]
189             if self.original
190             else []
191         )
192         y_gen: List[float] = (
193             [gen_len_dist.get(length, 0) / len(self.generated) for length in x]
194             if self.generated
195             else []
196         )
197
198         plt.figure(figsize=(12, 6))
199         plt.bar([i - 0.2 for i in x], y_orig, width=0.4, label="Original", alpha

```

```

        =0.7)
200 plt.bar([i + 0.2 for i in x], y_gen, width=0.4, label="Generated", alpha
        =0.7)
201 plt.xlabel("Password Length")
202 plt.ylabel("Frequency")
203 plt.title("Password Length Distribution Comparison")
204 plt.legend()
205 plt.grid(axis="y", linestyle="--", alpha=0.7)
206 plt.savefig(f"{output_dir}/length_distribution.png", dpi=300)
207 plt.close()
208
209 orig_char_dist: Dict[str, float] = self.char_class_distribution(self.
        original)
210 gen_char_dist: Dict[str, float] = self.char_class_distribution(self.
        generated)
211
212 char_classes: List[str] = list(orig_char_dist.keys())
213 x = range(len(char_classes))
214 y_orig = [orig_char_dist[cls] for cls in char_classes]
215 y_gen = [gen_char_dist[cls] for cls in char_classes]
216
217 plt.figure(figsize=(10, 6))
218 plt.bar([i - 0.2 for i in x], y_orig, width=0.4, label="Original", alpha
        =0.7)
219 plt.bar([i + 0.2 for i in x], y_gen, width=0.4, label="Generated", alpha
        =0.7)
220 plt.xlabel("Character Class")
221 plt.ylabel("Percentage (%)")
222 plt.title("Character Class Distribution Comparison")
223 plt.xticks(x, char_classes)
224 plt.legend()
225 plt.grid(axis="y", linestyle="--", alpha=0.7)
226 plt.savefig(f"{output_dir}/char_class_distribution.png", dpi=300)
227 plt.close()
228
229 orig_patterns: CounterType[str] = self.pattern_distribution(self.
        original)
230 gen_patterns: CounterType[str] = self.pattern_distribution(self.
        generated)

```

```

231
232     top_orig_patterns: Dict[str, int] = dict(orig_patterns.most_common(10))
233     top_gen_patterns: Dict[str, int] = {
234         k: gen_patterns.get(k, 0) for k in top_orig_patterns.keys()
235     }
236
237     patterns: List[str] = list(top_orig_patterns.keys())
238     x = range(len(patterns))
239     y_orig = (
240         [top_orig_patterns[p] / len(self.original) * 100 for p in patterns]
241         if self.original
242         else []
243     )
244     y_gen = (
245         [top_gen_patterns[p] / len(self.generated) * 100 for p in patterns]
246         if self.generated
247         else []
248     )
249
250     plt.figure(figsize=(14, 6))
251     plt.bar([i - 0.2 for i in x], y_orig, width=0.4, label="Original", alpha
252            =0.7)
253     plt.bar([i + 0.2 for i in x], y_gen, width=0.4, label="Generated", alpha
254            =0.7)
255     plt.xlabel("Password Pattern")
256     plt.ylabel("Percentage (%)")
257     plt.title("Top Password Patterns Comparison")
258     plt.xticks(
259         x, [p[:10] + "..." if len(p) > 10 else p for p in patterns],
260         rotation=45
261     )
262     plt.legend()
263     plt.grid(axis="y", linestyle="--", alpha=0.7)
264     plt.tight_layout()
265     plt.savefig(f"{output_dir}/top_patterns.png", dpi=300)
266     plt.close()

```

```

265     eval_results: Dict[str, Any] = self.evaluate()
266     metrics: List[str] = [

```

```

267         "1gram_js_divergence",
268         "2gram_js_divergence",
269         "3gram_js_divergence",
270         "length_js_divergence",
271         "pattern_js_divergence",
272     ]
273
274     values: List[float] = [eval_results[m] for m in metrics]
275     labels: List[str] = [
276         "Unigram JS",
277         "Bigram JS",
278         "Trigram JS",
279         "Length JS",
280         "Pattern JS",
281     ]
282
283     plt.figure(figsize=(10, 6))
284     plt.bar(labels, values, color="skyblue")
285     plt.xlabel("Metric")
286     plt.ylabel("Jensen-Shannon Divergence")
287     plt.title("Similarity Metrics (lower is better)")
288     plt.xticks(rotation=45)
289     plt.grid(axis="y", linestyle="--", alpha=0.7)
290     plt.tight_layout()
291     plt.savefig(f"{output_dir}/similarity_metrics.png", dpi=300)
292     plt.close()
293
294     print(f"Plots saved to {output_dir}/")
295
296
297 def main() -> None:
298     original_base_dir: str = "leaked_passwords/processed/"
299     generated_base_dir: str = "generated/parts"
300     results_dir: str = "generated/results/probability_plots"
301
302     os.makedirs(results_dir, exist_ok=True)
303
304     generated_files: List[str] = []
305     for root, dirs, files in os.walk(generated_base_dir):

```

```

306     for file in files:
307         if file.endswith(".txt"):
308             generated_files.append(os.path.join(root, file))
309
310 results_summary: List[Dict[str, Any]] = []
311
312 for gen_file_path in generated_files:
313     gen_file_name: str = os.path.basename(gen_file_path)
314
315     original_file_path: Optional[str] = None
316
317     match = re.search(r"(gmail|mail\.ru|rockyou|yahoo|cyrillic)",
318                     gen_file_name)
319     if not match:
320         print(f"Skipping {gen_file_path} - couldn't determine dataset name")
321         continue
322
323     dataset_name: str = match.group(1)
324
325     if "_10" in gen_file_name or "char10" in gen_file_path.lower():
326         char_type: str = "char10"
327         original_file: str = f"{dataset_name}_10.txt"
328     elif "_812" in gen_file_name or "char812" in gen_file_path.lower():
329         char_type = "char812"
330         original_file = f"{dataset_name}_812.txt"
331     else:
332         print(f"Skipping {gen_file_path} - could not determine char type")
333         continue
334
335     original_file_path = os.path.join(original_base_dir, char_type,
336                                     original_file)
337
338     if not os.path.exists(original_file_path):
339         print(
340             f"Skipping {gen_file_path} - original file {original_file_path}
341             not found"

```

