МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
НАЦІОНАЛЬНИЙ АВІАЦІЙНИЙ УНІВЕРСИТЕТ
КАФЕДРА КОМП'ЮТЕРИЗОВАНИХ СИСТЕМ ЗАХИСТУ ІНФОРМАЦІЇ

ДОПУСТИТИ ДО ЗАХИСТУ
Завідувач кафедри Комп'ютеризованих
систем захисту інформації
_____Михайло СТЕПАНОВ
«_____»_____ 2023 р.

На правах рукопису
УДК 004.056.5:510.22(043.3)

КВАЛІФІКАЦІЙНА РОБОТА

ЗДОБУВАЧА ВИЩОЇ ОСВІТИ
ОСВІТНЬОГО СТУПЕНЯ «МАГІСТР»

Тема: Програмний модуль для захищеної передачі даних

Виконавець: Ярослав ЛОТИШ
Керівник: д.т.н., проф., зав каф. Михайло СТЕПАНОВ
Консультант розділу «Охорона навколишнього середовища»: к.т.н., доцент Тетяна ДМИТРУХА
Нормоконтролер: д.т.н., проф., зав каф. Михайло СТЕПАНОВ

Київ 2023

НАЦІОНАЛЬНИЙ АВІАЦІЙНИЙ УНІВЕРСИТЕТ

Факультет: Кібербезпеки та програмної інженерії

Кафедра: Комп'ютеризованих систем захисту інформації

Освітній ступінь: Магістр

Спеціальність: 125 «Кібербезпека»

Освітньо-професійна програма: «Безпека інформаційних і комунікаційних систем»

ЗАТВЕРДЖУЮ

Завідувач кафедри Комп'ютеризованих систем захисту інформації

_____ Михайло СТЕПАНОВ

«____» _____2023 р.

ЗАВДАННЯ

на виконання кваліфікаційної роботи

здобувача вищої освіти Лотиша Ярослава Сергійовича

1. Тема: *Програмний модуль для захищеної передачі даних* затверджена наказом ректора від «15» 09 2023 № 1814/ст.
2. Термін виконання з 16.10.2023 по 31.12.2023
3. Вихідні дані: сучасний стан програм захищеної передачі даних. Алгоритми шифрування даних. Засоби розробки програм.
4. Зміст пояснювальної записки: захищена передача даних. Проектування програмного модуля для захищеної передачі даних.

5. КАЛЕНДАРНИЙ ПЛАН

виконання кваліфікаційної роботи

№ з/п	Етапи виконання кваліфікаційної роботи	Термін виконання етапів	Примітка
1.	Уточнення постановки задачі	16.10.23 – 20.10.23	*Виконано*
2.	Аналіз наявних засобів для шифрування електронних повідомлень	20.10.23 –26.10.23	*Виконано*
3.	Аналіз літературних джерел	26.10.23 –01.11.23	*Виконано*
4.	Аналіз методів шифрування файлів	01.11.23 –08.11.23	*Виконано*
5.	Ознайомлення з технологіями для розробки програмного забезпечення	08.11.23 –17.11.23	*Виконано*
6.	Проектування алгоритмів основних функції програми	17.11.23 –22.11.23	*Виконано*
7.	Програмна реалізація розроблених алгоритмів	22.11.23 30.11.23	*Виконано*
8.	Тестування розробленого програмного засобу	30.11.23 –05.12.23	*Виконано*
9.	Оформлення пояснювальної записки та графічного матеріалу	05.12.23 –22.12.23	*Виконано*

6. Консультанти з окремих розділів

Розділ	Консультант (посада, П.І.Б.)	Дата, підпис	
		Завдання видав	Завдання прийняв
Охорона навколишнього середовища	Дмитруха Т.І.		

7. Дана видачі завдання: «16» жовтня 2023 р.

Здобувач вищої освіти Ярослав ЛОТИШ

(підпис, дата)

Керівник кваліфікаційної роботи Михайло СТЕПАНОВ

(підпис, дата)

РЕФЕРАТ

Кваліфікаційна робота складається зі вступу, трьох розділів, загальних висновків, списку використаних джерел, додатків і має 75 сторінки основного тексту, 2 рисунка, 1 таблиця, 16 сторінок додатків. Список використаних джерел містить 9 найменувань і займає 1 сторінку. Загальний обсяг роботи 92 сторінки.

Метою роботи є розробка та реалізація програмного модуля, призначеного для захищеної передачі даних через мережеві з'єднання з використанням шифрування.

В роботі розроблено алгоритм та програмне забезпечення для захищеної передачі даних.

Розроблений алгоритм та програмне забезпечення відноситься до галузі кібербезпеки і може бути використане для підвищення рівня захищеності.

Питання захищеної передачі даних з'явилось разом з самим терміном шифрування. Використання програмного модуля для захищеної передачі даних покращує безпеку, проте при умові, що питання ключів вирішено належним чином. З розвитком технічного прогресу з'явились інструменти, які полегшують перехоплення даних. Програмний модуль захищеної передачі даних створює додатковий рівень шифрування, тому тема кваліфікаційної роботи «Програмний модуль для захищеної передачі даних» є актуальною.

Результати дипломного проектування рекомендується використовувати для забезпечення додаткової безпеки в підприємствах, державних установах , для яких важливим є питання захисту та в освітньому процесі НАУ.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ СКОРОЧЕНЬ

AES	Це стандарт шифрування, який використовує блочний шифр.
CBC	Режим роботи блочного шифру, де кожний блок даних комбінується з попереднім блоком перед шифруванням.
Crypto++	Це бібліотека криптографії з відкритим кодом для мови програмування *C++*.
DES	Це стандарт шифрування, який використовує блочний шифр з ключем фіксованої довжини.
ECB	Простий режим роботи блочного шифру, де кожен блок даних шифрується незалежно.
3DES	Розширення стандарту *DES*, використовується три послідовно застосованих ключів для посилення безпеки.
FTPS	Розширення протоколу *FTP* за допомогою *TLS* або *SSL* для шифрування передачі даних.
IoT	Мережа взаємопов'язаних пристроїв, вбудованих сенсорів, для обміну даними через Інтернет.
JSON	Формат обміну даними, базується на текстовому представленні структурних даних.
Linux	Це операційна система з відкритим кодом, що базується на ядрі *Linux*
MAC	Адреса керування доступом до мережі, унікальний ідентифікатор призначеного для мережевого обладнання
OpenSSL	Це відкрита бібліотека криптографії, яка надає інструменти для реалізації різноманітних криптографічних протоколів та алгоритмів

OpenVPN	Віртуальна приватна мережа з відкритим вихідним кодом, яка забезпечує безпечне з'єднання мережі через шифрування та тунелювання.
RSA	Це асиметричний криптографічний алгоритм, використовується для створення ключів та підпису даних.
SFTP	Це протокол передачі файлів, який використовує SSH для забезпечення захищеної та шифрованої передачі даних між комп'ютерами.
SSL	Це криптографічний протокол, призначений для забезпечення безпечної комунікації через мережу Інтернет
TCP	Це протокол транспортного рівня, який забезпечує надійну та послідовну передачу даних в мережі.
TLS	Криптографічний протокол, що визначає параметри безпечної комунікації, є наступником протоколу SSL.
UDP	Це протокол транспортного рівня, який забезпечує передачу даних без гарантії надійності та послідовності.
VPN	Це технологія, що забезпечує безпечне та конфіденційне з'єднання мережі через відкриті мережі, такі як Інтернет.
Windows	Це операційна система, розроблена корпорацією Microsoft, популярна серед користувачів особистих комп'ютерів та у робочих середовищах.
XML	Мова розмітки, призначена для зберігання та передачі структурованої інформації.

ВСТУП

У сучасному цифровому світі, де інформація є ключовим ресурсом, актуальність передачі даних у шифрованому вигляді стає надзвичайно важливою. Інтернет і мережеві технології використовуються у всіх аспектах нашого життя, починаючи від електронної пошти та соціальних мереж, закінчуючи фінансовими транзакціями та медичною інформацією. Захищена передача даних в шифрованому вигляді є необхідною для того, щоб убезпечити від ризиків кібератак, ідентифікаційного крадіжку, а також неприпустимої втрати конфіденційності особистої і корпоративної інформації.

Додатково, у зв'язку з посиленою увагою до регуляторної рамки, такої як Загальний регламент з захисту даних (GDPR), організації та підприємства зобов'язані забезпечувати високий рівень захисту даних своїх клієнтів і партнерів. Невдачі у забезпеченні безпеки даних можуть призвести до серйозних юридичних наслідків та втрати довіри громадськості. Таким чином, шифрування даних стає необхідним елементом стратегії кібербезпеки, щоб ефективно захистити важливі інформаційні ресурси від сучасних кіберзагроз.

Шифрування даних використовується для забезпечення конфіденційності та безпеки інформації під час її передачі або зберігання. Основною метою шифрування є перетворення звичайного тексту (даних) в нерозбірливий формат за допомогою спеціального алгоритму та ключа. Це перешкоджає несанкціонованому доступу та забезпечує лише власникові або дозволеним користувачам можливість читати чи розшифровувати інформацію.

Крім того, шифрування важливе для захисту від різноманітних кіберзагроз, таких як перехоплення даних, атаки "людина по середника" та інші форми хакерських атак. Цей захисний шар забезпечує безпечну передачу чутливої інформації, яка може включати особисті дані, банківські інформації, корпоративні секрети чи будь-яку іншу конфіденційну деталі. Загалом, шифрування даних є ключовим елементом стратегії кібербезпеки, що забезпечує

надійний захист від потенційних загроз та зберігає довіру в обробці та обміні інформацією в цифровому середовищі.

Шифрування даних, беззаперечно, є ефективним засобом забезпечення конфіденційності і безпеки інформації. Однак виникає низка проблем, пов'язаних із застосуванням шифрування в сучасному цифровому середовищі. По-перше, однією з ключових проблем є баланс між безпекою і зручністю використання. Додатковий рівень безпеки, найчастіше, веде до складнішого управління ключами або збільшення часу на розшифрування та шифрування даних, що може негативно позначитися на продуктивності та ефективності користувачів.

Тематика кваліфікаційної роботи включає в себе знання з дисциплін, які були пройдені та відповідає завданням в галузі розробки програмного забезпечення. Зокрема використовуються набуті знання по шифруванню, передачі даних в мережі.

Мета кваліфікаційної роботи – створення програмного засобу для шифрованої передачі даних.

Досягнення поставленої мети передбачає реалізацію таких завдань:

1) Дослідити наявні алгоритми шифрування;
2) Здійснити аналіз наявних рішень для захищеної передачі даних;
3) Розробити додаток з такими можливостями: клієнт та сервер з захищеним з’єднанням, авторизація, використання криптостійких алгоритмів для шифрування даних.

Об’єкт дослідження – технології створення додатків шифрування та безпечної передачі даних.

Предмет дослідження – розробка програмного модуля для захищеної передачі даних.

РОЗДІЛ 1. ЗАХИЩЕНІСТЬ ДАНИХ

1.1. Проблема шифрування даних

1.1.1. Визначення шифрування даних

В епоху постійного розвитку інформаційних технологій, де обмін конфіденційною інформацією відбувається в онлайн режимі, а величезні масиви даних зберігаються в цифровій формі, забезпечення безпеки цих даних стає завданням критичної важливості.

Одним із ключових засобів захисту інформації є шифрування даних, яке визначається як процес перетворення зрозумілого тексту в криптографічно захищений формат, непридатний для розуміння без відповідного ключа.

Шифрування грає важливу роль у забезпеченні конфіденційності та цілісності інформації. Однак, щоб зрозуміти сутність цієї техніки, важливо розглянути основні принципи, за якими вона функціонує.

На перший погляд, існують два основних підходи до шифрування: симетричний та асиметричний. Симетричне шифрування використовує один і той же ключ для як шифрування, так і розшифрування даних.

Це швидший метод, але виникає проблема обміну ключами між відправником і отримувачем. З іншого боку, асиметричне шифрування використовує пару ключів: публічний та приватний. Публічний ключ використовується для шифрування, а приватний — для розшифрування.

Поглиблюючись у розуміння шифрування, важливо розглянути його застосування на різних рівнях. Шифрування на рівні даних означає застосування криптографії безпосередньо до текстового або бінарного представлення даних. Шифрування на рівні комунікацій включає захист інформації під час передачі через мережі, а шифрування на рівні додатків забезпечує захист для конкретних програм чи сервісів.

Проте, навіть з усією важливістю шифрування, виникають проблеми та виклики. Технічні атаки на шифри, соціальна інженерія та обмеження, накладені законодавством, є складністю, яку необхідно подолати для забезпечення ефективної інформаційної безпеки.

У наступних розділах буде розглянуто сучасні тенденції у галузі шифрування, їх переваги та виклики. Також буде досліджено перспективи розвитку шифрування в контексті цифрового світу, який постійно змінюється.

Значення шифрування в сучасному світі є критично важливим, оскільки відображає зростаючий обсяг інформації, яку ми обробляємо, передаємо та зберігаємо в цифровій формі.

Це поняття не обмежується лише сферою інформаційної технології; воно знаходить своє втілення в усіх сферах життя, де зберігаються чи обробляються конфіденційні дані.

Докладніше, чому шифрування стало ключовим елементом в сучасному світі:

Електронна Переписка: Шифрування даних в електронних листах та чатах важливе для того, щоб забезпечити конфіденційність особистої інформації, бізнес-справжньості та іншої конфіденційної інформації.

Фінансові Транзакції: У банківській сфері шифрування використовується для захисту фінансових транзакцій та особистих даних клієнтів під час онлайн-операцій.

Захист від несанкціонованого доступу: Шифрування використовується для захисту інформації від несанкціонованого доступу, такого як хакерські атаки, фішинг та інші форми кіберзлочинності.

Доступ до даних на випадок втрати пристрою: Шифрування допомагає захистити дані на випадок втрати або крадіжки комп'ютера, смартфона чи інших пристроїв. Корпоративна інформаційна безпека: для підприємств шифрування важливе для захисту корпоративної інформації, клієнтських даних та інших конфіденційних ресурсів.

Захист інтелектуальної власності: в галузі наукових досліджень, технологічного розвитку та інновацій, шифрування використовується для захисту інтелектуальної власності та конфіденційних даних.

Захист від законодавства про персональні дані: шифрування може бути важливим елементом для дотримання законодавства про захист персональних даних, зокрема при передачі чи зберіганні чутливої інформації.

Інтернет Речей (*IoT*): У світі Інтернету Речей, де пристрої постійно з'єднані між собою, шифрування є ключовим для захисту комунікації та обміну даними між пристроями.

Захист Від Розповсюдження Шкідливих Програм: Шифрування грає важливу роль у захисті від шкідливих програм та вірусів, оскільки воно ускладнює або робить неможливим злам системи.

Узагальнюючи, шифрування стало фундаментальним елементом сучасної інформаційної безпеки, визначаючи засоби та підходи для захисту конфіденційної інформації в цифровому світі. Забезпечення безпеки даних стає ключовим завданням, а шифрування є невід'ємною частиною стратегій захисту від зовнішніх загроз.

1.1.2. Основні види шифрування

Симетричне шифрування: Симетричне шифрування використовує один і той же ключ для як шифрування, так і розшифрування даних. Основний принцип полягає в тому, що якщо ви маєте ключ для зашифрованого тексту, ви також можете використовувати його для розшифрування. Головні переваги - швидкість та ефективність[2]. Однак проблемою є безпека обміну ключами між відправником і отримувачем, оскільки сам ключ може бути предметом атаки.

Асиметричне шифрування: Асиметричне шифрування використовує пару ключів: публічний і приватний. Публічний ключ використовується для шифрування повідомлення, а приватний - для його розшифрування. Основний принцип полягає в тому, що, навіть якщо публічний ключ відомий всім, приватний ключ лишається секретним. Це робить асиметричне шифрування ефективним для захисту даних та обміну ключами.

Протоколи гібридного шифрування: Гібридне шифрування поєднує симетричне та асиметричне шифрування для забезпечення оптимального рівня безпеки та ефективності. Зазвичай, для обміну ключами використовують асиметричне шифрування, а фактичні дані - симетричне. Це знижує навантаження на систему, адже симетричне шифрування ефективніше, але при цьому забезпечує безпеку обміну ключами.

Хеш-функції та цифрові підписи: Хеш-функції використовуються для створення фіксованого розміру "відбитка" (хешу) для будь-якого об'єкта даних. Це дозволяє перевірити цілісність даних. Цифрові підписи використовують асиметричне шифрування для створення та перевірки підпису, що гарантує автентичність та невідмінність даних.

Сіль (*Salt*) та ітеративне хешування: Для ускладнення атак типу "підбор паролю" важливим є використання солі - випадкової додаткової інформації, яка додається до паролю перед хешуванням. Ітеративне хешування передбачає повторення процесу хешування кілька разів, що робить атаки більш витратними.

Протоколи безпеки транспортного рівня (*SSL/TLS*): *SSL* (*Secure Sockets Layer*) та його наступник *TLS* (*Transport Layer Security*) – це протоколи, які забезпечують безпеку під час обміну даними між клієнтом та сервером через Інтернет.

Ці принципи шифрування взаємодіють і мають на меті створення надійних та ефективних засобів захисту інформації в різних сферах використання.

1.2. Шифрування на рівні даних

Шифрування на рівні даних – це метод застосування криптографії безпосередньо до текстового або бінарного представлення конкретних даних. Цей підхід дозволяє забезпечити безпеку інформації навіть тоді, коли самі дані зберігаються або передаються в незахищеному вигляді.

Алгоритми Шифрування: Симетричні Алгоритми: В даному контексті застосовуються симетричні алгоритми, такі як *AES* (*Advanced Encryption*

Standard) або *DES* (*Data Encryption Standard*), де той самий ключ використовується як для шифрування, так і для розшифрування даних.

Хеш-функції: Хеш-функції, такі як *SHA*-256, можуть використовуватися для створення унікального "відбитка" даних, який може слугувати контрольною сумою.

