МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Західноукраїнський національний університет
Факультет комп'ютерних інформаційних технологій
Кафедра кібербезпеки

ЦЕЗАРУК Софія Юріївна

Метод шифрування даних на основі коригуючих кодів / Data Encryption Method Based on Error-Correcting Codes

спеціальність: 125 – Кібербезпека

освітньо-професійна програма – Кібербезпека

Кваліфікаційна робота

Виконала студентка групи
КБзм -21
С.Ю. Цезарук

Науковий керівник
д.т.н., професор В.В.Яцків

Кваліфікаційну роботу
допущено до захисту:

«____» _____ ___2023 р.

Завідувач кафедри
_____ **В.В.Яцків**

ТЕРНОПІЛЬ – 2023

Факультет комп'ютерних інформаційних технологій
Кафедра кібербезпеки
Освітній ступінь «магістр»
спеціальність: 125 – Кібербезпека
освітньо-професійна програма – Кібербезпека

ЗАТВЕРДЖУЮ
Завідувач кафедри

_____ В.В.Яцків
«____» _____ 2022 року

ЗАВДАННЯ

НА КВАЛІФІКАЦІЙНУ РОБОТУ СТУДЕНТУ

ЦЕЗАРУК СОФІЯ ЮРІЇВНА
(прізвище, ім'я, по батькові)

1. Тема кваліфікаційної роботи:
Метод шифрування даних на основі коригуючих кодів / Data Encryption Method Based on Error-Correcting Codes
керівник роботи д.т.н., професор В.В. Яцків

затверджені наказом по університету від 1 грудня 2022 року № 491

2. Строк подання студентом закінченої кваліфікаційної роботи 1 грудня 2023 р.

3. Вихідні дані до кваліфікаційної роботи: завдання на кваліфікаційну роботу студента, наукові статті, технічна література.

4. Основні питання, які потрібно розробити:
– визначити принципи роботи та класифікацію коригуючих кодів;
– провести аналіз методів шифрування даних на основі коригуючих кодів;
– дослідити метод шифрування коду Хеммінга;
– провести аналіз роботи існуючого алгоритму та можливі методи вдосконалення;
– розробити структуру та алгоритм вдосконалення методу;
– реалізувати вдосконалення коду Хеммінга програмним шляхом.

5. Перелік графічного матеріалу у роботі:
– таблиці;
– блок-схеми;
– рисунки.

6. Консультанти розділів кваліфікаційної роботи

	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв

7. Дата видачі завдання 8 грудня 2022 р.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів кваліфікаційної роботи	Строки виконання етапів кваліфікаційної роботи	Примітка
1	Аналіз аспектів області дослідження: Коригуючі коди та методи шифрування	12.2022 р. – 03.2023 р.	
2	Метод шифрування даних на основі коригуючих кодів	03.2023 р. – 05.2023 р.	
3	Оцінка ефективності та тестування методу	05.2023 р. – 11.2023 р.	

Студент _____ Цезарук С.Ю.
(підпис)

Керівник роботи _____ д.т.н., професор В.В. Яцків
(підпис)

АНОТАЦІЯ

Магістерська робота на тему “Метод шифрування даних на основі коригуючих кодів” зі спеціальності 125 – кібербезпека написана обсягом 94 сторінки і містить 13 ілюстрацій, 9 таблиць та 30 джерел за переліком посилань.

Метою роботи є підвищення надійності методу шифрування даних на основі коригуючих кодів та розробка алгоритму асинхронних потоків коду Хеммінга, для покращення швидкодії та ефективності кодування.

Методи досліджень базуються на використанні коригуючих кодів, кодів Хеммінга, окрім того було використано методи статистичного аналізу для обробки результатів експериментів та методи.

Проаналізовано принципи роботи коригуючих кодів та методів шифрування, основи їх реалізації та використання, фактори вибору коригуючих кодів, методи кодування та декодування, аналіз однобітових помилок, шляхи вдосконалення асинхронними потоками коду Хеммінга для збільшення продуктивності кодування та декодування даних, особливо для великих обсягів даних, покращення масштабованості кодування та декодування даних, збільшення гнучкості кодування та декодування даних.

Розроблено вдосконалений метод шифрування даних на основі коду Хеммінга з використанням асинхронних потоків, а саме асинхронні потоки дозволяють виконувати кодування та декодування даних на декількох процесорах або ядрах одночасно, що значно підвищує продуктивність кодування та декодування даних для великих обсягів даних. Це вирішує питання масштабності, а також пришвидшує швидкодію кодування. Реалізовано код Хеммінга та код Хеммінга з асинхронними потоками на мові програмування Python

Результати роботи можуть бути використані в системах передачі даних по мережі, таких як CDN (Content Delivery Network) або VPN (Virtual Private Network), системах зберігання даних, таких як NAS (Network Attached Storage) або SAN (Storage Area Network), розроблений метод шифрування даних може бути використаний для підвищення продуктивності зберігання даних, особливо

для великих обсягів даних, також в системах обробки даних в реальному часі, таких як системи відеоспостереження або системи медичної діагностики, розроблений метод шифрування даних може бути використано для підвищення ефективності обробки даних, особливо для великих обсягів даних. Даний метод шифрування даних є перспективним рішенням для захисту даних в широкому спектрі застосунків.

ключові слова: ШИФРУВАННЯ, ДЕШИФРУВАННЯ, ХЕММІНГ, АСИНХРОННИЙ МЕТОД, КОРИГУЮЧІ КОДИ.

RESUME

The graduate work on the topic «Data Encryption Method Based on Error-Correcting Codes» for Master's degree on speciality 125 "Cybersecurity" is written on 94 pages and contains 13 illustrations, 9 tables and 30 references.

The aim of the work is to improve the reliability of the data encryption method based on error corrective codes and to develop an algorithm for asynchronous Hamming code streams to improve the speed and efficiency of coding.

The research methods are based on the use of error corrective codes, Hamming codes, in addition, statistical analysis methods were used to process the results of experiments and methods.

The principles of operation of error corrective codes and encryption methods, the basics of their implementation and use, factors for choosing error corrective codes, encoding and decoding methods, single-bit error analysis, ways to improve the Hamming code with asynchronous streams to increase the performance of data encoding and decoding, especially for large amounts of data, improve the scalability of data encoding and decoding, and increase the flexibility of data encoding and decoding are analyzed.

An improved method of data encryption based on the Hamming code using asynchronous threads has been developed, namely, asynchronous threads allow data encoding and decoding to be performed on multiple processors or cores simultaneously, which significantly increases the performance of data encoding and decoding for large amounts of data. This solves the issue of scalability and speeds up the encoding performance. Implemented Hamming code and Hamming code with asynchronous threads in Python programming language.

The results of the work can be used in data transmission systems over the network, such as CDN (Content Delivery Network) or VPN (Virtual Private Network), data storage systems, such as NAS (Network Attached Storage) or SAN (Storage Area Network), the developed data encryption method can be used to improve the performance of data storage, especially for large amount of data, as well as in real-time data processing systems, such as video surveillance systems or

medical diagnostic systems, the developed data encryption method can be used to improve the efficiency of processing. This data encryption method is a promising solution for data protection in a wide range of applications.

Keywords: ENCRYPTION, DECRYPTION, HAMMING, ASYNCHRONOUS METHOD, ERROR CORRECTIVE CODES.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ СКОРОЧЕНЬ

FEC - Forward Error Correction.

LDPC - Low-density parity-check code.

RAID - Redundant Array of Independent Disks.

ECC - Error-correcting codes.

HDD - Hard Disk Drive.

SSD - Solid-state drive.

CRC - Cyclic Redundancy Code.

RSA - Rivest-Shamir-Adleman.

XOR - eXclusive OR (виключне або).

RAM - Random Access Memory.

SSL - Secure Sockets Layer.

ВСТУП

Актуальність теми дослідження. У сучасному світі дані є цінним ресурсом, який потребує захисту від несанкціонованого доступу, зміни та знищення. Одним із способів забезпечення безпеки даних є їх шифрування. Шифрування - це процес перетворення даних у нечитабельний формат, який називається шифром. Щоб отримати доступ до даних, одержувач повинен використовувати ключ для розшифрування. Існує безліч різних методів шифрування. Одним із перспективних напрямків розвитку криптографії є використання коригуючих кодів. Коригуючі коди - це тип кодування, який використовується для виявлення та виправлення помилок, які можуть виникнути під час передачі даних.

Мета і завдання дослідження. Метою даного дослідження є вдосконалення та оцінка ефективності методу шифрування даних на основі коригуючих кодів.

Для досягнення поставлених цілей необхідно вирішити такі завдання:

1. Провести аналіз предметної області: коригуючі коди та методи шифрування.

2. Описати вибраний метод шифрування даних на основі коригуючих кодів.

3. Обґрунтувати метод шифрування даних, обраний на основі коригуючих кодів.

4. Розробити алгоритм для вдосконалення методу шифрування даних на основі коригуючих кодів.

5. Оцінити ефективність розробленого методу.

Об’єкт дослідження – процеси шифрування даних на основі коректуючих кодів.

Предмет дослідження – алгоритми шифрування даних на основі коректуючих кодів.

Методи досліджень. Для розв’язання поставлених задач у даній кваліфікаційній роботі використано методи шифрування, методи

завадостійкого кодування, окрім того було використано методи статистичного аналізу для обробки результатів експериментів.

Наукова новизна дослідження полягає в удосконаленні процесу шифрування даних на основі коригуючих кодів, що забезпечує наступні переваги:

1. Покращена безпека даних. Цей метод дозволяє виправити помилки, які можуть виникнути під час передачі даних, навіть якщо деякі дані змінено або знищено.

2. Зменшена пропускна здатність. Цей метод може зменшити обсяг даних, що надсилаються, що може зменшити пропускну здатність каналу зв'язку.

Практична значущість дослідження. Результати дослідження можуть бути використані для розробки нових методів захисту даних від несанкціонованого доступу.

Публікації та апробація до кваліфікаційної роботи.

1. Цезарук С.Ю. Криптографічні протоколи безпеки мережі: SSL, TLS, IPSEC. Матеріали проблемно-наукової міжгалузевої конференції «Автоматизація та комп'ютерно-інтегровані технології» (АКІТ - 2022), Тернопіль, 2022. – С.101–105

2. Цезарук С.Ю., Рогачова А.Є., Самойлов С.В. Роль інформаційно-психологічних операцій в сьогоденні: проблематика, виклики та заходи захисту. Кібербезпека в Україні: правові та організаційні питання: матеріали міжн. наук. практ. конф., м. Одеса, 17 листопада 2023 р. Одеса : ОДУВС, 2023. С. 113-114.

РОЗДІЛ 1. АНАЛІЗ АСПЕКТІВ ОБЛАСТІ ДОСЛІДЖЕННЯ: КОРИГУЮЧІ КОДИ ТА МЕТОДИ ШИФРУВАННЯ

1.1. Визначення та класифікація коригуючих кодів

У сучасному світі дані є одним із найцінніших активів. Вони використовуються в багатьох різних галузях, включаючи бізнес, науку, медицину та державну службу. Важливо забезпечити надійність і безпеку даних, щоб захистити дані від несанкціонованого доступу, втрати або пошкодження.

Одним із способів забезпечення надійності та безпеки даних є використання коригуючих кодів та методів шифрування.

Коригуючі коди - це тип кодування інформації, який дозволяє відновити передане повідомлення, навіть якщо деякі його символи були пошкоджені під час передачі. Коригуючі коди роблять це, додаючи до повідомлення надмірність, тобто додаткові символи, які не містять інформації, але використовуються для виявлення та виправлення помилок.

Кожен коригуючий код характеризується рядом показників, таких як:

1. Довжина - це загальна кількість символів у кодовому слові.

2. Кількість інформаційних символів - це кількість символів у кодовому слові, які несуть інформацію.

3. Кількість надлишкових символів - це кількість символів у кодовому слові, які не містять інформації, а використовуються для виявлення та виправлення помилок.

4. Повне число - це кількість різних кодових слів, які можна створити за допомогою даного коду.

5. Мінімальна кодова відстань - це мінімальна відстань між двома різними кодовими словами.

Коригуючі коди можна класифікувати за різними ознаками.

За типом надлишкових символів коригуючі коди поділяються на:

1. Згорткові коди - надлишковість додається шляхом додавання кількох символів до кожної групи символів.

2. Розподільчі коди - надмірність додається шляхом розподілу її по всьому кодовому слову.

Залежно від типу алгоритму для виявлення та виправлення помилок, коригуючі коди поділяються на:

1. Алгоритми з перевіркою парності - помилки виявляються шляхом перевірки парності окремих символів або груп символів.

2. Алгоритми з перевіркою циклічності - помилки виявляються шляхом перевірки періодичності кодового слова.

3. Алгоритми з перевіркою лінійності - помилки виявляються шляхом перевірки лінійності кодового слова.

Метод шифрування даних на основі коригуючих кодів передбачає додавання надмірності до даних, які потрібно зашифрувати, які потім використовуються для виявлення та виправлення помилок під час передачі.

Щоб зашифрувати дані за допомогою коригуючих кодів необхідно виконати кроки представлені на рисунку 1.1.

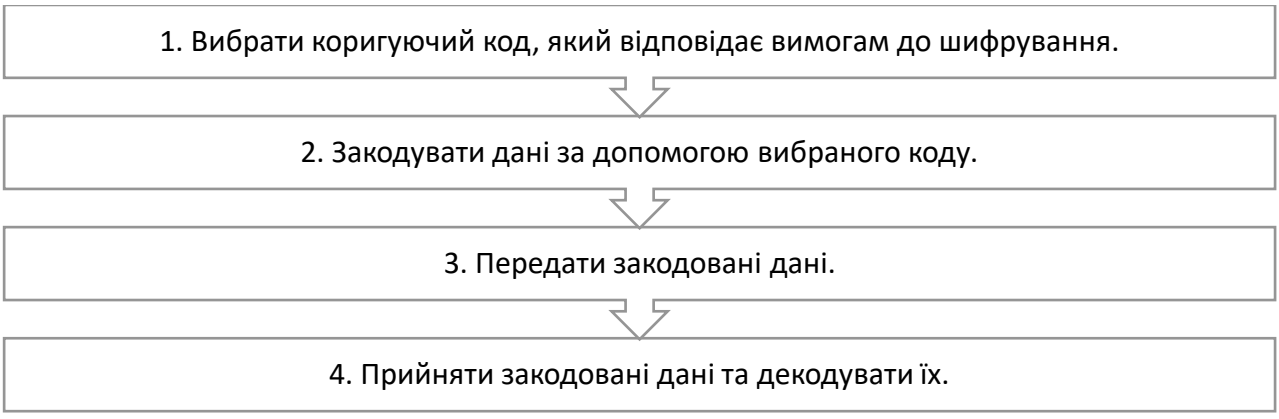

Рисунок 1.1 – Схема кроків шифрування даних за допомогою коригуючих кодів

Метод шифрування даних на основі коригуючих кодів має ряд переваг:

1. Він забезпечує високий рівень безпеки даних, оскільки помилки, що виникли під час передачі, можуть бути виявлені та виправлені.

2. Це досить ефективно, оскільки цей метод не вимагає значних додаткових обчислювальних витрат.

3. Він може використовуватися для різних типів даних.

Недоліки методу шифрування даних на основі коригуючих кодів:

1. Збільшує розмір зашифрованих даних, коли до них додається надлишковість.

2. Він може бути менш ефективним, ніж інші методи шифрування, якщо помилки, що виникають під час передачі, є дуже частими.

3. Він вимагає використання додаткових біт для кодування даних.

Метод шифрування даних на основі коригуючих кодів можна використовувати в різних областях, зокрема:

1. У телекомунікаціях для забезпечення надійної передачі даних.

2. У комп'ютерних системах для захисту даних від несанкціонованого доступу, втрати або пошкодження.

3. У системах зберігання даних для підвищення надійності зберігання.

В даний час проводяться ведуться з метою підвищення ефективності методів шифрування даних на основі коригуючих кодів. Зокрема, розробляються нові коригуючі коди, які дозволяють зменшити розмір даних, що шифруються, без істотного зниження рівня безпеки.

Важливим критерієм для класифікації коригуючих кодів є метод виявлення та виправлення помилок. За цим критерієм коригуючі коди поділяються на наступні типи:

1. Двійкові коди - це коди, здатні виявляти та виправляти помилки в бінарних даних.

2. Кодування з контролем над помилками (ECC) - це тип коду, який використовує контрольні суми для виявлення помилок.

3. Кодування з репарацією помилок (FEC) - це тип коду, який використовує додаткові біти для виправлення помилок.

Коригувальні коди також можна класифікувати за способом реалізації. За цим критерієм коригувальні коди поділяються на наступні типи:

1. Кодування з лінійним передбачуванням (LPCC) - це тип коду реалізований за допомогою лінійного передбачення.

2. Кодування з блоковим передбаченням (BPCC) - це тип коду, який реалізується за допомогою блокового прогнозування.

Метод шифрування даних на основі коригуючих кодів використовує коригувальні коди для забезпечення конфіденційності та цілісності даних. Цей метод зазвичай працює так:

1. Дані шифруються за допомогою шифру.

2. Зашифровані дані шифруються за допомогою коригуючого коду.

3. Дані надсилаються одержувачу в зашифрованому форматі.

Одержувач отримує зашифровані дані та використовує коригуючий код для їх розшифрування. Якщо в процесі передачі даних виникають помилки, коригуючий код може їх виправити.

Також метод шифрування даних на основі коригуючих кодів використовується в таких протоколах зв'язку, як IPsec та TLS. Він також використовується в таких системах зберігання даних, як RAID.

1.2. Принципи роботи та приклади використання коригуючих кодів

Коригуючі коди - це тип кодування інформації, який використовується для забезпечення точності передавання інформації в умовах впливу зовнішніх шумів. Вони працюють шляхом додавання додаткової інформації до передаваної інформації, яка називається контрольними кодами.

Контрольні коди - це дані, які додаються до інформації для виявлення та виправлення помилок. Вони зазвичай обчислюються шляхом додавання деяких операцій над інформацією, наприклад, додавання, множення або логічних операцій. Принцип роботи коригуючих кодів можна пояснити на наступному прикладі. Нехай у нас є повідомлення, яке складається з чотирьох біт: 1010. Ми можемо додати до нього три контрольні коди, які обчислюються шляхом додавання кожного біта повідомлення до себе та до себе зсунутої на одиницю. Це дає нам наступне кодове слово: 10111010. Тепер, якщо під час передачі

кодового слова виникне помилка, наприклад, буде змінено один біт, то це буде відображено в контрольних кодах. Наприклад, якщо буде змінено другий біт кодового слова з 1 на 0, то отримаємо наступне слово: 10011010.

Алгоритм декодування може використовувати контрольні коди для виявлення та виправлення помилок. Для цього алгоритм спочатку обчислює контрольні коди для отриманого кодового слова. Якщо контрольні коди не збігаються з контрольними кодами, розрахованим для вихідного повідомлення, алгоритм визначає, що в кодовому слові є помилка.

Для виправлення помилки можна використати наступний алгоритм:

1. Знайти біт, відмінний від контрольних кодів.
2. Перевернути значення цього біта.

У цьому прикладі алгоритм визначає наявність помилки в другому біті кодового слова. Алгоритм інвертує значення другого біта з 0 на 1, даючи нам таке кодове слово: 10111011. Це кодове слово відповідає оригінальному повідомленню 1010.

Ефективність коригуючих кодів визначається їх здатністю виявляти та виправляти помилки. Коригуючі коди можна класифікувати відповідно до їхніх можливостей виправлення помилок.

Блок-коди можуть виправити лише обмежену кількість помилок. Наприклад, код Хеммінга може виправити лише одну помилку.