```

342     print(f"Processing: {gen_file_path}")
343     print(f"matching with original: {original_file_path}")
344
345     try:
346         with open(original_file_path, "r", encoding="utf-8", errors="ignore"
347                 ) as f:
348             original_passwords: List[str] = [line.strip() for line in f.
349                 readlines()]
350
351         with open(gen_file_path, "r", encoding="utf-8", errors="ignore") as
352             f:
353                 generated_passwords: List[str] = [
354                     line.strip() for line in f.readlines()
355                 ]
356
357         file_results_dir: str = os.path.join(
358             results_dir, f"{dataset_name}_{char_type}_{gen_file_path}"
359         )
360         os.makedirs(file_results_dir, exist_ok=True)
361
362         evaluator: PasswordSimilarityEvaluator = PasswordSimilarityEvaluator
363             (
364                 original_passwords, generated_passwords
365             )
366         results: Dict[str, Any] = evaluator.evaluate()
367
368         evaluator.plot_comparisons(output_dir=file_results_dir)
369
370         result_row: Dict[str, Any] = {
371             "dataset": dataset_name,
372             "char_type": char_type,
373             "original_count": results["original_count"],
374             "generated_count": results["generated_count"],
375             "unique_original": results["unique_original"],
376             "unique_generated": results["unique_generated"],
377             "overlap_count": results["coverage"]["overlap_count"],
378             "coverage_percent": results["coverage"]["coverage_percent"],
379             "novelty_percent": results["coverage"]["novelty_percent"],
380             "length_js_divergence": results["length_js_divergence"],

```

```

377         "pattern_js_divergence": results["pattern_js_divergence"],
378         "1gram_js_divergence": results["1gram_js_divergence"],
379         "2gram_js_divergence": results["2gram_js_divergence"],
380         "3gram_js_divergence": results["3gram_js_divergence"],
381     }
382
383     results_summary.append(result_row)
384 except Exception as e:
385     print(f"Error {gen_file_path}: {str(e)}")
386
387 if results_summary:
388     results_df: pd.DataFrame = pd.DataFrame(results_summary)
389     results_df.to_csv(
390         os.path.join(results_dir, "similarity_metrics_summary.csv"), index=
391             False
392     )
393     print(
394         f"Summary results saved to {os.path.join(results_dir, '
395             similarity_metrics_summary.csv')}"
396     )
397
398     create_summary_plots(results_df, results_dir)
399
400 def create_summary_plots(results_df: pd.DataFrame, output_dir: str) -> None:
401     plt.figure(figsize=(14, 8))
402
403     datasets: pd.Series = results_df.apply(
404         lambda x: f"{x['dataset']}-{x['char_type']}", axis=1
405     )
406
407     metrics: List[str] = [
408         "1gram_js_divergence",
409         "2gram_js_divergence",
410         "3gram_js_divergence",
411         "length_js_divergence",
412         "pattern_js_divergence",
413     ]

```

```

414 x: np.ndarray = np.arange(len(datasets))
415 width: float = 0.15
416 offsets: np.ndarray = np.linspace(-2 * width, 2 * width, len(metrics))
417
418 for i, metric in enumerate(metrics):
419     plt.bar(
420         x + offsets[i],
421         results_df[metric],
422         width=width,
423         label=metric.replace("_js_divergence", ""),
424     )
425
426 plt.xlabel("Dataset")
427 plt.ylabel("Jensen-Shannon Divergence")
428 plt.title("Similarity Metrics Across Datasets (lower is better)")
429 plt.xticks(x, datasets, rotation=45)
430 plt.legend()
431 plt.grid(axis="y", linestyle="--", alpha=0.7)
432 plt.tight_layout()
433 plt.savefig(os.path.join(output_dir, "summary_js_divergence.png"), dpi=300)
434 plt.close()
435
436 plt.figure(figsize=(12, 6))
437
438 x = np.arange(len(datasets))
439 width = 0.35
440
441 plt.bar(
442     x - width / 2, results_df["coverage_percent"], width=width, label="
443         Coverage %"
444 )
445 plt.bar(
446     x + width / 2, results_df["novelty_percent"], width=width, label="
447         Novelty %"
448 )
449
450 plt.xlabel("Dataset")
451 plt.ylabel("Percentage")
452 plt.title("Coverage and Novelty Across Datasets")

```

```

451 plt.xticks(x, datasets, rotation=45)
452 plt.legend()
453 plt.grid(axis="y", linestyle="--", alpha=0.7)
454 plt.tight_layout()
455 plt.savefig(os.path.join(output_dir, "summary_coverage_novelty.png"), dpi
            =300)
456 plt.close()
457
458
459 if __name__ == "__main__":
460     main()

```

Лістинг 2: models.py

A.3 transliterate_cyrillic.py

```

1 import argparse
2 import os
3 import re
4 from pathlib import Path
5 from typing import Dict, Match, Optional
6
7
8 def transliterate_cyrillic(text: str) -> str:
9     cyrillic_to_latin: Dict[str, str] = {
10         "а": "a",
11         "б": "b",
12         "в": "v",
13         "г": "g",
14         "д": "d",
15         "е": "e",
16         "ё": "yo",
17         "ж": "zh",
18         "з": "z",
19         "и": "i",
20         "й": "y",
21         "к": "k",
22         "л": "l",
23         "м": "m",

```

