Міністерство освіти і науки України
Національний університет «Одеська політехніка»
Інститут інформаційної безпеки, радіоелектроніки та телекомунікацій
Кафедра кібербезпеки та програмного забезпечення

Ісаков Дмитро Андрійович,
студент групи РЗ-171

КВАЛІФІКАЦІЙНА РОБОТА МАГІСТРА

Розробка модифікованої криптосистеми McEliece для захисту мережевих ігор

Спеціальність:
125 Кібербезпека

Спеціалізація, освітня програма:
Кібербезпека

Керівник:
Соколов Артем Вікторович,
к.т.н., доцент

Одеса – 2022

Міністерство освіти і науки України
Національний університет «Одеська політехніка»
Інститут інформаційної безпеки, радіоелектроніки та телекомунікацій
Кафедра кібербезпеки та програмного забезпечення

Рівень вищої освіти другий (магістерський)
Спеціальність 125 Кібербезпека
Спеціалізація, освітня програма Кібербезпека

ЗАТВЕРДЖУЮ
Завідувач кафедри КБПЗ

_____
д.т.н.,проф. А.А.Кобозєва
____ _____2022р.

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ

Ісакову Дмитру Андрійовичу

1. Тема роботи: *Розробка модифікованої криптосистеми McEliece для захисту мережевих ігор*.
керівник роботи *Соколов Артем Вікторович, к. ф.-м. н., доцент,* затверджені наказом ректора Національного університету «Одеська політехніка» від „____”_____ 20___ р. №_____ .
2. Зміст роботи: *аналіз проблемної області, постановка задачі, модифікація криптосистеми McEliece з використанням четвіркових кодів Гемінга, розробка мережевої гри, вбудова модифікованої криптосистеми McEliece в гру.*
3. Перелік ілюстративного матеріалу: *блок схеми алгоритму, рисунки інтерфейсу програмного продукту, слайди презентації.*

4. Консультанти розділів роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		Завдання видав	Завдання прийняв

5. Дата видачі завдання “_____”_____2022 р.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів кваліфікаційної роботи	Строк виконання	Примітка
1	*Аналіз джерел з теми кваліфікаційної роботи*	*28-08-2022*	*виконано*
2	*Аналіз криптосистеми McEliece з використанням кодів Гемінга*	*03-09-2022*	*виконано*
3	*Модифікація криптосистеми McEliece на основі четвіркових кодів Гемінга*	*20-09-2022*	*виконано*
4	*Розробка програмного забезпечення*	*30-09-2022*	*виконано*
5	*Підготовка тексту роботи*	*01-11-2022*	*виконано*
6	*Підготовка презентації та доповіді*	*12-11-2022*	*виконано*
7	*Попередній захист*	*26-11-2022*	*виконано*
8	*Нормоконтроль, рецензування*	*15-12-2022*	*виконано*
9	*Допуск до захисту*	*19-12-2022*	*виконано*

Здобувач вищої освіти _____ *Ісаков Д.А.*

Керівник роботи _____ *Соколов А.В.*

АНОТАЦІЯ

Кваліфікаційна робота на тему «Розробка модифікованої криптосистеми McEliece для захисту мережевих ігор» на здобуття другого (магістерського) рівня вищої освіти за спеціальністю 125 Кібербезпека, спеціалізація, освітня програма: Кібербезпека, містить 17 рисунків, 1 таблиця, 1 додаток, 24 літературних джерела за переліком посилань. Робота виконана на 69 сторінках загального тексту і 46 сторінках основного тексту.

У результаті виконання кваліфікаційної роботи розроблено модифіковану криптосистему McEliece з використанням недвійкових кодів Гемінґа, мережеву гру, в яку було впроваджено модифіковану криптосистему McEliece. Проведено атаку на мережеву гру з перехопленням пакету. Зроблено висновок, що за рахунок зменшення розміру ключа, а саме це дозволило підвищити якість, ефективність та скоротити обчислювальні витрати, час обчислень.

Запропонована криптосистема може бути рекомендована до практичного використання у застосунках, що потребують великої швидкодії (наприклад, мобільних пристроях, пристроях IoT, вбудовуваних системах), а також значної криптостійкості, у тому числі, захищеності від перспективних квантових атак.

ШИФРУВАННЯ, АСИМЕТРИЧНЕ ШИФРУВАННЯ, PARITY CHECK, ЛІНІЙНІ КОДИ, КОДИ ГЕМЕНҐА, КРИПТОСИСТЕМА, MCELIECE, UNITY, МЕРЕЖЕВА ГРА.

ANNOTATION

Qualification work on the topic "Development of a modified McEliece cryptosystem for the protection of network games" for obtaining the second (master's) level of higher education in the specialty 125 Cybersecurity, specialization, educational program: Cybersecurity, contains 17 figures, 1 table, 1 appendix, 24 literary sources as listed links The work is completed on 69 pages of general text and 46 pages of the main text.

As a result of the qualification work, a modified McEliece cryptosystem was developed using non-binary Hamming codes, a network game in which the modified McEliece cryptosystem was implemented. A network game has been attacked with packet interception. It was concluded that due to the reduction of the size of the key, namely, it allowed to increase the quality, efficiency and reduce the computational costs and calculation time.

The proposed cryptosystem can be recommended for practical use in applications that require high speed (for example, mobile devices, IoT devices, embedded systems), as well as significant cryptoresistance, including protection against prospective quantum attacks.

ENCRYPTION, ASYMMETRIC ENCRYPTION, PARITY CHECK, LINEAR CODES, HAMMING CODE, CRYPTOSYSTEM, MCELIECE, UNITY, NETWORK GAME.

ЗМІСТ

ВСТУП

В даний час відбувається найбільша революція в іграх з моменту появи домашніх комп'ютерів: онлайн-ігри, які можна грати через модем у мережі Інтернет. Зараз багато компаній наполягають на тому, що кожна гра, яку вони розробляють, повинна мати онлайн-компонент. Найамбітніші онлайн-ігри – це постійні світи, які занурюють сотні гравців в єдине загальне середовище. В іграх для однієї чи двох осіб «чітерство» є незначною проблемою, оскільки воно торкається лише однієї чи двох осіб одночасно. Однак один шахрай в онлайн-грі може вплинути на тисячі людей та мати довгострокові наслідки.

З усіх проблем, з якими має зіткнутися розробник онлайн-ігор, безпека спочатку може здатися очевидною проблемою. Проте це важливо для підтримки працездатності системи. Помилка проектування безпеки гри – це перший крок до краху. У сценарії, коли клієнти платять за постійний доступ до гри, дозволити хакерам працювати безкоштовно – це вірний спосіб втратити більшу частину платоспроможної бази клієнтів. Необхідно зробити перший крок до захисту ігор, а саме використовувати більш швидкий асиметричний криптоалгоритм, який буде гарантувати крипостійкість і швидкість.

Асиметричні криптоалгоритми є ключовим елементом сучасних систем захисту інформації, від ефективності якого високою мірою залежить їх стійкість і швидкодія. На сьогоднішній день саме застосування асиметричної криптографії дозволяє вирішувати такі базові завдання систем захисту інформації як розподіл ключів, автентифікація користувачів, підтвердження авторства повідомлень, захист програмного забезпечення тощо, без забезпечення яких повноцінне функціонування практично будь-якої системи захисту є неможливим.

Як показують провідні сучасні дослідження, одним з потужних рішень постквантової криптографії, яке характеризується значним зменшенням обчислювальної складності (а, отже, потенційно підходить до застосування на ресурсообмежених пристроях), є крипто-кодові конструкції, серед яких найуживанішим рішенням є криптосистема МакЕліса. Зазначена криптосистема

характеризується нижчим рівнем обчислювальної складності у порівнянні із криптоалгоритмами RSA і Ель-Гамаля, значно вищим рівнем стійкості до атак квантового криптоаналізу. Тим не менш, незважаючи на свої переваги, криптосистема МакЕліса у своєму класичному розумінні характеризується великим розміром ключової інформації, яка необхідна для досягнення достатнього рівня криптографічної стійкості, що значно обмежує її використання на практиці.

Метою даної роботи зниження розміру ключової інформації у криптосистемі McEliece шляхом застосування четвіркових кодів Гемінга.

Для досягнення даної мети були поставлені наступні задачі:

– виконати огляд існуючої літератури;
– розробити сімейства четвіркових кодів Гемінга та дослідити їх властивості;
– модифікувати криптосистему McEliece на основі недвійкових кодів;
– реалізувати програмними засобами модифіковану криптосистему McEliece;
– вбудувати модифіковану криптосистему McEliece в мережеву гру.

Об'єкт дослідження — процес підвищення ефективності шифрування пакетів в мережевих іграх.

Предмет дослідження — модель шифрування пакетів з використанням криптосистеми McEliece на основі четвіркових кодів Гемінга.

Наукова новизна отриманих результатів. Вперше розроблено модифіковану криптографічну систему McEliece на основі четвіркових кодів Гемінга яка, на відміну від існуючих аналогів характеризується меншою довжиною відкритого ключа при збереженні стійкості до потенційних атак квантового криптоаналізу.

Практичне значення отриманих результатів полягає у їх доведенні до конкретних алгоритмів, які були вбудовані у онлайн гру, де показали зниження необхідного для зберігання та транспортування відкритого ключа обсягу пам'яті у 4.75 рази.

За тематикою кваліфікаційної магістерської роботи була опублікована стаття «McEliece cryptosystem based on quaternary Hamming codes» [1].

1 АНАЛІЗ ЛІТЕРАТУРИ ПО КРИПТОСИСТЕМІ MCELIECE

1.1 Асиметричне шифрування: призначення та види

Шифрування – це процес перетворення інформації на код або шифр, так що тільки авторизовані сторони зможуть розшифрувати та зрозуміти її. Зашифрована інформація повинна бути нерозбірливою для неуповноважених сторін.

Асиметричне шифрування – це розширена форма криптографії, в якій ключ, який використовується для шифрування даних, відрізняється від ключа, який використовується для їх розшифрування на стороні взаємодії. При асиметричному шифруванні будь-який користувач може використовувати відкритий ключ одержувача для шифрування повідомлення. Однак після шифрування лише власник закритого ключа передбачуваного одержувача може розшифрувати його.

Симетричне шифрування – найпростіший тип шифрування. У цій формі криптографії ключ, який використовується для перетворення повідомлення з звичайного тексту на зашифрований формат, є тим же ключем, який необхідно використовувати для розшифрування повідомлення назад в формат, що читається. Іншими словами, відправник та одержувач спільно використовують один шифр, також відомий як секретний ключ або закритий ключ, для шифрування та розшифрування інформації. Оскільки процес шифрування та дешифрування абсолютно однаковий на обох сторонах взаємодії, ця форма криптографії називається симетричним шифруванням.

Якщо схилятися на попередню мою статтю та дипломну роботу автентифіковане шифрування з використанням генератора псевдовипадкових ключових послідовностей (ГПКП) [2, 3], то це шифрування не підходить для мережевих ігор, тим що шифрування даних відбувається дуже повільно і необхідні витрати в обчислюванні, тому використовувати таке шифрування необхідно тільки тоді, коли потрібно зашифрувати якісь важливі дані.

Однією із найбільших проблем для симетричного шифрування є проблема обміну закритими ключами. Безпека зашифрованих повідомлень залежить від того, що лише авторизовані користувачі мають доступ до закритого ключа, необхідного для розшифрування. Однак це є проблемою. Як дві або більше сторони можуть домовитися про секретний ключ, якщо вони не можуть зустрітися віч-на-віч?

До появи Інтернету це означало, що хтось, наприклад, довірений кур'єр, відповідав за доставку закритого ключа від одного боку до іншого. Звісно, сам закритий ключ не можна було зашифрувати. Без закритого ключа будь-яка форма шифрування буде нерозбірливою для одержувача, навіть якщо він буде близьким союзником. В результаті довірений кур'єр повинен передати закритий ключ у текстовому форматі. Це давало можливість неавторизованій стороні отримати доступ до закритого ключа. Зрештою, зловмисники можуть підкупити чи змусити кур'єра видати закритий ключ.

У сучасному цифровому світі цієї проблеми більше не існує завдяки більш просунутим криптографічним методам, таким як асиметричне шифрування [4] та криптографія з відкритим ключем. Просто важливо відзначити, що симетричне шифрування, незалежно від того, наскільки складними можуть бути закритий ключ та шифрування, не вважається безпечним через проблему обміну закритим ключем.

Кожна людина, яка використовує асиметричне шифрування має унікальний закритий ключ. Щойно закритий ключ встановлено, він використовується визначення так званого відкритого ключа. Кожному закритому ключу відповідає один відкритий ключ, звідси і фраза «пара закритий-відкритий ключ».

На відміну від симетричного шифрування, обмін закритими ключами не потрібен. Фактично безпека асиметричного шифрування залежить від того, що кожна людина є єдиною людиною, яка має доступ до свого особистого ключа. Відкритий ключ, з іншого боку, може бути переданий будь-кому. Хоча кожен закритий ключ використовується для отримання лише одного відкритого ключа, неможливо визначити закритий ключ, знаючи відкритий ключ.

У більш старих методах асиметричного шифрування, таких як обмін ключами Діффі-Хеллмана, двоє людей обмінюються відкритими ключами і використовують їх для отримання загального закритого ключа. Закритий ключ кожної людини, як і раніше, залишається для них секретом, і, як і раніше, безпечно ділитися відкритим ключем з ким завгодно.

У нових методах асиметричного шифрування, таких як криптографія на еліптичних кривих (ECC) і алгоритми цифрового підпису, для шифрування інформації використовуються самі відкриті ключі. Тоді тільки передбачуваний одержувач зможе розшифрувати цю інформацію за допомогою відповідного закритого ключа. Відкриті ключі по суті діють як відкриті кодові замки, вам не потрібно знати комбінацію, щоб використовувати її для блокування або шифрування повідомлення. Після того, як замок замкнений, тільки той, хто знає комбінацію, тобто закритий ключ, може відкрити замок та отримати вміст.

В обох випадках проблеми обміну закритими ключами можна уникнути за допомогою асиметричного шифрування. Двом сторонам більше не потрібно надсилати закритий ключ у вигляді простого тексту.

Відкритий ключ – це криптографічний ключ, який може використовувати будь-яка людина для шифрування повідомлення, щоб тільки передбачуваний одержувач міг розшифрувати повідомлення за допомогою свого особистого ключа. Закритий ключ, який також називають закритим ключем, надається лише тому, хто його створив.

Коли хтось хоче надіслати зашифроване повідомлення, він може отримати відкритий ключ передбачуваного одержувача з загальнодоступного каталогу і використовувати його для шифрування повідомлення перед його відправкою. Отримувач повідомлення може використовувати закритий ключ для розшифровування повідомлення (рис 1.1).

Рисунок 1.1 – Принцип роботи асиметричної криптосистеми

Якщо відправник шифрує повідомлення за допомогою свого закритого ключа, повідомлення може бути розшифроване лише за допомогою відкритого ключа цього відправника, таким чином автентифікуючи відправника. Ці процеси шифрування та дешифрування відбуваються автоматично; користувачам не потрібно фізично блокувати та розблокувати повідомлення.

Багато протоколів засновані на асиметричній криптографії, включаючи протоколи безпеки транспортного рівня (TLS) та рівня захищених сокетів (SSL), які уможливлюють HTTPS.

Процес шифрування також використовується в програмах, яким необхідно встановити безпечне з'єднання в незахищеній мережі, наприклад, у браузерах через Інтернет, або в яких потрібна перевірка цифрового підпису.

Основною перевагою асиметричного шифрування є підвищена безпека даних. Це найбільш безпечний процес шифрування, оскільки він не вимагає від користувача розкривати або передавати свій закритий ключ, що знижує ймовірність того, що кіберзлочинець виявить закритий ключ користувача під час передачі.

Асиметричне шифрування використовує математично пов'язану пару ключів для шифрування та дешифрування: відкритий ключ та закритий ключ. Якщо для шифрування використовується відкритий ключ, для розшифровки використовується відповідний закритий ключ. Якщо для шифрування

використовується закритий ключ, використовується відповідний відкритий ключ для розшифровки.

Двома учасниками робочого процесу асиметричного шифрування є відправник та одержувач. У кожного своя пара відкритих та закритих ключів. Спочатку відправник одержує відкритий ключ одержувача. Потім відкрите текстове повідомлення шифрується відправником за допомогою відкритого ключа одержувача. Це створює зашифрований текст. Зашифрований текст відправляється одержувачу, який розшифровує його своїм закритим ключем, повертаючи його до розбірливого відкритого тексту. До 1970-х існувало лише симетричне шифрування. Навіть коли впроваджувані криптографічні методи були досконалішими, ніж простий шифр Цезаря, проблема обміну закритими ключами робила безпечний зв'язок надзвичайно ненадійною. У тих випадках, коли закритий ключ мав бути поширеним серед великих груп людей по всьому світу, було ще складніше запобігти перехопленню закритого ключа неавторизованими сторонами.

Крім того, якщо закритий ключ був скомпрометований, всім авторизованим сторонам потрібно буде надіслати новий ключ. Це знову означало, що проблема лишилася.

Криптографи зрозуміли, що для шифрування та дешифрування повідомлень необхідне найкраще рішення. Це призвело до того, що криптографи почали розробляти нову форму шифрування, яка називається асиметричним шифруванням.

Через односторонній характер функції шифрування один відправник не може прочитати повідомлення іншого відправника, навіть якщо кожен з них має відкритий ключ одержувача.

Алгоритми асиметричного шифрування, також відомі як схеми цифрового підпису, становлять основу безпечного зв'язку з використанням відкритих та закритих ключів. Порівняємо кілька популярних алгоритмів, що мають історичне або актуальне значення в епоху сучасного шифрування.

Приклади асиметричного шифрування [5] включають:

– Рівест Шамір Адлеман (ПАР);

– Стандарт цифрового підпису (DSS), який включає алгоритм цифрового підпису (DSA);

– Криптографія на еліптичних кривих (ECC);

– метод обміну Діффі-Хеллмана;

– Протокол TLS/SSL.

