СХІДНОУКРАЇНСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ
ІМЕНІ ВОЛОДИМИРА ДАЛЯ

Навчально-науковий інститут (факультет) інформаційних технологій та електроніки

Кафедра інформаційних технологій та програмування
(повна назва кафедри)

ПОЯСНЮВАЛЬНА ЗАПИСКА
до кваліфікаційної випускної роботи

освітній ступінь бакалавр
(бакалавр, магістр)

спеціальність 121 «Інженерія програмного забезпечення»
(шифр і назва спеціальності)

на тему «Розробка клієнт-серверного застосунку для захищеного обміну повідомленнями»

Виконав: студент групи ІПЗ-21д _____ Є.О. Вєжлєв
(підпис) (ініціали і прізвище)

Керівник _____ В.О. Лифар
(підпис) (ініціали і прізвище)

Завідувач кафедри _____ О.І. Захожай
(підпис) (ініціали і прізвище)

Рецензент О.І. Захожай

Київ - 2025

СХІДНОУКРАЇНСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ
ІМЕНІ ВОЛОДИМИРА ДАЛЯ

Навчально-науковий інститут (факультет) інформаційних технологій та електроніки

Кафедра інформаційних технологій та програмування
(повна назва кафедри)

Освітній ступінь бакалавр
(бакалавр, магістр)

спеціальність 121 «Інженерія програмного забезпечення»
(шифр і назва спеціальності)

ЗАТВЕРДЖУЮ

Завідувач кафедри
_____ Захожай О.І.
"___" _____ 2025 року

З А В Д А Н Н Я
НА КВАЛІФІКАЦІЙНУ ВИПУСКНУ РОБОТУ СТУДЕНТУ

Вєжлєв Єгор Олександрович
(прізвище, ім'я, по батькові)

1. Тема роботи Розробка клієнт-серверного застосунку для захищеного обміну повідомленнями

Керівник роботи Лифар Володимир Олексійович, доц., д.т.н,
(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)
затверджений наказом університету від "27" травня 2025 року №67/15.15-С

2. Строк подання студентом роботи 20.06.2025

3. Вихідні дані до роботи Об'єктом даної роботи є процес створення застосунку для захищеного обміну повідомленнями

4.Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити) Вступ. Опис засобів захисту інформації, опис проблем безпеки інформації. Поставлення мети розробки застосунку для вирішення цих проблем. Опис засобів реалізації. Опис архітектури проекту. Опис процесу розробки застосунку. Висновки. Список використаних джерел

5. Перелік графічного матеріалу (з точним зазначенням обов'язкових креслеників)

6. Консультанти розділів роботи

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

7. Дата видачі завдання__01.04.2025р__

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів виконання кваліфікаційної випускної роботи	Строк виконання етапів	Примітка
1	Одержання завдання на виконання роботи	01.04.25	виконано
2	Складання плану і етапів виконання роботи	12.04.25	виконано
3	Дослідження літературних джерел, укладання розділу «Аналіз предметної області»	28.04.25	виконано
4	Аналіз способів досягнення мети. Вибір способу реалізації	07.05.25	виконано
5	Розробка та тестування проекту	14.06.25	виконано
6	Укладання пояснювальної записки	20.04.25	виконано
7	Надання пояснювальної записки на кафедру	20.04.25	виконано
8	Підготовка доповіді та презентації	20.04.25	виконано

Студент _____ ___Є.О. Вєжлєв____
(підпис) (ініціали і прізвище)

Керівник роботи _____ ____В.О. Лифар____
(підпис) (ініціали і прізвище)

РЕФЕРАТ

Робота містить 50 сторінок основного тексту, 13 графічних зображень, 2 таблиці, 23 сторінки додатків, 10 використаних джерел.

Дипломна робота присвячена розробці клієнт-серверного застосунку для захищеного обміну повідомленнями. У роботі розглянуто проблеми інформаційної безпеки в умовах сучасного цифрового середовища, проаналізовано моделі загроз, методи забезпечення конфіденційності даних, а також недоліки централізованих систем, що піддаються компрометації. Головною метою дослідження є створення програмного забезпечення, яке дозволяє користувачам і їх даним залишатися в безпеці навіть у разі повного контролю над сервером з боку зловмисника. Для цього реалізовано архітектуру, в якій усі криптографічні операції виконуються на стороні клієнта, а сервер відіграє роль нейтрального ретранслятора зашифрованих повідомлень.

У процесі виконання роботи використано сучасні програмні засоби: мову програмування Python, фреймворк FastAPI, бібліотеки cryptography, websockets, uvicorn. Реалізовано наскрізне шифрування, автентифікацію та перевірку цілісності повідомлень. Особливу увагу приділено захисту даних у сценаріях зберігання повідомлень для офлайн-користувачів.

Практичне значення роботи полягає у створенні прототипу системи безпечного обміну, який може бути адаптований для використання в державних структурах, юридичній практиці, військових підрозділах та інших сферах, де критично важливим є захист даних від перехоплення або втрати.

ЗМІСТ

ВСТУП

Актуальність. В умовах стрімкого розвитку інформаційних технологій та зростання обсягів переданої через мережу Інтернет інформації питання забезпечення конфіденційності, цілісності та доступності даних набуває особливої актуальності. Сучасні засоби комунікації, зокрема месенджери та системи обміну повідомленнями, широко використовуються у всіх сферах професійної і непрофесійної діяльності. Проте, разом із зручністю використання, виникають і нові загрози, пов'язані з несанкціонованим доступом до інформації, її перехопленням чи модифікацією.

Захищений обмін повідомленнями – це важлива задача у сфері інформаційної безпеки. Для її вирішення застосовуються різноманітні криптографічні методи, протоколи шифрування та автентифікації користувачів. Водночас, важливою вимогою до таких систем є не лише забезпечення високого рівня захисту, а й зручність використання, швидкодія та можливість роботи у розподіленому середовищі.

Об'єктом дослідження є методи та засоби забезпечення захищеного обміну інформацією.

Предметом дослідження – програмне забезпечення для захищеного обміну повідомленнями.

Метою роботи є розробка програмного забезпечення для захищеного обміну повідомленнями з повною безпекою користувачів на випадок компроментації серверних даних або навіть отримання повного доступу до серверу. У процесі виконання роботи було проведено аналіз існуючих рішень, визначено основні вимоги до системи, спроектовано архітектуру та реалізовано програмний продукт, що дозволяє здійснювати обмін зашифрованими повідомленнями у режимі реального часу, а також зберігати повідомлення для користувачів, які тимчасово відсутні в мережі.

Практична цінність роботи полягає у створенні програмного продукту, який може бути використаний або бути зразком для налаштування безпечної комунікації у сферах діяльності, де особливо важливим є захист інформації від несанкціонованого доступу чи перехоплення. Зокрема, така система може використовуватися органами державної влади для обміну службовою інформацією, що має обмежений доступ, а також для координації дій між різними підрозділами.

В умовах сучасних викликів особливої актуальності набуває використання захищених каналів зв'язку у воєнній сфері, де від безпеки переданої інформації може залежати успішність виконання завдань та збереження життя особового складу. Система може бути використана для оперативного обміну повідомленнями між військовими підрозділами, передачі наказів, розпоряджень чи координат, а також для організації таємних операцій, де критично важливо уникнути витоку інформації.

Окрім того, розробка є корисною для забезпечення конфіденційного листування між адвокатами та їх клієнтами, що дозволяє дотримуватися принципу адвокатської таємниці та захищати права громадян. Використання сучасних криптографічних методів гарантує, що зміст повідомлень залишатиметься недоступним для сторонніх осіб навіть у разі перехоплення трафіку.

Таким чином, впровадження подібних технологій сприяє підвищенню рівня інформаційної безпеки у державному управлінні, військовій справі, юридичній практиці, а також може бути корисним для проведення спеціальних чи таємних операцій, де захист інформації є питанням стратегічної важливості.

РОЗДІЛ 1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ

1.1 Поняття захисту інформації

Захист інформації — це сукупність організаційних, технічних та програмних заходів, спрямованих на запобігання несанкціонованому доступу, викривленню, підробці, втраті або витоку інформації. Його основною метою є збереження властивостей, що забезпечують надійність даних: конфіденційності, цілісності, доступності, автентичності та незаперечності.

Конфіденційність означає, що доступ до інформації мають лише ті особи або системи, яким він був наданий. Цей принцип покликаний запобігти витоку чутливих даних до сторонніх осіб, зловмисників або конкурентів.

Цілісність забезпечує незмінність інформації в процесі її зберігання або передавання. Це означає, що будь-яка спроба змінити дані без відповідного дозволу має бути виявлена.

Доступність гарантує, що користувачі, які мають право на доступ до певної інформації, можуть отримати її у потрібний момент без зайвих затримок або збоїв.

Автентичність підтверджує справжність джерела інформації. Це дозволяє переконатися, що дані надійшли саме від того відправника, якого очікували.

Незаперечність — це властивість, що не дозволяє відправнику заперечити факт надсилання інформації. Вона є особливо важливою для юридично значущих операцій.

У сфері інформаційної безпеки активно застосовуються криптографічні методи, які дають змогу забезпечити перелічені властивості. До них належать симетричне шифрування, коли однаковий ключ використовується як для шифрування, так і для розшифрування, та асиметричне шифрування, яке використовує пару відкритого та закритого ключів. Також важливу роль відіграють криптографічні хеш-функції, цифрові підписи, механізми автентифікації, обмеження доступу та контроль за цілісністю. При передачі

повідомлень через відкриті канали виникає загроза перехоплення, модифікації або підробки даних. Тому особливого значення набувають протоколи захищеної комунікації, такі як TLS/SSL, VPN, захищені тунелі, а також технології шифрування «end-to-end», які дозволяють зашифровувати інформацію на стороні відправника й розшифровувати лише на стороні отримувача.

Важливо відзначити, що ефективний захист інформації не обмежується лише технічними засобами. Він також передбачає дотримання політик безпеки, навчання персоналу, управління ризиками, а також постійний моніторинг стану систем.

1.2 Забезпечення конфіденційності в інформаційному просторі

Забезпечення конфіденційності є однією з основоположних властивостей інформаційної безпеки а також головною метою даного проекту. Вона означає захист інформації від доступу сторонніх осіб і гарантує, що тільки уповноважені суб'єкти мають право ознайомлюватися з вмістом даних. Порушення конфіденційності може мати катастрофічні наслідки — від особистих репутаційних втрат до національних загроз безпеці. З огляду на стрімкий розвиток цифрових технологій та зростання обсягів оброблюваної інформації, забезпечення конфіденційності стало критично важливим у всіх сферах інформаційного простору.

У сфері приватної комунікації конфіденційність передбачає захист особистих повідомлень, фотографій, відео, аудіо та інших даних, які пересилаються або зберігаються через електронні сервіси. Люди очікують, що їхнє спілкування буде недоступне стороннім — як кіберзлочинцям, так і третім сторонам, включно з провайдерами, рекламодавцями чи державними структурами. Тому наскрізне шифрування (end-to-end encryption) є одним із найактуальніших методів забезпечення конфіденційності, де навіть сам сервіс не має доступу до змісту повідомлень.

У корпоративному секторі конфіденційність має стратегічне значення. Конкурентна боротьба, інтелектуальна власність, фінансова звітність, комерційна таємниця — все це потребує надійного захисту. Витік внутрішньої інформації може призвести до економічних втрат, зниження довіри клієнтів або навіть банкрутства. Саме тому компанії використовують складні системи контролю доступу, корпоративні VPN, політики розмежування повноважень, а також внутрішній аудит дій працівників.

У державному секторі конфіденційність пов'язана з національною безпекою, захистом службової та державної таємниці, безпекою персональних даних громадян. Інформація про внутрішні політики, військові плани, дипломатичні переговори або бази даних населення повинна бути захищена від іноземних спецслужб і кіберзлочинців. Для цього держава впроваджує стандарти безпеки, шифрування, систему допуску до інформації та постійний кібермоніторинг.

У фінансовій сфері конфіденційність охоплює банківські рахунки, історію транзакцій, дані про кредити, інвестиції, страхування. Порушення конфіденційності тут часто веде до фінансового шахрайства, крадіжок коштів і персональних даних. Для захисту застосовуються протоколи шифрування, багатофакторна автентифікація, токенізація даних і системи поведінкової аналітики.

У сфері хмарних обчислень та зберігання даних проблема конфіденційності набуває нових форм. Дані користувачів часто зберігаються на серверах третіх осіб, які фізично знаходяться в іншій країні або навіть на іншому континенті. Це піднімає питання юрисдикції, контролю над доступом, прозорості обробки даних і довіри до провайдера. У відповідь на це з'являються концепції «zero trust», децентралізованого зберігання, а також технології шифрування даних ще до відправки у хмару.

У науці та освіті конфіденційність забезпечує захист персональних даних студентів, наукових розробок, а також результатів досліджень до моменту їх

публікації. Особливо це важливо в умовах академічної доброчесності та захисту авторського права.

Отже, конфіденційність є критичною складовою будь-якої інформаційної системи. Її порушення призводить до втрати довіри, фінансових збитків, репутаційних ризиків і навіть кримінальної відповідальності. Забезпечення конфіденційності потребує системного підходу: поєднання правових норм, організаційних політик, технічних засобів і сучасних криптографічних методів.

1.3 Методи забезпечення конфіденційності даних

Спершу розглянемо моделі доступу до інформації, які використовуються в сучасних архітектурах інформаційних систем.