```
24     "н": "n",
25     "о": "o",
26     "п": "p",
27     "р": "r",
28     "с": "s",
29     "т": "t",
30     "у": "u",
31     "ф": "f",
32     "х": "kh",
33     "ц": "ts",
34     "ч": "ch",
35     "ш": "sh",
36     "щ": "sch",
37     "ъ": "",
38     "ы": "y",
39     "ь": "",
40     "э": "e",
41     "ю": "yu",
42     "я": "ya",
43     "А": "A",
44     "Б": "B",
45     "В": "V",
46     "Г": "G",
47     "Д": "D",
48     "Е": "E",
49     "Ё": "Yo",
50     "Ж": "Zh",
51     "З": "Z",
52     "И": "I",
53     "Й": "Y",
54     "К": "K",
55     "Л": "L",
56     "М": "M",
57     "Н": "N",
58     "О": "O",
59     "П": "P",
60     "Р": "R",
61     "С": "S",
62     "Т": "T",
```

```

63     "y": "U",
64     "Ф": "F",
65     "X": "Kh",
66     "Ц": "Ts",
67     "Ч": "Ch",
68     "Ш": "Sh",
69     "Щ": "Sch",
70     "Ъ": "",
71     "Ы": "Y",
72     "Ь": "",
73     "Э": "E",
74     "Ю": "Yu",
75     "Я": "Ya",
76 }
77
78 result: str = ""
79 for char in text:
80     result += cyrillic_to_latin.get(char, char)
81
82 return result
83
84
85 def transliterate_password_file(input_file: str) -> bool:
86     input_path: Path = Path(input_file)
87     output_file: str = str(
88         input_path.with_name(f"{input_path.stem}_transliterated{input_path.
89             suffix}")
90     )
91
92     output_dir: str = os.path.dirname(output_file)
93     if output_dir and not os.path.exists(output_dir):
94         os.makedirs(output_dir)
95
96     cyrillic_pattern = re.compile(r"[a-яA-ЯёЁ]")
97     processed_count: int = 0
98     transliterated_count: int = 0
99
100    print(f"Processing file: {input_file}")
101    print(f"Output will be saved to: {output_file}")

```

```

101
102     try:
103         with (
104             open(input_file, "r", encoding="utf-8", errors="ignore") as fin,
105             open(output_file, "w", encoding="utf-8") as fout,
106         ):
107             for i, line in enumerate(fin):
108                 if i % 100000 == 0 and i > 0:
109                     print(f"Processed {i} passwords...")
110
111                 password: str = line.strip()
112                 processed_count += 1
113
114                 match: Optional[Match[str]] = cyrillic_pattern.search(password)
115                 if match:
116                     transliterated_password: str = transliterate_cyrillic(
117                         password)
118                     transliterated_count += 1
119                 else:
120                     transliterated_password: str = password
121
122                 fout.write(transliterated_password + "\n")
123
124     except Exception as e:
125         print(f"Error during file processing: {e}")
126         return False
127
128     print(f"passwords processed: {processed_count}")
129     print(f"characters transliterated: {transliterated_count}")
130     print(f"file saved to: {output_file}")
131     return True
132
133 def main() -> None:
134     parser: argparse.ArgumentParser = argparse.ArgumentParser(
135         description="Transliterate Cyrillic characters in a password file"
136     )
137     parser.add_argument("input_file", help="Path to the password file")
138     args: argparse.Namespace = parser.parse_args()

```

