Міністерство освіти і науки України
Київський національний університет імені Тараса Шевченка

Факультет інформаційних технологій
Кафедра кібербезпеки та захисту інформації

ДОПУСТИТИ ДО ЗАХИСТУ:
В.о. завідувача кафедри
кібербезпеки та захисту
інформації
_____ Іван ПАРХОМЕНКО
« » червня 2025 р.

ПОЯСНЮВАЛЬНА ЗАПИСКА

кваліфікаційної роботи

галузь знань 12 Інформаційні технології
(шифр і назва галузі знань)
спеціальність 125 Кібербезпека
(код і назва спеціальності)
освітній ступень бакалавр
освітня програма Кібербезпека
(назва освітньо-професійної програми)
на тему: «Програмний застосунок скремблеру для шифрування данних»

Виконавець: студент IV курсу, групи КБ-41

_____ **Дмитро КРИЛОВ**
(підпис) (ім'я, прізвище)

	Підпис	**Ім'я, ПРІЗВИЩЕ**
Керівник		Юрій БАБЕНКО
Нормоконтроль		Інна МИХАЛЬЧУК

Київ 2025

Міністерство освіти і науки України
Київський національний університет імені Тараса Шевченка

Факультет інформаційних технологій
Кафедра кібербезпеки та захисту інформації

ЗАТВЕРДЖЕНО:
В.о. завідувача кафедри
кібербезпеки
та захисту інформації
_____ Іван ПАРХОМЕНКО
«29» листопада 2024 р.

ЗАВДАННЯ
на виконання кваліфікаційної роботи

спеціальності 125 Кібербезпека
(код і назва спеціальності)

освітньої програми Кібербезпека
(назва освітньо-професійної програми)

Студенту **КБ-41** (група) **Крилову Дмитру Олексійовичу** (прізвище ім'я по батькові)

Тема кваліфікаційної роботи Програмний застосунок скремблеру для шифрування данних

1. ПІДСТАВИ ДЛЯ ПРОВЕДЕННЯ РОБОТИ

Тема кваліфікаційної роботи затверджена на засіданні кафедри кібербезпеки та захисту інформації протокол №6 від 28.11.2024 р.

2. ВИХІДНІ ДАНІ ДЛЯ ПРОВЕДЕННЯ РОБІТ

Наявні методи скремблювання.
Потреба у розробці гнучкого програмного інструменту для обфускації даних.
Наявний стек технологій (.NET, Vue.js) для реалізації.

3. ЗМІСТ РОЗРАХУНКОВО-ПОЯСНЮВАЛЬНОЇ ЗАПИСКИ

Теоретичні основи захисту інформації та аналіз методів програмного скремблювання, розробка та теоретичне обґрунтування програмного скремблеру для захисту інформації, програмна реалізація та експериментальне дослідження скремблера даних

4. ВИМОГИ ДО РЕЗУЛЬТАТІВ ВИКОНАННЯ РОБОТИ

Практична цінність Розроблений програмний веб-застосунок для базового захисту текстових повідомлень та файлів

5. ДАТА ВИДАЧІ ЗАВДАННЯ

Дата видачі завдання: 29 листопада 2024 року

Завдання видав _____ (підпис) Юрій БАБЕНКО (ім'я, прізвище)

Завдання прийняв до виконання _____ (підпис) Дмитро КРИЛОВ (ім'я, прізвище)

КАЛЕНДАРНИЙ ПЛАН

№ п/п	Найменування етапів робіт	Строки виконання робіт (початок-кінець)	Відмітка про виконання
1	Уточнення постановки задачі	29.11.2024 – 15.01.2025	*виконано*
2	Аналіз літератури	16.01.2025 – 13.02.2025	*виконано*
3	Обґрунтування вибору рішення	14.02.2025 – 17.02.2025	*виконано*
4	Дослідження принципів скремблювання	18.02.2025 – 03.03.2025	*виконано*
5	Аналіз проблемних аспектів та невирішених задач у галузі програмного скремблювання	02.03.2025 – 24.03.2025	*виконано*
6	Вибір до аналіз алгоритмів скремблювання	25.03.2025 – 02.04.2025	*виконано*
7	Розробка веб-застосунку	03.04.2025 – 10.05.2025	виконано
8	Оформлення пояснювальної записки	11.05.2025 – 28.05.2025	*виконано*
9	Підготовка до захисту кваліфікаційної роботи	29.05.2025 – 13.06.2025	*виконано*

Завдання видав _____ (підпис) Юрій БАБЕНКО (ім'я, прізвище)

Завдання прийняв до виконання _____ (підпис) Дмитро КРИЛОВ (ім'я, прізвище)

Термін подання кваліфікаційної роботи до ЕК 13 червня 2025 року

РЕФЕРАТ

Кваліфікаційна робота складається зі вступу, трьох розділів, загальних висновків, списку використаних джерел та 7 додатків. Робота має 87 сторінок основного тексту, 1 таблицю, 7 формули та 15 рисунків. Список використаних джерел містить 26 найменувань і займає 3 сторінки.

Об’єкт дослідження – процес захисту інформації шляхом її перетворення методами програмного скремблювання.

Мета роботи – дослідження методів програмного скремблювання, розробка програмного засобу, що реалізує обрані алгоритми та їх комбінації, та аналіз його властивостей.

Предмет дослідження – алгоритми програмного скремблювання (XOR, блокова перестановка, адитивне перетворення), методи їх комбінованого застосування.

Методи дослідження: аналіз науково-технічної літератури, системний аналіз, методи теорії алгоритмів, об'єктно-орієнтоване проектування та програмування, експериментальне дослідження.

Практична цінність – розроблений програмний веб-додаток для скремблювання даних, придатний для базового захисту інформації від випадкового перегляду, для освітніх та демонстраційних цілей.

Ключові слова: скремблювання даних, програмний скремблер, захист інформації, алгоритм XOR, блокова перестановка, адитивне перетворення, каскадне скремблювання, веб-додаток, C#, .NET.

ВСТУП

В умовах стрімкої цифровізації суспільства та експоненційного зростання обсягів інформації, що генерується, обробляється та зберігається, проблема забезпечення її надійного захисту набуває критичної важливості. Щоденно збільшується кількість та ускладнюється характер кіберзагроз, що ставить під удар конфіденційність, цілісність та доступність даних як окремих громадян, так і організацій та державних інституцій.

Актуальність теми даної кваліфікаційної роботи зумовлена тим, що хоча для протидії складним атакам існують потужні криптографічні системи, залишається потреба в дослідженні та розробці більш простих, гнучких та швидких методів перетворення даних, таких як програмне скремблювання. Ці методи можуть знайти своє застосування у специфічних нішах, зокрема для маскування даних, захисту від випадкового несанкціонованого перегляду або для вирішення технічних завдань, де не висуваються вимоги до високого рівня криптографічної стійкості. Недостатня вивченість ефективності комбінованого застосування різних алгоритмів програмного скремблювання та властивостей отримуваних при цьому перетворень визначає необхідність проведення даного дослідження.

Для досягнення поставленої мети було визначено наступні *завдання дослідження:*

1. Провести огляд та аналіз наявних принципів, класифікацій та алгоритмів, що можуть використовуватися для програмного скремблювання даних;

2. Розробити теоретичні основи функціонування обраних алгоритмів скремблювання, включаючи принципи їх комбінованого застосування та використання ключів;

3. Спроектувати та реалізувати програмний засіб у вигляді веб-додатку, що дозволяє виконувати скремблювання та дескремблювання текстових даних і файлів з використанням обраних алгоритмів та їх послідовностей;

4. Провести експериментальне дослідження розробленого програмного засобу для оцінки його функціональності та швидкодії.

Для вирішення поставлених завдань у роботі були використані наступні *методи дослідження*: аналіз науково-технічної літератури для вивчення стану проблеми та наявних підходів; системний аналіз та порівняння для класифікації методів та засобів; методи теорії алгоритмів для опису та обґрунтування обраних алгоритмів скремблювання; об'єктно-орієнтоване проектування та програмування для розробки архітектури та реалізації програмного скремблера; експериментальне дослідження для тестування розробленого програмного засобу та оцінки його характеристик.

Практична значущість одержаних результатів полягає у створенні програмного веб-додатку, який може бути використаний для:

- базового захисту текстових повідомлень та файлів від випадкового несанкціонованого перегляду;
- перетворення даних для навчальних або демонстраційних цілей, що ілюструють принципи роботи простих алгоритмів зміни інформації;
- як інструмент для подальших досліджень властивостей комбінованого скремблювання. Розроблений програмний скремблер може знайти застосування в ситуаціях, де не вимагається високий рівень криптографічної стійкості, але важливі швидкість, простота використання та гнучкість у виборі методів перетворення.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ ТА СКОРОЧЕНЬ

БД – База даних;

ГПВЧ – Генератор псевдовипадкових чисел;

ІБ – Інформаційна безпека;

НСД – Несанкціонований доступ;

ОС – Операційна система;

ПЗ – Програмне забезпечення;

API – Application Programming Interface;

CIA – Confidentiality, Integrity, Availability;

CPU – Central Processing Unit;

CSS – Cascading Style Sheets;

DI – Dependency Injection;

DDoS – Distributed Denial of Service;

DLP – Data Loss Prevention;

GUI – Graphical User Interface;

HEX – Шістнадцятковий формат представлення даних;

HTML – HyperText Markup Language;

HTTP – HyperText Transfer Protocol;

IDE – Integrated Development Environment;

IoT – Internet of Things;

JSON – JavaScript Object Notation;

LFSR – Linear Feedback Shift Register;

RAM – Random Access Memory;

URL – Uniform Resource Locator;

XOR – Операція "виключне АБО";

РОЗДІЛ 1
ТЕОРЕТИЧНІ ОСНОВИ ЗАХИСТУ ІНФОРМАЦІЇ ТА АНАЛІЗ МЕТОДІВ ПРОГРАМНОГО СКРЕМБЛЮВАННЯ

1.1 Сучасний стан проблеми захисту інформації та роль скремблювання

В умовах стрімкого розвитку інформаційних технологій та глобалізації інформаційного простору, проблема забезпечення надійного захисту даних набуває виняткової гостроти та актуальності для всіх аспектів сучасного життя – від особистої безпеки громадян до функціонування критичної інфраструктури та національної безпеки в цілому. Цей розділ присвячений аналізу сучасного стану зазначеної проблеми, висвітленню ключових понять та термінології, класифікації основних загроз інформаційній безпеці.

1.1.1 Актуальність захисту інформації в сучасних умовах

В епоху цифрової трансформації, коли інформація перетворилася на один із найцінніших активів як для окремих осіб, так і для організацій та держав, питання її захисту набуває безпрецедентної актуальності. Сучасний світ характеризується експоненційним зростанням обсягів генерованих, оброблюваних та збережених даних, що охоплюють усі сфери людської діяльності – від персонального спілкування та фінансових транзакцій до функціонування критичної інфраструктури та забезпечення національної безпеки. Ця «інформаційна революція» несе не лише величезні можливості, але й породжує нові, дедалі складніші загрози.

Ключові фактори, що зумовлюють надзвичайну актуальність захисту інформації сьогодні, включають наступне:

1. Спостерігається лавиноподібне зростання обсягів даних. Щодня генеруються величезні масиви (Big Data) завдяки поширенню мобільних пристроїв, соціальних мереж, Інтернету речей, хмарних технологій та цифрових сервісів. Ці дані часто містять конфіденційну, комерційну, персональну або державну таємницю, що робить їх привабливою ціллю для зловмисників.

2. Відбувається постійне збільшення взаємопов'язаності систем та, як наслідок, розширення так званої «поверхні атаки». Глобальна мережа Інтернет, хмарні сервіси, корпоративні та державні мережі створюють складну, взаємопов'язану інфраструктуру. Кожен новий підключений пристрій, кожна нова програма чи сервіс потенційно збільшують кількість точок, через які може бути здійснена атака.

3. Триває невпинна еволюція та ускладнення кіберзагроз. Кіберзлочинність постійно розвивається, стаючи все більш організованою, технологічно оснащеною та фінансово мотивованою. Зростає кількість та витонченість таких загроз, як програми-вимагачі (ransomware), фішингові атаки, цілеспрямовані атаки на організації, використання штучного інтелекту для створення діпфейків та проведення автоматизованих атак, атаки на ланцюги постачання програмного забезпечення.

4. Успішні кібератаки призводять до значних економічних та репутаційних втрат. Це включає прямі фінансові збитки (крадіжка коштів, витрати на відновлення систем, штрафи регуляторів), а також значні непрямі втрати, такі як пошкодження ділової репутації, втрата довіри клієнтів та партнерів, зниження ринкової капіталізації.

5. Існують серйозні загрози для приватного життя та прав людини. Витоки персональних даних можуть призвести до крадіжки особистості, фінансового шахрайства, незаконного стеження, дискримінації та інших порушень основоположних прав і свобод громадян.

6. Кібератаки мають значний вплив на критичну інфраструктуру та національну безпеку. Втручання в роботу об'єктів критичної інфраструктури (енергетика, транспорт, охорона здоров'я, фінансові установи, державне

управління) можуть мати катастрофічні наслідки, призводячи до збоїв у їх роботі, соціальної напруги та навіть загрожуючи життю людей. Для держав інформаційна безпека є невід'ємною складовою національної безпеки, особливо в умовах гібридних воєн та геополітичної нестабільності.

7. Спостерігається посилення законодавчого та регуляторного тиску у сфері захисту даних. У відповідь на зростаючі загрози багато країн та міжнародних об'єднань впроваджують та посилюють законодавство у сфері захисту даних та кібербезпеки (наприклад, Загальний регламент про захист даних в ЄС, відповідні закони України «Про захист інформації в інформаційно-комунікаційних системах», «Про основні засади забезпечення кібербезпеки України», «Про захист персональних даних»). Недотримання цих вимог може призвести до значних штрафних санкцій.

1.1.2 Основні поняття та термінологія в галузі інформаційної безпеки та скремблювання

Для глибокого розуміння проблематики захисту інформації та, зокрема, методів програмного скремблювання, необхідно чітко визначити ключові поняття та термінологію, що використовуються в даній галузі. Це забезпечить однозначність трактування та сприятиме коректному викладенню матеріалу.

Інформаційна безпека – це стан захищеності інформації та інфраструктури, що її підтримує, від випадкових або навмисних впливів природного чи штучного характеру, які можуть завдати шкоди власникам або користувачам інформації. Основними властивостями або атрибутами інформаційної безпеки, які необхідно забезпечувати, традиційно вважаються:

Конфіденційність – властивість інформації бути недоступною або закритою для неавторизованих осіб, суб'єктів або процесів. Забезпечення конфіденційності спрямоване на запобігання несанкціонованому розкриттю даних.

Цілісність – властивість інформації зберігати свою структуру та зміст у процесі передачі, зберігання та обробки. Це означає забезпечення захисту від несанкціонованої модифікації або знищення даних, а також гарантію точності та повноти інформації.

Доступність – властивість інформації бути доступною та готовою до використання на запит авторизованого користувача в потрібний час. Забезпечення доступності спрямоване на запобігання відмовам в обслуговуванні та забезпечення своєчасного доступу до ресурсів.

Окрім цієї класичної «тріади CIA» (Confidentiality, Integrity, Availability), часто розглядають і такі важливі аспекти, як автентичність (authenticity) – гарантія того, що суб'єкт або ресурс є саме тим, за кого себе видає; невідмовність або неспростовність (non-repudiation) – неможливість для суб'єкта відмовитися від факту виконання певних дій або авторства повідомлення; та підзвітність (accountability) – можливість однозначно відстежувати дії суб'єктів у системі.

У контексті інформаційної безпеки важливими є також наступні терміни:

Актив – будь-що, що має цінність для організації або особи та потребує захисту (наприклад, дані, програмне забезпечення, обладнання, репутація).

Загроза – будь-яка потенційна причина інциденту інформаційної безпеки, що може завдати шкоди системі або організації шляхом порушення конфіденційності, цілісності чи доступності інформації.

Вразливість – слабке місце в активі, системі безпеки, процедурах, внутрішньому контролі або реалізації, яке може бути використане загрозою.

Ризик – міра ймовірності того, що загроза використає вразливість системи, та величина потенційних збитків або шкоди, що можуть виникнути внаслідок цього.

Атака – будь-яка спроба реалізації загрози, спрямована на знищення, викриття, зміну, виведення з ладу, викрадення або отримання несанкціонованого доступу до активу чи його несанкціонованого використання.

Захід захисту або контроль – засіб або процедура, що зменшує ризик шляхом зниження ймовірності реалізації загрози, зменшення впливу вразливості або мінімізації наслідків атаки.

Важливу роль у забезпеченні, зокрема, конфіденційності та автентичності інформації відіграє криптографія. Криптографія – це наука про математичні методи забезпечення конфіденційності, цілісності даних, автентифікації та неспростовності. Вона вивчає методи перетворення (шифрування) інформації з метою її захисту від несанкціонованого доступу.

Шифрування – процес перетворення вихідної інформації (відкритого тексту) у зашифровану форму (шифротекст) за допомогою певного алгоритму та, зазвичай, ключа, з метою зробити її незрозумілою для осіб, які не мають права доступу.

Дешифрування – процес зворотного перетворення шифротексту у відкритий текст за допомогою відповідного алгоритму та ключа.

Шифр – сукупність алгоритмів криптографічного перетворення (шифрування та дешифрування) та ключів, що використовуються для цього.

Ключ – секретний параметр (послідовність символів, число тощо), що використовується алгоритмом шифрування для перетворення відкритого тексту в шифротекст і навпаки.

У контексті даної роботи центральним є поняття скремблювання. Скремблювання (scrambling) – це процес перетворення послідовності даних з метою приведення її до вигляду, подібного до випадкової послідовності, або з метою ускладнення її інтерпретації без відповідного алгоритму відновлення (дескремблювання). На відміну від класичного шифрування, яке зазвичай спирається на складні математичні алгоритми та секретні ключі для забезпечення високого рівня криптографічної стійкості, скремблювання часто використовує простіші перетворення, такі як перестановка бітів або сегментів даних, інверсія сигналу, або застосування псевдовипадкових послідовностей для модифікації вихідних даних. Основні цілі скремблювання можуть включати:

1. Ускладнення несанкціонованого доступу до інформації або її випадкового перегляду.

2. Забезпечення технічних вимог у системах зв'язку (наприклад, усунення довгих послідовностей однакових бітів для стабільної роботи синхронізації приймача).

3. Маскування або заплутування даних (data obfuscation) для зниження їх чутливості без забезпечення строгої криптографічної гарантії.

Слід зазначити, що терміни «скремблювання» та «шифрування» іноді використовуються як взаємозамінні в популярній літературі, однак у технічному та науковому контексті скремблювання зазвичай розглядається як метод, що забезпечує нижчий рівень захисту порівняно з криптографічним шифруванням, і може не завжди використовувати ключі у строгому криптографічному сенсі.

Скремблер – пристрій або програмний алгоритм, що виконує операцію скремблювання.

Дескремблер – пристрій або програмний алгоритм, що виконує операцію, зворотну скремблюванню, тобто відновлює вихідну послідовність даних.

Використання чіткої та узгодженої термінології є фундаментальною передумовою для якісного дослідження у сфері інформаційної безпеки та розробки ефективних засобів захисту інформації.

1.1.3 Місце скремблювання серед інших методів захисту інформації

Забезпечення інформаційної безпеки є комплексним завданням, що вимагає застосування різноманітних методів та засобів, які діють на різних рівнях та протидіють різним типам загроз. Скремблювання, як один із методів перетворення інформації, займає свою специфічну нішу в цьому арсеналі. Для чіткого розуміння його ролі, доцільно розглянути його у порівнянні з іншими поширеними підходами до захисту.

Методи захисту інформації умовно можна поділити на декілька великих груп: криптографічні (шифрування, гешування, електронний цифровий підпис);

технічні програмно-апаратні засоби (контроль доступу, міжмережеві екрани, системи виявлення/запобігання вторгнень, антивірусне ПЗ); організаційні заходи (політики безпеки, навчання персоналу, фізична охорона); правові заходи (законодавство, стандарти); та стеганографічні методи (приховування факту існування інформації).

Скремблювання в цій системі методів часто розглядається як техніка обробки даних, що межує з криптографією, але зазвичай має простішу реалізацію та, відповідно, пропонує інший рівень захисту. Його місце визначається наступними характеристиками та сферами застосування:

1. Стосовно співвідношення з криптографічним шифруванням, скремблювання, як правило, забезпечує нижчий рівень конфіденційності порівняно зі стійкими криптографічними алгоритмами. Воно спрямоване радше на ускладнення розуміння даних для непідготовленого спостерігача, ніж на протидію цілеспрямованим атакам. Якщо скремблювання і використовує аналоги ключів, вони часто є простішими або фіксованими, на відміну від секретних криптографічних ключів. Основна мета стійкого шифрування – надійна конфіденційність, тоді як скремблювання може мати й інші цілі. Його можна розглядати як один із методів обфускації даних.

2. Щодо типових сфер застосування скремблювання, однією з найпоширеніших є технічні потреби в системах зв'язку (модеми, цифрове ТБ, супутниковий зв'язок, мережі передачі даних). Тут його завдання – забезпечення якості сигналу для надійної синхронізації та вирівнювання енергетичного спектру; алгоритми часто стандартизовані та публічні. Також воно може застосовуватися для захисту від випадкового перегляду при тимчасовому зберіганні даних, для маскування даних при тестуванні ПЗ або в простих системах захисту з дуже обмеженими обчислювальними ресурсами.

3. Аналізуючи переваги та недоліки, до переваг зазвичай відносять простоту реалізації та низькі обчислювальні затрати. Основним недоліком є низька стійкість до цілеспрямованого криптоаналізу, що робить його

непридатним для захисту критично важливої інформації від кваліфікованих зловмисників.

