НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ БІОРЕСУРСІВ
І ПРИРОДОКОРИСТУВАННЯ УКРАЇНИ

Факультет (ННІ) ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ

ПОГОДЖЕНО	**ДОПУСКАЄТЬСЯ ДО ЗАХИСТУ**
Декан факультету (Директор ННІ)	**Завідувач кафедри**
Інформаційних технологій	Комп'ютерних систем, мереж та кібербезпеки
__________ Болбот І.М., д.т.н, проф.	__________ Касаткін Д.Ю., к. пед.н, доц.
підпис ПІБ, вчене звання і ступінь	підпис ПІБ, вчене звання і ступінь
«___»___________2025 р.	«___»___________2025 р.

МАГІСТЕРСЬКА КВАЛІФІКАЦІЙНА РОБОТА

На тему: «Дослідження програмних засобів системи розумного будинку з додатковими функціями безпеки»

Спеціальність: 123 «Комп’ютерна інженерія»

Освітня програма: Комп’ютерні системи та мережі

Орієнтація освітньої програми: Освітньо-професійна

Гарант освітньої програми

д.т.н., доцент	____________	Шкарупило В.В.
(науковий ступінь та вчене звання)	(підпис)	(ПІБ)

Керівник магістерської кваліфікаційної роботи

д.т.н., професор	____________	Коваленко О.Є.
(науковий ступінь та вчене звання)	(підпис)	(ПІБ)

Виконав ____________________ Українець Дмитро Сергійович

(підпис) (ПІБ студента)

КИЇВ – 2025

НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ БІОРЕСУРСІВ
І ПРИРОДОКОРИСТУВАННЯ УКРАЇНИ

Факультет (ННІ) ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ

«ЗАТВЕРДЖУЮ»
завідувач кафедри
комп'ютерних систем, мереж та кібербезпеки
_____________ / Касаткін Д.Ю., к.пед.н., доц. /
підпис ПІБ, вчене звання і ступінь
«__» ___________ 2025 р.

З А В Д А Н Н Я

ДО ВИКОНАННЯ МАГІСТЕРСЬКОЇ КВАЛІФІКАЦІЙНОЇ РОБОТИ СТУДЕНТУ

Українець Дмитро Сергійович
(прізвище, ім'я, по батькові)

Спеціальність 123 «Комп'ютерна інженерія»
(код і назва)

Освітня програма Комп'ютерні системи та мережі
(назва)

Орієнтація освітньої програми Освітньо-професійна
(освітньо-професійна або освітньо-наукова)

Тема магістерської кваліфікаційної роботи: «Дослідження програмних засобів системи розумного будинку з додатковими функціями безпеки»

затверджена наказом ректора НУБіП України від " 29 " жовтня 2024 р. № 1941'С'

Термін подання завершеної роботи на кафедру 13 листопада 2025 р.
(рік, місяць, число)

Вихідні дані до магістерської кваліфікаційної роботи середовище розробки Visual Studio; мова програмування C#; мова програмування Python (з крипто-бібліотеками); фреймворк Windows Forms; база даних SQLite/SQL Server; протоколи та алгоритми – MQTT, TLS, AES, RSA, TOTP, Argon2.

Перелік питань, що підлягають дослідженню:

1. Програмні засоби систем розумного будинку, аналіз сучасних методів захисту інформації в IoT та огляд попередніх досліджень.
2. Методологія емпіричного дослідження, аналіз продуктивності та доцільності застосування програмних засобів захисту інформації.
3. Розроблення програмного забезпечення для системи розумного будинку з інтегрованими, збалансованими механізмами безпеки.

Перелік графічного матеріалу (за потреби) ____________________

Дата видачі завдання " 29 " жовтня 2024 р.

Керівник магістерської кваліфікаційної роботи ________ Коваленко О.Є.
(підпис) (прізвище та ініціали)

Завдання прийняв до виконання ________ Українець Д.С.
(підпис) (прізвище та ініціали студента)

РЕФЕРАТ

Пояснювальна записка: 80 сторінок, 19 рисунків, 3 таблиці, 19 лістингів коду, 3 додатки, 24 джерел.

ІНТЕРНЕТ РЕЧЕЙ, РОЗУМНИЙ БУДИНОК, ІНФОРМАЦІЙНА БЕЗПЕКА, ПРОТОКОЛИ ПЕРЕДАЧІ ДАНИХ, ШИФРУВАННЯ, АВТЕНТИФІКАЦІЯ.

Мета дослідження – проведення дослідження та аналізу продуктивності і доцільності застосування сучасних програмних засобів захисту інформації, з метою формування практичних рекомендацій щодо їх збалансованого поєднання в системах розумного будинку.

Об’єкт дослідження – процес забезпечення інформаційної безпеки в системі розумний будинок.

Предмет дослідження – програмні методи та протоколи захисту інформації та їх продуктивність на пристроях з обмеженими ресурсами.

Робота складається з наступних трьох розділів:

У першому розділі досліджено теоретичні основи, що охоплюють концепції та архітектури систем «розумний будинок», проаналізовано програмні засоби та протоколи передачі даних, які використовуються в системах IoT.

Другий розділ присвячено аналітичному дослідженню програмних методів захисту. Він обґрунтовує методологію дослідження та демонструє результати емпіричного тестування методів шифрування, автентифікації та протоколів зв’язку в системах IoT. На основі аналізу отриманих результатів сформульовано практичні висновки та постановку завдань для подальшої реалізації програми.

Третій розділ є практичною частиною роботи. У ньому представлено розроблену архітектуру програми, включаючи структуру проєкту, бази даних та модель потоків даних MQTT. Описано реалізацію основних функціональних модулів та впровадження механізмів безпеки.