Бази Даних: Шифрування на рівні даних широко використовується у системах управління базами даних (СУБД) для захисту конфіденційної інформації, такої як особисті дані користувачів чи фінансова інформація [3].

Клієнт-Серверні Додатки: В додатках, де інформація передається між клієнтом та сервером, шифрування на рівні даних гарантує, що дані залишаються захищеними під час транзакцій.

Преференції в Шифруванні: Шифрування Цілого Файлу: Дані можуть бути зашифровані як цілі файли. Це особливо корисно при зберіганні файлів на зовнішніх пристроях чи хмарних сервісах. Шифрування Чутливих Секторів: В окремих випадках шифрування може бути застосоване тільки до чутливих частин файлів чи баз даних, забезпечуючи гранульований рівень захисту.

Солі та Ітеративне Хешування: Додаткова безпека може бути досягнута за допомогою введення солей (додаткової інформації) та ітеративного хешування, особливо при роботі з паролями чи іншими секретними даними.

Шифрування на рівні даних грає важливу роль у сферах, де безпека даних важлива, і дозволяє ефективно захищати чутливу інформацію в будь-яких областях застосування від технологічних платформ до корпоративних баз даних.

1.2.1. Симетричне шифрування

Основні принципи та принцип дії

Використання Одного Ключа: Симетричне шифрування базується на використанні одного спільного секретного ключа для шифрування і розшифрування даних. Цей ключ повинен залишатися конфіденційним.

Принцип Дії: Процес симетричного шифрування розпочинається вибором конкретного алгоритму шифрування, такого як *AES* чи *DES*. Потім,

використовуючи обраний алгоритм та спільний ключ, дані перетворюються у криптографічно безпечний шифрований текст.

Ключ та алгоритм застосовуються до вхідних даних, перетворюючи їх у шифрований текст. Зашифрований текст виглядає не прочитано іншим особам, які не володіють ключем.

За наявності ключа отримувач використовує його разом з алгоритмом для розшифрування отриманих даних. Оригінальні дані відновлюються, і отримувач може їх прочитати.

Передача Ключа: Є однією з найважливіших аспектів симетричного шифрування. Ключ повинен бути переданий від одного комунікуючого партнера до іншого способом, що гарантує його конфіденційність. Це може бути реалізовано за допомогою захищеного каналу, але сам процес обміну ключами залишається вразливим перед атаками.

Режими Роботи: Симетричне шифрування може працювати в різних режимах, таких як *ECB* (*Electronic Codebook*), *CBC* (*Cipher Block Chaining*) та інші, які визначають, як саме блоки даних обробляються і комбінуються для створення шифрованого тексту.

Режим Електронного Кодового Книгування (*ECB*) в симетричному шифруванні. Розділення на Блоки: *ECB* розглядає вхідні дані як послідовність блоків однакового розміру (наприклад, 128 біт). Кожен блок обробляється незалежно од одного. Це означає, що однакові блоки у вхідних даних будуть зашифровані в ідентичні блоки шифрованого тексту. Вхідні дані подаються на вхід алгоритму шифрування блок за блоком. Ключ шифрування може застосовуватися до кожного блоку окремо або може застосовуватися єдиний ключ до всіх блоків, в залежності від конкретної реалізації.

Основною перевагою є простота реалізації: *ECB* легко реалізувати і розуміти. Блоки обробляються незалежно, що дозволяє паралельну обробку. Недоліком є податливість до Атак. Однакові блоки вхідних даних будуть мати ідентичні блоки шифрованого тексту. Це може розкривати патерни та інформацію, зокрема у випадку, коли у вхідних даних є повторювані частини.

ECB використовується у випадках, де не важлива взаємозалежність між блоками, і не виникає проблема однакових блоків у вхідних даних.

Забезпечення безпеки ключа - це важлива відповідальність, інакше може виникнути ризик доступу до шифрованих даних. Перевагою є ефективність, простота реалізації. Недоліком є проблема обміну ключами, загроза компрометації ключа. *ECB* використовують такі алгоритми *DES* (*Data Encryption Standard*), *AES* (*Advanced Encryption Standard*), *IDEA* (*International Data Encryption Algorithm*).

Режим Зціплення Блоків (*Cipher Block Chaining – CBC*) в симетричному шифруванні. Основні Принципи: Взаємозалежні Блоки: *CBC* взаємозалежно обробляє блоки вхідних даних. Кожен блок шифрується з урахуванням результату попереднього зашифрованого блоку. Ініціалізація Вектором Ініціалізації (IV): Перший блок даних комбінується з унікальним IV (ініціалізаційний вектор), який є випадковим ініціальним значенням, необхідним для перших обчислень.

Використання попереднього шифртексту: Кожен блок вхідних даних комбінується з шифрованим попереднім блоком перед шифруванням. Це забезпечує взаємозалежність блоків і додає додатковий рівень безпеки.

При розшифруванні блоки шифрованого тексту дешифруються, і результат комбінується з попереднім зашифрованим блоком або IV для отримання оригінальних даних.

Перевагою є взаємозалежність блоків. *CBC* дозволяє зменшити візуальну однаковість блоків у вихідних даних, яку можна помітити в режимі *ECB*. Помірне Збільшення Обчислювальної Вартості: Додаткова обчислювальна вартість через взаємозалежність блоків підвищує безпеку. Недоліком є відсутність паралельності.

Блоки мають бути оброблятися послідовно через їх взаємозалежність, що робить *CBC* менш ефективним для паралельних обчислень.

Випадки Застосування: *CBC* є розповсюдженим режимом для шифрування повідомлень та файлів, особливо там, де важлива взаємозалежність блоків.

1.2.2. Асиметричне шифрування

Асиметричне шифрування є криптографічним методом, що використовує пару ключів для забезпечення безпеки комунікації. Один ключ, публічний, використовується для шифрування повідомлення, тоді як інший ключ, приватний, використовується для його розшифрування. У порівнянні з симетричним шифруванням, де використовується один і той же ключ для обох операцій, асиметрична схема забезпечує ефективний та безпечний обмін ключами в мережі.

Однією з основних переваг асиметричного шифрування є можливість використання публічного ключа для підтвердження автентичності та відправника, а також для створення цифрових підписів, що гарантує цілісність переданих даних.

Проте, асиметричне шифрування має свої обмеження, такі як обчислювальна складність та обмін ключами великого розміру. Завдяки унікальній можливості використання пари ключів, асиметричне шифрування залишається невід'ємною складовою сучасної криптографії, забезпечуючи безпеку електронних комунікацій та транзакцій [4].

Для використання асиметричного шифрування необхідно два ключі публічний та приватний. Публічний ключ є центральною складовою асиметричного шифрування і грає ключову роль у забезпеченні безпеки комунікації між сторонами. Це числовий параметр, який створюється алгоритмом шифрування та є відкритим для загального використання. Відправник використовує публічний ключ одержувача для шифрування повідомлення перед відправленням його по відкритій мережі.

Цей зашифрований текст може бути розшифрований лише за допомогою відповідного приватного ключа, який виключно у власника публічного ключа. Така взаємодія забезпечує конфіденційність переданих даних та відсутність необхідності обміну конфіденційними ключами перед комунікацією.

Публічні ключі можуть вільно розповсюджуватися та використовуватися будь-ким, оскільки вони служать лише для шифрування даних або створення

цифрових підписів. Проте, безпека системи асиметричного шифрування високо залежить від надійності приватних ключів, які повинні залишатися в секреті у власника.

Такий підхід використовується для створення безпечних та надійних криптосистем для захисту електронної комунікації та забезпечення конфіденційності і цілісності даних.

Приватний ключ є критично важливою складовою асиметричного шифрування та відіграє ключову роль в забезпеченні безпеки системи. Це числовий параметр, який є секретним і використовується тільки власником публічного ключа для розшифрування отриманих шифрованих повідомлень або для створення цифрових підписів.

Існує сувора необхідність збереження приватного ключа в таємниці, оскільки будь-яке його розголошення може призвести до компрометації всієї системи шифрування та ризику неправомірного доступу до конфіденційної інформації.

Приватний ключ використовується для розшифрування даних, зашифрованих за допомогою відповідного публічного ключа. Цей процес гарантує, що тільки особа, яка володіє відповідним приватним ключем, може отримати доступ до оригінальної інформації. Такий розділ ключів забезпечує ефективний механізм аутентифікації та конфіденційності в асиметричних криптосистемах, забезпечуючи високий рівень безпеки в електронній комунікації та зберіганні даних.

Процес асиметричного шифрування складається з двох основних етапів: шифрування та розшифрування.

Процес шифрування в асиметричному шифруванні включає кілька ключових етапів, що гарантують безпеку передачі конфіденційної інформації. Початковим етапом є генерація пари ключів - публічного та приватного ключа. Публічний ключ розповсюджується для загального використання, тоді як приватний ключ залишається виключно у власника. Перед тим як відправник

захоче відправити зашифроване повідомлення отримувачу, він отримує публічний ключ отримувача.

Далі відправник використовує публічний ключ отримувача для шифрування тексту повідомлення. Цей етап гарантує, що тільки власник відповідного приватного ключа може розшифрувати дані. Зашифрований текст відправляється отримувачеві, який використовує свій приватний ключ для розшифрування отриманого повідомлення. Такий метод дозволяє забезпечити конфіденційність даних під час їхньої передачі через відкриті мережі, оскільки тільки власник відповідного приватного ключа може розкодувати вхідні дані. Крім того, асиметричне шифрування використовується для створення цифрових підписів, які підтверджують автентичність даних та неперевершеність повідомлення, що передається. Такий процес забезпечує високий рівень безпеки та довіри в електронних комунікаціях.

Процес розшифрування в асиметричному шифруванні є важливою частиною криптографічної схеми, яка забезпечує конфіденційність переданих даних. Відповідно до цього методу, для розшифрування зашифрованого тексту використовується приватний ключ, який є власністю отримувача.

Отримавши зашифроване повідомлення, отримувач використовує свій приватний ключ для розшифрування даних. Приватний ключ є унікальним і секретним для отримувача, тому лише він може використовувати його для відновлення оригінальної інформації. Цей етап розшифрування гарантує конфіденційність та безпеку обміну даними, оскільки лише правильний отримувач може отримати доступ до вихідної інформації.

Важливою аспектом асиметричного шифрування є також використання публічного ключа для створення цифрових підписів. Це дозволяє власнику приватного ключа підтверджувати автентичність та неперевершеність відправленого повідомлення, що додає ще один рівень безпеки та довіри в електронних комунікаціях. Таким чином, процес розшифрування в асиметричному шифруванні сприяє створенню безпечного та надійного механізму обміну конфіденційною інформацією в цифровому середовищі.

Захист приватного ключа є критично важливим елементом забезпечення безпеки асиметричного шифрування. Приватний ключ є основним інструментом для розшифрування даних, створення цифрових підписів та виконання інших криптографічних операцій, тому його конфіденційність має вирішальне значення. Один з ключових методів захисту приватного ключа полягає в його фізичному зберіганні в безпечному місці, наприклад, на токені або у захищеному апаратному модулі.

Додаткові заходи безпеки включають в себе використання парольної аутентифікації для доступу до приватного ключа. Це може бути пароль або PIN-код, які єдиний власник приватного ключа повинен знати. Багато факторів аутентифікації, таких як використання біометричних даних чи додаткових апаратних ключів, також можуть посилити захист приватного ключа. Крім того, важливо вживати заходів для запобігання витоку ключа, таких як застосування відповідних антивірусних заходів та регулярне оновлення безпеки системи. Такий комплексний підхід гарантує, що приватний ключ залишається в безпеці та використовується лише законним власником.

1.3. Цифровий підпис

Цифровий підпис є важливим елементом криптографічного забезпечення та використовується для підтвердження автентичності, цілісності та неперевершеності документа чи повідомлення в електронному середовищі. Цей механізм забезпечує можливість перевірки того, що вміст, підписаний особою чи організацією, не був змінений після його підписання. Для створення цифрового підпису використовуються асиметричні криптографічні ключі: автор використовує свій приватний ключ для створення підпису, і отримувач перевіряє його за допомогою публічного ключа.

Процес створення цифрового підпису включає хешування (створення короткого вирізу, або "хешу", вихідного повідомлення) та шифрування отриманого хешу приватним ключем автора. Такий підпис дозволяє іншим

сторонам перевірити, чи відбулося будь-яке змінення в початковому повідомленні, і чи відповідає воно підпису власника. Це робить цифровий підпис ефективним засобом підтвердження автентичності документів, електронних листів, програмних пакетів та інших цифрових ресурсів.

Цифрові підписи широко використовуються в електронній комунікації, електронній торгівлі та інших галузях, де важлива безпека і впевненість у тому, що дані не були порушені під час передачі чи зберігання. Вони допомагають забезпечити довіру до електронних транзакцій та документообігу в цифровому світі, зменшуючи ризики маніпуляції та незаконного доступу.

Процес створення цифрового підпису розпочинається з обчислення хеш-функції для вихідного повідомлення. Хеш-функція генерує унікальний хеш або "відбиток" для заданого вхідного тексту. Згодом, цей хеш шифрується за допомогою приватного ключа власника, створюючи електронний підпис. Цей процес забезпечує цифровий підпис, який унікально пов'язаний з конкретним вихідним текстом та приватним ключем власника.

Перевірка цифрового підпису включає два основних етапи. По-перше, відправник отримує підписане повідомлення та його цифровий підпис. Він обчислює хеш вихідного повідомлення, використовуючи ту саму хеш-функцію, яку використовував власник приватного ключа. Далі, відправник використовує публічний ключ власника, щоб розшифрувати отриманий цифровий підпис і отримати хеш, який він порівнює з тим, який він сам обчислив.

Якщо обчислені хеші співпадають, це означає, що підпис валідний, і відправник може бути впевнений в тому, що отримане повідомлення не було змінено і є автентичним. Якщо ж хоча б один біт тексту був змінений або повідомлення було змінено, це призведе до відмови в перевірці підпису. Такий механізм гарантує, що тільки особа з правильним приватним ключем могла б підписати дане повідомлення, а хеш служить відбитком для перевірки його неперевершеності.

Підписання та Підтвердження Автентичності: Асиметричне шифрування також може використовуватися для створення цифрових підписів. Відправник

може підписати дані своїм приватним ключем, і отримувач може підтвердити автентичність даних, використовуючи публічний ключ відправника.

1.4. Алгоритми шифрування

DES, або стандарт шифрування даних, був розроблений в 1970-х роках як симетричний алгоритм шифрування для захисту конфіденційності електронних даних. Розроблений урядовим агентством США, *DES* вперше став стандартом в 1977 році і широко використовувався протягом багатьох років у різних галузях, включаючи фінанси та електронні транзакції.

DES використовує 64-бітний блок тексту та 56-бітний ключ для шифрування. Згідно з алгоритмом, вихідне повідомлення розбивається на блоки по 64 біта, які обробляються 16 раундами перестановок та замін. Ключ також проходить через складний процес перестановок та замін для генерації ключів для кожного раунду. Завдяки цьому, *DES* вважався досить надійним в той час.

Згодом *DES* було надано критиці, основною причиною якої стало зростання потужності обчислювачів та методів криптоаналізу. Основною слабкістю став короткий довжина ключа в 56 біт, що ставало простішим для атаки методом перебору всіх можливих ключів. Це призвело до необхідності розробки більш безпечних стандартів шифрування, таких як *AES* (*Advanced Encryption Standard*).

Незважаючи на те, що *DES* втратив актуальність як основний криптографічний стандарт, його спадщина залишається важливою. Знання про *DES* допомагає розуміти еволюцію шифрування та вдосконалення нових стандартів. Сучасні шифри, такі як *AES*, взяли на увагу навчання з недоліків DES, забезпечуючи більшу довжину ключа та вищий рівень безпеки.

Triple DES, або *3DES*, є розширенням оригінального *DES* і був розроблений як спроба підвищення рівня безпеки шифрування. *DES* став зазнавати критики через зростання обчислювальної потужності комп'ютерів, яка ставила під загрозу його надійність. *3DES* вважався перехідним рішенням, яке використовувало *DES* у потрійному режимі.

Основна ідея *3DES* полягає в тому, щоб застосовувати *DES* тричі до кожного блоку даних. Це може бути реалізовано у двох режимах: *EDE* (*Encrypt*, *Decrypt*, *Encrypt*) та *EEE* (*Encrypt*, *Encrypt*, *Encrypt*). У режимі *EDE*, де ключі можуть бути різними, вхідні дані спочатку шифруються, потім розшифровуються тим самим ключем та знову шифруються іншим ключем.

3DES використовує ключі завдовжки 168 біт, що робить його більш стійким до атак на перебір ключів порівняно з оригінальним *DES*. Втім, через трикратне застосування *DES*, *3DES* виявився повільнішим порівняно з більш сучасними шифрами, такими як *AES*. З течією часу, *3DES* втратив свою популярність, оскільки новіший стандарт *AES* надав більшу безпеку та ефективність.

Незважаючи на те, що *3DES* стає менш вживаним через свою повільність, він все ще застосовується в ряді систем, де важлива сумісність із застарілими пристроями або програмним забезпеченням. Замість цього його важливість зменшується, і його місце у багатьох випадках займає більш продуктивний та безпечний *AES*.

Advanced Encryption Standard (*AES*) є сучасним стандартом симетричного шифрування, який замінив застарілий стандарт *DES*. Розроблений у 2001 році, *AES* став ключовим елементом для забезпечення безпеки і конфіденційності електронної інформації. Його вибір став результатом тендера, який залучав кращих криптографічних фахівців та компаній з усього світу.

AES використовує блочний шифр, що означає, що він опрацьовує блоки даних фіксованого розміру. Основною його особливістю є використання ключів різної довжини: 128, 192 та 256 біт. Кожен з цих ключів визначає рівень безпеки шифру, причому 128-бітний ключ зазвичай вважається достатньо стійким для багатьох застосувань.