Таблиця 1.3.1 – Моделі доступу до інформації

Назва моделі	Головні принципи	Переваги	Недоліки
Модель мандатного доступу (Mandatory Access Control, MAC)	Базується на жорстко визначених правилах доступу, які встановлюються адміністраторами системи. Об'єкти та суб'єкти мають певний рівень секретності (наприклад: "таємно", "секретно", "конфіденційно"), і система самостійно вирішує, хто має право на доступ.	-Високий рівень безпеки; -Неможливість користувачеві самостійно змінити політику доступу; -Підходить для урядових і військових систем.	-Низька гнучкість; -Складність адміністрування в динамічних середовищах; -Повільна адаптація до змін бізнес-процесів.
Модель дискреційного доступу (Discretionary Access Control, DAC)	Власник ресурсу (наприклад, файлу чи бази даних) сам визначає, хто і що може з ним робити (читати, змінювати, видаляти). Це характерно для більшості сучасних операційних систем.	-Гнучкість; -Простота у впровадженні; -Зручна для невеликих або внутрішніх систем.	-Вища вірогідність помилок користувачів; -Менш захищена від внутрішніх загроз; -Важче контролювати загальний рівень безпеки.
Рольова модель	Доступ надається не	-Масштабованість;	-Потребує

доступу (Role-Based Access Control, RBAC)	конкретному користувачу, а ролі, яку він виконує (наприклад: бухгалтер, адміністратор, аналітик). Це дозволяє централізовано керувати правами доступу в масштабних організаціях.	-Зручне адміністрування; -Добре підходить для великих підприємств.	ретельного проектування ролей; -Неналежне оновлення ролей призводить до надлишкових доступів; -Складність у динамічних або нестандартних сценаріях.
Атрибутивна модель доступу (Attribute-Based Access Control, ABAC)	Ця модель приймає рішення про доступ на основі множини атрибутів: хто запитує доступ, до чого, за яких умов, у який час тощо. Наприклад, система може надати доступ до документа лише в робочий час з IP-адреси корпоративної мережі.	-Висока гнучкість та адаптивність; -Підходить для хмарних та мобільних рішень; -Підтримка політик з великою кількістю умов.	-Висока складність реалізації; -Потребує детального аналізу атрибутів і політик; -Може мати знижену продуктивність при великій кількості перевірок.

Ці моделі створенні для контролю над зберігаємою інформацією, але всі вони об’єднані тим, що при отриманні контролю над сервером, дані можуть стати доступними зловмисникові, ворогу, державним органам чи навіть перейти у відкритий доступ.

Одним із найбільш резонансних випадків витоку даних в історії цифрової безпеки став злам серверів компанії Yahoo, який охоплював період 2013–2014 років. Цей інцидент вважається найбільшим за кількістю скомпрометованих акаунтів — за офіційною інформацією постраждало близько трьох мільярдів облікових записів, що фактично охоплювало кожного користувача сервісу на той момент. Перші публічні повідомлення про злам з’явилися у 2016 році, коли Yahoo визнала факт компрометації близько 500 мільйонів акаунтів, що стало на

той час одним із найбільших витоків в історії. Проте згодом, під час внутрішнього розслідування, стало відомо, що масштаб проблеми значно перевищує початкову оцінку. У 2017 році компанія офіційно заявила, що в результаті атаки у 2013 році були скомпрометовані всі облікові записи Yahoo — тобто понад 3 мільярди.

Зловмисники отримали доступ до імен користувачів, адрес електронної пошти, телефонних номерів, дати народження, хешованих паролів і, у деяких випадках, до резервних запитань та відповідей (навіть у незашифрованому вигляді). Найбільшою вразливістю виявилось те, що Yahoo зберігала паролі у вигляді хешів, згенерованих за допомогою алгоритму MD5, який вже на той час вважався ненадійним і застарілим. MD5 надто швидкий, що робить його вразливим до атак методом перебору (brute-force), особливо при наявності потужного обладнання. Крім того, Yahoo не застосовувала додаткове сольове хешування для кожного паролю, що лише полегшувало масове розкриття даних.

Атака залишалася непоміченою протягом тривалого часу. Деякі сліди свідчили про можливу участь хакерів, пов’язаних з урядовими структурами однієї з країн, хоча довести це офіційно так і не вдалося. Критичною виявилася не лише сама атака, а й реакція компанії на інцидент: Yahoo значно затримала публічне розголошення інформації, а її внутрішня безпека та протоколи реагування були визнані вкрай неефективними. Витік викликав гнів користувачів, численні колективні позови, а також масштабне розслідування в США. Цей інцидент відбувся якраз у той час, коли Verizon планувала придбати Yahoo. Внаслідок виявлення зламу, угоду переглянули, і вартість компанії було знижено на $350 мільйонів. Зрештою Yahoo втратила не лише гроші, а й залишки довіри з боку громадськості. Згодом компанія була інтегрована в структуру Verizon під брендом Oath.

Це стало ілюстрацією того, як недбале ставлення до захисту персональних даних, використання застарілих криптографічних механізмів і

слабкий контроль за внутрішньою безпекою можуть мати катастрофічні наслідки. Цей випадок змусив переглянути підходи до зберігання паролів, застосування хеш-функцій, а також нормативні вимоги до прозорості повідомлень про витоки. Він також став одним із ключових факторів, що пришвидшили впровадження більш жорстких законів про захист персональних даних, зокрема в Європі — таких як GDPR.

Отже, навіть найбільші технологічні компанії не застраховані від масштабних компрометацій, якщо в основі їхніх систем лежать застарілі практики безпеки.

Але нажаль навіть використання всіх новітніх методів захисту інформації не дає гарантій безбеки.

Один із найвідоміших і найпоказовіших випадків, коли навіть наявність сучасних методів захисту не змогла запобігти катастрофічному витоку даних, стався з американською компанією Equifax у 2017 році. Цей інцидент увійшов в історію як один з найсерйозніших зламів у фінансовому секторі, адже зловмисники отримали доступ до надзвичайно чутливих персональних даних понад 147 мільйонів громадян США.

Equifax — це одна з трьох найбільших кредитних агенцій у США, яка зберігає гігантські обсяги фінансової інформації про мільйони громадян: номери соціального страхування, дати народження, адреси, дані водійських прав, кредитну історію.

На момент зламу Equifax дійсно використовувала **сучасні методи захисту інформації**, зокрема:

- шифрування баз даних;
- застосування міжмережевих екранів (фаєрволів);
- систем виявлення вторгнень (IDS/IPS);
- контроль доступу на основі ролей;

- обмеження доступу до внутрішніх систем через VPN;
- резервне копіювання та логування;
- політику складних паролів і багатофакторну автентифікацію в окремих сегментах мережі.

Але не можливо бути готовим до всього. Інцидент стався через критичну уразливість CVE-2017-5638 у популярному Java-фреймворку Apache Struts. Ця вразливість дозволяла віддалено виконувати код на сервері через спеціальні HTTP-запити. Злом почався приблизно 12–13 травня 2017 року і тривав безперервно до виявлення інциденту 29 липня 2017 року. За цей час зловмисники отримали доступ до систем, що дозволило їм діяти як внутрішні користувачі: вони ексфільтрували дані про 147,9 мільйона громадян США, а також понад 15 мільйонів британців та близько 19 000 канадців.

Хоча зловмисники фактично діяли від імені внутрішніх користувачів, використовуючи повноваження для розшифровки даних, самі бази даних формально були захищені шифруванням. Однак наявність шифрування не рятувала, коли вразливі компоненти та ключі були доступні або внутрішньо використовувались користувачами з правами доступу.

Цей інцидент чітко ілюструє: навіть найсучасніші методи захисту не захищають від знаходження вразливостей. Часто виявляється, що чим більш складна система, тим більш ймовірне знаходження вразливостей.

1.4 Методи захисту інформації з клієнтським зберіганням ключів

Отже, у традиційних інформаційних системах сервер відіграє активну роль у зберіганні, обробці та шифруванні даних. Проте така централізована модель має критичну вразливість – компрометація сервера автоматично наражає на небезпеку всю систему, оскільки на ньому можуть зберігатися ключі, паролі або незашифровані дані. Тому у випадках, коли можна обійтися без розкриття даних серверу з’явилися підходи, в яких сервер не зберігає критично важливої інформації, зокрема криптографічних ключів. Вся

відповідальність за конфіденційність даних переноситься на клієнтську сторону. Така модель має низку важливих технічних реалізацій.

Таблиця 1.4.1 – Моделі з клієнтським зберіганням ключів

Назва моделі	Головні принципи	Переваги	Недоліки
End-to-End Encryption (E2EE)	-Симетричні ключі шифрування генеруються на пристрої користувача. -Асиметричні пари ключів (наприклад, Ed25519) використовуються для аутентифікації та розповсюдження ключів. -Сервер використовується лише як "труба" для обміну шифротекстом, без можливості його прочитати.	-Висока конфіденційність — навіть повний контроль над сервером не дає змоги розшифрувати вміст. -Незалежність від інфраструктури провайдера. -Відсутність централізованої точки відмови.	-Втрата ключа на клієнтському боці = втрата доступу до даних. -Ускладнене резервне копіювання.
Zero-Knowledge Encryption / Zero-Knowledge Proof Services	-Дані шифруються перед передачею на сервер. -Пароль або ключ ніколи не покидає пристрій користувача. -Сервер не має жодних знань про вміст, а лише зберігає шифротексти.	-Справжня конфіденційність у хмарі. -Знижений ризик витоку даних навіть при зламі сервера. -Відповідність політикам приватності (зокрема GDPR).	-Складність у впровадженні колективної роботи з даними. -Неможливість "відновлення доступу" без знання паролю. -Ускладнена інтеграція з іншими системами.
Поширення публічних ключів (Public Key Distribution without Server Trust)	-Кожен користувач має свою пару відкритого/закритого ключів. -Сервер може слугувати каталогом публічних ключів або ретранслятором	-Максимальна децентралізація. -Безпека за рахунок криптографії, а не довіри до сервера. -Можливість повної	-Складність для звичайного користувача. -Проблеми з управлінням ключами

	повідомлень. -Ідентифікація часто ґрунтується на підписах або fingerprint-ах ключів.	автономії та peer-to-peer взаємодії.	(відкликання, обмін). -Відсутність звичних механізмів відновлення доступу.

Вебтехнології також поступово пристосовуються до такого підходу. Сучасні браузери дозволяють використовувати Web Crypto API — набір функцій, які забезпечують повноцінне шифрування та цифрові підписи безпосередньо в середовищі браузера. Це дає змогу реалізовувати клієнтське шифрування навіть у вебдодатках, де раніше все передавалося у відкритому вигляді. У таких рішеннях сервер працює виключно як засіб пересилання вже зашифрованої інформації, не маючи жодної змоги її прочитати або змінити.

Хоча моделі захисту інформації, в яких сервер не має доступу до ключів шифрування, значно підвищують рівень конфіденційності користувачів, вони все ж не гарантують абсолютної безпеки. Навіть у таких системах існують певні загрози, які можуть виникнути у випадку повного контролю над серверною інфраструктурою з боку зловмисника чи зловживань адміністраторами.

Передусім, навіть якщо вміст повідомлень чи файлів надійно зашифрований, метадані у більшості випадків залишаються доступними для сервера. Це може включати інформацію про те, хто з ким комунікує, коли, з якою інтенсивністю, з яких IP-адрес, і навіть обсяги переданих даних. Зібравши ці дані протягом тривалого часу, зловмисник або державна структура може побудувати точний профіль користувача, виявити його соціальні зв’язки, часові звички, місце перебування та інші непрямі ознаки. Такий підхід називається аналізом метаданих, і навіть повністю зашифровані повідомлення можуть видавати вразливу інформацію саме через такий побічний канал.

Крім того, у випадку злому або компрометації сервера зловмисник отримує змогу впроваджувати атаки типу «людина посередині» (man-in-the-middle). Якщо система не забезпечує криптографічного підтвердження автентичності публічних ключів, то атакуючий сервер може підмінити ключі

однієї сторони іншим, підставним, і таким чином отримати можливість читати повідомлення до їх шифрування або після дешифрування. Це особливо критично у системах, де користувачі не перевіряють відбитки ключів вручну, а довіряють серверу як каталогу ключів. Навіть у системах з наскрізним шифруванням можливість атакувати перше з'єднання або запускати "silent key re-substitution" існує, якщо архітектура не передбачає захист від цього (наприклад, через механізм перевірки ідентичності або QR-коди між пристроями, як у Signal).

Ще однією потенційною загрозою є розповсюдження шкідливого коду з боку скомпрометованого сервера. У випадку вебдодатків, навіть якщо все шифрування виконується на клієнтському боці через Web Crypto API, зловмисник, який контролює сервер, може модифікувати JavaScript-код і, наприклад, додати функцію викрадення введених користувачем паролів або пересилання ключів на інший сервер. У цьому сенсі браузерна модель особливо вразлива: користувач отримує код щоразу знову, і йому важко відрізнити безпечну версію від скомпрометованої. Тому навіть за правильно реалізованого клієнтського шифрування безпека залежить від того, наскільки користувач довіряє джерелу коду.

Отже, навіть у системах, де шифрування повністю віднесене на клієнтську сторону, повна компрометація сервера все одно становить небезпеку. Хоча вміст повідомлень у такому випадку зазвичай залишається захищеним, загроза витоку метаданих, атак на перше з'єднання, підміни ключів або впровадження шкідливого коду з боку сервера залишається реальною. Тому дійсно безпечна система — це не лише питання правильного використання криптографії, а й питання загального дизайну, перевірки ідентичності, відкритості коду та рівня прозорості між клієнтом і сервером.

Тому ця робота присвячена моделюванню клієнт-серверного застосунку для обміну повідомленнями з захистом від компрометації даних та мінімізацією наслідків отримання контролю над сервером.

1.5 Висновки до розділу 1

У розділі було проведено детальний аналіз предметної області, присвяченої захисту інформації. Розглянуто фундаментальні поняття інформаційної безпеки. Було наголошено, що забезпечення конфіденційності є критично важливим у всіх сферах інформаційного простору: від приватної комунікації до корпоративного, державного та фінансового секторів, а також у хмарних обчисленнях та науці.

Проаналізовано різні моделі доступу до інформації. Хоча сучасні моделі ефективні для управління доступом до даних на рівні сервера, було доведено їхню вразливість у випадку повної компрометації серверної інфраструктури, що може призвести до розкриття конфіденційної інформації. Приклади масштабних витоків даних, таких як інциденти з Yahoo та Equifax, яскраво продемонстрували, що навіть використання сучасних методів захисту не гарантує абсолютної безпеки, якщо існують вразливості в системі або її компонентах.

Особливу увагу приділено методам захисту інформації з клієнтським зберіганням ключів. Проте, було підкреслено, що навіть такі системи мають ймовірність певних видів атак і завдання шкоди користувачам.

Таким чином, результати аналізу підкреслюють, що розробка програмного забезпечення для захищеного обміну повідомленнями вимагає не лише використання надійних криптографічних алгоритмів, але й комплексного підходу до дизайну системи, що враховує мінімізацію довіри до сервера. Це обґрунтовує необхідність моделювання клієнт-серверного застосунку, який забезпечуватиме високий рівень конфіденційності даних та стійкість до компрометації серверної інфраструктури, що є метою дипломної роботи.