```
139
140     if not os.path.exists(args.input_file):
141         print(f"Error: File {args.input_file} does not exist!")
142         return
143
144     transliterate_password_file(args.input_file)
145
146
147 if __name__ == "__main__":
148     main()
```

Лістинг 3: models.py

ДОДАТОК Б ОТРИМАНІ ОЦІНКИ ДЛЯ СТАТИСТИЧНИХ МЕТРИК

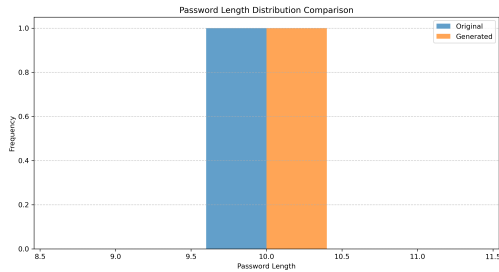


Рисунок Б.1 – Порівняння розподілу довжин паролів (S (Char10))

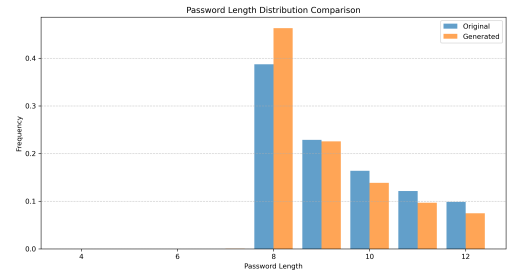


Рисунок Б.2 – Порівняння розподілу довжин паролів (S (Char812))

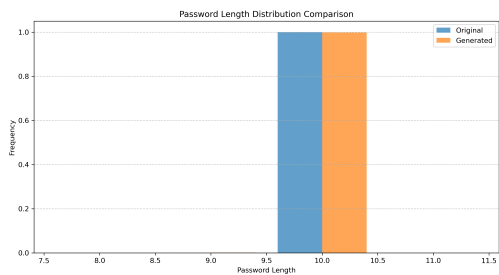


Рисунок Б.3 – Порівняння розподілу довжин паролів (RockYou (Char10))

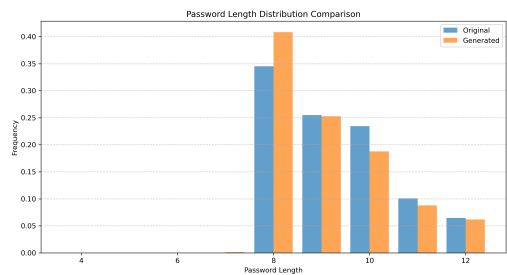


Рисунок Б.4 – Порівняння розподілу довжин паролів (RockYou (Char812))

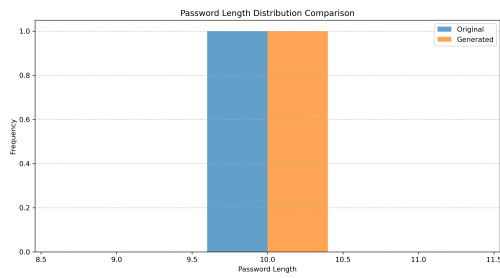


Рисунок Б.5 – Порівняння
розподілу довжин паролів (Yahoo
(Char10))

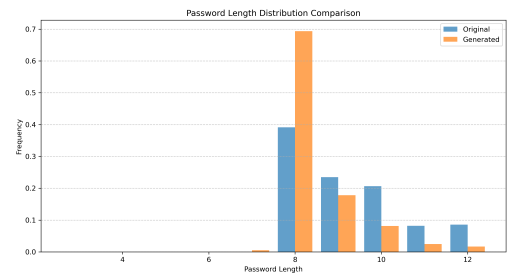


Рисунок Б.6 – Порівняння
розподілу довжин паролів (Yahoo
(Char812))

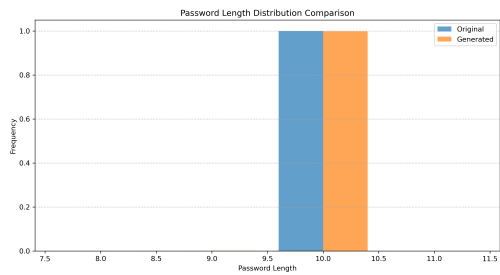


Рисунок Б.7 – Порівняння
розподілу довжин паролів (Gmail
(Char10))

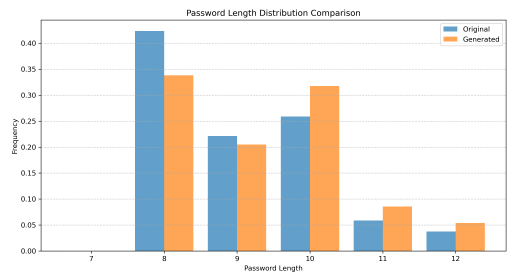


Рисунок Б.8 – Порівняння
розподілу довжин паролів (Gmail