Отже, скремблювання не є заміною надійним криптографічним методам захисту. Воно займає нішу допоміжного засобу, ефективного для вирішення специфічних завдань: технічного забезпечення якості сигналу, базового маскування даних або захисту від випадкового несанкціонованого доступу. В рамках комплексної системи «захисту в глибину» скремблювання може використовуватися як один із рівнів, але не як основний бар'єр. Розуміння його можливостей та обмежень є ключовим для коректного застосування, особливо в сучасних умовах, коли вимоги до надійності захисту інформації є надзвичайно високими.

1.2 Огляд наявних методів та засобів скремблювання даних

Після розгляду загального контексту проблеми захисту інформації, цей розділ зосереджується на детальному дослідженні безпосередньо методів та засобів скремблювання даних як одного зі способів зміни інформації з метою її захисту або для вирішення специфічних технічних завдань. В рамках даного розділу будуть проаналізовані фундаментальні принципи, що лежать в основі процесів скремблювання, розглянуто історичні етапи розвитку цих методів. Окрім того, буде представлена класифікація наявних типів скремблерів та проведено огляд поширених алгоритмів і програмних інструментів, що використовуються для програмного скремблювання.

1.2.1 Принципи скремблювання

Скремблювання даних, як метод їх перетворення, ґрунтується на наборі фундаментальних принципів та операцій, що спрямовані на зміну вихідної послідовності бітів або символів таким чином, щоб ускладнити її безпосереднє сприйняття або використання без знання алгоритму відновлення. Хоча

скремблювання часто не має на меті забезпечення такого ж високого рівня конфіденційності, як криптографічне шифрування, його принципи широко застосовуються для вирішення специфічних завдань, зокрема в телекомунікаціях для покращення характеристик сигналу, а також для базового маскування чи обфускації даних.

Основними операціями, на яких базуються принципи скремблювання, є перестановка, заміна та використання логічних операцій з псевдовипадковими послідовностями. Перестановка (Permutation) як принцип скремблювання полягає у зміні порядку слідування елементів даних (окремих бітів, байтів або більших блоків) за певним, заздалегідь визначеним правилом. Це може бути проста інверсія послідовності, циклічний зсув або складніша таблична перестановка. Результатом є «перемішування» вихідних даних, що може ускладнити їх аналіз, якщо правило перестановки невідоме. Заміна або підстановка (Substitution) передбачає заміну одних елементів даних на інші згідно з певним алгоритмом або таблицею відповідності. Наприклад, кожен біт або байт може бути замінений на інший, інвертований (шляхом операції XOR з маскою з одиниць) або перетворений за допомогою простої арифметичної чи логічної функції.

Одним із найпоширеніших принципів, особливо для потокового скремблювання, є використання логічних операцій, найчастіше операції «виключне АБО» (XOR), з псевдовипадковою бітовою послідовністю. Ця псевдовипадкова послідовність, яку також називають скремблюючою або ключовим потоком, генерується спеціальним пристроєм або алгоритмом, часто на основі лінійних зсувних регістрів зі зворотним зв'язком (LFSR). LFSR дозволяють генерувати довгі бітові послідовності з хорошими статистичними властивостями, які близькі до випадкових, при цьому маючи відносно просту реалізацію. Характеристики генерованої послідовності (її період, складність) визначаються довжиною зсувного регістру та обраними відводами (позиціями бітів, що використовуються для зворотного зв'язку), які математично описуються генераторним поліномом.

Залежно від того, як формується скремблююча послідовність і як вона взаємодіє з вихідним потоком даних, розрізняють два основні типи скремблерів за принципом їх дії:

1. Адитивні (самосинхронізуючі) скремблери:

В таких системах вихідний скрембльований біт формується шляхом логічної операції (зазвичай XOR) між поточним вхідним бітом даних та бітом, отриманим від генератора псевдовипадкової послідовності. При цьому сам генератор на передавальній стороні часто використовує для свого зворотного зв'язку вже скрембльовані дані. Ключовою особливістю дескремблера такого типу є його здатність до самосинхронізації: після отримання певної кількості коректних бітів він автоматично входить у правильний стан, навіть якщо початкова синхронізація була втрачена або відбулися помилки передачі. Однак, недоліком є те, що одиночна помилка в каналі зв'язку може призвести до появи декількох помилок на виході дескремблера (ефект розмноження помилок), хоча цей ефект є обмеженим.

2. Множинні (синхронні) скремблери або скремблери з початковою синхронізацією:

У цих системах скремблююча послідовність генерується незалежно від потоку вхідних даних. Генератори псевдовипадкових послідовностей на передавальній та приймальній сторонах мають бути ідентичними та синхронізовані між собою. Синхронізація зазвичай досягається шляхом передачі спеціальної початкової послідовності (початкового стану, «seed») або використання фіксованого початкового стану. Скрембльовані дані отримуються шляхом логічної операції (XOR) між вхідним потоком даних та генерованою псевдовипадковою послідовністю. Перевагою такого підходу є те, що одиночна помилка в каналі зв'язку призводить лише до однієї помилки на виході дескремблера (немає розмноження помилок). Однак, втрата синхронізації між генераторами може призвести до повної некоректності дескрембльованих даних до моменту відновлення синхронізації.

Незалежно від типу, фундаментальними принципами будь-якого скремблювання є детермінізм (однакові вхідні дані та параметри завжди дають однаковий результат скремблювання) та оборотність (можливість повного відновлення вихідних даних за допомогою відповідного процесу дескремблювання).

Важливо відзначити, що принципи скремблювання відрізняються від принципів сильного криптографічного шифрування. Скремблери зазвичай не використовують складних багатораундових перетворень, характерних для сучасних блокових шифрів. Якщо в скремблюванні і присутній елемент секретності, він може полягати в самому алгоритмі (що суперечить принципу Керкгоффса, який стверджує, що стійкість криптосистеми повинна залежати лише від секретності ключа, а не алгоритму) або ж алгоритм є відкритим і стандартизованим, як у багатьох телекомунікаційних системах, де основна мета не конфіденційність, а технічне забезпечення якості сигналу.

Розуміння цих базових принципів є ключовим для аналізу наявних методів скремблювання та розробки нових програмних рішень у цій галузі.

1.2.2 Класифікація скремблерів (апаратні, програмні)

Скремблери, як засоби перетворення даних, можуть бути класифіковані за різними ознаками, що відображають особливості їхньої побудови, принципу дії, типу оброблюваної інформації та сфери застосування. Розуміння цих класифікацій дозволяє краще орієнтуватися в різноманітті наявних рішень та обирати найбільш доцільні варіанти для конкретних завдань. Однією з ключових та найбільш загальних є класифікація за способом реалізації, яка поділяє скремблери на апаратні та програмні.

Апаратні скремблери реалізуються у вигляді окремих фізичних пристроїв, спеціалізованих мікросхем (наприклад, ASIC – Application-Specific Integrated Circuit, або FPGA – Field-Programmable Gate Array) або як вбудовані модулі в телекомунікаційному чи іншому обладнанні. Їхньою головною перевагою

традиційно вважається висока швидкість обробки даних, що досягається за рахунок апаратної оптимізації та можливості паралельної обробки сигналів. Це робить їх ефективними для роботи з високошвидкісними потоками даних у реальному часі. Також апаратна реалізація може забезпечувати вищий рівень фізичної захищеності самого пристрою від несанкціонованого втручання та не створює додаткового навантаження на центральний процесор системи, в яку вони інтегровані. Однак, апаратні скремблери зазвичай менш гнучкі: зміна або оновлення алгоритму скремблювання після виробництва є складним або неможливим процесом. Їх розробка та виробництво, особливо для малих партій, можуть бути значно дорожчими порівняно з програмними аналогами. Прикладами апаратних скремблерів є модулі в модемах, телевізійних декодерах систем умовного доступу, спеціалізованому військовому та комерційному обладнанні зв'язку.

Програмні скремблери, на відміну від апаратних, являють собою комп'ютерні програми, програмні бібліотеки, скрипти або модулі, що виконуються на універсальних обчислювальних пристроях (персональні комп'ютери, сервери, мікроконтролери). Основною перевагою програмних рішень є їхня висока гнучкість: алгоритми скремблювання можна легко модифікувати, оновлювати та адаптувати до конкретних потреб користувача або змін у вимогах безпеки. Вартість розробки та розгортання програмних скремблерів зазвичай нижча, і вони легше інтегруються з іншим програмним забезпеченням та інформаційними системами. Проте їхня швидкодія безпосередньо залежить від потужності процесора, ефективності програмної реалізації та загального завантаження системи. Програмні скремблери також можуть бути більш вразливими до різноманітних програмних атак, таких як віруси, троянські програми, або аналіз коду з метою реверс-інжинірингу. До прикладів програмних скремблерів можна віднести утиліти для обфускації даних у базах даних при тестуванні, інструменти для «захисту» файлів від випадкового перегляду, компоненти систем захисту програмного коду, а також програмні реалізації різноманітних скремблюючих алгоритмів, що розробляються для

дослідницьких цілей або специфічних прикладних завдань, що є актуальним напрямком досліджень.

Окрім поділу на апаратні та програмні, скремблери можна класифікувати й за іншими критеріями:

- За принципом дії: як було детально розглянуто в підрозділі 1.2.1, виділяють адитивні (самосинхронізуючі) скремблери, здатні до самостійного відновлення синхронізації, та множинні (синхронні) скремблери, які потребують зовнішньої або початкової синхронізації генераторів псевдовипадкових послідовностей на передавальній та приймальній сторонах.
- За типом оброблюваних даних: існують скремблери, призначені для цифрових даних (обробка бітових потоків, файлів), що є домінуючим типом для програмних реалізацій, та скремблери для аналогових сигналів (наприклад, історично важливі голосові скремблери, що реалізовували інверсію спектру чи перестановку частотних смуг).
- За призначенням або сферою застосування: скремблери знаходять своє застосування як телекомунікаційні (для забезпечення якості сигналу та синхронізації), в системах умовного доступу (наприклад, у платному телебаченні), для обфускації або маскування даних (при тестуванні, розробці), а також як засоби загального призначення для простого захисту файлів від несанкціонованого перегляду.

Розуміння наведених класифікацій є важливим для правильного вибору або проектування скремблера, що відповідатиме поставленим вимогам щодо швидкодії, гнучкості, рівня захисту та специфіки застосування. Вибір між апаратною та програмною реалізацією часто залежить від балансу між цими факторами та наявними ресурсами.

1.2.3 Аналіз відомих алгоритмів, що можуть використовуватись для програмного скремблювання

Програмна реалізація скремблювання даних відкриває можливості для використання різноманітних алгоритмічних підходів, від найпростіших перетворень до більш комплексних послідовностей операцій. Вибір конкретного алгоритму залежить від поставлених завдань: чи це забезпечення технічних характеристик каналу зв'язку, маскування даних для тестування, чи базовий захист від випадкового перегляду. Нижче розглянуто основні класи алгоритмів, які можуть бути ефективно реалізовані програмно для цілей скремблювання.

1. Алгоритми на основі лінійних зсувних регістрів зі зворотним зв'язком є одним із найпоширеніших підходів для генерації псевдовипадкових бітових послідовностей, які потім зазвичай побітово додаються за модулем 2 (операція XOR) до вихідного потоку даних. Принцип роботи полягає в тому, що LFSR генерує послідовність бітів, де кожен наступний біт є лінійною функцією (зазвичай XOR) кількох попередніх бітів у регістрі, так званих «відводів», що визначаються генераторним поліномом. Програмна реалізація LFSR є відносно простою і може бути виконана за допомогою бітових операцій над цілочисельними змінними, що представляють стан регістру; важливими параметрами при цьому є довжина регістру, генераторний поліном та початковий стан («seed»). Хоча LFSR-скремблери часто використовуються в апаратних телекомунікаційних системах (наприклад, стандарти V.34, G.709, Ethernet 1000BASE-T) для забезпечення достатньої кількості переходів бітів та «відбілювання» спектру, їхні принципи легко відтворюються програмно для симуляції, тестування або специфічних завдань потокового скремблювання, де не вимагається висока криптографічна стійкість. Перевагами такого підходу є простота програмної реалізації, висока швидкість генерації ПВП та добрі статистичні властивості послідовності при правильному виборі поліному. Однак, якщо структура LFSR (поліном, довжина) відома, алгоритм є криптографічно слабким і легко аналізується.

2. Алгоритми, засновані на простих перестановках (Permutation Algorithms), змінюють порядок слідування елементів даних (бітів, байтів, слів або блоків фіксованого розміру) за певним правилом. Елементи даних переставляються відповідно до заданої таблиці перестановки або на основі процедурного правила. Програмна реалізація може включати обмін місцями елементів масиву, бітові маніпуляції для перестановки бітів у байті/слові, або індексацію для копіювання даних у новому порядку. Такі алгоритми використовуються для «перемішування» даних, ускладнення їх візуального аналізу або як один з етапів у більш складних перетвореннях, і можуть бути ефективні для дезорієнтації простих методів аналізу, якщо дані мають легко впізнавану структуру. Програмна реалізація зазвичай дуже проста та швидка. Однак, фіксовані перестановки легко піддаються аналізу, особливо якщо є можливість аналізувати достатню кількість скрембльованих даних або відомі властивості вихідних даних (наприклад, частотний аналіз для символьних даних). Стійкість дещо зростає, якщо правило перестановки залежить від якогось параметра (псевдо-ключа).

3. Алгоритми, засновані на простих замінах/підстановках (Substitution Algorithms), замінюють кожен елемент даних на інший елемент згідно з визначеним правилом. Кожен біт, байт або символ вхідного потоку замінюється іншим на основі фіксованої таблиці заміни (S-box, хоча використання криптографічно стійких S-box вже наближає метод до шифрування), простої арифметичної чи логічної операції. Програмна реалізація може включати інверсію бітів (XOR з маскою з одиниць), арифметичні операції (наприклад, додавання константи за модулем), або використання масивів як таблиць заміни. Ці алгоритми застосовуються для простої обфускації даних; наприклад, афінне перетворення може використовуватися для скремблювання символьних даних, за формулою 1.1.

$$C = (a \cdot P + b) \bmod m \quad (1.1)$$

де P – вихідний символ,

C – скрембльований символ,

a, b, m – параметри перетворення.

Простота та швидкість реалізації є перевагами. Однак, прості моноалфавітні заміни дуже вразливі до статистичного аналізу (наприклад, частотного аналізу символів). Складність аналізу дещо зростає для поліалфавітних замін або якщо таблиця заміни є великою та нерегулярною.

4. Комбіновані алгоритми передбачають програмну реалізацію поєднання декількох простих методів скремблювання для потенційного ускладнення аналізу. Принцип їхньої роботи полягає у послідовному застосуванні операцій перестановки, заміни, XOR-операцій з генерованими послідовностями тощо. Модульна структура програми дозволяє легко комбінувати різні функції перетворення. Хоча така комбінація може дещо підвищити складність «ручного» відновлення даних порівняно з одиночною операцією, без глибокого криптографічного аналізу система, швидше за все, не матиме значної стійкості проти цілеспрямованих атак. Основна перевага – ускладнення для випадкового спостерігача.

5. Спеціалізовані алгоритми для маскування даних призначені не стільки для унеможливлення розшифровки, скільки для заміни реальних чутливих даних на фіктивні, але правдоподібні, зі збереженням їх формату та деяких статистичних властивостей. Це досягається шляхом заміни значень у полях баз даних або файлах на основі правил, наприклад, заміни всіх прізвищ на випадкові з довідника, номерів телефонів – на генеровані за шаблоном, фінансових сум – на випадкові в певному діапазоні. Програмна реалізація таких алгоритмів широко використовується у вигляді спеціалізованих програмних інструментів або скриптів для підготовки даних для тестування, розробки або аналітики. Ці методи ефективні для своєї конкретної мети – зниження ризиків витоку реальних

даних при роботі з не-продуктивними середовищами, але не є скремблюванням у класичному розумінні для захисту каналів зв'язку.

При виборі або розробці програмного алгоритму скремблювання важливо чітко усвідомлювати мету скремблювання. Якщо потрібен захист від серйозних загроз, слід використовувати повноцінні криптографічні методи. Якщо ж завдання полягає в технічній обробці сигналу, маскуванні даних або простому ускладненні несанкціонованого перегляду, перераховані алгоритми можуть бути доцільними завдяки своїй простоті та швидкодії при програмній реалізації.

1.2.4 Огляд наявних програмних продуктів та бібліотек для скремблювання даних

Після аналізу теоретичних алгоритмів, що можуть бути покладені в основу програмного скремблювання, доцільно розглянути практичні інструменти – програмні продукти та бібліотеки, які реалізують функції скремблювання, обфускації або маскування даних. Важливо зазначити, що термін «скремблювання» не завжди прямо використовується розробниками; часто схожі функції описуються як «маскування даних» (data masking), «обфускація даних» (data obfuscation) або «псевдонімізація», особливо коли йдеться про захист даних у непродуктивних середовищах або підготовку даних для аналітики зі збереженням приватності.

Існує значна кількість спеціалізованих програмних продуктів, призначених для маскування та обфускації даних. Ця категорія переважно представлена комерційними рішеннями, орієнтованими на корпоративний сектор для захисту чутливої інформації в базах даних та файлових системах під час розробки, тестування, навчання або аналітики. Такі продукти часто пропонують широкий набір технік, включаючи заміну даних на реалістичні, але фіктивні значення, перемішування, шифрування окремих полів, а також методи, що відповідають принципам скремблювання, наприклад, застосування простих

детермінованих перетворень. Деякі відомі представники цієї категорії наведені в табл. 1.1.

Таблиця 1.1.

Приклади спеціалізованих програмних продуктів для маскування даних

Назва продукту/рішення	**Розробник**	**Основне призначення та типові функції**
Informatica Persistent Data Masking	Informatica	Маскування даних у БД для розробки/тестування; підстановка, перемішування, шифрування полів; збереження цілісності даних.
IBM Security Guardium Data Protection for Databases	IBM	Комплексне маскування даних у непродуктивних середовищах; різноманітні алгоритми обфускації; забезпечення відповідності нормативним вимогам.
Oracle Data Masking and Subsetting	Oracle	Створення безпечних копій БД шляхом маскування чутливих полів; бібліотека форматів маскування, можливість визначення власних.
Delphix Masking	Delphix	Автоматизація роботи з даними, маскування даних для швидкого створення безпечних тестових середовищ.
Broadcom (CA) Test Data Manager	Broadcom	Генерація та управління тестовими даними, включаючи функції маскування для захисту реальної продуктивної інформації.

Іншою важливою категорією є бібліотеки для мов програмування, які надають розробникам функції для реалізації власних алгоритмів скремблювання, обфускації або генерації псевдовипадкових послідовностей, що можуть слугувати основою для таких алгоритмів. Для мови програмування Python, наприклад, популярною є бібліотека Faker, що дозволяє генерувати різноманітні фіктивні дані (імена, адреси, тексти), які можуть використовуватися для маскування шляхом заміни. Стандартні модулі, такі як random (для генерації псевдовипадкових чисел і послідовностей, корисних у XOR-скремблюванні або для випадкових перестановок) та struct (для роботи з бінарними даними), також є корисними інструментами. Існують і спеціалізовані бібліотеки для обфускації коду, принципи яких подекуди перетинаються з ідеями скремблювання. Для мови Java, бібліотеки Apache Commons Lang/Codec містять утиліти для маніпуляцій з даними, включаючи генерацію випадкових рядків та операції кодування/декодування, а Java Faker є аналогом Python Faker. Вбудовані класи, такі як java.util.Random або java.security.SecureRandom, надають засоби для генерації псевдовипадкових чисел. У середовищі JavaScript/Node.js також поширені бібліотеки типу Faker.js (та його форки) для генерації фіктивних даних, а вбудований модуль crypto в Node.js надає інструменти для генерації псевдовипадкових байтів. Для мов C++ та C# стандартні бібліотеки пропонують широкий набір засобів для роботи з бітовими операціями та генерації випадкових чисел, що є основою для реалізації багатьох алгоритмів скремблювання.

Окремо можна виділити утиліти та інструменти командного рядка. Часто це простіші програми, нерідко з відкритим кодом, які можуть виконувати базові операції перетворення файлів, що іноді характеризують як скремблювання. Їхня стійкість зазвичай низька, але вони можуть бути корисними для швидкого «приховування» файлів від сторонніх очей. Прикладами є інструменти на основі простого XOR-перетворення, що виконують побайтове XOR-ування файлу з заданим ключем або послідовністю, або невеликі утиліти для перестановки байтів у файлі чи застосування простих табличних замін.

Нарешті, програмне забезпечення для моделювання телекомунікаційних систем є ще однією нішею, де зустрічаються програмні реалізації скремблерів. У таких продуктах можуть бути втілені стандартизовані алгоритми скремблювання (наприклад, на основі LFSR, що використовуються в G.709, SDH/SONET або Ethernet) для цілей моделювання, аналізу та тестування протоколів зв'язку. Прикладами таких платформ є GNU Radio, відкрита система для розробки програмно-визначуваних радіосистем (SDR), де користувачі можуть створювати та моделювати різні блоки обробки сигналів, включаючи скремблери, а також MATLAB/Simulink, що широко використовуються для моделювання систем зв'язку.

Варто підкреслити, що наведений огляд не є вичерпним, а лише ілюструє основні категорії та приклади програмних засобів, які можуть бути використані для реалізації функцій, що відповідають поняттю програмного скремблювання. При виборі конкретного інструменту чи бібліотеки для розробки необхідно ретельно аналізувати його функціональні можливості, рівень забезпечуваного «заплутування» даних, продуктивність, ліцензійні умови та відповідність поставленим завданням, особливо враховуючи, що для надійного захисту чутливої інформації слід надавати перевагу перевіреним криптографічним методам.