AES базується на заміні та перестановці байтів у блоках даних, що повторюється для кожного раунду шифрування. В процесі шифрування використовуються ключі, які змінюють стан блока даних у кожному раунді. Важливою перевагою *AES* є його висока ефективність і стійкість до

різноманітних атак, що робить його основним інструментом для шифрування даних у багатьох сферах.

AES широко використовується в електронних комунікаціях, безпеці мереж, електронній комерції та інших областях. Він також входить у склад безпеки *Wi-Fi*, *TLS/SSL* протоколів та багатьох інших стандартів. З часом, його популярність збільшується, оскільки забезпечення великої стійкості і ефективності робить його привабливим вибором для захисту конфіденційної інформації.

RSA – це асиметричний алгоритм шифрування та цифрового підпису, який був вперше представлений у 1977 році Рональдом Рівестом, Аді Шамиром і Леонардом Адльманом. Названий на честь їхніх прізвищ, він став першим алгоритмом, який забезпечує можливість використання різних ключів для шифрування та розшифрування.

RSA базується на математичних властивостях складних простих чисел. В основі алгоритму лежить завдання факторизації: процес розкладання великого числа на множники простих чисел. Ключі *RSA* складаються з публічного та приватного ключів. Публічний ключ використовується для шифрування повідомлення, тоді як приватний ключ використовується для розшифрування.

Принцип роботи *RSA* включає генерацію двох великих простих чисел і вивчення їхнього добутку. Потім обчислюється експонента для обох ключів. Публічний ключ включає пару чисел (*n, e*), де *n* – добуток простих чисел, а *e* – експонента для шифрування. Приватний ключ включає пару (*n, d*), де *n* – той самий добуток, а *d* – експонента для розшифрування. *RSA* широко використовується для шифрування та підпису даних в електронних комунікаціях та безпеці інформації в цифровому світі.

RSA використовується у багатьох аспектах кібербезпеки, зокрема для захисту електронної пошти, електронних фінансових транзакцій та забезпечення безпеки передачі даних через Інтернет. Однак з розвитком квантових комп'ютерів, які можуть здійснювати швидке факторизування великих чисел, алгоритми, такі як *RSA*, можуть потрапити під загрозу. Це призводить до пошуку нових криптографічних методів, які будуть стійкими до квантових атак.

1.5. Апаратне шифрування

Апаратне шифрування даних – це метод шифрування, де процес шифрування та розшифрування виконується безпосередньо апаратним обладнанням, таким як процесор чи спеціальний криптографічний апарат. Відмінність цього методу від програмного полягає в тому, що весь шифрувальний процес здійснюється на апаратному рівні, що може забезпечити вищий рівень ефективності та безпеки.

У системах апаратного шифрування, часто використовуються спеціалізовані криптографічні чіпи або модулі, які вбудовані в обладнання, таке як жорсткі диски, флеш-накопичувачі, або сенсорні пристрої. Ці чіпи можуть використовувати апаратне реалізовані алгоритми шифрування, такі як *AES*, для захисту даних. Ключі шифрування та інші параметри також можуть зберігатися в пристрої, ускладнюючи можливість несанкціонованого доступу до них.

Апаратне шифрування базується на використанні вбудованого апаратного засобу для виконання криптографічних операцій. Це може включати в себе генерацію ключів, шифрування та розшифрування даних. Зазвичай такі апаратні модулі працюють над ключами і операціями шифрування без виведення цих ключів на зовнішні програмні рівні, що підвищує безпеку[5].

Апаратне шифрування використовує вбудовані апаратні ключі для забезпечення безпеки. Ці ключі можуть бути згенеровані самим апаратом або імпортовані. Використання апаратних ключів допомагає уникнути можливих загроз, пов'язаних з витіком ключів через програмне забезпечення.

Однією з ключових переваг апаратного шифрування є висока швидкість обробки даних, оскільки весь процес відбувається на апаратному рівні, не потребуючи значних ресурсів від центрального процесора.

Крім того, апаратне шифрування може забезпечити більшу стійкість до програмних атак, оскільки ключі та алгоритми знаходяться в захищеному апаратному середовищі.

Апаратне шифрування широко використовується в сферах, де важлива безпека даних, таких як в корпоративних інфраструктурах, урядових системах, а також в зберіганні і обробці конфіденційної інформації в банківській сфері. Також цей метод шифрування може використовуватися в мобільних пристроях та інших портативних пристроях, що дозволяє забезпечити безпеку даних при їх втраті чи крадіжці.

1.6 Аналіз засобів для захищеної передачі даних

На даний момент існує безліч програм для захищеної передачі даних в Інтернеті. Вибір конкретної програми може залежати від вашого конкретного використання та потреб.

1.6.1. *Virtual Private Network*

Virtual Private Network *(VPN)* – це технологія, яка створює захищене тунельне з'єднання між вашим пристроєм та Інтернетом. Головна мета використання *VPN* - це захист приватності та безпеки в Інтернеті, забезпечуючи шифрування трафіку та анонімність.

Типи *VPN*: *Remote Access VPN*: Забезпечує безпечний доступ до корпоративної мережі для віддалених користувачів через інтернет.

Site-to-Site VPN: З'єднує дві або більше фізичні мережі (наприклад, філіали підприємства) для безпечного обміну даними.

Extranet VPN: Дозволяє обмін даними між різними організаціями або підприємствами, забезпечуючи захист інформації.

VPN створює захищений тунель між вашим пристроєм та сервером *VPN*. Це означає, що ваш Інтернет-трафік проходить через цей тунель, що робить його захищеним від несанкціонованого доступу. Всі дані, які ви відправляєте або отримуєте через *VPN*, шифруються. Це означає, що інформація залишається конфіденційною та захищеною від перехоплення[6].

VPN може надавати нову *IP*-адресу вашому пристрою, що приховує вашу справжню мережеву ідентифікацію. Це дозволяє вам залишатися анонімним в

Інтернеті та обходити обмеження доступу, які можуть бути накладені на ваш реальний IP.

Протоколи *VPN*:

OpenVPN: Відкритий протокол, який підтримує широкий спектр конфігурацій та використовує відкриті технології.

IPsec (*Internet Protocol Security*): Сукупність протоколів для захисту *IP*-трафіку, яка може використовуватися для створення *VPN*.

L2TP/*IPsec* (*Layer* 2 *Tunneling Protocol*/*IP Security*): Комбінує *L2TP* для створення тунелю та *IPsec* для шифрування даних.

При використанні невідомих або ненадійних громадських мереж, наприклад, у кафе або аеропорту, *VPN* додає додатковий рівень безпеки, захищаючи трафік від можливих атак і перехоплення.

VPN можна використовувати для зміни віртуальної локації, що дозволяє обходити географічні обмеження та отримувати доступ до ресурсів, які можуть бути недоступні у вашому регіоні.

1.6.2. Шифровані месенджери

Шифровані месенджери – це додатки для обміну текстовими повідомленнями, фотографіями, відео та іншими файлами, які використовують криптографічні техніки для захисту конфіденційності та безпеки комунікації. Основна ідея полягає в тому, щоб забезпечити точку-до-точки шифрування, де тільки відправник та отримувач можуть прочитати зміст повідомлення.

End-to-end шифрування – це ключова функція, яка означає, що дані шифруються на пристрої відправника та розшифровуються тільки на пристрої отримувача. Навіть посередник (сервер чи розробник додатка) не може переглядати або розшифрувати дані.

Деякі шифровані месенджери також застосовують шифрування до медіафайлів, таких як фотографії та відео, щоб забезпечити конфіденційність зображень, які ви відправляєте.

Деякі месенджери надають можливість встановлення терміну життя для повідомлень, після якого вони автоматично знищуються. Це сприяє додатковій приватності і унеможливлює тривале збереження повідомлень на пристроях.

Деякі месенджери приділяють увагу не лише змісту повідомлень, але й метаданим, які включають в себе інформацію про те, хто, коли і звідки відправляє чи отримує повідомлення.

Деякі месенджери не зберігають дані про повідомлення на серверах, щоб уникнути можливості несанкціонованого доступу до цих даних.

Шифровані месенджери використовують різноманітні техніки для захисту від різних видів атак, включаючи перехоплення на мережевому рівні та атаки типу "людина посередник".

Деякі месенджери використовують асиметричне шифрування, де кожен користувач має публічний та приватний ключ. Публічні ключі можуть бути обмінювані для безпечної верифікації осіб, яким ви відправляєте повідомлення.

До популярних шифрованих месенджерів входять *Signal*, *WhatsApp* (з включеним *end-to-end* шифруванням), *Telegram* (по бажанню включити "секретні чати"), і багато інших.

1.6.3. *Tresorit*

Tresorit - це хмарна служба сховища даних і інструмент для спільної роботи, яка приділяє особливу увагу конфіденційності та безпеці даних. Ось детальна інформація про *Tresorit*:

Tresorit визначається своєю високою забезпеченістю та конфіденційністю. Всі дані, які ви завантажуєте до служби, шифруються на вашому пристрої перед тим, як вони будуть передані на сервери *Tresorit*. Використовується асиметричне шифрування, і тільки ви маєте ключ доступу до своїх файлів.

Tresorit використовує архітектуру "*zero-knowledge*", що означає, що навіть сама компанія не має доступу до вашого ключа шифрування або ваших конфіденційних даних. Таким чином, навіть якщо сервери *Tresorit* стануть жертвою атаки, ваші дані залишаються безпечними.

Tresorit дозволяє створювати "трезори" (*tresors*), що є цифровими контейнерами для вашої інформації. Ви можете додавати файли до цих трезорів та ділитися ними з іншими користувачами. Зручна функція синхронізації дозволяє мати доступ до своїх даних на різних пристроях. Tresorit працює на різних операційних системах, включаючи *Windows, macOS, Android* та *iOS*. Це робить його зручним і доступним для використання на багатьох пристроях.

Додаток *Tresorit* дозволяє налаштовувати рівні доступу до своїх трезорів, надаючи різним користувачам різні права доступу до файлів. Tresorit має штаб-квартиру в Швейцарії, що відомо своєю строгістю в питаннях конфіденційності та захисту даних.

1.6.4 *SFTP*

SFTP (*Secure File Transfer Protocol*) – це протокол передачі файлів, який забезпечує захищений і шифрований канал для передачі даних через мережу. Ось детальна інформація про *SFTP*:

SFTP використовує шифрування для захисту конфіденційності даних під час їх передачі через мережу.

За замовчуванням *SFTP* використовує *SSH* (*Secure Shell*) для створення захищеного тунелю між клієнтом і сервером, що забезпечує конфіденційність та цілісність даних. Клієнт і сервер обмінюються ключами *SSH* під час аутентифікації, що дозволяє перевірити взаємну достовірність сторін.

Для аутентифікації можуть використовуватися різні методи, такі як пароль, ключі SSH або інші методи, які підтримує обрана конфігурація.

SFTP надає можливість виконувати такі операції, як завантаження (завантаження) та вивантаження (вивантаження) файлів, створення та видалення каталогів, отримання списку файлів на сервері і т. д. Протокол також дозволяє виконувати деякі адміністративні функції, такі як перезапуск служби *SFTP*.

SFTP використовує *TCP*-порт 22 (той же, який використовується для *SSH*) для встановлення з'єднань. Хоча назви схожі, *SFTP* відмінний від *FTPS* (*FTP over SSL*/*TLS*), який використовує альтернативні порти та базується на *SSL*/*TLS* для шифрування.

SFTP є стандартом для безпечної передачі файлів і підтримується багатьма платформами, включаючи *Unix*, *Linux*, *Windows* та інші.Існують різноманітні клієнти та сервери, які підтримують протокол *SFTP*. Деякі операційні системи вже мають вбудовану підтримку *SFTP*.*SFTP* є надійним і безпечним вибором для передачі файлів у відкритих мережах, зокрема через Інтернет. Однак важливо враховувати, що в певних випадках може бути доцільно додатково застосовувати заходи безпеки, такі як використання *VPN*, для захисту передачі даних через *SFTP*.

1.6.5. *FTPS*

FTPS (*File Transfer Protocol Secure*) - це розширення протоколу *FTP*, яке використовує *TLS* (*Transport Layer Security*) або *SSL* (*Secure Sockets Layer*) для шифрування даних та забезпечення безпечної передачі файлів через мережу. Відмінності від *FTP* полягають в застосуванні шифрування і використанні альтернативного порту. Ось детальна інформація про *FTPS*:

FTPS використовує *TLS* або *SSL* для шифрування з'єднання між клієнтом і сервером, що забезпечує конфіденційність та цілісність даних.

TLS і *SSL* забезпечують захист від перехоплення і прослуховування данних, що робить *FTPS* відмінним вибором для безпечної передачі файлів через відкриті мережі.

FTPS може працювати в двох режимах: як *FTPS Implicit* і як *FTPS Explicit*. В режимі *Implicit*, шифрування встановлюється відразу під час підключення до серверу. У режимі *Explicit*, шифрування установлюється тільки після видачі відповідного командного коду.

FTPS використовує *TCP*-порт 21 для комунікації, що є стандартним портом для *FTP*. Також, для забезпечення шифрування, можуть використовуватися альтернативні порти, наприклад, 990 для контрольного каналу та 989 для передачі даних. *FTPS* може потребувати налаштувань брандмауера для відкриття відповідних портів для передачі даних.

Аутентифікація в *FTPS* може проводитися за допомогою пароля, імені користувача (*username*), а також використання сертифікатів для більш високого рівня безпеки.

Для встановлення шифрованого каналу використовуються цифрові сертифікати, які можуть бути самопідписаними або отриманими від відповідного центру сертифікації (*CA*).

Існують різні клієнти та сервери, які підтримують протокол *FTPS*. Багато операційних систем включають підтримку *FTPS* або можливість встановлення відповідного програмного забезпечення.

FTPS є популярним і ефективним засобом для захищеної передачі файлів, і він забезпечує великий рівень безпеки завдяки використанню *TLS* або *SSL* протоколів.

1.7. Передача даних через *TCP/IP*

Модель *TCP/IP* (*Transmission Control Protocol/Internet Protocol*) - це концептуальна модель комунікацій, що визначає, як різні протоколи повинні взаємодіяти в комп'ютерних мережах, зокрема в Інтернеті. Модель *TCP/IP* (рис. 1.1) складається з чотирьох рівнів, кожен з яких відповідає за конкретний аспект мережевої комунікації [7].

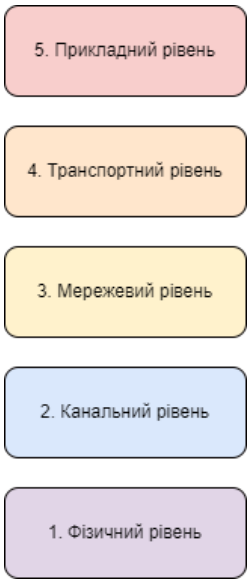

Рис. 1.1. Мережева *TCP/IP* модель

Фізичний рівень (*Physical Layer*) є найнижчим рівнем моделі *TCP*/*IP* і визначає фізичні аспекти передачі даних через мережу. Його основні завдання включають в себе передачу бітів через фізичні засоби зв'язку та забезпечення механічної, електричної, функціональної та логічної сумісності між пристроями в мережі. Основні характеристики фізичного рівня:

Медіа і Сигнали: Фізичний рівень визначає, як біти представлені на фізичному носії (кабелі, оптиці, бездротовому з'єднанні тощо). Медіа можуть бути коаксіальними кабелями, витими парами, оптичними волокнами, бездротовими.

Способи Кодування: Визначає, які електричні чи оптичні сигнали використовуються для представлення бітів. Методи кодування можуть включати *AM* (*Amplitude Modulation*), *FM* (*Frequency Modulation*), *PCM* (*Pulse Code Modulation*).

Фізичний Застосунок: Визначає механічні та електричні характеристики конекторів, кабелів, роз'ємів тощо. Регулює стандарти фізичного з’єднання між пристроями, такі як *USB*, *Ethernet*, *HDMI* тощо.

Передавачі та Приймачі: Визначає, як фізичний сигнал генерується передавачем та інтерпретується приймачем. Включає характеристики передавачів (передавальна потужність, швидкість передачі тощо) та приймачів (чутливість, діапазон частот тощо).

Топологія та Розподілення Спектра: Визначає, як пристрої фізично підключені один до одного, тобто топологію мережі (зірка, лінія, кільце тощо). Керує використанням частотних діапазонів для уникнення конфліктів та перешкод.

Модуляція та Мультиплексування: Модуляція визначає, як інформація передається через несучий сигнал. Мультиплексування визначає, як декілька сигналів об'єднуються для передачі по одному каналу.

Фізична Топологія: Визначає фізичну структуру мережі та розташування пристроїв. Мережі можуть бути локальними (*LAN*), метрополітенськими (*MAN*)

чи глобальними (*WAN*). Фізичний рівень виступає як інтерфейс між логічними бітами, які обробляються вищими рівнями.

Канальний рівень (*Data Link Layer*) є другим рівнем моделі *TCP*/*IP* і відповідає за передачу фреймів (кадрів) даних між двома призначеними пристроями в рамках одного локального сегмента мережі. Основні завдання канального рівня включають управління доступом до мережі, виявлення та корекцію помилок, адресацію фізичних пристроїв і керування потоком даних.

Основні аспекти канального рівня: Формування та Розбірка Кадрів (*Framing*):

Канальний рівень визначає структуру фреймів, яка включає заголовок та закінчення для визначення початку та кінця кожного кадру даних. Фрейми розбираються на приймачі для отримання оригінального повідомлення.

Адресація *MAC* (*Media Access Control*): Кожен мережевий адаптер (*Network Interface Card - NIC*) має унікальну фізичну адресу, відому як *MAC*-адреса. *MAC*-адреса використовується для визначення, кому адресовані дані в локальній мережі.

Керування Доступом до Мережі: Канальний рівень регулює доступ пристроїв до мережі, використовуючи принципи контролю доступу, такі як *CSMA*/*CD* (*Carrier Sense Multiple Access with Collision Detection*) для *Ethernet*. *CSMA*/*CD* дозволяє пристрою визначити, чи інші пристрої в даний момент передають дані.