РОЗДІЛ 2 ВИБІР ЗАСОБІВ ДЛЯ РЕАЛІЗАЦІЇ

2.1 Мова програмування Python

Python є однією з найпопулярніших мов програмування завдяки своїй простоті, універсальності та потужному набору бібліотек. Ця мова активно використовується в різних сферах, таких як розробка веб-застосунків, наукові обчислення, машинне навчання, автоматизація та, звісно, в інформаційній безпеці. Python був створений Гвідо ван Россумом в 1980-х роках і офіційно представлений у 1991 році. Мова орієнтована на простоту використання, що робить її доступною як для новачків, так і для досвідчених розробників. Однією з головних особливостей Python є його синтаксис, який дозволяє писати зрозумілий та чистий код. Python підтримує об'єктно-орієнтоване програмування, має велику стандартну бібліотеку, що включає модулі для роботи з мережами, базами даних, інтерфейсами користувача та іншими аспектами програмування. Оскільки Python є інтерпретованою мовою, розробники можуть значно спростити процес тестування та розробки завдяки відсутності необхідності компілювати код.

Для розробки застосунку для захищеного обміну повідомленнями Python є оптимальним вибором з кількох причин. По-перше, Python дозволяє швидко реалізувати необхідні криптографічні функції завдяки наявності потужних бібліотек, таких як PyCryptodome, cryptography і PyNaCl, що забезпечують надійне та зручне використання криптографічних алгоритмів. Це дозволяє швидко впроваджувати шифрування, цифрові підписи, хешування та автентифікацію користувачів, що є важливими аспектами при створенні безпечного обміну повідомленнями. Крім того, Python підтримує роботу з мережевими протоколами, зокрема з протоколами захищеного обміну, такими як TLS/SSL, що дає змогу інтегрувати необхідні технології шифрування для передачі даних. Python також забезпечує масштабованість, що важливо для створення розподілених систем, і має широкі можливості для інтеграції з

іншими мовами та платформами, що дозволяє збудувати надійну систему обміну повідомленнями.

До переваг Python відноситься також наявність великої спільноти розробників, яка активно працює над поліпшенням безпеки бібліотек. Це забезпечує стабільність і надійність системи, що критично важливо для створення застосунку для захищеного обміну повідомленнями, де безпека є пріоритетом. Незважаючи на численні переваги, Python має деякі недоліки, такі як порівняно нижча швидкодія в порівнянні з компільованими мовами, такими як C або C++. Оскільки Python є інтерпретованою мовою, це може призвести до зниження продуктивності при обробці великих обсягів даних, однак це не є критичним для більшості задач, які не вимагають надвисокої швидкості обробки. Також Python має обмеження при використанні багатопоточності через глобальну блокувальну перемінну (GIL), хоча це можна обійти за допомогою багатопроцесорних рішень або асинхронного програмування.

Вибір Python для цієї дипломної роботи обумовлений його здатністю швидко реалізувати необхідні криптографічні алгоритми та безпеку передачі даних, можливістю інтеграції з іншими системами та зручністю розробки. Python є чудовим інструментом для розробки прототипів та систем з високими вимогами до безпеки, тому він був обраний для розробки застосунку для захищеного обміну повідомленнями.

2.2 Середовище розробки PyCharm

PyCharm — це потужне інтегроване середовище розробки (IDE), яке було створене компанією JetBrains спеціально для програмування на Python. Це одне з найбільш популярних середовищ для розробки програм на Python завдяки своїй зручності, багатофункціональності та потужним можливостям, які підтримують усі етапи розробки програмного забезпечення — від написання коду до тестування та налагодження.

PyCharm має інтуїтивно зрозумілий інтерфейс, який дозволяє ефективно працювати навіть з великими проектами. Однією з основних переваг цього

середовища є потужна система автодоповнення коду, що значно прискорює процес написання програмного коду та знижує ймовірність помилок. PyCharm також підтримує інтеграцію з системами контролю версій, такими як Git, що робить командну роботу і керування версіями коду значно зручнішими. IDE має вбудовані інструменти для налагодження програм, що дозволяє швидко знаходити і виправляти помилки в коді. Особливо корисною є можливість відлагодження коду в реальному часі, що дає змогу ефективно контролювати хід виконання програм і перевіряти значення змінних у кожній точці виконання. Для розробників, які працюють з криптографічними алгоритмами та іншими складними системами, це важлива особливість, адже дозволяє детально аналізувати поведінку програм. PyCharm також підтримує різноманітні плагіни, що дозволяє розширювати функціональність середовища в залежності від потреб проекту. Зокрема, для розробки захищеного обміну повідомленнями можна налаштувати додаткові плагіни для роботи з криптографічними бібліотеками або з іншими інструментами безпеки. Ще однією важливою перевагою є наявність підтримки для роботи з віртуальними середовищами Python, що дозволяє ізолювати залежності проекту від системних бібліотек, забезпечуючи стабільну роботу програми. Це особливо важливо в проектах, що використовують специфічні бібліотеки для шифрування та безпеки, де можуть виникати конфлікти між різними версіями бібліотек. PyCharm дозволяє зручно створювати, налаштовувати і використовувати віртуальні середовища, що робить роботу з проектом більш гнучкою та безпечною.

Одним з основних недоліків PyCharm є його висока вимогливість до ресурсів комп'ютера. Для роботи з великими проектами або для одночасної роботи в кількох вікнах IDE може знадобитися значна кількість оперативної пам'яті та процесорних ресурсів. Це може стати проблемою при роботі на старих комп'ютерах або в разі обмежених ресурсів.

Проте, в рамках даного проекту дефіцит ресурсів не представояє загрози, тому PyCharm є ідеальним вибором для розробки предмету дослідження.

2.3 Обрані бібліотеки

Для реалізації програмного забезпечення, орієнтованого на захищений обмін повідомленнями в архітектурі клієнт–сервер, було обрано низку бібліотек Python, кожна з яких виконує специфічну роль у функціонуванні системи.

Бібліотека **cryptography** є сучасним і надійним інструментом для реалізації широкого спектра криптографічних алгоритмів. Вона підтримує як низькорівневі примітиви (симетричне шифрування, хеш-функції, HMAC, KDF), так і високорівневі механізми, включаючи цифрові підписи та асиметричні ключі. У даному проєкті бібліотека cryptography обрана для реалізації наскрізного шифрування повідомлень між клієнтами, а також для забезпечення їхньої цілісності та автентичності.

Бібліотека **FastAPI** обрана для реалізації серверної частини застосунку. FastAPI — це сучасний веб-фреймворк для мови програмування Python, призначений для створення високопродуктивних вебзастосунків, API-сервісів та серверних частин систем з підтримкою асинхронного програмування. Його було створено Себастьяном Раміресом і вперше представлено у 2018 році. З того часу він здобув широку популярність завдяки поєднанню простоти, зручності та високої швидкодії. FastAPI базується на стандартах OpenAPI (раніше Swagger) та JSON Schema, що дозволяє автоматично генерувати інтерактивну документацію для API, спрощує тестування та взаємодію з фронтендом або іншими клієнтами.

З переваг FastAPI можна виділити асинхронну природу. Він повністю сумісний з async/await синтаксисом Python, що дає змогу обробляти велику кількість одночасних підключень без блокування потоку виконання. Завдяки цьому фреймворк особливо ефективний для створення сервісів реального часу, таких як чати, системи моніторингу, API для мікросервісів, а також для роботи з WebSocket-з'єднаннями, які активно використовуються в даному проєкті. FastAPI також тісно інтегрований із системою типізації Python — типи параметрів, повертаних значень і змінних не лише слугують для перевірки

коректності коду, але й автоматично використовуються для валідації вхідних даних та формування OpenAPI-документації. Це суттєво зменшує обсяг шаблонного коду та підвищує надійність проєкту. Розробник отримує миттєвий зворотний зв'язок про помилки параметрів, що особливо корисно при швидкій розробці прототипів або масштабних API-сервісів.

Оскільки FastAPI є фреймворком, а не сервером, для фактичного обслуговування HTTP/ASGI-запитів йому потрібен сервер додатків.

В даному проєкті для виконання цієї задачі буде використовуватися бібліотека **unicorn**. Uvicorn — це легкий, асинхронний сервер, написаний на Cython та Python, що реалізує специфікацію ASGI (Asynchronous Server Gateway Interface). ASGI — це асинхронний аналог класичного WSGI, який використовується для обробки підключень, фонових задач, потокових відповідей та іншої асинхронної взаємодії. Uvicorn ідеально підходить для запуску FastAPI-застосунків завдяки малій затримці, високій продуктивності та повній сумісності з асинхронною екосистемою Python. Uvicorn також підтримує такі важливі функції, як автоматичне перезавантаження сервера при зміні коду (що буде корисно під час розробки та тестування проекту), підтримка TLS/SSL-сертифікатів для https:// або wss://, обробка великих обсягів трафіку через підтримку подійного циклу uvloop, і можливість інтеграції з більш складними системами через запуск у комбінації з Gunicorn або іншим ASGI/WSGI сервером.

Клієнтська частина програми буде використовувати бібліотеку **websockets**. Протокол WebSocket є мережевим протоколом, який забезпечує повнодуплексне з'єднання між клієнтом і сервером поверх одного TCP-з'єднання. Його було стандартизовано у 2011 році як частину специфікації HTML5 для того, щоб забезпечити ефективну та постійну комунікацію між браузером або будь-яким іншим клієнтом і сервером. На відміну від традиційної моделі HTTP-запит–відповідь, WebSocket дозволяє обом сторонам ініціювати обмін повідомленнями в будь-який момент часу без необхідності

повторного відкриття з’єднання. Тому WebSocket став незамінним у системах реального часу: чатах, онлайн-іграх, біржових платформах, системах моніторингу тощо. Бібліотека підтримує інтеграцію з asyncio, що дає змогу паралельно обробляти вхідні повідомлення, відстежувати статус доставки, взаємодіяти з чергою команді, водночас, не блокувати інтерфейс користувача.

Протокол WebSocket починається з "handshake" — початкового обміну, що виглядає як звичайний HTTP-запит з особливим заголовком Upgrade: websocket. Після цього HTTP-з’єднання "перетворюється" на WebSocket-з’єднання, і подальший обмін відбувається за окремим бінарним протоколом. Існують дві версії WebSocket-з’єднань: ws:// — незахищене з’єднання, і wss:// — захищене за допомогою TLS (аналог HTTPS). У випадку wss:// весь трафік між клієнтом і сервером шифрується стандартними засобами TLS, що захищає його від перехоплення, модифікації або підміни.

Але задля реалізації даного проекту достатньо використовувти ws:// з’єднання. Використання ws:// замість wss:// вважається небезпечним у більшості випадків, оскільки переданий трафік доступний для прослуховування будь-якому вузлу в мережі між клієнтом і сервером. Це може включати Wi-Fi роутери, інтернет-провайдерів, точки доступу в публічних місцях, або зловмисників у локальній мережі. Проте в окремих специфічних випадках, коли дані вже повністю зашифровані перед відправкою, як як в цій роботі, використання ws:// не становить серйозну загрозу для конфіденційності повідомлень. Якщо шифрування виконується ще на стороні клієнта за допомогою сучасних алгоритмів, а сервер лише передає зашифровані пакети, то навіть при повному доступі до мережевого трафіку зловмисник отримає лише шифротекст, який не має сенсу без ключа. Це значною мірою знижує ризики, пов’язані з використанням відкритого з’єднання.

Отже, обрані бібліотеки формують узгоджене середовище, де FastAPI та uvicorn забезпечують серверну інфраструктуру, websockets — стабільне з’єднання в режимі реального часу, а cryptography — весь обчислювальний

апарат для реалізації криптографічної моделі, яка гарантує конфіденційність, цілісність і автентичність даних без необхідності довіри до сервера. Це дозволить реалізувати систему обміну повідомленнями, у якій користувачі не залежать від централізованого сховища паролів або ключів, а вся безпека базується на криптографії, а не на надійності інфраструктури.

2.4 Висновки до розділу 2

У цьому розділі було обґрунтовано вибір основних засобів для розробки системи захищеного обміну повідомленнями, зокрема мови програмування Python, середовища розробки PyCharm і ключових бібліотек, що забезпечують основні функції застосунку.

Python було обрано завдяки його зручності, універсальності та наявності потужних бібліотек для роботи з криптографією та мережевими протоколами. Мова підтримує всі необхідні механізми для забезпечення безпеки, зокрема шифрування, цифрові підписи та хешування, що є критичними для побудови захищеної системи обміну повідомленнями. Хоча Python має деякі обмеження, зокрема низьку швидкодію порівняно з компільованими мовами, ці недоліки не становлять загрози для реалізації цього проекту, оскільки швидкість обробки не є критичною для цього застосунку.

Середовище розробки PyCharm було обрано завдяки своєму інтуїтивно зрозумілому інтерфейсу, потужним інструментам для налагодження коду, підтримці інтеграції з системами контролю версій і можливості працювати з віртуальними середовищами Python. Ці функції особливо важливі для роботи над проєктом, де безпека і стабільність є пріоритетами, а також для інтеграції з криптографічними бібліотеками та іншими інструментами безпеки.

Вибір бібліотек для реалізації функціоналу підтвердив свою доцільність. Бібліотека cryptography забезпечує надійне шифрування і автентифікацію, що є основою для захищеного обміну повідомленнями. FastAPI і uvicorn разом створюють потужну серверну інфраструктуру, що підтримує асинхронне програмування та високу продуктивність, що дозволяє реалізувати систему

реального часу для передачі повідомлень. Websockets забезпечує стабільне двостороннє з'єднання для обміну даними без необхідності повторного встановлення з'єднання, що є важливим для забезпечення безперервності та безпеки зв'язку.

Обрані засоби дають змогу масштабувати систему та інтегрувати її з іншими технологіями в майбутньому, що робить проєкт перспективним для подальшого розвитку.

РОЗДІЛ 3 РЕАЛІЗАЦІЯ ПРОЕКТУ

3.1 Побудова архітектури проекту

Проект для захищеного обміну повідомленнями базується на архітектурі клієнт-сервер, де сервер виконує роль лише посередника для встановлення з'єднання та передачі зашифрованих даних між користувачами. Ключовим принципом є забезпечення повної конфіденційності і цілісності повідомлень, незалежно від того, чи зловмисник отримає доступ до серверу. Уся криптографічна обробка даних відбувається на стороні клієнтів, що мінімізує ризики від компрометації серверної інфраструктури.