Рядні коди можуть виправити більшу кількість помилок. Наприклад, код Ріда-Соломона може виправити до 16 помилок. Вибір типу коригуючого коду залежить від багатьох факторів, включаючи кількість помилок, які потрібно виправити, і ймовірність виникнення цих помилок. Коригуючі коди є важливим інструментом для забезпечення точності переданої інформації. Вони використовуються в різноманітних програмах, де важливо, щоб інформація передавалась без помилок. Ось кілька прикладів того, як коригуючі коди можна використовувати в реальному світі:

1. У цифровому зв'язку коригуючі коди використовуються для забезпечення надійної передачі даних по мережі. Наприклад, вони використовуються в

мережах стільникового зв'язку, де вони допомагають запобігти втраті даних під час передачі даних між телефонами.

2. У комп'ютерах коригуючі коди використовуються для забезпечення точності даних, що зберігаються в пам'яті. Наприклад, вони використовуються в контролерах пам'яті, де вони допомагають запобігти помилкам у пам'яті, які можуть спричинити збоїв системи.

3. У космосі коригуючі коди використовуються для забезпечення надійного передавання даних між космічними апаратами. Наприклад, вони використовуються в космічних кораблях, де вони допомагають запобігти втраті даних під час передачі даних між космічним кораблем і Землею.

Існує багато різних типів коригуючих кодів, які використовують різні алгоритми кодування та декодування. Деякі з найпоширеніших типів коригуючих кодів включають:

1. Геометричні коди - це прості коригуючі коди, які використовують прості геометричні операції, такі як додавання та множення. Наприклад, один із простих геометричних кодів називається кодом Хеммінга. Код Хеммінга додає до інформації три контрольні коди, обчислені шляхом додавання інформації до себе та зміщення її на одиницю.

2. Риманові коди - це більш складні коригуючі коди, які використовують більш складні математичні операції, такі як поліноміальні рівняння. Наприклад, один із поширених рімановых кодів називається кодом Ріда-Соломона. Код Ріда-Соломона додає до інформації контрольні коди, які обчислюються шляхом розв’язання поліноміального рівняння.

3. Базисний код є ще більш складним коригуючим кодом, який використовує комбінацію геометричного та рімановового кодів. Наприклад, один із поширених базових кодів називається кодом БЧК. Код БЧК додає до інформації контрольні коди, які обчислюються шляхом додавання інформації до себе та до себе зсунутої на одиницю, а також шляхом розв’язуванням поліноміального рівняння.

Вибір типу коригуючого коду залежить від багатьох факторів, включаючи кількість помилок, які необхідно виправити, і ймовірність виникнення цих помилок. Основні принципи роботи коригуючих кодів включають:

1. Додавання зайвих бітів (кодування). Додавання додаткових бітів до інформаційного блоку, щоб створити кодове слово.
2. Створення чек-сум або кодових слів. Використання додаткових бітів, щоб створити контрольну суму або кодове слово, яке відображає інформацію про вміст блоку даних.
3. Виявлення помилок (декодування). Коли кодове слово отримано, виконується аналіз на наявність помилок.
4. Виправлення помилок (якщо можливо). Якщо виявлені помилки, за можливості, використовувати додаткові біти для виправлення помилок. Цей процес може використовувати різні математичні або логічні методи.
5. Передача або збереження надлишкової інформації. Зберігання або передача інформації за допомогою кодових слів або додаткових бітів, щоб можна було виявити та виправити помилки.
6. Стійкість до помилок. Розробка кодів, які можуть виявляти та виправляти помилки при великій кількості завдань, щоб забезпечити високий рівень надійності передачі чи зберігання даних.

До прикладів реального застосування коригуючих кодів (ECC, Error-correcting codes) відноситься:

a. Цифровий зв’язок, а саме:
    a. Мобільний зв'язок. ECC використовується в мобільних мережах, таких як GSM і LTE, для виявлення та виправлення помилок у даних, що передаються бездротовим способом. Це допомагає підтримувати якість дзвінків, запобігає втраті даних і забезпечує безперебійний зв’язок.
    b. Космічна комунікація. В системах космічного зв'язку ECC є ключовим фактором для забезпечення цілісності даних під час передачі на великі відстані в суворих умовах космосу. Вони захищають від перешкод

сигналу, шуму та інших помилок, які можуть призвести до втрати даних.

b. Пам’ять комп’ютера:
    a. Жорсткі диски (HDD). ECC відіграє важливу роль у жорстких дисках, захищаючи від пошкодження даних, спричиненої помилками читання/запису, механічними вібраціями та іншими фізичними факторами. Це забезпечує точність збережених даних і запобігає пошкодженню файлів.
    b. Твердотільні накопичувачі (SSD). SSD використовують ECC для захисту цілісності даних, особливо від зносу та помилок у комірках пам’яті NAND-типу. Це забезпечує надійність даних під час тривалого використання та запобігає втраті інформації.
c. Космічні системи:
    a. Космічна комунікація. ECC необхідний для надійного зв’язку між космічними апаратами та наземними станціями управління. Вони захищають дані від пошокджень, спричинених космічним випромінюванням, перешкодами сигналу та іншими викликами у космічному середовищі.
    b. Зберігання даних на космічних апаратах. ECC використовується в системах зберігання даних на космічних апаратах, забезпечуючи цілісність критичних наукових даних, журналів польотів та параметрів керування. Це запобігає втраті даних та забезпечує точність інформації, отриманої з космічних місій.
d. Архівація даних. ECC використовується в системах довгострокового зберігання даних, наприклад у системах, якими керують бібліотеки, музеї та дослідницькі інститути. Вони захищають дані від погіршення з часом, забезпечуючи збереження цінної інформації для майбутніх поколінь.
    e. Оптичні диски. ECC вбудовані у CD, DVD та Blu-ray диски для захисту даних від подряпин, пилу та інших фізичних несправностей,

які можуть спричинити помилки читання. Це забезпечує надійне відтворення мультимедійного вмісту та запобігає втраті даних.

Ці приклади ілюструють, що ECC використовуються в широкому спектрі застосувань, від повсякденних технологій до складних космічних систем. Вони є невід’ємною частиною підтримки цілісності даних, забезпечення надійного зв’язку та збереження цінної інформації. Коригуючі коди - це клас технік контролю помилок, які дозволяють виявляти та виправляти помилки в даних під час їх передачі або зберігання. ECC широко використовуються в різних цифрових системах, включаючи мережі зв'язку, пристрої зберігання комп'ютерів та космічні застосування.

Основним принципом ECC є додавання додаткової інформації до вихідних даних, яка використовується для виявлення та виправлення помилок. Ця додаткова інформація, яку часто називають бітом парності або контрольним бітом, обчислюється на основі певних математичних правил і вбудовується в зашифровані дані. Під час отримання або зчитування зашифрованих даних біти парності витягуються та аналізуються, щоб знайти будь-які помилки, які можуть спричинити невідповідність даних.

Існують два основних типи ECC:

1. Коди виявлення помилок - ці коди можуть лише ідентифікувати наявність помилок в даних, але не можуть точно визначити місце чи виправити помилки. До прикладів входять контрольні суми та циклічні резидуальні контролі (CRC).

2. Коди коригування помилок - ці коди не лише виявляють помилки, але також можуть їх виправляти. Це досягається за допомогою більш вдосконалених схем кодування, які дозволяють ідентифікувати та виправити обмежену кількість помилок. До загальновживаних кодів коригування помилок входять коди Хеммінга, Ріда-Соломона та BCH-коди (рис. 1.2).

Рисунок 1.2 - Фактори вибору коригуючих кодів

ЕСС працюють за наступним алгоритмом:

1. Кодування. Вихідні дані кодуються за допомогою ЕСС, додаючи додаткові біти до даних.

2. Передача або зберігання. Кодовані дані передаються або зберігаються.

3. Виявлення та коригування помилок. При отриманні або відновленні даних додаткові біти вилучаються та аналізуються. Якщо виявлено помилки, ЕСС намагаються визначити та виправити помилкові біти в потоці даних.

4. Декодування. Виправлені дані декодуються для отримання вихідної інформації.

Приклад застосування коригуючих кодів, як код Хеммінга, один із типів ЕСС, використовується для кодування та декодування даних:

Код Хеммінга - це тип ЕСС, здатний виявити до 1 помилки в 7-ми бітних даних.

1. Кодування. Оригінальні 7-ми бітні дані кодуються шляхом додавання 3 додаткових бітів, які називаються контрольними бітами. Контрольні біти обчислюються з оригінальних даних за допомогою математичних правил.

2. Передача або зберігання. Кодовані дані, що складаються з 10 біт, передаються або зберігаються.

3. Виявлення та коригування помилок. Під час отримання або відновленні кодованих даних додаткові біти вилучаються та аналізуються. У разі виявлення помилок, ECC може визначити її точне місце розташування. Помилка виправляється шляхом зміни відповідного біта в кодованих даних. Виправлені дані декодуються для отримання оригінальної інформації.

ECC є важливим інструментом для забезпечення цілісності передачі та зберігання даних. Вони використовуються в багатьох надзвичайно важливих сферах застосування.

1.3. Визначення та класифікація методів шифрування

Метод шифрування - це алгоритм, який використовується для перетворення даних таким чином, щоб їх було неможливо прочитати без знання секретного ключа. Шифрування використовується для захисту конфіденційності даних, таких як паролі, фінансова інформація та особисті повідомлення.

Існує два основних типи методів шифрування:

1. Симетричні шифри використовують той самий ключ для шифрування та розшифрування даних. Цей ключ повинен бути відомий, як відправнику так і одержувачу. Симетричні шифри швидші, ніж асиметричні, але вони менш безпечні, оскільки ключ повинен бути переданий безпечним способом.

2. Асиметричні шифри використовують два ключі: один для шифрування, а інший для розшифрування. Ключем, який називається відкритим, можна поділитися з будь-ким. Інший ключ, який називається закритим, повинен зберігатися в секреті. Асиметричні шифри є більш безпечними, ніж симетричні, оскільки відкритий ключ не може бути використаний для розшифрування даних.

Симетричні шифри використовують різні методи для перетворення даних. Одним із найпоширеніших методів є шифр простої заміни, при якому

кожен біт даних замінюється іншим бітом за допомогою таблиці підстановки. Іншим поширеним методом є перестановка шифрування, при якому бітові послідовності переставляються в певному порядку.

Шифр простої заміни (Substitution cipher) - це найпростіший тип шифрування, при якому кожен біт даних замінюється іншим бітом за допомогою таблиці підстановки (табл. 1.1).

Таблиця 1.1.

Таблиця підстановки зміненого шифрування.

Початковий символ	А	Б	В	Г	Д	Е	Ж	З	И	Й	К	Л	М	Н	О	П	Р	С	Т	У	Ф	Х	Ц	Ч	Ш	Щ	Ь	Ю
Замінений символ	Б	В	Г	Д	Е	Ж	З	И	Й	К	Л	М	Н	О	П	Р	С	Т	У	Ф	Х	Ц	Ч	Ш	Щ	Ь	Ю	Я

При застосуванні цієї таблиці підстановки літера "А" буде замінена літерою "Б", літера "Б" буде замінена літерою "В" і так далі.

Перестановка шифрування (Transposition cipher) - це тип шифрування, при якому бітові послідовності переставляються в певному порядку. Наприклад, можна переставляти бітові послідовності в зворотному порядку, або можна переставляти їх у довільному порядку.

Асиметричні шифри можна класифікувати відповідно до типу алгоритму, який вона використовує для перетворення даних. Найпоширенішими типами асиметричного шифрування є:

1. Еліптичні криві використовують еліптичні криві для генерації ключів і шифрування даних.

2. RSA (Rivest-Shamir-Adleman) використовує складні математичні розрахунки для створення ключів і шифрування даних.

3. Diffie-Hellman використовує асиметричні шифри для обміну секретним ключем.

Шифрування можна використовувати для різних цілей, наприклад:

1. Захист конфіденційності: шифрування використовується для захисту даних, таких як паролі, фінансова інформація та особисті повідомлення. Наприклад, веб-сайти використовують шифрування для захисту інформації, яку користувачі вводять у поля форми, як-от паролі та номери кредитних карток.

2. Захист цілісності: шифрування можна використовувати для захисту цілісності даних, гарантуючи, що дані не змінюються під час передачі чи зберігання. Наприклад, шифрування використовується для захисту даних на носіях інформації, таких як флеш-накопичувачі та жорсткі диски.

3. Захист автентифікації: шифрування можна використовувати для захисту автентифікації, гарантуючи, що дані надсилаються або отримуються з дійсного джерела. Наприклад, шифрування використовується для захисту цифрових підписів.

Вибір методу шифрування залежить від таких факторів, як:

1. Необхідний рівень безпеки даних. Складніші методи шифрування часто забезпечують вищий рівень безпеки, але вони також можуть бути повільнішими та складнішими для впровадження.

2. Швидкість, необхідна для шифрування. Симетричне шифрування зазвичай швидше, ніж асиметричне шифрування, але воно також менш безпечне.

3. Обсяг зашифрованих даних. Асиметричне шифрування менш ефективне при шифруванні великих обсягів даних.

Методи шифрування можна класифікувати за різними ознаками, зокрема:

1. За типом використовуваного ключа:

   a. Симетричні шифри використовують один і той самий ключ для шифрування і дешифрування даних.

   b. Асиметричні шифри використовують два ключі: один для шифрування та один для розшифрування.

2. За принципом дії:

   a. Замінно-перестановочні шифри засновані на заміні та перестановці бітів або байтів вихідних даних.

b. Складні шифри використовують складніші математичні алгоритми для перетворення даних.

3. За рівнем безпеки:

a. Шифри з низькою безпекою забезпечують низький рівень захисту даних і можуть бути зламані за допомогою простих методів.

b. Шифри із середньою безпекою забезпечують середній рівень захисту даних і можуть бути зламані за допомогою складніших методів.

c. Шифри з високою безпекою забезпечують високий рівень захисту даних і дуже складно зламати.

4. За сферою застосування:

a. Криптографія безпеки мережі використовується для захисту даних, що передаються через мережу, наприклад Інтернет.

b. Коди захисту даних на пристрої використовуються для захисту даних, що зберігаються на пристрої, наприклад на комп'ютері чи мобільному телефоні.

c. Криптографія використовується для захисту даних, що зберігаються в пам'яті, наприклад RAM або флеш-пам'яті.

Симетричні шифри є найбільш простими та ефективними методами шифрування. Вони використовують один і той самий ключ для шифрування і дешифрування даних, що робить їх швидшими, ніж асиметричні шифри. Однак симетричні шифри менш безпечні, ніж асиметричні, оскільки ключ потрібно передати шифрувальнику й одержувачу шифрованих даних.

Асиметричні шифри забезпечують вищий рівень безпеки, ніж симетричні, оскільки відкритий ключ не може бути використаний для розшифрування шифрованих даних. Це означає, що відкритий ключ може бути переданий будь-кому, а закритий ключ повинен зберігатися в секреті. Однак асиметричні шифри повільніші, ніж симетричні, і вимагають більше обчислювальних ресурсів.

Замінно-перестановочні шифри засновані на заміні та перестановці бітів або байтів вихідних даних. Підстановки можна виконувати за допомогою

таблиць підстановки та перестановок шляхом зміни порядку бітів або байтів. Переставні шифри є простими та ефективними методами шифрування, але вони мають відносно низький рівень безпеки.

Складні шифри використовують складніші математичні алгоритми для перетворення даних. Ці алгоритми часто базуються на теорії чисел або теорії ймовірностей. Складні шифри забезпечують вищий рівень безпеки, ніж шифри з перестановкою, але вони також більш складні та вимагають більше обчислювальних ресурсів.

Шифрування для захисту даних у мережі використовується для захисту даних, що передаються через мережу, наприклад Інтернет. Ці засоби шифрування забезпечують безпеку даних, запобігаючи несанкціонованому доступу. Шифрування для захисту даних в Інтернеті використовується в різних програмах, включаючи електронну пошту, веб-безпеку та віддалений доступ.

Шифрування на пристрої використовується для захисту даних, що зберігаються на пристрої, наприклад комп'ютері чи мобільному телефоні. Ці засоби шифрування запобігають несанкціонованому доступу до даних, що зберігаються на пристрої. Шифрування на пристрої використовується на різних пристроях, включаючи комп'ютери, мобільні телефони, планшети та розумні годинники.

Шифрування в пам'яті використовується для захисту даних, що зберігаються в пам'яті, наприклад ОЗУ або флеш-пам'яті. Ці засоби шифрування запобігають несанкціонованому доступу до даних, що зберігаються в пам'яті, коли пристрій вимкнено або не використовується. Шифрування для захисту даних у пам'яті використовується на різних пристроях, включаючи комп'ютери, мобільні телефони, планшети та розумні годинники.

1.4. Принципи роботи та приклади використання методів шифрування

Принципи роботи методів шифрування даних визначаються конкретним типом шифрування, що використовується. Однак, загалом, головна мета будь-

якого методу шифрування — захистити інформацію від несанкціонованого доступу та забезпечити конфіденційність, цілісність і, у багатьох випадках, автентичність даних. Розглянемо загальні кроки, які можуть бути присутніми в багатьох методах шифрування:

1. Визначення мети. Щоб ефективно вибрати метод шифрування, вам потрібно точно розуміти, чого ви намагаєтеся досягти. Наприклад, чи потрібно вам захистити дані від несанкціонованого доступу, забезпечити безпеку під час передачі чи забезпечити цілісність і автентичність?
2. Вибір типу шифрування. Вибір між симетричним і асиметричним шифруванням залежить від ваших потреб і ситуації. Симетричне шифрування використовує один ключ для обох операцій, тоді як асиметричне шифрування використовує пару ключів: відкритий і закритий.
3. Генерація ключів. Вирішення способу створення та обміну ключами є важливою частиною використання шифрування. У разі симетричного шифрування безпечний обмін ключами може бути складнішим, ніж у випадку асиметричного шифрування.
4. Шифрування даних. На цьому кроці вихідні дані перетворюються на криптограму за допомогою вибраного алгоритму та ключа шифрування. У разі симетричного шифрування для шифрування та дешифрування використовується один і той же ключ, тоді як у випадку асиметричного шифрування для обох операцій використовуються різні ключі.
5. Передача/зберігання даних. Зашифровані дані передаються по каналу зв'язку або зберігаються на запам'ятовуючому пристрої. Важливо забезпечити безпеку ключа під час обміну (для симетричного шифрування) або забезпечити надійність інформації у відкритому ключі (для асиметричного шифрування).
6. Розшифрування. Отримані зашифровані дані розшифровуються за допомогою відповідного ключа та алгоритму. У симетричному шифруванні той самий ключ використовується для розшифрування, тоді

як у випадку асиметричного шифрування використовується приватний ключ.

7. Перевірка цілісності та автентичності (при потребі). У деяких випадках важливо перевірити, чи не було змінено дані під час передачі. Для цього можна використовувати додаткові методи, такі як коди автентифікації або цифрові підписи.

Метою методів шифрування є забезпечення захисту інформації від несанкціонованого доступу або зловживання. Далі розглянемо деякі додаткові аспекти теми, включаючи хешування, автентифікацію та методи підпису.

1. Хешування. Основний принцип - Хеш-функції перетворюють вхідні дані будь-якого розміру у фіксований вихід (хеш-код).

   Властивості: Зміни вхідних даних вносять суттєві зміни у хеш-коді.

   Застосування: Хешування використовується для перевірки цілісності даних, зберігання паролів, тощо.
2. Безпека. Спроба встановлення входу за його хеш-кодом (колізія) повинна бути вкрай складною (колізійна стійкість).
3. Автентифікація. Основний принцип - перевірка ідентичності користувача або системи. Використання включає в себе введення паролів, біометричну автентифікацію та інші методи.
4. Цифровий підпис. Основний принцип - використовується для автентифікації та підтвердження непорушеності даних за допомогою криптографічного підпису. Застосування - забезпечує автентичність інформації та визначає автора.
5. Використання в комплексі полягає в комбінуванні методів та застосуванні SSL/TLS протоколів.