Виявлення та Корекція Помилок: Канальний рівень включає методи виявлення та корекції помилок у фреймах даних. Це важливо для забезпечення цілісності передаваних даних.

Керування Потоком Даних: Управління потоком даних регулює швидкість передачі даних між пристроями, щоб уникнути перевантаження отримуючого пристрою.

Розділення Підмереж (*Subnetting*): В деяких мережах канальний рівень може розділяти мережу на підмережі для кращого управління трафіком та адресацією.

Логічна Топологія: Визначає логічну організацію мережі, незалежну від фізичної топології. Включає поняття локальних сегментів та мостів (*bridges*).

Канальний рівень гарантує надійну передачу фреймів в межах локальної мережі, а також виконує функції, які необхідні для подальшої передачі на рівні мережі (*Network Layer*). Він є важливим для забезпечення стабільної та ефективної роботи мережі на фізичному рівні.

Мережевий рівень (*Network Layer*) є третім рівнем моделі *TCP*/*IP* і відповідає за передачу даних між пристроями в мережі. Основні завдання мережевого рівня включають роутінг (маршрутизацію), визначення оптимального шляху для передачі даних від джерела до призначення, а також фрагментацію та збірку пакетів для передачі через фізичні мережеві засоби. Основні характеристики мережевого рівня: IP-адреса:

Кожен пристрій в мережі має унікальну *IP*-адресу, яка ідентифікує його в Інтернеті чи локальній мережі. *IP*-адреса може бути використана для визначення пристрою в мережі та для маршрутизації даних.

Пакети: Дані, які передаються між пристроями, розбиваються на пакети на мережевому рівні. Кожен пакет містить заголовок та дані. Заголовок містить необхідні метадані, такі як *IP*-адреса джерела та призначення.

Маршрутизація: Мережевий рівень визначає оптимальний шлях для передачі пакетів від відправника до призначення. Маршрутизація вирішує, який маршрут повинен бути використаний для досягнення кінцевого пункту.

Протоколи маршрутизації: Декілька протоколів маршрутизації, таких як *RIP* (*Routing Information Protocol*), *OSPF* (*Open Shortest Path First*), *BGP* (*Border Gateway Protocol*), використовуються для визначення оптимальних маршрутів.

Фрагментація та Збірка Пакетів: Мережевий рівень може розділити великі пакети на менші фрагменти для передачі через мережу з обмеженим розміром пакетів (наприклад, *Ethernet*). На призначенні фрагменти об'єднуються для відновлення оригінального пакета. Логічна Адресація: IP-адреси надають логічну адресацію пристроям у мережі.

Дозволяє ідентифікувати пристрої без залежності від їх фізичного розташування чи фізичних характеристик.

Мережеві Протоколи: Основний протокол мережевого рівня - *Internet Protocol* (*IP*). Інші протоколи, такі як *ICMP* (*Internet Control* Message *Protocol*), *ARP* (*Address Resolution Protocol*), *IGMP* (*Internet Group Management Protocol*), також функціонують на цьому рівні.

Мережевий рівень визначає ключові аспекти мережевого сполучення та забезпечує ефективну передачу даних через мережу. Це дозволяє пристроям ефективно співпрацювати в Інтернеті та інших мережах.

Четвертий рівень моделі *TCP/IP* - це транспортний рівень (*Transport Layer*). Транспортний рівень відповідає за забезпечення ефективної та надійної передачі даних між пристроями, які можуть бути розташовані в різних частинах мережі. Основні завдання транспортного рівня включають сегментацію та збірку повідомлень, контроль потоку даних та забезпечення надійності доставки.

Основні характеристики транспортного рівня: Сегментація та Збірка Даних: Транспортний рівень розбиває дані на менші частини, відомі як сегменти, перед їхньою передачею по мережі. Збірка даних відбувається на призначенні, де сегменти об'єднуються для відновлення оригінального повідомлення.

Протоколи Транспортного Рівня: Два основних протоколи транспортного рівня в моделі *TCP/IP* - *Transmission Control Protocol* (*TCP*) та *User Datagram Protocol* (*UDP*). *TCP* надає надійну та орієнтовану на з'єднання передачу даних. Він забезпечує контроль порядку, перевірку доставки та повторну передачу у випадку втрати пакетів. *UDP* є менш надійним, але більш швидким протоколом, який не вимагає встановлення з'єднання та не надає гарантій доставки чи порядку.

Контроль Потоку: Транспортний рівень може використовувати різні методи контролю потоку для управління швидкістю передачі даних між відправником та призначенням. Контроль потоку забезпечує уникнення переповнення буфера та перевантаження призначеного пристрою.

Транспортний рівень грає ключову роль у забезпеченні надійності та ефективності передачі даних в мережі, а також визначає параметри для взаємодії між різними пристроями у мережі.

П'ятий рівень моделі *TCP*/*IP* - це рівень застосунків (*Application Layer*). Рівень застосунків визначає, які служби та протоколи використовуються для взаємодії між програмами та взаємодії з користувачем. Його основні завдання включають передачу даних, управління сесією та представлення інформації.

Основні характеристики рівня застосунків: Протоколи та Служби: Рівень застосунків включає різноманітні протоколи та служби для забезпечення конкретних функцій на рівні програм.

Наприклад, протокол *HTTP* (*Hypertext Transfer Protocol*) використовується для передачі веб-сторінок, *SMTP* (*Simple Mail Transfer Protocol*) - для електронної пошти, *FTP* (*File Transfer Protocol*) – для передачі файлів тощо.

Застосункові Програми: Рівень застосунків включає застосункові програми, які взаємодіють із користувачем та використовують мережеві служби. Прикладами є веб-браузери, електронні поштові клієнти, файло-обмінні програми та інші.

Представлення Даних: Рівень застосунків визначає структуру та представлення даних, що передаються між програмами. Забезпечує конвертацію даних у формат, зрозумілий для програм на обох сторонах комунікації.

Управління Сесією: Рівень застосунків може здійснювати управління сесією, яке включає у себе встановлення, управління та завершення сесій між програмами. Забезпечує можливість взаємодії та обміну даними між програмами.

Ідентифікація та Аутентифікація: Рівень застосунків включає протоколи та механізми для ідентифікації та аутентифікації користувачів та пристроїв у мережі.

Мова Запитань та Відповідей: Комунікація між програмами на рівні застосунків може відбуватися за принципом "запит-відповідь". Програми можуть обмінюватися запитаннями та відповідями для здійснення певної функціональності.

Інтерфейси Користувача: Програми на рівні застосунків надають інтерфейси користувача для взаємодії з користувачем.

UDP (*User Datagram Protocol*) не використовує фрейми, так як це рівень транспортного протоколу, а фрейми визначаються на рівні канального (*Data Link Layer*). Однак я можу надати інформацію про *UDP*-пакети та їх структуру.

UDP використовується для надання послуг, що не вимагають надійності або відновлення даних, таких як відправка аудіо або відео потоку. Його пакети (не фрейми) мають просту структуру:

Порт відправника (*Source Port*): 16 біт. Визначає номер порту відправника.

Порт призначення (*Destination Port*): 16 біт. Визначає номер порту призначення.

Довжина (*Length*): 16 біт. Вказує загальну довжину *UDP*-пакету в байтах (заголовок та дані).

Контрольна сума (*Checksum*):16 біт. Використовується для виявлення помилок в *UDP*-пакеті.

Дані (*Data*): Відсутній заголовок даних, лише корисне навантаження.

UDP-пакети мають мінімальну структуру, і вони передають дані без будь-якого механізму контролю порядку, повторного надсилання або встановлення з'єднання, що робить їх швидкими, але менш надійними в порівнянні з *TCP*.

UDP використовується там, де швидкість та ефективність мають більший пріоритет, ніж надійність передачі даних.

TCP (*Transmission Control Protocol*) також не використовує фрейми, але має заголовок пакету, який включає рядок полів для керування транспортною роботою. Нижче подано детальний опис заголовка *TCP*-пакета:

Порт відправника (*Source Port*): 16 біт. Вказує номер порту відправника.

Порт призначення (*Destination Port*): 16 біт. Вказує номер порту призначення.

Номер послідовності (*Sequence Number*): 32 біти. Це номер першого байта в цьому сегменті даних.

Номер підтвердження (*Acknowledgment Number*): 32 біти. Вказує на наступний очікуваний байт.

Довжина заголовку (*Header Length*): 4 біти. Вказує довжину заголовка в 32-бітних словах.

Прапори (*Flags*):9 біт. Містить різні флаги, такі як *URG* (*urgent pointer*), *ACK* (*acknowledgment*), *PSH* (*push*), *RST* (*reset*), *SYN* (*synchronize*), *FIN* (*finish*).

Вікно (*Window*): 16 біт. Вказує розмір вікна, який використовується для керування потоком.

Контрольна сума (*Checksum*): 16 біт. Використовується для виявлення помилок в TCP-пакеті.

Опції (*Options*): Розмір залежить від поля "Довжина заголовку" та може містити додаткові параметри, такі як максимальний розмір сегменту (*MSS*), вікно масштабування та інші.

Дані (*Data*): Тут розташовуються корисні дані, передані від вищих рівнів.

TCP використовується для забезпечення надійної передачі даних, управління потоком, встановлення та розриву з'єднань. Його заголовок дозволяє встановлювати контроль над передачею даних та забезпечує відновлення у випадку втрати пакетів.

1.8. Висновки за розділом

У світі, насиченому цифровими технологіями, питання забезпечення безпеки та конфіденційності інформації стає надзвичайно актуальним. Шифрування даних виступає ключовим елементом у вирішенні цих завдань, забезпечуючи захист від несанкціонованого доступу та збереження конфіденційності.

Розглядаючи основні принципи шифрування, можемо визначити два важливі підходи: симетричне та асиметричне. Симетричні алгоритми, такі як *AES*, забезпечують ефективне шифрування та розшифрування, використовуючи один ключ. З іншого боку, асиметричні алгоритми, зокрема *RSA*, вирішують

проблему обміну ключами та підпису даних, використовуючи публічний та приватний ключі.

Технології шифрування не лише грають ключову роль в захисті інформації, але й є невід'ємною частиною нашого цифрового повсякдення. Вони використовуються у банківській сфері, медичних закладах, корпоративних структурах, а також в особистих пристроях для забезпечення конфіденційності та недоступності даних для неповноважених осіб.

Однак із зростанням обчислювальної потужності та появою нових технологій, необхідно постійно вдосконалювати методи шифрування для забезпечення високого рівня безпеки. Також важливо розглядати етичні та правові аспекти використання шифрування, збалансовуючи безпеку з правами та свободами користувачів.

Для передачі даних в мережі краще підходить *TCP* протокол. *TCP* (*Transmission Control Protocol*) визначається як надійний і забезпечує точну передачу даних між двома пристроями в мережі. В порівнянні з іншими протоколами транспортного рівня, такими як *UDP* (*User Datagram Protocol*), *TCP* має кілька переваг, які роблять його відмінним вибором для багатьох додатків.

Перш за все, *TCP* гарантує доставку даних у правильному порядку, що важливо для додатків, де послідовність інформації має значення, наприклад, в передачі файлів або відеоконференціях. Крім того, *TCP* включає в себе механізми перевірки наявності та управління витратами, які дозволяють адаптуватися до різних умов мережі і забезпечують високу стійкість до втрати даних. Ці характеристики роблять *TCP* надійним ідеальним вибором для додатків, де важлива стійкість та точність передачі даних.

РОЗДІЛ 2. ПРОЕКТУВАННЯ ПРОГРАМНОГО МОДУЛЮ ЗАХИЩЕНОЇ ПЕРЕДАЧІ ДАНИХ

2.1. Вибір технологій для розробки програмного модулю

C++ — це компільована мова, що призводить до високої ефективності виконання та керування пам'яттю. Велика частина *C++*-програм може бути написана без великого накладу високорівневих абстракцій, що дозволяє оптимізувати код для високопродуктивних систем або задач, де кожен цикл і кожен байт пам'яті може важити.

C++ забезпечує доступ до низькорівневих операцій, таких як маніпуляція покажчиками і прямий доступ до пам'яті, що дозволяє вам більш тісно контролювати аспекти програми, такі як обробка введення/виведення апаратури або оптимізація алгоритмів.

Для великих, складних проектів, особливо в сферах, де потрібна велика продуктивність, *C++* може бути вибором завдяки своїм можливостям об'єктно-орієнтованого програмування і модульності. Системи з великою базою коду можуть виграти від строгої типізації, широких можливостей оптимізації та прямого управління ресурсами.

C++ використовує компіляцію, що дозволяє генерувати виконавчий код для конкретної платформи, що забезпечує високу швидкість виконання програм. У випадках, де важлива швидкодія, наприклад, у вбудованих системах, графічному програмуванні або великих обчисленнях, *C++* може бути ефективнішим в порівнянні з інтерпретованими мовами, такими як *Python*.

C++ надає доступ до низькорівневих операцій та прямого управління пам'яттю, що дозволяє програмістам тісно контролювати ресурси системи. Це особливо корисно при розробці системно-орієнтованих програм, драйверів або вбудованих систем, де керування ресурсами грає важливу роль.

C++ надає багатий набір можливостей для оптимізації коду. Відстроювання з підтримкою оптимізаційного компілятора, можливість використання *inline*-функцій, оптимізація пам'яті та інші інструменти дозволяють розробникам досягти високої продуктивності та ефективності коду. У великих проектах, де оптимізація та швидкодія є ключовими факторами, *C++* може бути перевагою перед *Python*.

Python – це високорівнева, інтерпретована мова програмування, яка відома своєю простотою та читабельністю коду. Вона використовується для розробки різноманітних програм, включаючи веб-додатки, штучний інтелект, аналіз даних, наукові обчислення та багато іншого. Однією з ключових переваг мови *Python* є її велика спільнота розробників, що призводить до широкого використання сторонніх бібліотек і модулів, що полегшує роботу програмістів та прискорює розробку.

Python також славиться своєю незалежністю від платформи, що означає, що програми, написані на *Python*, можуть запускатися на різних операційних системах без значних змін. Ця універсальність сприяє популярності мови серед розробників та допомагає створювати ефективні та переносні програмні рішення.

Стандартна бібліотека *Python* - це великий набір корисних модулів та пакетів, які постачаються разом із самою мовою. Ці бібліотеки роблять *Python* потужним інструментом для різних завдань програмування. Наприклад, модуль *os* дозволяє взаємодіяти з операційною системою, забезпечуючи функції для роботи з файлами, каталогами та іншими операціями системного рівня.

Python та *C++* – це дві різні мови програмування зі своїми сильними та слабкими сторонами, і обираючи між ними, розробники враховують різні аспекти проекту. Однією з основних переваг *Python* є його простота та читабельність коду. Він використовує високорівневий синтаксис, що дозволяє розробникам виражати ідеї меншим обсягом коду, порівняно з *C++*. Це полегшує розробку, зменшує ймовірність помилок та сприяє швидшій ітерації у процесі програмування.

Ще однією важливою перевагою *Python* є його висока рівень абстракції, що дозволяє розробникам уникати деталей, пов'язаних з управлінням пам'яттю, що є характерним для *C++*. Це дозволяє фокусуватися на логікі програми, а не на низькорівневих операціях. Також, Python має розгалуження у вигляді широкого спектру бібліотек та модулів, що спрощує вирішення різноманітних задач без необхідності писати код "з нуля".

Нарешті, Python володіє великою спільнотою розробників та широким спектром готових рішень, що дозволяє швидше розробляти програми та знаходити відповіді на запитання. У той час як *C++* є потужною мовою для системного програмування та високоефективних додатків, *Python* надає зручний та швидкий спосіб розробки, що робить його популярним в областях, де важлива продуктивність та швидкість розробки.

Python та *Java* - дві популярні мови програмування з власними сферами застосування, проте *Python* вибирається багатьма розробниками через свою простоту та гнучкість. Перша велика перевага *Python* полягає в його читабельності коду та простому синтаксисі. Він дозволяє виражати ідеї меншим обсягом коду порівняно з Java, що робить розробку більш зрозумілою та продуктивною.

Другою важливою перевагою *Python* є швидкість розробки. Велика кількість готових бібліотек і модулів дозволяє розробникам швидко вирішувати різноманітні задачі без необхідності повторного винаходження колеса. Java, хоч і має потужний екосистему, іноді може вимагати більше коду для досягнення тих самих результатів, що вводить *Python* у виграш.

Третя перевага полягає в тому, що *Python* є більш гнучкою та легкою для вивчення мовою, особливо для новачків. Його динамічна типізація та простий синтаксис роблять його менш строгим і більш доступним для широкого кола розробників. Це сприяє швидшому освоєнню мови та зручності в її використанні на всіх етапах розробки.

У підсумку, вибір мови програмування завжди залежить від конкретних потреб та характеру проекту, але *C++* видається дуже привабливим варіантом

через свої переваги. Його простий і зрозумілий синтаксис дозволяє швидко створювати програми, зменшуючи кількість коду і ризик помилок.

Крім того, *C++* відомий своєю універсальністю та великою активною спільнотою розробників. Ця мова дозволяє розробникам більше часу приділяти самому розв'язанню завдань, а не вирішенню технічних труднощів, що забезпечує ефективніше та задовільне програмування.

2.2. Вибір алгоритму шифрування

На сьогодні використовуються широкі масштаби систем із як симетричним, так і асиметричним шифруванням. Обидва підходи мають свої позитивні та негативні аспекти. Після аналізу переваг і недоліків обох систем вибрано ту, яка оптимально відповідає вимогам конкретного програмного продукту.

Симетричне шифрування виділяється своєю високою швидкістю роботи в порівнянні з асиметричним варіантом – алгоритм останнього працює значно повільніше, десятки разів. Однак основний недолік симетричного шифрування виявляється в потребі публічної передачі ключів. Цей аспект важливий, оскільки при великій кількості учасників стає важко, а навіть практично неможливо забезпечити безпеку та конфіденційність інформації з використанням симетричного шифрування.