1.3 Проблемні аспекти та невирішені задачі в галузі програмного скремблювання

Незважаючи на свою простоту реалізації та певну корисність у специфічних нішах, таких як маскування даних або технічна обробка сигналів, програмне скремблювання не позбавлене низки проблемних аспектів та відкритих питань. Цей розділ має на меті проаналізувати основні обмеження та виклики, пов'язані з використанням програмних скремблерів, зокрема, їхню стійкість до сучасних методів аналізу, питання ефективності та стандартизації. Виявлення цих слабких місць та невирішених задач є важливим для об'єктивної

оцінки доцільності застосування програмного скремблювання та для визначення перспективних напрямків подальших досліджень у цій сфері.

1.3.1 Аналіз стійкості наявних програмних скремблерів

Оцінка стійкості програмних скремблерів є ключовим аспектом при визначенні їхньої придатності для практичного застосування. Під стійкістю скремблера розуміється його здатність протистояти спробам несанкціонованого відновлення вихідної інформації або визначення параметрів перетворення без легального доступу до них. Важливо одразу наголосити, що стійкість програмних скремблерів, особливо тих, що не є частиною складних криптографічних систем, у переважній більшості випадків значно поступається стійкості сучасних алгоритмів шифрування. Це зумовлено як самою природою та цілями скремблювання, так і частою простотою використовуваних алгоритмів.

Аналіз стійкості програмних скремблерів зазвичай виявляє їхню вразливість до низки відомих методів криптоаналізу та специфічних атак, адаптованих до особливостей виконуваних перетворень. Одним з основних методів є статистичний аналіз. Багато видів інформації, такі як тексти природною мовою або структуровані дані, мають характерні статистичні закономірності, включаючи нерівномірний розподіл частот появи символів чи байтів та наявність надлишковості. Прості програмні скремблери, що реалізують, наприклад, лише операції заміни окремих символів (моноалфавітна підстановка) або елементарні перестановки, можуть недостатньо ефективно «перемішувати» ці статистичні властивості. Як наслідок, застосування частотного аналізу до скрембльованих даних може дозволити відновити таблицю заміни або ідентифікувати закономірності перестановки. Аналогічно, аналіз надлишковості та кореляційний аналіз можуть виявити слабкі місця в алгоритмі.

Програмні скремблери також часто вразливі до атак на основі відомого або обраного відкритого тексту. Якщо зловмисник має доступ до пар «вихідний текст – відповідний йому скрембльований текст», він може використати цю інформацію для визначення невідомих параметрів скремблера або навіть для повного відновлення алгоритму перетворення. В умовах програмної реалізації, де користувач може контролювати вхідні дані, іноді можлива й атака на основі обраного відкритого тексту, що значно полегшує аналіз. Скремблери, що використовують фіксовані, незмінні алгоритми, є особливо чутливими до таких атак.

Якщо програмний скремблер використовує певні параметри, що впливають на перетворення (наприклад, початковий стан для генератора псевдовипадкової послідовності, короткий ключ для простої заміни чи перестановки), і простір можливих значень цих параметрів є відносно невеликим, то можлива атака повного перебору (Brute-force). Зловмисник послідовно перебирає всі можливі значення параметрів, застосовуючи дескремблювання до перехоплених даних, доки не отримає осмислений результат. Сучасні обчислювальні потужності роблять такий підхід реальним для слабких скремблерів.

Однією з суттєвих вразливостей саме програмних скремблерів є можливість реверс-інжинірингу їхнього програмного коду. За допомогою декомпіляторів, дизасемблерів та відлагоджувачів зловмисник може дослідити логіку роботи програми, виявити використовувані алгоритми перетворення, таблиці заміни, константи або навіть вбудовані «ключі». Хоча методи обфускації коду програми-скремблера можуть ускладнити цей процес, вони рідко забезпечують надійний захист від кваліфікованого аналітика.

Існують також атаки, специфічні для певних класів алгоритмів. Наприклад, для скремблерів, що використовують лінійні зсувні регістри зі зворотним зв'язком, відомі ефективні математичні методи аналізу, такі як алгоритм Берлекемпа-Мессі, що дозволяє відновити поліном LFSR за наявності частини псевдовипадкової послідовності. Скремблери, що базуються на простих

перестановках бітів або байтів, можуть бути вразливі до диференціального аналізу або шляхом аналізу властивостей структурованих даних, наприклад, заголовків файлів відомих форматів.

Ключовими факторами, що зумовлюють низьку стійкість більшості програмних скремблерів, є використання алгоритмічно простих перетворень, відсутність або слабкість ключової системи (часто алгоритм є фіксованим і не залежить від секретного ключа), а також ігнорування фундаментальних криптографічних принципів, як-от принцип Керкгоффса. Багато скремблерів розробляються не з метою забезпечення криптографічної стійкості, а для вирішення інших завдань, тому їхня стійкість до цілеспрямованого аналізу не є пріоритетом.

1.3.2 Питання ефективності програмних скремблерів

Окрім стійкості, важливим аспектом оцінки програмних скремблерів є їхня ефективність, що охоплює як здатність досягати цілей обфускації даних, так і показники продуктивності – швидкодію та ресурсоємність. Ці параметри безпосередньо впливають на практичну придатність скремблера в реальних системах.

Швидкодія програмних скремблерів значною мірою визначається складністю обраного алгоритму та якістю його реалізації. Прості алгоритми, що базуються на побітових операціях (наприклад, XOR з псевдовипадковою послідовністю), елементарних арифметичних перетвореннях або нескладних перестановках/замінах, зазвичай демонструють високу швидкість обробки, оскільки такі операції є ефективними для сучасних процесорів. Загалом, програмні скремблери працюють швидше, ніж складні криптографічні алгоритми, що є їхньою перевагою при обробці великих обсягів даних, де високий рівень криптографічного захисту не є першочерговим. Однак, швидкодія може суттєво знижуватися при використанні складних комбінацій операцій або багаторазових проходів по даних.

Ресурсоємність програмних скремблерів, тобто обсяг використовуваних ресурсів центрального процесора та оперативної пам'яті, також є важливим показником. Більшість простих алгоритмів скремблювання є легковагими. Наприклад, потокові скремблери потребують мінімального обсягу пам'яті для зберігання свого стану і незначно навантажують процесор, що робить їх придатними для пристроїв з обмеженими ресурсами. Більш ресурсоємними можуть бути алгоритми, що вимагають обробки великих блоків даних або використання значних таблиць перетворень.

Здатність програмних скремблерів досягати поставлених цілей також потребує оцінки. Для простого ускладнення візуального аналізу або захисту від випадкового перегляду навіть елементарні методи можуть бути цілком ефективними. У задачах маскування даних ефективність визначається здатністю генерувати правдоподібні, але фіктивні дані. Якщо ж скремблер використовується для емуляції технічних функцій (наприклад, «відбілювання» спектру сигналу), його ефективність оцінюється за точністю відтворення бажаних характеристик.

Однак, існують і проблемні аспекти. Прагнення покращити «заплутуючі» властивості шляхом надмірного ускладнення алгоритму без належного теоретичного обґрунтування може призвести до значного падіння продуктивності без суттєвого приросту реальної стійкості. Неефективна програмна реалізація також може нівелювати переваги простого алгоритму.

Таким чином, питання ефективності програмних скремблерів є багатогранним. Вони часто виграють у швидкодії та ресурсоємності порівняно з криптографічними засобами, що робить їх привабливими для певних ніш. Проте, ця ефективність повинна розглядатися в контексті поставлених завдань та необхідного рівня «захисту» чи перетворення даних.

Висновки до розділу 1

У першому розділі даної кваліфікаційної роботи було проведено комплексний теоретичний аналіз проблеми захисту інформації та визначено специфіку використання програмного скремблювання як одного з її інструментів. Було підтверджено надзвичайну актуальність захисту інформації в сучасних умовах, особливо зважаючи на стрімке зростання обсягів даних та ускладнення кіберзагроз. В рамках розділу було систематизовано ключову термінологію в галузі інформаційної безпеки та скремблювання, розглянуто основні класифікації загроз інформаційній безпеці, а також окреслено місце скремблювання серед інших, більш потужних методів захисту, таких як криптографічне шифрування.

Детальний огляд наявних методів та засобів скремблювання даних виявив, що в основі цих процесів лежать принципи перестановки, заміни та використання псевдовипадкових послідовностей. Особливу увагу було приділено аналізу алгоритмів, що потенційно можуть використовуватися для програмного скремблювання, таких як методи на основі LFSR, прості перестановки та заміни, а також комбіновані підходи.

Водночас, проведений аналіз проблемних аспектів програмного скремблювання висвітлив низку суттєвих обмежень. Зокрема, було встановлено, що стійкість більшості відомих програмних скремблерів до цілеспрямованого криптоаналізу є невисокою, і вони не можуть слугувати надійною заміною криптографічним методам для захисту конфіденційної інформації. Питання ефективності програмних скремблерів, хоча й часто демонструють переваги у швидкодії та низькій ресурсоємності порівняно з шифруванням, також потребують ретельного розгляду в контексті конкретних завдань та очікуваного рівня «заплутування» даних. Залишаються актуальними питання стандартизації програмних скремблерів для нетелекомунікаційних застосувань та їх чіткого функціонального позиціонування.

РОЗДІЛ 2
РОЗРОБКА ТА ТЕОРЕТИЧНЕ ОБҐРУНТУВАННЯ ПРОГРАМНОГО СКРЕМБЛЕРУ ДЛЯ ЗАХИСТУ ІНФОРМАЦІЇ

2.1 Обґрунтування вибору напрямку дослідження та архітектури програмного скремблеру

Після детального аналізу теоретичних основ захисту інформації, огляду наявних методів скремблювання та виявлення проблемних аспектів у першому розділі, даний розділ присвячений обґрунтуванню вибору напрямку власного дослідження та розробки. Зокрема, тут буде обґрунтовано вибір конкретних алгоритмів скремблювання, розглянуто їхню теоретичну базу, а також запропоновано концептуальну архітектуру програмного засобу, що реалізує ці алгоритми. Метою є створення гнучкого програмного інструменту, що дозволяє виконувати операції скремблювання даних для ускладнення їх несанкціонованого аналізу або для інших прикладних завдань, де не висуваються вимоги до високої криптографічної стійкості, але важлива простота, швидкість та можливість комбінування перетворень.

2.1.1 Аналіз вимог до розроблюваного програмного скремблеру

На основі аналізу літературних джерел та сучасного стану проблеми, висвітленого в першому розділі, а також враховуючи специфіку програмного скремблювання як методу обфускації даних, було сформульовано ключові вимоги до розроблюваного програмного засобу. Передусім, передбачалася підтримка декількох алгоритмів скремблювання, що базуються на різних принципах (заміна, перестановка, побітові операції), для забезпечення гнучкості обробки даних. Важливою є можливість комбінованого (каскадного) застосування обраних алгоритмів з визначенням користувачем порядку їх

послідовного виконання, що потенційно ускладнює аналіз скрембльованих даних.

Обов'язковою вимогою є використання ключа, наданого користувачем, для керування процесом перетворення в усіх алгоритмах, що вносить варіативність та забезпечує різний результат для однакового відкритого тексту при різних ключах. Також система повинна надавати можливість генерації ключів належної якості. Фундаментальною є вимога оборотності всіх перетворень: кожен алгоритм та їх комбінація повинні мати відповідний процес дескремблювання для повного відновлення вихідних даних за тим самим ключем.

Програмний засіб має бути здатним обробляти як текстові дані, так і файли різних форматів. Доступність забезпечується реалізацією у вигляді веб-додатку, що дозволяє використання з різних пристроїв. Архітектура програмного засобу повинна бути модульною та розширюваною для легкого додавання нових алгоритмів скремблювання. Швидкодія програмного скремблера має бути прийнятною для інтерактивної роботи з даними помірного розміру. При цьому, вимога забезпечення високої криптографічної стійкості свідомо не ставилася як першочергова, оскільки фокус дослідження спрямований на методи обфускації.

2.1.2 Вибір математичного апарату та алгоритмічної основи для програмного скремблера

Вибір алгоритмічної основи для розроблюваного програмного скремблера здійснювався з урахуванням сформульованих у попередньому підрозділі вимог. Пріоритет надавався простоті реалізації, гнучкості, швидкодії та можливості комбінування перетворень, при цьому не ставилася мета досягнення високого рівня криптографічної стійкості. Зважаючи на ці міркування, було обрано набір класичних, добре вивчених та відносно простих у програмній реалізації алгоритмів, що базуються на різних фундаментальних принципах зміни даних.

Фундаментом математичного апарату обраних алгоритмів слугують побітові логічні операції, зокрема операція «виключне АБО» (XOR). Ця операція

широко використовується завдяки своїй інволютивності, тобто A⊕B⊕B=A, та високій швидкості виконання на сучасному обладнанні, що дозволяє ефективно «накладати» ключову послідовність на потік даних. Іншим важливим елементом математичного апарату є арифметичні операції за модулем N, де N відповідає розміру алфавіту оброблюваних символів, наприклад, 256 для байтових даних. Такий підхід є основою для адитивних скремблерів, аналогічних шифру Цезаря, і забезпечує зсув значень даних залежно від ключа. Також використовуються операції перестановки, які математично описуються функцією, що змінює порядок слідування елементів даних (байтів) у межах визначених блоків, ставлячи кожному індексу вихідного блоку у відповідність індекс у переставленому блоці. Генерація таких перестановок може базуватися на алгоритмах, що використовують генератори псевдовипадкових чисел, ініціалізовані ключем.

На основі цього математичного апарату було сформовано алгоритмічну базу, що включає три основні типи перетворень. Перший – це скремблювання на основі операції XOR, яке забезпечує потокове перетворення, де кожен байт даних комбінується з байтом ключа; цей метод обрано через його виняткову простоту та швидкість. Другий тип – скремблювання методом блокової перестановки, що руйнує локальні залежності в даних шляхом зміни порядку байтів у межах блоків; цей підхід обрано для демонстрації принципу «заплутування» через зміну структури, а не значень. Третім типом є скремблювання на основі адитивного перетворення, аналогічного модифікованому шифру Цезаря, що є простим методом заміни, який змінює значення байтів шляхом їх зсуву; його обрано як ще один приклад елементарного перетворення з відмінною математичною основою.

Ці алгоритми, хоч і є відносно простими з криптографічної точки зору, добре ілюструють різні принципи скремблювання та дозволяють реалізувати вимогу щодо комбінованого застосування для потенційного посилення ефекту обфускації. Їх вибір також зумовлений освітньо-дослідницьким характером

роботи, де важливо продемонструвати реалізацію та аналіз зрозумілих та контрольованих методів перетворення даних.

2.2. Теоретичні основи роботи запропонованого програмного скремблеру

В рамках даної роботи розглядається програмний скремблер, що реалізує набір обраних алгоритмів перетворення даних. Теоретичною базою для його функціонування є декілька фундаментальних підходів до зміни інформації, які можуть застосовуватися як окремо, так і послідовно для досягнення бажаного ефекту обфускації. Нижче наведено теоретичний аналіз кожного з цих підходів.

2.2.1 Скремблювання на основі побітової операції XOR

Одним з найпоширеніших та найпростіших у реалізації методів потокового перетворення даних є скремблювання за допомогою побітової операції «виключне АБО» (XOR). Цей метод належить до симетричних алгоритмів, оскільки один і той самий ключ та одна й та сама операція використовуються як для скремблювання, так і для дескремблювання.

Теоретично, кожен елемент (зазвичай байт або біт) вхідної послідовності даних $P=(p0,p1,...,pn-1)$ піддається операції XOR з відповідним елементом ключової послідовності $K=(k0,k1,...,km-1)$. Якщо довжина ключової послідовності m менша за довжину послідовності даних n, ключ зазвичай застосовується циклічно. Тобто, для і-го елемента даних pi використовується j-й елемент ключа, де $j=i(modm)$. Результатом є скрембльована послідовність $C=(c0,c1,...,cn-1)$, де кожен елемент ci обчислюється за формулою 2.1.

$$C_i = P_i \oplus K_j \tag{2.1}$$

де $\oplus$ позначає операцію «виключне АБО».

Ключовою властивістю операції XOR, що робить її зручною для таких перетворень, є її інволютивність, тобто самооберненість. Це означає, що повторне застосування операції XOR з тим самим ключем до скрембльованого елемента відновлює вихідний елемент за формулою 2.2.

$$\left(P_i \oplus K_j\right) \oplus K_j = P_i \oplus \left(K_j \oplus K_j\right) = P_i \oplus 0 = P_i \quad (2.2)$$

Таким чином, процедура дескремблювання повністю ідентична процедурі скремблювання, що значно спрощує реалізацію.

Ключ у даному алгоритмі відіграє вирішальну роль. Його довжина та ступінь випадковості безпосередньо впливають на теоретичну стійкість перетворення. Якщо ключ короткий та/або має передбачувану структуру (наприклад, складається з повторюваних патернів або є осмисленим текстом), алгоритм стає вразливим до різноманітних атак, зокрема до атаки на основі відомого відкритого тексту. Якщо зловмиснику відома хоча б частина відкритого тексту та відповідний йому шифротекст, він може легко відновити частину ключа *(Kj=Pi⊕Ci)*. В ідеальному випадку, для максимальної стійкості (як у шифрі Вернама, або «one-time pad»), ключ має бути абсолютно випадковим, дорівнювати за довжиною повідомленню та використовуватися лише один раз. Однак у практичних реалізаціях скремблерів ці умови часто не виконуються, і ключ є значно коротшим за дані та використовується багаторазово.

До беззаперечних переваг XOR-скремблювання належать його концептуальна простота та висока обчислювальна ефективність, оскільки операція XOR є однією з найшвидших для сучасних процесорів. Це дозволяє обробляти великі обсяги даних з мінімальними затримками. Однак, з точки зору стійкості, простий XOR-скремблер із повторюваним ключем є криптографічно слабким. Він не змінює статистичних характеристик відкритого тексту (якщо розглядати різницю між шифротекстом та відкритим текстом), а лише «маскує» його за допомогою ключа. Якщо один і той самий ключ використовується для скремблювання кількох повідомлень, і зловмисник отримує доступ до всіх

відповідних скрембльованих текстів, він може застосувати методи, засновані на аналізі накладання шифротекстів (наприклад, якщо $C1=P1 \oplus K$ та $C2=P2 \oplus K$, то $C1 \oplus C2=P1 \oplus P2$), що може призвести до розкриття інформації про відкриті тексти, особливо якщо вони мають відому структуру або надлишковість.

Незважаючи на низьку криптографічну стійкість при типовому використанні, XOR-скремблювання може бути корисним для дуже швидкого перетворення даних з метою ускладнення їх випадкового перегляду або як проміжний етап у більш складних комбінованих алгоритмах обфускації.

2.2.2 Скремблювання методом блокової перестановки байтів

Метод скремблювання шляхом блокової перестановки байтів належить до класу перестановочних перетворень, основна ідея яких полягає у зміні порядку слідування елементів даних, а не їхніх значень. Такий підхід спрямований на руйнування вихідної структури та статистичних закономірностей інформації шляхом «перемішування» її компонентів.

Теоретично, процес блокової перестановки передбачає поділ вхідного масиву байтів P на послідовність блоків $P0,P1,...,Pk-1$. Розмір кожного блоку N може бути фіксованим або адаптивним (наприклад, останній блок може бути коротшим, якщо загальна довжина даних не кратна розміру блоку). У рамках даного дослідження, розмір блоку обирається як фіксоване значення (наприклад, 64 байти) або дорівнює довжині вхідних даних, якщо вона менша за це фіксоване значення. Для кожного такого блоку $Pb=(pb,0,pb,1,...,pb,N-1)$ застосовується індивідуальна операція перестановки.

Перестановка всередині блоку визначається ключем перестановки πb, який є послідовністю індексів $(\pi b,0,\pi b,1,...,\pi b,N-1)$, де кожен індекс від 0 до $N-1$ зустрічається рівно один раз. Скрембльований блок $Cb=(cb,0,cb,1,...,cb,N-1)$ формується таким чином, що $cb,i=pb,\pi b,i$. Важливим аспектом є генерація самого ключа перестановки πb. Цей ключ має бути детерміновано згенерований на основі основного користувацького ключа K, щоб забезпечити можливість

відтворення тієї ж самої перестановки при дескремблюванні. Часто для цього використовують генератор псевдовипадкових чисел, який ініціалізується («seed») значенням, похідним від ключа K (наприклад, сумою його байтів). Потім, використовуючи ГПВЧ, генерується послідовність випадкових чисел, на основі якої формується перестановка, наприклад, за допомогою алгоритму тасування Фішера-Єйтса.

Оборотність даного перетворення є фундаментальною вимогою. Для відновлення вихідного порядку байтів у блоці при дескремблюванні необхідно застосувати обернену перестановку $\pi b{-}1$. Якщо скрембльований байт cb,i отриманий з вихідного байта pb,j за правилом $j{=}\pi b,i$ (тобто $cb,i{=}pb,\pi b,i$), то обернена перестановка $\pi b{-}1$ така, що pb,j знаходиться на позиції $\pi b,j{-}1$ у вихідному блоці, або, іншими словами, $pb,k{=}cb,\pi b,k{-}1$. Обернений ключ перестановки також повинен детерміновано генеруватися на основі того ж користувацького ключа K.

Вплив параметрів, таких як розмір блоку та характеристики ключа, на властивості скремблювання є значним. Менший розмір блоку обмежує ступінь «перемішування» даних у глобальному масштабі, оскільки перестановки відбуваються лише локально. Більший розмір блоку потенційно може забезпечити кращу обфускацію, але збільшує обчислювальну складність та вимоги до пам'яті. Якість генерації перестановки, що безпосередньо залежить від якості ГПВЧ та способу використання ключа K для його ініціалізації, визначає непередбачуваність та рівномірність перестановок. Якщо простір можливих перестановок малий або легко передбачуваний, стійкість алгоритму знижується.