(Char812))

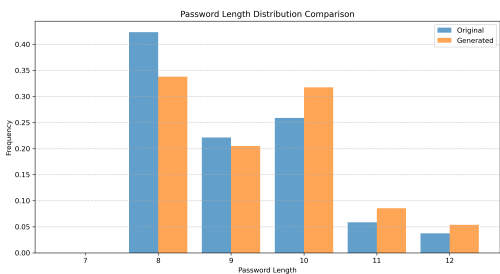


Рисунок Б.9 – Порівняння
розподілу довжин паролів (Cyrillic
(Char812))

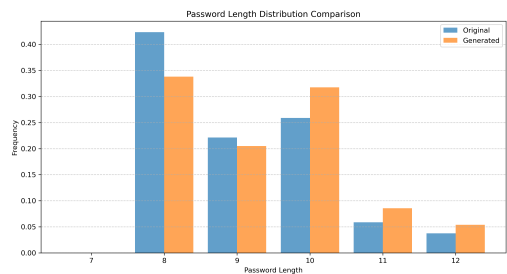


Рисунок Б.10 – Порівняння
розподілу довжин паролів (Cyrillic
Transliterated (Char812))

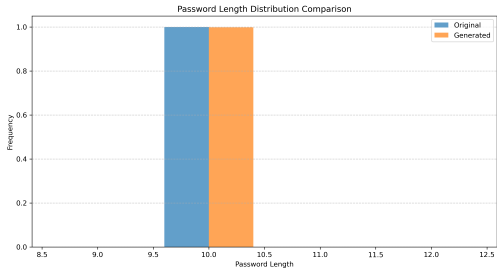


Рисунок Б.11 – Порівняння розподілу довжин паролів (Cyrillic (Char10))

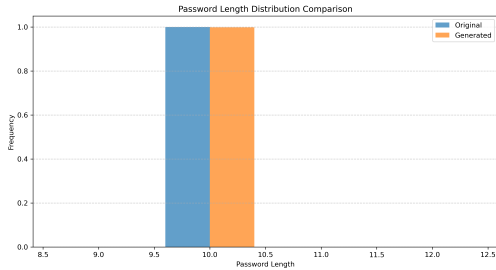


Рисунок Б.12 – Порівняння розподілу довжин паролів (Cyrillic Transliterated (Char10))

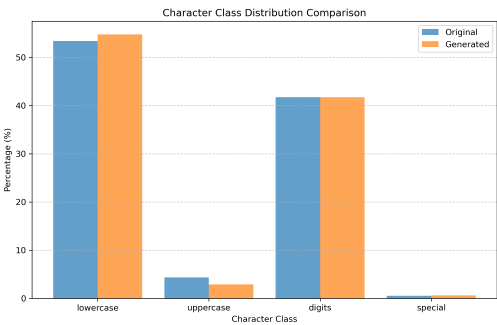


Рисунок Б.13 – Порівняння розподілу класів символів (S (Char10))

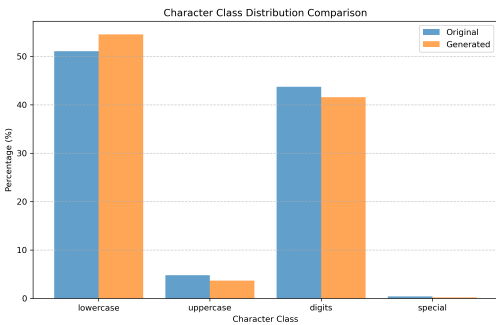


Рисунок Б.14 – Порівняння розподілу класів символів (S (Char812))

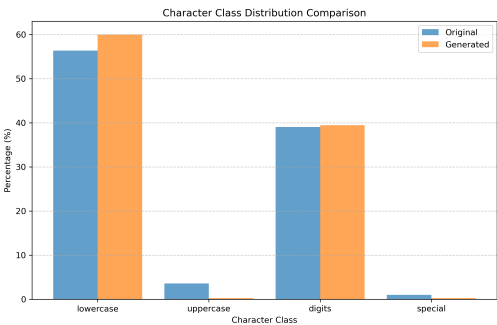


Рисунок Б.15 – Порівняння розподілу класів символів (RockYou (Char10))

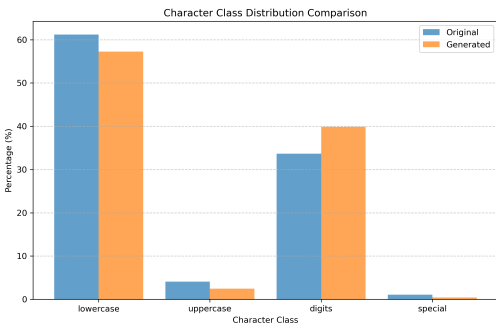


Рисунок Б.16 – Порівняння розподілу класів символів (RockYou (Char812))

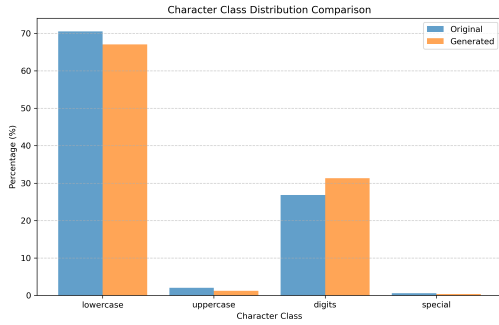


Рисунок Б.17 – Порівняння розподілу класів символів (Yahoo (Char10))

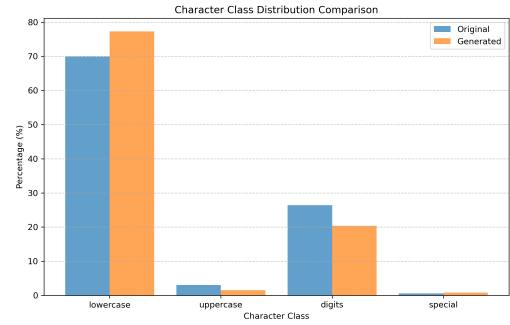


Рисунок Б.18 – Порівняння розподілу класів символів (Yahoo (Char812))

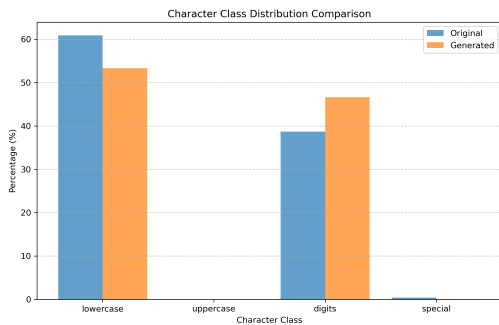