Опублікований у 1977 році, RSA є одним із найстаріших прикладів асиметричного шифрування. Розроблене Роном Рівестом, Аді Шаміром та Леонардом Адлеманом, шифрування RSA генерує відкритий ключ шляхом перемноження двох великих випадкових простих чисел і, використовуючи ці ж прості числа, генерує закритий ключ. Звідти походить стандартне асиметричне шифрування: інформація шифрується за допомогою відкритого ключа та розшифровується за допомогою закритого ключа.

DSS, який включає алгоритм цифрового підпису (DSA), є чудовим прикладом асиметричної автентифікації цифрового підпису. Закритий ключ відправника використовується для цифрового підпису повідомлення або файлу, а одержувач використовує відповідний відкритий ключ відправника, щоб підтвердити, що підпис походить від правильного відправника, а не з підозрілого чи неавторизованого джерела.

ECC – це альтернатива RSA, в якій використовуються ключі меншого розміру та математичні еліптичні криві для виконання асиметричного шифрування. Він часто використовується для цифрового підпису криптовалютних транзакцій; насправді, популярна криптовалюта Біткойн використовує ECC – алгоритм цифрового підпису на еліптичних кривих (ECDSA), а точніше – для цифрового підпису транзакцій та забезпечення того, щоб кошти витрачалися лише авторизованими користувачами.

ECC набагато швидше, ніж RSA, з точки зору генерації ключів та підписів, і багато хто вважає його майбутнім асиметричного шифрування, в основному для веб-трафіку та криптовалюти, але також і для інших додатків.

Діффі-Хеллман, один з найбільших проривів у криптографії, є методом обміну ключами, в якій дві сторони, які ніколи не зустрічалися, можуть використовувати для обміну парами відкритого та закритого ключів загальнодоступними, небезпечними каналами зв'язку.

До Діффі-Хеллмана дві сторони, які прагнуть зашифрувати свої повідомлення один з одним, мали попередньо фізично обмінятися ключами шифрування, щоб обидві сторони могли розшифрувати зашифровані повідомлення один одного.

Діффі-Хеллман зробив так, щоб цими ключами можна було безпечно обмінюватися загальнодоступними каналами зв'язку, де треті сторони зазвичай отримують конфіденційну інформацію та ключі шифрування, наприклад відома атака MITM.

TLS/SSL використовує асиметричне шифрування для встановлення безпечного сеансу клієнт-сервер, тоді як клієнт та сервер генерують симетричні ключі шифрування. Це відомо як рукостискання TLS. Після завершення рукостискання TLS сеансові ключі клієнт-сервер використовуються для шифрування інформації, яка обмінюються в цьому сеансі.

Незважаючи на проривний характер зазначених криптоалгоритмів, та їх безперечно високе значення у завданнях забезпечення безпеки сучасних інформаційних технологій, вони не позбавлені й суттєвих недоліків, серед яких: висока обчислювальна складність алгоритмів шифрування/розшифрування, жорсткі вимоги до ключової інформації, вразливість перед перспективними атаками квантового криптоаналізу [3].

Отже, ознайомившись з асиметричним шифруванням та його видами, а також визначивши переваги над симетричним, перейдемо до кодів, які виправляють помилки, адже існує асиметрична криптосистема, яка дозволяє не тільки шифрувати і розшифрувати повідомлення, а й виправляти та виявляти бітові помилки.

1.2 Двійкові коди Гемінґа

Лінійний блоковий код – це тип коду з виправленням помилок, в якому фактичні інформаційні біти лінійно комбінуються з бітами контролю парності, щоб генерувати лінійне кодове слово, яке передається каналом. Ще одним важливим типом коду з виправленням помилок є код згортки [6].

У методі лінійного блочного коду повне повідомлення ділиться на блоки, і ці блоки поєднуються з надмірними бітами, щоб мати справу з виявленням та виправленням помилок. Щоразу, коли справа доходить до передачі даних однією з основних проблем, пов'язаних з передачею даних, є внесення помилок. Повідомлення не можливо декодувати при помилках, але за рахунок лінійних блокових кодів все можливо. Ці помилки мають бути усунені, оскільки їх наявність змінює фактичний зміст інформації. Єдиною метою використання кодів виправлення помилок є полегшення виявлення та виправлення даних, що передаються каналом зв'язку, шляхом введення надмірності в фактичні біти повідомлення.

Коди з виправленням помилок додають надмірність (непотрібні біти) до фактичних біт повідомлення, тому одержувач повинен правильно декодувати фактичний сигнал повідомлення із закодованого сигналу повідомлення.

Зверніть увагу, що код виправлення помилок застосовується до цифрового сигналу, а не аналогового сигналу. Простіше кажучи, сигнали мають форму 1 та 0, тобто двійкові.

Існує два способи кодування з контролем помилок:

- лінійне блочне кодування;

- згорткове кодування.

У цій димпломній роботі розглядається лінійне блочне кодування.

Тому вже ясно, що кодування може додавати біти кодування потоку даних, що виконується на стороні передавача.

Декодування — це відновлення фактичного потоку даних із закодованого потоку даних, яке виконується на стороні, що приймає. Але кодеки кодують і декодують.

При блочному кодуванні повні біти повідомлення поділяються на блоки, де кожен блок містить однакову кількість бітів.

Припустимо, кожен блок містить k бітів, і кожні k бітів блоку визначають слово даних. Отже, загальна кількість слів даних буде 2^k. У цей момент ми не враховували жодних надмірностей, тому ми маємо лише фактичний бітовий потік повідомлення, перетворений на слова даних.

Тепер для виконання кодування слова даних кодуються як кодові слова, що мають n бітів.

Нещодавно згадувалося, що блок має k біт, і після кодування в кожному блоці буде n біт (звісно, $n > k$), і ці n біт будуть передаватися каналом. Хоча додаткові біти $n - k$ не є бітами повідомлення, оскільки вони називаються бітами парності, але під час передачі біти парності діють так, ніби вони є частиною бітів повідомлення.

Отже, структура кодового слово представлено на (рис. 1.2):

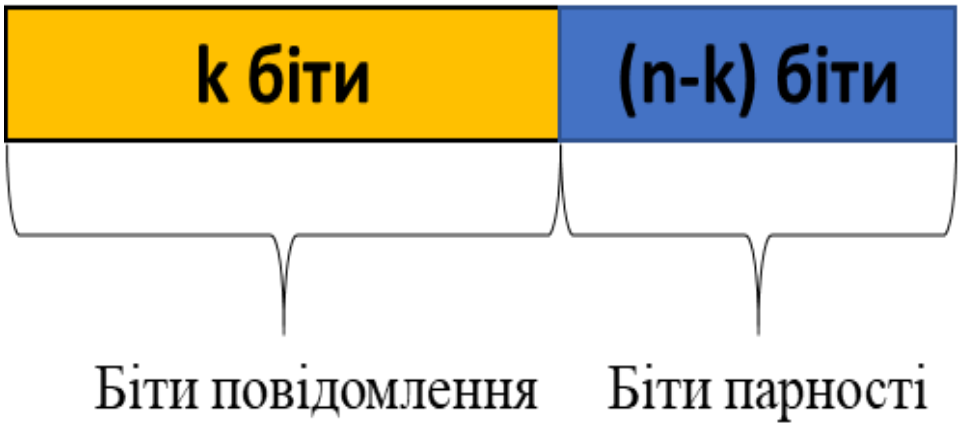

Рисунок 1.2 – Структура кодового слова

Отже, можливі кодові слова будуть 2^n, з яких 2^k містять слова даних. Під час передачі, якщо вводяться помилки, швидше за все, допустимі кодові слова

замінять надлишковими словами, які можуть бути виявлені приймачем як помилка.

Що стосується термінів, кодових слів та слів даних, термін швидкість кодування використовується як відношення бітів слова даних до біт кодового слова. Таким чином, представляється як:

$$r_c = \frac{k}{n}, \tag{1.1}$$

Код представляється як (n, k). Розглянемо приклад, коли n=6 і k=3, тоді код буде (6,3), що вказує на те, що слово даних із 3 бітів замінюється кодовим словом із 6 бітів.

Тепер розглянемо матриці, щоб краще зрозуміти лінійні блокові коди. Припустимо, ми маємо слово даних 101, представлене вектором-рядком d.

$$d = [101], \tag{1.2}$$

Кодовим словом для цього з підходом, що обговорюється вище, з парною парністю буде 1010, задане вектором-рядком c.

$$c = [1010], \tag{1.3}$$

Як правило, генераторна матриця G використовується для створення кодового слова із слова даних. Відношення між c, d та G визначається як:

$$c = dG, \tag{1.4}$$

Для коду (6,3) буде 6 бітів у кодовому слові та 3 біти у слові даних. У систематичному коді найбільш поширене розташування у тому, що слово даних перебуває на початку кодового слова. Щоб отримати це, підматриця ідентичності використовується у поєднанні з підматрицею парності, і, використовуючи відношення dG, можемо мати:

$$C = SGP = [1\ 0\ 1]\begin{bmatrix}1 & 0 & 0 & 1 & 0 & 1\\ 0 & 1 & 0 & 0 & 1 & 1\\ 0 & 0 & 1 & 1 & 1 & 0\end{bmatrix} = [1\ 0\ 1\ 0\ 1\ 1], \tag{1.5}$$

Згідно (2.12), $C = [1\ 0\ 1\ 0\ 1\ 1]$, інформаційна біти присутні, як перші 3 біти отриманого кодового слова, а решта 3 є бітами парності.

Для декодування фактичного слова даних з отриманого кодового слова приймачі виконується транспонування матриці парності.

$$P^T = \begin{bmatrix} 1 & 0 & 1 \\ 0 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix} \quad (1.6)$$

Крім того, матриця контролю парності H виходить комбінацією транспонування матриці парності та одиничної матриці.

$$H = \begin{bmatrix} 1 & 0 & 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 0 & 0 & 1 \end{bmatrix}, \quad (1.7)$$

Для матриці H декодер може аналізувати біт парності з отриманих кодових слів. Тут загальна кількість рядків у наведеній вище матриці представляє кількість бітів парності, тобто $n - k$, а кількість стовпців показує кількість бітів у кодовому слові, тобто n. У цьому конкретному прикладі кількість рядків дорівнює 3, що становить всього 3 біти парності, а кількість стовпців тут дорівнює 6, показуючи n, тобто загальна кількість бітів у кодовому слові.

Фундаментальна властивість кодових матриць свідчить, що

$$GH^T = 0, \quad (1.8)$$

Для отриманого кодового слова перевірка виправлення досягається шляхом множення коду H^T.

$$cH^T = 0, \quad (1.21)$$

Як ми це знаємо,

$$c = dG, \quad (1.21)$$

Отже, підставивши dG замість c у передостаннє рівняння, ми отримаємо,

$$dGH^T = 0, \quad (1.21)$$

Якщо це перемноження не дорівнює 0, це свідчить про наявність помилки. Як правило, називається синдром дається як:

$$s = cH^T, \quad (1.21)$$

Відношення між переданим та прийнятим кодовим словом записується таким чином, що

$$C_R = C_T + e, \quad (1.21)$$

Це означає, що прийняте кодове слово обов'язково має дорівнювати сумі фактично переданого кодового слова і вектора помилки.

В інформатиці та телекомунікаціях коди Гемінґа є сімейством лінійних кодів з виправленням помилок [7]. Коди Гемінґа можуть виявляти однобітові та двобітові помилки або виправляти однобітові помилки без виявлення невиправлених помилок. Навпаки, простий код парності не може виправляти помилки і може виявляти лише непарну кількість помилкових бітів.

Коди Гемінґа є досконалими кодами, тобто вони досягають максимально можливої швидкості для кодів з їх довжиною блоку та мінімальною відстанню, що дорівнює трьом. Річард У. Гемінґ винайшов коди Гемінґа в 1950 як спосіб автоматичного виправлення помилок, допущених зчитувачами перфокарт. У своїй оригінальній статті Гемінґ розробив свою загальну ідею, але спеціально зосередився на коді Гемінґа (7,4), який додає три біти парності до чотирьох біт даних [8].

З математичної точки зору коди Гемінґа є класом двійкових лінійних кодів. Для кожного цілого числа r ≥ 2 існує кодове слово з довжиною блоку $n = 2r - 1$ та довжиною повідомлення $k = 2r - r - 1$.

Отже, швидкість кодів Гемінґа дорівнює $R = k / n = 1 - r / (2r - 1)$, що є максимальним для кодів з мінімальною відстанню, що дорівнює трьом (тобто мінімальна кількість бітових змін, необхідних для переходу від будь-якого кодового слова до будь-якого іншого кодового слова, дорівнює трьом) і довжиною блоку $2r - 1$.

Як вже говорилось існують певні сімейства кодів Гемінґа, які задаються формулами. Перебравши всі задані r (перевірочні біти) від 1 до 8, отримаємо всі можливі коди Гемінґа (табл. 1.1).

Таблиця 1.1 – Коди Гемінґа при різних r

Біти парності	Всього біт	Біти даних	Сімейства	Швидкість
2	3	1	(3,1)	1/3 ≈ 0,333
3	7	4	(7,4)	4/7 ≈ 0,571
4	15	11	(15,11)	11/15 ≈ 0,733

5	31	26	(31,26)	26/31 ≈ 0,839
6	63	57	(63,57)	57/64 ≈ 0,905
7	127	120	(127,120)	120/127 ≈ 0,945
8	255	247	(255,247)	247/255 ≈ 0,969
…				
r	$n = 2^r - 1$	$k = 2^r - r - 1$	(n, k)	k/n

Парність — один із найпростіших кодів виявлення помилок. Додає біт, що вказує, чи є кількість одиниць (бітових позицій зі значенням 1) попередніх даних парним або непарним. Якщо під час передачі зміниться непарна кількість бітів, повідомлення змінить парність, і в цей момент може бути виявлено помилку. Однак модифікований біт може бути самим бітом парності. Найбільш поширена угода полягає в тому, що значення парності 1 вказує на непарні одиниці даних, а значення парності 0 вказує на парні одиниці. Якщо кількість змінених бітів парна, контрольний біт активується і помилка не виявляється.

Перевірочна матриця коду Гемінґа [9] будується шляхом перерахування всіх ненульових стовпців довжини r, що означає, що двійковий код коду Гемінґа є укороченим кодом Адамара. Матриця контролю парності має тим властивістю, що будь-які два стовпці попарно лінійно незалежні. Рядки матриці перевірки на парність є перевірками на парність кодового слова коду. Тобто він показує, як лінійна комбінація певних цифр кожного кодового слова дорівнює нулю.

Необхідно звернути увагу, що виявлення помилок та виправлення помилок різні. Виявлення помилок зазвичай набагато простіше та ефективніше, але виправлення помилок складніше та менш ефективно. Крім того, виявлення та виправлення помилок не завжди працює. Завжди є невеликий шанс зіштовхнутися із ситуаціями, коли виявлення та виправлення помилок не призначені. Через обмежену надмірність, яку коди Гемінґа додають до даних, коди Гемінґа можуть виявляти і виправляти помилки тільки при низькій частоті помилок.

Традиційні коди Гемінґа є кодами (7, 4), що кодують чотири біти даних в семибітові блоки (кодове слово Гемінґа). Додаткові три біти є бітами парності. Кожен із трьох бітів парності відповідає парності для трьох із чотирьох бітів даних, і жодні два біти парності не відповідають одним і тим же трьом бітам даних. Усі біти парності є парними.

Матриці G та H для лінійних блокових кодів повинні задовольняти $\mathrm{HG^T} = 0$, матриця з усіма нулями. Так як $[7, 4, 3] = [n, k, d] = [2^r - 1, 2^r - 1 - r, 3]$. Матриця перевірки на парність H коду Гемінґа будується шляхом перерахування всіх стовпців довжини m, які попарно незалежні.

Таким чином, H є матрицею, ліва частина якої складається з усіх ненульових n-кортежів, де порядок n-кортежів у стовпцях матриці не має значення. Права частина – це просто $(n - k)$ – одинична матриця.

Таким чином G може бути отримана з H шляхом транспонування лівої частини H з одиничною k - одиничною матрицею в лівій частині G.

Матриця G і матриця перевірки на парність H мають вигляд:

$$\mathrm{G} = \begin{pmatrix} 1 & 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 \end{pmatrix}_{4,7} \tag{1.12}$$

$$\mathrm{H} = \begin{pmatrix} 1 & 1 & 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 1 & 0 & 0 & 1 \end{pmatrix}_{3,7} \tag{1.13}$$

З наведеної вище матриці ми маємо $2^k = 2^4 = 16$ кодових слів. Нехай, $\vec{a} = [1\ 0\ 1\ 1]$, використовуючи матрицю G маємо,

$$\vec{x} = \vec{a} \cdot G = (1 \quad 0 \quad 1 \quad 1) \begin{pmatrix} 1 & 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 \end{pmatrix} = (1\ 0\ 1\ 1\ 0\ 1\ \ 0) \tag{1.14}$$

Отже, розібравшись детально в лінійних кодах, а тобто з кодами Гемінґа, перейдемо до криптосистеми McEliece, яка використовує коди, для виправлення помилок.

1.3 Криптосистема McEliece

У криптографії криптосистема McEliece є алгоритмом асиметричного шифрування, розроблений в 1978 році Робертом McEliece [10]. Це була перша подібна схема, яка використовувала рандомізацію у процесі шифрування. Алгоритм ніколи не отримував великого визнання в криптографічному співтоваристві, але є кандидатом для постквантової криптографії, оскільки він несприйнятливий до атак з використанням алгоритму Шора і, в більш загальному плані, до вимірювання суміжних станів з використанням вибірки Фур'є.

Алгоритм ґрунтується на складності декодування загального лінійного коду (який, як відомо, є NP-важким). Для створення закритого ключа вибирається код, що виправляє помилки, для якого відомий ефективний алгоритм декодування і здатний виправити t помилок. Оригінальний алгоритм використовує двійкові коди Гоппи (коди підполів геометричних кодів Гоппи кривої роду 0 над кінцевими полями характеристики 2). Відкритий ключ виходить із закритого ключа шляхом маскування вибраного коду під звичайний лінійний код. Для цього матриця коду G, що породжує, обурюється двома випадково обраними оборотними матрицями S і P.

Криптосистема МакЕліса має деякі переваги перед, наприклад, RSA, шифрування та дешифрування виконуються швидше. Довгий час вважалося, що МакЕліса не можна використовувати для створення підписів. Однак схема підпису може бути побудована на основі схеми Нідеррайтера, двоїстого варіанта схеми МакЕліса. Одним з основних недоліків МакЕліса є те, що закритий і відкритий ключі є великими матрицями. За стандартного набору параметрів довжина відкритого ключа становить 512 кілобіт.