До переваг методу блокової перестановки належить його здатність руйнувати локальні кореляції та патерни у вихідних даних, не змінюючи при цьому загального статистичного розподілу значень байтів (оскільки відбувається лише зміна їхніх позицій). Це може бути ефективним проти простих візуальних або статистичних методів аналізу, орієнтованих на локальні властивості. Обчислювальна складність залежить від розміру блоку та методу генерації перестановки, але для помірних розмірів блоків може бути прийнятною.

Недоліки пов'язані з тим, що сам по собі перестановочний шифр (а скремблювання перестановкою є його аналогом) є класично слабким проти атак на основі відомого відкритого тексту або якщо зловмиснику відомі певні структурні особливості даних (наприклад, формат файлу). Якщо ключ, що визначає перестановку, слабкий або може бути підібраний, то алгоритм легко компрометується. У випадку, коли перестановка однакова для всіх блоків (або залежить від ключа однаковим чином), це також спрощує аналіз. Використання ГПВЧ, ініціалізованого простою функцією від ключа (наприклад, сумою байтів), може призвести до колізій (різні ключі *K* дають однаковий «seed») або до малого простору ефективних ключів перестановки.

Таким чином, скремблювання методом блокової перестановки байтів є технікою, що забезпечує обфускацію даних шляхом зміни їхньої внутрішньої структури на рівні блоків. Його ефективність та стійкість залежать від обраних параметрів (розмір блоку, метод генерації перестановки) і, як правило, недостатні для захисту від цілеспрямованих криптоаналітичних атак, але можуть бути прийнятними для ускладнення поверхневого аналізу.

2.2.3 Скремблювання на основі адитивного перетворення.

Адитивне скремблювання, реалізоване в даній роботі як модифікація класичного шифру Цезаря для байтових даних, є методом заміни, при якому кожен байт вихідної інформації зсувається на певну величину. Ця величина зсуву визначається відповідним байтом ключа, що забезпечує залежність перетворення від секретного параметра. Такий підхід є одним з найпростіших способів зміни значень даних.

Теоретично, для кожного байта вхідної послідовності даних $P=(p0,p1,...,pn-1)$ та відповідного байта ключової послідовності $K=(k0,k1,...,km-1)$, де ключ застосовується циклічно $(j=i(modm), якщо\ m<n)$, скрембльований байт Ci формується шляхом додавання значення байта ключа до значення байта даних за

модулем розмірності алфавіту (для байтів це 256). Математично це можна представити за формулою 2.3.

$$C_i = \left(P_i + K_j\right) \bmod 256 \tag{2.3}$$

Це перетворення змінює числове значення кожного байта, не впливаючи на їхній порядок у послідовності.

Процес дескремблювання є оберненим до скремблювання і полягає у відніманні того ж значення байта ключа від скрембльованого байта, також за модулем 256. Для відновлення вихідного байта *Pi* з скрембльованого байта *Ci* використовується формула 2.4.

$$P_i = \left(C_i - K_j + 256\right) \bmod 256 \tag{2.4}$$

Додавання 256 у формулі дескремблювання забезпечує коректний результат для операції за модулем у випадку, коли *Ci*<*Kj*, запобігаючи отриманню від'ємних значень перед взяттям залишком від ділення. Оборотність гарантується тим, що операція віднімання є оберненою до операції додавання.

Роль ключа в цьому алгоритмі, як і в XOR-скремблюванні, є центральною. Кожен байт ключа *Kj* безпосередньо визначає величину зсуву для відповідного байта даних *Pi*. Якщо ключ короткий, він буде багаторазово повторюватися, що створює періодичність у перетворенні. Якщо ключ складається з однакових байтів або байтів з низькою варіативністю, то і зсуви будуть одноманітними. Стійкість такого перетворення прямо залежить від довжини та непередбачуваності ключа.

З теоретичної точки зору, основною перевагою даного методу є його надзвичайна простота та висока швидкість програмної реалізації, оскільки він вимагає лише однієї арифметичної операції на кожен байт даних. Однак, його криптографічна стійкість є дуже низькою. Якщо ключ короткий або відомий, або якщо відома частина відкритого тексту, алгоритм легко розкривається. Навіть

без знання ключа, скрембльовані таким чином дані, особливо якщо вони представляють текст або мають іншу відому статистичну структуру, вразливі до частотного аналізу. Розподіл частот байтів у скрембльованому тексті буде просто зсунутою версією розподілу частот у відкритому тексті, що дозволяє відносно легко відновити ключ або сам текст.

Незважаючи на ці суттєві недоліки з точки зору безпеки, адитивне скремблювання може застосовуватися для дуже базового рівня обфускації, коли основною вимогою є швидкість і простота, а не надійний захист від аналізу. Як і XOR-скремблювання, воно може слугувати одним з елементарних перетворень у складі більш складних комбінованих (каскадних) скремблерів, де його слабкості можуть частково компенсуватися іншими операціями. Проте, для захисту чутливої інформації від цілеспрямованих атак цей метод самостійно не рекомендується.

2.2.4 Принципи комбінованого (каскадного) застосування алгоритмів скремблювання

Прагнення посилити ефект обфускації даних та ускладнити їх аналіз природно приводить до ідеї послідовного застосування декількох різних методів скремблювання. Такий підхід, відомий як комбіноване або каскадне скремблювання, передбачає, що вихід одного алгоритму перетворення стає вхідними даними для наступного. Теоретичне обґрунтування цього принципу полягає в тому, що кожен наступний алгоритм вносить додаткові зміни в структуру та зміст даних, потенційно посилюючи загальний ефект «заплутування».

Основна ідея каскадного скремблювання полягає в тому, щоб властивості одного типу перетворення компенсували або ускладнювали аналіз слабкостей іншого. Наприклад, якщо перший алгоритм змінює значення байтів (як адитивне скремблювання або XOR), а другий змінює їхній порядок (як блокова перестановка), то очікується, що комбінований ефект буде сильнішим, ніж від

кожного з них окремо. Наприклад, метод, що просто змінює значення байтів, може бути вразливим до частотного аналізу, якщо позиції символів залишаються незмінними. Подальша перестановка цих змінених байтів може зруйнувати прості статистичні залежності, що полегшували б такий аналіз.

Нехай є послідовність *N* алгоритмів скремблювання *S*1, *S*2, ..., *S*N, кожен з яких може використовувати один і той самий ключ *K* або різні ключі *K*1, *K*2, ..., *KN*. У випадку даної роботи передбачається використання одного спільного ключа *K* для всіх алгоритмів у каскаді. Тоді процес скремблювання вихідних даних *P* можна представити як послідовне застосування функцій за формулою 2.5.

$$C = S_N\left(K, S_{N-1}\left(K, \dots S_2\left(K, S_1(K, P)\right) \dots\right)\right) \tag{2.5}$$

Результатом *C* є фінально скрембльовані дані. Кожен алгоритм *Si* вносить свій вклад у загальне перетворення. У програмній реалізації, що розглядається, користувач має можливість самостійно обирати набір та послідовність застосовуваних алгоритмів з числа доступних (XOR, блокова перестановка, адитивне скремблювання).

Оборотність каскадного скремблювання забезпечується оборотністю кожного окремого алгоритму *S*i в послідовності, тобто для кожного *S*i існує обернене перетворення S_1^{-1}. Для дескремблювання необхідно застосувати ці обернені перетворення у строго зворотному порядку відносно порядку їх застосування при скремблюванні:

$$P = S_1^{-1}\left(K, S_2^{-1}\left(K, \dots S_{N-1}^{-1}\left(K, S_N^{-1}(K, C)\right) \dots\right)\right) \tag{2.6}$$

Цей принцип інверсії порядку є критично важливим для коректного відновлення вихідних даних.

З теоретичної точки зору, комбінування різних простих перетворень може збільшити роботу, необхідну для аналізу скрембльованих даних, порівняно з одиночним простим перетворенням. Якщо один алгоритм добре приховує одні статистичні властивості, а інший – інші, їхня комбінація може створити більш рівномірний або менш передбачуваний розподіл скрембльованих значень. Наприклад, застосування XOR-скремблювання після блокової перестановки може змінити значення байтів, які вже були перемішані, додатково ускладнюючи відновлення вихідної послідовності.

Однак важливо розуміти, що сукупна стійкість каскаду простих скремблюючих алгоритмів не обов'язково дорівнює добутку їхніх індивідуальних стійкостей і, як правило, не досягає рівня стійкості, що забезпечується сучасними криптографічними шифрами. Слабкості окремих компонентів або їх невдала комбінація можуть призвести до того, що вся система залишиться вразливою. Наприклад, якщо один з етапів легко аналізується або обходиться, це може значно спростити аналіз усього каскаду. Без глибокого математичного аналізу властивостей конкретної комбінації алгоритмів (подібно до того, як аналізуються раунди в блокових шифрах), оцінка реального приросту «заплутуючих» властивостей залишається переважно евристичною.

Тим не менш, для цілей базової обфускації даних, де не стоїть завдання протистояти потужним цілеспрямованим атакам, комбіноване скремблювання може запропонувати прийнятний компроміс між простотою реалізації, швидкістю та ступенем ускладнення даних для неавторизованого або випадкового доступу. Можливість гнучкого вибору та впорядкування алгоритмів дозволяє користувачеві експериментувати з різними конфігураціями для досягнення бажаного ефекту.

2.2.5 Теоретичні аспекти генерації та використання ключів у запропонованих методах скремблювання

Ключ є фундаментальним елементом у більшості алгоритмів скремблювання, оскільки саме він вносить елемент невизначеності для стороннього спостерігача та керує процесом перетворення даних. Теоретичні аспекти, пов'язані з генерацією, представленням та використанням ключів, безпосередньо впливають на властивості та потенційну стійкість (навіть якщо вона обмежена) скремблюючих перетворень.

В ідеальному випадку, для досягнення максимального ефекту «заплутування» та ускладнення аналізу, ключ, що використовується в алгоритмах скремблювання, повинен мати властивості, близькі до властивостей криптографічних ключів. До них належать достатня довжина, що ускладнює атаки повного перебору, та високий ступінь випадковості (або псевдовипадковості) його компонентів, щоб уникнути легко вгадуваних патернів. Хоча для скремблерів, де криптографічна стійкість не є першочерговою вимогою, ці умови можуть бути менш жорсткими, використання слабких, коротких або передбачуваних ключів нівелює навіть обмежені захисні властивості скремблювання. Також передбачається, що ключ, який керує конкретним перетворенням, невідомий потенційному зловмиснику.

У програмних системах користувач зазвичай надає ключ у вигляді текстового рядка. Оскільки розглянуті алгоритми скремблювання (XOR, адитивне перетворення, блокова перестановка байтів) оперують на байтовому рівні, необхідний етап перетворення рядкового представлення ключа у послідовність байтів. Стандартним підходом для цього є застосування універсальних кодувань символів, таких як UTF-8. Таке перетворення може вплинути на ефективну байтову довжину ключа та розподіл значень його байтів, залежно від символів, використаних у вхідному рядку ключа.

Для уникнення використання користувачами слабких ключів, доцільно передбачити механізм їх автоматичної генерації. Теоретично, такий генератор

повинен спиратися на криптографічно стійкий генератор псевдовипадкових чисел (КГПВЧ), щоб гарантувати непередбачуваність та високу ентропію генерованих ключів. У програмній системі, що є основою для даної роботи, передбачена генерація послідовності випадкових байтів заданої довжини з використанням системних криптографічних засобів. Отримані байти потім можуть бути представлені користувачеві у зручному форматі, наприклад, у вигляді шістнадцяткового рядка, який при подальшому використанні знову перетворюється на байтову послідовність.

Спосіб використання ключа залежить від конкретного алгоритму скремблювання. В XOR-скремблюванні та адитивному скремблюванні байти ключа послідовно (і циклічно, якщо ключ коротший за дані) застосовуються до кожного байта даних для виконання відповідної операції (XOR або додавання за модулем). У випадку блокового перестановочного скремблера, ключ (або певне значення, похідне від нього, наприклад, сума його байтів) слугує початковим значенням («seed») для генератора псевдовипадкових чисел, який, у свою чергу, формує унікальну таблицю перестановки для кожного блоку даних.

Незважаючи на наявність ключа, важливо пам'ятати, що якщо сам алгоритм скремблювання є простим та відомим, то навіть формально довгий та випадковий ключ не гарантує високої стійкості. Наприклад, у простому XOR-скремблюванні з повторюваним ключем, наявність достатнього обсягу скрембльованих даних та знання певних властивостей відкритого тексту можуть дозволити відновити ключ. Аналогічно, для перестановочного скремблера, якщо метод генерації перестановки на основі ключа має слабкості (наприклад, якщо «seed» для ГПВЧ має малий діапазон можливих значень або легко виводиться з ключа), це обмежує ефективний простір ключів і спрощує аналіз.

Таким чином, хоча використання ключів у програмних скремблерах є обов'язковим для введення варіативності та керованості процесом перетворення, їхня якість (довжина, випадковість) та спосіб інтеграції в алгоритм є критичними факторами, що визначають ступінь досяжної обфускації. Для забезпечення надійного захисту інформації в умовах сучасних кіберзагроз, необхідно

покладатися на перевірені криптографічні системи з відповідними протоколами управління ключами, тоді як скремблювання з ключем може розглядатися лише як допоміжний засіб.

Висновки до розділу 2

У другому розділі кваліфікаційної роботи було представлено теоретичне обґрунтування та основні етапи розробки програмного скремблера для захисту інформації.

На основі аналізу вимог до розроблюваного засобу, який включав підтримку декількох алгоритмів, можливість їх каскадного застосування, використання ключів, оборотність перетворень та обробку різних типів даних, було обґрунтовано вибір напрямку дослідження. Пріоритет було надано реалізації набору класичних та відносно простих алгоритмів, що базуються на фундаментальних принципах зміни даних: побітових логічних операціях (XOR), арифметичних операціях за модулем (адитивне перетворення, аналогічне шифру Цезаря) та операціях перестановки елементів даних (блокова перестановка байтів). Такий вибір дозволив зосередитися на гнучкості, швидкодії та можливості комбінування перетворень, не ставлячи за першочергову мету досягнення високого рівня криптографічної стійкості, що відповідає освітньо-дослідницькому характеру роботи.

Детально проаналізовано теоретичні основи роботи кожного з обраних алгоритмів скремблювання. Для скремблювання на основі операції XOR розглянуто його принцип дії, інволютивність та вирішальну роль ключа у забезпеченні перетворення. Для методу блокової перестановки байтів описано процес поділу на блоки, генерації ключа перестановки на основі користувацького ключа та псевдовипадкового процесу, а також механізм оберненої перестановки для дескремблювання. Для скремблювання на основі адитивного перетворення проаналізовано математичну модель зсуву байтів за модулем та її оборотність. Також було розглянуто принципи комбінованого

(каскадного) застосування цих алгоритмів, що дозволяє потенційно ускладнити аналіз скрембльованих даних шляхом послідовного виконання різних типів перетворень. Окрему увагу приділено теоретичним аспектам генерації та використання ключів, наголошуючи на важливості їхньої довжини, випадковості та коректного застосування в кожному алгоритмі.

Запропоновано концептуальну архітектуру програмного засобу (описана детальніше в підрозділі 3.1.2), що передбачає модульний підхід та можливість розширення функціоналу.

Таким чином, у другому розділі закладено теоретичний фундамент для практичної реалізації програмного скремблера, обґрунтовано вибір алгоритмічної бази та визначено основні принципи його функціонування. Це створює передумови для подальшої розробки, тестування та аналізу властивостей програмного засобу, що буде представлено у наступному розділі.

РОЗДІЛ 3
ПРОГРАМНА РЕАЛІЗАЦІЯ ТА ЕКСПЕРИМЕНТАЛЬНЕ ДОСЛІДЖЕННЯ СКРЕМБЛЕРА ДАНИХ

3.1 Розробка програмного засобу для скремблювання даних

Цей розділ присвячений практичній реалізації програмного скремблера, теоретичні основи якого були обґрунтовані в попередніх розділах. Метою практичної частини є створення функціонального програмного продукту, що втілює обрані алгоритми скремблювання (XOR, блокова перестановка, адитивне перетворення) та їх комбіноване застосування. Розробка охоплює вибір інструментальних засобів, проектування архітектури програмного комплексу, реалізацію основних функціональних модулів та створення інтерфейсу користувача. В подальшому буде проведено експериментальне дослідження розробленого засобу для оцінки його функціональності та характеристик швидкодії.

3.1.1 Вибір інструментальних засобів та середовища розробки.

Для реалізації програмного скремблера було обрано сучасні та поширені технології, що забезпечують необхідну гнучкість, продуктивність та можливості для створення веб-додатку.

Серверна частина (back-end) розроблена з використанням мови програмування C# та платформи .NET, зокрема актуальної на момент розробки версії .NET 8. Вибір C# та .NET 8 зумовлений їхньою високою продуктивністю, надійністю, широким набором стандартних бібліотек для роботи з даними та криптографічними операціями (зокрема, System.Security.Cryptography. RandomNumberGenerator для генерації ключів), а також потужними засобами для створення веб-API. Платформа ASP.NET Core, що є частиною .NET, надає

ефективну інфраструктуру для побудови веб-сервісів, включаючи маршрутизацію, обробку HTTP-запитів та впровадження залежностей, що активно використовується в проекті.

Клієнтська частина (front-end) реалізована з використанням фреймворку Vue.js (версія 3.5.13) та системи збірки Vite. Vue.js обрано через його легкість вивчення, високу продуктивність для створення динамічних інтерфейсів користувача та наявність розвиненої екосистеми. Vite забезпечує швидку збірку проекту та зручне середовище для розробки клієнтської частини.

Розробка велася з використанням інтегрованого середовища розробки (IDE) Microsoft Visual Studio для серверної частини та Visual Studio Code для клієнтської частини, що забезпечують зручні інструменти для написання коду, відлагодження та управління проектом. Для забезпечення якості коду та дотримання єдиного стилю на клієнтській стороні використовується ESLint.

3.1.2 Архітектура розробленого програмного комплексу.

Розроблений програмний засіб для скремблювання даних реалізовано з використанням архітектури «клієнт-сервер». Такий підхід дозволяє чітко розділити логіку представлення даних та взаємодії з користувачем (клієнтська частина) від логіки обробки даних та виконання основних операцій скремблювання (серверна частина). Загальна архітектурна схема програмного комплексу наведена на рис. 3.1.

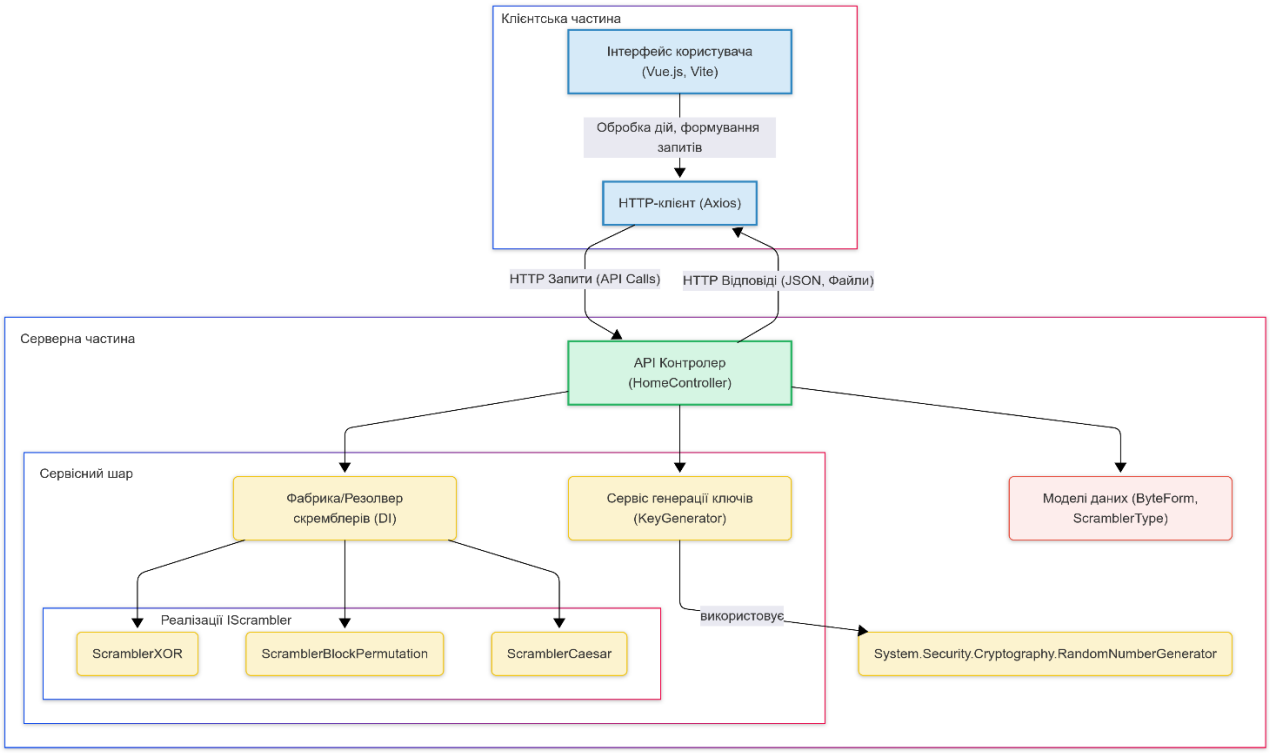

Рисунок 3.1 – Архітектурна схема програмного комплексу скремблювання даних

Як видно з наведеної схеми (рис. 3.1), клієнтська частина відповідає за формування користувацького інтерфейсу та взаємодію з користувачем. Вона розроблена з використанням сучасних веб-технологій, таких як фреймворк Vue.js та система збірки Vite. Основні функції клієнтської частини включають відображення елементів керування (поля введення, кнопки, списки вибору алгоритмів), обробку дій користувача (введення тексту, вибір файлів, вибір алгоритмів, натискання кнопок) та формування запитів до серверної частини. Для взаємодії з сервером використовується HTTP-клієнт (Axios), який надсилає HTTP-запити до серверного API та отримує відповіді (JSON, файли) для подальшого відображення користувачеві.