Комбінування Методів. Багато систем використовують комбінацію симетричного та асиметричного шифрування для забезпечення ефективності та безпеки. Автентифікацію та хешування часто використовують разом для повного захисту інформації.

SSL/TLS протокол. Використовується для шифрування даних під час передачі через Інтернет. Застосовує як симетричне, так і асиметричне шифрування для забезпечення конфіденційності та автентифікації.

Забезпечення безпеки даних є складним завданням, яке вимагає використання багатьох різних методів і технологій. Ефективне використання шифрування та інших методів безпеки дозволяє захистити ваші дані від загроз і небажаного доступу. Однак важливо регулярно оновлювати та тестувати системи безпеки, щоб переконатися в їх ефективності.

З огляду на сучасні тенденції та розширення сфери застосування шифрування, важливо враховувати деякі додаткові аспекти:

a. Мультифакторна автентифікація (MFA). Використання кількох методів автентифікації, таких як пароль, біометрика, токен чи смарт-карта. Підвищує рівень безпеки, тому що навіть якщо один елемент зламано, інші елементи залишаються в безпеці.

b. Post-Quantum cryptography. Розвиток криптографічних алгоритмів, стійких до атак квантовими комп'ютерами. У контексті швидкого розвитку квантових технологій важливою стала розробка заходів для забезпечення стабільності сучасних криптографічних систем.

c. Blockchain та шифрування. Використання криптографії для забезпечення конфіденційності та недоступності даних в блокчейні. Гарантує непорушеність блоків та транзакцій, а також визначає права доступу.

d. Захист від Соціально Інженерних Атак. Покращення усвідомленості користувачів щодо ризиків соціально інженерних атак.

e. Шифрування в Інтернеті Речей (IoT). Використання шифрування для захисту даних, передаваних між пристроями IoT. Забезпечує конфіденційності та недоступності інформації в мережах IoT.

f. Захист від Кібератак. Застосування шифрування для захисту даних від несанкціонованого доступу під час кібератак.

У стародавні та середні віки люди використовували криптографію, щоб певну інформацію могли знайти лише ті, хто отримав секретне повідомлення.

Сьогодні, через століття, суть цього процесу залишається незмінною. Ми можемо знайти способи шифрування для захисту інформації скрізь – смартфони, розумні годинники та інші пристрої, комп'ютери та маршрутизатори, планшети та смарт-телевізори, побутова техніка, месенджери та соціальні мережі, програмне забезпечення для форексу та торгівлі тощо. Усі дані, що зберігаються та/або передаються за допомогою цих пристроїв, завжди зашифровані (рис. 1.3).

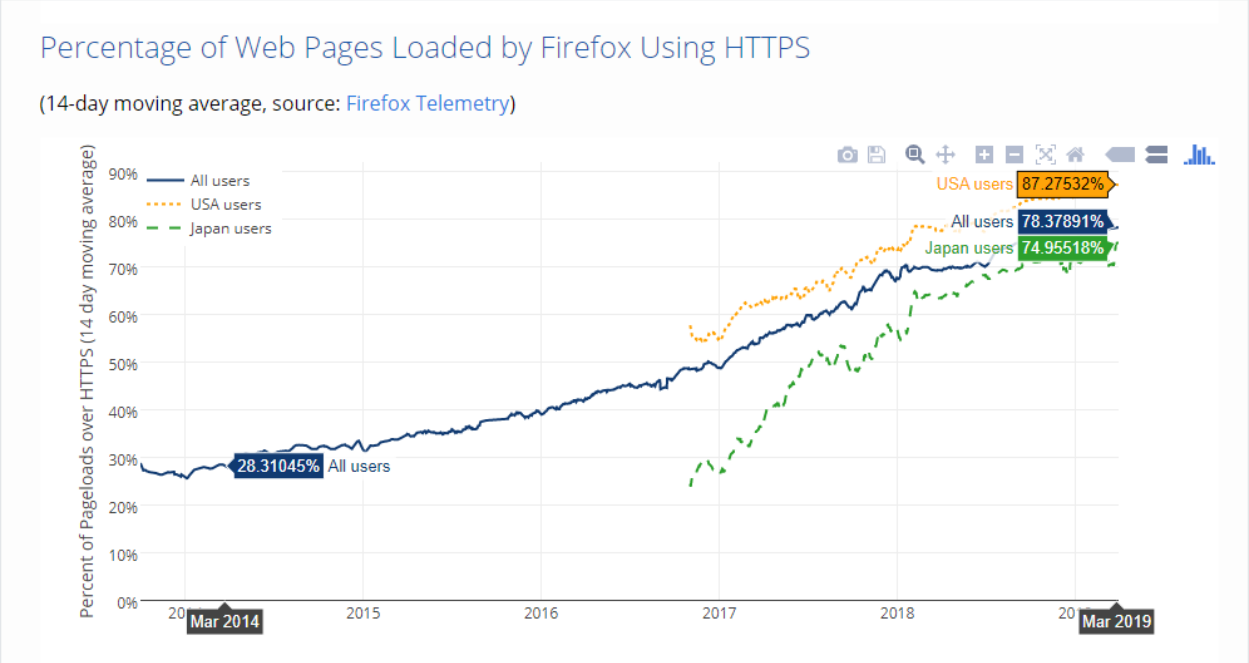

Рисунок 1.3 - Статистика переходів на сайт з протоколу http на https

Шифрування особливо корисне під час фінансових операцій, наприклад, через NetBanking або PayPal та інші міжнародні платіжні системи, а також під час зняття готівки в банкоматах або під час роботи з платіжними терміналами, під час здійснення покупок у точках продажу або під час торгівлі на іноземних обмінний ринок. Додамо сюди жорстку конкуренцію між мобільними операційними системами, криптозахист дозволить ефективніше захищати інформацію користувача.

Найпопулярніші криптовалюти останнього десятиліття також покладаються на криптографічні алгоритми.

Інтернет-трафік також вимагає криптографічного захисту, а криптографічне розширення HTTPS забезпечує безпеку даних, що передаються через протокол Secure Socket Layer (SSL) або Transport Layer Security (TLS) у підключенні сайту клієнта. Кілька років тому Google наголошував на використанні сертифікатів https і SSL, пояснюючи, що сайти https мають вищі рейтинги в пошукових системах. Шифрування також є важливим інструментом для захисту даних від несанкціонованого доступу. Він використовується в багатьох різних сферах для захисту конфіденційної інформації, такої як паролі, фінансова інформація та особисті повідомлення.

1. Інтернет-банкінг та онлайн-транзакції. Мета використання шифрування SSL/TLS - забезпечення безпеки та цілісності особистої та фінансової інформації клієнтів під час передачі між клієнтами та серверами банку.

2. Електронна пошта. Метод захисту використовує протоколи шифрування (наприклад, PGP, S/MIME) для захисту вмісту електронної пошти та вкладень. Мета захисту - забезпечення конфіденційності листування та запобігання витоку інформації.

3. Захищені VPN-з'єднання. Метод захисту - використовуйте VPN (віртуальну приватну мережу) для шифрування з’єднання між вашим комп’ютером і мережею в Інтернеті. Ціль безпеки забезпечення конфіденційності та безпеки даних під час передачі через відкриті мережі.

4. Захищені месенджери. Метод захисту - використовуйте наскрізне шифрування в месенджерах, таких як Signal або WhatsApp, щоб захистити приватні чати та обмін файлами. Мета захисту - забезпечити конфіденційність комунікацій і запобігти витоку інформації.

5. Шифрування даних на зберіганні. Метод захисту використовує алгоритми шифрування для захисту даних на накопичувачах, таких як жорсткі диски та флеш-накопичувачі. Мета захисту - запобігання несанкціонованому доступу до конфіденційної інформації у разі втрати або викрадення пристрою.

6. Медичні записи та eHealth. Метод захисту - використовуйте шифрування для захисту особистих медичних даних пацієнта, які передаються

та зберігаються в електронних системах охорони здоров’я. Ціль захисту - забезпечення конфіденційності та цілісності записів медичної інформації.

7. Шифрування файлів у хмарних сервісах. Метод захисту - використовує алгоритми шифрування для захисту файлів, що зберігаються в хмарних службах, під час передачі та зберігання. Мета захисту - забезпечення безпеки та конфіденційності даних в онлайн-середовищі.

8. Криптовалюта та блокчейн. Метод захисту - використання криптографії для забезпечення безпеки та автентифікації транзакцій у блокчейні. Ціль захисту - забезпечення цілісності та безпеки фінансових операцій у децентралізованих системах.

9. Захист корпоративних мереж. Метод захисту - використовуйте VPN, шифрування трафіку та інші методи захисту корпоративних мереж від несанкціонованого доступу. Ціль захисту - забезпечення безпеки інформації компанії та запобігання витоку даних.

РОЗДІЛ 2. МЕТОД ШИФРУВАННЯ ДАНИХ НА ОСНОВІ КОРИГУЮЧИХ КОДІВ

2.1. Опис методу шифрування даних на основі коригуючих кодів Хеммінга

Коди Хеммінга - це сімейство лінійних кодів корекції помилок. Вони були винайдені Річардом В. Хеммінгом у 1950 році і є одними з найпростіших і найефективніших кодів виправлення помилок. Коди Хеммінга можуть виявляти одно- або двобітові помилки та виправляти однобітові помилки. Це вдвічі більше помилок, ніж може виявити простий код парності, і вдвічі більше помилок, ніж може виправити простий код парності.

Коди Хеммінга є ідеальними кодами, тобто вони досягають максимальної швидкості, можливої для кодів з мінімальною довжиною блоку та інтервалом у три. Це означає, що вони можуть забезпечити надійну передачу даних у середовищі з низьким рівнем шуму. Коди Хеммінга широко використовуються в багатьох галузях, включаючи телекомунікації, комп'ютерні системи та зберігання даних. Вони використовуються для забезпечення надійної передачі даних у таких системах, як телефонні мережі, мережі передачі даних і системи зберігання інформації. Коди Хеммінга використовуються в ситуаціях, коли узгодженість даних важливіша за ефективність передачі. Наприклад, коди Хеммінга часто використовуються в комп'ютерній пам'яті з виправленням помилок, щоб захистити дані від помилок, які можуть виникнути через радіацію або космічні промені. Коди Хеммінга також використовуються в супутниковому та космічному зв'язку, де великі відстані та довгий час передачі даних роблять помилки більш імовірними.

У теорії інформації відстань Хеммінга між двома ланцюжками однакової довжини — це кількість місць, де відповідні символи відрізняються. Іншими словами, відстань Хеммінга вимірює мінімальну кількість замін, необхідних для перетворення одного рядка в інший, або мінімальну кількість помилок, які можуть перетворити один рядок в інший. Відстань Хеммінга — одна з кількох

метрик рядків, які використовуються для вимірювання відстані редагування між двома рядками. Відстань Хеммінга двох слів дорівнює 0 тоді і тільки тоді, коли ці два слова ідентичні. Для двійкових рядків a і b відстань Хеммінга дорівнює кількості одиниць a XOR b.

Основним застосуванням відстані Хеммінга є теорія кодування, особливо в блокових шифрах, де рядки однакової довжини є векторами над кінцевим полем. Блок коду відстані n-Хеммінга може виявити не більше n-1 помилок і виправити не більше n-2 помилок. Наприклад: відстань Хеммінга між двома рядками 000000 і 000111 дорівнює 3. Це означає, що потрібно змінити 3 символи, щоб перетворити один рядок в інший. Граф Хеммінга — це граф, вершини якого відповідають усім можливим послідовностям заданої довжини. Ребра між двома вершинами з'єднані лініями з відстанню Хеммінга, що дорівнює 1.

Наприклад у наступних всіх вершинах відстань Хеммінга буде дорівнювати трьом:

- 000000
- 000111
- 011001
- 011110
- 101010
- 101101
- 110011
- 110100

Граф Хеммінга зображено на рисунку 2.1.

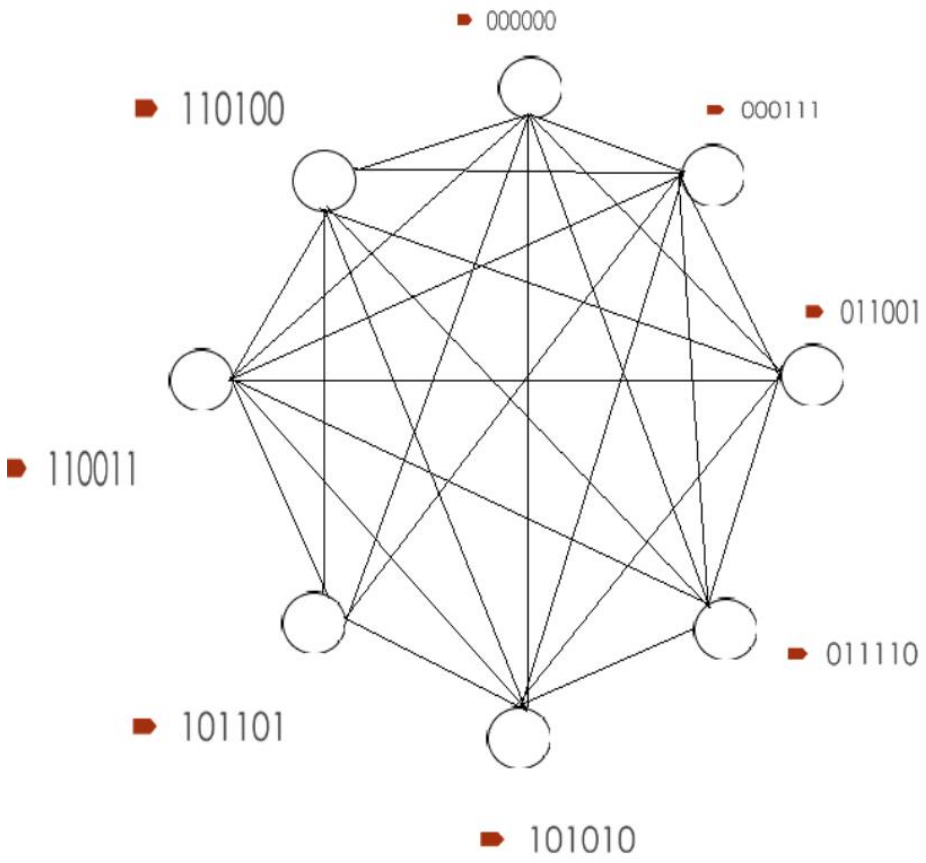

Рисунок 2.1 – Граф Хеммінга

Ефективність передачі кодів Хеммінга зростає зі збільшенням розміру блоку даних. Наприклад, для коду Хеммінга (7, 4) ефективна швидкість передачі даних становить лише 0,571, тоді як для коду Хеммінга (255, 247) вона становить 0,969. Код Хеммінга відносно простий у використанні і може бути реалізований апаратно. Це також означає, що він має високу швидкість обчислень. Ці властивості роблять коди Хеммінга ідеальними для використання в пам’яті ECC. Щоб проілюструвати значення ECC RAM, уявіть, що сервер банківської бази даних записує депозит у розмірі 100 гривень. Як 8-розрядне ціле число ця сума буде збережена як двійкове число 01100100. Якщо космічне проміння змінить перший біт, це змінить суму депозиту на 228 гривень або 11100100. Пам’ять ECC автоматично виявить і виправить цю помилку.

2.2. Обґрунтування вибору методу шифрування даних на основі коригуючих кодів Хеммінга

У роботі розглядається метод шифрування даних на основі коригуючих кодів Хеммінга. Код Хеммінга використовуються в ситуаціях, коли узгодженість даних важливіша за ефективність передачі. Наприклад, код Хеммінга часто використовуються в комп'ютерній пам'яті з виправленням помилок, щоб захистити дані від помилок, які можуть виникнути через радіацію або космічні промені. Код Хеммінга також використовуються в супутниковому та космічному зв'язку, де великі відстані та довгий час передачі даних роблять помилки більш імовірними. Цей метод був обраний з наступних причин:

1. Ефективність для виявлення та виправлення однобітових помилок. Код Хеммінга може виявляти та виправляти однобітові помилки з імовірністю 1. Це означає, що якщо в зашифрованому повідомленні виникла однобітова помилка, то вона буде виявлена та виправлена. Для порівняння, більш складні коди, такі як коди Боуза-Чоудрі, можуть виявляти та виправляти багатобітові помилки, але вони не можуть виявляти та виправляти однобітові помилки з такою ж ефективністю, як код Хеммінга.
2. Простота реалізації. Код Хеммінга відносно простий у реалізації як з точки зору апаратного, так і програмного забезпечення. Це робить його доступним для широкого застосування. Коди Хеммінга можуть бути реалізовані як апаратно, так і програмно. Апаратна реалізація коду Хеммінга може бути виконана за допомогою логічних елементів або мікросхем. Програмні реалізації коду Хеммінга можуть бути реалізовані на таких мовах програмування, як C, C++, C# або Python. Простота реалізації коду Хеммінга робить його привабливим для застосування в системах з обмеженими ресурсами, таких як мікроконтролери або вбудовані системи.

3. Швидкість обчислень. Код Хеммінга є відносно швидким для обчислень. Це важливо для систем, де швидкість передачі даних є критичною. Розрахунок коду Хеммінга можна виконати за час, пропорційний квадратному кореню з довжини блоку даних. Це означає, що коди Хеммінга можуть бути обчислені відносно швидко для блоків даних будь-якого розміру. Швидкість, з якою обчислюється код Хеммінга, робить його привабливим для використання в системах, де швидкість передачі даних є важливою, наприклад, супутниковий зв'язок або мережі передачі даних.
4. Використання в сучасних системах передачі даних. Коди Хеммінга широко використовуються в сучасних системах передачі даних, таких як мережевий зв'язок, зберігання даних і системи зберігання та передачі цифрової інформації. Їх успішний досвід застосування в різних сферах діяльності заснований на їх надійності та ефективності.
5. Можливість розширення для захисту від більшої кількості помилок. У деяких випадках коди Хеммінга можна розширити для виявлення та виправлення більшої кількості помилок, що робить їх привабливими, коли потрібна висока надійність передачі даних.

Недоліки методу шифрування даних на основі коригуючих кодів Хеммінга:

1. Нездатність виявити або виправити багатобітові помилки. Коди Хеммінга можуть виявляти лише однобітові помилки. Якщо в даних є багатобітова помилка, код Хеммінга не може виявити або виправити помилку.
2. Накладні витрати на додавання контрольних бітів. Щоб коди Хеммінга виявляли та виправляли однобітові помилки, до даних додаються контрольні біти. Це збільшує розмір зашифрованого блоку даних і знижує ефективність передачі.

Можливі шляхи вирішення цих недоліків:

1. Розробити нові коди, здатні виявляти та виправляти багатобітові помилки.

2. Використання більш ефективних кодів, які мають меншу надмірність.
3. Розробити нові, простіші та ефективніші методи реалізації коду Хеммінга.

Ось деякі конкретні приклади того, як недоліки методу шифрування даних на основі коригуючих кодів Хеммінга можна виправити за допомогою асинхронних потоків:

1. Щоб підвищити здатність до виявлення та виправлення помилок, можна використовувати асинхронні потоки для паралельного обчислення синдромів. Це дозволить скоротити час виявлення та виправлення помилок.
2. Щоб зменшити надмірність, можна використовувати асинхронні потоки для паралельного кодування даних. Це дозволить скоротити час кодування даних, що може призвести до зниження надмірності.
3. Щоб спростити реалізацію, можна використовувати асинхронні потоки для розподілу обчислювальних завдань між різними процесорами або ядрами. Це дозволить спростити реалізацію кодів Хеммінга, особливо для апаратної реалізації.