Є три основні проблеми із криптосистемою McEliece.

Розмір відкритого ключа (G') досить великий. Використовуючи код Гоппи з параметрами, запропонованими McEliece, відкритий ключ складатиметься з 2^{19} біт. Це, безумовно, спричинить проблеми з реалізацією.

Ще один недолік, що зашифроване повідомлення набагато довше за відкрите текстове повідомлення. Це збільшення пропускної спроможності робить систему більш схильною до помилок передачі.

Отже, можна точно сказати, що симетричне шифрування, наприклад автентифіковане шифрування з використанням генератора псевдовипадкових ключових послідовностей (ГПКП) [2] не підходить для мережевих ігор. В наступному розділі буде вирішено вище перелічені недоліки за рахунок модифікації криптосистеми McEliece з використанням недвійкових кодів Гемінґа.

2 РОЗРОБКА МОДИФІКОВАНОЇ КРИПТОСИСТЕМИ MCELIECE

2.1 Розробка модифікованої криптосистеми McEliece на основі четвіркових кодів Гемінґа

Перед початком розробки криптосистеми McEliece необхідно дослідити четвіркові коди Гемінґа, які відіграють ключову роль в даній криптосистемі. Потрібно отримати нові сімейства, новий алгоритм генерації матриць парності.

2.1.1 Четвіркові коди Гемінґа

Коди Гемінґа є часто застосовуваними на практиці лінійними кодами для двійкового алфавіту із кодовою відстанню $d = 3$, що дозволяють детектувати двократні та виправляти однократні помилки [11]. На сьогоднішній день коди Гемінґа є досить добре дослідженими та розповсюджені на практиці. Однак, незважаючи на отримані суттєві наукові результати у теорії завадостійкого кодування для класів двійкових лінійних кодів, велика низка питань, що стосуються застосування недвійкових кодів лишається невирішеною. Концептуальна ідея недвійкових кодів Гемінґа також введена у роботі [12].

Проведені дослідження показують, що коди Гемінґа можуть бути побудовані не тільки для розширеного поля $GF(2^2)$, але і для введених в роботах [13, 14] розширень розширених полів $GF(q^r)$, де $q = 2^u, u = 2,3,\ldots$.

Даний факт дозволяє говорити про існування нових сімейств кодів Гемінґа над розширеннями розширених полів. Наприклад, нехай задано поле $GF(q^r)$, де r — число перевірочних символів у коді Гемінґа, що будується. Тоді параметри коду визначатимуться наступними співвідношеннями: довжина коду $n = \frac{q^r - 1}{q - 1}$, число інформаційних символів $k = \frac{q^r - 1}{q - 1} - r$, число помилок, що детектуються кодом $f = 2$, число помилок, що виправляються кодом $t = 1$. У цій роботі ми розглядаємо розширене поле Галуа $GF(2^2)$, арифметика якого може бути

побудована відповідно до єдиного існуючого первісного незвідного полінома $f(x) = x^2 + x + 1$ степені $\deg(f(x)) = 2$. Наведемо таблиці складання та множення у полі $GF(2^2) = GF(4)$, арифметика якого визначається зазначеним поліномом.

+	0	1	2	3
0	0	1	2	3
1	1	0	3	2
2	2	3	0	1
3	3	2	1	0

,

·	0	1	2	3
0	0	0	0	0
1	0	1	2	3
2	0	2	3	1
3	0	3	1	2

. (2.1)

Зважаючи на співвідношення (2.1), ми можемо говорити про існування нових класів кодів Гемінґа над полем розширення розширеного поля $GF(4^r)$ із параметрами (5, 3), (21, 18), (85, 81) при $r = 2,3,4$. Дані коди можуть бути задані за допомогою перевірочної матриці H, яка визначається наступним співвідношенням

$$H = [H_{r,k} \mid I_{r,r}], \tag{2.2}$$

де $I_{r,r}$ — матриця порядку r стовбці якої мають одиничну вагу; матриця $H_{r,k}$ розміру $r \times k$ містить стовбці, що є ортогональними над полем $GF(4^r)$ один одному, а також стовбцям матриці $I_{r,r}$. Отже, за побудовою, всі стовбці матриці H є взаємо ортогональними.

На основі матриці перевірки на парність може бути побудована і генераторна матриця шляхом транспонування лівої частини H з одиничною k - одиничною матрицею в лівій частині G.

$$G = [I_{k,k} \mid -H_{r,k}^T], \tag{2.3}$$

де T — символ транспонування.

Розглянемо приклад побудови (5, 3)-коду Гемінґа над розширенням розширеного поля $GF(2^2)$. Поле $GF(2^2)$ містить 15 ненульових елементів, кожен з яких ми можемо представити у вигляді четвіркового вектору. При цьому, зазначені елементи формують $(q^r - 1)/3$ класів, всередині яких містяться лінійно-залежні вектори, тобто такі, які можуть бути отримані один з одного шляхом множення на константу $a \in \{1,2,3\}$ над полем $GF(2^2)$, тобто відповідно до таблиць множення (2.1). Для нашого прикладу, означені $(4^2 - 1)/3 = 5$ класів лінійно

незалежних четвіркових векторів-елементів поля $GF(4^2)$ представлені за допомогою різних кольорів

$$V = \{01,02,03,10,11,12,13,20,21,22,23,30,31,32,33\}. \quad (2.4)$$

Вибираючи по одному вектору з кожного класу можемо сформувати матрицю перевірочну H у відповідності до (2.2)

$$H = \begin{bmatrix} 1 & 1 & 1 & 1 & 0 \\ 1 & 2 & 3 & 0 & 1 \end{bmatrix}, \quad (2.5)$$

яка повністю визначає перевірочні рівняння

$$\begin{cases} x_1 + x_2 + x_3 + x_4 = s_1; \\ x_1 + 2x_2 + 3x_3 + x_5 = s_2. \end{cases} \quad (2.6)$$

де арифметичні дії виконуються відповідно до таблиць (2.1).

Вкажемо також, що стовпці матриці Н можуть бути розташовані в довільному порядку, так само як і можуть бути обрані будь які вектори з еквівалентних класів (2.4). При цьому коректувальні здатності здатність коду не змінюється.

Відзначимо, що декодування інформації, як і у випадку двійкових кодів Гемінґа, може бути виконано на основі методу синдрому [11]. При цьому, конкретне значення синдрому $S = HC'^T = [s_1 \quad s_2]^T$ збігається із стовбцем матриці H, який відповідає символу, де трапилася помилка, який помножено на амплітуду помилки. Таким чином, обчисливши синдром стає можливим виправити помилку, що трапилася.

На основі перевірочної матриці H може бути побудована генераторна матриця G у відповідності до (2.3)

$$G = \begin{bmatrix} 1 & 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 & 2 \\ 0 & 0 & 1 & 1 & 3 \end{bmatrix}, \quad (2.7)$$

яка визначає рівняння кодування перевірочних символів

$$\begin{cases} x_4 = x_1 + x_2 + x_3; \\ x_5 = x_1 + 2x_2 + 3x_3. \end{cases} \quad (2.8)$$

Розглянемо приклад роботи запропонованого коректувального коду Гемінґа. Нехай задано інформаційне повідомлення $A = [0 \quad 1 \quad 2]$, яке закодуємо

запропонованим кодом Гемінґа $C = AG = [0\ \ 1\ \ 2\ \ 3\ \ 3]$. Уявимо, що помилка амплітудою 2 трапилася у другому розряді кодового слова (який відповідає другому інформаційному розряду), тобто вектор помилки $e = [0\ \ 2\ \ 0\ \ 0\ \ 0]$, тоді як прийняте кодове слова матиме вигляд $C' = [0\ \ 3\ \ 2\ \ 3\ \ 3]$.

На приймальній стороні розраховуємо значення синдрому, яке дорівнюватиме $S = [2\ \ 3]^T$. Оскільки отриманий синдром відповідає другому стовбцю матриці H, помноженому на константу $a = 2$ робимо висновок, що помилка трапилася у другому розряді кодового слова, тоді як амплітуда зміни складала 2. Означене дозволяє відтворити правильне слово C.

2.1.2. Алгоритм генерації перевірочних матриць

Відзначимо, що у випадку застосування розширень розширених полів, зокрема, для побудови четвіркових кодів Гемінґа нагальною є необхідність виділення у повному коді четвіркових векторів лінійно-незалежних векторів, тобто його класифікація на $q^r - 1/3$ класів, у кожному з яких містяться по 3 лінійно незалежні вектори. Вирішення цього завдання для великих значень r може бути досить обчислювально складним, оскільки пов’язане із перебором множини з $q^r - 1$ елементів.

Для вирішення цього завдання може бути застосований запропонований алгоритм генерації коду всіх можливих векторів, що не містить їх лінійних комбінацій:

Крок 1. Для значення $r = 1$ даний код містить лише одне кодове слово $\{1\}$.

Крок 2. Для заданого значення r код лінійно-незалежних векторів будується наступним чином на основі коду лінійно-незалежних векторів довжини $r - 1$.

Крок 2.1. Дописати старшим розрядом до кожного кодового слова коду лінійно-незалежних векторів довжини $r - 1$ символ «0».

Крок 2.2. Решта кодових слів коду лінійно-незалежних векторів будується шляхом дописування символу «1» старшим розрядом до повного коду четвіркових векторів довжини $r-1$.

Наприклад, за допомогою запропонованого алгоритму побудуємо код лінійно-незалежних векторів для значення $r=2$

$$\{0\ \ 1\};\ \ \{1\ \ 0\};\ \ \{1\ \ 1\};\ \ \{1\ \ 2\};\ \ \{1\ \ 3\}, \tag{2.9}$$

на основі якого може бути побудований код лінійно-незалежних векторів для значення $r=3$

$$\begin{array}{c} \{0\ \ 0\ \ 1\};\ \ \{0\ \ 1\ \ 0\};\ \ \{0\ \ 1\ \ 1\};\ \ \{0\ \ 1\ \ 2\};\ \ \{0\ \ 1\ \ 3\}; \\ \{1\ \ 0\ \ 0\};\ \ \{1\ \ 0\ \ 1\};\ \ \{1\ \ 0\ \ 2\};\ \ \{1\ \ 0\ \ 3\}; \\ \{1\ \ 1\ \ 0\};\ \ \{1\ \ 1\ \ 1\};\ \ \{1\ \ 1\ \ 2\};\ \ \{1\ \ 1\ \ 3\}; \\ \{1\ \ 2\ \ 0\};\ \ \{1\ \ 2\ \ 1\};\ \ \{1\ \ 2\ \ 2\};\ \ \{1\ \ 2\ \ 3\}; \\ \{1\ \ 3\ \ 0\};\ \ \{1\ \ 3\ \ 1\};\ \ \{1\ \ 3\ \ 2\};\ \ \{1\ \ 3\ \ 3\}, \end{array} \tag{2.10}$$

і так далі.

Відзначимо при цьому, що для побудови перевірочної матриці H коду Геммінга в якості стовбців можуть бути взяті або лінійно незалежні вектора коду лінійно-незалежних векторів у незмінному вигляді, або їх лінійні комбінації з певними константами $a_i, i=1,2,...,4^k/3-1$.

2.1.3 Криптосистема МакЕліса на основі четвіркових кодів Гемінга

Запропоновані четвіркові коди Геммінга можуть бути основою криптосистеми МакЕліса, при цьому надаючи їй конкретні переваги перед існуючими варіантами цієї криптосистеми на основі інших кодів. Розглянемо алгоритми роботи криптосистеми МакЕліса на основі четвіркових кодів Гемінга у режимі передавання інформації від сторони Б (Боба) стороні А (Алісі), супроводжуючи його конкретними прикладами.

Алгоритм генерації ключової інформації

Крок 1. Аліса вибирає (*n*,*k*)-лінійний код Υ, що виправляє одну помилку. Потім для коду Υ обчислюється генераторна $k\times n$ матриця G.

Крок 2. З метою ускладнення задачі відновлення вихідного коду, Аліса генерує випадкову невироджену $k \times k$ матрицю S над алфавітом $\{0,1\}$.

Крок 3. Аліса генерує довільну $n \times n$ матрицю перестановки P над алфавітом $\{0,1\}$.

Крок 4. Аліса обчислює $k \times n$ матрицю $G' = SGP$, яка є відкритим ключем. Закритим ключем є набір $\{S, G, P\}$.

В якості прикладу оберемо синтезовану нами у Розділі 3 генераторну матрицю G (2.7), а також згенеруємо випадкову невироджену матрицю S та матрицю перестановки P

$$S = \begin{bmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 1 \end{bmatrix}; \quad P = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \end{bmatrix}. \qquad (2.11)$$

Обчислюємо матрицю відкритого ключа

$$G' = SGP = \begin{bmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 1 \end{bmatrix} \begin{bmatrix} 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 & 2 \\ 0 & 0 & 1 & 1 & 3 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 3 & 0 & 1 \\ 1 & 0 & 2 & 1 & 0 \\ 0 & 1 & 2 & 0 & 1 \end{bmatrix}, \qquad (2.12)$$

для зберігання якої знадобиться $2kn$ двійкових розрядів, оскільки елементи даної матриці є четвірковими.

Алгоритм шифрування інформації

Опишемо алгоритм шифрування у вигляді конкретних кроків.

Крок 1. Боб представляє своє повідомлення *m* як послідовностей чотирирічних символів x довжини *k*.

Крок 2. Боб генерує випадковий вектор e довжини *n* ваги Гемінґа *t*.

Крок 3. Боб обчислює шифротекст як $y = xG' + e$ і передає його Алісі.

Нехай після отримання від Аліси відкритого ключа G' Боб сформував повідомлення $x = [1 \quad 2 \quad 1]$, а також обрав вектор помилки [0 1 0 0 0]. Тоді Боб може виконати шифрування свого відкритого повідомлення

$$y = [1 \quad 2 \quad 1] \begin{bmatrix} 1 & 0 & 3 & 0 & 1 \\ 1 & 0 & 2 & 1 & 0 \\ 0 & 1 & 2 & 0 & 1 \end{bmatrix} + [0 \quad 1 \quad 0 \quad 0 \quad 0] = [3 \quad 0 \quad 2 \quad 2 \quad 0]. \qquad (2.13)$$

Після шифрування відкрите повідомлення передається Алісі, яка виконую наступні дії для розшифрування повідомлення із застосуванням свого секретного ключа.

Алгоритм розшифрування інформації

Крок 1. Аліса обчислює зворотну матрицю P^{-1}.

Крок 2. Аліса обчислює $\hat{y} = yP^{-1}$.

Крок 3. Аліса використовує алгоритм декодування коду Υ, щоб отримати $\hat{x}$ з $\hat{y}$.

Крок 4. Аліса обчислює $x = \hat{x}S^{-1}$.

Для нашого прикладу

$$P^{-1} = \begin{bmatrix} 0 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \end{bmatrix}, \qquad (2.14)$$

тоді як $\hat{y} = [0 \quad 2 \quad 0 \quad 3 \quad 2]$.

Застосуємо алгоритм декодування четвіркового коду, бачимо, що синдром дорівнює $[1 \quad 0]$: через перестановки помилка перемістилася до 1-го символу $eP^{-1} = [1 \quad 0 \quad 0 \quad 0 \quad 0]$. Оскільки отриманий синдром відповідає стовбцю матриці H, а не його лінійній комбінації, то робимо висновок, що амплітуда помилки становила 1, тобто виправлене кодове слово $\hat{\hat{y}} = [1 \quad 2 \quad 0 \quad 3 \quad 2]$, тоді як $\hat{x} = [1 \quad 2 \quad 0]$. Знаходимо, що $x = [1 \quad 2 \quad 1]$, що відповідає вихідному тексту.

2.1.4 Стійкість системи

У разі застосування двійкових кодів Гемінга, як запропоновано у роботі [15] для організації криптосистеми МакЕліса кількість можливих генераторних матриць G визначатиметься як $n!$ через обумовлені властивостями коду Гемінга можливі перестановки її рядків. При цьому довжина відритого ключа складатиме kn. З іншого боку, у разі застосування четвіркового коду Гемінга, ми можемо обрати кожен із стовбців матриці G одним із 3-х способів із множини його

лінійних комбінацій, тобто ми отримуємо $(3n)!$ різних матриць. При цьому довжина відритого ключа складатиме $2kn$.

Наприклад, при застосуванні двійкового (63, 57)-коду Гемінга число можливих генераторних матриць становитиме $1.9826 \cdot 10^{87}$ при необхідних 3591 бітах для зберігання відкритого ключа, тоді як для четвіркового (21,18)-коду Гемінга при тому ж числі генераторних матриць буде необхідно лише 756 бітів відкритого ключа, тобто у 4.75 разів меньше. При цьому, через те, що криптосистема МакЕліса на основі кодів Гемінга над розширеннями розширених полів Галуа здатно одночасно оброблювати більшу кількість вхідної інформації, вона потенційно допускає більш швидкодіючі алгоритмічні реалізації (порівняно з двійковими кодами Гемінга).

Відзначимо також, що у загальному випадку рішення систем лінійних рівнянь над розширеними полями Галуа є більш складним з обчислювальної точки зору завданням, тоді як для більших значень u конкретний від ізоморфізму поля може бути частиною секретного ключа.

Перспективним вбачається застосування системи МакЕліса на основі каскадних четвіркових кодів Гемінга, для забезпечення стійкості щодо атаки Сидельникова. Так, наприклад, 17 заширований системою МакЕліса на основі (5,3)-коду Гемінга векторів відкритого тексту, можуть бути перешифровані системою МакЕліса на основі (85,81) - коду Гемінга, що має забезпечити велику складність взаємозв’язку між елементами вхідного та вихідного тексту.

Відзначимо, що більш детальні аспекти безпеки запропонованої модифікації криптосистеми МакЕліса на основі недвійкових кодів Гемінга мають стати предметом побільших досліджень.

2.2 Вбудова модифікованої криптосистеми McEliece в мережеву гру

Ігрова мережа [16] - це одна з категорій комп'ютерних мереж, яка включає синхронізацію станів, інтерполяцію сутностей, прогнозування введення і компенсацію затримок для розрахованих на багато користувачів ігор.