Advanced Encryption Standard (*AES*) представляє собою сучасний і дуже ефективний симетричний алгоритм шифрування, який широко використовується для захисту конфіденційної інформації в різних сферах, включаючи інтернет-передачу даних, зберігання інформації та інші аспекти кібербезпеки. *AES* замінив застарілий *DES* (*Data Encryption Standard*) і забезпечив вищий рівень безпеки. Алгоритм використовує ключі різної довжини (128, 192 або 256 біт), що дозволяє забезпечити гнучкість та високий рівень захисту від криптоаналізу.

Однією з суттєвих переваг *AES* є його висока швидкість обробки даних, що робить його ефективним для використання в реальному часі. Цей алгоритм

володіє ефективністю та стійкістю до різних видів криптоаналізу, забезпечуючи надійний рівень безпеки в сучасному цифровому середовищі.

Вибір розміру ключа є важливим аспектом у використанні *Advanced Encryption Standard* (*AES*), оскільки він безпосередньо впливає на безпеку і стійкість системи шифрування. *AES* підтримує ключі різної довжини, включаючи 128, 192 та 256 біт. Чим більший розмір ключа, тим більше можливостей для різноманітних комбінацій шифрування, що може забезпечити вищий рівень безпеки.

Звичайно, вибір оптимального розміру ключа повинен враховувати конкретні потреби конкретного застосування. У сучасних системах рекомендується використовувати ключі довжиною 256 біт для забезпечення максимальної стійкості до атак. Однак для деяких областей застосування, де вимоги до продуктивності важливі, може вибиратися менший розмір ключа, якщо це не суперечить конкретним вимогам безпеки. Такий гнучкий підхід дозволяє адаптувати систему *AES* до різноманітних сценаріїв використання, враховуючи співвідношення між безпекою і продуктивністю.

Асиметричне шифрування відзначається високою криптографічною стійкістю та можливістю використання для перевірки цілісності даних. Проте його основний недолік полягає в обмеженій швидкості виконання операцій шифрування і розшифрування, що зумовлено необхідністю проведення ресурсоємних операцій. Це може призводити до неприйнятних вимог до апаратної складової такої системи, що обмежує його застосування в деяких випадках[9].

RSA (*Rivest–Shamir–Adleman*) є асиметричним алгоритмом шифрування та підпису, який заснований на математичних властивостях складності факторизації великих простих чисел. Розроблений у 1977 році Рональдом Рівестом, Аді Шаміром та Леонардом Адлеманом, *RSA* використовує два ключі: публічний і приватний. Публічний ключ використовується для шифрування даних, а приватний - для їх розшифрування або для створення цифрового підпису. Одною

з суттєвих переваг *RSA* є висока стійкість до атак, пов'язаних з факторизацією великих чисел.

Незважаючи на велику криптографічну стійкість *RSA*, у деяких випадках важливо враховувати його обчислювальні витрати, оскільки алгоритм може вимагати значних обчислень для генерації ключів та шифрування даних.

Вибір розміру ключа для *RSA* є важливим етапом у забезпеченні ефективності та безпеки криптосистеми. Зазвичай розмір ключа визначається в бітах і може бути різним в залежності від конкретного застосування. Чим більший розмір ключа, тим більше вірогідність, що алгоритм залишиться стійким до атак.

Деякі стандарти рекомендують використовувати розмір ключа 2048 біт або більше для надійного захисту в сучасних умовах кібербезпеки, проте в окремих випадках може бути вибрано інший розмір, залежно від потреб конкретного застосування та вимог до продуктивності.

Швидкодія різних алгоритмів шифрування зображена на табл 2.1.

Таблиця 2.1

Швидкість шифрування різними алгоритмами

Алгоритм	Швидкість шифрування	Довжина ключа
DES	1-5 Мбіт/с	56 біт
3DES	0,33 Мбіт/с	168 біт
AES-128	15 Гбіт/с	128 біт
AES-192	10 Гбіт/с	192 біти
AES-256	5 Гбіт/с	256 біт

В результаті аналізу алгоритмів шифрування, для шифрування файлів було обрано два алгоритми RSA та AES. Вибір алгоритму надається користувачу, що в свою чергу додає більш гнучкі налаштування модуля шифрованої передачі даних.

2.3. Проектування архітектури для програмного модуля

Клієнт-серверна архітектура є моделлю розподіленого взаємодії в обчислювальних системах, де програма або додаток розділені на дві основні компоненти: клієнт і сервер. У цій архітектурі клієнт відповідає за користувацький інтерфейс і взаємодію з користувачем, тоді як сервер забезпечує обробку даних, зберігання ресурсів та виконання бізнес-логіки.

Один з ключових аспектів цієї архітектури - це взаємодія через мережу. Клієнт і сервер обмінюються даними за допомогою різних протоколів передачі даних, таких як *HTTP* в інтернет-додатках або *TCP*/*IP* в корпоративних системах. Це дозволяє розділити завдання і забезпечити ефективну комунікацію між компонентами.

Кожен з компонентів клієнт-серверної архітектури може функціонувати на різних фізичних пристроях чи віртуальних середовищах, дозволяючи розподілити обчислювальні ресурси та полегшити підтримку та розвиток системи. Такий підхід дозволяє досягти більшої масштабованості та ефективності в роботі програмного забезпечення.

Нещодавні тенденції в розробці програмного забезпечення включають в себе використання хмарних технологій, що ще більше підкреслює переваги клієнт-серверної архітектури, забезпечуючи гнучкість, масштабованість та ефективність в управлінні ресурсами.

Клієнт-серверна архітектура має кілька значущих переваг, які роблять її популярним варіантом для розробки програмного забезпечення. Ось кілька ключових переваг:

Ефективне управління ресурсами: Розділення функцій між клієнтом і сервером дозволяє оптимізувати використання ресурсів. Сервер відповідає за обробку даних, логіку додатку і управління ресурсами, тоді як клієнт забезпечує інтерфейс та взаємодію з користувачем.

Безпека і управління доступом: Сервер контролює доступ до ресурсів і даних, що сприяє підвищенню рівня безпеки. Централізований контроль також спрощує управління правами доступу та ідентифікацією користувачів.

Легко масштабується: Клієнт-серверна архітектура дозволяє легко масштабувати систему, додаючи нові клієнти або сервери в залежності від потреби. Це робить її гнучкою та ефективною для розробки великих і розподілених систем.

Економія пропускної здатності мережі: Завдяки розділенню функцій, обмін даними між клієнтом і сервером може бути оптимізований. Тільки необхідні дані передаються по мережі, що допомагає економити пропускну здатність та зменшує затримки.

Завдяки розділенню на клієнтську та серверну частину, різні типи клієнтів можуть легко взаємодіяти зі спільним сервером. Це дозволяє розробникам підтримувати різні платформи та пристрої.

Внесення змін в логіку або функціонал системи може бути спрощеним завдяки тому, що оновлення може бути внесено лише на серверній стороні, тоді як клієнти можуть залишатися незмінними.

Ці переваги роблять клієнт-серверну архітектуру важливим інструментом для розробників при створенні розподілених програмних систем.

Хоча клієнт-серверна архітектура має численні переваги, вона також володіє своїми недоліками. Деякі з найбільш суттєвих мінусів включають:

Вразливість сервера: Якщо сервер перестає працювати або стає недоступним, це може призвести до збою всієї системи. Клієнти зазвичай залежать від доступності сервера для виконання своїх функцій.

Залежність від мережі: Взаємодія між клієнтом і сервером вимагає передачі даних через мережу. Збільшення обсягу даних або нестабільність мережі може призвести до затримок у відповідях та зниження продуктивності.

Споживання ресурсів клієнтом: Клієнти можуть зберігати певні дані або стан, що ускладнює управління консистентністю та синхронізацією між різними клієнтами і сервером.

Загрози безпеки: Оскільки сервер відповідає за обробку багатьох операцій і зберігання даних, він може стати об'єктом атак і загроз для безпеки, особливо якщо не приділено достатньої уваги заходам безпеки.

Складність розгортання та оновлення: Оновлення та розгортання нових версій програм може бути складним завданням, особливо якщо потрібно забезпечити сумісність між старими та новими версіями клієнтів і серверів. Клієнт і сервер можуть використовувати різні технології та мови програмування, що може ускладнити обслуговування та розвиток системи. Не зважаючи на ці мінуси, багато з них можуть бути подолані за допомогою правильного проектування, впровадження та управління системою. Вибір архітектурного підходу повинен базуватися на конкретних потребах проекту і ураховувати його контекст та вимоги.

Доступність сервера є критичним аспектом для багатьох систем, особливо тих, які використовують клієнт-серверну архітектуру. Існує кілька причин, чому важливо, щоб сервер був завжди доступний:

Деякі бізнес-процеси та операції можуть бути невід'ємно пов'язаними з роботою сервера. Наприклад, у фінансових системах, електронних комерційних платформах чи інших важливих додатках неприпустимі перерви в роботі сервера, оскільки це може вести до втрати даних, клієнтів та фінансових втрат.

Відсутність доступу до сервера може призвести до зупинки або обмеження функціоналу для користувачів. Це особливо критично для онлайн-систем, де надто довгий відсутній час може призвести до розколу користувачів.

Доступність сервера також важлива для забезпечення безпеки системи. Відсутність доступу може виграти атакам інші можливості для вторгнень та злому безпеки.

Багато систем повинні дотримуватися угод щодо часу доступності (*SLA*) або інших вимог замовників. Недотримання цих угод може призвести до юридичних проблем та втрати довіри з боку клієнтів.

Багато додатків зберігають дані на сервері. Відсутність доступу може призвести до втрати або пошкодження даних, що може бути неприйнятним для багатьох організацій та клієнтів.

Завдяки постійному доступу до сервера можна уникнути втрат прибутку, особливо у випадках, коли сервер використовується для обробки транзакцій, виконання операцій інтернет-магазину чи інших фінансових операцій.

Таким чином, стабільність і доступність сервера є важливими аспектами для забезпечення ефективної роботи системи та задоволення потреб користувачів та бізнес-вимог.

2.3.1 Проектування модулю роботи з файлами

Програмний модуль для роботи з файлом, що перевіряє наявність, зчитує дані, а також створює та відкриває файл на зчитування, може бути важливим інструментом для оптимізації обробки та управління інформацією. Основні функціональні можливості цього модуля включають:

Перевірка наявності файла: Модуль може перевіряти наявність файлу в заданому шляху або за його іменем. Це дозволяє визначити, чи існує файл до подальших операцій.

Зчитування даних з файлу: Модуль може забезпечувати можливість зчитування даних з вказаного файлу. Це може бути корисно для подальшого аналізу, обробки чи використання інформації, що знаходиться в файлі.

Створення файлу: Якщо файл не існує, модуль може створювати його за вказаним шляхом або ім'ям. Це особливо корисно, якщо програмі потрібно створити файл для подальшого збереження даних.

Відкриття файлу на зчитування: Модуль може відкривати файл з можливістю зчитування. Це забезпечує можливість отримати доступ до вмісту файлу та використовувати його у подальших операціях обробки даних.

2.3.2. Проектування модуля шифрування даних

Програмний модуль шифрування файлів представляє собою інструмент для забезпечення безпеки конфіденційної інформації, зокрема при зберіганні або передачі файлів. Основні характеристики цього модуля включають:

Отримання шляху до файлу: Модуль отримує вхідним параметром шлях до цільового файлу, який потрібно зашифрувати. Це може бути абсолютний або відносний шлях до файлу.

Зчитування файлу: Після отримання шляху до файлу, модуль виконує операцію зчитування, щоб отримати вміст файлу. Це дозволяє модулю працювати з даними для подальшого шифрування.

Шифрування даних: Модуль використовує криптографічні алгоритми для шифрування вмісту файлу. Шифрування забезпечує захист від несанкціонованого доступу та зберігає конфіденційні дані у безпечному вигляді.

Запис шифрованого файлу: Після шифрування модуль записує отриманий зашифрований вміст назад у файл. Це дозволяє зберегти конфіденційні дані у зашифрованому вигляді, готовому для подальшого використання чи зберігання.

Обробка помилок та винятків: Модуль може містити механізми для обробки помилок, таких як неправильний шлях до файлу, відсутність доступу, чи невірний ключ шифрування. Це допомагає забезпечити стабільну роботу програмного модуля.

Застосування подібного модуля дозволяє забезпечити конфіденційність інформації під час зберігання чи обміну файлами, що особливо важливо в сферах, де безпека даних є пріоритетом.

2.3.3. Проектування модуля для розшифровки даних

Програмний модуль розшифрування файлів представляє собою компонент, який дозволяє відновити вихідний вміст файлу, раніше зашифрованого для забезпечення конфіденційності. Нижче представлено ключові функціональності такого модуля:

Отримання шляху до файлу: Модуль отримує вхідним параметром шлях до зашифрованого файлу, який потрібно розшифрувати. Це може бути абсолютний або відносний шлях до файлу.

Зчитування зашифрованого файлу: Після отримання шляху до файлу, модуль виконує операцію зчитування, щоб отримати вміст зашифрованого файлу.

Розшифрування даних: Модуль використовує криптографічні алгоритми та ключ розшифрування для відновлення вмісту файлу в його оригінальному вигляді. Цей процес забезпечує доступ до інформації, яка раніше була зашифрована.

Запис розшифрованого вмісту: Після розшифрування модуль записує отриманий вміст назад у файл. Це відновлює оригінальний файл та дозволяє його використовувати або зберігати у звичайному вигляді.

Обробка помилок та винятків: Модуль може включати механізми для обробки помилок, таких як невірний шлях до файлу, неправильний ключ розшифрування чи інші проблеми, що можуть виникнути під час операції розшифрування.

Застосування подібного модуля є ключовим у сценаріях, де інформація повинна бути зашифрована для забезпечення безпеки, а потім розшифрована для подальшого використання чи обміну.

2.4 Проектування мережевої частини

2.4.1 Вибір протоколу транспортного рівня

На даний час існує два основних протоколи транспортного рівня в моделі *TCP/IP – TCP* і *UDP*. Обидва ці протоколи використовуються для передачі даних в мережі, але вони мають різні характеристики та призначення.

Переваги *UDP* для передачі файлів: Швидкість: *UDP* є легким та швидким протоколом, оскільки він не вимагає встановлення з'єднання та не надає гарантії доставки даних. Ніякого витрачання часу на встановлення з'єднання:
Особливо корисний для великої кількості коротких відправлень даних.

Придатний для реального часу: Ідеальний для застосувань реального часу, де швидкість та невелике затримання є більш важливими, ніж доставка всіх пакетів.

Невисока навантаження на мережу: Має менший заголовок порівняно з *TCP*, що зменшує навантаження на мережу. Недоліки *UDP* для передачі файлів:

Втрата даних: Не надає гарантії доставки, тому може виникати втрата пакетів при передачі даних.

Не відновлення помилок: Відсутність механізму повторного надсилання чи відновлення даних, як у *TCP*, може призводити до втрати або пошкодження інформації.

Неупорядкованість даних: *UDP* не гарантує порядок доставки даних. Це може бути проблемою для деяких застосувань, де важливий порядок отримання даних.

Неактивне виявлення перевантаження мережі: Відсутність вбудованих механізмів для реагування на перевантаження мережі може призводити до втрати пакетів.

Неадекватний для великих обсягів даних: Для передачі великих обсягів даних, де важлива надійність, може бути краще використовувати *TCP*.

У випадках, де важливий швидкий обмін даними та втрата певної кількості пакетів є прийнятною, *UDP* може бути ефективним вибором для передачі файлів. Однак для сценаріїв, де важлива надійність та гарантія доставки, *TCP* може бути більш відповідним варіантом[8].

Переваги *TCP* для передачі файлів: Надійність передачі: *TCP* гарантує доставку даних та повторне надсилання в разі втрати або пошкодження пакетів.

Контроль порядку: *TCP* забезпечує відправку та отримання даних в правильному порядку, що важливо для багатьох застосувань.

Відновлення після розриву: Механізми відновлення з'єднання та перепідключення дозволяють відновлювати передачу даних після втрати чи перерви з'єднання.

Контроль потоку: *TCP* автоматично регулює швидкість передачі даних, щоб уникнути переповнення буферів. Здатність до встановлення з'єднання: Встановлення з'єднання та обмін синхронізаційними пакетами дозволяють визначити параметри передачі даних.

Недоліки *TCP* для передачі файлів: Високе навантаження на мережу: Має більший заголовок порівняно з *UDP*, що призводить до вищого навантаження на

мережу. Затримка передачі даних: Відправка даних може затримуватися через встановлення та управління з'єднанням.

Неідеальний для реального часу: Затримки та великі витрати ресурсів роблять його неідеальним для застосувань реального часу. Невідповідність для великих обсягів даних: Великий обсяг додаткових сегментів для керування з'єднанням може вплинути на швидкість передачі даних.

Постійна перевірка цілісності: Включає перевірку цілісності для кожного пакета, що може вплинути на швидкість передачі даних. Неактивне виявлення перевантаження мережі: Немає вбудованих механізмів для реагування на перевантаження мережі. У випадках, де важлива надійність та гарантія доставки, *TCP* є відмінним вибором для передачі файлів.

Однак для сценаріїв, де важливий швидкий обмін даними та втрата певної кількості пакетів є прийнятною, *UDP* може бути ефективним вибором.

2.4.2. Проектування модуля передачі зашифрованого файлу

Програмний модуль для роботи з мережею, який відкриває сокет, підключається за вказаною *IP*-адресою та передає файл, є важливим компонентом для обміну даними між вузлами в мережевому середовищі. Його основні функціональності включають:

Відкриття сокету: Модуль ініціює відкриття сокету, який є точкою вхідно-вихідного потоку даних для мережевого з'єднання. Це може бути здійснено використовуючи *TCP* або *UDP*, в залежності від вимог застосунку.

Підключення за вказаною *IP*-адресою: Модуль установлює мережеве з'єднання, підключаючись до іншого вузла за вказаною *IP*-адресою та можливо портом. Це може бути важливим в сценаріях, де важливо встановити точне мережеве з'єднання для обміну даними.