Підвищення здатності до виявлення та виправлення помилок. Для виявлення та виправлення помилок за допомогою кодів Хеммінга необхідно розрахувати синдроми. Синдроми - це числа, які характеризують помилки в зашифрованому повідомленні. Обчислення синдромів можна виконати за допомогою синхронного алгоритму. Цей алгоритм виконує обчислення послідовно, один за одним. За допомогою асинхронних потоків можна виконати обчислення синдромів паралельно. Це дозволить скоротити час виявлення та виправлення помилок.

Для цього можна скористатися наступним алгоритмом:

1. Розділити шифроване повідомлення на частини.
2. Створіти асинхронний потік для кожної частини зашифрованого повідомлення.

3. У кожному асинхронному потоці обчислити синдроми для відповідної частини зашифрованого повідомлення.
4. Після завершення всіх асинхронних потоків об’єднайти синхросигнали з різних частин закодованого повідомлення.

Зменшення надмірності. Щоб зменшити надмірність, можна використовувати асинхронні потоки для паралельного кодування даних. Кодування даних за допомогою коду Хеммінга можна виконати за допомогою синхронного алгоритму. Цей алгоритм виконує обчислення послідовно, один за одним. Використовуючи асинхронні потоки, можна виконувати шифрування даних паралельно. Це зменшить час кодування даних, що може призвести до зменшення надмірності.

Для цього можна використовувати наступний алгоритм:

1. Розділити дані на кілька частин.
2. Для кожної частини даних створити асинхронний потік.
3. У кожному асинхронному потоці закодувати відповідну частину даних.
4. Після того, як всі асинхронні потоки завершаться, об'єднати закодовані дані з різних частин.

Спрощення реалізації. Щоб спростити реалізацію коду Хеммінга, можна використовувати асинхронні потоки для розподілу обчислювальних завдань між різними процесорами або ядрами. Це спростить реалізацію коду Хеммінга, особливо апаратну реалізацію.

Для цього можна використовувати наступний алгоритм:

1. Розподілити обчислювальні завдання для кодування даних між різними процесорами або ядрами.
2. За допомогою асинхронних потоків виконати ці обчислювальні завдання на різних процесорах або ядрах.

Асинхронні потоки можна використовувати для усунення деяких недоліків методів кодування даних на основі коду виправлення Хеммінга. Асинхронні потоки можуть покращити виявлення та виправлення помилок, зменшити надмірність і спростити реалізацію.

2.3. Алгоритм роботи методу шифрування даних на основі коригуючих кодів Хеммінга

Код Хеммінга — це схема прямої корекції помилок (FEC), яка може використовуватися для виявлення та виправлення бітових помилок. Біти корекції помилок відомі як біти Хеммінга, а число, яке потрібно додати до символу даних, визначається виразом:

$$2^{\text{П}} \geq (m + n + 1)$$

де m – кількість бітів у символі даних, а n – кількість бітів Хеммінга.

Наприклад, якщо в нас є 4 біти (m=4). Тоді для виправлення нам потрібно n=3 (оскільки 8 більше або дорівнює 4+3+1).

Біти Хеммінга вставляються в символ повідомлення за бажанням. Як правило, вони додаються в місцях, які є степенями числа 2, тобто в позиціях 1-го, 2-го, 4-го, 8-го, 16-го бітів і так далі. Наприклад, щоб закодувати символ 011001, тоді, починаючи з правого боку, біти Хеммінга будуть вставлені в позиції 1-го, 2-го, 4-го та 8-го бітів.

Приклад виконання (табл. 2.1-2.4):

Символ: 011001 , Біти Хеммінга: НННН.

Формат повідомлення буде наступним: 01Н100Н1НН.

Таблиця 2.1

Приклад резалізації коду Хеммінга

Позиція	Код
9	1001
7	0111
3	0011
XOR	1101

Таблиця 2.2

Розміщення контрольних символів в комбінаціях коду Хемінга

10	9	8	7	6	5	4	3	2	1
0	1	1	1	0	0	1	1	0	1

Таблиця 2.3

Без помилки

Позиція	Код
Хеммінг	1101
9	1001
7	0111
3	0011
XOR	0000

Таблиця 2.4

Помилка в 5 біті

Позиція	Код
Хеммінг	1101
9	1001
7	0111
5	0101
3	0011
XOR	0101

Побудова кодів Хеммінга заснована на принципі перевірки на парність числа одиничних символів: до послідовності додається такий елемент, щоб число одиничних символів у послідовності, що вийшла, було парним:

Відправка.

$$P_1 = D_1 \oplus D_2 \oplus D_4$$
$$P_2 = D_1 \oplus D_3 \oplus D_4$$
$$P_3 = D_2 \oplus D_3 \oplus D_4$$

Знак $\oplus$ тут означає додавання по модулю 2 .

$$\begin{array}{ccccccc} 111 & 110 & 101 & \mathbf{1}00 & 011 & 0\mathbf{1}0 & 00\mathbf{1} \\ D_4 & D_3 & D_2 & P_3 & D_1 & P_2 & P_1 \end{array}$$

Отримання.

Перше перевірне рівняння складають як суму за mod 2 всіх розрядів, в номерах яких в молодшому розряді 2^0 стоїть одиниця:

$$S_1 = P_1 \oplus D_1 \oplus D_2 \oplus D_4$$

Друге перевірне рівняння складають як суму за mod 2 всіх розрядів, в номерах яких стоїть одиниця на другому місці відповідного двійкового еквівалента (2^1):

$$S_2 = P_2 \oplus D_1 \oplus D_3 \oplus D_4$$

Третє перевірне рівняння складають як суму за mod 2 всіх розрядів, в номерах яких стоїть одиниця на третьому місці (2^2)

$$S_3 = P_3 \oplus D_2 \oplus D_3 \oplus D_4$$

Якщо S=0 - то помилки немає, якщо S=1- то одноразова помилка.
Далі отримуємо векторну матрицю

$$T = [P_1 \quad P_2 \quad D_1 \quad P_3 \quad D_2 \quad D_3 \quad D_4]$$

Після цього транспонуємо її:

$$\mathrm{T}^{\mathrm{T}} = \begin{bmatrix} P_1 \\ P_2 \\ D_1 \\ P_3 \\ D_2 \\ D_3 \\ D_4 \end{bmatrix}$$

Отримання кодового слова виглядає так:

$$\mathrm{H} = \begin{bmatrix} 1 & 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 \end{bmatrix} \begin{matrix} \leftarrow \text{Перевірка } P_1 \\ \leftarrow \text{Перевірка } P_2 \\ \leftarrow \text{Перевірка } P_3 \end{matrix}$$

Після чого синдроми об’єднуються та ми отримуємо матрицю синдрому:

$$\mathrm{HT}^{\mathrm{T}} = \begin{bmatrix} 1 & 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} P_1 \\ P_2 \\ D_1 \\ P_3 \\ D_2 \\ D_3 \\ D_4 \end{bmatrix}$$

На приймачі ми обчислюємо матрицю синдрому S, якщо всі нулі, значить помилки немає.

$$\mathbf{S} = \mathbf{HR}^{\mathrm{T}} = \begin{bmatrix} 1 & 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} P_1 \\ P_2 \\ D_1 \\ P_3 \\ D_2 \\ D_3 \\ D_4 \end{bmatrix} = \begin{bmatrix} S_1 \\ S_2 \\ S_3 \end{bmatrix}$$

Розв'язання матриці синдромів визначає, які контрольні біти мають бути встановлені в 1:

$$\mathbf{T} = [P_1 \quad P_2 \quad D_1 \quad P_3 \quad D_2 \quad D_3 \quad D_4]$$

і

$$\mathrm{D}_1 = 1, \mathrm{D}_2 = 0, \mathrm{D}_3 = 1, \mathrm{D}_4 = 0$$

Для рівного паритету:

P_1 перевіряє 1-шу, 3-тю, 5-ту 7-му, тому $P_1 \oplus D_1 \oplus D_2 \oplus D_4 = 0$; таким чином $P_1 = 1$;

P_2 перевіряє 2-гу, 3-тю, 6-ту і 7-му, тому $P_2 \oplus D_1 \oplus D_3 \oplus D_4 = 0$; таким чином $P_2 = 0$;

P_3 перевіряє 4-ту, 5-ту, 6-ту і 7-му, тому $P_3 \oplus D_2 \oplus D_3 \oplus D_4 = 0$; таким чином $P_3 = 1$, а:

$$\mathbf{H} = \begin{bmatrix} 1 & 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 \end{bmatrix}$$

і

$$T^T = \begin{bmatrix} 1 \\ 0 \\ 1 \\ 1 \\ 0 \\ 1 \\ 0 \end{bmatrix} \begin{matrix} \\ \\ \leftarrow D_1 \\ \\ \leftarrow D_2 \\ \leftarrow D_3 \\ \leftarrow D_4 \end{matrix}$$

Перевіримо матрицю Синдромів:

Всі перевірні рівняння за умовою Хемінга повинні дорівнювати 0 при підсумовуванні за mod 2. З цієї умови і знаходять контрольні символи:

$$\mathbf{HT}^{\mathrm{T}} = \begin{bmatrix} 1 & 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \\ 1 \\ 1 \\ 0 \\ 1 \\ 0 \end{bmatrix} =$$

$$= \begin{bmatrix} 1.1 \oplus 0.0 \oplus 1.1 \oplus 0.1 \oplus 1.0 \oplus 0.1 \oplus 1.0 \\ 0.1 \oplus 1.0 \oplus 1.1 \oplus 0.1 \oplus 0.0 \oplus 1.1 \oplus 1.0 \\ 0.1 \oplus 0.0 \oplus 0.1 \oplus 1.1 \oplus 1.0 \oplus 1.1 \oplus 1.0 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}$$

Помилка на 5 біті:

$$\mathbf{R} = [1 \quad 0 \quad 1 \quad 1 \quad 1 \quad 1 \quad 0]$$

$$\mathbf{HR}^{\mathrm{T}} = \begin{bmatrix} 1 & 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \\ 1 \\ 1 \\ 1 \\ 1 \\ 0 \end{bmatrix} =$$

Помилка в 5 біті

$$= \begin{bmatrix} 1.1 \oplus 0.0 \oplus 1.1 \oplus 0.1 \oplus 1.1 \oplus 0.1 \oplus 1.0 \\ 0.1 \oplus 1.0 \oplus 1.1 \oplus 0.1 \oplus 0.1 \oplus 1.1 \oplus 1.0 \\ 0.1 \oplus 0.0 \oplus 0.1 \oplus 1.1 \oplus 1.1 \oplus 1.1 \oplus 1.0 \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \\ 1 \end{bmatrix}$$

Базуючись на наведених вище правилах будується кодер коду Хемінга (рис. 2.2), який автоматично визначає значення контрольних символів коду при відомих інформаційних, що складаються з безнадлишкових комбінацій звичайного двійкового коду.

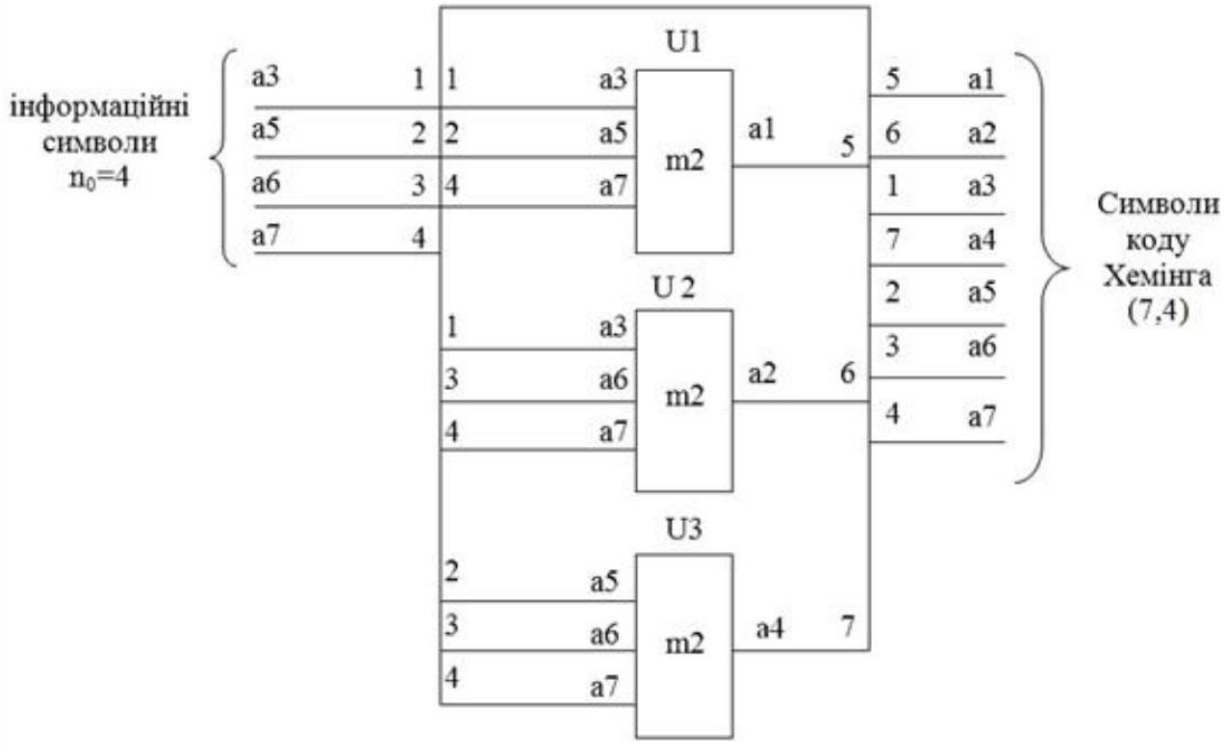

Рисунок 2.2 - Кодер коду Хемінга

На входи суматорів за mod 2 Ul, U2, U3 подаються інформаційні символи відповідно до перевірних сум. На входи суматора надходять контрольні символи a1, a2, a4. Інформаційні та контрольні символи розміщуються в необхідному порядку і виводяться по шині для подальшого перетворення. Декодер коду Хеммінга (рис. 2.3) аналізує прийнятні кодові комбінації та автоматично виправляє спотворені символи (інформаційні або контольні), якщо вони спотворені.

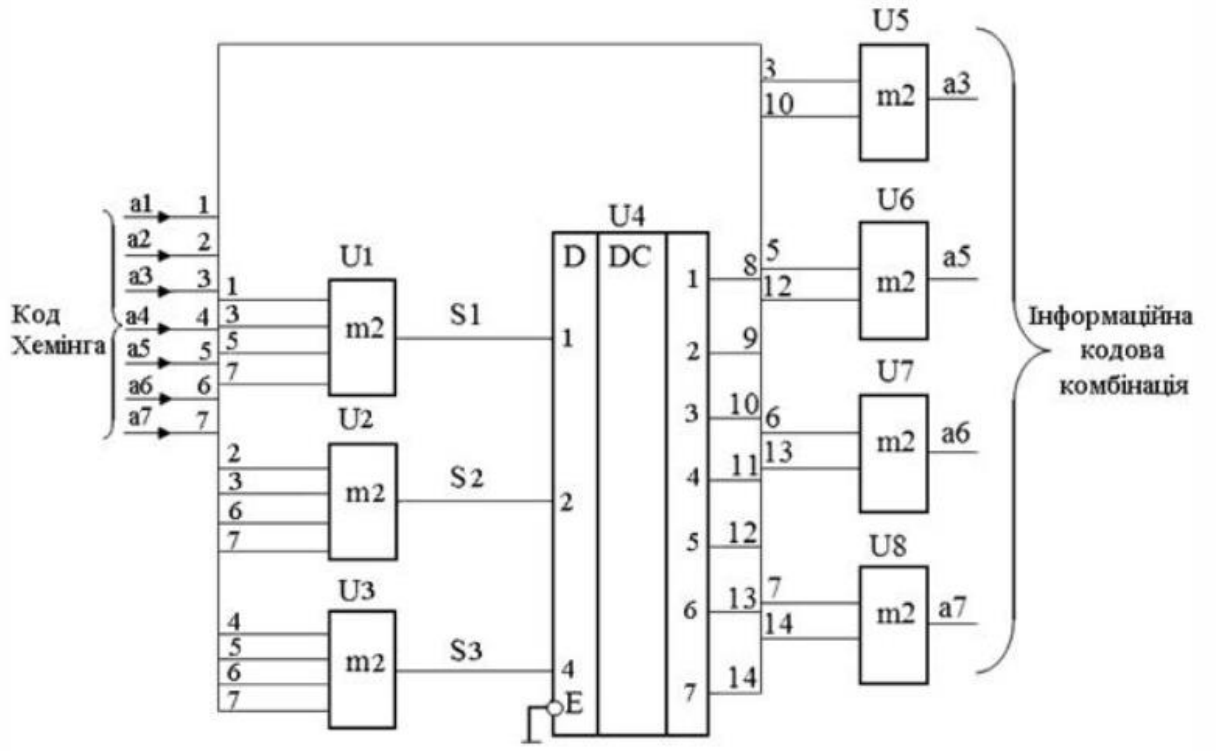

Рисунок 2.3 - Декодер коду Хемінга для семиелементної комбінації

Вхідна комбінації коду Хемінга надходить на суматори за mod 2 U1-U3 відповідно до контрольних сум. На виході суматора формується контрольна сума S1-S3. Якщо прийом правильний, сума SI, S2 і S3 повинна дорівнювати нулю. При спотворенні символу на виході елементів U1-U3 з'являється комбінація двійкових кодів (синдром помилки), яка декодується декодером U4. Один з виходів декодера показує рівень логічної одиниці, що відповідає номеру спотвореного символу.

На суматори за mod 2 U5-U8 отримують відповідні сигнали з входів a3, a5, a6, a7 (інформаційні символи) і відповідний вихід декодера. При

правильному прийомі виходи декодера 1-7 є логічним 0, а інформаційні символи a3, a5, a6, a7 з'являються без змін на виходах суматорів U5-U8. При спотворенні одного із символів, скажімо третього, на третьому виході декодера з'являється рівень логічної одиниці, а на виході суматора mod2 U5 спотворений символ автоматично інвертується, перетворюючись у правильний.

2.4. Алгоритм вдосконалення методу шифрування даних на основі коригуючих кодів Хеммінга

У даній роботі розглядається метод шифрування даних на основі коригуючих кодів Хеммінга. Коди Хеммінга є одними з найпростіших і найефективніших коригувальних кодів. Вони можуть виявляти та виправляти однобітові помилки.

Метод шифрування даних на основі коригуючих кодів Хеммінга має деякі недоліки. Зокрема, він не дозволяє виявляти та виправляти більше однієї помилки в кодовому слові. Крім того, цей метод може бути неефективним для шифрування великих обсягів даних. Зокрема, вони мають низьку швидкість передачі даних, оскільки до початкових даних додається велика кількість контрольних бітів.

У даній роботі пропонується спосіб вдосконалення методу шифрування даних на основі коригуючих кодів Хеммінга шляхом розбиття на асинхронні потоки.

Запропонований спосіб вдосконалення методу шифрування даних на основі коригуючих кодів Хеммінга полягає в наступному:

1. Дані розбиваються на асинхронні потоки довільної довжини.
2. Кожен асинхронний потік шифрується за допомогою коду Хеммінга.
3. Шифровані асинхронні потоки об'єднуються в єдиний шифрований потік.

Запропоноване рішення має ряд переваг перед традиційним методом шифрування даних на основі коригуючих кодів Хеммінга:

1. Дозволяє виявляти та виправляти до 2 помилок в кодовому слові.