Процес підключення користувача до сервера в даній архітектурі здійснюється через WebSocket-з'єднання, яке є основним протоколом для двостороннього обміну повідомленнями в реальному часі. Кожен користувач підключається до сервера, використовуючи свій ідентифікатор (client_id). Варто зазначити, що задля мінімізації корисних даних на сервері система авторизації буде відсутньо, тому не буде реалізовано системи унікальних айді. В такій моделі авторизація виконується на рівні встановлення каналу шифрування-дешифрування на базі спільного паролю. Тому буде реалізована система з вибором будь-якого айді без перевірки чи зайнятий він. Система буде розсилати повідомлення всім користувачам з даним айді, але розшифрувати повідомлення зможуть лише користувачі зі спільним домовдленим паролем. Така модель забезпечує відсутність критичних даних на сервері і зберігає можливість авторизувати канал зв’язку без необхідності в унікальних

аккаутнах. Після встановлення з'єднання сервер приймає підключення через протокол ws://, що дозволяє безпечно передавати лише зашифровані дані, залишаючи сервер без доступу до самого змісту повідомлень.

Після підключення користувач отримує можливість обмінюватися повідомленнями з іншими учасниками. Кожен користувач має можливість ініціювати з'єднання через WebSocket, зашифровувати повідомлення перед передачею та здійснювати перевірку автентичності отриманих повідомлень, використовуючи криптографічні механізми, як-от цифрові підписи, HMAC, а також симетричне шифрування через AES-GCM.

Задля зручності буде створено альтернативний метод надсилання повідомлення зі збереженням на сервері до моменту підключення отримувача. В такому випадку на сервері будуть зберігатися лише зашифровані пакети повідомлень для тих користувачів, які на даний момент офлайн. Сервер не має доступу до шифрів і ключів, тому навіть у разі компрометації серверної інфраструктури зловмисники не зможуть отримати жодної корисної інформації.

Надсилання повідомлення між користувачами відбувається через канал WebSocket. Кожен клієнт ініціює процес відправки, шифруючи дані на своєму боці, перед тим як передати їх на сервер. Повідомлення містить наступні компоненти:

- Контент повідомлення — зашифрований текст повідомлення;
- Цифровий підпис — для перевірки цілісності та автентичності повідомлення;
- HMAC — для додаткової верифікації відсутності змін у даних під час передачі;
- Nonce — унікальний ідентифікатор для кожного повідомлення для запобігання повторного використання.

Після того, як сервер отримує зашифроване повідомлення, він просто передає його отримувачу, не маючи деталей шифрування. Кожен отримувач на своїй стороні розшифровує повідомлення, використовуючи відповідні

криптографічні ключі, перевіряючи цифровий підпис і HMAC, щоб впевнитися в цілісності даних.

Завдяки використанню симетричного шифрування та криптографічних алгоритмів перевірки підписів, передача повідомлень є надійною, і навіть в разі перехоплення трафіку на сервері або в мережі, зловмисники не зможуть отримати доступ до змісту повідомлень без ключів дешифрування.

3.2 Розробка серверної частини

Процес створення серверної частини проєкту базувався на принципі мінімального довірчого навантаження на сервер, у якого не повинно бути жодного доступу до незашифрованих даних. Сервер виконує функції релею повідомлень між клієнтами, не зберігаючи жодної конфіденційної інформації, ключів чи історії переписки. Він виступає як транспортний вузол, який приймає зашифровані дані від одного клієнта та пересилає їх іншому.

Для реалізації цієї логіки було обрано вебфреймворк FastAPI, що підтримує асинхронне програмування, є сумісним із специфікацією ASGI (Asynchronous Server Gateway Interface) та забезпечує високу продуктивність при обробці великої кількості одночасних WebSocket-з'єднань. Весь функціонал серверу буде написано в файлі server.py.

Спочатку імпортуємо основні компоненти, які забезпечують маршрутизацію повідомлень між клієнтами:

```
from fastapi import FastAPI, WebSocket, WebSocketDisconnect
from typing import Dict, List
import json
from datetime import datetime
import logging
import uvicorn
```

FastAPI використовується для створення веб-сервісу, WebSocket для двостороннього зв'язку з клієнтами, а logging — для ведення журналу операцій, що є критично важливим для діагностики та моніторингу. Тому наступним кроком одразу налаштуємо систему логування.

```
# Налаштування логування
logging.basicConfig(
    level=logging.INFO,
    format='%(asctime)s - %(levelname)s - %(message)s',
    datefmt='%Y-%m-%d %H:%M:%S'
)
logger = logging.getLogger(__name__)
```

Рис. 3.2.1 – Налаштування логування

Таке налаштування дозволяє зберігати логи у форматі, зручному для аналізу, з вказанням часу події та рівня її важливості.

Основним компонентом серверу буде ConnectionManager. Він відповідає за управління підключеннями клієнтів, а також для збереження повідомлень, якщо користувач хоче залишити повідомлення співбесіднику, який знаходиться в офлайні.

```
async def connect(self, websocket: WebSocket, client_id: str) -> None:
    """Додати нове WebSocket-підключення для client_id"""
    await websocket.accept()
    if client_id not in self.active_connections:
        self.active_connections[client_id] = []
    self.active_connections[client_id].append(websocket)
    logger.info(f"Нове підключення для ID {client_id}. Всього підключень "
                f"для цього ID: {len(self.active_connections[client_id])}")
```

Рис. 3.2.2 – Метод connect

Метод connect додає нове підключення для кожного користувача, використовуючи ідентифікатор клієнта (client_id). При цьому, як було зазначено раніше, зберігається можливість використання додаткових

повторюваних ідентифікаторів. Тому в системі без аутентифікації на сервері немає загрози, що чийсь айді стане зайнятим. А повідомлення все одно будуть в безпеці без знання паролю. Після прийому WebSocket-з'єднання від клієнта сервер відправляє йому всі збережені повідомлення.

Метод store_message зберігає повідомлення призначені відправником для зберігання, а send_stored_messages відправляє користувачу всі повідомлення, що були збережені для нього в офлайні:

```python
async def store_message(self, message: dict, recipient_id: str) -> None:  1 usage
    """Зберегти повідомлення для офлайн користувача"""
    if recipient_id not in self.stored_messages:
        self.stored_messages[recipient_id] = []
    self.stored_messages[recipient_id].append(message)
    logger.info(f"Повідомлення збережено для {recipient_id}")

async def send_stored_messages(self, client_id: str, websocket: WebSocket) -> None:  1 usage
    """Надіслати збережені повідомлення користувачу при підключенні"""
    if client_id in self.stored_messages:
        messages = self.stored_messages[client_id]
        for message in messages:
            try:
                await websocket.send_json(message)
                logger.info(f"Надіслано збережене повідомлення до {client_id}")
            except Exception as e:
                logger.error(f"Помилка надсилання збереженого повідомлення: {str(e)}")
        # Очищуємо збережені повідомлення після надсилання
        del self.stored_messages[client_id]
```

Рис. 3.2.3 – Методи збереження та надсилання повідомлень для оффлайн користувачів

Функція disconnect відповідає за видалення підключення клієнта з активних з’єднань та очищення пам'яті:

```python
async def disconnect(self, websocket: WebSocket, client_id: str) -> None:
    """Видалити WebSocket-підключення для client_id"""
    if client_id in self.active_connections:
        try:
            self.active_connections[client_id].remove(websocket)
            if not self.active_connections[client_id]:
                del self.active_connections[client_id]
            logger.info(f"Підключення видалено для ID {client_id}. Залишилось підключень"
                        f" для цього ID: {len(self.active_connections.get(client_id, []))}")
        except ValueError:
            pass  # WebSocket вже видалено
```

Рис. 3.2.3 – Реалізація менеджменту видалення неактивних підключень

Далі на черзі створення основної компоненту, що обробляє підключення клієнтів через WebSocket та керує процесом передачі повідомлень між ними – функція websocket_endpoint. Вона реалізує обробку реального часу, дозволяючи клієнтам здійснювати двостороннє спілкування без необхідності повторно ініціювати з'єднання для кожного нового повідомлення. Функція працює асинхронно, що забезпечує високу продуктивність та ефективне обслуговування численних одночасних підключень.

При отриманні запиту на підключення через WebSocket функція спочатку викликає метод connect з класу ConnectionManager, що додає нове підключення до активних з'єднань сервера. Кожен клієнт підключається до сервера з унікальним client_id, що дає змогу серверу ідентифікувати та обробляти підключення окремих користувачів. Всі ці підключення зберігаються в словнику активних з'єднань, що дає змогу серверу ефективно керувати ними та забезпечувати доставку повідомлень.

Однією з важливих функцій цієї частини коду є відправка збережених повідомлень для офлайн-користувачів. Якщо користувач підключається до сервера після відсутності в онлайні, йому надсилаються всі збережені повідомлення, які не були доставлені під час його відсутності. Цей процес автоматично виконується за допомогою методу send_stored_messages, описаного вище.

Після встановлення з'єднання, функція переходить у нескінченний цикл, де постійно чекає нових повідомлень від клієнта. Кожне повідомлення приймається через websocket.receive_text(), що забезпечує отримання текстових даних, переданих клієнтом. Далі сервер намагається перетворити отриманий текст у формат JSON для подальшої обробки.

Перш ніж обробляти повідомлення, сервер перевіряє його формат. Якщо відправлене повідомлення не містить необхідних ключів, таких як recipient (отримувач) і content (вміст повідомлення), воно ігнорується, а користувач отримує повідомлення про помилку. Якщо повідомлення має правильну структуру, сервер додає додаткові атрибути, такі як sender (ідентифікатор відправника) та timestamp (час відправлення), що є важливими для подальшої маршрутизації та для забезпечення автентичності повідомлення.

Якщо повідомлення доставлено успішно, відправнику відправляється підтвердження доставки, яке включає статус повідомлення, час його доставки та ідентифікатор отримувача. Це підтвердження дозволяє клієнту знати, чи було його повідомлення доставлено, але при цьому не видається таємниця – чи успішно отримувач розшифрував його, цез роблено задля запобігання можливих вразливостей.

Якщо з'єднання з клієнтом розривається (наприклад, через відключення клієнта), сервер виводить в журнал інформацію про це та очищає відповідне підключення зі списку активних з'єднань. Це відбувається у блоці except WebSocketDisconnect, що дозволяє серверу коректно завершити сесію з користувачем.

Окрім відключень, функція також обробляє різні типи помилок, такі як некоректно сформовані JSON-дані, помилки під час обробки повідомлень або будь-які інші непередбачувані ситуації. Це дозволяє серверу не зупиняти свою роботу через помилки одного користувача і забезпечує стабільність роботи всього застосунку. У разі відключення клієнта від WebSocket-з'єднання сервер коректно завершує процес взаємодії з ним. Якщо з'єднання не було закрите

через відключення клієнта, а в результаті іншої помилки, функція все одно намагається відключити клієнта та видалити його з активних з'єднань.

3.3 Реалізація клієнтського застосунку

Розробка клієнтської частини захищеного обміну повідомленнями спрямована на створення інтерактивного додатку, який дозволяє користувачам взаємодіяти з сервером через WebSocket-з'єднання для відправки та отримання зашифрованих повідомлень. Клієнтська частина використовує асинхронну бібліотеку asyncio для забезпечення ефективної роботи з подіями та потоками, що дозволяє обробляти декілька задач одночасно, таких як надсилання, отримання повідомлень та введення команд користувача.

Спочатку створемо також логування подій на клієнті для зручнішості на етапах розробки та тестування.

Далі створюємо клас SecureMessagingClient, який відповідає за ініціалізацію з'єднання з сервером та обробку основних операцій, таких як підключення, надсилання повідомлень і отримання відповідей. Параметри клієнта включають URL сервера, ID користувача та спільний пароль для шифрування:

```
class SecureMessagingClient:  1 usage
    def __init__(self, server_url: str, client_id: str, passphrase: str):
        """Ініціалізація клієнта захищеного обміну повідомленнями"""
        self.server_url = f"{server_url}/ws/{client_id}"
        self.client_id = client_id
        self.crypto_manager = CryptoManager(passphrase)
        self.websocket: Optional[websockets.client.WebSocketClientProtocol] = None
        self.command_queue: Queue = Queue()
        self._is_connected = False
```

Рис. 3.3.1 – Ініціалізація клієнта

Клас отримує URL сервера, унікальний ідентифікатор клієнта та пароль для шифрування, що є критично важливим для забезпечення безпеки обміну повідомленнями. Для шифрування та розшифрування повідомлень

використовується об'єкт CryptoManager, який ініціалізується через спільний пароль.

Для встановлення з'єднання з сервером використовується метод connect, що викликає асинхронну функцію підключення до WebSocket-сервера. Якщо підключення вдається, користувач бачить вітальне повідомлення та список доступних команд:

```
async def connect(self) -> None:
    """Підключення до сервера обміну повідомленнями"""
    try:
        self.websocket = await websockets.client.connect(self.server_url)
        self._is_connected = True
        logger.info("Підключено до сервера")
        await self._show_welcome_message()
    except Exception as e:
        self._is_connected = False
        logger.error(f"Помилка підключення: {str(e)}")
        raise
```

Рис. 3.3.2 – Функція підключення

У разі успішного підключення клієнт отримує доступ до функціоналу для відправки та отримання повідомлень.

Для надсилання повідомлень клієнт використовує метод send_message, який шифрує повідомлення за допомогою CryptoManager та відправляє його через WebSocket з додатковими даними, такими як ID отримувача та флаг для збереження повідомлення у разі, якщо відправник хоче залишити повідомлення до моменту підключення отримувача:

```python
async def send_message(self, recipient_id: str, message: str,  1 usage
                       store_if_offline: bool = False) -> None:
    """Зашифрувати та надіслати повідомлення отримувачу"""
    if not self.is_connected:
        raise ConnectionError("Немає підключення до сервера")

    try:
        # Шифруємо повідомлення
        encrypted_data = self.crypto_manager.encrypt_message(message)

        # Формуємо та надсилаємо пакет
        message_packet = {
            "type": "message",
            "recipient": recipient_id,
            "content": encrypted_data,
            "store_if_offline": store_if_offline
        }

        # Перевірка для лінтера
        assert self.websocket is not None

        # Надсилаємо повідомлення
        await self.websocket.send(json.dumps(message_packet))

    except Exception as e:
        logger.error(f"Помилка надсилання повідомлення: {str(e)}")
        raise
```

Рис. 3.3.3 – Метод надсилання повідомлень

Перед відправкою повідомлення клієнт перевіряє статус підключення до сервера, і якщо підключення є активним, повідомлення шифрується і надсилається в пакеті через WebSocket.