Синхронізація стану. Під час синхронізації стану оптимізуються мережні пакети/зв'язок між сервером та клієнтами. Ідея полягає в тому, щоб відправляти як вхідні дані (від гравців), так і стан гри, що працює на обох кінцях (мережі - сервер та клієнти) у приблизній синхронізації з втратами (таких значень, як цілі числа, числа з плаваючою комою, рядки та логічні значення). Стратегія. Перевага цієї стратегії полягає у виборі найважливішого об'єкта, який повинен надсилати оновлення для кожного пакета (пакету даних). Цей процес вимагає багато часу на екстраполяцію (розшифрування повідомлення), але це простий та ефективний процес.

Інтерполяція об'єктів. Як правило, майже кожна розрахована на багато користувачів гра має інтерполяцію сутностей у своїй мережній моделі, щоб приховати затримки і тремтіння. Затримки виникають, коли оновлення надсилаються рідше (в секунду) або через відкидання пакетів, що більше, якщо в моделі використовується UDP. Інтерполяція допомагає згладити перетворення, а також наближає їх до вихідних рухів.

Вхідне передбачення. Прогнозування введення створює плавніший рух гравця ще до того, як сервер відповість на введення клієнта. Він використовує попередній стан гравців плюс введення гравця для малювання наступного прогнозованого стану (створеного з використанням дельта-алгоритму, який є попередньою позицією плюс наступну очікувану позицію). І один раз, коли він отримує оновлення із сервера, він коригує поточний стан гравця та видаляє старі дані. Цей процес повторюється між кожним оновленням, щоб забезпечити плавний ігровий процес.

Існують важливі концепції ігрової мережі.

Затримка. Затримка - це середній час в обидва кінці, що витрачається робочою станцією на відправку пакетів даних на сервер (для ігор це ігровий сервер) та отримання пакетів даних з сервера. Чим вище латентність, тим повільніше йде гра - клієнти не зможуть повною мірою насолодитися ігровим процесом.

Виділений сервер. Виділений сервер — це виділена робоча станція, на якій встановлена копія гри та яка запускає гру як звичайний сервер.

Хост-сервер. На хост-сервері немає виділеної робочої станції/комп'ютера для запуску гри. Один із клієнтів (хост-клієнт) також гратиме роль сервера, створюючи екземпляр гри, до якого можуть підключатися інші гравці.

Міграція хоста У клієнт-серверній моделі централізований сервер має владу над грою. Якщо сервер виходить із ладу або виходить з ладу, ігровий процес переривається. Щоб уникнути цього, ми можемо створити міграцію хоста, за допомогою якої на всіх клієнтах підтримуватиметься копія гри, яка використовуватиметься для перенесення гри на хост при поточному відключенні сервера.

Обчислення. Ігровий сервер передбачає наступний хід на основі останнього ходу.

Відкочування. Коли виправляють клієнти, спотворюючи неузгоджені прогнози з ігрового сервера, це називається відкатом.

Модель OSI розроблена Міжнародною організацією зі стандартизації для стандартизації системи зв'язку у 1978 році [17]. У цій моделі закладено стандартизацію того, як одна система має взаємодіяти з іншою. Він має сім різних рівнів і кожен рівень призначений для роботи над частиною комунікаційної моделі.

Мережевий рівень – ключ в ігровій мережі.

Мережевий рівень – це третій рівень моделі OSI (Open System Interconnection), який відповідає за передачу пакетів даних із комп'ютера/вузла на інший комп'ютер/вузол. Він забезпечує маршрутизацію даних та комутацію для створення шляхів для мережної взаємодії.

Задачі мережного рівня:

- встановлення логічного зв'язку між вузлами;
- пересилання даних;
- маршрутизація;
- надсилання звіту про помилку.

Існує кілька моделей ігрових серверів, які широко популярні в ігрових мережах.

Однорангова мережа (P2P). У Peer-to-Peer [18] гравці підключаються безпосередньо один до одного без централізованої системи. Таким чином, у P2P кожен гравець ділиться інформацією про свій ігровий статус з кожним гравцем у грі. При цьому кожне з'єднання є вузлом/бенкетом, і кожен бенкет має рівні повноваження в грі.

З розвитком моделі клієнт-сервер ця модель вже не така популярна, як раніше, в розрахованих на багато користувачів іграх.

У моделі клієнт-сервер гравці (клієнти) підключаються через централізований сервер. Цей сервер має повну владу над ігровим процесом. Централізований сервер, через який підключаються гравці, називається ігровим сервером. Ігровий сервер повністю відповідає за відстеження ігрового статусу кожного клієнта та поширення інформації серед решти гравців; тобто вся взаємодія з клієнтом відбувається через ігровий сервер.

Наприклад: коли клієнт оновлює сервер про своє розташування, здоров'я, кількість боєприпасів і т. д., ця інформація оновлюється в системах інших клієнтів ігровим сервером.

Принцип роботи криптосистеми McEliece в клієнт-серверній грі не досить складний.

1. Користувач з’єднується з сервером, генерує зв’язку ключів відкритий і секретний.

2. Клієнт по не захищеному каналу відправляє серверові свій відкритий ключ.

3. Сервер прийняв пакет, генерує свою зв’язку ключів відкритий і секретний ключ та відправляє відкритий ключ у відповідь клієнтові.

Відбувся обмін між сторонами, тобто тепер утворився захищений канал для подальшого шифрування.

На клієнті та сервері було реалізовано асиметричне шифрування та дешифрування даних, тим самим утворюється захищений канал зв’язку (рис 2.1).

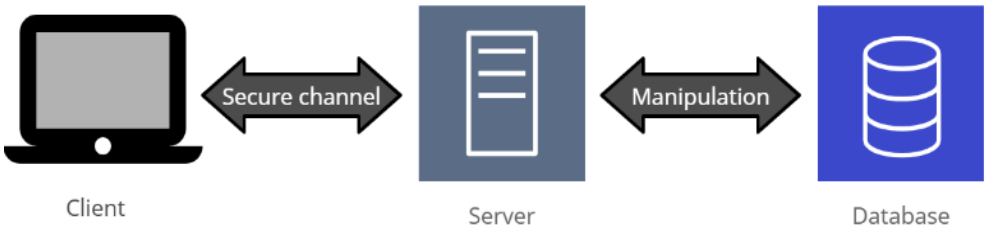

Рисунок 2.1 – Взаємодія клієнта з сервером по захищеному каналу

Кіберзлочинність стає все більш витонченою, а це означає, що постачальники керованих послуг (MSP) повинні постійно оновлювати свої методи та політики безпеки. Збільшилася кількість атак, націлених на облікові дані за допомогою таких методів, як фішинг, підбір методом підбору та атаки за словником. Це означає, що автентифікація більше не може покладатися лише на паролі.

У поєднанні з додатковими методами аутентифікації аутентифікація на основі токенів може створити складніший бар'єр, що не дозволяє досвідченим хакерам використовувати вкрадені паролі.

Токени можна отримати тільки з того унікального пристрою, який їх створив (наприклад, зі смартфона), що робить сьогодні дуже ефективним методом авторизації.

Незважаючи на те, що платформи токенів аутентифікації мають багато переваг, завжди залишається деякий ризик. Токени, розміщені на мобільних пристроях, зручні у використанні, але можуть бути вразливі через вразливість пристрою. Якщо токени надсилаються у вигляді тексту, їх можна легко перехопити у дорозі. Якщо пристрій втрачено або викрадено, зловмисник може отримати доступ до токенів, що зберігаються на ньому.

Аутентифікація на основі токенів – це протокол, що генерує зашифровані токени безпеки [19]. Це дозволяє користувачам підтверджувати свою особу на веб-сайтах, які потім генерують унікальний зашифрований токен автентифікації. Цей токен надає користувачам доступ до захищених сторінок та ресурсів протягом обмеженого періоду часу без необхідності повторного введення імені

користувача та пароля. Цей процес на основі токенів доводить, що користувачу було надано доступ до додатків, веб-сайтів та ресурсів без необхідності підтверджувати свою особу кожного разу, коли він переходить на новий сайт. Токен може зберігатися не тільки в базі даних, але й локально.

Веб-сайти можуть додавати додаткові рівні безпеки крім традиційних паролів, не змушуючи користувачів повторно підтверджувати свою особу, що підвищує зручність використання та безпеку.

Аутентифікація на основі токенів також є величезним кроком уперед у порівнянні з традиційними паролями, які за своєю суттю є небезпечними. Паролі генеруються людиною, що робить їх слабкими та легкими для злому хакерами. Наприклад, люди схильні повторно використовувати паролі для різних облікових записів, тому що це допомагає запам'ятати свої дані для входу.

Крім того, системи на основі паролів вимагають, щоб користувачі неодноразово вводили облікові дані для входу в систему, що витрачає час і може дратувати, особливо якщо вони забувають свій пароль.

При підході на основі токенів користувачеві потрібно запам'ятати лише один пароль, що швидше і простіше і спонукає використовувати більш надійний пароль.

Під час запуску гри, система перевіряє на вхід в ігровий профіль, за це відповідає функція isAuth(), інакше цей процес називається автентифікація. В клієнта на локальному сховищі зберігається унікальний токен, який генерується сервером при авторизації. Клієнт відсилає цей токен до сервера.

Сервер перевіряє цей токен в базі, якщо токен існує і збігається з тим що в базі, відсилає дозвіл на аутентифікацію, з прикладеним логіном і результатом true, інакше false. Слід відмітити, що ця функція визивається кожен раз при завантаженні даних.

Для кращого розуміння на рис. 2.2. представлена схема автентифікації.

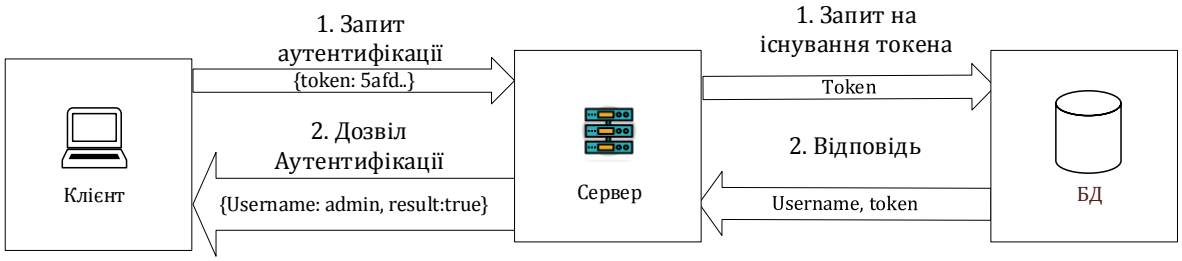

Рисунок 2.2 – Механізм роботи автентифікації

Авторизація – це механізм безпеки для визначення рівнів доступу або привілеїв користувача/клієнта, пов'язаних із системними ресурсами, включаючи файли, служби, комп'ютерні програми, дані та функції програм.

Це процес надання або відмови у доступі до мережного ресурсу, який дозволяє користувачу отримувати доступ до різних ресурсів на основі особи користувача.

На базовому рівні авторизація дозволяє отримати доступ до ігрового профілю гри за допомогою імені користувача та пароля.

Аутентифікація та авторизація часто використовуються як взаємозамінні, але є деякі важливі відмінності. Авторизація слідує за аутентифікацією. Те, що користувачі автентифіковані, не означає, що вони отримують доступ до всіх програм, служб та даних у корпоративній мережі. Натомість вони отримують доступ лише до ресурсів, які їм дозволено використовувати.

Два етапи управління доступом, авторизація та автентифікація, необхідні користувачеві для взаємодії з корпоративною мережею. Якщо користувачі не автентифіковані, вони не можуть користуватись жодними службами, оскільки не можуть увійти до мережі. Якщо користувач не авторизований, він не зможе використовувати будь-які служби, навіть якщо він пройшов автентифікацію.

Авторизація в мережевій грі працює за таким принципом. Якщо функція автентифікації поверне true – загрузка меню і завантаження даних, інакше – вікно авторизації. Далі необхідно ввести логін і пароль в графічному інтерфейсі. Відбувається перевірка на порожній рядок, і хешування паролю. Пароль ніколи не передається в відкритому вигляді. Формується спеціальний клас запит, який

серіалізується та шифрується за допомогою автентифікованого шифрування, і надсилається на сервер. Сервер розшифровує пакет, десеріалізує та робить запит до бази даних.

На сервері відбувається перевірка логіна та пароля з бази, якщо true – дані збігаються і генерується токен, який заноситься до бази, інакше false – логін та пароль введений невірно. Далі сервер формує клас відповідь, серіалізує та виконує асиметричне шифрування МакЕліса. Зашифрований пакет надсилається до клієнта. Клієнт виконує дешифрування МакЕліса, десерілізацію. Далі перевіряється результат пакету, якщо true – внесення токену в локальну базу та загрузка меню, інакше false – повторна авторизація.

Для кращого розуміння на рис. 2.3. представлена схема логіки UI & BACKEND гри.

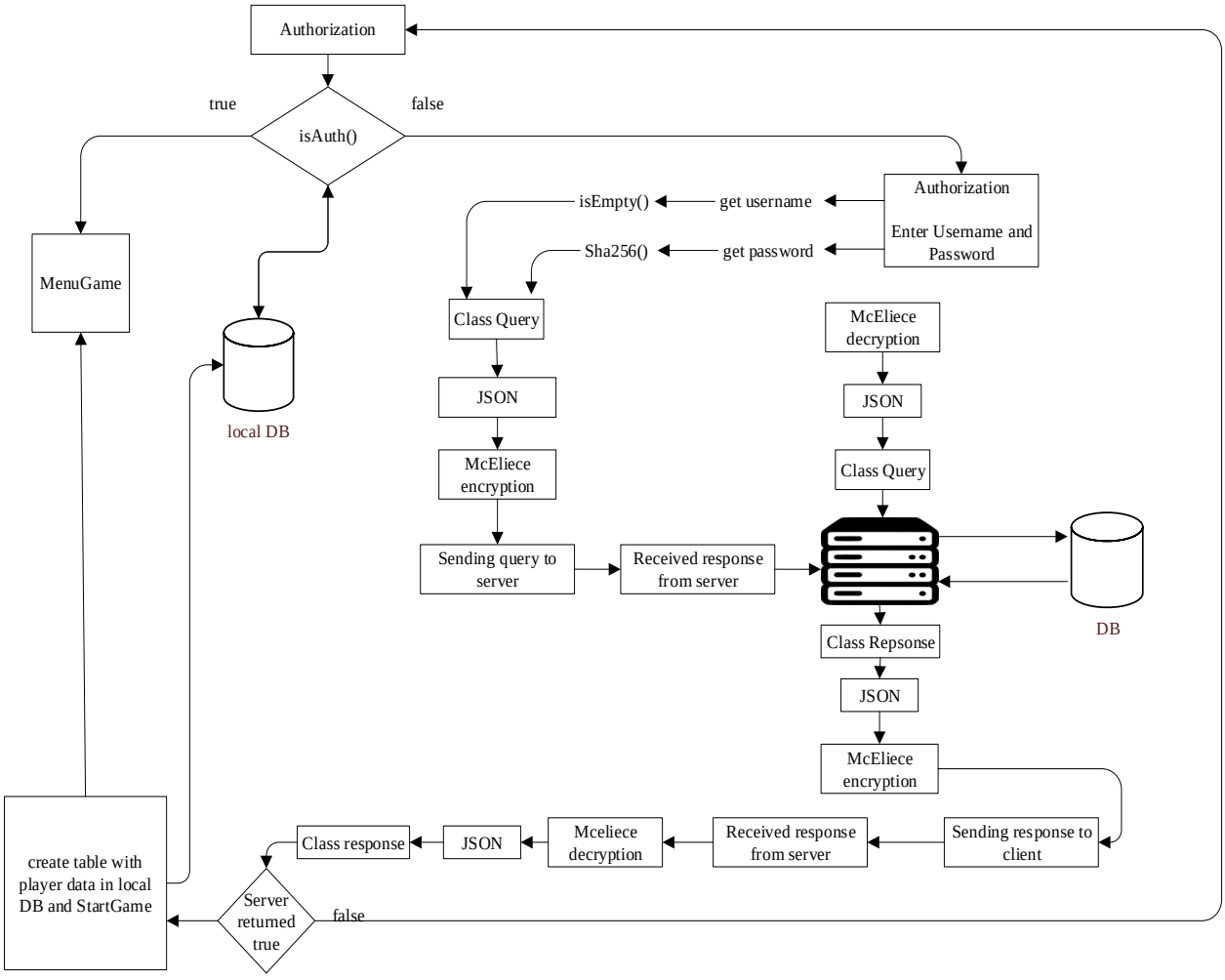

Рисунок 2.3 – Схема логіки входу в UI & BACKEND частині

Далі розглянемо алгоритм реєстрації, тобто створення ігрового профілю.

1. Вводимо логін, пароль та натискаємо реєстрація.

2. Формуємо запит клієнта, який складається з таких полів: query (назва запиту - Register), data (дані) та надсилаємо до серверу.

4. Сервер приймає запит та по назві запиту визначає призначення пакету.

5. Сервер робить запит до бази даних на перевірку логіна в базі, якщо логіна немає, то користувач не зареєстрований, інакше записуємо логін і пароль до бази.

6. Далі сервер формує відповідь клієнту результат, якої true чи false та надсилає до клієнту.

7. Отримавши відповідь від серверу клієнт витягує перевіряє результат свого запиту, якщо результат true, то реєстрація пройшла успішно, інакше користувач зареєстрований.

Розглянемо схему логіки реєстрації на рис. 2.4.