ЗМІСТ

ВСТУП 6
1 ТЕОРЕТИЧНІ ОСНОВИ 7
1.1. Концепція та архітектура систем розумний будинок 7
1.2. Програмні засоби у системах IoT 10
1.3. Протоколи передачі даних 12
1.4. Проблематика інформаційної безпеки 15
1.5. Методи та засоби забезпечення безпеки 18
2 АНАЛІТИЧНЕ ДОСЛІДЖЕННЯ ПРОГРАМНИХ МЕТОДІВ ЗАХИСТУ 22
2.1. Методологія дослідження 22
2.2. Емпіричне дослідження методів захисту в системах IoT 24
2.2.1. Тестування методів шифрування 24
2.2.2. Тестування методів автентифікації 29
2.2.3. Тестування протоколів зв'язку 33
2.3. Аналіз результатів тестування 38
2.4. Вихідні практичні висновки та рекомендації на основі тестів 50
2.5. Постановка завдань для практичної реалізації 53
3 ПРАКТИЧНА ЧАСТИНА 55
3.1. Архітектура програми 56
3.1.1. Загальна структура проєкту 56
3.1.2. Структура бази даних 60
3.1.3. Архітектура потоків даних MQTT 62
3.2. Реалізація основних функціональних модулів програми 64
3.2.1. Модуль інтерфейсу користувача 64
3.2.2. Модуль роботи з базою даних 66
3.2.3. Модуль взаємодії з MQTT 68
3.3. Реалізація основних механізмів безпеки та їх аналіз 70
3.3.1. Реалізація механізмів автентифікації та контролю доступу 70
3.3.2. Реалізація гібридної схеми шифрування даних 72
3.3.3. Тестування та аналіз продуктивності 74
ВИСНОВКИ 76

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ 78
ДОДАТОК А ДОДАТКОВІ РЕЗУЛЬТАТИ ТЕСТУВАННЯ ПРОДУКТИВНОСТІ ПРОТОКОЛУ MQTT 81
ДОДАТОК Б ПОСИЛАННЯ НА РЕПОЗИТОРІЇ (QR-КОДИ) 82
ДОДАТОК В ДОПОМІЖНИЙ ІЛЮСТРАТИВНИЙ МАТЕРІАЛ 83

ВСТУП

Сучасний етап розвитку інформаційних технологій характеризується стрімким поширенням концепції Інтернету речей, яка знаходить широке застосування у різних сферах людської діяльності. Одним з важливих напрямків цієї концепції стали системи розумного будинку. Вони інтегрують різні електронні пристрої такі як датчики, камери, виконавчі механізми тощо, у єдину програмно-керовану екосистему.

Загальна ефективність таких систем значною мірою визначається якістю програмних засобів моніторингу, керування та взаємодії з користувачем. Проте однією з актуальних проблем таких систем залишається їхня стійкість до загроз інформаційної безпеки. Кожен підключений пристрій стає потенційною точкою входу для зловмисника та створює вразливість, відкриваючи доступ до приватних даних користувачів та критичної інфраструктури будинку. Ключова проблема полягає в тому, що традиційні корпоративні методи захисту є непридатними для більшості IoT-пристроїв через їхні жорсткі апаратні обмеження: низьку обчислювальну потужність, малий обсяг пам’яті та критичні вимоги до енергоефективності.

Тому актуальність теми дослідження зумовлена необхідністю аналізу програмних засобів захисту інформації, що забезпечують функціонування розумного будинку, з урахуванням їхніх можливостей та вразливостей у сфері інформаційної безпеки. Багато сучасних рішень переважно орієнтовані на функціональність і зручність використання, тоді як питання захисту даних та протидії кіберзагрозам залишаються другорядними. Це створює потенційні ризики як для кінцевих користувачів, так і для ширших інформаційних екосистем.

1 ТЕОРЕТИЧНІ ОСНОВИ

1.1. Концепція та архітектура систем розумний будинок

Концепція «розумного будинку» (Smart Home) є однією з найбільш перспективних сфер, напрямків, розвитку Інтернету речей (Internet of Things, надалі IoT). Передусім це поєднання апаратного та програмного забезпечення, тобто передбачає інтеграцію фізичних пристроїв, сенсорів, виконавчих механізмів та програмного забезпечення в єдину автоматизовану систему управління. Основною метою таких систем є підвищення рівня комфорту, енергоефективності та безпеки за допомогою технологій автоматизації, моніторингу та віддаленого керування.

Розглядаючи сучасну архітектуру розумного будинку (системи IoT) зазвичай складається з кількох основних компонентів: сенсорів, виконавчих пристроїв, контролерів, комунікаційних мереж та інтерфейсів взаємодії з користувачем. Приклад деякої сучасної архітектури системи IoT наведено на рисунку 1.1. Сенсори здійснюють збір даних про навколишнє середовище (температура, вологість, освітленість, рух), виконавчі пристрої виконують команди (вмикання чи вимикання освітлення, регулювання клімату, замикання дверей тощо), контролери здійснюють обробку даних та прийняття рішень, комунікаційні мережі забезпечують передачу інформації між компонентами системи, а інтерфейси взаємодії з користувачем (це можуть бути мобільні додатки, голосові асистенти, веб-інтерфейси чи прикладні програми) дозволяють здійснювати моніторинг, керування системою, забезпечують зворотній зв’язок з користувачем.