Для отримання повідомлень від сервера використовується метод receive_messages, який асинхронно чекає на вхідні повідомлення від WebSocket. Кожне отримане повідомлення розшифровується та обробляється:

```python
async def receive_messages(self) -> None:  1 usage
    """Отримання та розшифрування вхідних повідомлень"""
    if not self.is_connected:
        raise ConnectionError("Немає підключення до сервера")

    # Перевірка для лінтера
    assert self.websocket is not None

    while True:
        try:
            message = await self.websocket.recv()
            data = json.loads(message)

            if data.get("type") == "message":
                await self._handle_message(data)
            elif data.get("type") == "delivery_status":
                await self._handle_delivery_status(data)
            else:
                logger.warning(f"Невідомий тип повідомлення:"
                               f"{data.get('type', 'unknown')}")

        except ConnectionClosed:
            self._is_connected = False
            logger.warning("З'єднання з сервером втрачено")
            print("\n[Система]: З'єднання з сервером втрачено")
            break
        except Exception as e:
            logger.error(f"Помилка отримання повідомлення: {str(e)}")
            if "disconnect message has been received" in str(e):
                self._is_connected = False
                print("\n[Система]: З'єднання з сервером втрачено")
                break
            print(f"\n[Помилка]: {str(e)}")
```

Рис. 3.3.4 – Метод отримання повідомлень

Якщо повідомлення має тип "message", воно передається на обробку методом _handle_message, який розшифровує вміст повідомлення, використовуючи CryptoManager, будову якого буде описано в наступному розділі:

```python
async def _handle_message(self, data: dict) -> None:  1 usage
    """Обробка вхідного повідомлення"""
    try:
        decrypted_message = self.crypto_manager.decrypt_message(
            data["content"],
            sender_id=data["sender"]
        )

        if decrypted_message is None:
            # Повідомлення невалідне, рахуємо кількість таких повідомлень
            _, invalid_count = self.crypto_manager.validate_message(
                data["content"],
                sender_id=data["sender"]
            )
            if invalid_count is not None:
                system_msg = (
                    f"[Система]: Отримано {invalid_count} невалідних "
                    f"повідомлень від користувача {data['sender']}"
                )
                print(f"\n{system_msg}")
                logger.warning(system_msg)  # Без \n для логу
            return

        msg_output = f"\n[Повідомлення від {data['sender']}]: {decrypted_message}"
        print(msg_output)

    except Exception as e:
        logger.error(f"Помилка обробки повідомлення: {str(e)}")
```

Рис. 3.3.5 – Обробка повідомлень зі зверненням до CryptoManager

Інтерфейс клієнта розроблений простим текстовим інтерфейсом, що працює через командний рядок. Цей проект присвячений технії, тому графічний інтерфейс не є пріорітетом. Коли користувач запускає клієнт програми, йому надається вітальне повідомлення, що включає доступні команди. Користувач отримує на екран такі інструкції, як ID клієнта для підключення, а також команди для відправки повідомлень, збереження повідомлень на сервері (якщо отримувач офлайн) і виходу з програми.

Користувач може ввести одну з доступних команд в командному рядку. Усі команди мають чітко визначену структуру, що дозволяє швидко зрозуміти, як їх використовувати. Список команд, які доступні для користувача:

- /send <отримувач> <повідомлення> — команда для відправки зашифрованого повідомлення певному отримувачу.
- /store <отримувач> <повідомлення> — команда для відправки повідомлення з опцією збереження на сервері, якщо отримувач офлайн.
- /quit — команда для виходу з програми.
- /help — команда для виведення списку доступних команд та допомоги.

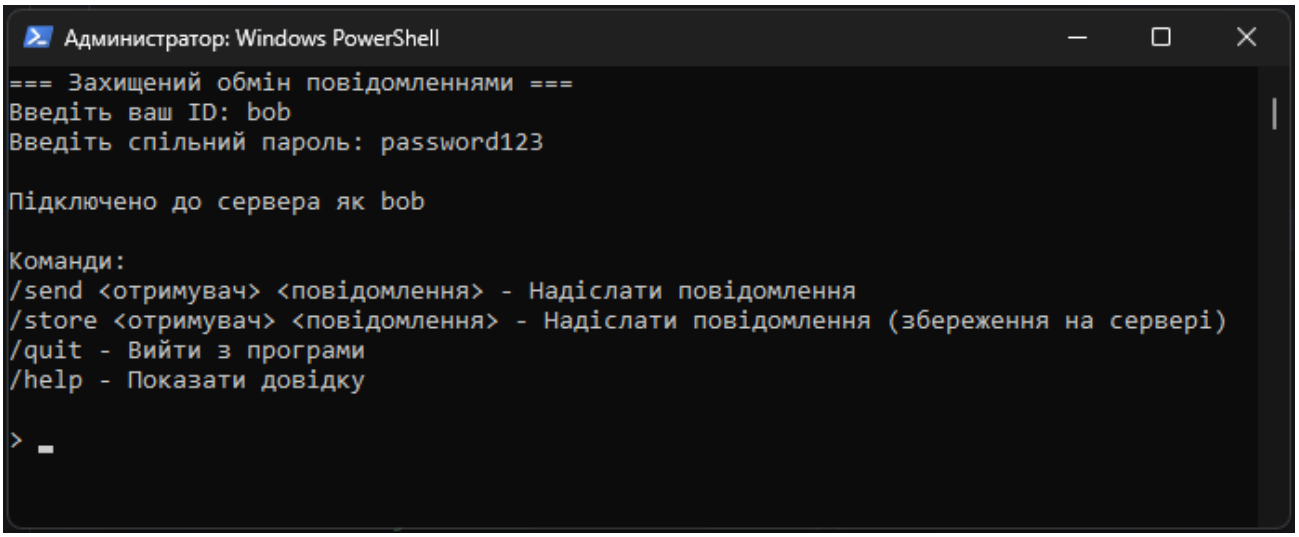

Рис. 3.3.6 – Вигряд інтерфейсу користувача

Метод обробки команд користувача у клієнтській частині реалізовано в асинхронній функції, що безперервно чекає на введення користувача та відповідно реагує на команди, що вводяться в командному рядку.

Коли користувач вводить команду, вона додається до черги command_queue, яка обробляється окремим асинхронним потоком. Цей потік перевіряє введену команду та визначає, яку операцію виконати, залежно від того, чи є команда відправкою повідомлення, збереженням повідомлення або іншим запитом.

Якщо користувач вводить команду /send <отримувач> <повідомлення>, система розбирає введений текст і передає параметри (отримувач та текст повідомлення) на обробку методу, який шифрує повідомлення та відправляє його через з'єднання WebSocket. Якщо ж команда вказує на необхідність збереження повідомлення для офлайн-отримувача (через команду /store), відповідне повідомлення зберігається на сервері для наступної доставки, коли отримувач буде доступний.

```
async def process_commands(self) -> None:  1 usage
    """Обробка команд користувача"""
    while True:
        try:
            command = await self.command_queue.get()
            if command.startswith('/send '):
                await self._handle_send_command(command, store_if_offline=False)
            elif command.startswith('/store '):
                await self._handle_send_command(command, store_if_offline=True)
            elif command == '/quit':
                print("\n[Система]: Вихід...")
                await self.close()
                return
            elif command == '/help':
                await self._show_welcome_message()
            else:
                print("\n[Помилка]: Невідома команда. Введіть /help для списку команд")
        except Exception as e:
            print(f"\n[Помилка]: {str(e)}")
```

Рис. 3.3.7 – Метод обробки команд

На рисунку 3.3.7 також показано, як реалізован механізм обробки помилок. Якщо команда введена некоректно або з неповними параметрами, користувач отримує відповідне повідомлення про помилку, і він може виправити свою помилку або звернутися до довідки. Усі ці дії відбуваються асинхронно, що дозволяє програмі працювати ефективно та реагувати на події без затримок.

Було також проведену низку невеликих покращень для обробки виключень в різних методах та оптимізація коду.

3.4 Криптографічний модуль

3.4.1 Вибір криптографічних механізмів

Було проведено багато досліджень та тестувань задля визначення потрібних механізмів.

Головна задача – забезпечити алгоритми синхронного шифрування без передачі жодних деталей шифрування.

Базовим шуфрувальним механізмом був обраний метод **AES-GCM**. По-перше, це сучасний симетричний алгоритм шифрування, який є дуже швидким та ефективним у використанні. Крім того, він підтримує вбудовану перевірку цілісності даних через механізм автентичності (AEAD — Authenticated Encryption with Associated Data), що дозволяє гарантувати, що дані не були змінені під час передачі. Це особливо важливо для забезпечення цілісності повідомлень у розподілених системах. AES-GCM використовує 128-бітний ключ і 12-бітний ініціалізаційний вектор (IV), що є оптимальним з точки зору балансу між безпекою та ефективністю. Іншою важливою перевагою є те, що AES-GCM є стандартом, широко підтримуваним на різних платформах, що забезпечує сумісність і зручність реалізації.

Для генерації та перевірки підписів повідомлень обрано **Ed25519** — одну з найбільш сучасних і безпечних асиметричних криптографічних схем. Вибір Ed25519 обумовлений його високою швидкістю, безпекою та надійністю. Він забезпечує високий рівень захисту навіть при коротших ключах порівняно з іншими алгоритмами, такими як RSA або ECDSA, і є стійким до атак з боку квантових комп’ютерів, що робить його дуже перспективним для довгострокового використання. Ed25519 використовує 256-бітові ключі, що забезпечує високий рівень криптографічної стійкості при помірних вимогах до обчислювальних ресурсів. Крім того, його швидкість генерації підписів та перевірки значно краща за інші сучасні алгоритми, що робить його ідеальним для реальних систем з високими вимогами до продуктивності.

Для створення ключів та шифрування з пароля був обраний **PBKDF2HMAC** (Password-Based Key Derivation Function 2 з HMAC) — алгоритм, що застосовує функцію хешування для створення ключів на основі паролів. Формально, PBKDF2HMAC як назва не є окремим стандартом чи поняттям у криптографії — це назва конкретної реалізації алгоритму PBKDF2 з HMAC у бібліотеці cryptography для Python. Вибір цього механізму обумовлений його високою безпекою при достатньо великій кількості ітерацій, що значно ускладнює атаки типу перебору. За допомогою PBKDF2HMAC можна генерувати ключі, які мають високу стійкість до атак на основі зловживань зі слабкими паролями. Також було вирішено збільшити кількість ітерацій до 300 тис., що ще більше підвищує стійкість до атак методом перебору. Така конфігурація дозволяє гарантувати високий рівень безпеки навіть при використанні паролів середньої складності.

HMAC (Hash-based Message Authentication Code) був обраний як додатковий механізм забезпечення цілісності повідомлень та верифікації їх автентичності. Цей механізм працює на основі криптографічних хеш-функцій і є швидким і ефективним методом перевірки того, що повідомлення не було змінене під час передачі. HMAC використовує секретний ключ і повідомлення для обчислення контрольної суми, яку можна порівняти на отримувача для перевірки цілісності. Вибір HMAC обумовлений його здатністю працювати з різними хеш-функціями, що дозволяє легко змінювати алгоритм без впливу на всю систему, а також його стійкістю до атак, таких як атаки з використанням колізій.

SHA-256 – є частиною сімейства функцій хешування SHA-2, був обраний для використання в криптографічному модулі через його високий рівень безпеки та широке застосування в сучасних криптографічних протоколах. SHA-256 виконує хешування даних та генерує 256-бітні хеші, що дозволяє забезпечити значну стійкість до атак, зокрема до зіткнень, які є критичними в контексті криптографії. Він має стійкість до сучасних атак, таких як атаки на

колізії. Колізія – це ситуація, коли два різних набори даних генерують однакові хеш-значення. Для алгоритмів старішого покоління, таких як MD5 або SHA-1, були знайдені вразливості, що дозволяють легко знайти такі колізії. SHA-256, завдяки більш складним математичним операціям, значно ускладнює пошук таких колізій, роблячи його набагато більш стійким до подібних атак. З огляду на свою довжину хешу (256 біт) SHA-256 забезпечує дуже високий рівень ентропії, що значно зменшує ймовірність того, що два різних набори даних призведуть до однакових хешів. Це є критичним, коли мова йде про перевірку автентичності повідомлень, де важливо, щоб кожен вхідний набір даних мав унікальне хеш-значення, яке неможливо підробити.

SHA-256 також обраний через його ефективність у контексті обчислювальних ресурсів. Алгоритм швидко виконується на сучасних обчислювальних пристроях і забезпечує високу продуктивність, що є важливим фактором при обміні повідомленнями в реальному часі. Використання цієї функції дозволяє забезпечити високий рівень безпеки без значного навантаження на систему.

Використання SHA-256 у комбінації з іншими механізмами, такими як HMAC для перевірки цілісності повідомлень, створює додаткову криптографічну безпеку. Завдяки поєднанню цього хешування з секретним ключем, HMAC здатний гарантувати, що повідомлення не було змінено в процесі передачі, забезпечуючи цілісність та автентичність даних. В результаті SHA-256 стає важливою частиною більш складної криптографічної структури, що забезпечує надійний захист повідомлень від зловмисників.

Також для додаткового захисту від аналізу зашифрованих даних було вирішено скористатися **паддінгом**, який додається до даних перед їх шифруванням. Паддінг створюється випадковим чином, щоб приховати реальну довжину повідомлення та ускладнити потенційним зловмисникам можливість визначення розміру даних без розшифрування. Це забезпечує

додатковий рівень захисту, роблячи систему більш стійкою до атак типу "аналітичного дослідження повідомлень".

Ці криптографічні механізми були вибрані не лише з огляду на їх індивідуальні властивості, але й з урахуванням їх сумісності між собою, що дозволяє створити систему, яка є як ефективною, так і стійкою до сучасних та потенційних загроз. Вибір таких алгоритмів гарантує надійність системи навіть в умовах високої активності зловмисників. А також забезпечує високу продуктивність при розробці та використанні криптографічних засобів в реальних системах обміну повідомленнями.