Серверна частина, побудована на платформі ASP.NET Core (.NET 8), виконує всю основну логіку обробки даних. Центральним елементом є API Контролер (HomeController), який приймає HTTP-запити від клієнта, обробляє їх та викликає відповідні компоненти Сервісного шару. Сервісний шар інкапсулює логіку і включає Фабрику/Резолвер скремблерів, реалізовану через механізм

впровадження залежностей (DI). Ця фабрика надає контролеру екземпляри конкретних реалізацій інтерфейсу IScrambler, таких як ScramblerXOR, ScramblerBlockPermutation та ScramblerCaesar. Також до сервісного шару належить Сервіс генерації ключів (KeyGenerator), який використовує системні криптографічні засоби (System.Security.Cryptography. RandomNumberGenerator) для створення випадкових ключів. Для обміну даними між клієнтом та сервером, а також всередині серверної частини використовуються Моделі даних, такі як ByteForm та перелічення ScramblerType.

Така архітектура забезпечує чіткий поділ відповідальностей між компонентами системи, сприяє її модульності, полегшує тестування окремих частин та подальшу підтримку і розширення функціоналу програмного засобу.

3.1.3 Модуль генерації ключів

Для забезпечення можливості використання випадкових та потенційно більш стійких ключів у системі реалізовано окремий модуль генерації ключів. Цей модуль інкапсульований у класі KeyGenerator, який реалізує інтерфейс IKeyGenerator.

Основним методом даного модуля є метод Generate(int length), який приймає як параметр бажану довжину ключа у байтах. Передбачено перевірку вхідного параметра: довжина ключа має бути додатним числом. Ця перевірка реалізована наступним чином:

```
    if (length <= 0)
    {
        throw new ArgumentOutOfRangeException(nameof(length),
"Length must be a positive number.");
    }
```

Для генерації самої послідовності байтів ключа використовується криптографічно стійкий генератор псевдовипадкових чисел, наданий

платформою .NET – клас System.Security.Cryptography.RandomNumberGenerator. Метод GetBytes() цього класу заповнює масив байтів криптографічно надійними випадковими значеннями, як показано у фрагменті коду:

```
    byte[] randomBytes = new byte[length];
    using (RandomNumberGenerator rng =
RandomNumberGenerator.Create())
    {
        rng.GetBytes(randomBytes);
    }
```

Це забезпечує високу ентропію генерованого ключа, що є важливим для ускладнення його можливого підбору або передбачення.

Після генерації масиву випадкових байтів, він перетворюється на рядкове представлення у шістнадцятковому форматі. Кожен байт представляється двома шістнадцятковими символами. Реалізація цього перетворення:

```
    return string.Concat(randomBytes.Select(b =>
b.ToString("x2")));
```

Таке представлення є зручним для відображення ключа користувачеві, його копіювання та подальшого використання у запитах до системи.

Цей модуль реєструється в системі впровадження залежностей, що дозволяє легко використовувати його в інших частинах програми, зокрема в контролері для обробки запитів на генерацію ключа.

3.1.4 Модуль скремблювання на основі операції XOR

Реалізація модуля скремблювання на основі побітової операції «виключне АБО» (XOR) є одним з фундаментальних компонентів розробленого

програмного засобу. Цей алгоритм, інкапсульований у класі ScramblerXOR, що реалізує інтерфейс IScrambler, забезпечує потокове перетворення даних.

Основний метод Scramble(byte[] data, byte[] key) приймає на вхід масив байтів даних та масив байтів ключа. Перед виконанням операцій здійснюється перевірка вхідних параметрів: якщо масив даних є null, генерується виняток ArgumentNullException. Якщо ключ є null або його довжина дорівнює нулю, то вхідні дані повертаються без змін, оскільки операція XOR з порожнім або відсутнім ключем не має сенсу для скремблювання.

```
if (data == null)
{
    throw new ArgumentNullException(nameof(data));
}
if (key == null || key.Length == 0)
{
    return data;
}
```

Процес скремблювання полягає у послідовному побайтовому застосуванні операції XOR до кожного байта вхідних даних з відповідним байтом ключа. Для цього створюється новий масив scrambledData такої ж довжини, як і вхідний масив даних. Обхід вхідних даних здійснюється в циклі, а для вибору байта ключа використовується індекс keyIndex, який циклічно змінюється в межах довжини ключа. Це забезпечує повторне використання ключа, якщо його довжина менша за довжину даних.

```
byte[] scrambledData = new byte[data.Length];
int keyIndex = 0;
for (int i = 0; i < data.Length; i++)
{
    scrambledData[i] = (byte)(data[i] ^ key[keyIndex]);
    keyIndex = (keyIndex + 1) % key.Length;
```

```
    }
    return scrambledData;
```

Результатом роботи методу є масив скрембльованих байтів.

Метод Descramble(byte[] data, byte[] key) для дескремблювання даних реалізований шляхом виклику того ж самого методу Scramble(data, key). Це є коректним, оскільки операція XOR є інволютивною (самооберненою), тобто подвійне застосування XOR з тим самим ключем повертає вихідне значення.

Реєстрація цього модуля в системі впровадження залежностей дозволяє гнучко використовувати його в контролері при обробці запитів користувача на скремблювання або дескремблювання даних за типом ScramblerType.XOR.

3.1.5 Модуль скремблювання методом блокової перестановки байтів

Модуль скремблювання методом блокової перестановки, реалізований у класі ScramblerBlockPermutation (який також імплементує інтерфейс IScrambler), призначений для зміни порядку байтів у межах окремих блоків даних. Це вносить елемент «перемішування», що руйнує локальні послідовності байтів.

Метод Scramble(byte[] data, byte[] key) спочатку виконує перевірку вхідних даних: якщо масив data є null або порожнім, він повертається без змін. Далі визначається розмір блоку blockSize, який встановлюється як мінімальне значення з 64 байтів або фактичної довжини вхідних даних (якщо вона менша за 64 байти). Потім обчислюється кількість блоків blocksCount, на які будуть розділені дані.

```
    if (data == null || data.Length == 0) return data;
    int blockSize = Math.Min(64, data.Length);
    byte[] result = new byte[data.Length];
    int blocksCount = (int)Math.Ceiling((double)data.Length /
blockSize);
```

Обробка даних відбувається поблоково в циклі. Для кожного блоку спочатку виділяється відповідна частина з вхідного масиву даних. Потім викликається приватний метод GeneratePermutationKey(key, blockLength) для генерації ключа перестановки для поточного блоку. Цей ключ є масивом байтів, що містить індекси для перестановки. Після цього, за допомогою приватного методу PermuteBlock(block, permutationKey), байти поточного блоку переставляються згідно зі згенерованим ключем перестановки. Результат копіюється у відповідну позицію вихідного масиву result.

```
     for (int blockIndex = 0; blockIndex < blocksCount;
blockIndex++)
     {
          int start = blockIndex * blockSize;
          int end = Math.Min(start + blockSize, data.Length);
          int blockLength = end - start;

          byte[] block =
data.Skip(start).Take(blockLength).ToArray();
          byte[] permutationKey = GeneratePermutationKey(key,
blockLength);
          byte[] scrambledBlock = PermuteBlock(block,
permutationKey);

          Array.Copy(scrambledBlock, 0, result, start,
blockLength);
     }
```

Метод Descramble(byte[] data, byte[] key) працює аналогічно до методу скремблювання, з тією відмінністю, що для перестановки байтів у блоці використовується обернений ключ перестановки. Цей обернений ключ генерується приватним методом InversePermutationKey(permutationKey) на основі того ж ключа перестановки, що використовувався при скремблюванні.

```
    byte[] permutationKey = GeneratePermutationKey(key,
blockLength);
    byte[] descrambledBlock = PermuteBlock(block,
InversePermutationKey(permutationKey));
```

Генерація ключа перестановки (GeneratePermutationKey) відбувається шляхом створення початкової послідовності індексів від 0 до blockLength - 1. Потім ця послідовність тасується за алгоритмом Фішера-Єйтса з використанням генератора псевдовипадкових чисел System.Random, який ініціалізується («seed») сумою байтів основного ключа, наданого користувачем.

```
    private byte[] GeneratePermutationKey(byte[] key, int
blockLength)
    {
        byte[] permutation = Enumerable.Range(0,
blockLength).Select(i => (byte)i).ToArray();
        int seed = key.Sum(b => b);
        Random rng = new Random(seed);

        for (int i = blockLength - 1; i > 0; i--)
        {
            int j = rng.Next(i + 1);
            (permutation[i], permutation[j]) = (permutation[j],
permutation[i]);
        }
        return permutation;
    }
```

Метод PermuteBlock безпосередньо виконує перестановку байтів згідно з отриманим ключем перестановки, а InversePermutationKey створює ключ, необхідний для відновлення початкового порядку байтів.

Цей модуль також реєструється в системі DI для використання при виборі ScramblerType.BlockPermutation.

3.1.6 Модуль скремблювання на основі адитивного перетворення

Третім реалізованим методом скремблювання є адитивне перетворення, яке є модифікацією класичного шифру Цезаря, адаптованою для обробки байтових даних. Цей модуль представлений класом ScramblerCaesar, що також реалізує інтерфейс IScrambler. Принцип дії полягає у побайтовому зсуві значень вхідних даних на величину, що визначається відповідним байтом ключа.

Метод Scramble(byte[] data, byte[] key) приймає масив байтів даних та ключ. На початку виконується перевірка: якщо масив даних data є null або порожнім, він повертається без змін. В іншому випадку створюється вихідний масив result тієї ж довжини, що й вхідний.

```
if (data == null || data.Length == 0) return data;
byte[] result = new byte[data.Length];
```

Далі, в циклі обробляється кожен байт вхідних даних. Значення поточного байта даних data[i] додається до значення відповідного байта ключа key[i % key.Length] (ключ використовується циклічно). Результат операції береться за модулем 256, щоб гарантувати, що скрембльоване значення залишиться в межах одного байта.

```
for (int i = 0; i < data.Length; i++)
{
    int keyVal = key[i % key.Length] % 256;
    result[i] = (byte)((data[i] + keyVal) % 256);
}
return result;
```

Процес дескремблювання, реалізований у методі Descramble(byte[] data, byte[] key), є оберненим до скремблювання. Він також виконує перевірку на null

або порожній масив даних. Для кожного скрембльованого байта від його значення віднімається значення відповідного байта ключа. Щоб коректно обробити операцію за модулем 256 (уникнути від'ємних результатів перед взяттям залишком), до різниці додається 256 перед обчисленням залишку.

```
for (int i = 0; i < data.Length; i++)
{
    int keyVal = key[i % key.Length] % 256;
    result[i] = (byte)((data[i] - keyVal + 256) % 256);
}
return result;
```

Даний модуль, як і інші алгоритми скремблювання, реєструється в системі впровадження залежностей для можливості його вибору користувачем під типом ScramblerType.Caesar.

3.1.7 Модуль каскадного (комбінованого) скремблювання

Однією з ключових особливостей розробленого програмного засобу є можливість комбінованого, або каскадного, застосування кількох алгоритмів скремблювання до одного набору даних. Логіка цієї функціональності реалізована переважно в серверній частині, зокрема в HomeController.

Принцип роботи модуля каскадного скремблювання полягає в послідовному виклику обраних користувачем алгоритмів скремблювання. Клієнтська частина передає на сервер список типів алгоритмів (List<ScramblerType>) у тому порядку, в якому їх слід застосувати. На сервері, в методах контролера, що відповідають за скремблювання (наприклад, Scramble та ScrambleFile), відбувається ітерація по цьому списку. На кожній ітерації за допомогою фабричного методу (реалізованого через Func<ScramblerType, IScrambler>) отримується екземпляр відповідного скремблера, і його метод

Scramble викликається для даних, що були результатом роботи попереднього алгоритму в каскаді (або вихідними даними для першого алгоритму).

Фрагмент коду, що ілюструє послідовне застосування алгоритмів при скремблюванні:

```
byte[] result = form.Data.ToArray();
var keyBytes = Encoding.UTF8.GetBytes(form.Key);
foreach (var algorithm in form.Algorithms)
{
    var scrambler = _scramblerResolver(algorithm);
    result = scrambler.Scramble(result, keyBytes);
}
```

Для процесу дескремблювання застосовується той самий набір алгоритмів, але у зворотному порядку. Це є критично важливим для коректного відновлення вихідних даних, оскільки кожна операція скремблювання має бути обернена у відповідній послідовності. У методах контролера, що відповідають за дескремблювання (UnscrambleByte та UnscrambleFile), список обраних алгоритмів спочатку інвертується за допомогою методу Reverse(). Після цього, аналогічно до процесу скремблювання, відбувається ітерація по інвертованому списку, і для кожного типу алгоритму викликається відповідний метод Descramble.

Фрагмент коду, що ілюструє інверсію порядку та послідовне дескремблювання:

```
byte[] result = form.Data.ToArray();
var keyBytes = Encoding.UTF8.GetBytes(form.Key);
form.Algorithms.Reverse();
foreach (var algorithm in form.Algorithms)
{
    var scrambler = _scramblerResolver(algorithm);
    result = scrambler.Descramble(result, keyBytes);
```

```
    }
```

Такий підхід до реалізації каскадного скремблювання дозволяє користувачеві гнучко комбінувати різні методи перетворення, потенційно ускладнюючи структуру скрембльованих даних. Усі алгоритми в каскаді використовують один і той самий ключ, наданий користувачем.

3.1.8 Опис інтерфейсу користувача

Інтерфейс користувача розробленого програмного скремблера реалізовано у вигляді веб-сторінки, що забезпечує доступність та зручність використання з будь-якого пристрою, оснащеного веб-браузером. Головна сторінка додатку (рис. 3.2) надає користувачеві всі необхідні інструменти для виконання операцій скремблювання та дескремблювання як текстових даних, так і файлів.

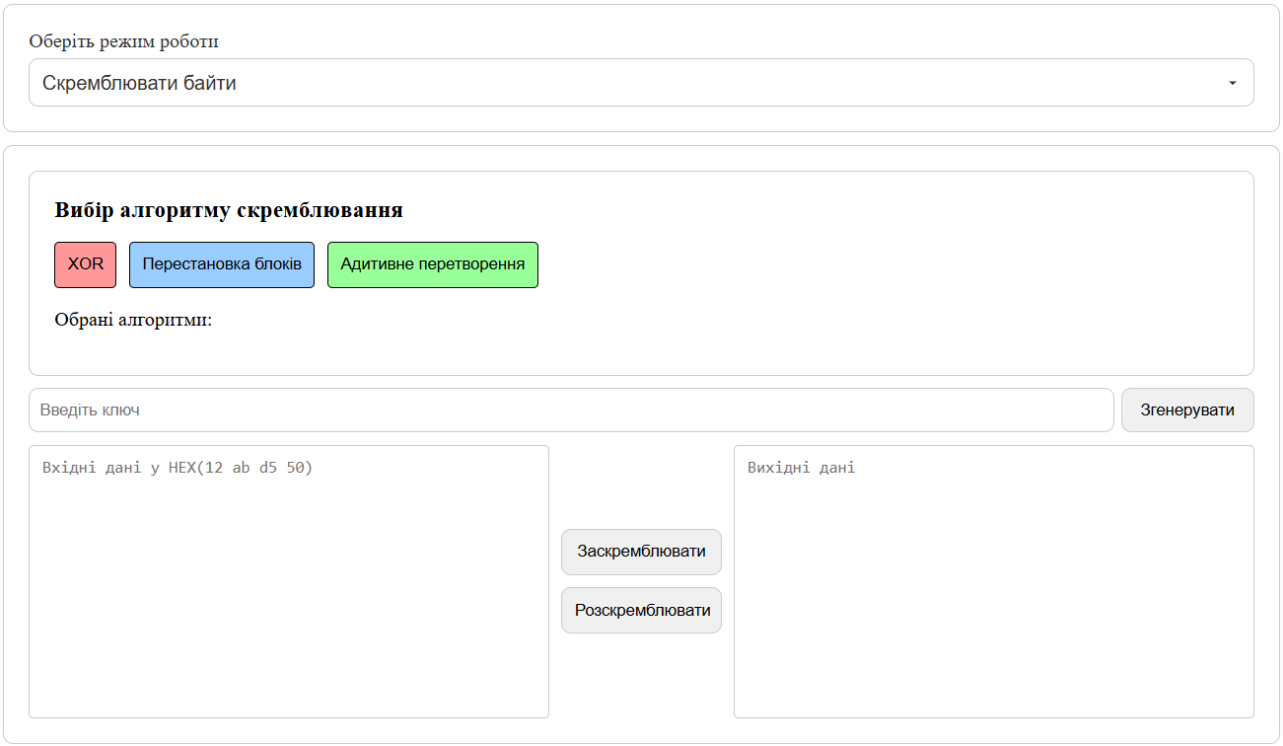

Рисунок 3.2 – Головна сторінка додатку

Першим кроком при роботі з програмою є вибір режиму роботи. Користувачеві пропонується випадаючий список (рис. 3.3), з якого можна обрати один з трьох основних режимів: «Скремблювати байти» (для роботи з байтами, які подаються у HEX-форматі), «Скремблювати текст» (для роботи з текстовими даними) та «Скремблювати файли» (для роботи з файлами). Цей вибір визначає, який тип даних буде оброблятися та які елементи інтерфейсу будуть активними для введення цих даних.

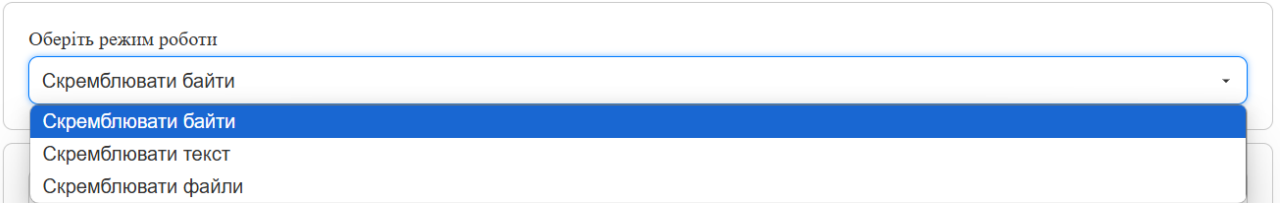

Рисунок 3.3. Вибір режиму роботи

Після вибору режиму, користувачеві стають доступними інші елементи керування:

1. Вибір алгоритмів скремблювання (рис. 3.4)

Набір кнопок дозволяє обрати один або декілька алгоритмів (XOR, блокова перестановка, адитивне перетворення). Обрані алгоритми додаються до візуального списку.

Рисунок 3.4 – Вибір алгоритмів скремблювання

Інтерфейс підтримує можливість зміни порядку застосування алгоритмів методом перетягування (drag-and-drop) для гнучкого налаштування каскаду перетворень (рис. 3.5). Кожен елемент у списку також має кнопку для його видалення.

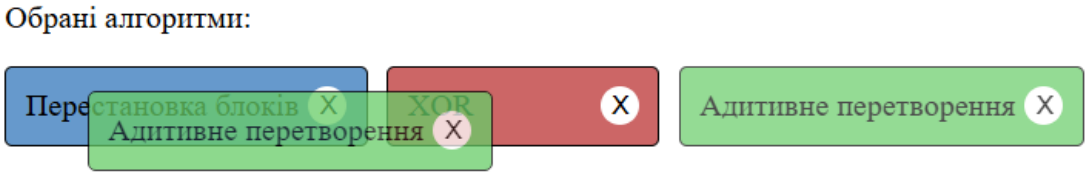

Рисунок 3.5 – Зміна послідовності обраних алгоритмів

2. Поле для введення ключа (рис. 3.6)

Окреме текстове поле призначене для введення або вставки ключа. Ключ може бути введеним будь-якими символами будь-якої довжини, які автоматично перетворяться у масив байтів. Поруч із цим полем розташована кнопка для автоматичної генерації ключа, що ініціює запит до серверної частини на генерацію ключа довжиною 256 біт.

Рисунок 3.6 – Поле для введення або генерації ключа

3. Поля для введення та відображення даних

Вигляд поля залежить від вибору режиму роботи. У режимі «Скремблювати байти» та «Скремблювати текст» передбачено два текстових поля (`<textarea>`), розташованих поруч. Одне призначене для введення вхідного тексту, а інше – для відображення результату скремблювання/дескремблювання (рис. 3.7). Також присутній набір кнопок «Заскремблювати» та «Розскремблювати», який ініціює процеси обробки даних.

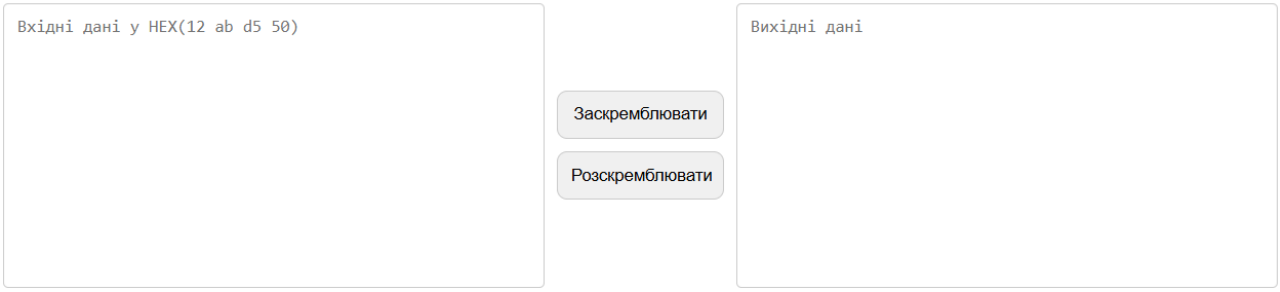

Рисунок 3.7 – Поля для введення та відображення даних

У режимі «Скремблювати файли» інтерфейс змінюється на набір кнопок «Обрати файл», «Заскремблювати» та «Розскремблювати» (рис. 3.8). При натисканні кнопки «Обрати файл» ініціалізується запит до операційної системи, у якому користувач обирає файл для обробки. Після вибору файлу його назва з’являється у інтерфейсі.