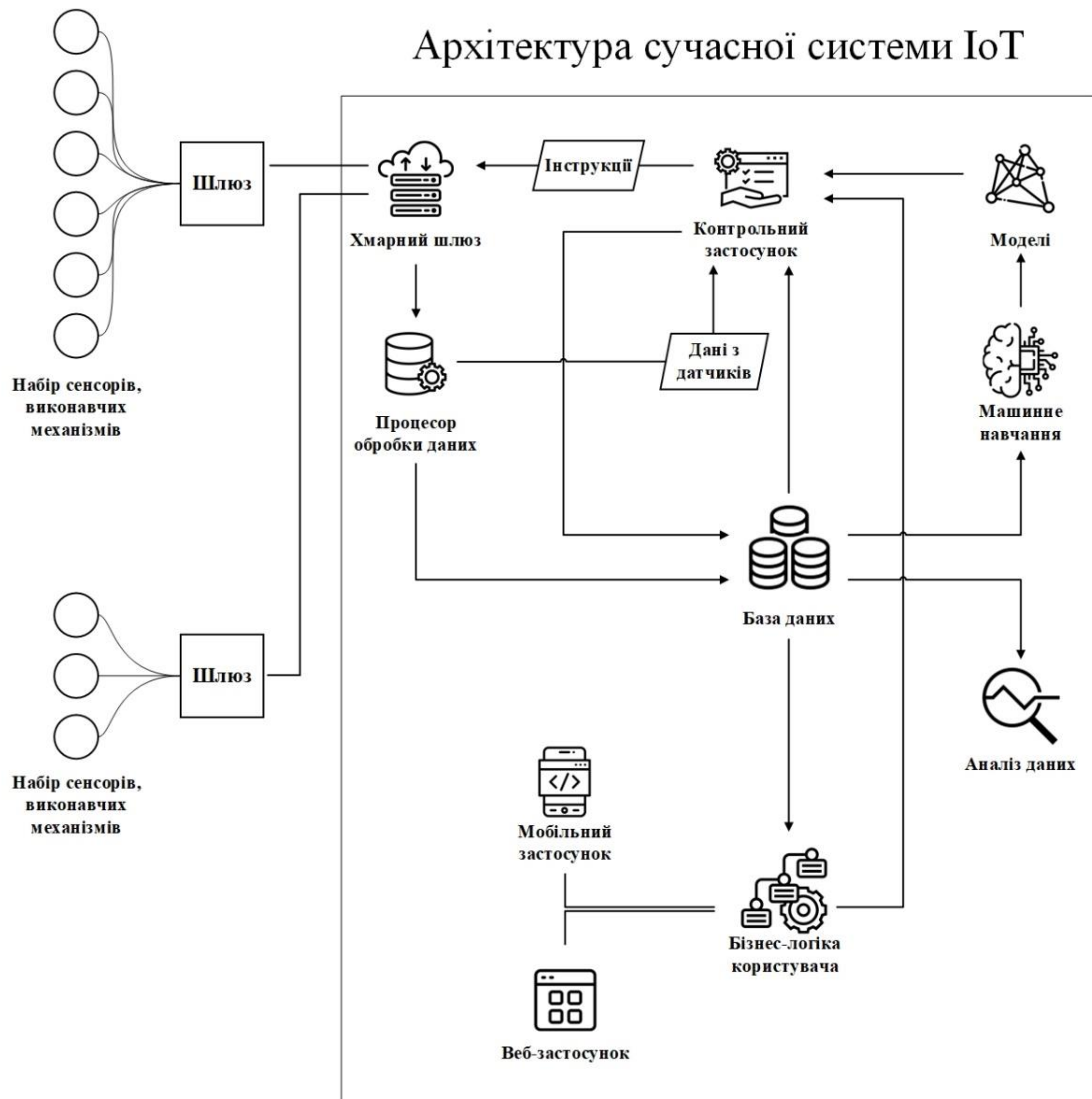

Рисунок 1.1 – Ілюстративний приклад сучасної архітектури системи IoT

Однією з ключових особливостей таких систем є їх здатність до контекстного сприйняття. Контекстне сприйняття – це коли система може адаптувати свою поведінку залежно від змін у навколишньому середовищі або діяльності користувачів. Наприклад, система може автоматично регулювати температуру в кімнаті залежно від часу доби, або вмикати освітлення при виявленні руху в приміщенні. Для реалізації таких функцій використовуються різноманітні сенсори та алгоритми обробки даних, що дозволяють системі оцінювати поточний контекст і відповідно реагувати на нього.

Протоколи передачі даних відіграють важливу роль в таких системах. Адже протоколи такі як MQTT, CoAP, Zigbee, Z-Wave, Wi-Fi та Bluetooth,

забезпечують комунікацію між компонентами системи. Вибір протоколу залежить від вимог до енергоспоживання, дальності зв'язку, швидкості передачі даних та сумісності з іншими пристроями. Наприклад, протокол Zigbee та Z-Wave є енергоефективними протоколами, тому використовуються для зв'язку між сенсорами та виконавчими пристроями, тоді як Wi-Fi та Bluetooth забезпечують вже з'єднання з інтерфейсами взаємодії з користувачем [1].

Інтеграція хмарних технологій у архітектуру розумного будинку є перспективним рішенням, яке дозволяє забезпечити віддалений доступ до системи, зберігання та обробку великих обсягів даних, а також реалізацію складних сценаріїв автоматизації. Хмарні платформи надають можливість централізованого управління та моніторингу, а також інтеграції з іншими системами та сервісами, що розширює функціональні можливості розумного будинку [2].

Розрізняють два різних підходи до організації обробки даних і управління пристроями – централізовані та децентралізовані архітектури. У централізованих системах ключові функції відповідно зосереджені на центральному сервері або хмарній платформі, що забезпечує єдину точку контролю, узгоджене управління, масштабовану аналітику та інтеграцію з більшістю бізнес-процесами. Такий підхід спрощує адміністрування, проте очевидно створює пряму залежність від доступності мережі та підвищує ризики, пов’язані з відмовостійкістю та кібербезпекою. З іншого боку, децентралізовані системи базуються на принципі розподілу обчислювальних і керуючих функцій між локальними вузлами, що дозволяє зменшити затримку, підвищити автономність пристроїв та мінімізувати обсяг даних, які передаються у хмару. Водночас децентралізовані рішення вимагають складніших алгоритмів синхронізації та безпеки, оскільки розподілені вузли повинні узгоджено функціонувати у середовищах. Вибір ж між централізованою та децентралізованою архітектурою визначається балансом між вимогами до продуктивності, безпеки, гнучкості та ресурсної ефективності.

Добре спроектована архітектура системи розумного будинку повинна бути гнучкою та масштабованою, щоб забезпечити можливість додавання нових пристроїв та функцій без значних змін у структурі системи. Це досягається за рахунок використання модульного підходу до проектування, стандартизованих інтерфейсів та протоколів, а також відкритих платформ та стандартів.