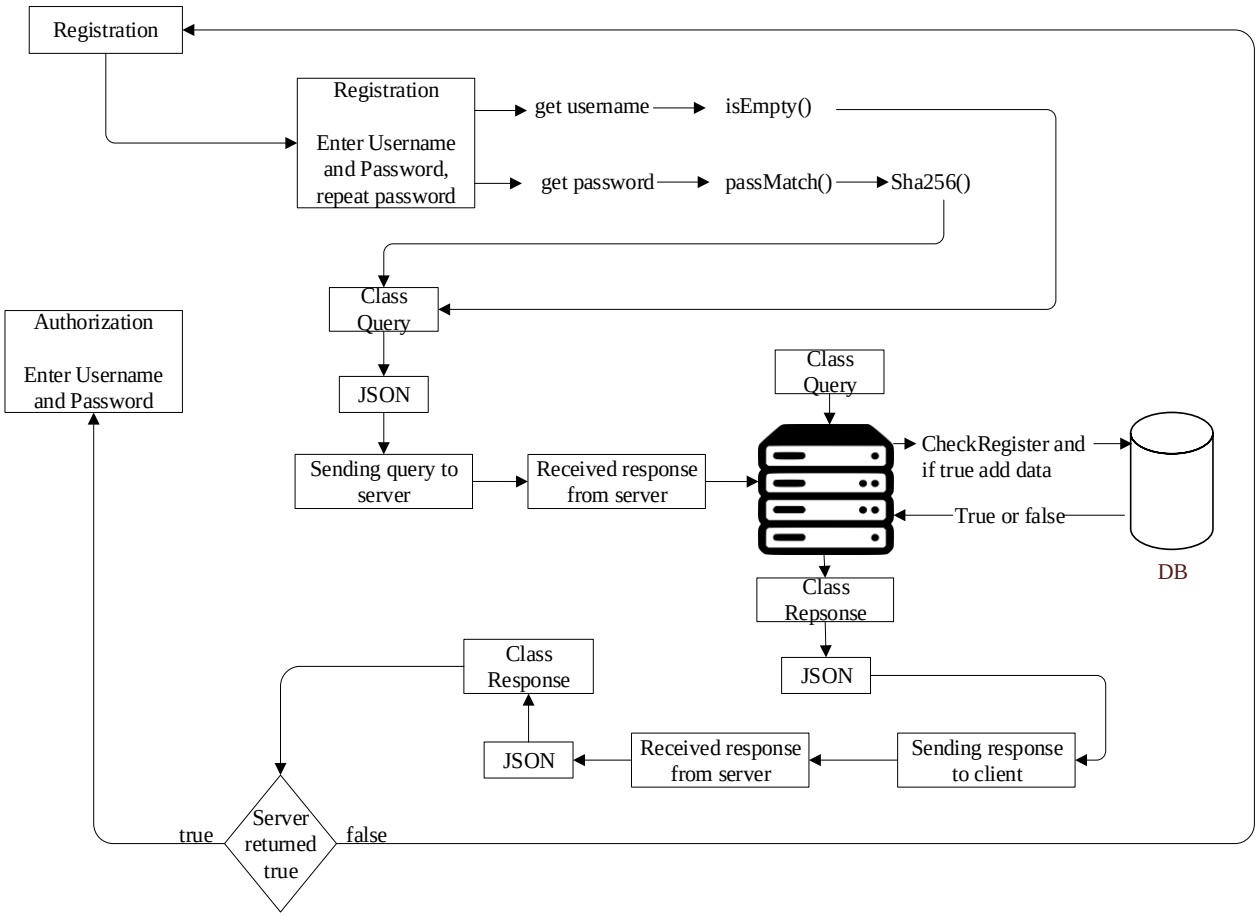

Рисунок 2.4 – Схема логіки реєстрації в UI & BACKEND частині

Після авторизації або автентифікації відбувається загрузка ігрових даних, тобто процедура обміну пакетами стандартна, сервер відсилає пакет в якому містяться збережені дані до клієнта, клієнт отримавши пакет робить загрузка даних . Зберігання аналогічно завантаженню даних, тільки тоді коли відбуваються, якісь зміни під час гри, наприклад, потрібно перезаписати прогрес гравця.

Отже, після того як розробили модифіковану криптосистему McEliece та представили її в онлайн грі, перейдемо до програмної реалізації.

3 РОЗРОБКА КРИПТОСИСТЕМИ MCELIECE ДЛЯ МЕРЕЖЕВОЇ ГРИ

3.1 Середовище програмування

Існує безліч мов програмування загального призначення, але розробники, напевно, погодяться з тим, що C# — одна з найкращих. Ймовірно, це пов'язано з тим, що C# є дуже універсальним, пропонує плавну криву навчання і є об'єктно-орієнтованим [20].

C# — це мова програмування загального призначення з мультипарадигмальним підходом, що охоплює кілька дисциплін програмування, таких як статична типізація, імперативне, декларативне, функціональне, об'єктно-орієнтоване та компонентно-орієнтоване програмування. Саме цей підхід дозволяє C# бути настільки універсальним, що ви можете використовувати його для багатьох проектів.

Розроблений Microsoft в 2000 році, C# був створений для задоволення попиту, що росте, на веб-додатки. У той час, як у компанії з Редмонда були Visual Basic і C++ для роботи над такими додатками, реальність така, що обидві мови мали проблеми з випуском високопродуктивного програмного забезпечення. Ось чому C# так швидко знайшов свою інженерну нішу — тому що його архітектура дотримується кращих практик Java, щоб забезпечити кращий підхід до розробки додатків.

Серед інших видатних особливостей C# є можливість повторного використання компонентів для прискорення розробки та гнучкі типи даних без помилок. Начебто цього було недостатньо, C# має широкий спектр компонентів, які можуть легко прискорити будь-який проект, чи то системний, чи то бізнес-орієнтований.

Будучи частиною платформи .NET, C# ідеально підходить для створення динамічних веб-сайтів та програм. Його об'єктно-орієнтований характер робить його ідеальним для розробки веб-сайтів, які відрізняються високою ефективністю та легко масштабуються. Мова програмування C# широко відомий як одна з найкращих мов програмування для ігор, особливо для ігор Unity, тому що

інтегрується з двигуном Unity, щоб забезпечити найкраще середовище для розробки мобільних ігор, і ви навіть можете використовувати його для розробки консольних ігор з крос-платформними технологіями.

Unity – це інструмент, який може виконувати безліч типів завдань, пов'язаних із процесом створення гри. Unity надає розробникам ігор 2D- та 3D-платформу для створення відеоігор [21].

Багато користувачів ігри Unity працюють наступним чином: спочатку гравець запускає гру в якості хоста (вибравши LAN Host). Хост працює як клієнт та сервер одночасно. Потім інші гравці можуть підключатися до цього хоста як клієнти (вибравши LAN Client).

Python – це інтуїтивно зрозуміла об'єктно-орієнтована мова програмування. Це мова загального призначення, що використовується для різних цілей, включаючи розробку програмного забезпечення, веб-скриптів та програм машинного навчання. Python був заснований Гвідо Ван Россумом у 1991 році. Сьогодні він є однією з найпопулярніших мов програмування. Статистика показує, що популярність Python постійно зростає [22].

Python підходить для різних проектів веб-програмування, як простих, так і складних. Він широко використовується для веб-розробки, тестування програмного забезпечення, сценаріїв, а також у різних галузях, таких як подорожі, охорона здоров'я, транспорт та фінанси.

Популярність Python обумовлена безліччю його переваг, таких як простота та елегантність, які приваблюють великі компанії, такі як Dropbox, Instagram та Spotify. Однак, незважаючи на безліч переваг використання Python для веб-розробки, є деякі підводні камені.

Писати та підтримувати асинхронний код за допомогою Python легко, тому що немає взаємоблокувань, наукових суперечок та інших заплутаних питань. Кожна така одиниця коду працює незалежно, що дозволяє швидко вирішувати різні ситуації та проблеми.

Python чудово підходить для настільних та веб-серверних додатків. На жаль, Python не підходить для розробки мобільних додатків та ігор через споживання

пам'яті та швидкості. Для мобільних додатків є такі виграшні технології, як React Native та Flutter для розробки під iOS та Android з єдиної кодової бази. Однією з переваг використання Python для сценаріїв на стороні сервера є його простий синтаксис, значно прискорює процес. Код складається з функціональних модулів та зв'язків між ними, що дозволяє виконувати алгоритм програми на основі дій користувача. Python також підтримує графічні інтерфейси користувача, необхідні для веб-розробки.

Для розробки клієнт-серверної частини використовуються сокети.

Мережевий сокет – це програмний компонент у вузлі комп'ютерної мережі, який діє як кінцева точка для доставки та отримання даних [23]. Інтерфейс прикладного програмування (API) для мережевої архітектури визначає структуру та властивості сокету. Сокети створюються лише протягом життєвого циклу процесу у додатку з урахуванням вузла.

Оскільки протоколи TCP/IP були стандартизовані під час створення Інтернету, слово "мережевий сокет" найчастіше використовується в контексті набору інтернет-протоколів, і тому його називають "інтернет-сокетом". У цьому контексті адреса сокета, що є тріадою транспортного протоколу, IP-адреси та номера порту, використовується для зовнішньої ідентифікації для інших хостів.

Канали формуються за допомогою системного виклику Pipe, а сокети – за допомогою системного виклику socket. По мережі сокет забезпечує двонаправлений зв'язок FIFO. На кожному кінці з'єднання формується сокет, що підключається до мережі. Кожен сокет має свою унікальну адресу. IP-адреса плюс номер порту складають цю адресу.

У більшості клієнт-серверних програм використовуються сокети. Сервер створює сокет, підключає його до мережного порту і чекає, доки до нього підключиться клієнт. Після створення сокету клієнт намагається підключитись до сокету сервера. Дані передаються після встановлення зв'язку.

PostgreSQL – це потужна система об'єктно-реляційних баз даних з відкритим вихідним кодом, відома своєю надійністю, функціональністю та

продуктивністю. PostgreSQL стає кращою базою даних для дедалі більшої кількості підприємств.

PostgreSQL – це реляційна база даних. Він зберігає точки даних у рядках зі стовпцями як різні атрибути даних [24]. У таблиці зберігається кілька зв'язаних рядків. Реляційна база даних є найпоширенішим типом використовуваної бази даних. Він відрізняється акцентом на інтеграції та розширюваності. Він працює з багатьма іншими технологіями та відповідає різним стандартам баз даних, що забезпечує його розширюваність.

PostgreSQL працює в більшості популярних операційних систем – майже у всіх дистрибутивах Linux та Unix, Windows, Mac OS X. Його відкритий вихідний код спрощує оновлення чи розширення. У PostgreSQL ви можете визначати власні типи даних, створювати власні функції і навіть писати код іншою мовою програмування (наприклад, Python) без перекомпіляції бази даних.

PostgreSQL має багату історію підтримки розширених типів даних і підтримує рівень оптимізації продуктивності, який зазвичай пов’язаний з комерційними аналогами баз даних, такими як Oracle і SQL Server. PostgreSQL використовується як основне сховище даних або сховище даних для багатьох веб-, мобільних, геопросторових і аналітичних програм.

PostgreSQL може зберігати структуровані та неструктуровані дані в одному продукті. Неструктуровані дані, які містяться в аудіо, відео, електронних листах і публікаціях у соціальних мережах, можна використовувати для покращення обслуговування клієнтів, виявлення нових вимог до продукту та пошуку способів запобігти відтоку клієнтів серед незліченних інших видів використання.

Отже, згідно з вище сказаним, серверна частина буде розроблена на мові програмуванні Python, а клієнтська частина буде написана мовою C#.

3.2 Криптосистема McEliece

Програмна реалізація криптосистеми McEliece складається з таких компонентів.

1. Модуль для визначення сімейства при заданому r.
2. Модуль для генерації перевірочних матриць четвіркових кодів Гемінґа.
3. Криптосистема McEliece з використанням четвіркових кодів Гемінґа.

На рис. 3.1 представлено програмний модуль, який виводить всі сімейства четвіркових кодів Гемінґа при заданому r. Задати r можна будь-яке, але потрібно враховувати, що фізично комп’ютер не зможе обчислювати, такі матриці.

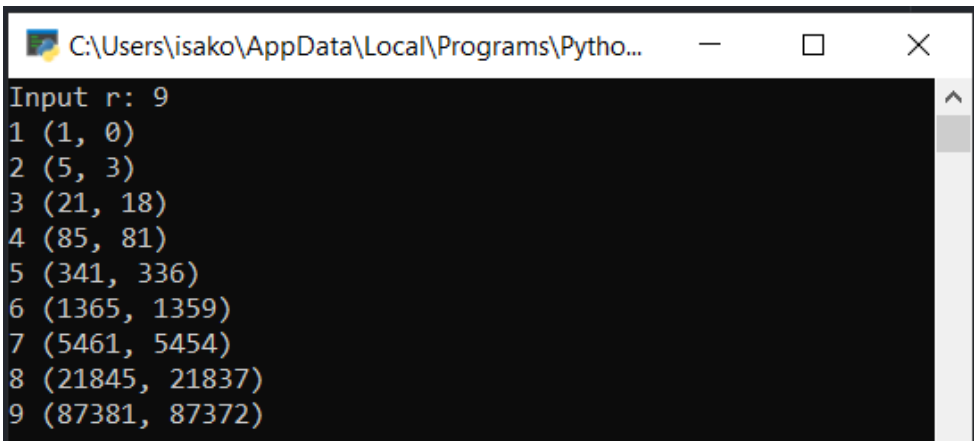

Рисунок 3.1 – Четвіркові коди Гемінґа при різних r

Наступним компонентом є генерація перевірочної матриці, за допомогою вище згаданого алгоритма (рис. 3.2). Максимальне сімейство, яке можна згенерувати r = 12, але в даний момент використовувати такі матриці для кодування Гемінґа зараз неможливо, тому що перемножувати матриці великих розмірів дуже складно.

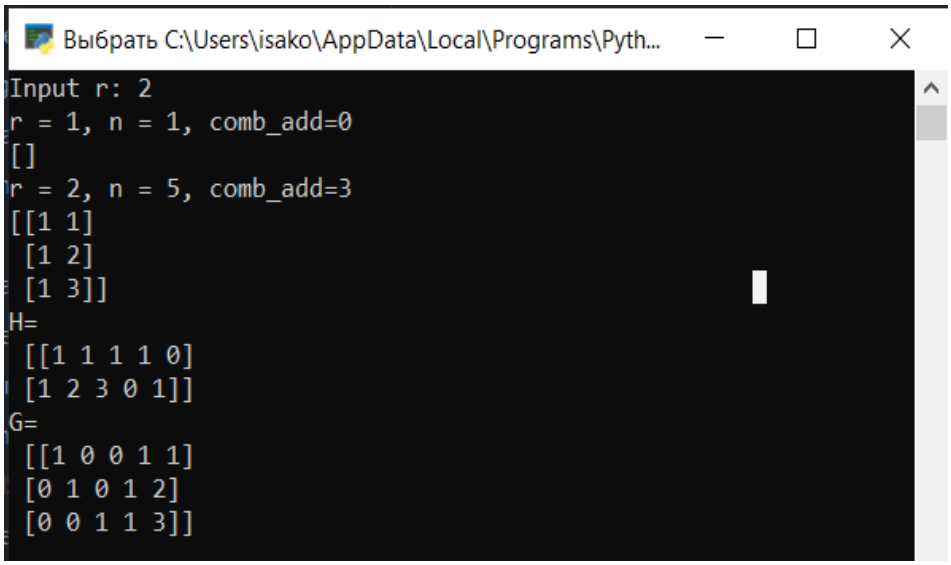

Рисунок 3.2 – Генерація перевірочних матриць для четвіркових кодів Гемінґа

На рис. 3.3 представлено кодування четвіркових кодів Гемінґа, при r = 2, сімейство (5, 3). Програма генерує матрицю H та G за допомогою розробленого мною алгоритма.

Далі необхідно ввести 3 інформаційних біта [0, 1, 2] для подальшого кодування. Отримуємо закодоване повідомлення за рахунок множення вектора, тобто повідомлення на матрицю G.

Тепер необхідно спотворити закодоване повідомлення для подальшого виправлення та виявлення помилки.

Для цього потрібно вести позицію помилки, тобто задається вектор помилки, розміром загальної кількості всіх блоків. і ми отримаємо, нове повідомлення з помилкою, наприклад, в першій позиції.

Далі, проводимо декодування спотвореного повідомлення, для цього запускається алгоритм декодування, або іншими словами функція для декодування. Вона працює дуже просто, матриця перевірки на парність перемножується з повідомленням, в результаті чого ми отримуємо синдром.

Якщо синдром нулі, то це означає лише одне, що ніяких змін при передачі повідомлення не відбувалося, тому помилки відсутні, інакше зміни були і помилка є. Тобто результат показує вектор-стовпець матриці перевірки на парність H, якщо цей елемент, то визиваємо функцію на визначення позиції елемента.

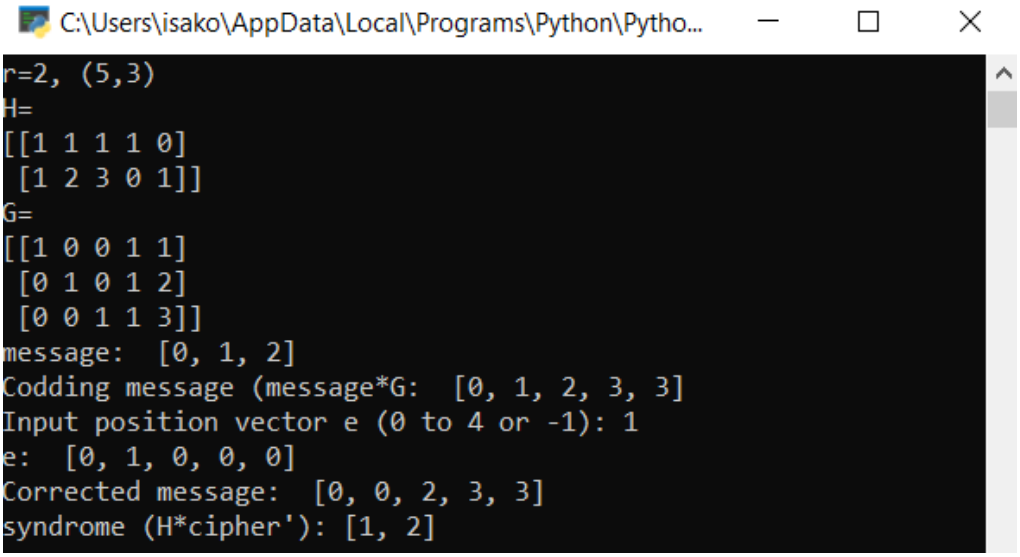

Рисунок 3.3 – Кодування та декодування четвіркових кодів Гемінґа

Для початку вводимо будь-яке повідомлення, та вказуємо кількість перевірочних біт, для генерації матриць кодів Гемінґа. Кожна буква повідомлення переводиться в ASCII формат, в результаті чого ми отримаємо число, тобто позицію букви в таблиці ASCII. Далі, відбувається переклад числа з десяткової в четвіркову систему числення, та доповнюємо це число до 6 біт. Кожна буква буде розбиватися на блоки по 6 біт. При r = 2, маємо (5, 3), звідси 3 інформаційних біта, тобто розмір одного блока 3 біта.