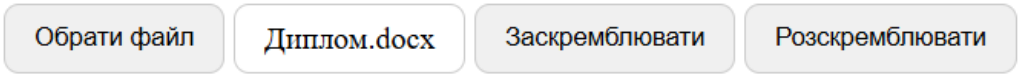

Рисунок 3.8 – Інтерфейс у режимі роботи з файлами

При натисканні кнопки «Заскремблювати» файл з ключем та обраними алгоритмами відправляється на сервер, проводиться скремблювання. Далі скрембльований файл повертається до клієнта з префіксом у назві «scrambled_». При натисканні кнопки «Розскремблювати» відбуваються аналогічні дії з відмінністю у тому що приймаються лише файли з префіксом «scrambled_», та при відправці до клієнта даний префікс прибирається.

Загальний дизайн інтерфейсу, визначений у globals.css, спрямований на простоту та інтуїтивність. Клієнтська логіка, реалізована на Vue.js, забезпечує динамічну взаємодію та асинхронний обмін даними з сервером.

3.2 Експериментальне дослідження розробленого програмного скремблера

Після завершення розробки програмного засобу для скремблювання даних, описаного в підрозділі 3.1, було проведено серію експериментальних досліджень. Метою цих досліджень була перевірка коректності реалізації заявленого функціоналу, зокрема точності відновлення даних після циклу «скремблювання-дескремблювання», та детальна оцінка часових характеристик (швидкодії) розроблених алгоритмів скремблювання (XOR, блокова перестановка, адитивне перетворення) та їх комбінацій при обробці різних обсягів даних. Основна увага при оцінці швидкодії приділялася вимірюванням безпосередньо на серверній частині за допомогою спеціально розроблених тестів (юніт-тестів або спеціальних тестових методів) для мінімізації впливу зовнішніх факторів, таких як мережеві затримки або продуктивність клієнтського веб-браузера.

3.2.1 Методика проведення експериментів

Для забезпечення об'єктивності та відтворюваності результатів експериментального дослідження було розроблено відповідну методику, що включає визначення тестового середовища, формування наборів тестових даних, опис критеріїв оцінки та порядку проведення експериментів.

Експерименти з оцінки коректності та швидкодії алгоритмів проводилися безпосередньо на серверній частині програмного комплексу. Для цього використовувався персональний комп'ютер з наступними основними характеристиками: процесор AMD Ryzen 7 5800H 3200Мг, обсяг оперативної пам'яті 16 ГБ, тип накопичувача SSD, операційна система Windows 11. Програмне забезпечення включало середовище виконання .NET 8. Для точного вимірювання часу виконання операцій скремблювання та дескремблювання використовувався клас System.Diagnostics.Stopwatch мови C#. Функціональне

тестування веб-інтерфейсу та коректності взаємодії клієнт-сервер здійснювалося у веб-браузері Opera GX.

Для дослідження були підготовлені масиви байтів, що представляли дані різного обсягу. Для тестів швидкодії використовувалися набори даних розміром 1 МБ (1 048 576 байт), 10 МБ (10 485 760 байт) та 100 МБ (104 857 600 байт). Ці масиви байтів генерувалися псевдовипадковим чином для імітації обробки різнорідної інформації. Для тестів на коректність та функціональних тестів також використовувалися менші текстові рядки та файли різних типів. Ключі для експериментів з оцінки швидкодії використовувалися довжиною 8 біт, 32 біт та 256 біт, згенеровані псевдовипадковим чином.

Основними критеріями оцінки розробленого програмного скремблера були:

Коректність функціонування: Перевірка точності відновлення вихідних даних після виконання послідовності операцій «скремблювання-дескремблювання» з використанням того самого ключа для всіх реалізованих алгоритмів та їх комбінацій. Основний показник – повне побайтове співпадіння вихідного та відновленого масивів даних.

Швидкодія: Вимірювання середнього часу (в мілісекундах), необхідного для виконання операцій скремблювання та дескремблювання (окремо) для кожного з трьох базових алгоритмів. Час вимірювався для трьох зазначених розмірів вхідних даних (1 МБ, 10 МБ, 100 МБ) та трьох довжин ключа (8, 32, 256 байт).

Порядок проведення експериментів з оцінки швидкодії.

1. Для кожного з трьох базових алгоритмів скремблювання (ScramblerXOR, ScramblerBlockPermutation, ScramblerCaesar) були створені спеціальні тестові методи.

2. Для кожної комбінації «розмір даних – довжина ключа» готувався відповідний масив вхідних байтів та масив байтів ключа.

3. Перед початком вимірювань виконувався один «прогрівочний» запуск операції скремблювання та дескремблювання для стабілізації середовища виконання.

4. Безпосередньо операція скремблювання (а потім окремо операція дескремблювання для відповідних скрембльованих даних) виконувалася багаторазово: 20 разів для даних обсягом 1 МБ, 5 разів для 10 МБ, та 2 рази для 100 МБ. Час кожного виконання фіксувався за допомогою System.Diagnostics.Stopwatch.

5. Обчислювався середній час виконання для кожної серії тестів.

6. Паралельно перевірялася коректність відновлення даних.

7. Отримані середні значення часу виконання фіксувалися для подальшого аналізу та представлення у вигляді таблиць та/або графіків.

3.2.2 Тестування функціональності програмного засобу

Перед проведенням детальних вимірювань швидкодії, першочерговим завданням було тестування загальної функціональності розробленого програмного скремблера. Метою цього етапу була перевірка коректності виконання всіх заявлених операцій, правильності обробки даних та взаємодії компонентів системи. Тестування охоплювало як окремі алгоритми скремблювання, так і їх комбіноване (каскадне) застосування. Для цього використовувалися як спеціально створені юніт-тести для перевірки логіки серверної частини, так і ручне тестування через розроблений веб-інтерфейс для оцінки роботи системи в цілому.

Ключові аспекти, що підлягали перевірці, включали коректність операцій скремблювання та дескремблювання. Для кожного з трьох базових алгоритмів – XOR, блокової перестановки та адитивного перетворення (модифікований шифр Цезаря) – та для різних їх комбінацій перевірялася фундаментальна умова: дані після послідовного застосування операцій скремблювання та дескремблювання з використанням одного й того ж ключа повинні повністю співпадати з

вихідними даними. Це тестувалося на різноманітних наборах текстових даних (включаючи рядки різної довжини, з різними наборами символів) та на файлах різних типів (текстові, бінарні), що дозволило переконатися у правильності реалізації алгоритмів. Юніт-тести, приклади яких були наведені раніше, автоматизували цю перевірку шляхом побайтового порівняння вихідного та відновленого масивів даних.

Окрему увагу було приділено роботі з ключами. Тестувалася функція генерації ключа, включаючи генерацію ключів різної заданої довжини та оцінку їхнього представлення у шістнадцятковому форматі. Перевірялося, що використання різних ключів призводить до різних результатів скремблювання для однакових вхідних даних, а також тестувалася поведінка системи при використанні ключів різної довжини відносно довжини даних. Також перевірялася коректна обробка некоректних або порожніх ключів на рівні контролера.

Тестування обробки різних типів даних охоплювало як текстові дані, що вводяться через веб-інтерфейс (з урахуванням їх перетворення в байтовий масив у кодуванні UTF-8 на сервері), так і файлові дані. Для файлів перевірялися процеси завантаження, скремблювання, повернення скрембльованого файлу, а також аналогічний зворотний процес дескремблювання. Контролювалася коректність обробки імен файлів (додавання та видалення префіксів «scrambled_» та «unscrambled_») та цілісність файлів після перетворень. Окремо тестувалася обробка граничних випадків, таких як завантаження порожнього файлу або спроба дескремблювання файлу без відповідного префіксу імені.

Функціональність каскадного скремблювання також була ретельно перевірена. Тестувалася можливість вибору користувачем декількох алгоритмів та їх послідовне застосування до даних. Ключовим моментом була перевірка того, що при дескремблюванні алгоритми застосовуються у строго зворотному порядку, що забезпечує коректне відновлення вихідних даних, як це реалізовано в логіці контролера.

Нарешті, перевірялася загальна робота інтерфейсу користувача: коректність функціонування всіх елементів керування (вибір режиму роботи, введення даних, генерація ключа, вибір та можливе впорядкування алгоритмів, завантаження/вивантаження файлів) та адекватність реакції інтерфейсу на дії користувача, включаючи відображення повідомлень про помилки або успішне виконання операцій.

За результатами тестування функціональності було встановлено, що розроблений програмний засіб коректно виконує всі заявлені операції скремблювання та дескремблювання для обраних алгоритмів та їх комбінацій, обробляючи як текстові дані, так і файли, за умови використання однакового ключа та правильної послідовності операцій.

3.2.3 Дослідження часових характеристик (швидкодії)

Одним із ключових аспектів експериментального дослідження розробленого програмного скремблера була оцінка його швидкодії. Для цього було проведено вимірювання середнього часу виконання операцій скремблювання та дескремблювання для кожного з трьох базових алгоритмів: на основі операції XOR, методом блокової перестановки байтів та на основі адитивного перетворення (шифр Цезаря). Дослідження проводилося для наборів даних різного обсягу (1 МБ, 10 МБ та 100 МБ) та з використанням ключів різної довжини (8 біт, 32 біт та 256 біт) згідно з методикою, описаною в підрозділі 3.2.1. Детальні таблиці з числовими результатами вимірювань наведено у Додатку Ж.

На графіку (рис. 3.9) чітко простежується лінійна залежність часу обробки від обсягу вхідних даних як для операції скремблювання, так і для дескремблювання. При цьому час виконання цих двох операцій є практично ідентичним, в межах похибки вимірювання, що підтверджує теоретичні очікування, пов'язані з інволютивністю операції XOR.

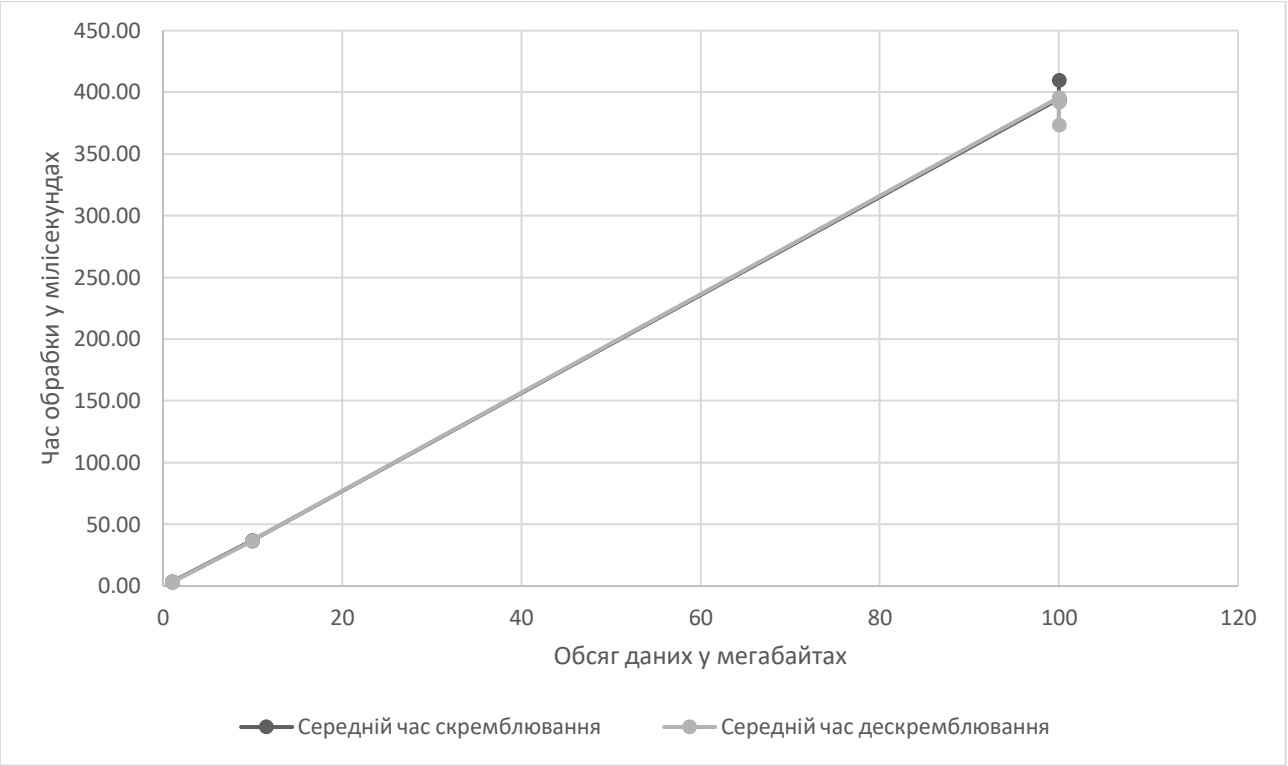

Рисунок 3.9 – Залежність часу виконання XOR-скремблювання від обсягу даних

Вплив довжини ключа на загальну швидкодію алгоритму виявився мінімальним (рис. 3.10), що також узгоджується з тим, що основні обчислювальні витрати припадають на побайтові операції XOR, кількість яких залежить лише від обсягу даних.

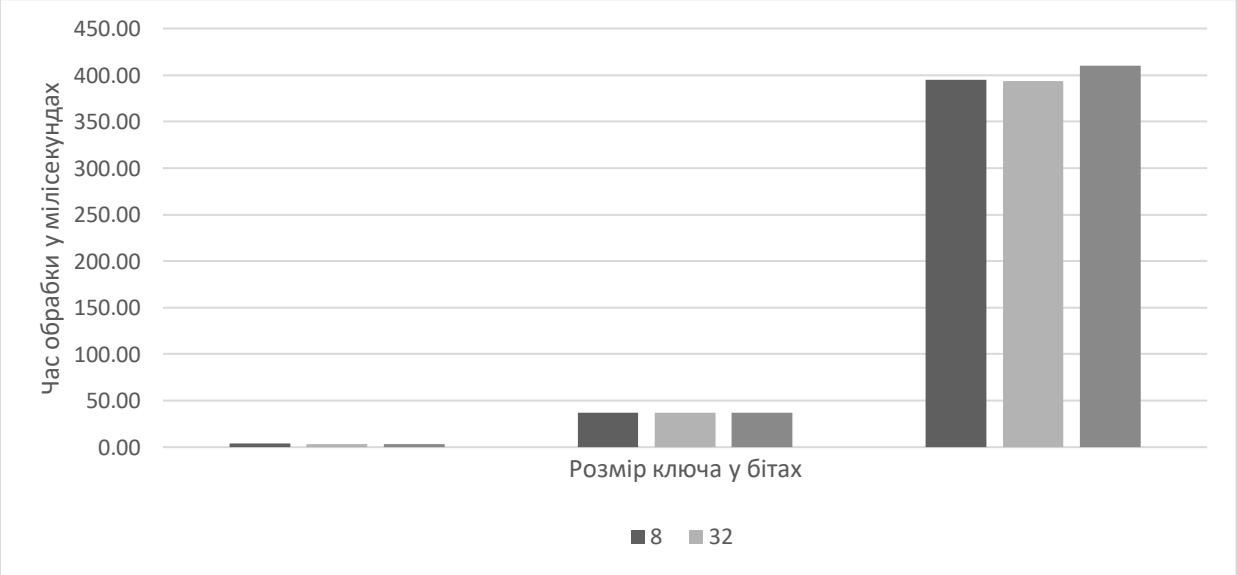

Рисунок 3.10 – Вплив довжини ключа на час виконання XOR-скремблювання

Для алгоритму скремблювання методом блокової перестановки байтів (з розміром блоку 64 байти) також були проведені дослідження часових характеристик. Як видно з графіку (рис. 3.11), даний алгоритм демонструє значно більший час обробки порівняно з XOR-скремблюванням. Це пояснюється складністю операцій, що включають поділ даних на блоки, генерацію унікального ключа перестановки для кожного блоку на основі користувацького ключа (з використанням System.Random, ініціалізованого сумою байтів ключа), та безпосередньо операції перестановки байтів у кожному блоці, що включають копіювання даних (зокрема, через використання Skip().Take().ToArray()). Час дескремблювання є дещо більшим через необхідність обчислення оберненої перестановки. Залежність часу обробки від обсягу даних також має тенденцію до лінійної, але з суттєво більшим коефіцієнтом пропорційності.

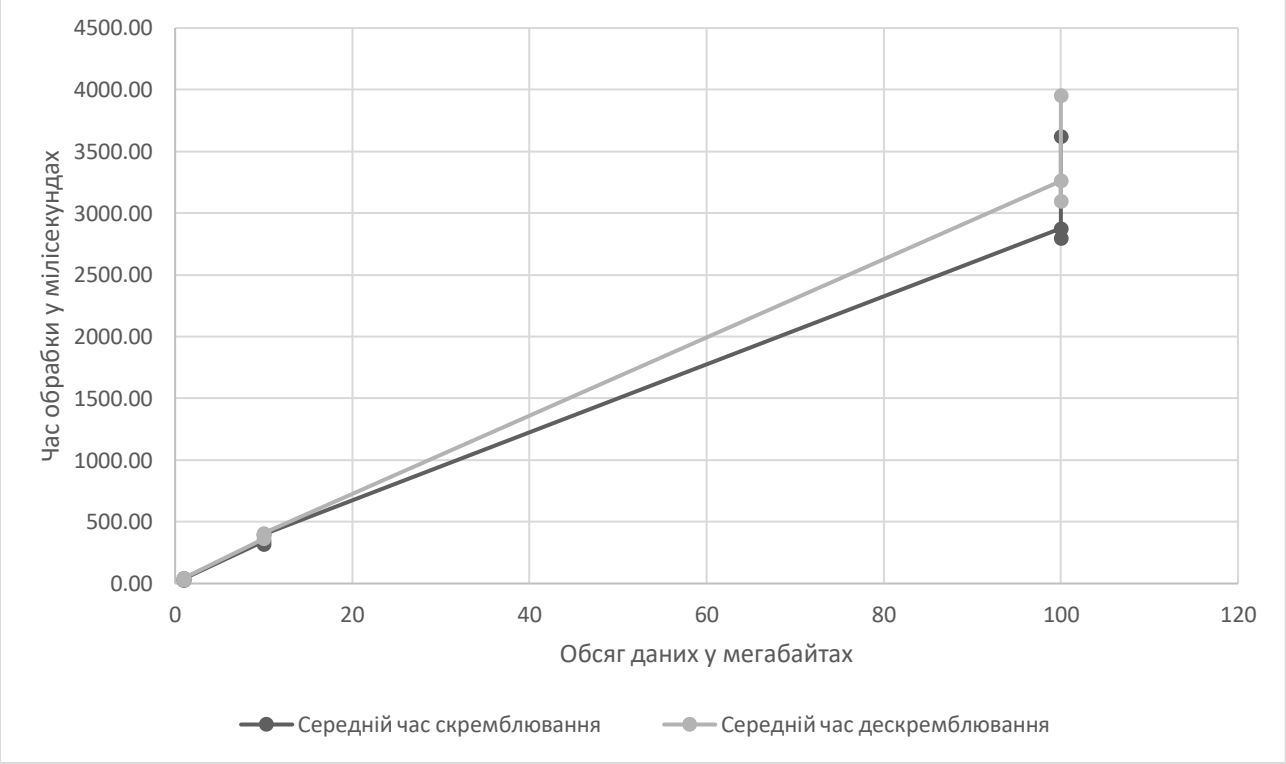

Рисунок 3.11 – Залежність часу виконання блокової перестановки від обсягу даних

Вплив довжини ключа на загальну швидкість, як і у випадку XOR, є менш вираженим (рис. 3.12).

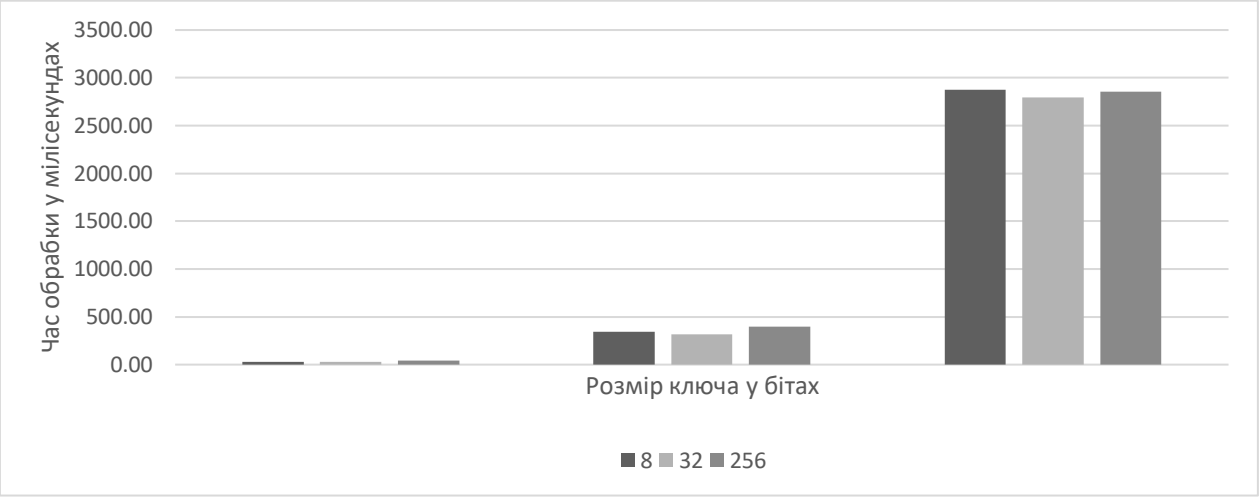

Рисунок 3.12 – Вплив довжини ключа на час виконання блокової перестановки

Результати вимірювань для алгоритму скремблювання на основі адитивного перетворення (шифр Цезаря) наведені на рис. 3.13. Даний алгоритм, подібно до XOR-скремблювання, показує високу швидкодію. Час виконання операцій скремблювання та дескремблювання є практично однаковим, оскільки вони базуються на простих побайтових арифметичних операціях (додавання або віднімання за модулем).