Сучасні системи розумного будинку є складними багаторівневими структурами, вони поєднують у собі апаратні та програмні компоненти, комунікаційні мережі та інтерфейси взаємодії з користувачем. Вони здатні адаптувати свою поведінку залежно від контексту, забезпечувати віддалене керування та інтеграцію з іншими системами, що робить їх потужними інструментами автоматизації деякого простору для якого планується їх впровадження.

1.2. Програмні засоби у системах IoT

Як вже було сказано вище, системи IoT це поєднання апаратних та програмних засобів. А розвиток та впровадження таких систем значною мірою залежать від багатогранного набору програмних компонентів, які забезпечують взаємодію пристроїв, обробку даних, інтерфейси для користувачів та загальну сумісність систем. Такі програмні рішення можна класифікувати на чотири основні категорії: хмарні платформи, мобільні додатки, програмне забезпечення шлюзів та локальні контролери (або хаби), кожна з яких виконує свої специфічні, але взаємодоповнюючі функції в екосистемі IoT. Далі наведений огляд кожної класифікації окремо.

Хмарні платформи є центральними компонентами сучасних IoT-екосистем, оскільки вони добре забезпечують масштабованість, аналітику та централізоване управління пристроями. Зокрема, платформи типу Microsoft Azure IoT, AWS IoT Core, Google Cloud IoT та IBM Watson IoT підтримують надійну комунікацію між пристроями та хмарою, шифрування, аутентифікацію,

віддалене керування та аналітику на основі машинного навчання, що дозволяє прогнозувати події й оптимізувати роботу системи в цілому [3].

Мобільні додатки виконують роль основного інтерфейсу для кінцевого користувача. Вони подаються як зручні та інтуїтивні інструменти які надають можливість дистанційного моніторингу, конфігурації та управління пристроями. Існують такі сучасні рішення які використовують мобільні додатки, як-от Google Home, Apple HomeKit, Samsung SmartThings. Такі рішення дозволяють реалізувати голосове керування, створення сценаріїв автоматизації, групування пристроїв, отримання сповіщень – загалом усе це робить взаємодію інтуїтивно зрозумілою й доступною для користувачів мало знайомих з цією концепцію.

Програмне забезпечення шлюзу (з англ. мови gateway software) виконує важливу роль мосту між обмеженими ресурсами пристроїв IoT та хмарними або локальними мережевими компонентами. Стає очевидним що щлюзи належать до проміжного рівня архітектури й реалізують переклад протоколів, агрегацію й фільтрацію даних, обробку на периферії, зберігання даних у разі втрати з’єднання, локальне керування пристроями з мінімальною затримкою, тощо, усе це критично для чутливих до часу сценаріїв, наприклад, у промисловості чи охороні здоров’я [4][5]. Більшість шлюзів мають підтримку таких розповсюджених протоколів MQTT, CoAP, Zigbee, що гарантує сумісність пристроїв різних виробників. Також зростає популярність гібридних шлюзів, які поєднують програмну логіку й периферійні обчислювальні потужності, зокрема це Azure IoT Edge, що дозволяє запускати контейнери з логікою або AI-модулями безпосередньо на пристроях [6].

Локальні контролери або хаби забезпечують автономну роботу системи навіть при відсутності доступу до хмари, сприяють конфіденційності та підтримці складних сценаріїв автоматизації. Представники таких відкритих платформ це Home Assistant та OpenHAB – вони надають розширені можливості інтеграції, сумісності і локального контролю, суміщаючи різні протоколи зв’язку в єдиному інтерфейсі, що можна адаптувати під потреби користувачів.

Розглянувши та синтезуючи ці аспекти, можна зазначити, що програмні засоби в системах IoT утворюють складну, багаторівневу екосистему, в якій хмарні сервіси забезпечують масштабовану аналітику й керування, шлюзи й edge-платформи – швидку реакцію й оптимізацію даних, а локальні хаби – автономність і конфіденційність. Мобільні інтерфейси забезпечують зручність користувацької взаємодії та інтуїтивність а платформи управління пристроями – безперебійну експлуатацію та безпеку на масовому рівні. Така структурована гнучкість забезпечує широку адаптивність рішень для різноманітних сценаріїв використання – починаючи від домашньої автоматизації та закінчуючи промисловими і міськими IoT-мережами.

1.3. Протоколи передачі даних

Так як протоколи передачі даних відіграють ключову, можна сказати фундаментальну, роль у системах IoT, бо саме вони формують основу для ефективної взаємодії між усіма компонентами таких систем – варто розглянути їх окремо. Правильно обраний протокол забезпечує ефективність комунікації, оптимальне використання ресурсів, масштабованість системи а також належний рівень інформаційної безпеки. Окремо варто зазначити – середовище IoT характеризується різнорідністю пристроїв, воно може включати сенсори та виконавчі механізми з обмеженими обчислювальними можливостями або високопродуктивні контролери чи сервери. Тому це обумовлює існування широкого спектра протоколів, які відрізняються за архітектурою, принципами роботи, рівнем енергоспоживання та вимогами до мережі.

Загалом протоколи IoT можна поділити на дві великі категорії: протоколи мережевої комунікації та прикладні протоколи. Де перші забезпечують базову можливість передавання пакетів у мережі (до них можна віднести: Wi-Fi, Bluetooth Low Energy, Zigbee, Z-Wave, Thread тощо), тоді як другі визначають

правила організації діалогу між пристроями та сервісами (це: MQTT, CoAP, AMQP, HTTP/HTTPS).

Зазначу що у межах даного дослідження основну увагу буде зосереджено на аналізі мережевих та прикладних протоколів, як ключових компонентів комунікаційної архітектури IoT-систем, що забезпечують передачу даних між пристроями та інтеграцію з хмарними сервісами. Можна провести аналогію та сказати що модель протоколів, що використовується в системах IoT, структурно нагадує еталонну мережеву модель OSI, оскільки вона також передбачає ієрархічний поділ на функціональні рівні, де для даної моделі будуть: фізичні та канальні протоколи, мережеві, транспортні та прикладні. Узагальнена порівняльна характеристика наведена в таблиці 1.1.