Передача файлу: Модуль забезпечує передачу файлу через встановлене мережеве з'єднання. Це може включати зчитування файлу, розбиття його на пакети та відправлення їх через сокет.

Обробка помилок та винятків: Модуль має вбудовані механізми для обробки помилок, таких як втрата з'єднання або неправильні параметри передачі файлу. Це підвищує надійність та стійкість в роботі з мережею.

Застосування такого модуля дозволяє програмам ефективно взаємодіяти через мережу, обмінюючись файлами чи іншими даними, і є важливим для багатьох мережевих застосунків, таких як передача файлів.

2.4.3. Проектування модуля отримання даних

Програмний модуль для роботи з мережею повинен слухати порт, при отримані підключення зчитати файл. Даний модуль відіграє важливу роль у створенні серверної частини мережевого застосунку. Основні функціональності такого модуля включають:

Модуль створює сокет та починає слухати вказаний порт на сервері. Це дозволяє іншим вузлам здійснювати з'єднання з цим портом для обміну даними.

Коли інший вузол намагається підключитися, модуль приймає з'єднання, створює окремий сокет для цього з'єднання та взаємодіє з клієнтом. Це може включати обмін даними про параметри з'єднання.

Обидва вузли повинні працювати використовуючи *TCP* з'єднання, так як воно забезпечує додаткову перевірку надсилання пакетів. Протокол *UDP* такого не забезпечує, що не підходить для передачі файлів в комп’ютерній мережі

Після встановлення з'єднання модуль може зчитувати файл, що передається через мережу. Це може включати зчитування файлу блоками або в цілому, залежно від вимог застосунку.

Модуль повинен містити механізми обробки помилок, таких як втрата з'єднання чи неправильні дані, для забезпечення стійкої роботи в мережевому середовищі.

Цей модуль визначає основну інфраструктуру для прийому та обробки з'єднань та даних через мережу, що робить його важливим елементом для серверних застосунків, обміну файлами та інших мережевих сценаріїв.

2.5 Висновки за розділом

У результаті проведеного проектування програмного модуля для захищеної передачі даних було ретельно вивчено та враховано важливі аспекти в сфері кібербезпеки та конфіденційності інформації. Вибір мови програмування, схеми роботи додатка та використання алгоритму шифрування є ключовими елементами, що забезпечують ефективну та безпечну передачу даних.

Обрана мова програмування *C++* відповідає вимогам проекту з точки зору продуктивності та можливостей розширення. Схема роботи додатка ретельно розроблена з метою мінімізації можливих вразливостей та максимізації ефективності обміну даними. Це включає в себе оптимальне використання ресурсів, контроль доступу до інформації та ефективний обмін ключами шифрування.

Обрана мова програмування *C++* виявилася відмінним вибором для реалізації програмного модуля для захищеної передачі даних. *C++* є високорівневою мовою, що відзначається простотою та зрозумілістю синтаксису, що значно полегшує розробку та підтримку коду.

Повсюдна популярність *C++* в області кібербезпеки та розробки безпечних додатків гарантує доступність великої кількості бібліотек та фреймворків, спрощуючи реалізацію різноманітних можливостей, зокрема, шифрування та безпечної передачі даних.

Алгоритм шифрування, обраний для реалізації безпечної передачі даних, відповідає сучасним стандартам криптографії та враховує високий рівень захисту від різноманітних атак. Його впровадження в проект дозволяє забезпечити надійну конфіденційність та цілісність даних.

Обраний алгоритм шифрування *Advanced Encryption Standard* (*AES*) є одним з найбільш надійних та широко використовуваних алгоритмів в області криптографії. *AES* був стандартизований Національним інститутом стандартів і технологій (NIST) і замінив застарілий *DES* (*Data Encryption Standard*). В основі успіху *AES* лежить його здатність поєднувати високий рівень безпеки з високою

ефективністю обчислень, що робить його ідеальним вибором для захисту конфіденційної інформації у програмному модулі для захищеної передачі даних.

AES використовує симетричний ключовий підхід, що означає, що той самий ключ використовується для як шифрування, так і розшифрування даних. Алгоритм має фіксований блоковий розмір (128 біт) і ключі розміром 128, 192 або 256 біт. Один із основних плюсів *AES* полягає в його оптимальній здатності протистояти різноманітним атакам, включаючи атаки типу "*brute force*" завдяки великій ключовій просторі.

Завдяки своїм властивостям стійкості та ефективності *AES* став стандартом для захищеного обміну конфіденційною інформацією в багатьох сферах, включаючи фінанси, медицину та інформаційні технології. Обраний алгоритм додає високий рівень безпеки до розробленого програмного модуля, забезпечуючи захищену передачу даних між вузлами мережі. Алгоритм роботи клієнта та сервера зображено на рис. 2.1

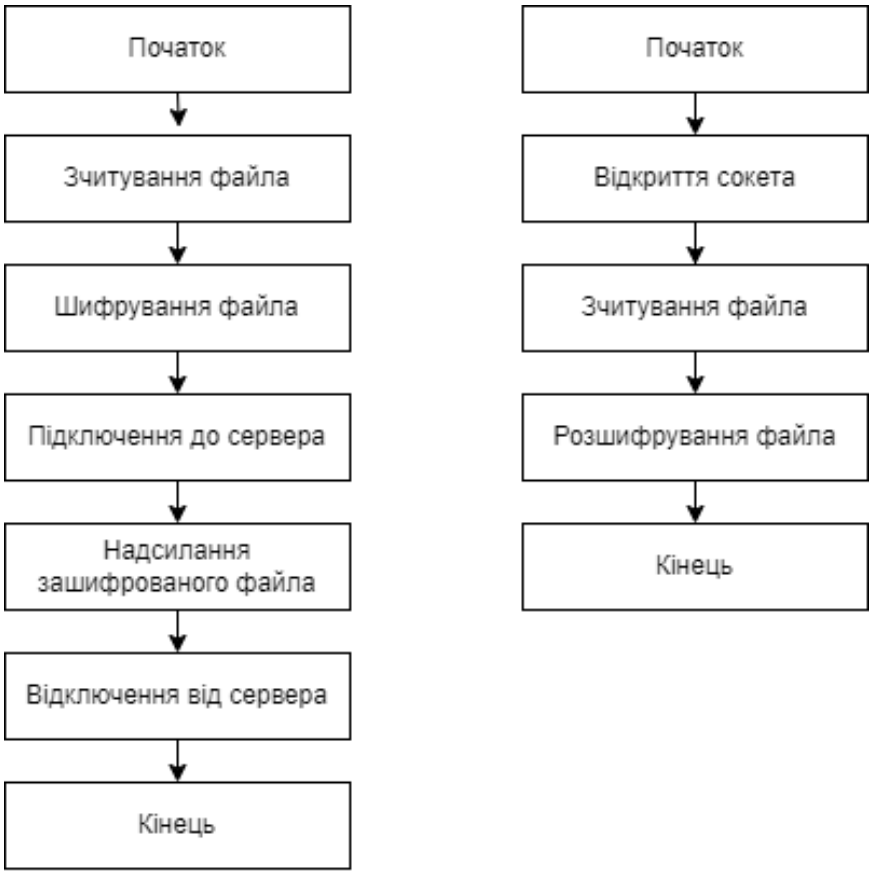

Рис. 2.1. Алгоритм роботи клієнтського та серверного додатку

РОЗДІЛ 3. РОЗРОБКА ПРОГРАМНОГО МОДУЛЮ ДЛЯ ЗАХИЩЕНОЇ ПЕРЕДАЧІ ДАНИХ

3.1. Програмна реалізація

3.1.1. Створення та налаштування проекту

Створення проекту *C++* у *Visual Studio* включає кілька етапів, таких як встановлення інтегрованого середовища розробки (*IDE*), створення нового проекту, конфігурацію параметрів компіляції та налаштування середовища розробки. Ось кроки для створення простого консольного проекту *C++* у *Visual Studio*:

Встановлення *Visual Studio*: Перш за все, вам потрібно встановити *Visual Studio*. Ви можете завантажити його з офіційного сайту *Microsoft*: *Visual Studio Downloads*.

Запуск *Visual Studio*: Після встановлення відкрийте *Visual Studio*. Виберіть "*Create a new project*" або "*File*" -> "*New*" -> "*Project...*"

Вибір типу проекту: Оберіть "*Installed*" у лівому меню. Виберіть "*Visual C++*" у області шаблонів. Обрати "*Empty Project*" або інший тип проекту за вашими потребами.

Задання імені та місця проекту: Вказати ім'я проекту та місце його збереження. Натисніть "*Create*".

Додавання файлу джерела: у *Solution Explorer* виберіть ваш проект правою кнопкою миші і виберіть "*Add*" -> "*New Item...*". Обрати "*C++ File (.cpp)*" та введіть ім'я файлу, наприклад, *main.cpp*.

Натисніть "*Add*".

3.1.2 Розробка модулю налаштувань

Збереження налаштувань програми у файлі є важливою практикою з точки зору зручності, можливості портування та збереження стану програми між

різними сеансами. Ось кілька причин, чому корисно зберігати налаштування у файлі:

Збереження налаштувань у файлі дозволяє програмі запам'ятовувати стан певних параметрів між різними запусками. Це особливо важливо, якщо програма має конфігураційні параметри, які користувач може налаштовувати за своїми потребами.

Завдяки збереженню налаштувань у файлі, користувачі можуть легко редагувати конфігураційні параметри, використовуючи текстовий редактор чи спеціальні інструменти. Це полегшує роботу зі структурою та значеннями параметрів.

Коли налаштування зберігаються у файлі, програма стає більш портабельною. Користувач може копіювати або передавати цей файл, щоб легко встановлювати ті ж самі налаштування на іншому пристрої чи обмінюватися ними з іншими користувачами.

Використання файлу для збереження налаштувань дозволяє програмі ефективно управляти конфігураційними параметрами. Це робить можливим використання інструментів для управління версіями, зберігання резервних копій та інших аспектів керованості конфігурацією.

Якщо у програмі використовуються конфіденційні або критичні дані, збереження їх у файлі дозволяє легше керувати доступом та застосовувати відповідні заходи безпеки до цього файлу.

Загалом, збереження налаштувань у файлі дозволяє розширити функціональність програми та зробити її більш гнучкою та зручною у використанні.

Переваги JSON для файлу налаштувань: *JSON* (*JavaScript Object Notation*) має простий та легко читабельний синтаксис, який добре підходить для конфігураційних файлів. Це полегшує вручну редагування файлу користувачами або розробниками. *JSON* є легким для використання та розуміння. Велика кількість мов програмування підтримують роботу з *JSON*, і існують багато бібліотек для роботи з ним у різних середовищах.

JSON дозволяє використовувати структури даних, такі як об'єкти та масиви, що дозволяє вам логічно групувати та організовувати конфігураційні параметри.

JSON підтримує різні типи даних, такі як рядки, числа, булеві значення, масиви та об'єкти, що робить його універсальним для зберігання різних видів налаштувань.

JSON легко піддається валідації та дозволяє легко виявляти помилки у структурі файлу. Багато текстових редакторів та інструментів надають можливості автодоповнення та перевірки належності до схеми *JSON*.

Недоліки *JSON* для файлу налаштувань: *JSON* не підтримує коментарів, що може ускладнити розуміння конфігураційного файлу. Це особливо боляче, коли потрібно пояснити чи документувати конкретні параметри.

JSON призначений для опису даних, а не виконання функцій. Якщо вам потрібно викликати функції або проводити складні обчислення при обробці налаштувань, *JSON* може бути обмеженням.

Для простих конфігураційних файлів *JSON* може бути зайвим, оскільки його структура може виглядати дещо надто складною для простих параметрів.

Хоча *JSON* базується на синтаксисі *JavaScript*, він не підтримує деякі вирази та можливості мови, такі як використання *undefined* чи вирази типу *new Date*().

Переваги *XML* для файлу налаштувань: XML (*eXtensible Markup Language*) має явну структуру тегів, що дозволяє створювати вкладені та ієрархічні конфігураційні файли. Це полегшує розуміння структури даних.

XML дозволяє вставляти коментарі, що полегшує документування та пояснення різних частин конфігурації. *XML* є розширюваним, що дозволяє легко додавати нові елементи та атрибути до конфігураційного файлу без значних змін у структурі. *XML* може представляти різні типи даних, включаючи рядки, числа, булеві значення та дати.

XML є мовою нейтральною до мов програмування, і багато мов програмування надають інструменти для роботи з *XML*. Недоліки *XML* для файлу налаштувань: *XML* може виглядати надто громіздким через велику кількість тегів

та зайвих символів порівняно з іншими форматами, що може призводити до менш ефективного використання місця.

XML-файли можуть бути менш зручними для читання та редагування вручну порівняно з іншими форматами, особливо для користувачів, які не знайомі з їх синтаксисом. У порівнянні з іншими форматами, *XML*-файли можуть бути більшими у розмірі через додаткові теги та структури що повторюються. *XML*-файли можуть бути важко перевіряти відносно до схеми, і це може призводити до виникнення помилок у структурі файлу.

Обираючи між *JSON* та *XML* для файлу налаштувань, важливо враховувати специфіку проекту та вимог користувачів, а також враховувати розглянуті переваги та недоліки кожного з цих форматів.

Для роботи з налаштуваннями створено клас *FileManager*.

Члени класу:

std::string jsonSettings: Рядок, в якому зберігається *JSON*-вміст з файлу "*settings.json*".

std::string AesKeyString: Рядок, в якому зберігається *AES*-ключ.

FileReader fileReader: Об'єкт класу *FileReader*, який використовується для читання файлу "*settings.json*".

FileReader keyReader: Об'єкт класу *FileReader*, який використовується для читання файлу, шлях до якого міститься в *keyFilePath*.

std::string IP: Рядок, в якому зберігається *IP*-адреса з *JSON*-вмісту.

std::string PORT: Рядок, в якому зберігається порт з *JSON*-вмісту.

std::string keyFilePath: Рядок, в якому зберігається шлях до файлу *AES*-ключа з *JSON*-вмісту.

Методи класу:

Конструктор *FileManager*(): Викликається при створенні об'єкту класу. Встановлює шлях до файлу "*settings.json*", читає його вміст, парсить *JSON* та зберігає значення відповідних полів.

Призначений для ініціалізації об'єкту та зчитування основних налаштувань з файлу.

Таким чином, клас *FileManager* відповідає за зчитування основних налаштувань з файлу "*settings.json*", парсинг *JSON*-вмісту та зберігання цих значень для подальшого використання. Крім того, він також використовує об'єкт класу *FileReader* для зчитування *AES*-ключа з файлу, шлях до якого отримує з *JSON*-вмісту.

Члени класу:

std::string filePath: Шлях до файлу, який буде читатися методами класу.

Методи класу:

void SetPath(const std::string& newPath): Метод для встановлення нового шляху до файлу.

bool readFileToString(std::string& content) const: Метод, який зчитує вміст файлу до рядка *std::string*. Параметр *content* передається за посиланням і буде містити вміст файлу після виклику методу.

3.1.3 Розробка клієнтського модулю шифрування файлів

Для клієнтського модуля шифрування створено клас *File Encryptor*.

Клас *FileEncryptor* реалізує функціональність шифрування файлу за допомогою *AES* ключа. Основна ідея полягає в тому, щоб забезпечити безпеку конфіденційної інформації, шифруючи вміст файлу, і забезпечити можливість використання власного *AES* ключа.

Члени класу:

filePath: Шлях до файлу, який планується зашифрувати.

keyPath: Шлях до файлу, в якому зберігається *AES* ключ.

plaintext: Рядок, який містить вміст файлу для шифрування.

key: *AES* ключ, який буде використовуватися для шифрування.

ctx: Контекст шифрування, що використовується бібліотекою *OpenSSL*.

Методи класу:

FileEncryptor(*const std::string& filePath, const std::string& keyPath*): Конструктор класу, приймає шлях до файлу та шлях до файлу для ключа.

void encryptFileWithKey(const std::string& aesKey): Метод для шифрування файлу з використанням переданого AES ключа.

void readInputFile(): Зчитує вміст файлу та зберігає його у змінну *plaintext*.

void initializeCipher(): Ініціалізує контекст шифрування.

void encryptData(): Шифрує дані, використовуючи контекст шифрування.

void writeEncryptedFile(): Записує зашифровані дані в новий файл.

void cleanup(): Виконує очищення контексту та інших ресурсів після завершення шифрування.

3.1.4. Розробка клієнтського модулю передачі файла

Клас *FileSender* призначений для надсилання файлу через мережу за допомогою TCP-сокету. Основні методи та члени класу включають:

Члени класу:

filePath: Шлях до файлу, який необхідно передати.

ipAddress: *IP*-адреса сервера, до якого буде встановлено з'єднання.

port: Порт сервера, на який буде встановлено з'єднання.

socketFD: Файловий дескриптор сокету.

Методи класу:

FileSender(const std::string& filePath, const std::string& ipAddress, int port): Конструктор класу, приймає шлях до файлу, IP-адресу та порт сервера.

void sendFile(): Метод для надсилання файлу по мережі.

bool initializeConnection(): Метод для ініціалізації з'єднання з сервером.

void sendFileContents(std::ifstream& fileStream): Метод для надсилання вмісту файлу через сокет.

void closeConnection(): Метод для закриття з'єднання з сервером.

3.1.5. Розробка серверного модулю отримання файла

Клас *FileReceiver* призначений для прийому файлу через мережу за допомогою *TCP*-сокетів на операційній системі Windows. Ось опис всіх його членів та методів:

Члени класу:

port: Ціле число, що вказує порт для прослуховування підключень.

listenerSocket: Сокет для прослуховування підключень.

clientSocket: Сокет клієнта, який використовується для прийому файлу.

Методи класу:

FileReceiver(*int port*): Конструктор класу, приймає порт для прослуховування.