Слід зазначити, що вивчалися і інші методи шифрування, такі як метод **DH** (протокол Діффі — Геллмана англ. Diffie-Hellman). Протокол обміну ключами Диффі-Геллмана був одним із перших криптографічних алгоритмів, який дозволяє двом сторонам, що не мають спільного секрету, безпечним чином обмінюватися ключем через незахищену комунікаційну лінію. Протокол був запропонований Уїтфілдом Діффі і Мартіном Геллманом в 1976 році, і з того часу став основою для більшості сучасних криптографічних систем, зокрема для протоколів безпечного обміну даними в Інтернеті. Важливою особливістю цього протоколу є те, що, навіть якщо третя сторона має доступ до всієї інформації, яка передається між сторонами, вона не зможе обчислити спільний ключ без знання певної математичної інформації, що є достатньо складною для розв’язання. Протокол забезпечує конфіденційність при створенні спільного секретного ключа, який можна використовувати для шифрування подальшої комунікації.

Але сучасні спостереження виявляють цей протокол ненадійним. Це пов’язано не тільки з підвищенням ресурсів обчислення, але з атаками типу "людина посередині" (MITM), замінивши публічні ключі і встановивши власний спільний секрет між собою і кожним учасником. Для захисту від таких атак протокол можна вдосконалити за допомогою додаткових механізмів

автентифікації, таких як підписи або сертифікати, що підтверджують справжність публічних ключів.

Саме тому для даного проекту була обрана модель заздалегідь домовленного спільного паролю.

3.4.2 Розробка криптографічного модулю

Для основних криптографічних процесів створемо клас CryptoManager. Він буде працювати з різними криптографічними алгоритмами та інструментами, зокрема AES-GCM для шифрування, Ed25519 для підпису, HMAC для верифікації цілісності та PBKDF2HMAC для генерації ключів. Усі ці механізми забезпечують комплексний рівень захисту і працюють у тісній взаємодії.

На самому початку, коли ми створюємо криптографічний модуль, важливо зберігати дані в захищеному вигляді. Для цього ми використовуємо PBKDF2HMAC, який застосовується для генерації ключів на основі пароля. Оскільки пароль користувача є менш надійним, ніж випадково згенерований ключ, ми використовуємо PBKDF2HMAC, який здійснює багаторазове хешування пароля (300 000 ітерацій), що значно ускладнює перебір паролів навіть при слабших паролях. Вибір цього алгоритму обумовлений його надійністю та стійкістю до атак на основі перебору.

Першим кроком є генерація сілі для ключа підпису. Це відбувається за допомогою хешування пароля з додатковим контекстом. Створюється хеш за допомогою SHA-256, а потім результат хешування використовується для створення сілі (random salt), яка додається до пароля перед застосуванням алгоритму PBKDF2HMAC. Така структура забезпечує створення унікальних ключів для кожного користувача, навіть якщо їх паролі схожі.

```
digest = hashes.Hash(hashes.SHA256())

digest.update(b"signing_key_salt")

digest.update(self.passphrase
```

```
signing_salt = digest.finalize()
```

Далі за допомогою PBKDF2HMAC ми отримуємо ключ для підпису на основі цієї солі. Оскільки Ed25519 вимагає 32-байтовий ключ, ми забезпечуємо його правильну генерацію через PBKDF2HMAC.

```
kdf = PBKDF2HMAC(
        algorithm=hashes.SHA256(),
        length=32,  # Розмір ключа Ed25519
        salt=signing_salt,  # Сіль залежить від паролю
        iterations=self.PBKDF2_ITERATIONS,
    )
    signing_seed = kdf.derive(self.passphrase)
self.signing_key =
ed25519.Ed25519PrivateKey.from_private_bytes(signing_seed)
```

Ключ для HMAC створюється аналогічним чином, але з використанням іншої солі. HMAC в даному контексті служить для перевірки цілісності повідомлення, що є важливим механізмом для підтвердження того, що дані не були змінені під час передачі.

Далі ми використовуємо AES-GCM для шифрування повідомлень. Цей алгоритм забезпечує одночасно конфіденційність і цілісність даних завдяки підтримці автентичності. Для шифрування створюється випадкова сіль та ініціалізаційний вектор (IV). Шифрування в AES-GCM вимагає додаткових даних, таких як nonce та timestamp, що допомагає забезпечити унікальність кожного повідомлення.

Для того, щоб уникнути зворотного аналізу довжини повідомлень, ми використовуємо паддінг, що додає випадкову кількість байтів до повідомлення.

Це робить його довжину непередбачуваною, що забезпечує додаткову безпеку від можливих атак, орієнтованих на довжину повідомлення.

```python
def _add_padding(self, data: bytes) -> bytes:  1 usage
    """Додає випадковий паддінг для маскування реальної довжини повідомлення"""
    pad_len = secrets.randbelow(self.MAX_PADDING + 1)  # 0..128
    padding = os.urandom(pad_len)
    return bytes([pad_len]) + data + padding

def _strip_padding(self, padded: bytes) -> bytes:  1 usage
    """Видаляє паддінг та повертає оригінальне повідомлення"""
    pad_len = padded[0]
    if pad_len:
        return padded[1:-pad_len]
    return padded[1:]
```

Рис. 3.4.2.1 – Функції роботи з паддінгом

Після цього повідомлення шифрується, і отриманий результат (шифротекст) підписується за допомогою Ed25519, щоб гарантувати, що повідомлення не було змінено під час передачі.

signature = self.signing_key.sign(ciphertext)

Після підпису, для забезпечення автентичності та цілісності, до шифрованого пакету додається HMAC, що використовується для перевірки правильності повідомлення на стороні отримувача.

hmac_value = self._create_hmac(ciphertext, nonce, timestamp)

Таким чином, пакет містить не лише зашифровані дані, але й інформацію для перевірки їх цілісності, підпису та автентичності.

Для дешифрування повідомлення спочатку перевіряється його валідність, зокрема перевіряються HMAC і підпис. Якщо всі перевірки успішні, повідомлення розшифровується за допомогою AES-GCM, після чого з нього видаляється паддінг.

padded_plaintext = aesgcm.decrypt(iv, ciphertext, None)

plaintext_bytes = self._strip_padding(padded_plaintext)

Ми допускаємо, що загроза спаму чи replay атак має велику ймовірність в системі, де всі користувачі можуть обирати повторювальні client_id. Тому створимо захист від невалідних повідомлень. validate_message відповідає за перевірку, що відправник насправді є очікуваним співбесідником за допомогою перевірки успішності розшифрування. Якщо повідомлення невдалося розшифрувати, то воно не буде відображатися в інтерфейсі, але система буде з деякою періодичністю сповіщати про наявність невалідних повідомлень з їх лічильником, щоб користувач знав про неполадки або якщо він став об'єктом спам атаки. А також ця функція перевіряє чи не було змінено повідомлення під час його доставки. Задля захисту від replay атак перевіряється, чи не прострочено повідомлення (наприклад, через значення timestamp), що додатково гарантує, що нове повідомлення не є дублікатом старого.

```python
def validate_message(self, encrypted_package_str: str, sender_id: str = "unknown") -> tuple[bool, Optional[int]]:
    """
    Перевірити валідність повідомлення.
    Повертає (is_valid, invalid_count), де invalid_count - кількість неправильних повідомлень
    для сповіщення, якщо досягнуто поріг, інакше None
    """
    try:
        # Декодуємо пакет
        encrypted_package = json.loads(b64decode(encrypted_package_str).decode('utf-8'))

        # Декодуємо base64 компоненти
        ciphertext = b64decode(encrypted_package['ciphertext'])
        signature = b64decode(encrypted_package['signature'])
        hmac_value = b64decode(encrypted_package['hmac'])
        nonce = encrypted_package['nonce']
        timestamp = encrypted_package['timestamp']

        # Перевіряємо час
        if abs(time.time() - timestamp) > 3600:  # 1 година
            return False, self.validator.record_invalid_message(sender_id)

        # Перевіряємо HMAC
        if not self._verify_hmac(ciphertext, nonce, timestamp, hmac_value):
            return False, self.validator.record_invalid_message(sender_id)

        # Перевіряємо підпис
        try:
            self.verify_key.verify(signature, ciphertext)
        except Exception:
            return False, self.validator.record_invalid_message(sender_id)

        return True, None

    except Exception:
        return False, self.validator.record_invalid_message(sender_id)
```

Рис. 3.4.2.1 – Метод валідації повідомлень

Метод encrypt_message реалізує повний цикл шифрування повідомлення, який забезпечує одночасно конфіденційність, цілісність, автентичність і захист від повторного використання даних. Тому треба зробити докладний розбір його роботи.

У процесі розробки цього методу ставилося завдання створити такий формат повідомлення, який буде містити в собі всю необхідну інформацію для перевірки, розшифрування та підтвердження справжності без зберігання стану на сервері.

Першим етапом у цьому методі є генерація випадкових значень, які забезпечують унікальність кожного шифрованого повідомлення. Для цього

створюється сіль (salt) розміром 16 байт та ініціалізаційний вектор (iv) довжиною 12 байт за допомогою функції os.urandom.

```
salt = os.urandom(16)

iv = os.urandom(12)
```

Ці значення гарантують, що навіть якщо буде зашифроване однакове повідомлення з тим самим паролем, результат шифрування буде іншим. Сіль надалі використовується в алгоритмі PBKDF2HMAC для детермінованої генерації ключа шифрування на основі паролю, а IV — безпосередньо у шифруванні AES-GCM.

Щоб уникнути атак типу replay, ми додаємо унікальний nonce і часову мітку timestamp, які потім використовує метод валідації, описаний вище. nonce створюється як шістнадцятковий рядок довжиною 32 символи (тобто 16 байт ентропії), а timestamp фіксує точний час створення повідомлення в секундах.

```
nonce = secrets.token_hex(16)

timestamp = time.time()
```

Наступним кроком ми використовуємо функцію derive_key, яка отримує 32-байтовий ключ шифрування з паролю користувача та випадково згенерованої солі. Сам ключ застосовується для створення об'єкта AESGCM, який реалізує шифрування з автентифікацією.

```
key = self.derive_key(salt)

aesgcm = AESGCM(key)
```

Перш ніж здійснити шифрування, до повідомлення додається випадковий паддінг, щоб замаскувати реальну довжину повідомлення. Цей паддінг додається методом _add_padding, описаним вище. Один байт на початку зберігає довжину паддінгу, щоб при розшифруванні його можна було відновити.

message_bytes = self._add_padding(message.encode('utf-8'))

Після цього виконується шифрування padded-повідомлення за допомогою алгоритму AES у режимі GCM. Цей режим дозволяє шифрувати й автентифікувати дані за один прохід, що знижує ризики та підвищує продуктивність. Отриманий шифротекст (ciphertext) захищений як від прочитання, так і від модифікації.

ciphertext = aesgcm.encrypt(iv, message_bytes, None)

Щоб забезпечити автентичність повідомлення, ми підписуємо шифротекст за допомогою приватного ключа Ed25519. Цифровий підпис дозволяє стороні, яка має публічний ключ, переконатися в тому, що повідомлення було створене саме власником приватного ключа і не змінювалось.

signature = self.signing_key.sign(ciphertext)

Для додаткового рівня захисту та цілісності ми створюємо HMAC над комбінацією ciphertext, nonce і timestamp. HMAC виконує роль контрольної суми, яка з високою точністю дозволяє виявити найменші зміни в повідомленні або його параметрах.

hmac_value = self._create_hmac(ciphertext, nonce, timestamp)

Заключним етапом є пакування всіх компонентів у єдину структуру — encrypted_package. До неї входять: шифротекст, IV, сіль, підпис, HMAC, nonce та часовий штамп. Усі бінарні дані кодуються в Base64, оскільки фінальний результат потрібно буде передавати у вигляді рядка. Після цього словник серіалізується у JSON, кодується в UTF-8 та ще раз в Base64 для безпечного транспортування.

```
encrypted_package = {
    'ciphertext': b64encode(ciphertext).decode('utf-8'),
    'iv': b64encode(iv).decode('utf-8'),
```

```
        'salt': b64encode(salt).decode('utf-8'),
        'signature': b64encode(signature).decode('utf-8'),
        'hmac': b64encode(hmac_value).decode('utf-8'),
        'nonce': nonce,
        'timestamp': timestamp
    }
```

Закінчується метод поверненням зашифрованого пакету як звичайного рядка.

```
return b64encode(json.dumps(encrypted_package).encode('utf-8')).decode('utf-8')
```

Задля усунення сумнівів, слід зазначити, що зберігання всіх елементів в єдиному зашифрованому пакеті є безпечним, оскільки кожен з компонентів захищений за допомогою сильних криптографічних методів. Шифротекст (ciphertext) зашифрований за допомогою AES-GCM, що забезпечує як конфіденційність, так і цілісність даних. IV і сіль гарантують унікальність кожного шифрованого повідомлення, навіть якщо самі дані і пароль повторюються. Використання HMAC та цифрового підпису додає додаткові шари перевірки автентичності і захисту від модифікацій, а також захищає від атак повторного використання пакету. Оскільки для кожного повідомлення генеруються нові значення IV, nonce і сіль, шифрування стає стійким до атак методом брутфорсу. Крім того, використання сильних алгоритмів, таких як Ed25519 для підпису та AES-GCM для шифрування, робить такий підхід надзвичайно складним для злому, навіть при спробах перебору ключів або HMAC.

Навіть з повним доступом до сервера і пакету, навіть зі знанням структури шифрування, зловмисник не зможе розшифрувати повідомлення, якщо не має правильного пароля (passphrase). Усі ключі — для AES, HMAC і

цифрового підпису — детерміновано виводяться з пароля за допомогою PBKDF2 з великою кількістю ітерацій, що робить перебір або обхід практично неможливим. Навіть маючи всі елементи пакету — ciphertext, salt, iv, signature, hmac, nonce, timestamp — без правильного пароля отримати ключ і розшифрувати повідомлення криптографічно неможливо.

3.5 Висновки до розділу 3

Розробка проекту захищеного обміну повідомленнями базується на архітектурі клієнт-сервер, де основним принципом є забезпечення конфіденційності та цілісності повідомлень, навіть у випадку компрометації серверної інфраструктури. Вся криптографічна обробка даних здійснюється на клієнтській стороні, що мінімізує ризики від зловмисників, які можуть отримати доступ до сервера.