Таблиця 1.1 – Порівняльна характеристика протоколів обміну даних в IoT

Протокол	Тип	Переваги	Обмеження
MQTT	Прикладний	«Легкість», модель pub-sub, ефективність при низькій пропускній здатності	Потребує брокера
CoAP	Прикладний	Оптимізація для обмежених пристроїв, підтримка режиму спостереження	Базується на UDP, менш поширений
AMQP	Прикладний	Гарантована доставка, складна маршрутизація	Велика ресурсоємність і складність
HTTP/ HTTPS	Прикладний	Універсальність, підтримка REST API, безпека (HTTPS)	Високі накладні витрати
Zigbee	Мережевий	Низьке енергоспоживання, топологія mesh	Обмежена швидкість

Продовження таблиці 1.1

Протокол	Тип	Переваги	Обмеження
Z-Wave	Мережевий	Сумісність, стабільна робота	Ліцензований, нижча швидкість порівняно з Zigbee
BLE	Мережевий	Дуже низьке енергоспоживання, простота	Коротка відстань передачі
Thread	Мережевий	Відкритий стандарт, безпека	Потребує сумісних пристроїв

Нижче наведено детальніший огляд поширених протоколів обміну даних:

Протокол «Message Queuing Telemetry Transport» (MQTT) є одним з популярних що пояснюється винятковою простотою та «легким», в тому контексті що робить його оптимальним для сенсорів і пристроїв з мінімальним енергоспоживанням. MQTT функціонує за моделлю «публікація-підписка», де всі повідомлення проходять через брокера (модель обміну повідомленнями), що виконує роль посередника. Умовно, якщо провести аналогію для кращого розуміння, то брокер можна порівняти з «поштовим відділенням»: клієнт відправляє повідомлення, а всі підписники отримують його у зручний для них момент. Саме така організація дозволяє підтримувати стабільність роботи навіть за нестабільного з'єднання та масштабувати систему на сотні й навіть тисячі вузлів.

Наступний протокол – Constrained Application Protocol (CoAP), він створений спеціально для пристроїв із обмеженими ресурсами. Ґрунтується на UDP (User Datagram Protocol, протокол який в першу чергу ставить швидкість а не надійність), що забезпечує низьку затримку, і за структурою нагадує HTTP, але набагато спрощений і адаптований до середовищ із обмеженою пропускною здатністю. Важливою особливістю CoAP є підтримка асинхронної комунікації та

механізму «спостереження» (від слова observe), який дозволяє клієнтам підписуватися на оновлення з датчика. Наприклад, у системі розумного будинку датчик якості повітря може автоматично надсилати повідомлення контролеру про перевищення рівня вуглекислого газу, що дає змогу миттєво активувати вентиляцію. Тобто, CoAP ідеально підходить для сценаріїв, де важлива швидка реакція.

HTTP/HTTPS зберігають значну роль завдяки своїй універсальності та широкій підтримці у вебсередовищі. Хоча вони створюють більші накладні витрати та не завжди оптимальні для слабких пристроїв, протоколи залишаються актуальними у випадках, коли необхідна інтеграція IoT із вебсервісами чи іншими API.

Bluetooth Low Energy (BLE), це мережевий протокл ближнього радіусу дії, варто відзначити що він забезпечує дуже низьке енергоспоживання і є базовим стандартом для smart-аксесуарів та компактних сенсорних систем. Zigbee та Z-Wave спеціалізуються на автоматизації житла, підтримуючи топологію mesh, яка дозволяє кожному пристрою виступати вузлом для ретрансляції сигналу й тим самим розширювати мережу. Існує також відносно новий протокол Thread орієнтований на розумні будинки, який поєднує mesh-архітектуру з підвищеною увагою до безпеки та відкритого стандарту.

1.4. Проблематика інформаційної безпеки

Зі зростанням кількості підключених пристроїв можна сформувати гіпотезу, що ризики кіберзагроз у системах IoT зростають пропорційно. Так, згідно з дослідження, кількість атак на IoT-системи лише у 2022 році перевищила 112 мільйонів випадків, що на 87% більше, ніж роком раніше [7]. Забігаючи наперед можна сказати що така динаміка пояснюється не лише збільшенням числа підключених пристроїв, що підтверджує гіпотизу, але й низьким рівнем захищеності домашніх мереж, недостатнім використанням складних паролів,

відсутністю шифрування трафіку, ненадійними інтерфейсами, відкритими портами та слабким контролем доступу.

Серед найпоширеніших загроз для розумних будинків є несанкціонований доступ до пристроїв, перехоплення даних, підміна інформації та атаки на відмову в обслуговуванні [8]. А основним підґрунтям для виникнення цих проблем часто лежить недотримання елементарних правил безпеки користувачами. Типовим прикладом є використання заводських паролів на кшталт «admin», що відкриває шлях зловмисникам до повного контролю над пристроями.

Серед ключових загроз для IoT варто виділити кілька категорій. Однією з найнебезпечніших є атаки типу «людина посередині» (man-in-the-middle), це коли зловмисник перехоплює трафік між пристроями чи між пристроєм і сервером. У результаті він може не лише отримати доступ до конфіденційних даних, але й змінювати їх без відома користувача. Найчастіше такі атаки здійснюються через відкриті або слабко захищені Wi-Fi-мережі, які нерідко використовуються в побутових, буденних, умовах.

Ще одним серйозним ризиком є атаки на відмову в обслуговуванні (DoS - Denial-of-Service). У цьому випадку система перевантажується запитами і перестає виконувати свої функції. Для розумного будинку це означає відсутність реакції на базові команди: не вмикається освітлення, не спрацьовує сигналізація, блокуються дверні замки. У критичних сценаріях наслідки можуть створювати безпосередню загрозу безпеці користувачів.

Наступною і не менш небезпечною є вразливість, пов’язана з перепрошиванням пристроїв. Зловмисники можуть завантажувати або видавати за офіційне (фішинг) змінене програмне забезпечення, яке дозволяє їм отримати контроль над обладнанням або включити його ресурси до ботнет-мереж. На жаль, значна частина IoT-пристроїв або взагалі не підтримує оновлення, або оновлюється нерегулярно, що створює вразливе місце в архітектурі безпеки.