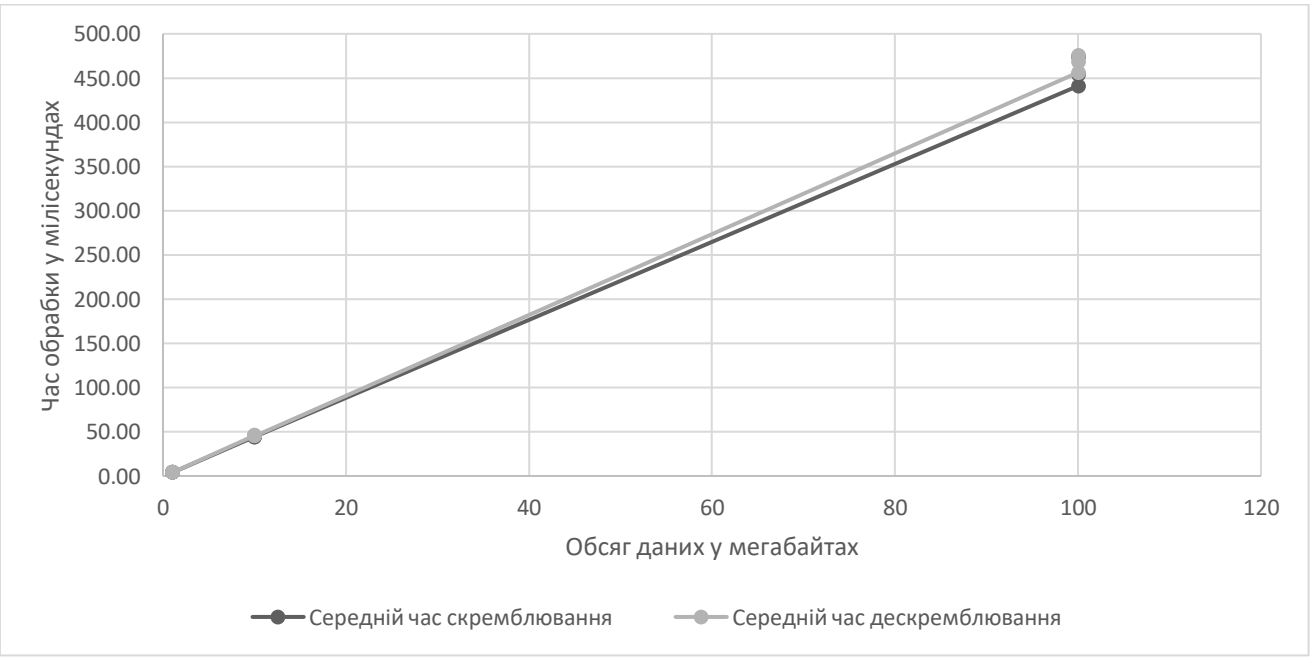

Рисунок 3.13 – Залежність часу виконання блокової перестановки від обсягу даних

Залежність часу від обсягу даних є лінійною, а вплив довжини ключа на продуктивність – мінімальним (рис. 3.14).

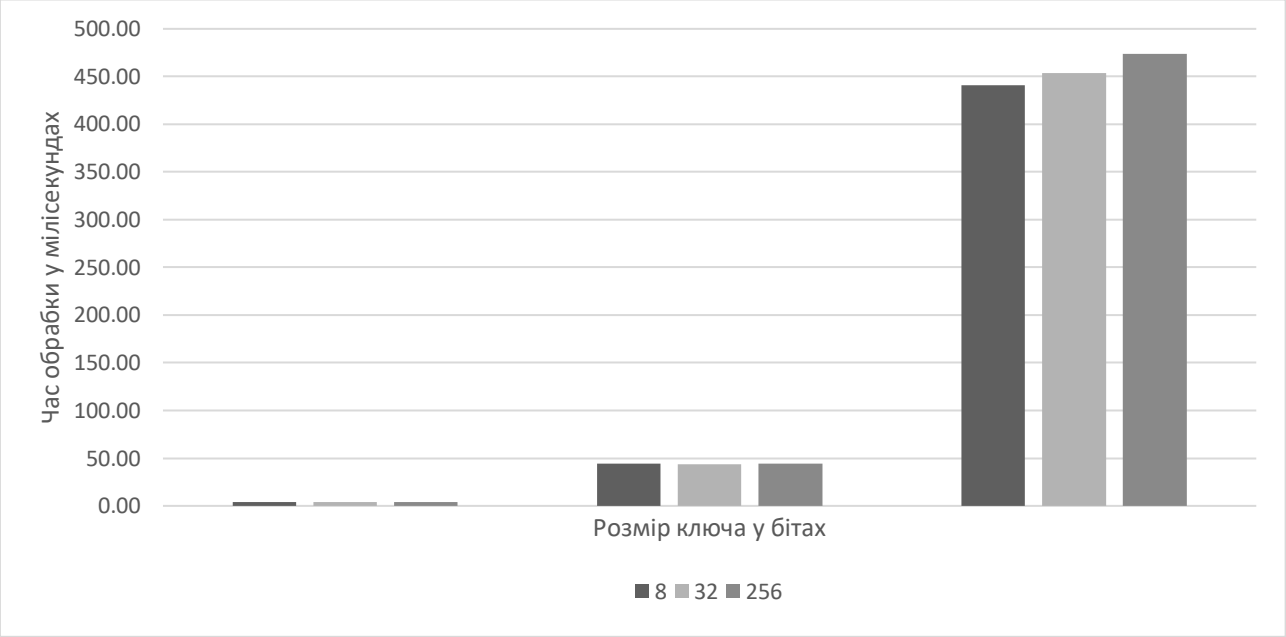

Рисунок 3.14 – Вплив довжини ключа на час виконання блокової перестановки

Порівняльний аналіз швидкодії трьох реалізованих алгоритмів (приклад наведено на рис. 3.15 для даних обсягом 10 МБ та ключа довжиною 32 біта) чітко показує, що алгоритми XOR та адитивного перетворення є значно ефективнішими з точки зору часових витрат, ніж алгоритм блокової перестановки в його поточній реалізації.

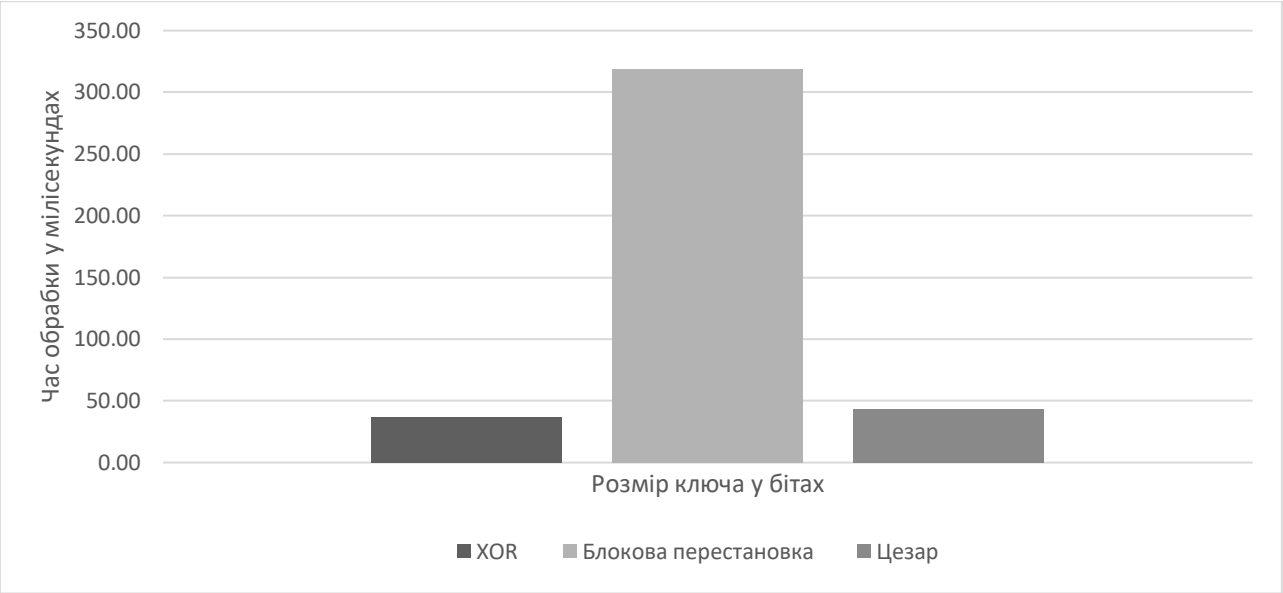

Рисунок 3.15 – Порівняння середнього часу скремблювання для різних алгоритмів

Проведене дослідження часових характеристик дозволяє зробити висновок, що для завдань, де критичною є швидкість обробки великих обсягів даних, найбільш прийнятними є алгоритми, що базуються на операції XOR або простому адитивному перетворенні. Алгоритм блокової перестановки, хоча й пропонує інший механізм обфускації, потребує подальшої оптимізації для ефективної роботи з великими файлами.

3.3. Аналіз результатів експериментальних досліджень та практичні рекомендації

Проведене експериментальне дослідження розробленого програмного скремблера, що включало тестування функціональності (підрозділ 3.2.2) та оцінку часових характеристик (підрозділ 3.2.3), дозволило отримати низку важливих результатів, які тепер підлягають узагальненню та аналізу. На основі цих результатів можна зробити висновки щодо відповідності програмного засобу поставленим вимогам, а також сформулювати практичні рекомендації щодо його застосування.

Узагальнення результатів тестування функціональності підтвердило коректність реалізації всіх основних заявлених можливостей програмного

скремблера. Було встановлено, що кожен з трьох базових алгоритмів скремблювання (на основі операції XOR, блокової перестановки байтів та адитивного перетворення (шифр Цезаря)) забезпечує повне відновлення вихідних даних після послідовного застосування операцій скремблювання та дескремблювання з використанням одного й того ж ключа. Модуль генерації ключів успішно створює ключі заданої довжини. Функція каскадного (комбінованого) скремблювання коректно застосовує обрані користувачем алгоритми в заданій послідовності та у зворотному порядку при дескремблюванні. Програмний засіб продемонстрував здатність обробляти як текстові дані, так і файли. Веб-інтерфейс користувача виявився функціональним та інтуїтивно зрозумілим.

Аналіз часових характеристик (швидкодії), результати якого представлені графічно у підрозділі 3.2.3 (з детальними табличними даними у Додатку Ж), показав, що алгоритми скремблювання на основі операції XOR та адитивного перетворення (шифр Цезаря) демонструють найвищу швидкість обробки даних. Час виконання для цих алгоритмів практично лінійно залежить від обсягу даних і залишається низьким навіть для файлів розміром 100 МБ. Довжина ключа має мінімальний вплив на їхню продуктивність. Натомість, алгоритм блокової перестановки байтів у поточній реалізації виявився значно повільнішим, особливо при обробці великих обсягів даних, що пов'язано з накладними витратами на операції з блоками. Час скремблювання та дескремблювання для кожного окремого алгоритму є співмірним.

Оцінка відповідності розробленого програмного засобу поставленим вимогам (сформульованим у підрозділі 2.1.1) загалом є позитивною. Реалізовано підтримку декількох алгоритмів, можливість їх каскадного застосування, використання ключа, оборотність перетворень, обробка текстових даних та файлів, а також забезпечено модульність архітектури. Швидкодія для алгоритмів XOR та Цезаря відповідає очікуванням високої ефективності. Проте, ефективність алгоритму блокової перестановки для великих файлів потребує

покращення. Вимога щодо високої криптографічної стійкості свідомо не ставилася, і реалізовані алгоритми призначені для цілей обфускації даних.

На основі проведеного дослідження можна сформулювати наступні практичні рекомендації щодо використання програмного скремблера. Розроблений програмний засіб рекомендується використовувати для завдань, де не потрібен високий рівень криптографічного захисту, але є потреба в швидкому «заплутуванні» даних, наприклад, для ускладнення випадкового несанкціонованого перегляду файлів або текстових повідомлень, для маскування даних при підготовці тестових наборів або для демонстраційних та освітніх цілей. При виборі алгоритмів для максимальної швидкодії при базовій обфускації доцільно використовувати алгоритми XOR або адитивного перетворення. Алгоритм блокової перестановки може бути корисним для руйнування локальної структури даних, але його слід використовувати з обережністю для файлів великого розміру. Каскадне застосування кількох алгоритмів може дещо посилити ефект обфускації, однак це збільшить час обробки. Рекомендується використовувати ключі достатньої довжини (наприклад, не менше 16-32 біт), згенеровані за допомогою вбудованої функції програми. Важливо пам'ятати, що даний програмний скремблер не є заміною повноцінним системам шифрування.

Можливі напрямки подальшого вдосконалення програмного засобу включають оптимізацію алгоритму блокової перестановки для підвищення його швидкодії; додавання нових алгоритмів скремблювання; реалізацію можливості використання різних ключів для різних етапів каскадного скремблювання; розширення функціоналу веб-інтерфейсу; та проведення більш глибокого аналізу стійкості реалізованих алгоритмів та їх комбінацій до різних типів атак, хоча це виходить за рамки першочергових завдань для скремблера, орієнтованого на обфускацію.

Проведене дослідження та розробка програмного засобу підтвердили можливість створення гнучкого інструменту для програмного скремблювання даних, що може бути корисним у певних практичних сценаріях.

Висновки до розділу 3

У третьому розділі кваліфікаційної роботи було детально описано процес розробки програмного засобу для скремблювання даних та представлено результати його експериментального дослідження.

В рамках розробки було обрано сучасні інструментальні засоби: мова програмування C# та платформа .NET 8 (ASP.NET Core) для серверної частини, фреймворк Vue.js для клієнтської частини. Запропоновано та описано клієнт-серверну архітектуру програмного комплексу, що включає API контролер для обробки запитів, сервісний шар з реалізаціями трьох алгоритмів скремблювання (на основі операції XOR, методом блокової перестановки байтів та на основі адитивного перетворення (шифр Цезаря)), модуль генерації ключів та механізм для їх каскадного застосування. Також було описано реалізацію веб-інтерфейсу користувача, що надає доступ до всіх функцій програми.

Експериментальне дослідження розробленого програмного скремблера включало тестування його функціональності та оцінку часових характеристик. Тестування функціональності підтвердило коректність виконання всіх заявлених операцій, включаючи скремблювання та дескремблювання текстових даних і файлів кожним алгоритмом окремо та їх комбінаціями, а також правильність роботи функції генерації ключів. Було підтверджено, що дані повністю відновлюються після циклу «скремблювання-дескремблювання» за умови використання однакового ключа та правильної послідовності операцій.

Дослідження швидкодії показало, що алгоритми скремблювання на основі операції XOR та адитивного перетворення (шифр Цезаря) демонструють високу продуктивність, обробляючи дані обсягом 100 МБ за час порядку сотень мілісекунд. Час їх виконання лінійно залежить від обсягу даних, тоді як вплив довжини ключа є мінімальним. Алгоритм блокової перестановки байтів у поточній реалізації виявився значно повільнішим, особливо на великих обсягах даних, що вказує на потенційні напрямки для оптимізації.

Аналіз отриманих результатів свідчить, що розроблений програмний засіб відповідає більшості поставлених вимог, зокрема щодо підтримки кількох алгоритмів, їх комбінування, використання ключа та обробки різних типів даних. Було сформульовано практичні рекомендації щодо використання розробленого скремблера для завдань, де не потрібна висока криптографічна стійкість, та окреслено можливі шляхи його подальшого вдосконалення.

Таким чином, у третьому розділі було успішно реалізовано та досліджено програмний скремблер даних, що підтверджує досягнення практичної частини завдань, поставлених у даній роботі.

ВИСНОВКИ

У даній кваліфікаційній роботі було проведено дослідження теоретичних основ та практичних аспектів захисту інформації з використанням програмного скремблера. Метою роботи було дослідження методів програмного скремблювання даних, розробка програмного засобу, що реалізує обрані алгоритми та їх комбінації, та аналіз його властивостей. Для досягнення поставленої мети було вирішено низку завдань.

У першому розділі було проведено комплексний аналіз теоретичних основ захисту інформації та специфіки застосування методів програмного скремблювання. Було підтверджено актуальність проблеми інформаційної безпеки в сучасних умовах, визначено ключову термінологію, розглянуто класифікацію загроз та окреслено місце скремблювання серед інших засобів захисту. Детальний огляд наявних методів скремблювання охопив їхні принципи, історичний розвиток, класифікацію, а також аналіз відомих алгоритмів та програмних продуктів. Аналіз проблемних аспектів виявив обмеження програмного скремблювання, зокрема, щодо стійкості до цілеспрямованого аналізу та питань ефективності, що обґрунтувало необхідність подальших досліджень у цій сфері.

У другому розділі було закладено теоретичний фундамент для розробки власного програмного скремблера. На основі аналізу вимог до програмного засобу, що включали підтримку декількох алгоритмів, можливість їх каскадного застосування, використання ключів та оборотність перетворень, було обґрунтовано вибір напрямку дослідження. Було обрано та теоретично проаналізовано три базові алгоритми скремблювання: на основі побітової операції XOR, методом блокової перестановки байтів та на основі адитивного перетворення (модифікований шифр Цезаря для байтів). Також було розглянуто принципи їх комбінованого (каскадного) застосування та теоретичні аспекти

генерації і використання ключів. Це створило необхідні передумови для практичної реалізації та подальшого дослідження програмного засобу.

У третьому розділі було описано процес розробки програмного засобу для скремблювання даних та представлено результати його експериментального дослідження. Було обґрунтовано вибір інструментальних засобів (C#, .NET 8, ASP.NET Core для серверної частини; Vue.js, Vite для клієнтської частини) та детально описано клієнт-серверну архітектуру розробленого веб-додатку, включаючи реалізацію основних функціональних модулів: генерації ключів, трьох алгоритмів скремблювання, механізму їх каскадного застосування та інтерфейсу користувача. Експериментальне дослідження підтвердило коректність функціонування розробленого програмного засобу для всіх заявлених операцій. Дослідження швидкодії показало високу продуктивність алгоритмів XOR та адитивного перетворення, тоді як алгоритм блокової перестановки у поточній реалізації виявився значно повільнішим на великих обсягах даних. На основі аналізу результатів було сформульовано практичні рекомендації щодо використання розробленого скремблера та окреслено можливі шляхи його подальшого вдосконалення.

Таким чином, усі поставлені на початку кваліфікаційної роботи завдання були виконані, а мета дослідження – досягнута.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Гулак Г. М. Методологія захисту інформації. Аспекти кібербезпеки : навч. посіб. Київ : НА СБ України, 2020. 248 с.

2. Gaži P., Maurer U. Cascade Encryption Revisited. IACR Cryptology ePrint Archive. 2009. Report 2009/093 [Електронний ресурс]. – URL: https://eprint.iacr.org/2009/093.pdf. – Дата звернення: 17.01.2025.

3. Leblanc C. 6 Effective Data Transformation Techniques. CData Blog. 2025. 9 січ. [Електронний ресурс]. – URL: https://www.cdata.com/blog/data-transformation-techniques. – Дата звернення: 17.01.2025.

4. Arulanandam S., Selvan P. A. A Review of Analog Audio Scrambling Methods for Residual Intelligibility. ResearchGate. 2012. січ. [Електронний ресурс]. – URL: https://www.researchgate.net/publication/279494781_A_Review_of_AnalogAudio_Scrambling_Methods_for_Residual_Intelligibility. – Дата звернення: 17.01.2025.

5. Data Scrambling | What It Is and How It Works. Perforce Blog. 2021. 1 січ. [Електронний ресурс]. – URL: https://www.perforce.com/blog/pdx/data-scrambling. – Дата звернення: 17.01.2025.

6. Data Scrambling vs. Encryption: Key Differences You Should Know. SecureITworld. 2024. 26 вер. [Електронний ресурс]. – URL: https://www.secureitworld.com/cloud-security/data-scrambling-vs-encryption-key-differences-you-should-know/. – Дата звернення: 18.01.2025.

7. Cheng L. B., Yeh R. J. Trial encoding algorithms ensemble. SpringerPlus. 2013. Vol. 2, Art. 316 [Електронний ресурс]. – URL: https://pmc.ncbi.nlm.nih.gov/articles/PMC4796158/. – Дата звернення: 19.01.2025.

8. Zheng J., Bao T. An Image Encryption Algorithm Using Cascade Chaotic Map and S-Box. Entropy. 2022. Vol. 24, No. 12. Art. 1827 [Електронний ресурс]. – URL: https://pmc.ncbi.nlm.nih.gov/articles/PMC9777660/. – Дата звернення: 19.01.2025.

9. Additive Ciphers (Vernam One-Time-Pad). Stanford CS [Електронний ресурс]. – URL: https://cs.stanford.edu/people/eroberts/courses/soco/projects/2008-09/colossus/addativeciphers.html. – Дата звернення: 19.01.2025.

10. Anupriya E., Soni S., Agnihotri A., Babelay S. Encryption using XOR based Extended Key for Information Security – A Novel Approach. International Journal on Computer Science and Engineering. 2011. Vol. 3, No. 1. P. 231–238.

11. Caesar Cipher in Cryptography. GeeksforGeeks. 2025. 23 квіт. [Електронний ресурс]. – URL: https://www.geeksforgeeks.org/caesar-cipher-in-cryptography/. – Дата звернення: 19.01.2025.

12. Christensen C. Permutation Ciphers. MAT/CSC 483, Northern Kentucky University. Spring 2015 [Електронний ресурс]. – URL: https://websites.nku.edu/~christensen/1402%20permutation%20ciphers.pdf. – Дата звернення: 20.01.2025.