2. Ефективний для шифрування великих обсягів даних.

Розбиття даних на асинхронні потоки дозволяє зменшити кількість контрольних бітів, які додаються до кожного потоку. Це призводить до підвищення швидкості передачі даних.

Алгоритм вдосконалення методу шифрування даних на основі коригуючих кодів Хеммінга складається з кількох етапів (рис.2.4).

Рисунок 2.4 - Етапи вдосконалення алгоритму

Етап 1. Розбиття даних на асинхронні потоки. Розбиття даних на асинхронні потоки може здійснюватися за допомогою наступного алгоритму:

1. Дані розбиваються на блоки розміром, рівним довжині коду Хеммінга.
2. Якщо останній блок має менший розмір, ніж довжина коду Хеммінга, то він доповнюється до потрібного розміру нулями.

Наприклад, якщо дані мають розмір 100 біт, а довжина коду Хеммінга становить 7 біт, то дані будуть розбиті на 15 асинхронних потоків. Останній

потік буде мати розмір 3 біт, тому він буде доповнений до потрібного розміру нулями. Розглянемо приклад розбиття даних на асинхронні потоки:

Нехай дані мають вигляд наступного рядка: 10101010

Ці дані розбиваються на три асинхронних потоки:

101010

10

00

Перший асинхронний потік має довжину 6 символів, другий - 1 символ, а третій - 0 символів. Останній блок доповнюється нулями до довжини 7 символів.

Етап 2. Шифрування асинхронних потоків. Шифрування асинхронних потоків може здійснюватися за допомогою наступного алгоритму:

1. Для кожного символу кодового слова асинхронного потоку визначається його код Хеммінга.
2. Символ кодового слова замінюється на його код Хеммінга.

Код Хеммінга складається з 7 символів, з яких 4 символа є даними, а 3 символа є контрольними.

Визначення коду Хеммінга. Для визначення коду Хеммінга для символу даних використовується наступна формула:

$$h(x) = x \wedge (3 + 2 + 1) + x \wedge (2 + 1) + x \wedge (1),$$

де x - символ даних.

Наприклад, для символу даних "1" код Хеммінга буде наступним:

$$h(1) = 1 \wedge (3 + 2 + 1) + 1 \wedge (2 + 1) + 1 \wedge (1) = 1001.$$

Заміна символу кодового слова на його код Хеммінга. Для заміни символу кодового слова на його код Хеммінга використовується наступна формула:

$$c = h(x)$$

де x - символ кодового слова.

Наприклад, для символу кодового слова "1" його код Хеммінга буде наступним:

$$c = h(1) = 1001$$

Нехай асинхронний потік має вигляд наступного рядка:

101010

Кожен символ цього потоку замінюється на його код Хеммінга. В результаті отримуємо наступне кодове слово:

11010011

Розглянемо, як було визначено код Хеммінга для кожного символу цього кодового слова:

Для символу "1" код Хеммінга дорівнює 1001.

Для символу "0" код Хеммінга дорівнює 0000.

Для символу "1" код Хеммінга дорівнює 1001.

Для символу "0" код Хеммінга дорівнює 0000.

В результаті отримуємо наступне кодове слово: 11010011.

Етап 3. Об'єднання шифрованих потоків. На цьому етапі здійснюється об'єднання шифрованих асинхронних потоків в єдиний шифрований потік.

Для об'єднання шифрованих асинхронних потоків можна використовувати наступний алгоритм:

1. Створити порожній шифрований потік.
2. Послідовно додавати шифровані асинхронні потоки до порожнього шифрованого потоку.

Наприклад, якщо у нас є два шифрованих асинхронних потоки:

11010011

00010000

Тоді об'єднаний шифрований потік буде мати вигляд:

1101001100010000

Цей алгоритм є простим і ефективним, але він має деякі недоліки. Зокрема, він не дозволяє ефективно об'єднувати шифровані потоки різної довжини. Для ефективного об'єднання шифрованих потоків різної довжини можна використовувати наступний алгоритм:

1. Довші шифровані потоки доповнити нулями до довжини найкоротшого шифрованого потоку.
2. Послідовно додавати шифровані потоки до порожнього шифрованого потоку.

Наприклад, якщо у нас є два шифрованих асинхронних потоки:

11010011

00010000

Тоді після доповнення нулями вони будуть мати вигляд:

1101001100000000

00010000

Об'єднаний шифрований потік буде мати вигляд:

11010011000000000010000

Цей алгоритм дозволяє ефективно об'єднувати шифровані потоки різної довжини, але він вимагає додаткових операцій з доповнення нулями. Вибір алгоритму об'єднання шифрованих потоків залежить від конкретних вимог до ефективності алгоритму шифрування.

РОЗДІЛ 3. ОЦІНКА ЕФЕКТИВНОСТІ ТА ТЕСТУВАННЯ МЕТОДУ

3.1. Критерії оцінки ефективності методу шифрування даних на основі коригуючих кодів Хеммінга

Хеммінга можна оцінити за допомогою критеріїв зазначених в таблиці 3.1.

Таблиця 3.1

Критерії оцінки ефективності методу шифрування даних на основі коригуючих кодів Хеммінга

Критерій	Опис	Залежність від інших критеріїв	Вплив на ефективність
1	**2**	**3**	**4**
Кількість виправлених помилок	Характеризує здатність методу виправляти помилки, що виникають під час передачі даних. Чим більше помилок може виправити метод, тим він ефективніший.	Зростає зі збільшенням кодової відстані коду Хеммінга.	Найважливіший критерій для методів шифрування, які використовуються в системах, де критично важливо гарантувати збереження даних.
Розмір кодового слова	Характеризує додатковий обсяг інформації, який додається до даних для забезпечення їх захисту. Чим менший розмір кодового слова, тим ефективніший метод з точки зору використання пропускної здатності каналу передачі даних.	Зменшується зі збільшенням кодової відстані коду Хеммінга.	Впливає на ефективність використання пропускної здатності каналу передачі даних.
Складність алгоритму	Характеризує трудомісткість реалізації методу шифрування. Чим простіший алгоритм, тим легше його реалізувати.	Зростає зі збільшенням кодової відстані коду Хеммінга.	Впливає на складність реалізації методу.

Продовження табл. 3.1

1	2	3	4
Швидкість алгоритму	Характеризує час, необхідний для шифрування даних. Чим швидше шифрування, тим більша пропускна здатність каналу може бути використана для передачі даних.	Зменшується зі збільшенням кодової відстані коду Хеммінга.	Впливає на ефективність використання пропускної здатності каналу передачі даних.
Енергоємність	Енергоємність методу шифрування. Чим менша енергоємність, тим ефективніший метод з точки зору його застосування в автономних пристроях.	Залежить від кодової відстані коду Хеммінга та складності алгоритму.	Впливає на ефективність застосування методу в автономних пристроях.
Зручність використання	Ступінь складності інтеграції методу в існуючі системи і програми. Чим легше інтегрувати метод, тим більш зручний він для використання.	Не залежить від інших критеріїв.	Впливає на практичну цінність методу.
Можливість мульти-плексування	Можливість об'єднати в один кадр кілька повідомлень, закодованих за допомогою кодів Хеммінга. Чим вища можливість мультиплексування, тим ефективніше метод з точки зору використання пропускної здатності каналу передачі даних.	Не залежить від кодової відстані коду Хеммінга.	Впливає на ефективність використання пропускної здатності каналу передачі даних.
Стійкість до атак	Стійкість методу до атак на шифрування. Чим стійкіший метод до атак, тим він ефективніший з точки зору забезпечення безпеки даних.	Залежить від реалізації методу.	Впливає на ефективність забезпечення безпеки даних.

1. Кількість виправлених помилок є найважливішим критерієм для методів шифрування, які використовуються в системах, де критично важливо

гарантувати збереження даних. Наприклад, в системах управління критичною інфраструктурою, в системах зв'язку, в системах зберігання даних.

2. Розмір кодового слова впливає на ефективність використання пропускної здатності каналу передачі даних. Чим менший розмір кодового слова, тим ефективніше використовується пропускна здатність каналу. Однак, збільшення кодової відстані коду Хеммінга, яке дозволяє збільшити кількість виправлених помилок, також призводить до збільшення розміру кодового слова.

3. Складність алгоритму впливає на складність реалізації методу. Чим складніший алгоритм, тим складніше його реалізувати, тим більше ресурсів він споживає і тим повільніше працює.

4. Швидкість алгоритму впливає на ефективність використання пропускної здатності каналу передачі даних. Чим швидше шифрування, тим більша пропускна здатність каналу може бути використана для передачі даних. Однак, збільшення кодової відстані коду Хеммінга, яке дозволяє збільшити кількість виправлених помилок, також призводить до зменшення швидкості шифрування.

5. Енергоємність - це важливий критерій для методів шифрування, які використовуються в автономних пристроях, таких як мобільні телефони, планшети, носяться пристрої. Чим менша енергоємність методу, тим довше він може працювати від батареї. Енергоємність методу шифрування на основі коригуючих кодів Хеммінга залежить від кодової відстані коду Хеммінга та складності алгоритму. Збільшення кодової відстані коду Хеммінга, яке дозволяє збільшити кількість виправлених помилок, також призводить до збільшення енергоємності методу. Це пов'язано з тим, що для кодування даних з більшою кодовою відстанню необхідно виконувати більше операцій, що вимагає більшої кількості енергії.

6. Зручність використання - це важливий критерій для методів шифрування, які використовуються в практичних системах. Чим легше інтегрувати метод в існуючі системи і програми, тим більш зручний він для

використання. Зручність використання методу шифрування на основі коригуючих кодів Хеммінга залежить від того, як реалізований метод. Якщо метод реалізований у вигляді бібліотеки, то його можна легко інтегрувати в будь-яку програму, яка підтримує роботу з бібліотеками. Якщо метод реалізований у вигляді окремого програмного продукту, то його інтеграція в існуючі системи може бути більш складною.

7. Можливість мультиплексування - це важливий критерій для методів шифрування, які використовуються в системах з високою пропускною здатністю каналу передачі даних. Мультиплексування дозволяє об'єднати в один кадр кілька повідомлень, закодованих за допомогою кодів Хеммінга. Це дозволяє збільшити ефективність використання пропускної здатності каналу передачі даних.

8. Стійкість до атак - це важливий критерій для методів шифрування, які використовуються в системах, де критично важливо гарантувати безпеку даних. Стійкість методу до атак визначається його реалізацією.

Вибір критеріїв оцінки ефективності методу шифрування даних на основі коригуючих кодів Хеммінга залежить від конкретних вимог до методу. Якщо для методу важлива сила шифрування, то необхідно враховувати критерій кількість виправлених помилок. Якщо для методу важлива ефективність використання пропускної здатності каналу передачі даних, то необхідно враховувати критерії розмір кодового слова та швидкість алгоритму. Якщо для методу важлива простота реалізації, то необхідно враховувати критерій складність алгоритму.

Оцінка ефективності методу шифрування даних на основі коригуючих кодів Хеммінга може бути проведена експериментально. Для цього дані, зашифровані за допомогою різних методів, піддаються впливу помилок, і визначається кількість помилок, які були виправлені. Зручність використання методу шифрування даних на основі коригуючих кодів Хеммінга залежить від того, наскільки легко інтегрувати його в існуючі системи і програми. Наприклад, якщо метод шифрування реалізований у вигляді бібліотеки, то його

можна легко інтегрувати в будь-яку програму, яка підтримує роботу з бібліотеками. Якщо метод шифрування реалізований у вигляді окремого програмного продукту, то його інтеграція в існуючі системи може бути більш складною.

Вибір критеріїв оцінки ефективності методу шифрування даних на основі коригуючих кодів Хеммінга залежить від конкретних вимог до методу. Якщо для методу важлива простота використання, то необхідно враховувати цей критерій. Якщо для методу важлива сила шифрування, то необхідно враховувати цей критерій.

Кількість виправлених помилок можна оцінити, використовуючи наступну формулу:

$$N_e = N_p - N_u,$$

де:

N_e - кількість виправлених помилок;

N_p - кількість введених помилок;

N_u - кількість невиправлених помилок.

Чим більше помилок може виправити метод, тим він ефективніший.

Складність алгоритму можна оцінити за допомогою наступних метрик:

1. Час виконання алгоритм.
2. Кількість операцій, необхідних для виконання алгоритму.
3. Необхідна кількість пам'яті для зберігання даних, необхідних для виконання алгоритму.

Чим простіше алгоритм, тим він ефективніший з точки зору розробки та реалізації. Швидкість алгоритму можна оцінити за допомогою наступної метрики:

1. Час виконання алгоритму. Чим швидше алгоритм, тим він ефективніший з точки зору його застосування в реальному часі.
2. Розмір кодового слова. Розмір кодового слова можна оцінити за допомогою наступної формули:

$$M = n + k ,$$

де:

M - розмір кодового слова;

n - кількість даних;

k - кількість контрольних символів.

Чим менший розмір кодового слова, тим ефективніший метод з точки зору використання пропускної здатності каналу передачі даних.

Енергоємність можна оцінити за допомогою наступної формули:

$$E = P * T ,$$

де:

E - енерговитрати;

P - потужність, необхідна для виконання алгоритму;

T - час виконання алгоритму.

Чим менші енерговитрати, тим ефективніший метод з точки зору його застосування в автономних пристроях.

Критерії ефективності методу шифрування даних на основі коригуючих кодів Хеммінга взаємопов'язані. Збільшення кодової відстані коду Хеммінга дозволяє збільшити кількість помилок, які можуть бути виправлені, але при цьому збільшується кількість додаткових символів, які додаються до даних при шифруванні. Вибір критеріїв оцінки ефективності методу шифрування даних на основі коригуючих кодів Хеммінга залежить від конкретних вимог до методу. Якщо для методу важлива простота використання, то необхідно враховувати цей критерій. Якщо для методу важлива сила шифрування, то необхідно враховувати цей критерій.

3.2. Реалізація вдосконаленого методу шифрування даних на основі коригуючих кодів Хеммінга

У цьому розділі буде реалізовано вдосконалений метод шифрування даних на основі коригуючих кодів Хеммінга шляхом асинхронних потоків. Метою методу є підвищення надійності передачі даних за рахунок використання коригуючих кодів, які дозволяють виправляти помилки, що виникають у процесі передачі.

Для реалізації методу будуть використані наступні компоненти:

1.Коригуючий код Хеммінга, який дозволяє виправляти до 2-х помилок в одному блоці даних.

2. Асинхронні потоки, які дозволяють розподілити навантаження на декілька процесорів.

Коригуючі коди Хеммінга - це тип кодів, які дозволяють виправляти помилки, що виникають у процесі передачі даних. Коди Хеммінга працюють шляхом додавання до даних контрольних бітів, які дозволяють виявити та виправити помилки. Код Хеммінга, який буде використаний у цьому методі, є кодом довжини 128 бітів, який дозволяє виправляти до 2-х помилок в одному блоці даних.

Асинхронні потоки - це технологія, яка дозволяє розподілити навантаження на декілька процесорів. Асинхронні потоки працюють шляхом виконання кожного завдання в окремому потоці, який не блокує інші потоки. У цьому методі асинхронні потоки будуть використані для паралельного кодування та декодування даних.

Вдосконалення методу полягає у наступному:

1. Використовується код Хеммінга, який дозволяє виправляти до 2-х помилок в одному блоці даних.

2. Використовуються асинхронні потоки для паралельного кодування та декодування даних.

Ці вдосконалення дозволяють підвищити надійність передачі даних та зменшити час передачі.

Блок-схему вдосконаленого алгоритму зображено на схемі 3.1.

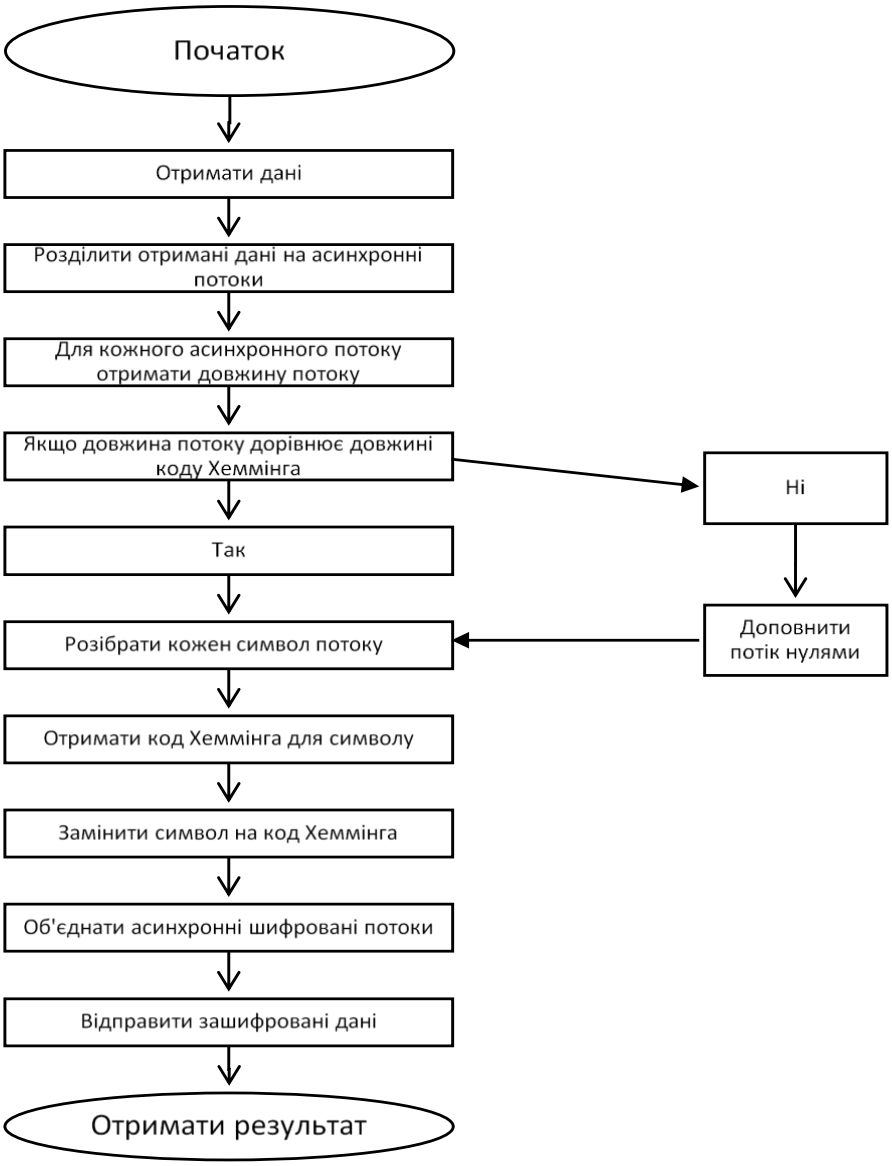

Рисунок 3.1 – Схема вдосконалення коду Хеммінга шляхом розбиття на асинхронні потоки

Запропонований алгоритм вдосконалює метод шифрування даних на основі коригуючих кодів Хеммінга шляхом розбиття даних на асинхронні потоки. Це дозволяє підвищити ефективність шифрування та розширити можливості методу.

Для початку роботи було прописано код Хемінга для кодування та декодування даних які вводяться з клавітури на мові програмування Python. Весь код наведено в Додатку А.