Окрему групу ризиків становлять програмні вразливості у веб-інтерфейсах та мобільних додатках для керування системою (API). Недоліки в цих компонентах дають змогу обходити автентифікацію, отримувати доступ до

функціоналу пристроїв та керувати ними віддалено. Приклад такої атаки це може бути SQL-ін'єкція в ході якої зловмисник може отримати доступ до системи. Навіть офіційні додатки можуть стати точкою входу для атак.

Також не слід недооцінювати і роль людського фактору в цьому питанні. Користувачі часто нехтують правилами безпеки: використовують однакові паролі для різних пристроїв, зберігають облікові дані у незахищених місцях або підключають пристрої до публічних Wi-Fi-мереж. Значною загрозою залишаються фішингові атаки, коли зловмисники розсилають підроблені повідомлення з вимогою встановити «оновлення», які фактично є шкідливим програмним забезпеченням. Користувачем таких систем може виступати як технічно підготовлений фахівець, так і особа без попереднього досвіду.

Узагальнюючи, усі загрози можна поділити на внутрішні та зовнішні. Внутрішні виникають унаслідок помилок користувача, некоректних налаштувань чи дефектів у програмному забезпеченні. Зовнішні загрози формуються внаслідок дій зловмисників, спрямованих на викрадення даних, порушення роботи або отримання доступу до інфраструктури. Але в обох випадках наслідки можуть бути критичними для функціонування системи розумного будинку.

Виходячи з всього вище згаданого – проблематика інформаційної безпеки в IoT є багатовимірною і охоплює як технічні, так і організаційні аспекти. Захист має будуватися комплексно – починаючи від апаратного рівня та прошивки пристроїв, і завершуючи політикою доступу та поведінкою користувачів. Лише інтеграція механізмів автентифікації, шифрування, централізованого моніторингу та постійного оновлення системи здатна суттєво знизити ризики. Розуміння різних категорій загроз та їхньої динаміки є необхідною умовою для створення ефективних стратегій безпеки у середовищі розумного будинку [8].

1.5. Методи та засоби забезпечення безпеки

У попередньому підрозділі 1.3 було встановлено, що архітектура розумного будинку ґрунтується на різноманітних протоколах передачі даних, які мають власні переваги та недоліки в контексті захищеності. Також розглянута проблематика інформаційної безпеки в підрозділі 1.4 продемонструвала, що зростання кількості підключених пристроїв збільшує поверхню атак, а отже слідом – підвищує ризики для користувачів. Саме тому виникає необхідність у впровадженні комплексних методів і засобів захисту, які б поєднували технічні, програмні та організаційні підходи.

Першим і одним із фундаментальних напрямів який варто розглянути є захист каналів зв’язку між пристроями. Протоколи TLS (Transport Layer Security) та DTLS (Datagram Transport Layer Security) дозволяють гарантувати цілісність та конфіденційність даних навіть у разі перехоплення трафіку. Особливо актуальною є реалізація легковагих, що не потребують великих обчислювальних ресурсів, криптографічних алгоритмів (lightweight cryptography), оптимізованих для сенсорів та пристроїв із низьким енергоспоживанням [9]. Без таких криптографічних алгоритмів навіть найефективніша сучасна архітектура може стати вразливою до таких атак як «людина посередині», яка як вже було з’ясовано несе за собою серйозні наслідки.

Продовжуючи тему шифрування не можна не згадати алгоритми симетричного та асиметричного шифрування. AES (Advanced Encryption Standard) належить до симетричних шифрів і використовується для захисту даних у процесі їх зберігання та передавання. Його перевага полягає у високій швидкості обчислень, що робить його придатним для пристроїв з обмеженими обчислювальними ресурсами. І алгоритм RSA (Rivest–Shamir–Adleman), який на відміну від AES, є асиметричним. Він базується на складності факторизації великих чисел і забезпечує механізм розподілу ключів та цифрового підпису. Водночас, RSA є потребує більше ресурсів, тому його застосування на рівні

шлюзів або хмарних компонентів (сервери), де наявні достатні обчислювальні ресурси.

Інший важливий аспект це механізми автентифікації та контролю доступу. Варто наголосити на різниці між термінами автентифікація та авторизація – де основна мета першого процесу це підтвердження особи (захист від несанкціонованого доступу), а другого ж керування доступом до даних та функцій. Набагато ефективніше застосовувати багатофакторну автентифікацію (MFA – Multi-Factor Authentication), яка поєднує декілька механізмів захисту, наприклад: пароль, біометричні дані чи одноразові токени. Для IoT також набувають поширення сертифікати X.509, які дозволяють підтверджувати справжність пристрою в мережі. Без належного контролю доступу навіть коректно зашифрований трафік не забезпечить захист, адже зловмисник може проникнути в систему через скомпрометований вузол.

Вочевидь оновлення прошивки та програмного забезпечення – базові речі, проте не менш важливі яким також варто приділити належну увагу. Як показали попередні дослідження, значна частина пристроїв не підтримує регулярні оновлення, що створює критичну вразливість [10]. Виробники дедалі частіше впроваджують механізми безпечного оновлення (secure firmware update), які передбачають цифровий підпис оновлень та перевірку їх цілісності перед інсталяцією. Що дозволяє мінімізувати ризик навмисного перепрошивання пристрою зловмисником.

Для наочності у таблиці 1.2 наведено основні категорії методів та засобів безпеки, їх приклад та їхнє практичне застосування у системах розумного будинку.