Робота програми буде складеться з двох частин. Перша частина це генерація закритих ключів, тобто генерується матриця H та G, а також матриці скремблера та перестановки, в результаті певних операцій отримуємо відкритий ключ, який необхідний для подальшого шифрування.

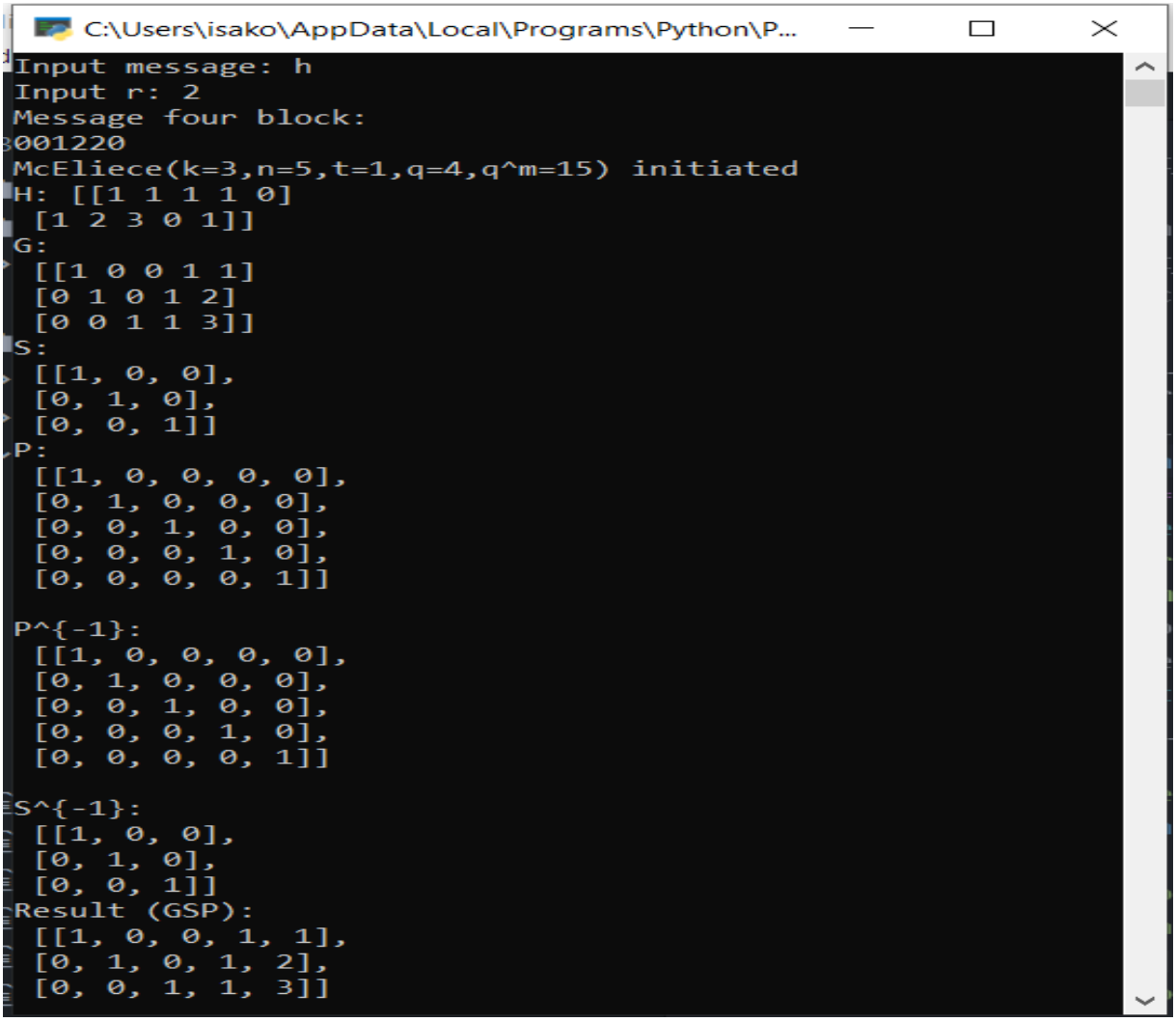

Рисунок 3.4 – Генерація відкритого та закритого ключів криптосистеми McEliece

Друга частина програми шифрування та розшифрування з декодуванням. На рис. 3.5 представлено кодування четвіркових кодів Гемінґа. Кожен блок розбивається на блоки, розміром інформаційних бітів, шифрується, та задається вектор помилки, але він тут нульовий, тому помилки немає. На виході ми отримуємо зашифроване повідомлення, до якого додається вектор помилки. Дешифрування відбувається по блокам, та обчислюється синдром для визначення помилок, як ми бачимо вони відсутні, потім прибираються доповнені блоки, відбувається переведення повідомлення розміром 6 біт з четвіркової системи в десяткову систему числення. Далі, необхідно перевести в ASCII, для того щоб повернути повідомлення в початковий вигляд. Як ми бачимо, повідомлення розшифрувалось успішно.

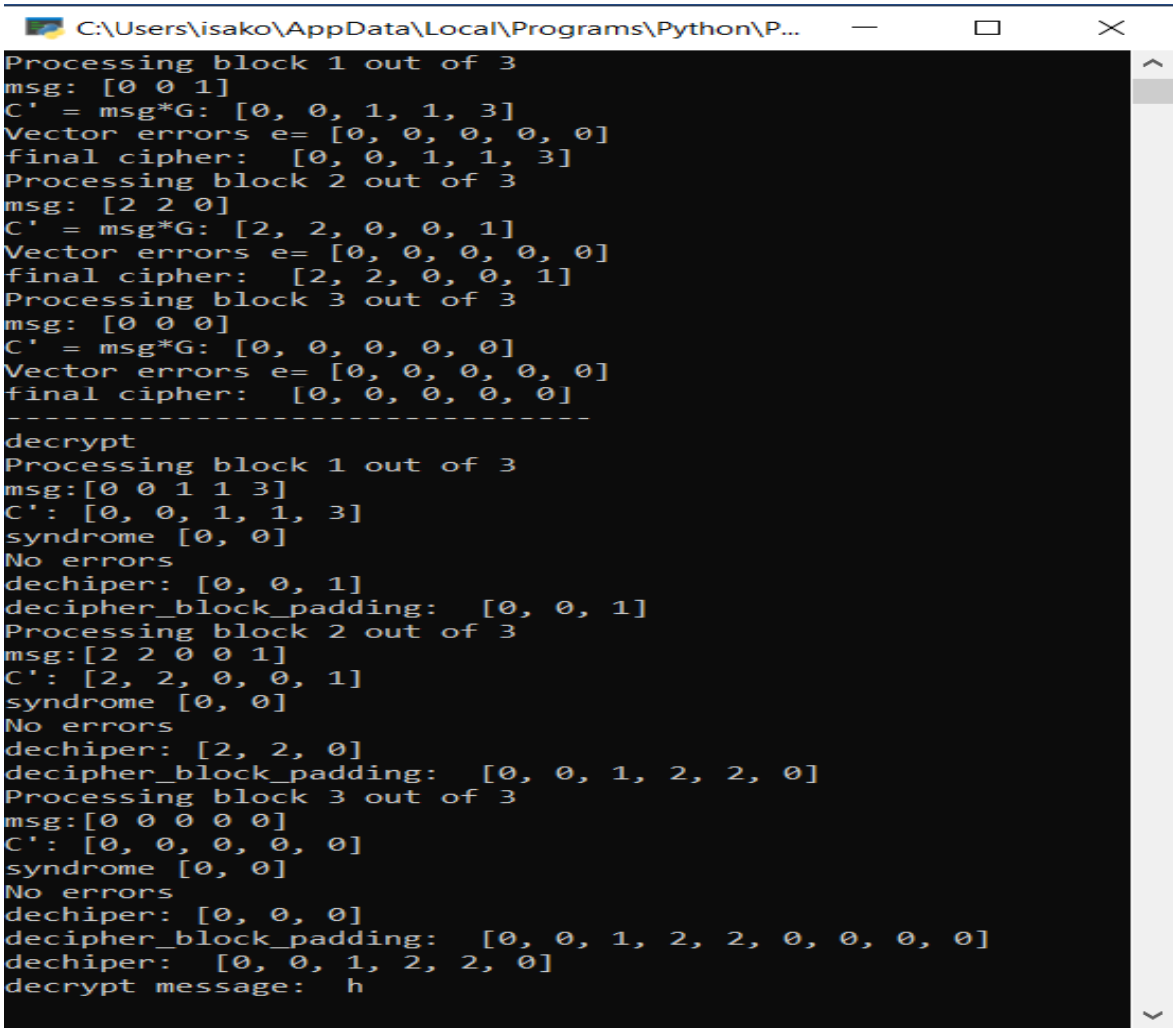

Рисунок 3.5 – Шифрування та дешифрування повідомлення з використанням McEliece

Гра була взята з попереднього диплому [3] тільки зі значними змінами, тому весь опис та функціонал можна прочитати там.

Була реалізована авторизація (рис 3.6). Інтерфейс та користування гри звичайне. Вводимо логін та пароль, і успішно входимо в гру, якщо ви не зареєстровані, то необхідно пройти реєстрацію.

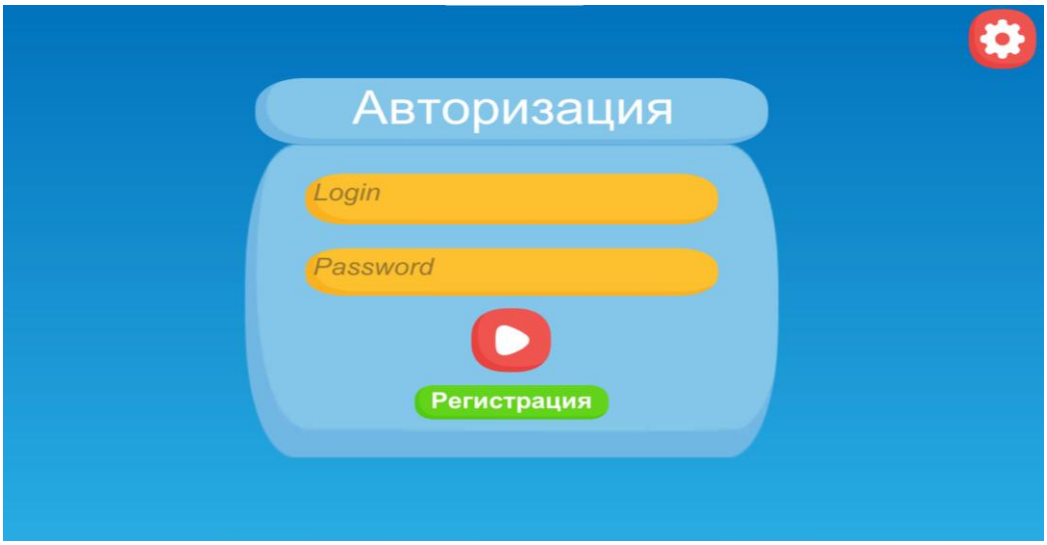

Рисунок 3.6 – Вхід в гру

Реєстрація ігрового профілю показана на рис.3.7.

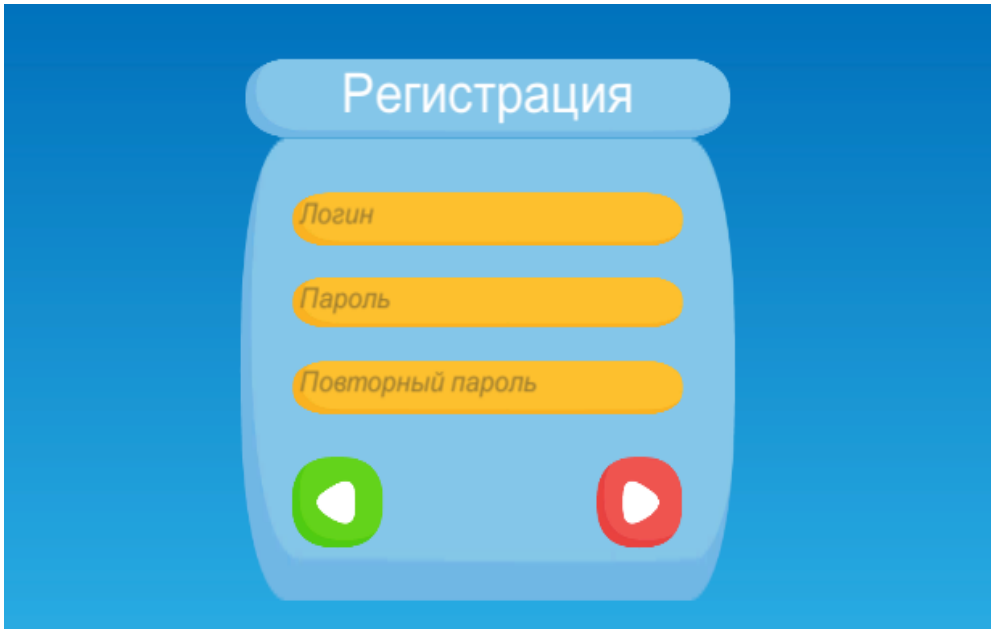

Рисунок 3.7 – Створення нового ігрового профілю

В грі реалізована передача даних на сервер (рис 3.8). Сервер має ір адресу 127.0.0.1 та порт 25565. Знаючи цю інформацію можна до нього підключатися, але це підключення вбудовану в гру тому змінити його не можна, але задати другі параметри сервера можна, наприклад 127.0.0.1 це ір адреса локальної мережі, але можна змінити адрес на глобальний, або поставити цей сервер на хостинг. Також

сервер містить логи, тому всю інформацію про шифрування та дешифрування пакетів буде видно на сервері.

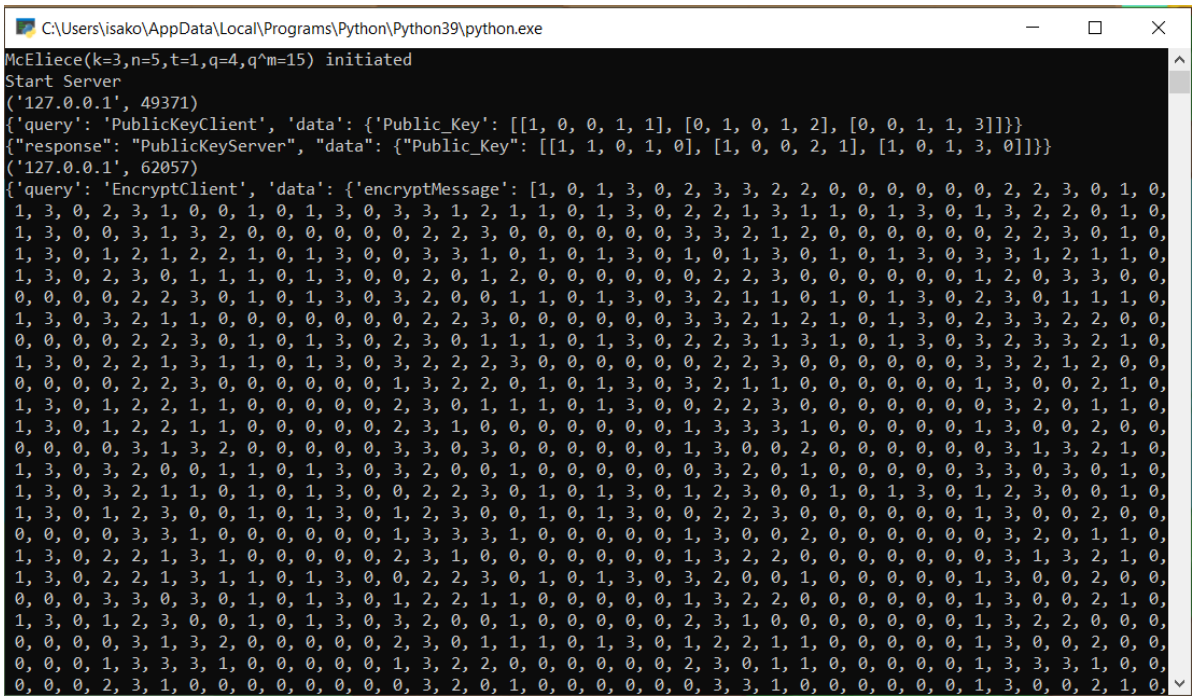

Рисунок 3.8 – Сервер гри

На сервері знаходиться база даних до якої надсилаються дані (рис 3.9). В базі даних створено дві таблиці users та heroes, але взагалі їх три, тому третя таблиця в доробці.

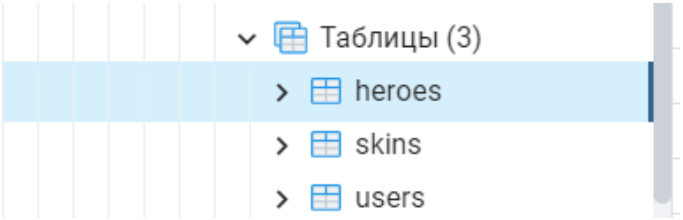

Рисунок 3.8 – Таблиці бази даних

Таблиця users містить id, логін та пароль, токен (рис 3.9). Тут зберігаються дуже важливі дані, адже атакувавши успішно базу даних, можна витягнути багато корисної інформації для подальшого взлому гри.