Рисунок Б.19 – Порівняння розподілу класів символів (Gmail (Char10))

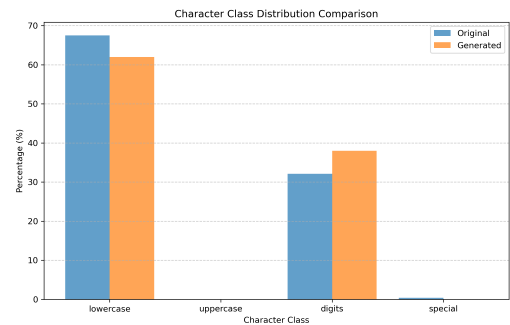


Рисунок Б.20 – Порівняння розподілу класів символів (Gmail (Char812))

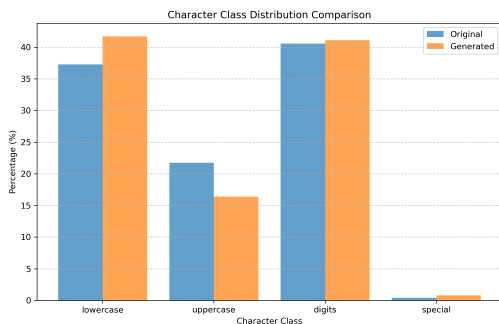


Рисунок Б.21 – Порівняння розподілу класів символів (Cyrillic (Char10))

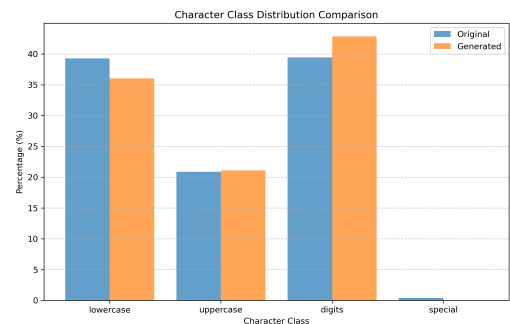


Рисунок Б.22 – Порівняння розподілу класів символів (Cyrillic Transliterated (Char10))

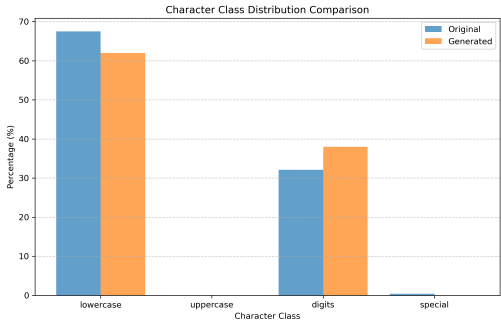


Рисунок Б.23 – Порівняння розподілу класів символів (Cyrillic (Char812))

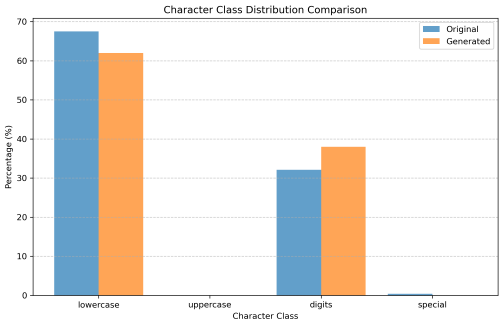


Рисунок Б.24 – Порівняння розподілу класів символів (Cyrillic Transliterated (Char812))

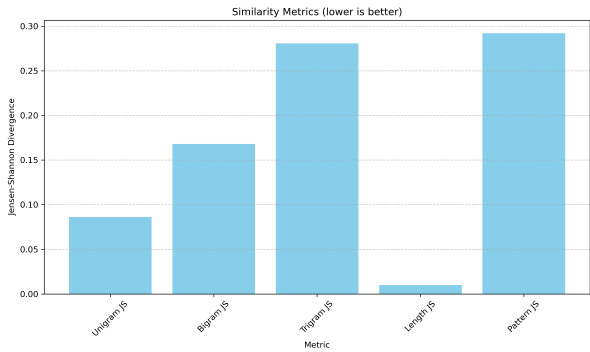


Рисунок Б.25 – Значення дивергенції Єнсена-Шеннона (S (Char10))

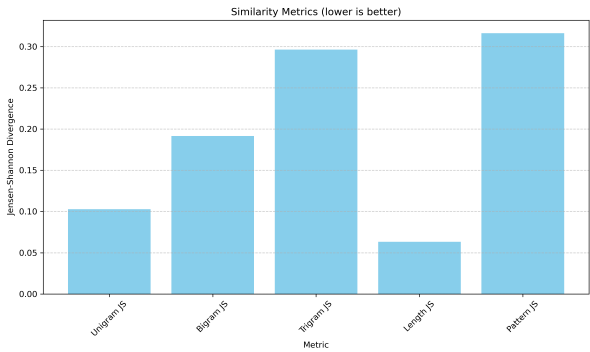


Рисунок Б.26 – Значення дивергенції Єнсена-Шеннона (S (Char812))

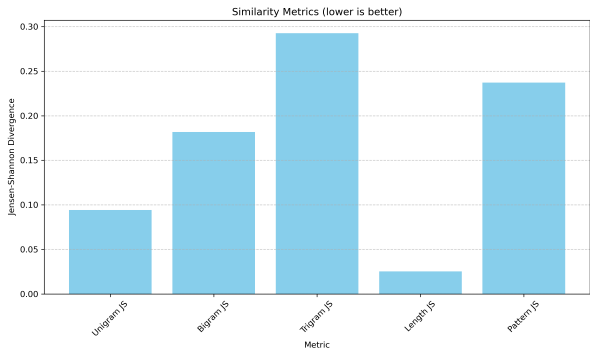


Рисунок Б.27 – Значення дивергенції Єнсена-Шеннона (RockYou (Char10))

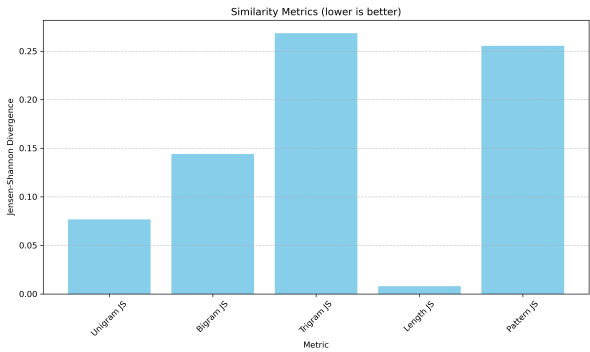


Рисунок Б.28 – Значення дивергенції Єнсена-Шеннона (Yahoo (Char10))

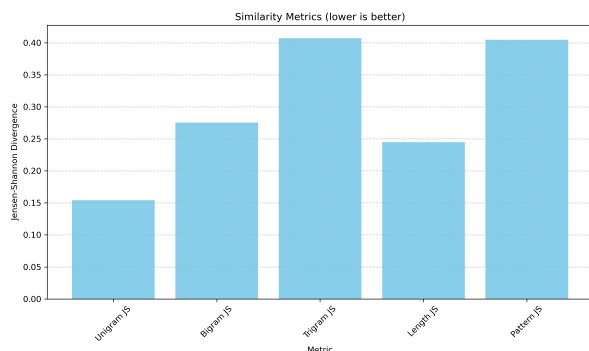