Таблиця 1.2 – Методи та засоби забезпечення безпеки IoT-систем

Категорія	Приклади методів	Практичне застосування
Шифрування	AES, RSA, TLS/DTLS	Захист даних у мережі та збережених файлів
Автентифікація та доступ	MFA, сертифікати X.509, OAuth	Верифікація пристроїв та користувачів
Оновлення ПЗ	Secure Firmware Update	Запобігання шкідливому перепрошиванню
Моніторинг і виявлення	IDS/IPS, SIEM, anomaly detection	Виявлення вторгнень і аномальної активності
Організаційні заходи	Політики паролів, навчання користувачів	Зменшення ризиків людського фактору

Як видно з таблиці 1.2, безпека в IoT не обмежується лише технологічними рішеннями. Важливу роль відіграють системи моніторингу та виявлення аномалій. Використання систем виявлення та запобігання вторгненням (IDS/IPS, intrusion detection system та prevention system відповідно) дозволяє виявляти підозрілу активність у режимі реального часу. Більш ефективні рішення, наприклад такі як SIEM (Security Information and Event Management), поєднують централізований моніторинг, аналіз журналів і автоматичне реагування на загрози [11]. У сфері розумного будинку такі системи можуть інтегруватися з хмарними сервісами, забезпечуючи виявлення атак навіть на рівні окремих сенсорів.

Деякі протоколи ближнього радіусу дії передбачають вбудований захист, наприклад, стандарт Zigbee передбачає використання AES-128 для шифрування трафіку, тоді як новіший протокол Thread має вбудовану підтримку IPv6 та передбачає більш гнучкі механізми автентифікації. Bluetooth Low Energy у новіших версіях також підтримує захищене з’єднання через Secure Connections,

однак має вразливість до атак націлених на отримання зашифрованих ключів, що змушує розробників впроваджувати додаткові рівні захисту.

Організаційні заходи знижують ризики безпеки та насамперед сприяють формуванню культури безпечного користування. Підрозділ 1.4 підкреслив думку що людський фактор найчастіше стає слабким місцем в питані безпеки, через: використання простих паролів, підключення до відкритих мереж Wi-Fi чи ігнорування оновлень – усе це суттєво знижує рівень захисту. Тому доцільні наступні рішення: створення політики безпеки IoT-пристроїв, реєстрація та інвертиризація пристроїв, розмежування прав доступу, створення простих посібників та навчання персоналу, використання стандартів.

Проведення аудиту безпеки що включає систематичне сканування та тестування системи розумного будинку з метою виявлення потенційних вразливостей та окреслення поверхні атак. Під час аудиту перевіряється: конфігурація пристроїв, протоколи передачі даних, інфраструктура системи, політики доступу, відповідність стандартам та фізичний захист.

Насамкінець у науковій літературі дедалі більше уваги приділяється адаптивним методам безпеки, які базуються на машинному навчанні. Наприклад, алгоритми anomaly detection здатні виявляти нетипові патерни (шаблони) у роботі пристроїв, що може свідчити про атаку ще до її активної фази [12]. У перспективі розвиток таких рішень дозволить створювати системи, що самонавчаються та підлаштовуються під змінювані умови, зо є досі далекосяжним рішенням.

2 АНАЛІТИЧНЕ ДОСЛІДЖЕННЯ ПРОГРАМНИХ МЕТОДІВ ЗАХИСТУ

2.1. Методологія дослідження

Цей розділ присвячений аналітичному дослідженню програмних методів захисту інформації в IoT-середовищу, таке як розумний будинок.

Методологія дослідження розрахована таким чином, аби забезпечити систематичне порівняння криптографічних засобів, механізмів автентифікації та реалізацій захищеної передачі даних. Дослідження відбувається за наступними показниками, це: затрачений час, ресурсоспоживання, продуктивність та стабільність. Умови проведення тесту, наближені до реальної експлуатації середовища розумного будинку.

Основна мета – отримати кількісні та відтворювані характеристики, що візьмуться за базу для практичних рекомендацій щодо вибору і конфігурації засобів захисту.

Методи захисту що досліджуються:

1. Шифрування: симетричний режим AES-GCM (з параметрами 128/256 біт) та асиметричний RSA з OAEP-обгорткою (RSA-OAEP);
2. Автентифікація: одноразові токени TOTP (Time-based One-Time Password) та хеш-функціональні схеми побудови паролів/хешування Argon2;
3. Протоколи захищеного зв’язку, це використання MQTT з TLS (версія TLS 1.2/1.3) та DTLS для UDP-орієнтованих сценаріїв.

Для реалізації тестів було використано мову програмування Python 3.13.7, середовище розробки Visual Studio Code (чудова альтернатива – Google Colab) а також бібліотеки, перелік основних наведений в таблиці 2.1.

Таблиця 2.1 – Використані бібліотеки

Категорія	Бібліотека	Опис
Шифрування / Криптографія	cryptography	для реалізації алгоритмів AES, RSA; генерації ключів; сертифікатів
	pyotp	для генерації одноразових кодів TOTP для автентифікації
	argon2-cffi	для хешування паролів
Комунікація	paho-mqtt	для клієнту MQTT, публікація/підписка
	dtls	для реалізації захищеної передачі поверх UDP – тобто це DTLS
Аналіз продуктивності	time	для вимірювання часу
	psutil	для моніторингу використання ресурсів машини
Обробка та візуалізація даних	pandas	для створення таблиць та обробки результатів
	matplotlib	для візуалізації даних

Усі емпіричні вимірювання, отримані в процесі тестування, зберігаються у форматі у CSV-файлах за власною уніфікованою схемою. Для кращого сприйняття дані візуалізовано у вигляді графіків чи гістограм у форматі PNG за допомогою бібліотеки matplotlib, які окремо будуть розглянуті окремо у відповідних підпунктах.

Сформований проект для дослідження містить модульну структуру. Кожен тест представлений як окремий програмний модуль (скрипт) із власними конфігураційними параметрами. Запуск модулів можливий як індивідуально, так і через сукупний скрипт, що забезпечує послідовне виконання всіх тестів. Кожен модуль підтримує параметризацію запуску, що дозволяє налаштовувати умови тестування.

Важливо зазначити що симуляція навантаження для тестів здійснюється на власній машині (персональний комп'ютер), в рамках встановлених меж навантаження (кількість ітерацій, об'єм пам'яті тощо). Всі параметри тестування задокументовано у вигляді коментарів до коду або в окремому файлі, який містить загальну довідкову інформацію та рекомендації щодо використання бенчмарків (тестів).