Результат | План выполнения | Сообщения | Notifications

	user_id [PK] integer	username text	password text	token text
1	1	Sparxes	9463d0aa45a9518581c0fef252c3ee55	a7e12f1ac7cfff82590686aea9bed928b0635ac5316e187f100be17ef8697ffc
2	2	Daz	4297f44b13955235245b2497399d7a93	435f563173e1166cb0d71b0225b7dd01e13be854276947559d82937e2c59837d
3	3	ISAKOV	25d55ad283aa400af464c76d713c07ad	b3c5374bfa652af2572a49ca5b68030a4c92b34b71acc7fe66c3b6df3a4fec61
4	4	dima	81dc9bdb52d04dc20036dbd8313ed055	80c3bad75ad9856f9a9c85f8539f2e1c1f57fe274c67f5c68167dc01f4fcdfc5
5	5	admin	9463d0aa45a9518581c0fef252c3ee55	53160686c065ad2b36af381edd32fb91c38adde4181280fc66702748620fb29e
6	6	Admin	25d55ad283aa400af464c76d713c07ad	015d48d3c68253e405eef131974511cc8c5bbeb7a65fc5f1943e8642391b64e0

Рисунок 3.7 – Таблиця users

Таблиця heroes відповідає за дані ігрового профілю (рис 3.8). Атакувавши цю таблицю можна повністю редагувати дані кожного профілю, наприклад, змінювати кількість ігрової валюти.

Результат | План выполнения | Сообщения | Notifications

	hero_id [PK] integer	coins integer	current_skin integer	user_id integer	skin_id integer
1	1	159	0	1	1
2	2	100	0	2	2
3	3	100	0	3	3
4	4	100	0	4	4
5	5	100	0	5	5
6	6	100	0	6	6
7	7	100	0	7	7
8	8	100	0	8	8
9	9	100	0	9	9

Рисунок 3.8 – Таблиця heroes

Wireshark – це аналізатор мережевих протоколів або програма, яка перехоплює пакети з мережевого підключення, наприклад, з комп'ютера в домашній офіс або в Інтернет. Пакет – це ім'я, дане дискретній одиниці даних у типовій мережі Ethernet.

Як і будь-який інший аналізатор пакетів, Wireshark робить три речі.

Захоплення пакетів. Wireshark прослуховує мережеве з'єднання в режимі реального часу, а потім захоплює цілі потоки трафіку - цілком можливо, десятки тисяч пакетів за один раз.

Фільтрування. Wireshark може нарізати всі ці випадкові живі дані за допомогою фільтрів. Використовуючи фільтр, ви можете отримати лише ту інформацію, яка вам потрібна.

Візуалізація. Wireshark, як і будь-який хороший аналізатор пакетів, дозволяє зануритися прямо в середину пакету мережі. Це також дозволяє вам візуалізувати цілі розмови та мережеві потоки.

Wireshark може захоплювати не тільки паролі, а й будь-яку інформацію, що проходить через мережу - імена користувачів, адреси електронної пошти, особисту інформацію, зображення, відео, що завгодно. Поки ми можемо перехоплювати мережевий трафік, Wireshark може перехоплювати паролі, що проходять. За допомогою Wireshark можна перехопити пакет з клієнта на сервер.

Головний недолік програми, це те що Wireshark лише збирає інформацію з мережі, але не може надсилати, тобто провести атаку людина по середині ми не зможемо, а тільки впевнитися в тому що пакет, який надходить з сервер до клієнта має зашифрований вигляд.

На рис 3.9 показаний перехват пакет авторизації гравця. Добре видно, що клієнт виконав дію авторизації, ми бачимо запит клієнт, тобто у пакеті вказані головні дані, логін пароль, але ці дані ніякої важливості не несуть, тому що пароль в захешованому вигляді.

Наступний пакет це відповідь сервера, в якому він вказує дозвіл на авторизацію, та токен для автентифікації. Якщо ці дані потраплять в руки зловмисника, то він з легкістю обійде авторизацію, та зайде під будь-яким користувачем, а саме підробивши пакет. Для вирішення цієї проблеми необхідно передавати дані по захищеному каналу.

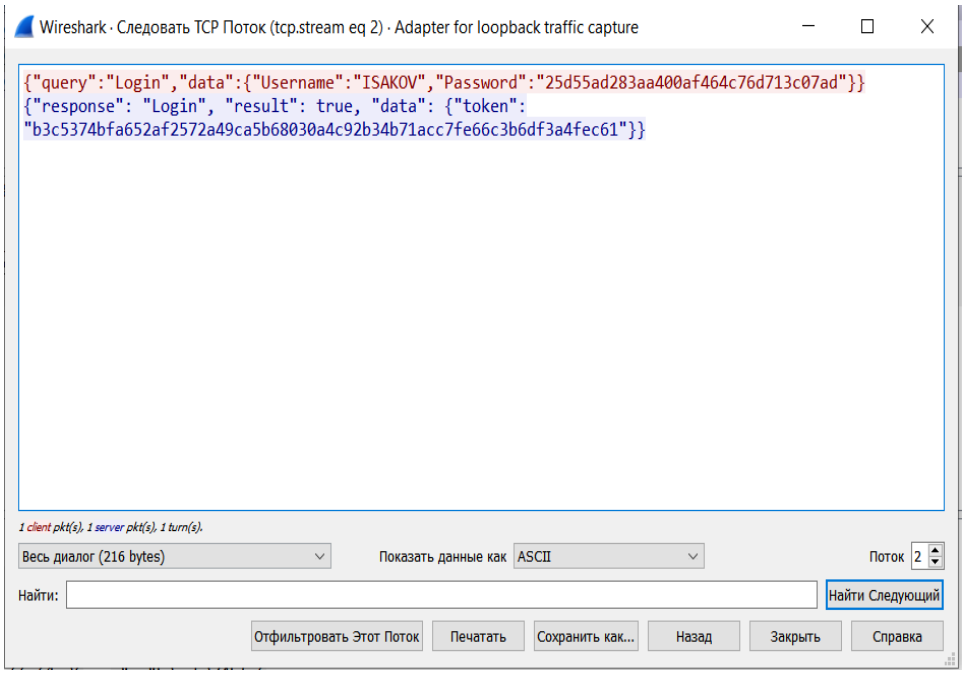

Рисунок 3.9 – Перехват незахищеного пакету

На рис 3.10 ми можемо побачити, що при вході в гру, як тільки користувач зайшов, тобто встановив з’єднання, генеруються публічні ключі, відбувається обмін ключами для подальшого шифрування. До цього моменту ключ передавався по незахищеному каналу.

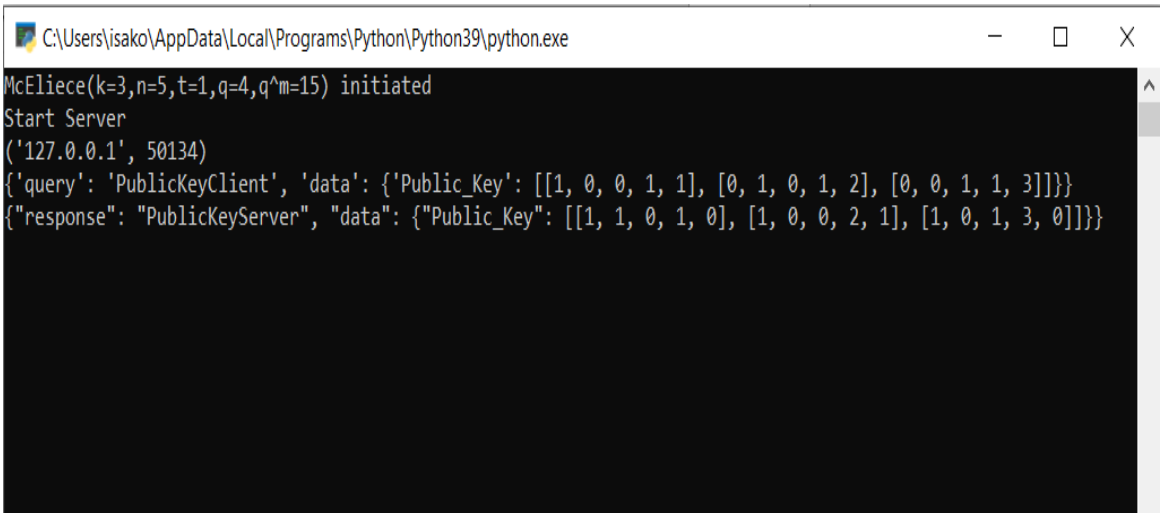

Рисунок 3.9 – Генерація ключів при першому з’єднанні

Далі утворюється захищений канал, і всі дані, які передаються мають зашифрований вигляд, тому реалізація атаки «Людина по середині» неможлива.

Рисунок 3.10 – Лог консолі

Проведемо повторно атаку на мережеву гру в яку вбудували модифіковану криптосистему McEliece з використанням недвійкових кодів Гемінґа (рис 3.11).

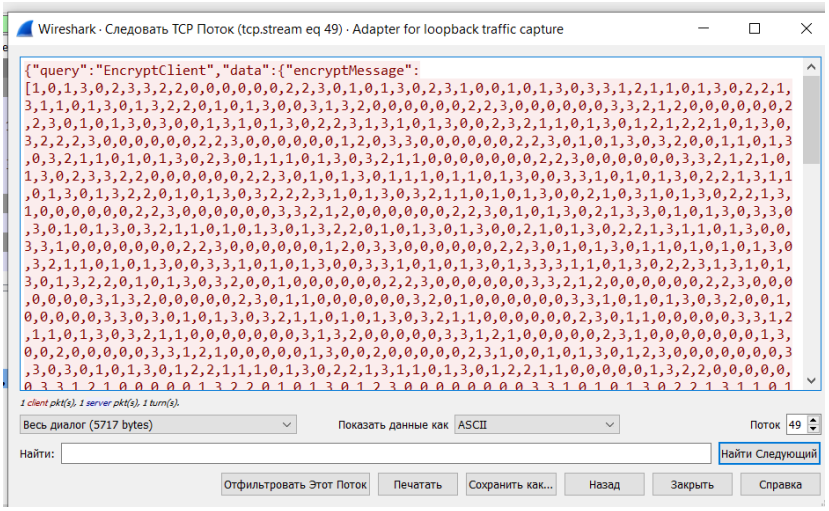

Рисунок 3.11 – Пакет в зашифрованому вигляді

Отже, атака як ми бачимо атака не відбулася, тому що всі дані надходять від клієнта до сервера, або від сервера до клієнта в зашифрованому вигляді.

3.3 Системні вимоги для коректної роботи програмного продукту

Для інсталяції цього програмного продукту потрібен комп'ютер з операційною системою Windows. Для правильної роботи програмного продукту повинні бути дотримані такі вимоги:

ОС: Windows 10;

Процесор: з тактовою частотою 1,5 ГГц;

Оперативна пам'ять: 512 МБ;

Жорсткий диск: 250 МБ вільного місця.

Місце на диску: 100 MB.

Серверна частина, яка була розроблена Python, має наступні вимоги:

ОС: Windows 10;

Процесор: з тактовою частотою 1,5 ГГц;

Оперативна пам'ять: 1024 МБ;

Жорсткий диск: 2048 МБ вільного місця.

Місце на диску: 1024 MB.

Наступна програма це база даних PostgreSQL, яку необхідно обов’язково встановити. Наступні вимоги до інсталяції припускають, що під час інсталяції ви вибрали параметри за замовчуванням.

Мінімальне обладнання, необхідне для встановлення та запуску PostgreSQL:

- процесор 1 ГГц;
- 2 Гб оперативної пам'яті;
- 512 МБ HDD.

PostgreSQL ґрунтується на процесах, а це означає, що він може виконувати буквально лише одну операцію для кожного процесу, для кожного ядра в кожний момент часу. Використання таких технологій як перемикання контексту допомагає, але невід'ємне фізичне обмеження все ще існує.

2 ГБ пам'яті – це рекомендована пам'ять, яку ви можете виділити для PostgreSQL поза операційною системою. Якщо у вас невеликий набір даних, вам все одно знадобиться достатньо пам'яті для кешування більшості ваших гарячих даних. Для даних або допоміжних компонентів потрібен додатковий простір на диску.

У розділі три описано модифіковану криптосистему McEliece. Гра була створена на платформі Unity, а серверна частина була написана мовою програмування python.

Також в даному розділі було проведено атаку на мережеву.

В третьому розділі також описано детальну інструкцію щодо користування криптосистемою, а також її окремими компонентами та вимоги програмного забезпечення для його швидкої та надійної роботи.

Отже, розроблена криптосистема McEliece (на двох різних мовах програмуваннях), гра, серверна частина, база даних.

В ході виконаної роботи в третьому розділі було вирішено всі поставлені задачі, а також був повністю реалізований другий розділ.

ВИСНОВКИ

В даній дипломній роботі був проведений огляд проблем криптосистеми McEliece, на рахунок цього було запропоновано модифікувати криптосистему McEliece. Дана модифікована криптосистема була застосована до мережевої гри.

Ігрова мережа – це відправлення та оптимізація (це те, що необхідно оновлювати з боку гравця чи сервера) мережевих пакетів, створення централізованого органу для запобігання шахрайству та розміщення гри. Вибір правильної мережі може бути складним, якщо ви не знайомі з основами мережі. Хоча ігрова мережа, що обговорювался в цій дипломній роботі, відноситься до Unity 3D, концепції ігрової мережі залишаються незмінними незалежно від ігрового двигуна.

Відзначимо основні результати проведених досліджень.

1. Запропоновано нові сімейства (*n*,*k*)-кодів Гемінга над розширеннями розширених полів $GF(q^r)$, де $q = 2^u, u = 2,3,\ldots$. Встановлено властивості даних кодів, їх здатність детектувати $f = 2$ помилок та коректувати $t = 1$ помилок. Встановлено особливості застосування декодування методом синдрому для розроблених сімейств кодів Гемінга.

2. Запропоновано асиметричну криптосистему МакЕліса на основі кодів Гемінга над розширеннями розширених полів. Показано, що застосування розширень розширених полів дозволяє забезпечити більше число можливих генераторних матриць коду Гемінга (тобто більше число рівнів захисту системи) при меншій довжині відкритого ключа.

3. Запропонована система, базуючись на класичній криптосистемі МакЕліса, є стійкою до потенційних атак квантового криптоаналізу, має менші значення довжини відкритого ключа, а отже може бути рекомендованою до практичного використання у застосунках, що потребують на ефективні та стійкі асиметричні криптографічні алгоритми.

ПЕРЕЛІК ПОСИЛАНЬ

1. Isakov D.A, Sokolov A.V. McEliece cryptosystem based on quaternary hamming codes. 2022.
2. Sokolov A.V. Isakov D. A. Authenticated encryption mode with blocks skipping. *Системный анализ и прикладная информатика*. 2021. № 3. С. 59-65.
3. Ісаков Д.А. Кваліфікаційна робота бакалавра. Розробка системи захисту ігор на платформі Unity. 2021.
4. Асиметричні алгоритми шифрування. Матеріал з вікіпедії – вільної енциклопедії. URL: uk.wikipedia.org/wiki/Асиметричні алгоритми шифрування.
5. Schneier B. Applied Cryptography: Protocols, Algorithms, and Source Code in C. Wiley,1996. 758 p.
6. Linear Code. Матеріал з вікіпедії – вільної енциклопедії. URL: https://en.wikipedia.org/wiki/Linear_code
7. Hamming, Richard Wesley "Error detecting and error correcting codes". Bell System Technical Journal. 1950, 147–160.
8. Dipperstein M. Hamming (7,4) Code Discussion and Implementation. URL: michaeldipperstein.github.io/hamming.html#example3.
9. Parity check matrix. Матеріал з вікіпедії – вільної енциклопедії. URL: en.wikipedia.org/wiki/Parity-check_matrix.
10. McEliece cryptosystem. Матеріал з вікіпедії – вільної енциклопедії. URL: en.wikipedia.org/wiki/McEliece_cryptosystem.
11. Мазурков М.І. Основи теорії передавання інформації. Одеса: Наука і Техніка , 2005. 168 с.
12. Yiyi H., Junbiao D. Quaternary checksum, redundancy and Hamming code. Preprint at Researchgate. 2022. P. 1-8.
13. Мазурков М.И., Конопака Е.А. Семейства линейных рекуррентных последовательностей на основе полных множеств изоморфных полей Галуа. *Известия ВУЗов. Радиоэлектроника*. 2005. № 11. С. 58-65.

14. Mazurkov M.I., Sokolov A.V. Nonlinear transformations based on complete classes of isomorphic and automorphic representations of field GF(256). *Radioelectronics and Communications Systems*. 2013. Vol. 56, No. 11. P. 513-521.
15. Kabeya T.C. McEliece's Crypto System based on the Hamming Cyclic Codes. *International Journal of Innovative Science and Research Technology*. 2019. Vol. 4, No 7. P. 293-296.
16. Madhav S., Glaze J. Multiplayer Game Programming: Architecting Networked Games (Game Design), Addison-Wesley Professional;2015.
17. OSI MODEL. Матеріал з вікіпедії – вільної енциклопедії. URL: wikipedia.org/wiki/OSI_model.
18. Hayes A. Peer-to-Peer (P2P) Service: Definition, Facts, and Examples URL: investopedia.com/terms/p/peertopeer-p2p-service.asp.
19. Pollicove M. Ultimate Guide to Token-based Authentication. URL: pingidentity.com/en/resources/blog/post/ultimate-guide-token-based-authentication.html.
20. C# Sharp. Матеріал з вікіпедії – вільної енциклопедії. URL: uk.wikipedia.org/wiki/C_Sharp.
21. Unity – ігровий рушій. Матеріал з вікіпедії – вільної енциклопедії. URL: uk.wikipedia.org/wiki/Unity_(рушій_гри).
22. Jennings N. Socket Programming in Python. URL: realpython.com/python-sockets.
23. PostgreSQL. Матеріал з вікіпедії – вільної енциклопедії. URL: uk.wikipedia.org/wiki/PostgreSQL.
24. Kinzie K. How to Use Wireshark: Comprehensive. URL: https://www.varonis.com/blog/how-to-use-wireshark.