Рисунок Б.29 – Значення дивергенції Єнсена-Шеннона (Yahoo (Char812))

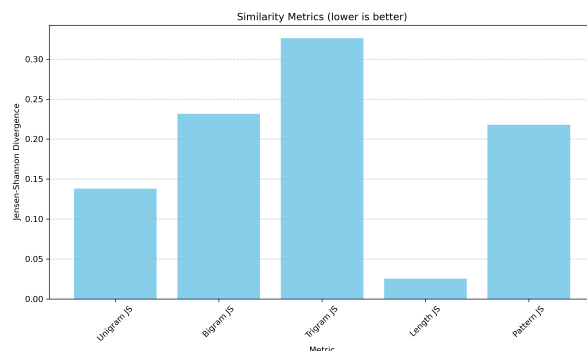


Рисунок Б.30 – Значення дивергенції Єнсена-Шеннона (Gmail (Char10))

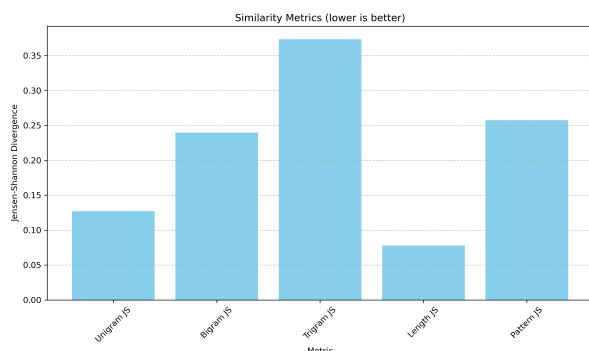


Рисунок Б.31 – Значення дивергенції Єнсена-Шеннона (Gmail (Char812))

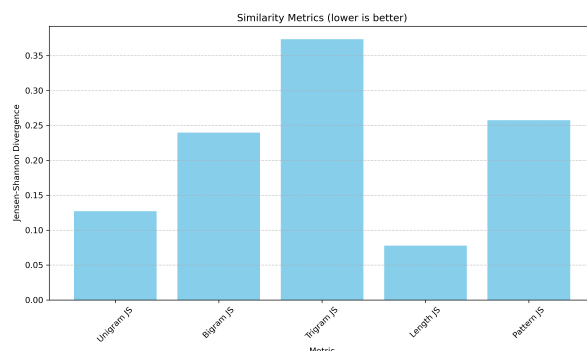


Рисунок Б.32 – Значення дивергенції Єнсена-Шеннона (Cyrillic (Char812))

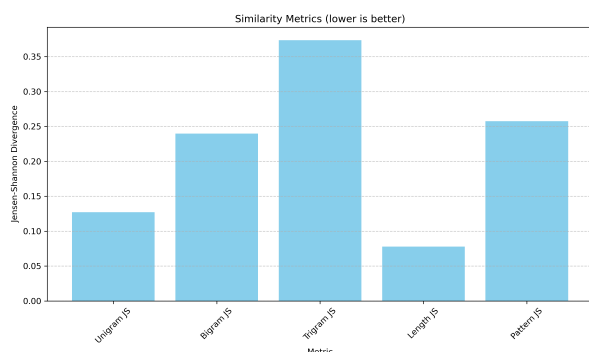


Рисунок Б.33 – Значення дивергенції Єнсена-Шеннона (Cyrillic Transliterated (Char812))

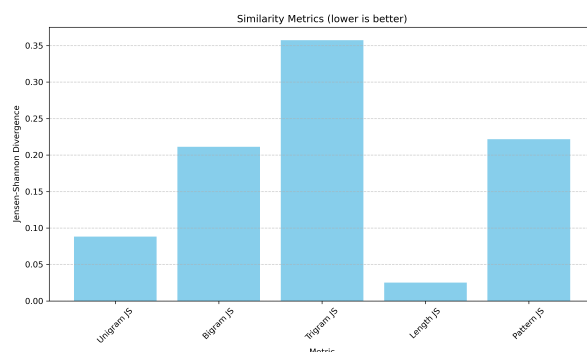


Рисунок Б.34 – Значення дивергенції Єнсена-Шеннона (Cyrillic (Char10))

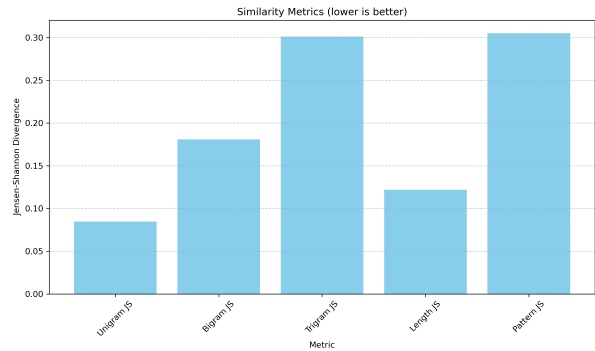


Рисунок Б.35 – Значення дивергенції Єнсена-Шеннона (Cyrillic Transliterated (Char10))

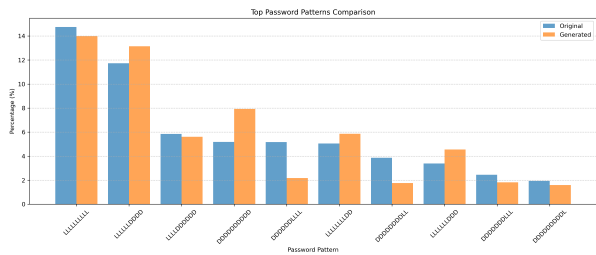


Рисунок Б.36 – Порівняння розподілу найпопулярніших структурних патернів (S (Char10))

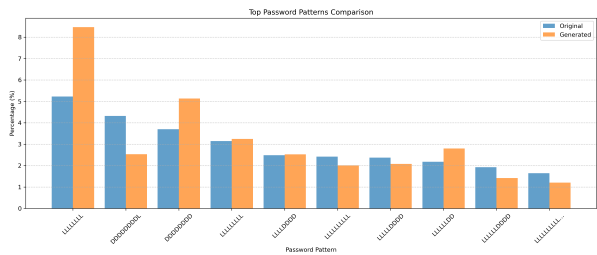


Рисунок Б.37 – Порівняння розподілу найпопулярніших структурних патернів (S (Char812))

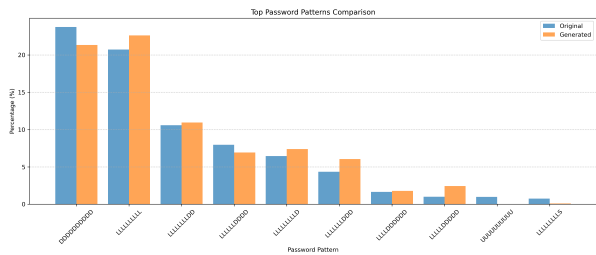


Рисунок Б.38 – Порівняння розподілу найпопулярніших структурних патернів (RockYou (Char10))

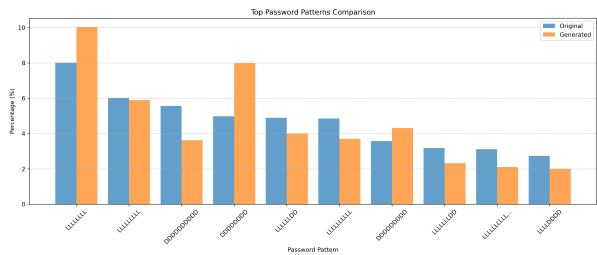