~FileReceiver(): Деструктор класу, викликає метод *cleanup* для очищення ресурсів.

void receiveFile(): Головний метод класу, який ініціалізує прослуховування, приймає підключення та запускає прийом файлу.

bool initializeListener(): Ініціалізує сокет для прослуховування та встановлює його на порт.

bool acceptConnection(): Приймає підключення від клієнта.

void receiveFileContents(): Приймає вміст файлу через сокет та записує його у файл "*received_file.txt*".

void closeConnection(): Закриває сокет клієнта.

void cleanup(): Закриває всі сокети та очищає ресурси.

3.1.6 Розробка серверного модулю розшифрування файлів перевірки файлів

Клас *FileDecryptor* надає інтерфейс для розшифрування файлу за допомогою переданого *AES* ключа. Основні методи включають зчитування зашифрованого файлу, ініціалізацію контексту розшифрування, розшифрування даних та запис розшифрованих даних в новий файл.

Члени класу:

encryptedFilePath: Шлях до зашифрованого файлу.

keyFilePath: Шлях до файлу, в якому зберігається AES ключ.

ciphertext: Зашифровані дані.

key: *AES* ключ для розшифрування.

ctx: Контекст розшифрування, що використовується бібліотекою *OpenSSL*.

Методи класу:

FileDecryptor(const std::string& encryptedFilePath, const std::string& keyFilePath): Конструктор класу, приймає шлях до зашифрованого файлу та шлях до файлу для ключа.

void decryptFileWithKey(const std::string& aesKey): Метод для розшифрування файлу з використанням переданого *AES* ключа.

void readEncryptedFile(): Зчитує вміст зашифрованого файлу.

void initializeCipher(): Ініціалізує контекст розшифрування.

void decryptData(): Розшифровує дані.

void writeDecryptedFile(): Записує розшифровані дані в новий файл.

void cleanup(): Виконує очищення контексту та інших ресурсів після завершення розшифрування.

РОЗДІЛ 4.
ОХОРОНА НАВКОЛИШНЬОГО СЕРЕДОВИЩА

Забруднення літосфери є серйозною загрозою для навколишнього середовища, природи та, в кінцевому рахунку, для здоров'я людей. Літосфера, яка включає в себе земну поверхню, є ключовим компонентом природи, забезпечуючи життя рослин та тварин. Однак негативний вплив антропогенних дій, таких як викиди промислових речовин та надмірне використання хімічних речовин, призводить до хімічного забруднення ґрунту, що є основною складовою літосфери.

Хімічне забруднення ґрунту стає результатом викидів промислових виробництв, використання пестицидів та мінеральних добрив. Ці речовини можуть накопичуватися в ґрунті, порушуючи його природний баланс і впливаючи на рослинність. Надмірне використання хімічних речовин може також впливати на якість ґрунту та водних ресурсів, впливаючи на екосистеми та на водні резервуари, що є необхідними для життя багатьох видів.

Загальне розуміння проблем забруднення літосфери вимагає впровадження ефективних заходів з контролю та зменшення викидів, а також просвітницької роботи щодо екологічно відповідальної поведінки. Забруднення літосфери несе не лише негативні наслідки для природи, але і ставить під загрозу наше власне благополуччя. Тому, розглядаючи проблему забруднення літосфери, важливо розглядати шляхи збереження природи та планети як спільної відповідальності всього людства.

Фізичне забруднення ґрунту є іншою важливою аспектом проблеми літосферного забруднення. Однією з основних причин фізичного забруднення є накопичення відходів, яке може мати серйозний вплив на екосистеми та біорізноманіття. Наприклад, невідповідно утилізовані тверді відходи можуть накопичуватися на природних територіях, що призводить до засмічення та знищення природних екосистем.

Велике значення має також засмічення територій. Від сміттєвих полігонів до неправильного утилізації відходів від населення, засмічення призводить до порушення естетики природних ландшафтів та може впливати на якість повітря та водних джерел. Засмічення також може впливати на життя тварин та рослин, адже багато видів стають жертвами пластикових відходів та інших матеріалів.

Щоб зменшити фізичне забруднення літосфери, необхідно впроваджувати стратегії відновлення відходів, переробки та повторного використання. Ефективна система управління відходами, розвинута інфраструктура для переробки та активна участь громадян у програмах збору та вторинної переробки можуть сприяти збереженню літосфери та забезпеченню її стійкості у майбутньому.

Хімічне забруднення літосфери стає однією з найбільш актуальних та загрозливих екологічних проблем сучасності. Викиди промислових речовин, використання пестицидів та мінеральних добрив призводять до негативного впливу на якість ґрунту та його природний склад. Зокрема, забруднення хімічними речовинами може викликати зниження родючості ґрунту, порушення природного циклу рослинних елементів та впливати на розвиток різних форм життя в літосфері.

Використання пестицидів і хімічних добрив для підвищення урожайності сільськогосподарських культур призводить до накопичення шкідливих речовин у ґрунті. Це може мати серйозні наслідки для рослин, в тому числі для сільськогосподарських культур, і впливати на якість продукції. Крім того, відходи від промислових процесів можуть містити небезпечні хімічні сполуки, які потрапляють в ґрунт та можуть перейти в рослини, що в результаті може впливати на здоров'я тварин та людей, які споживають ці продукти.

Для запобігання хімічному забрудненню літосфери, необхідно впроваджувати екологічно чисті технології та методи сільськогосподарського виробництва. Важливо розробляти та застосовувати методи органічного сільськогосподарського виробництва, які дозволяють зберігати родючість ґрунту та знижувати використання хімічних речовин. Перехід до сталого та екологічно

відповідального сільськогосподарського виробництва може сприяти збереженню літосфери та забезпеченню стійкості агроекосистем.

Наслідки забруднення літосфери мають значущий вплив на рослинний світ, порушуючи природні екосистеми та викликаючи серйозні проблеми для різноманіття видів. Однією з основних наслідків є зниження родючості ґрунту через накопичення токсичних речовин та порушення природного циклу поживних речовин. Це може призвести до втрати рослинного покриву, загрози для сільськогосподарських урожаїв та загострення проблеми продовольства.

Негативний вплив на тваринний світ є ще однією важливою стороною наслідків літосферного забруднення. Токсичні речовини, які потрапляють в ґрунт, можуть накопичуватися в тканинах тварин, що призводить до отруєння та зниження чисельності популяцій. Це може викликати дисбаланс у природних екосистемах та призводити до зникнення деяких видів. Крім того, токсичні речовини можуть потрапляти в ланцюг живлення, впливаючи на здоров'я людей, які споживають продукти тваринного походження.

Забруднення літосфери створює загрозу для здоров'я людини, оскільки токсичні речовини можуть потрапляти в її організм через воду, продукти харчування та повітря. Це може викликати ряд захворювань, включаючи алергії, хронічні захворювання та рак. Таким чином, ефективний захист літосфери від забруднення є не лише необхідністю для природи, але і ключовим аспектом збереження здоров'я людей та забезпечення сталого розвитку.

Способи боротьби з забрудненням літосфери включають в себе широкий спектр заходів, спрямованих на зменшення викидів, покращення управління відходами та стимулювання сталого використання ресурсів. Ефективне вирішення проблеми вимагає впровадження екологічно чистих технологій у виробництві, що дозволяє зменшити викиди токсичних речовин та інших забруднюючих вуглеводень. Також важливо розвивати та впроваджувати методи органічного сільськогосподарського виробництва, які дозволяють підтримувати родючість ґрунтів без використання шкідливих хімікатів.

Повторне використання та вторинна переробка відходів є ще одним ключовим аспектом стратегії боротьби з забрудненням літосфери. Системи вторинної переробки дозволяють використовувати ресурси більш ефективно та зменшувати обсяги відходів, що потрапляють на сміттєві полігони чи в природу. Заохочення громадян до відділення та сортування відходів, а також розвиток інфраструктури для їх подальшої переробки є важливими складовими цього підходу.

Екологічна освіта та підвищення екологічної свідомості громадян є необхідним елементом вирішення проблем забруднення літосфери. Широка розповсюдженість інформації про наслідки забруднення та ефективні методи його запобігання може мотивувати людей до відповідального ставлення до природи та власного внеску у збереження літосфери. Всебічний підхід до цих способів боротьби може сприяти сталому розвитку та збереженню літосфери для майбутніх поколінь.

Світові ініціативи та міжнародні домовленості грають важливу роль у регулюванні проблем забруднення літосфери та здійсненні спільних зусиль для її охорони. Кіотський протокол, укладений в 1997 році, спрямований на зменшення викидів парникових газів, які впливають на клімат і можуть мати додатковий вплив на якість ґрунту. Цей протокол викликав широкий обговорення та заохочення країн світу долучитися до зусиль з мінімізації впливу промисловості на навколишнє середовище.

Паризька угода, укладена в 2015 році, є ще однією ключовою міжнародною ініціативою, спрямованою на боротьбу зі зміною клімату та стимулювання сталого розвитку. Одним із завдань угоди є зменшення викидів та підтримка країн у переході до низьковуглецевих технологій, що може сприяти зменшенню забруднення літосфери через зменшення використання шкідливих речовин.

Цілі сталого розвитку ООН є комплексним підходом до вирішення екологічних проблем, включаючи забруднення літосфери. Зокрема, мета № 15 "Життя на суші" спрямована на захист, відновлення та раціональне використання екосистем на суходолі. Це включає у себе також заходи з запобігання та

зменшення деградації ґрунтів, що може бути спричинено забрудненням літосфери.

Міжнародні угоди та ініціативи є важливим кроком у вирішенні глобальних проблем забруднення літосфери, оскільки вони стимулюють співпрацю країн та спільні зусилля для збереження природи та забезпечення сталого розвитку планети.

Проведене дослідження підтверджує, що забруднення літосфери є серйозним викликом для природи та людства. Хімічне та фізичне забруднення ґрунту має руйнівний вплив на природні екосистеми, знижуючи родючість ґрунту та призводячи до втрати біорізноманіття. Наслідки цього забруднення впливають на тваринний світ та загрожують здоров'ю людини через споживання забруднених продуктів.

Способи боротьби з забрудненням літосфери, такі як використання екологічно чистих технологій, вторинна переробка відходів та екологічна освіта, визначаються як ефективні стратегії для збереження природи та забезпечення сталого розвитку. Важливість цих заходів підкреслюється світовими ініціативами, такими як Кіотський протокол, Паризька угода та Цілі сталого розвитку ООН, які встановлюють конкретні цілі та завдання для країн у сфері екології та зменшення викидів парникових газів.

У зв'язку з викликами, які ставить перед сучасністю забруднення літосфери, необхідною є активна участь громадян, компаній та урядів для успішної реалізації запропонованих заходів.

ВИСНОВКИ

Під час роботи над кваліфікаційною роботою було розглянуто поняття шифрування. Проаналізовано сучасні проблеми безпечної передачі даних, їх походження та спосіб додаткового захищення. Було виявлено критичну проблему безпеки при передачі не шифрованих даних. Підхід додаткового шифрування дозволить збільшити захист даних при передачі їх по мережі Інтернет.

Також критичним є розмір ключа. Необхідно створити всі умови, щоб при недостатньому розмірі ключа система видавала відповідні помилки.

Саме при такій умові, ситуація, коли дані було розшифровано через недостатню довжину ключа, зловмисники змогли отримати розшифровані дані мінімізується.

У сучасному інформаційному суспільстві, де обмін конфіденційною інформацією є необхідною складовою багатьох сфер, питання безпеки даних стає надзвичайно важливим. Зростання кількості та складності кіберзагроз, а також розширення можливостей технологій зберігання та передачі даних створюють серйозні виклики для забезпечення конфіденційності та цілісності інформації. У цьому контексті розробка програмних модулів, які забезпечують захищену передачу даних, набуває великого значення для забезпечення безпеки та захисту інформації в мережевих взаємодіях.

Вибір теми "програмний модуль захищеної передачі даних" обумовлений необхідністю розробки ефективних і сучасних методів забезпечення безпеки в мережевих комунікаціях. У цьому контексті особливу увагу заслуговує шифрування *AES* та протокол *TCP*. Шифрування *Advanced Encryption Standard* (*AES*) визнано надійним та широко використовується у світі криптографії, тоді як протокол *TCP* забезпечує забезпечену та надійну передачу даних в мережі. Розробка програмного модуля, який поєднує ці дві технології, визначається актуальністю та потребою у засобах, що гарантують конфіденційність даних при їх передачі через відкриті мережеві канали.

Шифрування *Advanced Encryption Standard* (*AES*) представляє собою блочний шифр, який визначається своєю високою стійкістю та широким спектром застосувань. Алгоритм використовує ключ з фіксованою довжиною (128, 192 або 256 біт), що робить його відмінним вибором для захисту конфіденційності даних у різних сценаріях.

Однією з основних переваг *AES* є його ефективність та висока швидкодія. Алгоритм оптимізований для апаратного та програмного виконання, що робить його практично застосовним в широкому спектрі пристроїв та систем. Додатково, *AES* відомий своєю стійкістю до криптоаналітичних атак, таких як диференціальний та лінійний криптоаналіз, що підсилює впевненість в його безпеці.

Істотним аспектом є також можливість використання різних режимів роботи AES (наприклад, *CBC*, *GCM*), що розширює спектр застосувань шифрування в залежності від конкретних вимог системи чи додатку. У контексті програмного модуля захищеної передачі даних використання AES надає надійність та гнучкість для забезпечення конфіденційності даних під час їхньої передачі через мережу.

Протокол *Transmission Control Protocol* (*TCP*) визначається як надійний та з'єднані орієнтований протокол транспортного рівня, призначений для ефективної та послідовної передачі даних між двома вузлами в мережі. Аналізуючи протокол *TCP*, визначаємо ключові аспекти, які роблять його ідеальним кандидатом для використання у захищеному програмному модулі передачі даних.

Однією з визначальних рис *TCP* є його здатність забезпечити послідовність доставки даних, тобто дані, які відправляються відправником, приходять в тому ж порядку до отримувача. Це особливо важливо в контексті безпечної передачі даних, де збереження послідовності має визначальне значення, при передачі файлів.

Додатково, *TCP* використовує механізми витрат контролю та підтвердження прийому даних, що забезпечує надійність передачі. У випадку

виникнення помилок чи втрати пакетів, *TCP* може відновлювати з'єднання та відновлювати втрачені дані, що робить його дуже стійким у непередбачуваних умовах мережі.

Також важливим аспектом є широка підтримка *TCP* в різних операційних системах та пристроях, що забезпечує сумісність та надійність роботи протоколу у різних середовищах. Застосування протоколу *TCP* у програмному модулі захищеної передачі даних дозволяє забезпечити надійність, послідовність та стійкість передачі інформації, необхідних для забезпечення безпеки даних у мережі.

У процесі вивчення та розробки програмного модуля захищеної передачі даних з використанням шифрування *AES* та протоколу *TCP* виявлено можливості спрощення реалізації безпеки. Однією з ключових вагомих альтернатив є використання готових криптографічних бібліотек, які надають високорівневі інтерфейси для роботи із шифруванням. Використання таких бібліотек може значно полегшити розробку, зменшити ймовірність помилок та забезпечити високий стандарт безпеки.

Зокрема, можливо використання криптографічних бібліотек, таких як *OpenSSL* або *Crypto*++, для імплементації алгоритмів шифрування та дешифрування на рівні даних. Це дозволяє використовувати перевірені та оптимізовані реалізації шифрування, позбавляючи розробників від необхідності написання власного коду криптографії.

Додатково, можливо розглядати різні стратегії керування ключами, такі як використання систем керування ключами (*KMS*), що дозволяють автоматизувати генерацію, зберігання та обмін ключами шифрування. Це допоможе впровадити ефективний та безпечний механізм обробки ключів, зменшуючи ризики, пов'язані з їх управлінням.

В результаті розробки та аналізу програмного модуля захищеної передачі даних з використанням шифрування *AES* та протоколу *TCP* виявлено кілька напрямків для подальших досліджень та удосконалення розробленого рішення.

Розширення функціональності: Майбутні дослідження можуть бути спрямовані на розширення функціональності програмного модуля. Додавання підтримки інших алгоритмів шифрування, розробка механізмів аутентифікації користувачів чи підтримка інших протоколів передачі даних може розширити сферу застосування розробленого модуля та його узгодженість з вимогами безпеки

Оптимізація продуктивності: Дослідження може фокусуватися на оптимізації продуктивності програмного модуля. Використання методів паралельного обчислення, оптимізації алгоритмів чи удосконалення механізмів керування ресурсами може покращити швидкодію та ефективність модуля в умовах високих навантажень та обмежених ресурсів.

Аналіз стійкості до нових загроз: Здійснення аналізу та вдосконалення стійкості програмного модуля до нових кіберзагроз є важливим напрямком подальших досліджень. Розгляд можливих атак, їхніх варіацій та розробка механізмів, що виявлять та протистояти новим загрозам, допоможе забезпечити тривалу ефективність та захищеність модуля в майбутньому.

Застосування у специфічних галузях: Можливості використання розробленого програмного модуля можуть бути досліджені в специфічних галузях, таких як фінанси, медицина чи інші сфери, де конфіденційність даних має критичне значення. Адаптація та вдосконалення модуля для конкретних вимог цих галузей може покращити його придатність та розширити сферу його застосування.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Положення про дипломні роботи випускників Національного Авіаційного Університету / Уклад.: О. П. Слободян. С. В. Бойченко, О. В, Іванченко – К.: НАУ, 2017.- 63 с.
2. Вербицький О.В. Введення в криптологію. – Львів: Наук.–техн. літ., 1998. – 248 с.
3. Математичні основи криптографії: навч. посібник / Г.В. Кузнецов, В.В. Фомичов, С.О.Сушко, Л.Я. Фомичова. – Д.: Національний гірничий університет, 2004. – Ч.1. – 391 с.
4. Захарченко М. В. Асиметричні методи шифрування в телекомунікаціях – О.: ОНАЗ, 2011. – 184 с.
5. Бабак В. П. Теоретичні основи захисту інформації : підручник // Ба–бак В. П. – Книжкове видавництво НАУ, 2008. – 752 с.
6. *P. Gauravaram. Cryptanalysis of a class of cryptographic hash functions / P. Gauravaram, J. Kelsey //Cryptology ePrint Archive. – 30 с. – Режим доступу до ресурсу : http://eprint.iacr.org/2007/277.pdf*
7. Горбенко І.Д. Захист інформації в інформаційно–телекомунікаційних системах // І.Д. Горбенко, Т.О. Гріненко. – Х. : 2004. – 222 с
8. Юдін О.К. Захист інформації в мережах передачі даних : Підручник / О.К. Юдін, О.Г. Корченко, Г.Ф. Конахович. – К. : Видавництво «*DIRECTLINE*», 2009. – 714 с.
9. Горохов С. М. Сучасні криптографічні системи: навч. посіб. / С. М. Горохов, Л. Г. Йона, О. В. Онацький; за ред. М. В. Захарченка. – Одеса: ОНАЗ ім. О. С. Попова, 2007. – 152 с.