Використання WebSocket-з'єднання для комунікації в реальному часі та шифрування повідомлень перед їх передачею дозволяє забезпечити захищений обмін даними. Крім того, зберігання повідомлень на сервері для офлайн-отримувачів без доступу до ключів шифрування забезпечує додатковий рівень безпеки. Всі повідомлення передаються у зашифрованому вигляді, що робить їх недоступними для сторонніх осіб.

Розробка серверної частини здійснена з мінімальним довірчим навантаженням на сервер, який лише пересилає зашифровані дані між клієнтами, не зберігаючи конфіденційну інформацію або ключі. Для реалізації сервера був вибраний веб-фреймворк FastAPI, який забезпечує високу продуктивність і підтримує асинхронне програмування для обробки численних одночасних підключень.

Клієнтська частина забезпечує зручний інтерфейс для обміну зашифрованими повідомленнями та підтримує кілька функцій, включаючи збереження повідомлень для офлайн-отримувачів. Асинхронна обробка повідомлень через бібліотеку asyncio дозволяє ефективно обробляти великі обсяги даних і зберігати стабільність роботи додатку.

У результаті, проект забезпечує високу конфіденційність та цілісність переданої інформації, навіть якщо сервер буде скомпрометований.

ВИСНОВКИ РОБОТИ

У ході виконання дипломного проекту було спроєктовано, реалізовано та протестовано систему захищеного обміну повідомленнями, що функціонує на основі архітектури «клієнт — сервер», у якій уся обробка конфіденційної інформації здійснюється виключно на клієнтській стороні. Такий підхід забезпечує високий рівень безпеки, мінімізуючи залежність від довіри до серверної інфраструктури. Сервер виступає лише транспортним вузлом для пересилання зашифрованих повідомлень та збереження недоставлених пакетів без доступу до їх змісту.

Основною задачею роботи було створення механізму, здатного забезпечити конфіденційність, автентичність, цілісність та стійкість до повторного використання повідомлень. Для досягнення цієї мети було застосовано набір сучасних криптографічних методів: алгоритм симетричного шифрування AES у режимі GCM, цифрові підписи на основі Ed25519, механізм HMAC на базі SHA-256, а також функцію деривації ключів PBKDF2 з великою кількістю ітерацій. Ретельний вибір і взаємодія цих механізмів дозволили побудувати систему, стійку до атак на різних рівнях — від перехоплення трафіку до компрометації сервера.

У процесі реалізації клієнтської частини було створено консольний інтерфейс із підтримкою простого набору команд, який дозволяє надсилати повідомлення, зберігати їх на сервері для офлайн-отримувачів та взаємодіяти з системою без складних налаштувань. Серверна частина, реалізована за допомогою FastAPI та WebSocket, забезпечує ефективну маршрутизацію повідомлень та масштабовану роботу з багатьма одночасними підключеннями.

Окрему увагу приділено захисту від типових загроз: реалізовано валідацію вхідних повідомлень, механізми виявлення спаму та перевірку часової актуальності даних, що дозволяє забезпечити стійкість системи до атак повторного використання (replay-атаки) і масованих спроб компрометації.

Таким чином, розроблена система відповідає вимогам до безпеки персональних комунікацій, дозволяє обмінюватися захищеними повідомленнями без необхідності централізованого зберігання чутливих даних та є надійною платформою для подальшого розвитку функціоналу. Робота демонструє можливість побудови захищеного середовища обміну інформацією на основі відкритих стандартів і технологій, навіть у контексті частково довіреного середовища. Такий продукт може знадобитися в сферах життя, в яких приватні користувачі володіють життєво важливимо таємницями, наприклад у воєнній сфері, у сфері секретних досліджень, спеціальних операціях, для допомоги підтримання адвокатської таємниці і подібних.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Вербіцький О.В. Вступ до криптографії / О.В. Вербіцький. – Львів: Видавництво науково-технічної літератури, 1998. – 247 с.
2. Говард Б., Парідеанс О., Гамм Б. Інформаційна безпека: загрози та механізми захисту. – 2003. – с. 117–121.
3. Усов, Я. Ю. (2021). Проблеми захищеності інформаційного середовища. Технічні науки та технології, 1(15), 145–151.
4. ДСТУ 3396.2-97. Захист інформації. Технічний захист інформації. Терміни та визначення. [Чинний від 01.01.1998 р.]
5. Наукоємні технології в інфокомунікаціях: обробка та захист інформації: колективна монографія / за редакцією В. М. Безрука, В. В. Баранника. Харків: Компанія СМІТ, 2013. 398 с.
6. Documentation. Signal Messenger. [Електронний ресурс] / Режим доступу: https://signal.org/docs/
7. Specifications >> The Double Ratchet Algorithm. Signal Messenger. [Електронний ресурс] / Режим доступу: https://signal.org/docs/specifications/doubleratchet/
8. Looking back at how Signal works, as the world moves forward. Signal Messenger. [Електронний ресурс] / Режим доступу: https://signal.org/blog/looking-back-as-the-world-moves-forward/
9. A. Mallik, A. Ahsan, M.M.Z. Shahadat, J.-C. Tsou Man-in-the-middle-attack: Understanding in simple words // International Journal of Data and Network Science. – 2019.
10. Брюс М. Ван Хорн, Куан Нгуен: PyCharm: професійна робота на Python / пер. з англ. І. Л. Люско. – М.: ДМК Пресс. – 2024. – 618 с.

ДОДАТОК А
ПРОГРАМНИЙ КОД

server.py

```
from fastapi import FastAPI, WebSocket, WebSocketDisconnect
from typing import Dict, List
import json
from datetime import datetime
import logging
import uvicorn

# Налаштування логування
logging.basicConfig(
    level=logging.INFO,
    format='%(asctime)s - %(levelname)s - %(message)s',
    datefmt='%Y-%m-%d %H:%M:%S'
)
logger = logging.getLogger(__name__)

class ConnectionManager:
    def __init__(self):
        # Активні WebSocket підключення (client_id -> список підключень)
        self.active_connections: Dict[str, List[WebSocket]] = {}
        # Повідомлення для офлайн користувачів (recipient_id -> список повідомлень)
        self.stored_messages: Dict[str, List[dict]] = {}

    async def connect(self, websocket: WebSocket, client_id: str) -> None:
        """Додати нове WebSocket-підключення для client_id"""
        await websocket.accept()
```

```
        if client_id not in self.active_connections:
            self.active_connections[client_id] = []
        self.active_connections[client_id].append(websocket)
        logger.info(f"Нове підключення для ID {client_id}. Всього підключень: {len(self.active_connections[client_id])}")

        # Доставка збережених повідомлень
        await self.send_stored_messages(client_id, websocket)

    async def disconnect(self, websocket: WebSocket, client_id: str) -> None:
        """Видалити WebSocket-підключення для client_id"""
        if client_id in self.active_connections:
            try:
                self.active_connections[client_id].remove(websocket)
                if not self.active_connections[client_id]:
                    del self.active_connections[client_id]
                logger.info(f"Підключення видалено для ID {client_id}. Залишилось підключень"
                            f" для цього ID: {len(self.active_connections.get(client_id, []))}")
            except ValueError:
                pass # WebSocket вже видалено

    async def store_message(self, message: dict, recipient_id: str) -> None:
        """Зберегти повідомлення для офлайн користувача"""
        if recipient_id not in self.stored_messages:
            self.stored_messages[recipient_id] = []
        self.stored_messages[recipient_id].append(message)
        logger.info(f"Повідомлення збережено для {recipient_id}")
```

```
    async def send_stored_messages(self, client_id: str, websocket: WebSocket) -> None:
        """Надіслати збережені повідомлення користувачу при підключенні"""
        if client_id in self.stored_messages:
            messages = self.stored_messages[client_id]
            for message in messages:
                try:
                    await websocket.send_json(message)
                    logger.info(f"Надіслано збережене повідомлення до {client_id}")
                except Exception as e:
                    logger.error(f"Помилка надсилання збереженого повідомлення: {str(e)}")
            # Очищуємо збережені повідомлення після надсилання
            del self.stored_messages[client_id]

    async def broadcast_to_recipient(self, message: dict, recipient_id: str, store_if_offline: bool = False) -> bool:
        """Надіслати повідомлення всім підключенням отримувача. Повертає True якщо повідомлення доставлено."""
        if recipient_id not in self.active_connections:
            if store_if_offline:
                await self.store_message(message, recipient_id)
                logger.info(f"Отримувач {recipient_id} офлайн, повідомлення збережено")
            else:
                logger.warning(f"Немає активних підключень для отримувача {recipient_id}")
            return False

        delivered = False
```

```
        failed_connections = []
        for websocket in self.active_connections[recipient_id]:
            try:
                await websocket.send_json(message)
                delivered = True
            except Exception as e:
                logger.error(f"Не вдалося надіслати повідомлення до {recipient_id}:
{str(e)}")
                failed_connections.append(websocket)

        # Очищення неактивних підключень
        for failed in failed_connections:
            await self.disconnect(failed, recipient_id)

        if delivered:
            logger.debug(f"Повідомлення надіслано до
{len(self.active_connections[recipient_id])} підключень отримувача
{recipient_id}")

        return delivered

app = FastAPI()
manager = ConnectionManager()

@app.websocket("/ws/{client_id}")
async def websocket_endpoint(websocket: WebSocket, client_id: str):
    """Обробка WebSocket-підключень та маршрутизація повідомлень"""
    await manager.connect(websocket, client_id)

    try:
```

```
        while True:
            try:
                # Отримання повідомлення
                data = await websocket.receive_text()
                message = json.loads(data)

                # Валідація формату повідомлення
                if not all(k in message for k in ['recipient', 'content']):
                    logger.error(f"Невірний формат повідомлення від {client_id}")
                    continue

                # Метадані повідомлення
                message.update({
                    "sender": client_id,
                    "timestamp": datetime.now().isoformat()
                })

                # Опція зберігання для офлайн отримувачів
                store_if_offline = message.get("store_if_offline", False)

                # Спроба доставки повідомлення
                delivered = await manager.broadcast_to_recipient(message,
message["recipient"], store_if_offline)

                # Відправка статусу доставки
                delivery_status = {
                    "type": "delivery_status",
                    "delivered": delivered,
                    "recipient": message["recipient"],
                    "timestamp": datetime.now().isoformat()
```

```
                }
                await websocket.send_json(delivery_status)

            except WebSocketDisconnect:
                logger.info(f"Клієнт {client_id} відключився")
                break
            except json.JSONDecodeError:
                logger.error(f"Невірний JSON від {client_id}")
            except Exception as e:
                logger.error(f"Помилка обробки повідомлення від {client_id}:
{str(e)}")
                if "disconnect message has been received" in str(e):
                    break

    except WebSocketDisconnect:
        logger.info(f"WebSocket-з'єднання закрито для ID {client_id}")
    except Exception as e:
        logger.error(f"Неочікувана помилка для {client_id}: {str(e)}")
    finally:
        await manager.disconnect(websocket, client_id)

if __name__ == "__main__":
    logger.info("Запуск сервера захищеного обміну повідомленнями...")
    uvicorn.run(app, host="26.238.86.96", port=8000)
```