Додаток А. Лістинг програмного продукту

Модуль №1 gen_parity_check.py

```
import numpy as np

def family_Hamming(q,r):
    n = int((q**r-1)/(q-1))
    k = int((q**r-1)/(q-1)-r)
    res = (n,k)
    return res

BASE='0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ' #until base 36, because
why not :)

def base(n,b,acc=None):
    if acc is None:  # I need a accumulator, if I don't get one I
provide one
        acc=[]
    if n < b:
        acc.append(BASE[n])
        return "".join(reversed(acc))
    else:
        n,d = divmod(n,b)      # n//b and n%b at once
        acc.append(BASE[d])
        return base(n,b,acc)

def gen_parity_check(_r):
    count_prev = 0

    res = np.empty((0, _r), int)
    for r in range(1,_r+1):

        _l = int((4**r-1)/3)
        x_add = _l-count_prev-1
        print("r = {0}, n = {1}, comb_add={2}".format(r,_l,x_add))

        if(r==1):
            a = np.ones((x_add,1),dtype=int)
            b = np.zeros((x_add,_r-r),dtype=int)
            first_vector = np.column_stack([b, a])

            #res = np.append(res,first_vector,0)

        elif(r!=_r):
            a = np.ones((x_add,1),dtype=int)

            #Генерация всех возможных комбинаций при r семейства
            b = []
            for i in range(1, x_add+1):
                four = base(i,4).zfill(r-1)
```

```
                res2 = []

                for j in range(len(four)):
                    res2.append(int(four[j]))
                b.append(res2)
            c = np.zeros((x_add,_r-r),dtype=int)

            res1 = np.column_stack([a, b])
            res2 = np.column_stack([c, res1])
            res = np.append(res,res2,0)



        if(r == _r):
            a = np.ones((x_add,1),dtype=int)

            #Генерация всех возможных комбинаций при r семейства
            b = []
            for i in range(1, x_add+1):
                four = base(i,4).zfill(r-1)
                res2 = []

                for j in range(len(four)):
                    res2.append(int(four[j]))
                b.append(res2)

            res1 = np.column_stack([a, b])
            res = np.append(res,res1,0)

        print(res)
        count_prev = _l

    family_Ham = family_Hamming(4,_r)
    n = family_Ham[0]
    k = family_Ham[1]
    arr_eye = np.eye(r,dtype=int)
    res = np.row_stack([ res,arr_eye])
    arr_eye = []
    h = res.transpose()

    arr_eye_g = np.eye(k,dtype=int)
    g = np.column_stack([arr_eye_g,res[:k]])
    return (g, h)

r = int(input("Input r: "))
(g,h) = gen_parity_check(r)
print("H="+"\n",h)
print("G="+"\n", g)
```

Модуль №2 mceliececipher.py

```
import logging
```

```
from hamming.hammingcodegenerator import *
from mathutils import *
import galois
log = logging.getLogger("ntrucipher")

GF = galois.GF(2**2)

class McElieceCipher:

    def __init__(self, r, t=1):
            self.r = r
            self.q = 4
            self.n = int((self.q**r-1)/(self.q-1))
            self.t = t
            self.G = None
            self.H = None
            self.k = int((self.q**self.r-1)/(self.q-1)-self.r)
            self.P = None
            self.P_inv = None
            self.S = None
            self.S_inv = None
            self.e = None

print(f"McEliece(k={self.k},n={self.n},t={self.t},q={self.q},q^m={se
lf.q ** self.r-1}) initiated")

    def generate_random_keys(self):

        self.G, self.H = gen_parity_check(self.r)
        self.k = self.k = self.G.shape[0]
        self.P = GF(np.eye(self.n,dtype=int))
        self.P_inv = GF(np.linalg.inv(self.P))
        self.S = GF(np.eye(self.k,dtype=int))
        self.S_inv = GF(np.linalg.inv(self.S))

        matrix = np.dot(self.S,GF(self.G))
        self.Gp = np.dot(matrix,self.P)

        print("H:",self.H)
        print('G:\n',self.G)
        print('S:\n',self.S)
        print('P:\n',self.P)
        print('\nP^{-1}:\n',self.P_inv)
        print ('\nS^{-1}:\n',self.S_inv)
        print('Result (GSP):\n',self.Gp)

    def encrypt(self, msg_arr):
        if len(msg_arr) != self.Gp.shape[0]:
            raise Exception(f"Wrong message length. Should be
{self.Gp.shape[0]} bits.")
        print(f"msg: {msg_arr}")
        Cp = np.matmul(GF(msg_arr),self.Gp)
```

```
        print(f"C' = msg*G: {Cp}")
        self.e = GF(np.zeros(self.n,dtype=int))
        self.e[0] = 0
        print("Vector errors e=",self.e)
        Cp+=self.e
        print("final cipher: ", Cp)
        #bits_to_flip = np.random.choice(len(Cp), size=self.t,
replace=False)
        # # log.debug(f"bits_to_flip: {bits_to_flip}")
        # for b in bits_to_flip:
        #     Cp[b] = Cp[b].flip()
        # log.debug(f"C': {Cp}")
        return Cp

    def decode(self, msg_arr):

        syndrome = np.matmul(GF(self.H),np.transpose(msg_arr))
        print("syndrome",syndrome)
        if not all(syndrome == 0):
            ee = self.e*self.P_inv
            msg_arr = msg_arr-self.e
        else: print("No errors")

        return msg_arr

    def decrypt(self, msg_arr):

        if len(msg_arr) != self.H.shape[1]:
            raise Exception(f"Wrong message length. Should be
{self.H.shape[1]} bits.")
        Cp = np.matmul(GF(msg_arr),self.P_inv)
        print(f"C': {Cp}")
        Cp_dec = self.decode(Cp)

        decipher = np.matmul(Cp_dec[0:self.k],self.S_inv)
        print("dechiper:",decipher)
        return decipher
```

Модуль №3 padding.py

```
import numpy as np

def padding_encode(input_arr, block_size):
    n = block_size - len(input_arr) % block_size
    if n == block_size:
        return np.pad(input_arr, (0, n), 'constant')
    last_block = np.pad(np.ones(n,dtype=int), (block_size - n, 0),
'constant')
    return np.concatenate((np.pad(input_arr, (0, n), 'constant'),
last_block))
```

```
def padding_decode(input_arr, block_size):
    last_block = input_arr[-block_size:]
    zeros_to_remove = len(np.trim_zeros(last_block))
    return input_arr[:-(block_size + zeros_to_remove)]
```

Модуль №4 mceliece.py

```
from mceliececipher import McElieceCipher
import numpy as np
import logging
from  mc.padding.padding import *
from hamming.hammingcodegenerator import *

BASE='0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ' #until base 36, because
why not :)

def base(n,b,acc=None):
    if acc is None:  # I need a accumulator, if I don't get one I
provide one
        acc=[]
    if n < b:
        acc.append(BASE[n])
        return "".join(reversed(acc))
    else:
        n,d = divmod(n,b)      # n//b and n%b at once
        acc.append(BASE[d])
        return base(n,b,acc)

log = logging.getLogger("mceliece")


def encrypt(input_arr, block=False):


    if not block:
        if mceliece.Gp.shape[0] < len(input_arr):
            raise Exception(f"Input is too large for current N.
Should be {mceliece.Gp.shape[0]}")
        output = mceliece.encrypt(input_arr)
    else:
        input_arr = padding_encode(input_arr, mceliece.Gp.shape[0])
        input_arr = input_arr.reshape((-1, mceliece.Gp.shape[0]))
        output = np.array([])
        block_count = input_arr.shape[0]
        for i, b in enumerate(input_arr, start=1):
            print("Processing block {} out of {}".format(i,
block_count))
            next_output = mceliece.encrypt(b)
            if len(next_output) < mceliece.Gp.shape[1]:
                print(f"Padding added in block {i}")
                next_output      =      np.pad(next_output,      (0,
mceliece.Gp.shape[1] - len(next_output)), 'constant')
```

```
            output = np.concatenate((output, next_output))

    return np.array(output,dtype=int).flatten()

def decrypt(input_arr, block=False):
    if not block:
        if len(input_arr) < mceliece.H.shape[1]:
            input_arr = np.pad(input_arr, (0, mceliece.H.shape[1] -
len(input_arr)), 'constant')
        return mceliece.decrypt(input_arr)

    if len(input_arr)%mceliece.H.shape[1] > 0:
        input_arr = np.pad(input_arr, (0, mceliece.H.shape[1] -
len(input_arr)%mceliece.H.shape[1]), 'constant')
    input_arr = input_arr.reshape((-1, mceliece.H.shape[1]))
    output = np.array([],dtype=int)
    block_count = input_arr.shape[0]
    for i, b in enumerate(input_arr, start=1):
        print("Processing block {} out of {}".format(i,
block_count))
        print(f"msg:{b}")
        next_output = mceliece.decrypt(b)
        if len(next_output) < mceliece.G.shape[0]:
            print(f"Padding added in block {i}")
            next_output = np.pad(next_output, (0,
mceliece.H.shape[1] - len(next_output)), 'constant')
        output = np.concatenate((output, next_output))
        print("decipher_block_padding: ",output)
    return padding_decode(output.flatten(), mceliece.G.shape[0])

text = input("Input message: ")
r = int(input("Input r: "))
text_to_four = []
print("Message four block: ")
for i in range(0,len(text)):
    letter = ord(text[i])


    four_num = base(letter,4)
    if(len(four_num)<6):
        four_num = four_num.zfill(6)

    a = [int(x) for x in four_num]
    text_to_four.append(a)
    print(four_num)

text_to_four = np.array(text_to_four)
text_to_four = text_to_four.reshape((1,
text_to_four.shape[0]*text_to_four.shape[1]))[0]
mceliece = McElieceCipher(r, 1)
mceliece.generate_random_keys()
a = encrypt(text_to_four,True)
```

```
print("--------------------------------")
print("decrypt")
dechiper = decrypt(a,True)
print("dechiper: ",dechiper)

#decrypt
four_to_text = []
res = ""

for i in range(0,len(dechiper),6):
    four = dechiper[i:i+6]
    temp=""
    for j in range(0,len(four)):
        temp+=str(four[j])
    letter = chr(int(temp,4))
    res+=letter
print("decrypt message: ", res)
```

Модуль №5 server.py

```
import socket
import json
import sqlite3
import psycopg2
import secrets
from colorama import Fore, Back, Style
from mc.mceliece import *

def isAuth(conn,cursor):
    #Запрос на имя пользователя в базе
    cursor.execute("""select token, username from users
where users.token = '{0}'""".format(data["data"]["token"]))
    #Получаем результат
    records = cursor.fetchall()[0]
    load_token=data["data"]["token"]
    real_token=records[0]
    username=records[1]
    if(load_token==real_token):
        seng_data                                                    =
json.dumps({"response":"isAuth","result":True,"data":{"Username":use
rname}})
        print("Server send: ",seng_data)
    else:
        seng_data                                                    =
json.dumps({"response":"isAuth","result":False,"data":{"token":load_
token}})
        print("Server send: ",seng_data)
    return seng_data

def LoadData(conn,cursor):

    cursor.execute("""
```

```
SELECT
    json_build_object(
          'id', users.user_id,
          'username', users.username,
        'coins', heroes.coins,
        'Player', heroes.current_skin
    )
FROM heroes
JOIN users ON users.user_id = heroes.user_id
JOIN skins ON skins.skin_id = heroes.skin_id
Where users.username='{0}'""".format(data["data"]["Username"]))
    #Получаем результат

    records = cursor.fetchall()[0][0]
    #Если пусто значит пользователя нет в базе, иначе есть
    seng_data    =    json.dumps({"response":"LoadData","result":True,
"data":records})
    print("Server send: ",seng_data)
    return seng_data


def SaveData(conn,cursor):
    print(json.loads(data["data"]["GameData"])["coins"])
    id = json.loads(data["data"]["GameData"])["id"]
    coins = json.loads(data["data"]["GameData"])["coins"]
    #Запрос на имя пользователя в баз
    cursor.execute("""UPDATE heroes SET coins = '{0}' where hero_id
= '{1}' ;""".format(coins,id))
    seng_data="SaveData"
    return seng_data

def LoginPlayer(conn,cursor):
    #Запрос на имя пользователя в базе
    cursor.execute("""select username, password from users
where users.Username = '{0}'""".format(data["data"]["Username"]))
    #Получаем результат
    records = cursor.fetchall()
    #Если пусто значит пользователя нет в базе, иначе есть
    if(len(records)==0):
        #Формирую результат и отправляю на клиент
        seng_data = json.dumps({"response":"Login","result":False})
        #                          encrypt_data                          =
encrypt(mceliece_cipher,convertMessage_to_four(seng_data),True).toli
st()
        #    seng_data    =    json.dumps({"response":"EncryptServer",
"data":{"encryptMessage":encrypt_data}})

        print("Server send: "+seng_data)
        # socket_fuwu.send(seng_data.encode('utf-8'))
    #Пользователь есть в базе
    else:
```

```
        #Логин и пароль с базы
        username_load=records[0][0]
        password_load=records[0][1]
        #Полученные данные с коиента
        username_real=data["data"]["Username"]
        password_real=data["data"]["Password"]
        #Сравнение данных
        if(password_real==password_load):
            token=secrets.token_hex(32)
            cursor.execute("""UPDATE users SET token = '{0}'
            where Username = '{1}'""".format(token,username_load))
            print(records)
            seng_data                                                =
json.dumps({"response":"Login","result":True,
"data":{"token":token}})
            print("Server send: ",seng_data)

        else:
            print("Login no")
            seng_data                                                =
json.dumps({"response":"Login","result":False, "data":{"1":"2"}})
            print("Server send: ",seng_data)
    return seng_data

def RegisterPlayer(conn,cursor):
    cursor.execute("""select username, password from users
where users.Username = '{0}'""".format(data["data"]["Username"]))
    records = cursor.fetchall()
    print(len(records))
    if(len(records)==0):
        username_real=data["data"]["Username"]
        password_real=data["data"]["Password"]
        player_data={"Username":         "{0}".format(username_real),
"money": 100}
        cursor.execute("""INSERT INTO users  (Username,password)
        VALUES('{0}','{1}');""".format(username_real,password_real))
        cursor.execute("""INSERT      INTO      skins      (red,blue)
VALUES('{0}',{1});""".format(1,0))
        cursor.execute("""INSERT  INTO  heroes  (coins,current_skin)
VALUES('{0}',{1});""".format(100,0))
        seng_data                                                =
json.dumps({"response":"Register","result":True})
        print("Server send: "+seng_data)
        # socket_fuwu.send(seng_data.encode('utf-8'))
    else:
        #                          seng_data                       =
json.dumps({"response":"Register","result":False, "data":{"1":"2"}})
        print("Server send: ",seng_data)
    return seng_data


tcp_socket_host = socket.socket(socket.AF_INET,socket.SOCK_STREAM)
```

```
mceliece_cipher = McElieceCipher(2, 1)
mceliece_cipher.generate_random_keys()
#                              ciphertext                                  =
encrypt(mceliece_cipher,convertMessage_to_four("h"),True)
# 2 Порт привязки
host = "127.0.0.1"
port=25565
tcp_socket_host.bind((host,port))
r = 2
key_client = []
_key = []
# 3 монитор становится пассивной розеткой
tcp_socket_host.listen(7)      # 128Максимальное количество, которое
можно отслеживать, максимальное количество ссылок
flag = True
print ('Start Server')
while True:
    # 4 Подождите, пока клиент подключится
    socket_fuwu,addr_client=tcp_socket_host.accept()
#accept(new_socket,addr)
    print(addr_client)
    #print(socket_fuwu)
    # 5 читать и писать
    conn = psycopg2.connect(dbname='GameData', user='postgres',
                            password='711976', host='localhost')
    cursor = conn.cursor()
    recv_data = socket_fuwu.recv(4096)
    #print(recv_data.decode('utf-8'))
    data=json.loads(recv_data)
    print(data)
    if (data["query"] == "EncryptClient"):
        dechiper                                                            =
decrypt(mceliece_cipher,data["data"]["encryptMessage"],True)
        data = json.loads(convertFour_to_Message(dechiper))
        print("Server decrypt message: ", data)
        pass

    elif(data["query"]=="PublicKeyClient"):
        #print("PublicKeyClient:", data["data"]["Public_Key"])
        key_client = data["data"]["Public_Key"]
        mceliece_cipher.generate_random_keys()
        key_server = mceliece_cipher.Gp
        mceliece_cipher.Gp = mceliece_cipher.GF(key_client)
        _key = [[int(x) for x in line] for line in key_server]
        #print("PublicKeyServer: ", _key)
        seng_data     =     json.dumps({"response":"PublicKeyServer",
"data":{"Public_Key":_key}})
        socket_fuwu.send(seng_data.encode('utf-8'))

        print(seng_data)
        socket_fuwu.send(seng_data.encode('utf-8'))
```

```
    if(data["query"]=="Login"):

        seng_data=LoginPlayer(conn,cursor)

        encrypt_data                                                    =
encrypt(mceliece_cipher,convertMessage_to_four(seng_data),True).toli
st()
        seng_data      =      json.dumps({"response":"EncryptServer",
"data":{"encryptMessage":encrypt_data}})

        socket_fuwu.send(seng_data.encode('utf-8'))
        print(seng_data)

    elif(data["query"]=="Register"):
        seng_data=RegisterPlayer(conn,cursor)
        encrypt_data                                                    =
encrypt(mceliece_cipher,convertMessage_to_four(seng_data),True).toli
st()
        seng_data      =      json.dumps({"response":"EncryptServer",
"data":{"encryptMessage":encrypt_data}})
        socket_fuwu.send(seng_data.encode('utf-8'))

    elif(data["query"]=="isAuth"):
        seng_data=isAuth(conn,cursor)
        encrypt_data                                                    =
encrypt(mceliece_cipher,convertMessage_to_four(seng_data),True).toli
st()
        seng_data      =      json.dumps({"response":"EncryptServer",
"data":{"encryptMessage":encrypt_data}})
        socket_fuwu.send(seng_data.encode('utf-8'))
        print(seng_data)


    elif(data["query"]=="LoadData"):
        seng_data=LoadData(conn,cursor)
        encrypt_data                                                    =
encrypt(mceliece_cipher,convertMessage_to_four(seng_data),True).toli
st()
        seng_data      =      json.dumps({"response":"EncryptServer",
"data":{"encryptMessage":encrypt_data}})
        print("Server send: ",seng_data)

        socket_fuwu.send(seng_data.encode('utf-8'))

    elif(data["query"]=="SaveData"):
        seng_data=SaveData(conn,cursor)
        encrypt_data                                                    =
encrypt(mceliece_cipher,convertMessage_to_four(seng_data),True).toli
st()
        seng_data      =      json.dumps({"response":"EncryptServer",
"data":{"encryptMessage":encrypt_data}})
```

```
    print("Server send: ",seng_data)
    #socket_fuwu.send(seng_data.encode('utf-8'))
conn.commit()
```