Рисунок Б.39 – Порівняння розподілу найпопулярніших структурних патернів (RockYou (Char812))

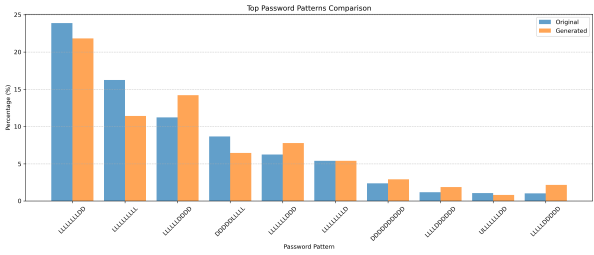


Рисунок Б.40 – Порівняння розподілу найпопулярніших структурних патернів (Yahoo (Char10))

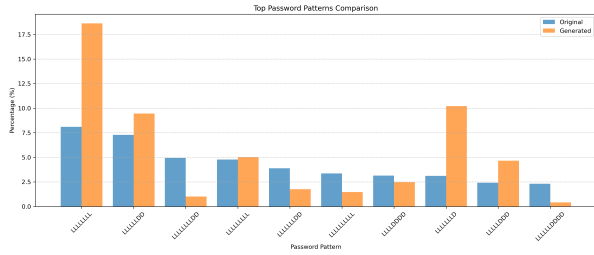


Рисунок Б.41 – Порівняння розподілу найпопулярніших структурних патернів (Yahoo (Char812))

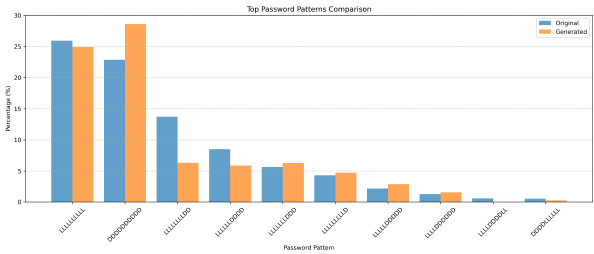


Рисунок Б.42 – Порівняння розподілу найпопулярніших структурних патернів (Gmail (Char10))

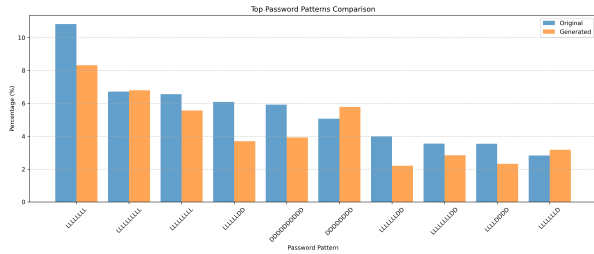


Рисунок Б.43 – Порівняння розподілу найпопулярніших структурних патернів (Gmail (Char812))

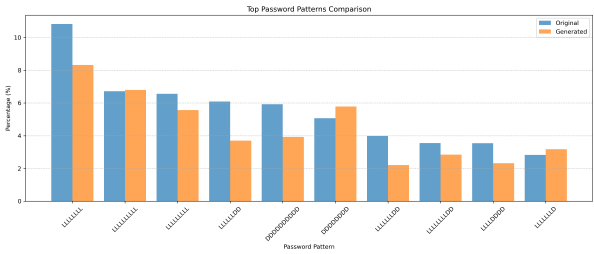


Рисунок Б.44 – Порівняння розподілу найпопулярніших структурних патернів (Cyrillic (Char812))

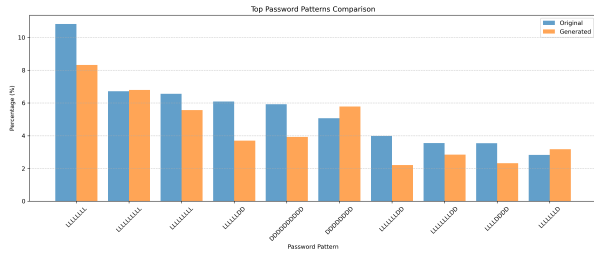


Рисунок Б.45 – Порівняння розподілу найпопулярніших структурних патернів (Cyrillic Transliterated (Char812))

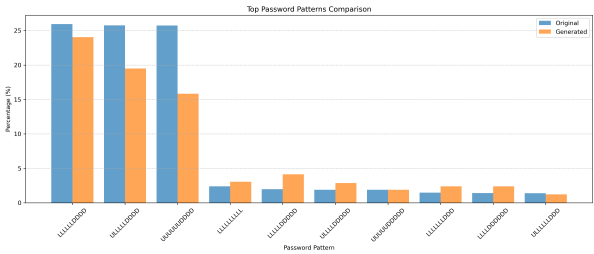


Рисунок Б.46 – Порівняння розподілу найпопулярніших структурних патернів (Cyrillic (Char10))

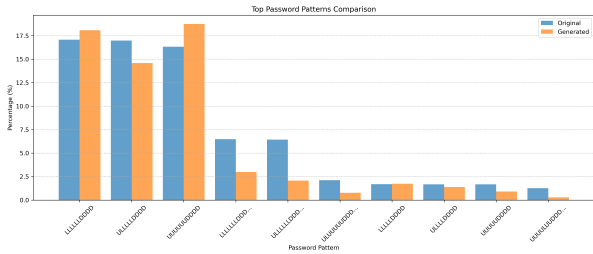


Рисунок Б.47 – Порівняння розподілу найпопулярніших структурних патернів (Cyrillic Transliterated (Char10))