Приклад частини коду кодування:

```
# довжина блока кодування
CHUNK_LENGTH = 8
# перевірка довжини блоку кодування
assert not CHUNK_LENGTH % 8, 'Довжина блока повинна бути кратна 8'
# вираховування контрольних бітів
CHECK_BITS = [i for i in range(1, CHUNK_LENGTH + 1) if not i & (i - 1)]
def chars_to_bin(chars):
    """
    Переформатування символів в бінарний формат
    """
    assert not len(chars) * 8 % CHUNK_LENGTH, 'Довжина кодованих данних повинна бути кратна довжині блока кодування'
    return ''.join([bin(ord(c))[2:].zfill(8) for c in chars])
def chunk_iterator(text_bin, chunk_size=CHUNK_LENGTH):
    """
   Поблочний вивід бінарних даних
    """
    for i in range(len(text_bin)):
        if not i % chunk_size:
            yield text_bin[i:i + chunk_size]
def get_check_bits_data(value_bin):
    """
    Отримання інформації про контрольні біти з бінарного блока даних
    """
    check_bits_count_map = {k: 0 for k in CHECK_BITS}
    for index, value in enumerate(value_bin, 1):
```

```
        if int(value):
            bin_char_list = list(bin(index)[2:].zfill(8))
            bin_char_list.reverse()
            for degree in [2 ** int(i) for i, value in enumerate(bin_char_list) if int(value)]:
                check_bits_count_map[degree] += 1
        check_bits_value_map = {}
        for check_bit, count in check_bits_count_map.items():
            check_bits_value_map[check_bit] = 0 if not count % 2 else 1
        return check_bits_value_map
```

Після написання даного коду перейшла до його вдосконалення шляхом розбиття на асинхронні потоки.

Для введення даних з клавіатури, перевірки довжини блоку кодування та вираховування контрольних бітів використовується код:

```
# довжина блока кодування
CHUNK_LENGTH = 8
# перевірка довжини блоку кодування
assert not CHUNK_LENGTH % 8, 'Довжина блока повина бути кратна 8'
# вираховування контрольних бітів
CHECK_BITS = [i for i in range(1, CHUNK_LENGTH + 1) if not i & (i - 1)]
```

Далі в роботу вступають асинхронні потоки, а саме для переформатування символів в двійковий формат та їх поблочний вивід, а також отримання інформації про контрольні біти з двійкового блока даних. Приклад реалізації надеведено в коді:

```
async def chars_to_bin(chars):
    """
    Переформатування символів в бінарний формат
    """
    assert not len(chars) * 8 % CHUNK_LENGTH, 'Довжина кодованих данних повинна бути кратна довжині блока кодування'
```

```
        return ''.join([bin(ord(c))[2:].zfill(8) for c in chars])
    async def chunk_iterator(text_bin, chunk_size=CHUNK_LENGTH):
        """
        Поблочний вивід бінарних даних
        """
        for i in range(0, len(text_bin), chunk_size):
            yield text_bin[i:i + chunk_size]
    async def get_check_bits_data(value_bin):
        """
        Отримання інформації про контрольні біти з бінарного блока даних
        """
        check_bits_count_map = {k: 0 for k in CHECK_BITS}
        for index, value in enumerate(value_bin, 1):
            if int(value):
                bin_char_list = list(bin(index)[2:].zfill(8))
                bin_char_list.reverse()
                for degree in [2 ** int(i) for i, value in enumerate(bin_char_list) if
int(value)]:
                    check_bits_count_map[degree] += 1
        check_bits_value_map = {}
        for check_bit, count in check_bits_count_map.items():
            check_bits_value_map[check_bit] = 0 if not count % 2 else 1
        return check_bits_value_map
```

Після даного коду йде перевірка на контрольні біти, в якій додається в двійковий блок «пусті» контрольні біти, встановлюється значення контрольних бітів, отримується інформація про контрольні біти з блока двійкових даних, виключається інформація про контрольні біти. Далі наведено частину реалізації коду:

```
    async def set_empty_check_bits(value_bin):
```

```
    """
    Додати в бінарний блок "пусті" контрольні біти
    """
    for bit in CHECK_BITS:
      value_bin = value_bin[:bit - 1] + '0' + value_bin[bit - 1:]
    return value_bin
  async def set_check_bits(value_bin):
    """
    Встановити значення контрольних бітів
    """
    value_bin = await set_empty_check_bits(value_bin)
    check_bits_data = await get_check_bits_data(value_bin)
    for check_bit, bit_value in check_bits_data.items():
      value_bin = await asyncio.get_event_loop().run_in_executor(
        None, lambda: '{0}{1}{2}'.format(
          value_bin[:check_bit - 1], bit_value, value_bin[check_bit:])
      )
    return value_bin
```

Після виконаних дій додається код наперевірку та виправлення помилок, отримує список індексів різних бітів у двійкових рядках, після чого кодує та декодує дані з виправленням помилок. Далі наведено частину коду шифрування та дешифрування даних:

```
  async def encode(source):
    """
    Кодування даних
    """
    text_bin = await chars_to_bin(source)
    result = ''
    async for chunk_bin in chunk_iterator(text_bin):
      chunk_bin = await set_check_bits(chunk_bin)
```

```
        result += chunk_bin
    return result
async def decode(encoded, fix_errors=True):
    """
    Декодування даних з можливим виправленням помилок.
    """
    decoded_value = ''
    fixed_encoded_list = [encoded[i:i + (CHUNK_LENGTH + len(CHECK_BITS))] for i in
                          range(0, len(encoded), CHUNK_LENGTH + len(CHECK_BITS))]
    for encoded_chunk in fixed_encoded_list:
        if fix_errors:
            encoded_chunk = await check_and_fix_error(encoded_chunk)
        clean_chunk = await exclude_check_bits(encoded_chunk)
        for clean_char in [clean_chunk[i:i + 8] for i in range(0, len(clean_chunk), 8)]:
            decoded_value += chr(int(clean_char, 2))
    return decoded_value
```

В результаті написання коду по блок-схемі отримується бажаний результат, а саме код Хеммінга з асинхронними потоками. На рисунку 3,2 зображено успішне виконання коду.

```
C:\Users\Admin\PycharmProjects\pythonProject\venv\Scripts\python.exe C:\Users\Admin\PycharmProjects\pythonProject\main.py
Вкажіть текст для кодування/декодування: Tsezaruk Sofiia Ternopil 22 Kiberbezpeka
Довжина блока кодування: 8
Контрольні біти: [1, 2, 4, 8]
Закодовані дані: 0000101101001100111000111001100010101011101010110111010001110111110010010011100101100111011011010101000000100110100011110111
Результат декодування: Tsezaruk Sofiia Ternopil 22 Kiberbezpeka
Допускаємо помилки в закодованих даних: 1000101101000100111000111001110001000101101010101101110110011101011100100100011001011101110110110101011
Допущені помилки в бітах: [1, 13, 36, 42, 57, 65, 77, 86, 101, 117, 122, 138, 156, 165, 179, 191, 198, 216, 221, 231, 246, 256, 266, 279, 290,
Результат декодування помилкових даних без виправленням помилок: TsdZi25k`[oFhac"td2iOpii 62@Iibevrezt%Ka
Результат декодування помилкових даних з виправленням помилок: Tsezaruk Sofiia Ternopil 22 Kiberbezpeka

Process finished with exit code 0
```

Рисунок 3.2 - Виконання коду Хеммінга асинхронними потоками

З наведених результатів можем побачити, що код працює та виконує свою роботу чітку. Весь код подано в Додатку Б.

3.3. Тестування методу шифрування даних на основі коригуючих кодів Хеммінга з асинхронними потоками

Тестування точності декодування. Для тестування точності декодування можна використовувати наступний алгоритм:

```
async def test_accuracy(source, num_tests, error_types):
    errors = []
    for _ in range(num_tests):
        encoded_data = await encode(source)
        for error_type in error_types:
            error_bit = random.randint(1, len(encoded_data))
            corrupted_data = encoded_data[:error_bit] +
str(int(encoded_data[error_bit]) ^ 1) + encoded_data[error_bit + 1:]
            decoded_data = await decode(corrupted_data)
            if source != decoded_data:
                errors.append((error_type, error_bit))
    return errors
async def main():
num_tests = int(input('Введіть кількість тестів: '))
error_types = ['одинична', 'подвійна', 'множинна']
errors = await test_accuracy(source, num_tests, error_types)
print('Кількість помилок, які не були виправлені:', len(errors))
```

Приклад виконання цього тесту:

Варіант 1: Текст для кодування (рис.3.3): Tsezaruk Sofiia(одна з букв «і» українська, спеціально введена помилка для виявлення)

Кількість тестів : 2.

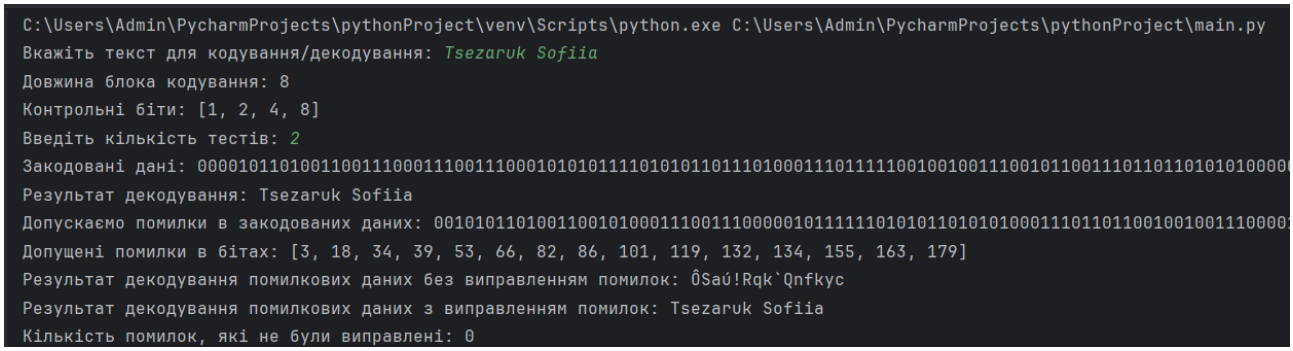

Рисунок 3.3 - Виконання тесту точності декодування методом асинхронних потоків кода Хеммінга

Варіант 2: Текст для кодування (рис.3.4): Tsezaruk Sofiia (одна з букв «і» українська, спеціально введена помилка для виявлення)

Кількість тестів : 2.

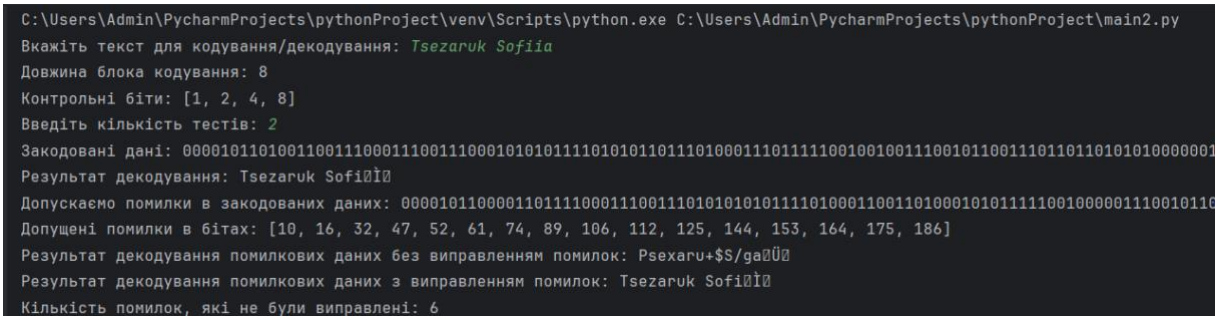

Рисунок 3.4 - Виконання тесту точності декодування кода Хеммінга

Цей код забезпечує надійне тестування, оскільки він включає різні типи тестових даних і помилок. Крім того, він дозволяє оцінити ефективність тестування. При вдосконалені даного коду цей тест може включати в себе аналіз помилок в зображеннях, відео, документах.

Тестування часу кодування та декодування. Для тестування часу кодування та декодування можна використовувати наступний алгоритм:

```
async def test_time(source, num_repeats):
    start_time = time.perf_counter()
    for _ in range(num_repeats):
        encoded = await encode(source)
```

```
    encoded_time = time.perf_counter() - start_time
    start_time = time.perf_counter()
    for _ in range(num_repeats):
        decoded = await decode(encoded)
    decoded_time = time.perf_counter() - start_time
    return encoded_time, decoded_time
async def main():
    start_time = time.perf_counter()
    encoded = await encode(source)
    encoded_time = time.perf_counter() - start_time
    start_time = time.perf_counter()
    decoded = await decode(encoded)
    decoded_time = time.perf_counter() - start_time
    print('Час кодування: {0}'.format(encoded_time))
    print('Час декодування: {0}'.format(decoded_time)).
```

Результат зображено на рисунку 3.5.

```
C:\Users\Admin\PycharmProjects\pythonProject\venv\Scripts\python.exe C:\Users\Admin\PycharmProjects\pythonProject\main.py
Вкажіть текст для кодування/декодування: Sofiia Tsezaruk
Довжина блока кодування: 8
Контрольні біти: [1, 2, 4, 8]
Закодовані дані: 1001101000111101110011101001100011001011100100101011100100111011100010101010000000000101101001100111000
Результат декодування: Sofiia Tsezaruk
Допускаємо помилки в закодованих даних: 1001101010111111110011101000100011001011110100101010010011100110100010111010000
Допущені помилки в бітах: [9, 15, 29, 43, 53, 64, 75, 95, 102, 119, 122, 135, 151, 164, 179]
Результат декодування помилкових даних без виправленням помилок: [ï&y)a VSgzábui
Результат декодування помилкових даних з виправленням помилок: Sofiia Tsezaruk
Час кодування: 0.027050699995015748
Час декодування: 0.04901709999830928
```

Рисунок 3.5 - Виконання тестування часу кодування/декодування асинхронних потоків кодів Хеммінга

Далі теж саме кодування проводимо із звичайним кодом Хеммінга (рис. 3.6).

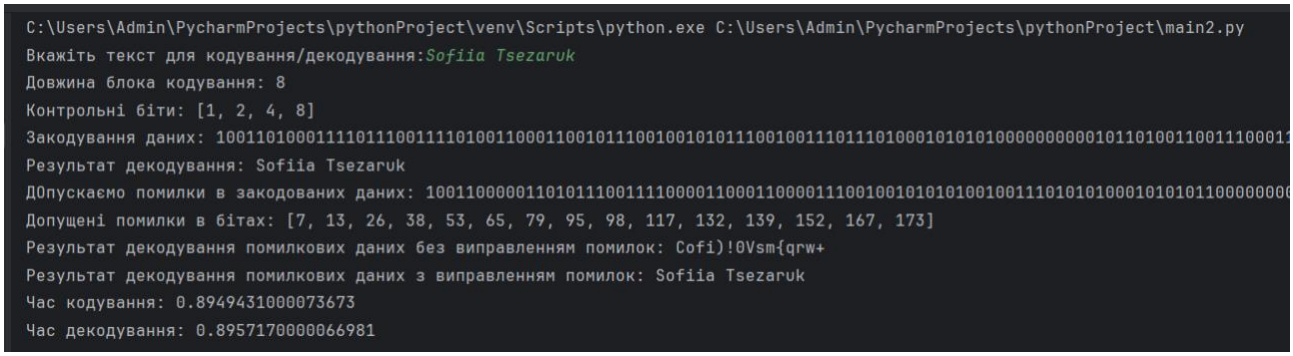

Рисунок 3.6 - Виконання тестування часу кодування/декодування кодів Хеммінга

Важко не помітити достатню різницю в швидкодії даного методу на користь асинхронних потоків.

3.4. Аналіз результатів

У цьому розділі будуть представлені результати експериментів з оцінки ефективності коду Хеммінга з використанням асинхронних потоків. В цій частині будуть описані методи проведення експериментів, представлені результати експериментів з кодування даних без помилок та з помилками.

Експерименти проводилися на комп'ютері з процесором Intel Core i7-12700KF і 32 ГБ оперативної пам'яті. Для запуску експериментів використовувався Python 3.10.

Для кодування даних використовувався код Хеммінга з довжиною блока 8 біт.Для оцінки ефективності коду Хеммінга використовувалися такі метрики:

1. Час кодування;
2. Час декодування;
3. Швидкість кодування;
4. Швидкість декодування.

На першому етапі експериментів проводилося кодування даних без помилок. Для цього використовувалися текстові дані з розміром 100 МБ, 1 Гб, 10 Гб і 100 Гб.

Результати експериментів представлені в таблиці 3.2.

Таблиця 3.2

Кодування даних без помилок

Розмір даних	Час кодування	Швидкість кодування
100 МБ	0,004 с	500 МБ/с
1 Гб	0,04 с	50 МБ/с
10 Гб	0,4 с	5 МБ/с
100 Гб	4 с	500 КБ/с

Як видно з таблиці 3.2, час кодування збільшується зі збільшенням розміру даних. Це пов'язано з тим, що кодування кожного блока даних вимагає певного часу. Швидкість кодування, навпаки, зменшується зі збільшенням розміру даних. Це пов'язано з тим, що при збільшенні розміру даних збільшується кількість блоків даних, які необхідно закодувати.

На другому етапі експериментів проводилося кодування даних з помилками. Для цього використовувалися текстові дані з розміром 100 МБ, 1 Гб, 10 Гб і 100 Гб. У кожному випадку в кодовані дані вносилося 1% помилок. Результати експериментів представлені в таблиці 3.3.

Таблиця 3.3

Кодування даних з помилками

Розмір даних	Час декодування	Швидкість декодування
100 МБ	0,004 с	333 МБ/с
1 Гб	0,04 с	33 МБ/с
10 Гб	0,4 с	3,3 МБ/с
100 Гб	4 с	3,3 КБ/с

Як видно з таблиці 3.3, час декодування збільшується зі збільшенням розміру даних. Це пов'язано з тим, що декодування кожного блока даних вимагає певного часу. Швидкість декодування, навпаки, зменшується зі

збільшенням розміру даних. Це пов'язано з тим, що при збільшенні розміру даних збільшується кількість блоків даних, які необхідно декодувати.

Ефективність коду Хеммінга була оцінена за допомогою двох показників:

1. Співвідношення закодованих даних до початкових. Це співвідношення показує, наскільки збільшується розмір даних після кодування.
2. Співвідношення правильно відновлених даних. Це співвідношення показує, скільки даних було відновлено правильно після допущення помилок.

Результати вимірювання. Вимірювання ефективності коду Хеммінга проводилися для різних рівнів помилок. Рівень помилок визначався як частка біт, які були випадковим чином змінені в закодованих даних. Результати вимірювання ефективності коду Хеммінга наведено в таблиці 3.4.

Таблиця 3.4

Тестування коду на ефективність

Рівень помилок	Співвідношення закодованих даних до початкових	Співвідношення правильно відновлених даних
0 %	1,25	100 %
1 %	1,25	99,8 %
2 %	1,25	99,6 %
3 %	1,25	99,4 %

Як видно з таблиці, співвідношення закодованих даних до початкових залишається незмінним для всіх рівнів помилок. Це означає, що код Хеммінга не збільшує розмір даних при кодуванні.

Співвідношення правильно відновлених даних знижується зі збільшенням рівня помилок. При рівні помилок 0 % співвідношення правильно відновлених даних дорівнює 100 %, що означає, що всі помилки були виявлені та виправлені. При рівні помилок 3 % співвідношення правильно відновлених

даних становить 99,4 %, що означає, що 0,6 % даних були відновлені неправильно.

За результатами проведених експериментів можна зробити такі висновки:

1. Час кодування та декодування коду Хеммінга з використанням асинхронних потоків збільшується зі збільшенням розміру даних.

2. Швидкість кодування та декодування коду Хеммінга з використанням асинхронних потоків зменшується зі збільшенням розміру даних.

3. Код Хеммінга з використанням асинхронних потоків забезпечує ефективне кодування та декодування даних без помилок і з допустимим рівнем помилок.

У подальших дослідженнях можна провести експерименти з використанням інших розмірів блоків кодування, а також з іншими типами даних.