ДОДАТОК А

Лістинг коду клієнтської частини

A.1 main.cpp

```
#include <iostream>
#include "ArduinoJson6.h"

#include <iostream>
#include <fstream>
#include <string>
#include <iostream>
#include <fstream>
#include <openssl/evp.h>
#include <openssl/rand.h>
#include <winsock2.h>
#pragma comment(lib, "ws2_32.lib")

class FileReader {
public:
    void SetPath(const std::string& newPath) { filePath = newPath; }
    bool readFileToString(std::string& content) const {
        std::ifstream fileStream(filePath);

        if (!fileStream.is_open()) {
            // Обробка помилки читання файлу
            std::cerr << "Не вдалося відкрити файл: " << filePath << std::endl;
            return false;
        }

        // Отримання розміру файлу
        fileStream.seekg(0, std::ios::end);
        size_t fileSize = fileStream.tellg();
        fileStream.seekg(0, std::ios::beg);

        // Виділення буфера для збереження даних файлу
        content.resize(fileSize);

        // Зчитування даних з файлу в std::string
        fileStream.read(&content[0], fileSize);

        // Закриття файлу
        fileStream.close();

        return true;
```

```
    }

private:
    std::string filePath;  // Шлях до файлу
};
std::string SETTINGS_PATH = { "D:\\210\\1\\settings.json" };
class FileManager
{
public:
    FileManager()
    {
        fileReader.SetPath(SETTINGS_PATH);
        fileReader.readFileToString(jsonSettings);

        DynamicJsonDocument jsonDoc(1024);  // Розмір буфера для парсингу (змініть його відповідно до вашого JSON)


        DeserializationError error = deserializeJson(jsonDoc, jsonSettings.c_str());

        if (error) {
            // Обробка помилок парсингу
            std::cerr << "Помилка парсингу JSON: " << error.c_str() << std::endl;
            return;
        }

        // Отримання значень полів
        keyFilePath = jsonDoc["key_file_path"].as<std::string>();
        IP = jsonDoc["ip"].as<std::string>();
        PORT= jsonDoc["port"].as<std::string>();

        // Вивід отриманих значень
        std::cout << "key_file_path: " << keyFilePath << std::endl;
        std::cout << "ip: " << IP << std::endl;
        std::cout << "port: " << PORT << std::endl;
        keyReader.SetPath(keyFilePath);
        keyReader.readFileToString(AesKeyString);
        std::cout << "AesKeyString size [" << AesKeyString.size() << "]\n";
    }

private:
    std::string jsonSettings;
    std::string AesKeyString;
    FileReader fileReader;
```

```
    FileReader keyReader;
    std::string IP;
    std::string PORT;
    std::string keyFilePath;
};

class FileSender {
public:
    FileSender(const std::string& filePath, const std::string& ipAddress, int port)
        : filePath(filePath), ipAddress(ipAddress), port(port),
socketFD(INVALID_SOCKET) {
        WSADATA wsaData;
        if (WSAStartup(MAKEWORD(2, 2), &wsaData) != 0) {
            std::cerr << "Failed to initialize Winsock\n";
        }
    }

    ~FileSender() {
        cleanup();
        WSACleanup();
    }

    // Метод для надсилання файлу по мережі
    void sendFile() {
        if (initializeConnection()) {
            std::ifstream fileStream(filePath, std::ios::binary);
            if (fileStream.is_open()) {
                sendFileContents(fileStream);
                fileStream.close();
                closeConnection();
                std::cout << "File sent successfully.\n";
            }
            else {
                std::cerr << "Unable to open file for reading.\n";
            }
        }
        else {
            std::cerr << "Unable to initialize connection.\n";
        }
    }

private:
    std::string filePath;  // Шлях до файлу, який необхідно передати
    std::string ipAddress;  // IP-адреса сервера
```

```
    int port;  // Порт сервера
    SOCKET socketFD;  // Сокет

    // Метод для ініціалізації з'єднання з сервером
    bool initializeConnection() {
        // Створення сокету
        socketFD = socket(AF_INET, SOCK_STREAM, 0);
        if (socketFD == INVALID_SOCKET) {
            std::cerr << "Error creating socket: " << WSAGetLastError() << "\n";
            return false;
        }

        // Підготовка структури адреси сервера
        sockaddr_in serverAddress;
        serverAddress.sin_family = AF_INET;
        serverAddress.sin_port = htons(port);
        inet_pton(AF_INET, ipAddress.c_str(), &(serverAddress.sin_addr));

        // З'єднання з сервером
        if (connect(socketFD, reinterpret_cast<struct sockaddr*>(&serverAddress),
sizeof(serverAddress)) == SOCKET_ERROR) {
            std::cerr << "Error connecting to server: " << WSAGetLastError() << "\n";
            return false;
        }

        return true;
    }

    // Метод для надсилання вмісту файлу через сокет
    void sendFileContents(std::ifstream& fileStream) {
        const int bufferSize = 1024;
        char buffer[bufferSize];
        ssize_t bytesRead;

        while (!fileStream.eof()) {
            fileStream.read(buffer, bufferSize);
            bytesRead = fileStream.gcount();

            if (send(socketFD, buffer, bytesRead, 0) == SOCKET_ERROR) {
                std::cerr << "Error sending file: " << WSAGetLastError() << "\n";
                break;
            }
        }
    }
```

```
    // Метод для закриття з'єднання
    void closeConnection() {
        closesocket(socketFD);
    }

    // Метод для очищення ресурсів
    void cleanup() {
        if (socketFD != INVALID_SOCKET) {
            closeConnection();
        }
    }
};


class FileEncryptor {
public:
    FileEncryptor(const std::string& filePath, const std::string& keyPath)
        : filePath(filePath), keyPath(keyPath), ctx(nullptr) {}

    // Метод для шифрування файлу з використанням переданого AES ключа
    void encryptFileWithKey(const std::string& aesKey) {
        readInputFile();  // Зчитує вміст файлу
        key = aesKey;  // Встановлює переданий AES ключ
        initializeCipher();  // Ініціалізує контекст шифрування
        encryptData();  // Шифрує дані
        writeEncryptedFile();  // Записує зашифровані дані в файл
        cleanup();  // Виконує очищення після завершення шифрування
        std::cout << "File encrypted successfully.\n";
    }
    std::string retEncFile() { return filePath; }
private:
    std::string filePath;  // Шлях до файлу для шифрування
    std::string keyPath;  // Шлях до файлу, в якому зберігається ключ

    std::string plaintext;  // Текст для шифрування
    std::string key;  // AES ключ для шифрування

    EVP_CIPHER_CTX* ctx;  // Контекст шифрування

    // Метод для зчитування вмісту файлу
    void readInputFile() {
        std::ifstream inputFile(filePath, std::ios::binary);
```

```
        plaintext = std::string((std::istreambuf_iterator<char>(inputFile)),
std::istreambuf_iterator<char>());
        inputFile.close();
    }

    // Метод для ініціалізації контексту шифрування
    void initializeCipher() {
        ctx = EVP_CIPHER_CTX_new();
        EVP_EncryptInit_ex(ctx, EVP_aes_256_cbc(), NULL, reinterpret_cast<const
unsigned char*>(key.c_str()), NULL);
    }

    // Метод для шифрування даних
    void encryptData() {
        int ciphertextLen = 0;
        int len = 0;
        std::string ciphertext(plaintext.length() +
EVP_CIPHER_block_size(EVP_aes_256_cbc()), '\0');

        EVP_EncryptUpdate(ctx, reinterpret_cast<unsigned char*>(&ciphertext[0]),
&len,
            reinterpret_cast<const unsigned char*>(plaintext.c_str()),
plaintext.length());
        ciphertextLen += len;

        EVP_EncryptFinal_ex(ctx, reinterpret_cast<unsigned
char*>(&ciphertext[ciphertextLen]), &len);
        ciphertextLen += len;
        ciphertext.resize(ciphertextLen);

        plaintext.clear();  // Очищаємо оригінальні дані, якщо потрібно
        plaintext.shrink_to_fit();
        plaintext.swap(ciphertext);
    }
    // Метод для запису зашифрованих даних в файл
    void writeEncryptedFile() {
        std::ofstream outputFile(filePath + ".enc", std::ios::binary);
        outputFile << plaintext;
        outputFile.close();
    }
    // Метод для очищення контексту та інших ресурсів
    void cleanup() {
        EVP_CIPHER_CTX_free(ctx);
    }
```

```
};
int main(int argc, char* argv[]) {
    // Перевірка, чи передано достатньо аргументів
    if (argc < 3) {
        std::cerr << "Usage: " << argv[0] << " <your_string>" << std::endl;
        return 1; // Повертаємо 1, означає помилку
    }
    std::string filePath = argv[1];
    std::string keyFilePath = argv[2];

    // Зчитує AES ключ з файлу
    std::ifstream keyFile(keyFilePath);
    std::string aesKey((std::istreambuf_iterator<char>(keyFile)),
std::istreambuf_iterator<char>());
    keyFile.close();

    FileEncryptor encryptor(filePath, keyFilePath);
    encryptor.encryptFileWithKey(aesKey);
    FileSender sender(encryptor.retEncFile());
    sender.sendFile();
    return 0;
}
```

ДОДАТОК В

Лістинг коду серверної частини

```
#include <iostream>
#include <fstream>
#include <winsock2.h>

#pragma comment(lib, "ws2_32.lib")



#include <iostream>
#include <fstream>
#include <openssl/evp.h>

class FileDecryptor {
public:
   FileDecryptor(const std::string& encryptedFilePath, const std::string& keyFilePath)
       : encryptedFilePath(encryptedFilePath), keyFilePath(keyFilePath), ctx(nullptr) {}

   // Метод для розшифрування файлу з використанням переданого AES ключа
   void decryptFileWithKey(const std::string& aesKey) {
       readEncryptedFile();  // Зчитує вміст зашифрованого файлу
       key = aesKey;  // Встановлює переданий AES ключ
       initializeCipher();  // Ініціалізує контекст розшифрування
       decryptData();  // Розшифровує дані
       writeDecryptedFile();  // Записує розшифровані дані в новий файл
       cleanup();  // Виконує очищення після завершення розшифрування
       std::cout << "File decrypted successfully.\n";
   }
```

```
private:
    std::string encryptedFilePath;  // Шлях до зашифрованого файлу
    std::string keyFilePath;  // Шлях до файлу, в якому зберігається AES ключ

    std::string ciphertext;  // Зашифровані дані
    std::string key;  // AES ключ для розшифрування

    EVP_CIPHER_CTX* ctx;  // Контекст розшифрування

    // Метод для зчитування вмісту зашифрованого файлу
    void readEncryptedFile() {
        std::ifstream encryptedFile(encryptedFilePath, std::ios::binary);
        ciphertext = std::string((std::istreambuf_iterator<char>(encryptedFile)),
std::istreambuf_iterator<char>());
        encryptedFile.close();
    }

    // Метод для ініціалізації контексту розшифрування
    void initializeCipher() {
        ctx = EVP_CIPHER_CTX_new();
        EVP_DecryptInit_ex(ctx, EVP_aes_256_cbc(), NULL, reinterpret_cast<const
unsigned char*>(key.c_str()), NULL);
    }

    // Метод для розшифрування даних
    void decryptData() {
        int plaintextLen = 0;
        int len = 0;
```

```
        std::string plaintext(ciphertext.length() + EVP_CIPHER_block_size(EVP_aes_256_cbc()), '\0');

        EVP_DecryptUpdate(ctx, reinterpret_cast<unsigned char*>(&plaintext[0]), &len,
            reinterpret_cast<const unsigned char*>(ciphertext.c_str()), ciphertext.length());
        plaintextLen += len;

        EVP_DecryptFinal_ex(ctx, reinterpret_cast<unsigned char*>(&plaintext[plaintextLen]), &len);
        plaintextLen += len;

        plaintext.resize(plaintextLen);

        ciphertext.clear();  // Очищаємо зашифровані дані, якщо потрібно
        ciphertext.shrink_to_fit();
        ciphertext.swap(plaintext);
    }

    // Метод для запису розшифрованих даних в файл
    void writeDecryptedFile() {
        std::ofstream decryptedFile(encryptedFilePath + ".dec", std::ios::binary);
        decryptedFile << ciphertext;
        decryptedFile.close();
    }

    // Метод для очищення контексту та інших ресурсів
    void cleanup() {
        EVP_CIPHER_CTX_free(ctx);
```

```
    }
};

class FileReceiver {
public:
    FileReceiver(int port)
        : port(port), listenerSocket(INVALID_SOCKET),
clientSocket(INVALID_SOCKET) {
        WSADATA wsaData;
        if (WSAStartup(MAKEWORD(2, 2), &wsaData) != 0) {
            std::cerr << "Failed to initialize Winsock\n";
        }
    }

    ~FileReceiver() {
        cleanup();
        WSACleanup();
    }

    // Метод для прийому файлу через мережу
    void receiveFile() {
        if (initializeListener()) {
            if (acceptConnection()) {
                receiveFileContents();
                closeConnection();
                std::cout << "File received successfully.\n";
            }
            else {
                std::cerr << "Failed to accept connection.\n";
            }
```

```
        }
        else {
            std::cerr << "Failed to initialize listener.\n";
        }
    }

private:
    int port;  // Порт для прослуховування
    SOCKET listenerSocket;  // Сокет для прослуховування підключень
    SOCKET clientSocket;  // Сокет клієнта

    // Метод для ініціалізації сокету для прослуховування
    bool initializeListener() {
        // Створення сокету для прослуховування
        listenerSocket = socket(AF_INET, SOCK_STREAM, 0);
        if (listenerSocket == INVALID_SOCKET) {
            std::cerr << "Error creating listener socket: " << WSAGetLastError() <<
"\n";
            return false;
        }

        // Підготовка структури адреси
        sockaddr_in serverAddress;
        serverAddress.sin_family = AF_INET;
        serverAddress.sin_port = htons(port);
        serverAddress.sin_addr.s_addr = INADDR_ANY;

        // Прив'язка сокету до адреси
        if (bind(listenerSocket, reinterpret_cast<struct sockaddr*>(&serverAddress),
sizeof(serverAddress)) == SOCKET_ERROR) {
```

```
            std::cerr << "Error binding listener socket: " << WSAGetLastError() << "\n";
            closesocket(listenerSocket);
            return false;
        }

        // Перехід в режим прослуховування
        if (listen(listenerSocket, SOMAXCONN) == SOCKET_ERROR) {
            std::cerr << "Error listening for connections: " << WSAGetLastError() <<
"\n";
            closesocket(listenerSocket);
            return false;
        }

        return true;
    }

    // Метод для прийому підключення
    bool acceptConnection() {
        clientSocket = accept(listenerSocket, NULL, NULL);
        if (clientSocket == INVALID_SOCKET) {
            std::cerr << "Error accepting connection: " << WSAGetLastError() << "\n";
            return false;
        }
        return true;
    }

    // Метод для прийому вмісту файлу через сокет
    void receiveFileContents() {
        const int bufferSize = 1024;
        char buffer[bufferSize];
```

```
        ssize_t bytesReceived;

        std::ofstream outputFile("received_file.txt", std::ios::binary);

        while ((bytesReceived = recv(clientSocket, buffer, bufferSize, 0)) > 0) {
            outputFile.write(buffer, bytesReceived);
        }

        outputFile.close();
    }

    // Метод для закриття підключення
    void closeConnection() {
        closesocket(clientSocket);
    }

    // Метод для очищення ресурсів
    void cleanup() {
        if (listenerSocket != INVALID_SOCKET) {
            closesocket(listenerSocket);
        }
        if (clientSocket != INVALID_SOCKET) {
            closesocket(clientSocket);
        }
    }
};

int main(int argc, char* argv[]) {
    // Перевірка, чи передано достатньо аргументів
    if (argc < 4) {
```

```
        std::cerr << "Usage: " << argv[0] << " <ip_address> <port> <decryption_key>" << std::endl;
        return 1; // Повертаємо 1, означає помилку
    }

    // Отримуємо ip-адресу та порт з аргументів командного рядка
    std::string ipAddress = argv[1];
    int port = std::stoi(argv[2]);

    // Створюємо екземпляр FileReceiver
    FileReceiver receiver(ipAddress, port);

    // Отримуємо файл від відправника
    std::string receivedFilePath = receiver.receiveFile("received_encrypted_file.txt");

    if (!receivedFilePath.empty()) {
        std::cout << "File received successfully: " << receivedFilePath << std::endl;
        // Отримуємо ключ для розшифрування з аргументів командного рядка
        std::string decryptionKey = argv[3];
        // Створюємо екземпляр FileDecryptor
        FileDecryptor decryptor(decryptionKey);
        // Розшифровуємо отриманий файл
        std::string decryptedFilePath = decryptor.decryptFile(receivedFilePath, "decrypted_file.txt");

        if (!decryptedFilePath.empty()) {
            std::cout << "File decrypted successfully: " << decryptedFilePath << std::endl;
        }
        else {
```

```
            std::cerr << "Error decrypting the file." << std::endl;
            return 1; // Повертаємо 1, означає помилку
        }
    }
    else {
        std::cerr << "Error receiving the file." << std::endl;
        return 1; // Повертаємо 1, означає помилку

    return 0; // Повертаємо 0, означає успішне виконання
}
```