client.py

```
import asyncio
import json
import websockets.client
from crypto_utils import CryptoManager
from datetime import datetime
```

```
from websockets.exceptions import ConnectionClosed
import logging
from asyncio import Queue
from typing import Optional

# Конфігурація логування
logging.basicConfig(
    level=logging.INFO,
    format='%(asctime)s - %(levelname)s - %(message)s',
    datefmt='%Y-%m-%d %H:%M:%S',
    handlers=[
        logging.FileHandler('client.log'),  # Логування в файл
    ]
)
logger = logging.getLogger(__name__)

class SecureMessagingClient:
    def __init__(self, server_url: str, client_id: str, passphrase: str):
        """Ініціалізація клієнта захищеного обміну повідомленнями"""
        self.server_url = f"{server_url}/ws/{client_id}"
        self.client_id = client_id
        self.crypto_manager = CryptoManager(passphrase)
        self.websocket: Optional[websockets.client.WebSocketClientProtocol] =
None
        self.command_queue: Queue = Queue()
        self._is_connected = False

    @property
    def is_connected(self) -> bool:
        """Перевірка підключення до сервера"""
```

```
        return self._is_connected and self.websocket is not None

    async def connect(self) -> None:
        """Підключення до сервера обміну повідомленнями"""
        try:
            self.websocket = await websockets.client.connect(self.server_url)
            self._is_connected = True
            logger.info("Підключено до сервера")
            await self._show_welcome_message()
        except Exception as e:
            self._is_connected = False
            logger.error(f"Помилка підключення: {str(e)}")
            raise

    async def _show_welcome_message(self) -> None:
        """Вивід вітального повідомлення та команд"""
        print(f"\nПідключено до сервера як {self.client_id}")
        print("\nКоманди:")
        print("/send <отримувач> <повідомлення> - Надіслати повідомлення")
        print("/store <отримувач> <повідомлення> - Надіслати повідомлення (збереження на сервері)")
        print("/quit - Вийти з програми")
        print("/help - Показати довідку")

    async def send_message(self, recipient_id: str, message: str,
                           store_if_offline: bool = False) -> None:
        """Зашифрувати та надіслати повідомлення отримувачу"""
        if not self.is_connected:
            raise ConnectionError("Немає підключення до сервера")
```

```
        try:
            # Шифрування повідомлення
            encrypted_data = self.crypto_manager.encrypt_message(message)

            # Формування пакету повідомлення
            message_packet = {
                "type": "message",
                "recipient": recipient_id,
                "content": encrypted_data,
                "store_if_offline": store_if_offline
            }

            # Валідація WebSocket з'єднання
            assert self.websocket is not None

            # Надсилання повідомлення
            await self.websocket.send(json.dumps(message_packet))

        except Exception as e:
            logger.error(f"Помилка надсилання повідомлення: {str(e)}")
            raise

    async def receive_messages(self) -> None:
        """Отримання та розшифрування вхідних повідомлень"""
        if not self.is_connected:
            raise ConnectionError("Немає підключення до сервера")

        # Валідація WebSocket з'єднання
        assert self.websocket is not None
```

```
        while True:
            try:
                message = await self.websocket.recv()
                data = json.loads(message)

                # Обробка різних типів повідомлень
                if data.get("type") == "message":
                    await self._handle_message(data)
                elif data.get("type") == "delivery_status":
                    await self._handle_delivery_status(data)
                else:
                    logger.warning(f"Невідомий тип повідомлення: {data.get('type', 'unknown')}")

            except ConnectionClosed:
                self._is_connected = False
                logger.warning("З'єднання з сервером втрачено")
                print("\n[Система]: З'єднання з сервером втрачено")
                break
            except Exception as e:
                logger.error(f"Помилка отримання повідомлення: {str(e)}")
                if "disconnect message has been received" in str(e):
                    self._is_connected = False
                    print("\n[Система]: З'єднання з сервером втрачено")
                    break
                print(f"\n[Помилка]: {str(e)}")

    async def _handle_message(self, data: dict) -> None:
        """Обробка вхідного повідомлення"""
        try:
```

```
            decrypted_message = self.crypto_manager.decrypt_message(
                data["content"],
                sender_id=data["sender"]
            )

            if decrypted_message is None:
                # Повідомлення невалідне, рахуємо кількість таких повідомлень
                _, invalid_count = self.crypto_manager.validate_message(
                    data["content"],
                    sender_id=data["sender"]
                )
                if invalid_count is not None:
                    system_msg = (
                        f"[Система]: Отримано {invalid_count} невалідних "
                        f"повідомлень від користувача {data['sender']}"
                    )
                    print(f"\n{system_msg}")
                    logger.warning(system_msg)  # Без \n для логу
                return

            msg_output = f"\n[Повідомлення від {data['sender']}]: {decrypted_message}"
            print(msg_output)

        except Exception as e:
            logger.error(f"Помилка обробки повідомлення: {str(e)}")

    async def _handle_delivery_status(self, data: dict) -> None:
        """Обробка статусу доставки повідомлення"""
        recipient = data.get("recipient", "невідомий")
```

```
        if data.get("delivered", False):
            print(f"\n[Система]: Повідомлення для {recipient} успішно доставлено")
        else:
            print(f"\n[Система]: Не вдалося доставити повідомлення для {recipient} (користувач офлайн)")

    async def close(self) -> None:
        """Закрити з'єднання"""
        if self.websocket:
            await self.websocket.close()
            self._is_connected = False
            logger.info("З'єднання закрито")

    async def process_commands(self) -> None:
        """Обробка команд користувача"""
        while True:
            try:
                command = await self.command_queue.get()
                if command.startswith('/send '):
                    await self._handle_send_command(command, store_if_offline=False)
                elif command.startswith('/store '):
                    await self._handle_send_command(command, store_if_offline=True)
                elif command == '/quit':
                    print("\n[Система]: Вихід...")
                    await self.close()
                    return
                elif command == '/help':
                    await self._show_welcome_message()
                else:
```

```
                print("\n[Помилка]: Невідома команда. Введіть /help для списку команд")
            except Exception as e:
                print(f"\n[Помилка]: {str(e)}")

    async def _handle_send_command(self, command: str, store_if_offline: bool = False) -> None:
        """Обробка команди надсилання повідомлення"""
        prefix_len = 7 if store_if_offline else 6  # /store or /send
        parts = command[prefix_len:].split(' ', 1)
        if len(parts) == 2:
            recipient, message = parts
            await self.send_message(recipient, message, store_if_offline)
        else:
            cmd = "/store" if store_if_offline else "/send"
            print(f"\n[Помилка]: Невірний формат. Використовуйте: {cmd} <отримувач> <повідомлення>")

async def input_handler(queue: Queue) -> None:
    """Обробка введення користувача в окремому потоці"""
    while True:
        try:
            command = await asyncio.get_event_loop().run_in_executor(None, input, "\n> ")
            await queue.put(command.strip())
            if command.strip() == '/quit':
                break
        except EOFError:
            break
```

```
def get_user_input() -> tuple[str, str]:
    """Отримати ID користувача та пароль"""
    print("\n=== Захищений обмін повідомленнями ===")

    while True:
        client_id = input("Введіть ваш ID: ").strip()
        if client_id:
            break
        print("Помилка: ID не може бути порожнім!")

    while True:
        passphrase = input("Введіть спільний пароль: ").strip()
        if passphrase:
            break
        print("Помилка: Пароль не може бути порожнім!")

    return client_id, passphrase

async def main() -> None:
    """Точка входу програми"""
    client_id, passphrase = get_user_input()

    client = SecureMessagingClient(
        server_url="ws://26.238.86.96:8000",
        client_id=client_id,
        passphrase=passphrase
    )

    try:
        await client.connect()
```

```
            # Запуск фонових задач
            input_task = asyncio.create_task(input_handler(client.command_queue))
            receive_task = asyncio.create_task(client.receive_messages())
            command_task = asyncio.create_task(client.process_commands())

            # Очікування завершення задач
            await asyncio.gather(input_task, receive_task, command_task)

        except KeyboardInterrupt:
            print("\nВихід з клієнта...")
        except Exception as e:
            print(f"\n[Помилка]: {str(e)}")
        finally:
            await client.close()

    if __name__ == "__main__":
        try:
            asyncio.run(main())
        except KeyboardInterrupt:
            print("\nДодаток завершено користувачем")
```

crypto_utils.py

```
import os
from base64 import b64encode, b64decode
from cryptography.hazmat.primitives import hashes, hmac
from cryptography.hazmat.primitives.kdf.pbkdf2 import PBKDF2HMAC
from cryptography.hazmat.primitives.ciphers.aead import AESGCM
from cryptography.hazmat.primitives.asymmetric import ed25519
from cryptography.exceptions import InvalidKey
import json
```

```
import secrets
import time
from typing import Dict, Optional

class MessageValidator:
    def __init__(self):
        """Ініціалізація валідатора повідомлень"""
        self.invalid_message_counts: Dict[str, int] = {}  # sender -> count
        self.last_notification_time: Dict[str, float] = {}  # sender -> timestamp
        self.notification_threshold = 10  # повідомляти кожні N неправильних повідомлень
        self.notification_cooldown = 10.0  # мінімальний час між повідомленнями в секундах

    def record_invalid_message(self, sender_id: str) -> Optional[int]:
        """
        Записати неправильне повідомлення та повернути кількість неправильних повідомлень,
        якщо досягнуто поріг для сповіщення, інакше None
        """
        current_time = time.time()

        # Ініціалізація лічильників для нового відправника
        if sender_id not in self.invalid_message_counts:
            self.invalid_message_counts[sender_id] = 0
            self.last_notification_time[sender_id] = 0

        # Збільшуємо лічильник
        self.invalid_message_counts[sender_id] += 1
        count = self.invalid_message_counts[sender_id]
```

```
        # Перевіряємо чи потрібно показати сповіщення
        if (count >= self.notification_threshold and
            count % self.notification_threshold == 0 and
            current_time - self.last_notification_time[sender_id] >=
self.notification_cooldown):

            self.last_notification_time[sender_id] = current_time
            return count

        return None

class CryptoManager:
    PBKDF2_ITERATIONS = 300000  # Кількість ітерацій для захисту від перебору паролів
    MAX_PADDING = 128           # Максимальна кількість байтів випадкової вставки

    def __init__(self, passphrase: str):
        """Ініціалізація менеджера криптографії з паролем"""
        self.passphrase = passphrase.encode('utf-8')
        self.validator = MessageValidator()

        # Генерація солі для ключа підпису
        digest = hashes.Hash(hashes.SHA256())
        digest.update(b"signing_key_salt")  # Унікальний контекст для ключа підпису
        digest.update(self.passphrase)      # Додавання паролю користувача
        signing_salt = digest.finalize()    # 32-байтова сіль
```

```
        # Генерація seed для Ed25519 з паролю
        kdf = PBKDF2HMAC(
            algorithm=hashes.SHA256(),
            length=32,  # Розмір ключа Ed25519
            salt=signing_salt,  # Унікальна сіль для кожного паролю
            iterations=self.PBKDF2_ITERATIONS,
        )
        signing_seed = kdf.derive(self.passphrase)

        # Створення ключової пари Ed25519
        self.signing_key =
ed25519.Ed25519PrivateKey.from_private_bytes(signing_seed)
        self.verify_key = self.signing_key.public_key()

        # Генерація ключа для HMAC
        hmac_digest = hashes.Hash(hashes.SHA256())
        hmac_digest.update(b"hmac_key_salt")  # Унікальний контекст для HMAC
        hmac_digest.update(self.passphrase)
        hmac_salt = hmac_digest.finalize()

        kdf = PBKDF2HMAC(
            algorithm=hashes.SHA256(),
            length=32,
            salt=hmac_salt,
            iterations=self.PBKDF2_ITERATIONS,
        )
        self.hmac_key = kdf.derive(self.passphrase)

    def derive_key(self, salt: bytes) -> bytes:
        """Отримати ключ шифрування з паролю та солі через PBKDF2"""
```

```
        kdf = PBKDF2HMAC(
            algorithm=hashes.SHA256(),
            length=32,
            salt=salt,
            iterations=self.PBKDF2_ITERATIONS,
        )
        return kdf.derive(self.passphrase)

    def _add_padding(self, data: bytes) -> bytes:
        """Додає випадковий паддінг для маскування реальної довжини повідомлення"""
        pad_len = secrets.randbelow(self.MAX_PADDING + 1)  # 0..128
        padding = os.urandom(pad_len)
        return bytes([pad_len]) + data + padding

    def _strip_padding(self, padded: bytes) -> bytes:
        """Видаляє паддінг та повертає оригінальне повідомлення"""
        pad_len = padded[0]
        if pad_len:
            return padded[1:-pad_len]
        return padded[1:]

    def _create_hmac(self, data: bytes, nonce: str, timestamp: float) -> bytes:
        """Створити HMAC для повідомлення"""
        h = hmac.HMAC(self.hmac_key, hashes.SHA256())
        h.update(data)
        h.update(nonce.encode('utf-8'))
        h.update(str(timestamp).encode('utf-8'))
        return h.finalize()
```

```
    def _verify_hmac(self, data: bytes, nonce: str, timestamp: float, hmac_value: bytes)
-> bool:
        """Перевірити HMAC повідомлення"""
        try:
            h = hmac.HMAC(self.hmac_key, hashes.SHA256())
            h.update(data)
            h.update(nonce.encode('utf-8'))
            h.update(str(timestamp).encode('utf-8'))
            h.verify(hmac_value)
            return True
        except InvalidKey:
            return False

    def validate_message(self, encrypted_package_str: str, sender_id: str =
"unknown") -> tuple[bool, Optional[int]]:
        """
        Перевірити валідність повідомлення.
        Повертає (is_valid, invalid_count), де invalid_count - кількість неправильних
повідомлень
        для сповіщення, якщо досягнуто поріг, інакше None
        """
        try:
            # Декодуємо пакет
            encrypted_package                                                    =
json.loads(b64decode(encrypted_package_str).decode('utf-8'))

            # Декодуємо base64 компоненти
            ciphertext = b64decode(encrypted_package['ciphertext'])
            signature = b64decode(encrypted_package['signature'])
            hmac_value = b64decode(encrypted_package['hmac'])
```

```
            nonce = encrypted_package['nonce']
            timestamp = encrypted_package['timestamp']

            # Перевіряємо час
            if abs(time.time() - timestamp) > 3600:  # 1 година
                return False, self.validator.record_invalid_message(sender_id)

            # Перевіряємо HMAC
            if not self._verify_hmac(ciphertext, nonce, timestamp, hmac_value):
                return False, self.validator.record_invalid_message(sender_id)

            # Перевіряємо підпис
            try:
                self.verify_key.verify(signature, ciphertext)
            except Exception:
                return False, self.validator.record_invalid_message(sender_id)

            return True, None

        except Exception:
            return False, self.validator.record_invalid_message(sender_id)

    def decrypt_message(self, encrypted_package_str: str, sender_id: str = "unknown")
-> Optional[str]:
        """
        Розшифрувати повідомлення з зашифрованого пакету.
        Повертає розшифроване повідомлення або None якщо повідомлення
невалідне.
        """
```

```
        is_valid, invalid_count = self.validate_message(encrypted_package_str, sender_id)
        if not is_valid:
            return None

        try:
            # Декодуємо пакет
            encrypted_package = json.loads(b64decode(encrypted_package_str).decode('utf-8'))

            # Декодуємо base64 компоненти
            ciphertext = b64decode(encrypted_package['ciphertext'])
            iv = b64decode(encrypted_package['iv'])
            salt = b64decode(encrypted_package['salt'])

            # Отримуємо ключ та розшифровуємо повідомлення
            key = self.derive_key(salt)
            aesgcm = AESGCM(key)
            padded_plaintext = aesgcm.decrypt(iv, ciphertext, None)
            plaintext_bytes = self._strip_padding(padded_plaintext)

            return plaintext_bytes.decode('utf-8')

        except Exception:
            return None

    def encrypt_message(self, message: str) -> str:
        """Зашифрувати повідомлення та повернути єдиний зашифрований пакет"""
        # Генеруємо нову сіль та IV для кожного повідомлення
```

```
salt = os.urandom(16)
iv = os.urandom(12)

# Генеруємо nonce та timestamp
nonce = secrets.token_hex(16)
timestamp = time.time()

# Отримуємо ключ шифрування та створюємо AESGCM cipher
key = self.derive_key(salt)
aesgcm = AESGCM(key)

# Шифруємо повідомлення з паддінгом
message_bytes = self._add_padding(message.encode('utf-8'))
ciphertext = aesgcm.encrypt(iv, message_bytes, None)

# Створюємо цифровий підпис
signature = self.signing_key.sign(ciphertext)

# Створюємо HMAC
hmac_value = self._create_hmac(ciphertext, nonce, timestamp)

# Пакуємо всі деталі шифрування
encrypted_package = {
    'ciphertext': b64encode(ciphertext).decode('utf-8'),
    'iv': b64encode(iv).decode('utf-8'),
    'salt': b64encode(salt).decode('utf-8'),
    'signature': b64encode(signature).decode('utf-8'),
    'hmac': b64encode(hmac_value).decode('utf-8'),
    'nonce': nonce,
    'timestamp': timestamp
```

```
    }

    return b64encode(json.dumps(encrypted_package).encode('utf-8')).decode('utf-
8')
```