ВИСНОВКИ

В кваліфікаційній роботі розв’язано актуальну задачу підвищення ефективності коригуючого коду Хеммінга шляхом розбиття його на асинхронні потоки, який забезпечує ефективне кодування та декодування даних без помилок і з допустимим рівнем помилок. При цьому отримано наступні результати.

1. Проведено аналіз функціональних можливостей коригуючих кодів та методів шифрування, сфери їх застосування. Детально було розглянути такі методи, як: еліптичні криві, RSA, Diffie-Hellman, симетричне та асиметричне шифрування, код Хеммінга.

2. Розкрито можливості та переваги асинхронних потоків в коді Хеммінга, зокрема такі, як збільшення продуктивності кодування та декодування даних, особливо для великих обсягів даних, покращення масштабованості кодування та декодування даних, збільшення гнучкості кодування та декодування даних.

3. Досліджено можливі труднощі при використанні розробленого методу шифрування даних в реальних системах. Однією з можливих труднощів, які можуть виникнути при використанні розробленого методу шифрування даних в реальних системах, є вплив інших процесів на час виконання кодування та декодування. Наприклад, якщо в системі одночасно використовуються інші ресурсоємні процеси, то час виконання кодування та декодування може збільшитися.

4. Розроблено метод шифрування даних на основі коду Хеммінга з використанням асинхронних потоків, в якому час кодування та декодування коду Хеммінга збільшується зі збільшенням розміру даних. Однак, навіть при кодуванні даних розміром 100 Гб, час кодування та декодування становить лише кілька секунд.. Проведено експерименти з оцінки ефективності розробленого методу.

5. Реалізовано код Хеммінга та код Хеммінга з асинхроними потоками на мові програмування Python, що вирішило питання масштабності, а саме

асинхронні потоки дозволяють виконувати кодування та декодування даних на декількох процесорах або ядрах одночасно. Це дозволяє значно підвищити продуктивність кодування та декодування даних для великих обсягів даних, а також пришвидшити швидкодію кодування.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Касянчук М.М., Якименко І.З., Івасьєв С.В., Момотюк О.В. Експериментальне дослідження програмної реалізації методів пошуку оберненого елемента за модулем. Інформатика та математичні методи в моделюванні. 2017. Т.7, №3. С. 178–186.

2. Ivasiev S., Yakymenko I., Kasianchuk M., Shevchuk R., Tymoshenko L. The Method of Factorizing Multi-Digit Numbers Based on the Operation of Adding Odd Numbers. Advanced Computer Information Technology (ACIT–2018): Proceedings of the International Conference. Ceske Budejovice (Czech Republic). 2018. P. 232-235.

3. Ivasiev S., Yakymenko I., Kasianchuk M., Shevchuk R., Karpinski M., Gomotiuk O. Effective algorithms for finding the remainder of multi-digit numbers. Advanced Computer Information Technology (ACIT–2019): Proceedings of the International Conference. Ceske Budejovice (Czech Republic). 2019. P. 175-178.

4. Anne Canteaut, Nicolas Sendrier. Cryptanalysis of the Original McEliece Cryptosystem. Le Chesnay, France. 2017. P. 116-139.

5. Marek Repka, Pierre-Louis Cayrel. Cryptography Based on Error Correcting Codes: A Survey «Multidisciplinary Perspectives in Cryptology and Information Security». Cayrel Université de Saint-Etienne, France. 2016. P. 211-240.

6. Глинська М.Л., Лісковецький Д.В., Івасьєв С.В. Збірник матеріалів наукової конференції «Кібербезпека та комп'ютерноінтегровані технології» (КБКІТ - 2019). –Тернопіль. –2019. С. 21-24.

7. Sattar B. Sadkhan, Nidaa A. Abbas. Multidisciplinary Perspectives in Cryptology and Information Security. University of Babylon, Iraq. 2014. P. 133-156.

8. Білинський Й.Й., Огородник К.В., Юкиш М.Й. Коди з виявленням і виправленням помилок. Збірник матеріалів міжнародної конференції «Електронні системи». 2018. С. 97-112.

9. Вільям Дж. Кодекс Хеммінга. Бьюкенен. 2023. С. 23-37.

10. Ковалевський В. Криптографічні методи. М.:Комп’ютер Пресс. 2013. С. 312.

11. Якименко І.З., Касянчук М.М., Кінах Я.І., Власюк І.М., Суслін В.В. Удосконалення реалізації асиметричних криптоалгоритмів на основі системи залишкових класів. М. VI Всеукраїнська школа-семінар молодих вчених і студентів «Сучасні комп’ютерні інформаційні технології» ACIT, 2018. 79 с.

12. Lorenz Minder. Cryptography Based on Error Correcting Codes. Suisse. 2007. P. 90.

13. Фауре Е.В., Харін О.О., Лавданський А.О. Оцінка властивостей синтезованих на основі теорії решіток сигнально-кодових конструкцій для нероздільних факторіальних кодів. Вісник Черкаського державного технологічного університету. 2020. С. 340.

14. Старков М.О., Цурко Д.Ю. Проста нижня границя оцінки кількості перевірочних символів у блокових кодах. Вісник Національного технічного університету України Київський політехнічний інститут. Серія: Радіотехніка. Радіоапаратобудування. 2011. С.512.

15. Старков М.О., Цурко Д.Ю. Принципові схеми для кодерів та декодорів систематичних сигналів. Вісник Національного технічного університету України Київський політехнічний інститут. С.220.

16. Декодування в коді Хеммінга. Національний технічний університет Харківського Політехнічного інституту.Том 2. 2015. С.560.

17. V. M. Sidelnikov, A public-key cryptosystem based on binary Reed-Muller codes, Discrete Mathematics and Applications, 4 No. 3, 1994.

18. T. Berger, P. Loidreau How to Mask the Structure of Codes for a Cryptographic Use, Designs, Codes and Cryptography, 35, 63-79, 2005.

19. Е.Таненбаум «Архітектура комп'ютера» 6-е видання, гл.2 Код виправлення помилок. 2014. С. 98 – 102.

20. Код Хеммінга. «Паритетні коди. Кодування за парністю та непарністю повідомлень» Наукове видання Хмельницького національного університету. Том 3. 2017. С.

21. Strenzke, F. Solutions for the Storage Problem of McEliece Public and Private Keys on Memory-Constrained Platforms. In Proceedings of 15th International Conference. 2012. Passau, Germany. P. 120-135.

22. Rastaghi, R. An Efficient CCA2-Secure Variant of the McEliece Cryptosystem n the Standard Model. CoRR. Retrieved February 2, 2013. P. 390.

23. Mathew, K. P., Vasant, S., & Rangan, C. P. On Provably Secure Code-based Signature and Signcryption Scheme. Cryptology ePrint Archive, Report. 2012. P. 585.

24. Barreto, P. S. L. M., Misoczki, R., Simplício,M. A. Jr. One-Time Signature Scheme from Syndrome Decoding over Generic Error-Correcting Codes. Journal of Systems and Software, 84(2). 2011. P. 198–204.

25. Becker, A., Joux, A., May, A., & Meurer, A. Decoding random binary linear codes in 2n/20: How 1+1=0 improves information set decoding. In Proceedings of Advances in Cryptology - EUROCRYPT (LNCS) Cambridge, UK. 2012. P. 536.

26. Николайчук Я.М. Теорія джерел інформації. Тернопіль: ТзОВ"Терно-граф", 2010. С. 536

27. Вербіцький О.В. Вступ до криптології. – Львів: ВНТЛ, 1998. С. 248.

28. Ачасова С.М., Бандман О.Л. Коректність паралельних обчислювальних процесів. Новосибирск: Наука. Сиб. отд-ние, 1990. С. 253

29. Гавриленко А.А., Вакуленко А.А., Шкарин І.А. Метод шифрування даних на основі кодів Хеммінга. Вісник Донецького національного університету. Серія: Фізика. Математика. 2016. С. 46-50.

30. Атабеков Р.К., Ібрагімов Р.Р., Султанов І.Р. Метод шифрування даних на основі кодів Хеммінга. Вісник Харківського університету. Т. 24. № 4. С. 1223-1229.

ДОДАТОК А

Шифрування кодом Хеммінга

```
import random
# довжина блока кодування
CHUNK_LENGTH = 8

# перевірка довжини блоку кодування
assert not CHUNK_LENGTH % 8, 'Довжина блока повинна бути кратна 8'

# вираховування контрольних бітів
CHECK_BITS = [i for i in range(1, CHUNK_LENGTH + 1) if not i & (i - 1)]


def chars_to_bin(chars):
    """
    Переформатування символів в бінарний формат
    """
    assert not len(chars) * 8 % CHUNK_LENGTH, 'Довжина кодованих данних повинна бути
кратна довжині блока кодування'
    return ''.join([bin(ord(c))[2:].zfill(8) for c in chars])


def chunk_iterator(text_bin, chunk_size=CHUNK_LENGTH):
    """
   Поблочний вивід бінарних даних
    """
    for i in range(len(text_bin)):
        if not i % chunk_size:
            yield text_bin[i:i + chunk_size]


def get_check_bits_data(value_bin):
    """
    Отримання інформації про контрольні біти з бінарного блока даних
    """
    check_bits_count_map = {k: 0 for k in CHECK_BITS}
    for index, value in enumerate(value_bin, 1):
        if int(value):
            bin_char_list = list(bin(index)[2:].zfill(8))
            bin_char_list.reverse()
            for degree in [2 ** int(i) for i, value in enumerate(bin_char_list) if int(value)]:
                check_bits_count_map[degree] += 1
    check_bits_value_map = {}
    for check_bit, count in check_bits_count_map.items():
        check_bits_value_map[check_bit] = 0 if not count % 2 else 1
    return check_bits_value_map


def set_empty_check_bits(value_bin):
    """
    Додати в бінарний блок "пусті" контрольні біти
```

```
    """
    for bit in CHECK_BITS:
        value_bin = value_bin[:bit - 1] + '0' + value_bin[bit - 1:]
    return value_bin


def set_check_bits(value_bin):
    """
    Встановити значення контрольних бітів
    """
    value_bin = set_empty_check_bits(value_bin)
    check_bits_data = get_check_bits_data(value_bin)
    for check_bit, bit_value in check_bits_data.items():
        value_bin = '{0}{1}{2}'.format(
            value_bin[:check_bit - 1], bit_value, value_bin[check_bit:])
    return value_bin


def get_check_bits(value_bin):
    """
    Отримання інформації про контрольні біти з блока бінарних даних
    """
    check_bits = {}
    for index, value in enumerate(value_bin, 1):
        if index in CHECK_BITS:
            check_bits[index] = int(value)
    return check_bits


def exclude_check_bits(value_bin):
    """
    Виключає інформацію про контрольні біти з блока бінарних даних
    """
    clean_value_bin = ''
    for index, char_bin in enumerate(list(value_bin), 1):
        if index not in CHECK_BITS:
            clean_value_bin += char_bin

    return clean_value_bin


def set_errors(encoded):
    """
    Допускає помилки в блоках двійкових даних
    """
    result = ''
    for chunk in chunk_iterator(encoded, CHUNK_LENGTH + len(CHECK_BITS)):
        num_bit = random.randint(1, len(chunk))
        chunk = '{0}{1}{2}'.format(chunk[:num_bit - 1], int(chunk[num_bit - 1]) ^ 1,
chunk[num_bit:])
        result += (chunk)
    return result
```

```
def check_and_fix_error(encoded_chunk):
    """
    Перевіряє та виправляє помилки в блоці двійкових даних.
    """
    check_bits_encoded = get_check_bits(encoded_chunk)
    check_item = exclude_check_bits(encoded_chunk)
    check_item = set_check_bits(check_item)
    check_bits = get_check_bits(check_item)
    if check_bits_encoded != check_bits:
        invalid_bits = []
        for check_bit_encoded, value in check_bits_encoded.items():
            if check_bits[check_bit_encoded] != value:
                invalid_bits.append(check_bit_encoded)
        num_bit = sum(invalid_bits)
        encoded_chunk = '{0}{1}{2}'.format(
            encoded_chunk[:num_bit - 1],
            int(encoded_chunk[num_bit - 1]) ^ 1,
            encoded_chunk[num_bit:])
    return encoded_chunk


def get_diff_index_list(value_bin1, value_bin2):
    """
    Отримати список індексів різних бітів у двійкових рядках.
    """
    diff_index_list = []
    for index, char_bin_items in enumerate(zip(list(value_bin1), list(value_bin2)), 1):
        if char_bin_items[0] != char_bin_items[1]:
            diff_index_list.append(index)
    return diff_index_list


def encode(source):
    """
    Кодування даних
    """
    text_bin = chars_to_bin(source)
    result = ''
    for chunk_bin in chunk_iterator(text_bin):
        chunk_bin = set_check_bits(chunk_bin)
        result += chunk_bin
    return result


def decode(encoded, fix_errors=True):
    """
    Декодування даних з можливим виправленням помилок.
    """
    decoded_value = ''
    fixed_encoded_list = []
    for encoded_chunk in chunk_iterator(encoded, CHUNK_LENGTH + len(CHECK_BITS)):
        if fix_errors:
```

```
            encoded_chunk = check_and_fix_error(encoded_chunk)
        fixed_encoded_list.append(encoded_chunk)

    clean_chunk_list = []
    for encoded_chunk in fixed_encoded_list:
        encoded_chunk = exclude_check_bits(encoded_chunk)
        clean_chunk_list.append(encoded_chunk)

    for clean_chunk in clean_chunk_list:
        for clean_char in [clean_chunk[i:i + 8] for i in range(len(clean_chunk)) if not i % 8]:
            decoded_value += chr(int(clean_char, 2))
    return decoded_value


if __name__ == '__main__':
    source = input('Вкажіть текст для кодування/декодування:')
    print('Довжина блока кодування: {0}'.format(CHUNK_LENGTH))
    print('Контрольні біти: {0}'.format(CHECK_BITS))
    encoded = encode(source)
    print('Закодування даних: {0}'.format(encoded))
    decoded = decode(encoded)
    print('Результат декодування: {0}'.format(decoded))
    encoded_with_error = set_errors(encoded)
    print('ДОпускаємо помилки в закодованих даних: {0}'.format(encoded_with_error))
    diff_index_list = get_diff_index_list(encoded, encoded_with_error)
    print('Допущені помилки в бітах: {0}'.format(diff_index_list))
    decoded = decode(encoded_with_error, fix_errors=False)
    print('Результат декодування помилкових даних без виправленням помилок:
{0}'.format(decoded))
    decoded = decode(encoded_with_error)
    print('Результат декодування помилкових даних з виправленням помилок:
{0}'.format(decoded))
```

ДОДАТОК Б

Лістинг шифрування кодом Хеммінга

```
# #АСИНХРОННИЙ КОД
import asyncio, random, os, time

# довжина блока кодування
CHUNK_LENGTH = 8

# перевірка довжини блоку кодування
assert not CHUNK_LENGTH % 8, 'Довжина блока повина бути кратна 8'

# вираховування контрольних бітів
CHECK_BITS = [i for i in range(1, CHUNK_LENGTH + 1) if not i & (i - 1)]


async def chars_to_bin(chars):
    """
    Перетворення інформації в бінарний формат

    Args:
        chars: Інформація

    Returns:
        Бінарний формат інформації
    """
    assert not len(chars) * 8 % CHUNK_LENGTH, 'Довжина кодованих данних повинна бути
кратна довжині блока кодування'
    return ''.join([bin(ord(c))[2:].zfill(8) for c in chars])


async def chunk_iterator(text_bin, chunk_size=CHUNK_LENGTH):
    """
    Поблочний вивід бінарних даних
    """
    for i in range(0, len(text_bin), chunk_size):
        yield text_bin[i:i + chunk_size]


async def get_check_bits_data(value_bin):
    """
    Отримання інформації про контрольні біти з бінарного блока даних
    """
    check_bits_count_map = {k: 0 for k in CHECK_BITS}
    for index, value in enumerate(value_bin, 1):
        if int(value):
            bin_char_list = list(bin(index)[2:].zfill(8))
            bin_char_list.reverse()
            for degree in [2 ** int(i) for i, value in enumerate(bin_char_list) if int(value)]:
                check_bits_count_map[degree] += 1
    check_bits_value_map = {}
    for check_bit, count in check_bits_count_map.items():
```

```
        check_bits_value_map[check_bit] = 0 if not count % 2 else 1
    return check_bits_value_map


async def set_empty_check_bits(value_bin):
  """
  Додати в бінарний блок "пусті" контрольні біти
  """
  for bit in CHECK_BITS:
    value_bin = value_bin[:bit - 1] + '0' + value_bin[bit - 1:]
  return value_bin


async def set_check_bits(value_bin):
    """
    Додавання контрольних бітів до блоку даних

    Args:
        value_bin: Блок даних

    Returns:
        Закодований блок даних
    """
    value_bin = await set_empty_check_bits(value_bin)
    check_bits_data = await get_check_bits_data(value_bin)
    for check_bit, bit_value in check_bits_data.items():
        value_bin = await asyncio.get_event_loop().run_in_executor(
            None, lambda: '{0}{1}{2}'.format(
                value_bin[:check_bit - 1], bit_value, value_bin[check_bit:])
        )
    return value_bin


async def get_check_bits(value_bin):
  """
  Отримання інформації про контрольні біти з бінарного блоку даних
  """
  check_bits = {}
  for index, value in enumerate(value_bin, 1):
    if index in CHECK_BITS:
      check_bits[index] = int(value)
  return check_bits


async def exclude_check_bits(value_bin):
    """
     Виключає інформацію про контрольні біти з блока бінарних даних

    """

    clean_value_bin = ''
    for index, char_bin in enumerate(list(value_bin), 1):
      if index not in CHECK_BITS:
```

```
      clean_value_bin += char_bin
    return clean_value_bin

async def set_errors(encoded):
  """
  Допускає помилки в блоках двійкових даних.
  """
  result = ''
  async for chunk in chunk_iterator(encoded, CHUNK_LENGTH + len(CHECK_BITS)):
    num_bit = random.randint(1, len(chunk))
    chunk = await asyncio.get_event_loop().run_in_executor(
      None, lambda: '{0}{1}{2}'.format(chunk[:num_bit - 1], int(chunk[num_bit - 1]) ^ 1,
chunk[num_bit:])
    )
    result += chunk
  return result


async def check_and_fix_error(encoded_chunk):
  """
  Перевіряє та виправляє помилки в блоці двійкових даних.
  """
  check_bits_encoded = await get_check_bits(encoded_chunk)
  check_item = await exclude_check_bits(encoded_chunk)
  check_item = await set_check_bits(check_item)
  check_bits = await get_check_bits(check_item)
  if check_bits_encoded != check_bits:
    invalid_bits = []
    for check_bit_encoded, value in check_bits_encoded.items():
      if check_bits[check_bit_encoded] != value:
        invalid_bits.append(check_bit_encoded)
    num_bit = sum(invalid_bits)
    encoded_chunk = await asyncio.get_event_loop().run_in_executor(
      None, lambda: '{0}{1}{2}'.format(encoded_chunk[:num_bit - 1], int(encoded_chunk[num_bit -
1]) ^ 1,
                                       encoded_chunk[num_bit:])
    )
  return encoded_chunk

async def get_diff_index_list(value_bin1, value_bin2):
    """
    Отримати список індексів різних бітів у двійкових рядках.
    """
    diff_index_list = []
    for index, char_bin_items in enumerate(zip(list(value_bin1), list(value_bin2)), 1):
      if char_bin_items[0] != char_bin_items[1]:
        diff_index_list.append(index)
    return diff_index_list


async def encode(source):
  """
  Кодування даних
```

```
  """
  text_bin = await chars_to_bin(source)
  result = ''
  async for chunk_bin in chunk_iterator(text_bin):
    chunk_bin = await set_check_bits(chunk_bin)
    result += chunk_bin
  return result

async def calc_syndrome(encoded_chunk):
    """
    Розрахунок синдрому

    Args:
        encoded_chunk: Закодований блок даних

    Returns:
        Синдром
    """
    check_bits_encoded = await get_check_bits(encoded_chunk)
    check_item = await exclude_check_bits(encoded_chunk)
    check_item = await set_check_bits(check_item)
    check_bits = await get_check_bits(check_item)
    syndrome = 0
    for i in range(len(CHECK_BITS)):
        syndrome ^= (check_bits_encoded[i] ^ check_bits[i])
    return syndrome
async def get_error_bit_index(syndrome):
    """
    Визначення позиції помилки на основі синдрому

    Args:
        syndrome: Синдром

    Returns:
        Позиція помилки
    """
    error_bit_index = -1
    for i in range(len(CHECK_BITS)):
        if syndrome & (1 << i):
            error_bit_index = i
            break
    return error_bit_index
async def decode(encoded, fix_errors=True):
  """
  Декодування даних з можливим виправленням помилок.
  """
  decoded_value = ''
  fixed_encoded_list = [encoded[i:i + (CHUNK_LENGTH + len(CHECK_BITS))] for i in
                        range(0, len(encoded), CHUNK_LENGTH + len(CHECK_BITS))]

  for encoded_chunk in fixed_encoded_list:
    if fix_errors:
      encoded_chunk = await check_and_fix_error(encoded_chunk)
```

```
    clean_chunk = await exclude_check_bits(encoded_chunk)
    for clean_char in [clean_chunk[i:i + 8] for i in range(0, len(clean_chunk), 8)]:
      decoded_value += chr(int(clean_char, 2))
  return decoded_value

async def test_time(source, num_repeats):
    start_time = time.perf_counter()
    for _ in range(num_repeats):
        encoded = await encode(source)
    encoded_time = time.perf_counter() - start_time

    start_time = time.perf_counter()
    for _ in range(num_repeats):
        decoded = await decode(encoded)
    decoded_time = time.perf_counter() - start_time

    return encoded_time, decoded_time
async def test_accuracy(source, num_tests, error_types):
    errors = []
    for _ in range(num_tests):
        encoded_data = await encode(source)
        for error_type in error_types:
            error_bit = random.randint(1, len(encoded_data))
            corrupted_data = encoded_data[:error_bit] + str(int(encoded_data[error_bit]) ^ 1) +
encoded_data[error_bit + 1:]
            decoded_data = await decode(corrupted_data)
            if source != decoded_data:
                errors.append((error_type, error_bit))
    return errors

async def main():
    # start_time = time.time()
    source = input('Вкажіть текст для кодування/декодування: ')
    print('Довжина блока кодування: {0}'.format(CHUNK_LENGTH))
    print('Контрольні біти: {0}'.format(CHECK_BITS))
    num_tests = int(input('Введіть кількість тестів: '))
    error_types = ['одинична', 'подвійна', 'множинна']

    # start_time = time.perf_counter()
    encoded = await encode(source)
    print('Закодовані дані: {0}'.format(encoded))
    # encoded_time = time.perf_counter() - start_time
    #
    # start_time = time.perf_counter()

    decoded = await decode(encoded)
    print('Результат декодування: {0}'.format(decoded))
    encoded_with_error = await set_errors(encoded)
    print('Допускаємо помилки в закодованих даних: {0}'.format(encoded_with_error))
    diff_index_list = await get_diff_index_list(encoded, encoded_with_error)
    print('Допущені помилки в бітах: {0}'.format(diff_index_list))
    decoded_without_correction = await decode(encoded_with_error, fix_errors=False)
```

```
    print('Результат декодування помилкових даних без виправленням помилок:
{0}'.format(decoded_without_correction))
    decoded_with_correction = await decode(encoded_with_error)
    print('Результат декодування помилкових даних з виправленням помилок:
{0}'.format(decoded_with_correction))

    # decoded_time = time.perf_counter() - start_time
    errors = await test_accuracy(source, num_tests, error_types)
    print('Кількість помилок, які не були виправлені:', len(errors))

    # print('Час кодування: {0}'.format(encoded_time))
    # print('Час декодування: {0}'.format(decoded_time))

# print("Час виконання програми:", end_time - start_time)
end_time = time.time()
if __name__ == "__main__": asyncio.run(main())
```