13. Qu L., He H., Chen F. On the Security of Block Permutation and Co-XOR in Reversible Data Hiding. IEEE Transactions on Circuits and Systems for Video Technology. 2021. Vol. 32, No. 3. P. 920–933 [Електронний ресурс]. – URL: https://www.researchgate.net/publication/350501776_On_the_Security_of_Block_Permutation_and_Co-XOR_in_Reversible_Data_Hiding/download. – Дата звернення: 20.01.2025.

14. Security of cascaded ciphers, including question on counter-example from a paper. Cryptography Stack Exchange [Електронний ресурс]. – URL: https://crypto.stackexchange.com/questions/112450/security-of-cascaded-ciphers-including-question-on-counter-example-from-a-paper. – Дата звернення: 20.01.2025.

15. Stevens D. XOR Known-Plaintext Attacks. NVISO Blog. 2023. 12 жовт. [Електронний ресурс]. – URL: https://blog.nviso.eu/2023/10/12/xor-known-plaintext-attacks/. – Дата звернення: 20.01.2025.

16. Vulnerabilities of simple ciphers. FutureLearn / University of Leeds [Електронний ресурс]. – URL: https://www.futurelearn.com/info/courses/an-intro-to-cryptography/0/steps/413950. – Дата звернення: 23.01.2025.

17. Kalinda C. C# XOR (How it Works for Developers). IronPDF Blog. 2025. 29 трав. [Електронний ресурс]. – URL: https://ironpdf.com/blog/net-help/csharp-xor/. – Дата звернення: 23.01.2025.

18. mightyCoCo. Implementing XOR cipher encryption / decryption in C#. CodeProject. 2009. 18 бер. [Електронний ресурс]. – URL: https://www.codeproject.com/Articles/34232/Implementing-XOR-cipher-encryption-decryption-in-C. – Дата звернення: 23.01.2025.

19. Rafiq Z. A Comprehensive Guide to Secure Coding in C#. C# Corner [Електронний ресурс]. – URL: https://www.c-sharpcorner.com/article/a-comprehensive-guide-to-secure-coding-in-c-sharp/. – Дата звернення: 23.01.2025.

20. Rousos M. ASP.NET Core Performance Best Practices. Microsoft Learn. 2024. 27 вер. [Електронний ресурс]. – URL: https://learn.microsoft.com/en-us/aspnet/core/fundamentals/best-practices?view=aspnetcore-9.0. – Дата звернення: 24.01.2025.

21. Basatwar G. Code Obfuscation: A Comprehensive Guide Against Reverse-Engineering Attempts. Doverunner. 2025. 14 бер. [Електронний ресурс]. – URL: https://doverunner.com/blogs/code-obfuscation-guide-against-reverse-engineering-attempts/. – Дата звернення: 24.01.2025.

22. Best Data Masking Reviews 2025. Gartner Peer Insights. 2025 [Електронний ресурс]. – URL: https://www.gartner.com/reviews/market/data-masking. – Дата звернення: 24.01.2025.

23. Bowman K. Data Obfuscation Techniques: Securing Your Data Without Compromising Usability. Pathlock. 2025. 13 лют. [Електронний ресурс]. – URL: https://pathlock.com/learn/data-obfuscation/. – Дата звернення 25.01.2025.

24. Collberg C., Thomborson C., Low D. A Taxonomy of Obfuscating Transformations. Technical Report #148, Department of Computer Science, The University of Auckland [Електронний ресурс]. – URL: https://researchspace.auckland.ac.nz/bitstreams/990cb7de-2b44-4a24-acee-82b0efa8e689/download. – Дата звернення: 25.01.2025.

25. Data obfuscation. Talend [Електронний ресурс]. – URL: https://www.talend.com/resources/data-obfuscation/. – Дата звернення 27.01.2025.

26. ISO 27001 vs NIST Cybersecurity Framework. OneTrust Blog [Електронний ресурс]. – URL: https://www.onetrust.com/blog/iso-27001-vs-nist-cybersecurity-framework/. – Дата звернення: 27.01.2025.

ДОДАТКИ

Додаток А

Лістинг фрагменту коду контроллера HomeController.cs

```
using Microsoft.AspNetCore.Mvc;
using System.Text.Json;
using ScramblerWeb.Server.Models;
using Services.Keys;
using Services.Scramblers;
using System.Text;
namespace ScramblerWeb.Server.Controllers
{
    [Route("api/[controller]")]
    [ApiController]
    public class HomeController : Controller
    {
        private readonly Func<ScramblerType, IScrambler> _scramblerResolver;
        private readonly IKeyGenerator _keyGenerator;
        public HomeController(Func<ScramblerType, IScrambler> scramblerResolver,
IKeyGenerator keyGenerator)
        {
            _scramblerResolver = scramblerResolver;
            _keyGenerator = keyGenerator;
        }
        [HttpGet("generateKey")]
        public JsonResult GenerateKey(int length)
        {
            return Json(_keyGenerator.Generate(length));
        }
        private string BytesToHex(byte[] bytes)
        {
            return string.Concat(bytes.Select(b => b.ToString("x2")));
        }
        [HttpPost("scramble")]
        public JsonResult Scramble(ByteForm form)
        {
            var keyBytes = Encoding.UTF8.GetBytes(form.Key);
            byte[] result = form.Data.ToArray();
            foreach (var algorithm in form.Algorithms)
            {
                var scrambler = _scramblerResolver(algorithm);
                result = scrambler.Scramble(result, keyBytes);
            }

            return Json(BytesToHex(result));
        }
        [HttpPost("unscramble")]
        public JsonResult UnscrambleByte(ByteForm form)
        {
            var keyBytes = Encoding.UTF8.GetBytes(form.Key);
            byte[] result = form.Data.ToArray();

            form.Algorithms.Reverse();
            foreach (var algorithm in form.Algorithms)
```

Продовження додатку А

```
{
                var scrambler = _scramblerResolver(algorithm);
                result = scrambler.Descramble(result, keyBytes);
            }

            return Json(Convert.ToBase64String(result));
        }
        [HttpPost("scrambleFile")]
        public async Task<IActionResult> ScrambleFile(IFormFile file, [FromForm]
string key, [FromForm] string algorithms)
        {
            if (file == null || file.Length == 0)
            {
                return BadRequest("Файл не вибрано або він пустий.");
            }

            if (string.IsNullOrWhiteSpace(key))
            {
                return BadRequest("Ключ не може бути пустим.");
            }

            if (string.IsNullOrWhiteSpace(algorithms))
            {
                return BadRequest("Алгоритми не вибрані.");
            }

            List<ScramblerType> selectedAlgorithms =
JsonSerializer.Deserialize<List<ScramblerType>>(algorithms);

            using var memoryStream = new MemoryStream();
            await file.CopyToAsync(memoryStream);
            byte[] fileBytes = memoryStream.ToArray();

            var keyBytes = Encoding.UTF8.GetBytes(key);
            byte[] scrambledData = fileBytes;

            foreach (var algorithm in selectedAlgorithms)
            {
                var scrambler = _scramblerResolver(algorithm);
                scrambledData = scrambler.Scramble(scrambledData, keyBytes);
            }

            return File(scrambledData, "application/octet-stream", "scrambled_"
+ file.FileName);
        }
        [HttpPost("unscrambleFile")]
        public async Task<IActionResult> UnscrambleFile(IFormFile file,
[FromForm] string key, [FromForm] string algorithms)
        {
            try
            {
                if (file == null || file.Length == 0)
                    return BadRequest("Файл не вибрано або він пустий.");

                if (!file.FileName.StartsWith("scrambled_"))
                    return BadRequest("Файл не є заскрембльованим (очікується
префікс 'scrambled_').");

                if (string.IsNullOrWhiteSpace(key))
                    return BadRequest("Ключ не може бути пустим.");
```

Продовження додатку А

```
                if (string.IsNullOrWhiteSpace(algorithms))
                    return BadRequest("Алгоритми не вибрані.");

                List<ScramblerType> selectedAlgorithms =
JsonSerializer.Deserialize<List<ScramblerType>>(algorithms);


                using var memoryStream = new MemoryStream();
                await file.CopyToAsync(memoryStream);
                byte[] fileBytes = memoryStream.ToArray();

                var keyBytes = Encoding.UTF8.GetBytes(key);
                byte[] unscrambledData = fileBytes;

                selectedAlgorithms.Reverse();

                foreach (var algorithm in selectedAlgorithms)
                {
                    var scrambler = _scramblerResolver(algorithm);
                    unscrambledData = scrambler.Descramble(unscrambledData,
keyBytes);
                }

                string newFileName = "unscrambled_" +
file.FileName.Substring(10);

                return File(unscrambledData, "application/octet-stream",
newFileName);
            }
            catch (Exception ex)
            {
                return StatusCode(500, $"Внутрішня помилка сервера:
{ex.Message}");
            }
        }
    }
}
```

Додаток Б

Лістинг фрагменту коду модуля скреблювання XOR

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Services.Scramblers
{
    public class ScramblerXOR : IScrambler
    {
        public byte[] Scramble(byte[] data, byte[] key)
        {
            if (data == null)
            {
                throw new ArgumentNullException(nameof(data));
            }
            if (key == null || key.Length == 0)
            {
                return data;
            }

            byte[] scrambledData = new byte[data.Length];

            int keyIndex = 0;
            for (int i = 0; i < data.Length; i++)
            {
                scrambledData[i] = (byte)(data[i] ^ key[keyIndex]);
                keyIndex = (keyIndex + 1) % key.Length;
            }

            return scrambledData;
        }
        public byte[] Descramble(byte[] data, byte[] key)
        {
            return Scramble(data, key);
        }
```

Додаток В

Лістинг фрагменту коду модуля скремблювання перестановки блоків

```
namespace Services.Scramblers
{
    public class ScramblerBlockPermutation : IScrambler
    {
        public byte[] Scramble(byte[] data, byte[] key)
        {
            if (data == null || data.Length == 0) return data;

            int blockSize = Math.Min(64, data.Length);
            byte[] result = new byte[data.Length];
            int blocksCount = (int)Math.Ceiling((double)data.Length /
blockSize);

            for (int blockIndex = 0; blockIndex < blocksCount; blockIndex++)
            {
                int start = blockIndex * blockSize;
                int end = Math.Min(start + blockSize, data.Length);
                int blockLength = end - start;

                byte[] block = data.Skip(start).Take(blockLength).ToArray();
                byte[] permutationKey = GeneratePermutationKey(key,
blockLength);
                byte[] scrambledBlock = PermuteBlock(block, permutationKey);

                Array.Copy(scrambledBlock, 0, result, start, blockLength);
            }
            return result;
        }

        public byte[] Descramble(byte[] data, byte[] key)
        {
            if (data == null || data.Length == 0) return data;

            int blockSize = Math.Min(64, data.Length);
            byte[] result = new byte[data.Length];
            int blocksCount = (int)Math.Ceiling((double)data.Length /
blockSize);

            for (int blockIndex = 0; blockIndex < blocksCount; blockIndex++)
            {
                int start = blockIndex * blockSize;
                int end = Math.Min(start + blockSize, data.Length);
                int blockLength = end - start;

                byte[] block = data.Skip(start).Take(blockLength).ToArray();
                byte[] permutationKey = GeneratePermutationKey(key,
blockLength);
                byte[] descrambledBlock = PermuteBlock(block,
InversePermutationKey(permutationKey));

                Array.Copy(descrambledBlock, 0, result, start, blockLength);
            }
            return result;
        }

        private byte[] GeneratePermutationKey(byte[] key, int blockLength)
        {
            byte[] permutation = Enumerable.Range(0, blockLength).Select(i =>
(byte)i).ToArray();
```

Продовження додатку В

```
 int seed = key.Sum(b => b);
            Random rng = new Random(seed);

            for (int i = blockLength - 1; i > 0; i--)
            {
                int j = rng.Next(i + 1);
                (permutation[i], permutation[j]) = (permutation[j],
permutation[i]);
            }

            return permutation;
        }

        private byte[] PermuteBlock(byte[] block, byte[] permutationKey)
        {
            byte[] scrambledBlock = new byte[block.Length];
            for (int i = 0; i < block.Length; i++)
            {
                scrambledBlock[i] = block[permutationKey[i]];
            }
            return scrambledBlock;
        }

        private byte[] InversePermutationKey(byte[] permutationKey)
        {
            int length = permutationKey.Length;
            byte[] inverseKey = new byte[length];

            for (int i = 0; i < length; i++)
            {
                inverseKey[permutationKey[i]] = (byte)i;
            }

            return inverseKey;
        }
    }
}
```

Додаток Г

Лістинг фрагменту коду модуля скремблювання адитивним перетворенням

```
namespace Services.Scramblers
{
    public class ScramblerCaesar : IScrambler
    {
        public byte[] Scramble(byte[] data, byte[] key)
        {
            if (data == null || data.Length == 0) return data;

            byte[] result = new byte[data.Length];

            for (int i = 0; i < data.Length; i++)
            {
                int keyVal = key[i % key.Length] % 256;
                result[i] = (byte)((data[i] + keyVal) % 256);
            }
            return result;
        }

        public byte[] Descramble(byte[] data, byte[] key)
        {
            if (data == null || data.Length == 0) return data;

            byte[] result = new byte[data.Length];

            for (int i = 0; i < data.Length; i++)
            {
                int keyVal = key[i % key.Length] % 256;
                result[i] = (byte)((data[i] - keyVal + 256) % 256);
            }
            return result;
        }

    }
}
```

Додаток Д

Лістинг фрагменту коду модуля генерації ключів

```
using System.CodeDom.Compiler;
using System.Security.Cryptography;

namespace Services.Keys
{
    public class KeyGenerator : IKeyGenerator
    {
        public string Generate(int length)
        {
            if (length <= 0)
            {
                throw new ArgumentOutOfRangeException(nameof(length), "Length
must be a positive number.");
            }

            byte[] randomBytes = new byte[length];
            using (RandomNumberGenerator rng = RandomNumberGenerator.Create())
            {
                rng.GetBytes(randomBytes);
            }

            return string.Concat(randomBytes.Select(b => b.ToString("x2")));
        }
    }
}
```

Додаток Е

Лістинг фрагменту коду клієнтської частини ByteMode.vue

```
<template>
  <div class="container">
    <div class="container">
      <h3>Вибір алгоритму скремблювання</h3>
      <div class="algorithm-buttons">
        <button v-for="algorithm in algorithms"
                :key="algorithm.name"
                :style="{ backgroundColor: algorithm.color }"
                @click="addAlgorithm(algorithm)">
          {{ algorithm.label }}
        </button>
      </div>
      <div class="selected-algorithms">
        <p>Обрані алгоритми:</p>
        <div class="horizontal-list">
          <div v-for="(algorithm, index) in selectedAlgorithms"
               :key="index"
               class="algorithm-item"
               draggable="true"
               :class="{ dragging: dragIndex === index }"
               @dragstart="dragStart(index)"
               @dragover.prevent
               @drop="drop(index)"
               :style="{ backgroundColor: algorithm.color }">
            {{ algorithm.label }}
            <button @click="removeAlgorithm(index)">X</button>
          </div>
        </div>
      </div>

    </div>
    <div class="row">
      <input type="text" v-model="byteForm.key" placeholder="Введіть ключ">
      <button @click="generateButton">Згенерувати</button>
    </div>
    <div class="textarea-container">
      <textarea v-model="byteForm.data" placeholder="Вхідні дані у HEX(12 ab d5
50)"></textarea>
      <div class="buttons">
        <button @click="scrambleButton">Заскремблювати</button>
        <button @click="unscrambleButton">Розскремблювати</button>
      </div>
      <textarea v-model="outputData" placeholder="Вихідні дані"></textarea>
    </div>
  </div>
</template>

<script>
  import axios from 'axios';
  export default {
    name: "app",
    components: {},
    data() {
      return {
        algorithms: [
          { name: "0", label: "XOR", color: "#ff9999" },
          { name: "1", label: "Перестановка блоків", color: "#99ccff" },
          { name: "2", label: "Адитивне перетворення", color: "#99ff99" },
```

Продовження додатку Е

```
  ],
        selectedAlgorithms: [],
        dragIndex: null,
        outputData: "",
        byteForm: {
          key: "",
          data: "",
          algorithms: []
        }
      };
    },
    methods: {
      hexToBytes(hex) {
        const bytes = [];
        const sanitizedHex = hex.replace(/\s/g, '');
        for (let i = 0; i < sanitizedHex.length; i += 2) {
          bytes.push(parseInt(sanitizedHex.substring(i, i + 2), 16));
        }
        return bytes;
      },
      bytesToHex(base64Data) {
        try {
          const byteString = atob(base64Data);
          const bytes = Array.from(byteString, (char) => char.charCodeAt(0));
          let hex = '';
          for (let byte of bytes) {
            hex += ('0' + (byte & 0xFF).toString(16)).slice(-2) + ' ';
          }
          return hex.trim();
        } catch (error) {
          console.log("Can't convert base64 to hex: " + error);
          return base64Data;
        }
      },
      addSpacesToHex(hexString) {
        if (!hexString) return "";
        let result = "";
        for (let i = 0; i < hexString.length; i += 2) {
          result += hexString.substring(i, Math.min(i + 2, hexString.length));
          if (i + 2 < hexString.length) {
            result += " ";
          }
        }

        return result.trim();
      },
      async generateButton() {
        try {
          const response = await
axios.get("https://localhost:7168/api/Home/generateKey?length=32");
          this.byteForm.key = response.data;
          console.log('Generated key:', this.byteForm.key)
        } catch (error) {
          console.error('Error generating key:', error);
        }
      },
      async scrambleButton() {
        try {
          const algorithmsToSend = this.selectedAlgorithms.map(algo =>
parseInt(algo.name));
```

Продовження додатку Е

```
   const textToBytes = (text) => new TextEncoder().encode(text);

          const formData = {
            key: this.byteForm.key,
            data: this.hexToBytes(this.byteForm.data),
            algorithms: algorithmsToSend
          };
          console.log(formData);
          const response = await
axios.post("https://localhost:7168/api/Home/scramble", formData, {
            headers: {
              "Content-Type": "application/json"
            }
          });

          console.log("Response:", response.data);
          this.outputData = this.addSpacesToHex(response.data);
        } catch (error) {
          console.error("Error submitting form:", error);
        }
      },
      async unscrambleButton() {
        try {
          const algorithmsToSend = this.selectedAlgorithms.map(algo =>
parseInt(algo.name));
          const textToBytes = (text) => new TextEncoder().encode(text);

          const formData = {
            key: this.byteForm.key,
            data: this.hexToBytes(this.byteForm.data),
            algorithms: algorithmsToSend
          };
          console.log(formData);
          const response = await
axios.post("https://localhost:7168/api/Home/unscramble", formData, {
            headers: {
              "Content-Type": "application/json"
            }
          });

          console.log("Response:", response.data);
          this.outputData = this.bytesToHex(response.data);
        } catch (error) {
          console.error("Error submitting form:", error);
        }
      },
      addAlgorithm(algorithm) {
        this.selectedAlgorithms.push({
          ...algorithm,
          color: this.darkenColor(algorithm.color, 20),
        });
      },
      removeAlgorithm(index) {
        this.selectedAlgorithms.splice(index, 1);
      },
      dragStart(index) {
        this.dragIndex = index;
      },
      drop(index) {
        const draggedItem = this.selectedAlgorithms[this.dragIndex];
        this.selectedAlgorithms.splice(this.dragIndex, 1);
```

Продовження додатку Е

```
 this.selectedAlgorithms.splice(index, 0, draggedItem);
        this.dragIndex = null;
      },
      darkenColor(color, percent) {
        const num = parseInt(color.slice(1), 16);
        const amt = Math.round(2.55 * percent);
        const R = (num >> 16) - amt;
        const G = ((num >> 8) & 0x00ff) - amt;
        const B = (num & 0x0000ff) - amt;
        return `#${(0x1000000 + (R < 0 ? 0 : R > 255 ? 255 : R) * 0x10000 + (G <
0 ? 0 : G > 255 ? 255 : G) * 0x100 + (B < 0 ? 0 : B > 255 ? 255 : B))
          .toString(16)
          .slice(1)}`;
      },
    },
  };
</script>
```

Додаток Ж

Результати дослідження часових характеристик алгоритмів скремблювання

Таблиця Ж.1

Результати дослідження часових характеристик алгоритму XOR

Обсяг даних (мегабайт)	Скремблювання			Дескремблювання		
	Розмір ключа (біт)					
	8	32	256	8	32	256
1	3,95мс	3,95мс	3,15мс	3,70 мс	3,15мс	3,10мс
10	37,00мс	37,00мс	36,80мс	36,40мс	36,60мс	37,00мс
100	394,60мс	394,60мс	393,40мс	395,80мс	373,40мс	392,20мс

Таблиця Ж.2

Результати дослідження часових характеристик алгоритму перестановки блоків

Обсяг даних (мегабайт)	Скремблювання			Дескремблювання		
	Розмір ключа (біт)					
	8	32	256	8	32	256
1	28,00мс	26,10мс	42,10мс	32,00мс	29,00мс	44,60мс
10	341,66мс	318,66мс	397,00мс	363,00мс	397,00мс	407,00мс
100	2876,00мс	2795,00мс	3619,00мс	3261,00мс	3097,00мс	3955,00мс

Таблиця Ж.3

Результати дослідження часових характеристик алгоритму адитивного перетворення

Обсяг даних (мегабайт)	Скремблювання			Дескремблювання		
	Розмір ключа (біт)					
	8	32	256	8	32	256
1	4,00мс	4,00мс	4,05мс	4,00мс	4,00мс	4,05мс
10	44,60мс	43,60мс	44,00мс	46,00мс	45,60мс	45,00мс
100	441,00мс	453,50мс	474,00мс	456,00мс	468,00мс	475,50мс