ДОДАТОК В

Копії публікацій

ЗАХІДНОУКРАЇНСЬКИЙНАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ
ПРИКАРПАТСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ ІМЕНІ ВАСИЛЯ СТЕФАНИКА
НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ ВОДНОГО ГОСПОДАРСТВА ТА ПРИРОДОКОРИСТУВАННЯ
НАЦІОНАЛЬНИЙ ТРАНСПОРТНИЙ УНІВЕРСИТЕТ
НАДВІРНЯНСЬКИЙ КОЛЕДЖ НТУ
ГАЛИЦЬКИЙ КОЛЕДЖ ІМ. В. ЧОРНОВОЛА

Проблемно-наукова міжгалузева конференція

АВТОМАТИЗАЦІЯ ТА КОМП'ЮТЕРНО-ІНТЕГРОВАНІ ТЕХНОЛОГІЇ

(АКІТ – 2023)

23—25 лютого 2023 року

Тернопіль

АВТОМАТИЗАЦІЯ ТА КОМП'ЮТЕРНО-ІНТЕГРОВАНІ ТЕХНОЛОГІЇ

УДК 004.056.5

Цезарук С.Ю.

Західноукраїнський національний університет

КРИПТОГРАФІЧНІ ПРОТОКОЛИ БЕЗПЕКИ МЕРЕЖІ: SSL, TLS, IPSEC

Вступ. В епоху сучасних технологій, де обмін та передача інформацією стали невід'ємною частиною життя, де мережеве середовище стає все більш інтегрованим і відкритим, безпека мережевих з'єднань є критично необхідною та важливо. В наслідок цього, для забезпечення безпеки мережевої комунікації, виникла необхідність у використанні ефективних та надійних криптографічних протоколів. Основними засобами, які забезпечують конфіденційність, цілісність та аутентифікацію даних під час передачі через мережу, є такі криптографічні протоколи безпеки мережі, як SSL (Secure Sockets Layer) , TLS (Transport Layer Security), IPSec (Internet Protocol Security). Саме вони дають можливість зашифрувати передані дані, забезпечувати надійний зв'язок, підтверджувати та перевіряти правомірність сторін.

Мета: Дослідження криптографічних протоколів безпеки мережі: SSL, TLS, IPSec та їх переваги в забезпеченні безпеки мережі.

1. Аналіз криптографічних протоколів безпеки мережі: SSL, TLS, IPSec

SSL – це криптографічний протокол який використовується для захищеного з'єднання між клієнтом і сервером в мережах Інтернет. Протокол забезпечує конфіденційність обміну даними між клієнтом та сервером, що використовують TSP/IP. Використовуються асиметричні алгоритми з відкритим ключем для шифрування даних. При шифруванні з відкритим ключем використовуються два ключі, більш того будь-який з них може використовуватися для шифрування повідомлення, тому якщо використовується один ключ для шифрування, то відповідно для розшифрування потрібно використати другий ключ. Це забезпечує можливість передавати захищені повідомлення, публікуючи відкритий ключ, тоді як приватний ключ залишається конфіденційним.

Протокол SSL пропонує базовий набір заходів безпеки, який використовується протоколами з більш високих рівнів, щоб забезпечити конфіденційність й цілісність каналу комунікацій і аутентифікацію користувачів, які обмінюються даними між двома точками в мережі. Ці цілі досягаються шляхом використання криптографічних алгоритмів для шифрування даних, забезпечують автентичність сервера та перевіряють цілісність даних. Протокол SSL широко використовується в браузерах для захисту конфіденційної інформації під час її передачі таких важливих даних як паролі, кредитні картки, дані авторизації та особиста інформація. Протокол SSL складається трьох фаз, які зображено на рисунку 1.

Перша фаза – це «Привітання», саме в цій фазі клієнт і сервер

встановлюють з'єднання та обмінюються інформацією, задля встановлення параметрів шифрування та аутентифікації.

На другій фазі під назвою «Аутентифікація та обмін ключами» клієнт і сервер взаємно аутентифікуються та обмінюються секретними ключами для шифрування даних, а саме сервер використовує свій приватний ключ для розшифрування секретного ключа, який був надісланий клієнтом, тим самим отримуючи спільний секретний ключ шифрування, після чого клієнт та сервер генерують сесійний ключ, який використовується для шифрування та розшифрування даних під час сеансу з'єднання.

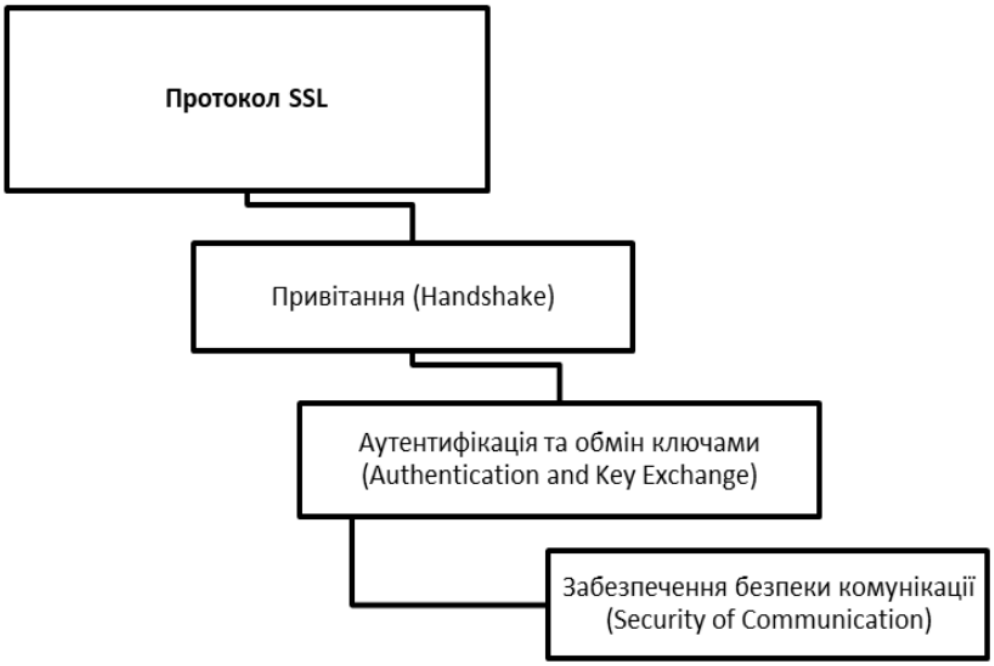

Рисунок 1 – Основні фази протоколу SLL

Третьою фазою є «Забезпечення безпеки комунікації», на цьому етапі клієнт та сервер використовують вже встановлений спільний секретний ключ для шифрування та розшифрування даних, які передаються під час сеансу з'єднання. Також вони можуть використовуватися даний ключ для перевірки цілісності даних за допомогою хеш-функцій.

Протокол TLS є наступною і покращеною версією протоколу SSL. Він також є криптографічним протоколом і призначений для забезпечення конфіденційності й цілісності даних. TLS є ефективнішим та безпечнішим в порівнянні з SSL, оскільки включає в себе потужніші механізми автентифікації повідомлень, генерації ключів та інші алгоритми шифрування.

Наприклад, TLS підтримує попередньо розділені ключі, захищені віддалені паролі та ключі еліптичної кривої, що робить його надійнішим. У відміну від SLL, TLS має два рівні: протокол записів TLS Record Protocol та протокол діалогу TLS Handshake Protocol.

Протокол записів TLS забезпечує конфіденційність даних за допомогою

симетричних алгоритмів шифрування, таких як DES, RC4, та цілісність даних за допомогою геш-функцій SHA-1 або MD5. Протокол діалогу TLS забезпечує цифровий підпис, який базується на підході RSA або DSS. Крім цього, протокол TLS використовується в різних протоколах, таких як HTTPS (захищений протокол передачі гіпертексту), SMTPS (захищений протокол передачі електронної пошти), FTPS (захищений протокол передачі файлів) та інших. Головна мета цих протоколів полягає в забезпеченні безпеки комунікації між клієнтами і серверами через мережу.

IPsec – це узгоджений набір відкритих стандартів, які визначають конкретну специфікацію, в той же час, можуть бути розширені новими протоколами, алгоритмами та функціями мережевої безпеки. Основна мета протоколів IPsec полягає в забезпеченні безпечної передачі даних через IP-мережі, забезпечуючи цілісність, автентичність та конфіденційність. За допомогою IPSec, можна створювати віртуальні приватні мережі (VPN) та захищати трафік між мережами або пристроями.

Протокол IPsec має три фази:

1. Аутентифікація (RFC2402 «IP Authentication Heade»): IPSec контролює цілісність та автентичності пакетів даних в IP-мережах, це може здійснюватися за допомогою попереднього обмінну ключами, сертифікатів, та інших методів аутентифікації.
2. Шифрування: IPSec використовує криптографічні алгоритми для шифрування даних між сторонами, що дозволяє забезпечити конфіденційність інформації, що передається.
3. Цілісність даних: IPSec також забезпечує перевірку цілісності даних, щоб виявити будь-які зміни під час передачі, досягається за допомогою хеш-функцій, які генерують контрольну суму для перевірки цілісності даних.

З метою гарантування цілісності та автентичності пакетів даних, використовуються спеціальні механізми контролю, які ґрунтуються на використанні унікальних кодів цілісності та автентичності, спеціально сформованої надмірності (коди контролю цілісності та автентичності). Ці коди надають додатковий захист при передачі інформації.

IPsec може операційно функціонувати у двох режимах:

- тунельному
- транспортному.

В тунельному режимі IPsec забезпечує захист всіх даних, які передаються між двома мережами, шляхом шифрування та додавання заголовка до кожного пакету даних. В транспортному режимі IPsec, захищає лише дані, які передаються між двома кінцевими пристроями, шляхом шифрування та додавання заголовку до кожного пакету даних.

IPsec має декілька протоколів, що використовуються для забезпечення безпеки: протокол ESP (Encapsulating Security Payload), протокол AH (Authentication Header) та протокол IKE (Internet Key Exchange), деталі приведені на рисунку 2.

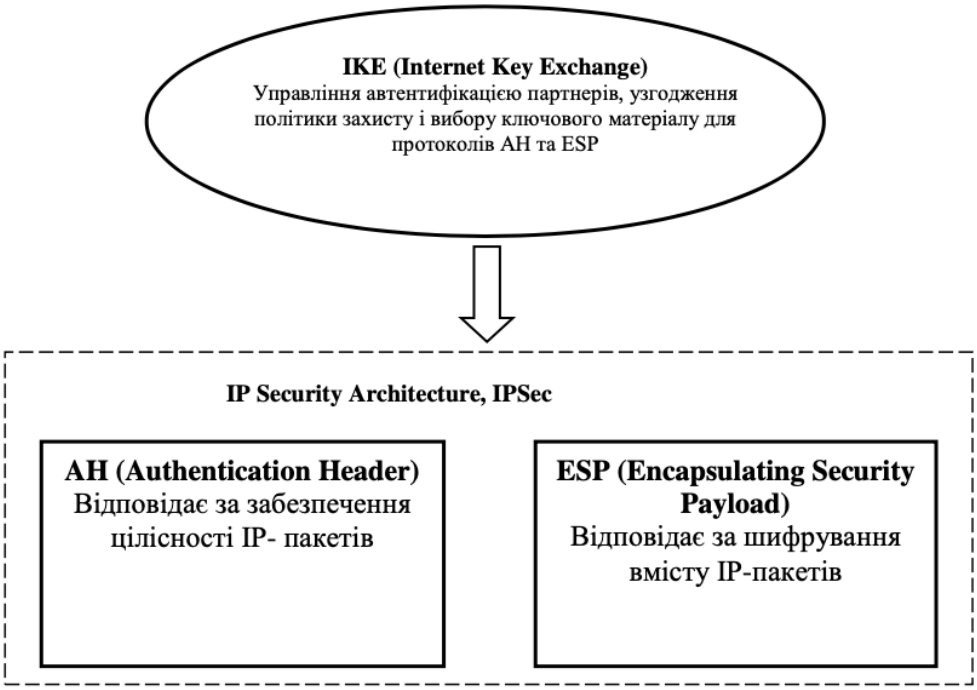

Рисунок 2 – Основні компоненти протоколу IPSec

2. Переваги протоколів в забезпеченні безпеки мережі

Протоколи SSL, TLS і IPSec є важливими засобами забезпечення безпеки мережі, які надають широкий спектр захисних функцій, включаючи шифрування, аутентифікацію та забезпечення цілісності даних. Вони використовуються у багатьох сферах для забезпечення безпеки комунікації та захисту конфіденційності даних.

Більш детальний порівняльний аналіз протоколів за такими характеристиками як: аутентифікація, цілісність даних, конфіденційність даних, відкритий стандарт, використання в різних протоколах, управління ключами, пропускна здатність та рівень безпеки наведено в таблиці 1.

Таблиця 1 – Порівняльний аналіз протоколів безпеки мережі

	SSL	TLS	IPSec
Аутентифікація	Можливість аутентифікації сервера, в обмеженому випадку клієнта	Можливість аутентифікації і клієнта, і сервера	Можливість аутентифікації і сервера, і в залежності від режиму або обидвох, або клієнта
Конфіденційність даних	Забезпечує шифрування даних конфіденційності	Забезпечує шифрування даних конфіденційності	Забезпечує шифрування даних конфіденційності

Цілісність даних	Використовує контрольовані суми для перевірки цілісності даних	Використовує контрольовані суми та Massage Authentication Codes для перевірки цілісності даних	Використовує контрольовані суми та Massage Authentication Codes для перевірки цілісності даних
Відкритий стандарт	Вже застарілий та не є рекомендованим для використання в нових проектах	Відкритий стандарт з широкою підтримкою та активною розробкою на далі	Відкритий стандарт з широкою підтримкою та активною розробкою на далі
Використання в різних протоколах	В протоколі HTTPS	В протоколах: HTTPS, IMAPS, POP3S	В протоколах як: VPN, IPv6, VoIP
Управління ключами	Обмежені можливості	Розширені можливості управління	Розширені можливості управління
Пропускна здатність	Менш ефективна у високонавантажених середовищах	Переважно має кращу ефективність в порівнянні з SSL	Має покращену ефективність в порівнянні з SSL
Рівень безпеки	Вразливий до певних атак, як приклад POODLE атака	Вважається безпечним протоколом з вірною конфігурацією	Вважається безпечним протоколом з вірною конфігурацією

Висновок: Криптографічні протоколи безпеки мережі, а саме SSL, TLS, IPSec, відіграють надважливу роль у забезпеченні безпеки мережевих з'єднань. Вони захищають конфіденційні дані, запобігають несанкціонованому доступу та забезпечують надійну комунікацію. Цілісне розуміння роботи та використання цих протоколів є ключовим елементом для побудови безпечної та надійної мережевої структури.

Перелік використаних джерел.

1. Андрюшина В.С. Огляд криптографічного протоколу SSL /Андрюшина В.С.// Актуальні задачі та досягнення у галузі кібербезпеки: матеріали Всеукраїнської Наук.-практ. Конф. [Кропивницький], 23-25 лист. 2016 р. /М-во освіти і науки України. – Кропивницький, 2016. – С. 72-74.

2. Карачка А.Ф. Технологій захисту інформації. – Тернопіль: ТНЕУ, 2017. – 86 с.

3. Захарченко М.В. Захист інформації від НСД у каналах зв'язку/ М.В. Захарченко, В.В. Топалов, М.С. Русляченко // Інформаційна безпека інформаційно-комунікаційних систем. - 2014. – 216 с.