2.2. Емпіричне дослідження методів захисту в системах IoT

2.2.1. Тестування методів шифрування

Розуміння продуктивності криптографічних механізмів на пристроях з обмеженими ресурсами та у мережах з обмеженою пропускною здатністю є критичним в рамках системи розумного будинку. Тест aes_vs_rsa_benchmark.py порівнює два фундаментально різні підходи: симетричне шифрування AES та асиметричне RSA (зокрема у гібридному режимі: RSA шифрує симетричний ключ). Дослідження вимірює час виконання (це шифрування/дешифрування) та зміну RSS пам'яті (через бібліотеку psutil) на різних розмірах plaintext.

Ключова функція для вимірювання це measure_time_and_memory, яка стандартизовано вимірює час і зміну RSS процесу – див. лістинг коду 2.1.

Лістинг коду 2.1 – Функція вимірювання часу й пам'яті

```
def measure_time_and_memory(func, *args, **kwargs):
    gc.collect()
    mem_before = _process.memory_info().rss
    t_start = time.perf_counter()
    result = func(*args, **kwargs)
    t_end = time.perf_counter()
    gc.collect()
    mem_after = _process.memory_info().rss
    return result, (t_end - t_start), (mem_after - mem_before)
```

Пояснення термінів, що зустрічаються в коді: plaintext – це набір даних який буде зашифрований, ciphertext – зашифровані дані після відповідного процесу (незрозумілий набір символів).

Розглядаємо лістинг коду 2.1. Для вимірювання часу виконання криптографічних операцій використовується time.perf_counter(), яка забезпечує високоточний таймер, який особливо доречний при оцінці коротких затримок, де важлива точність до мікросекунд. Оцінка використання оперативної пам’яті здійснюється за допомогою методу memory_info().rss, що повертає обсяг пам’яті, який є фактично зарезервованої процесом у системі. Стосовно gc.collect(), цей рядок коду неодноразово буде зустрічатись у коді всіх тестів, необхідний він для очищення пам’яті щоб зменшити вплив на результати та мінімізувати похибку.

Наступний фрагмент коду містить реалізацію самого процесу шифрування та дешифрування – лістинг коду 2.2.

Лістинг коду 2.2 – Функції генерації ключа, шифрування та дешифрування

```
def encryption_aes_generate_key() -> bytes:
    return secrets.token_bytes(encryption_aes_key_size_bytes)


def encryption_aes_encrypt(encryption_aes_key: bytes, encryption_aes_plaintext: 
bytes):
    aesgcm = AESGCM(encryption_aes_key)
    encryption_aes_nonce = secrets.token_bytes(12)
    encryption_aes_ciphertext = aesgcm.encrypt(encryption_aes_nonce, 
encryption_aes_plaintext, associated_data=None)
    return encryption_aes_nonce, encryption_aes_ciphertext


def encryption_aes_decrypt(encryption_aes_key: bytes, encryption_aes_nonce: bytes, 
encryption_aes_ciphertext: bytes):
    aesgcm = AESGCM(encryption_aes_key)
    encryption_aes_plaintext = aesgcm.decrypt(encryption_aes_nonce, 
encryption_aes_ciphertext, associated_data=None)
    return encryption_aes_plaintext
```

Симетричне шифрування реалізоване за алгоритмом AES у режимі GCM (Galois/Counter Mode), який належить до класу AEAD (Authenticated Encryption with Associated Data). Суть використання цього режим в тому що він забезпечує

не лише конфіденційність даних, але й їхню цілісність, оскільки до зашифрованого повідомлення автоматично додається автентифікаційний тег.

Для кожної операції шифрування генерується унікальний послідовність (має назву nonce) з розміром 12 байт. Ця послідовність не має повторювався при використанні одного й того ж ключа, оскільки це може призвести до компрометації безпеки.

Алгоритм AES-GCM демонструє лінійну часову (тобто лінійне зростання O(n)) складність відносно розміру вхідного повідомлення. Його ефективність особливо висока на платформах з апаратною підтримкою AES-інструкцій (наприклад це AES-NI). У випадку IoT-пристроїв, продуктивність залежить від наявності відповідних апаратних ресурсів.

У межах тестування алгоритм застосовується до всього тексту plaintext з метою оцінки реальної вартості симетричного шифрування при обробці великих обсягів даних, що в свою чергу дозволяє отримати репрезентативні показники продуктивності, які будуть далі використані для порівняльного аналізу.

Асиметричне шифрування за алгоритмом RSA (лістинг 2.3) реалізовано з використанням OAEP (Optimal Asymmetric Encryption Padding). Такий підхід рекомендований у сучасних криптографічних системах для забезпечення семантичної захищеності, іншими словами це гарантії того що однакові повідомлення не шифруються в однаковий спосіб [13].

Важливо зазначити наступні два аспекти. Генерація RSA-ключів це значене навантаження на обчислювальні ресурси процесора а також через специфіку роботи генератора простих чисел, результати постійно б коливались що ускладнює процес дослідження, тому пара RSA-ключів генерується один раз. Інший аспект який варто враховувати це те що пряме RSA шифрування має обмеження в розмірі plaintext, це невеликі блоки, розмір яких розраховується за формулою 2.1.

$$\mathrm{max_plaintext} = \mathrm{key_bytes} - 2 \cdot \mathrm{hash_size} - 2 \qquad (2.1)$$

де key_bytes – довжина RSA-ключа у байтах;

hash_size – це довжина хешу, в байтах, для SHA-256 це 32 байти.

Якщо ж plaintext більший – застосовується гібридний сценарій, детальніше про нього в наступному фрагменті (лістинг коду 2.4).

Лістинг коду 2.3 – Функції асиметричного шифрування з OAEP

```
def encryption_hybrid_encrypt(encryption_rsa_public_key, encryption_aes_plaintext: bytes):
    encryption_hybrid_sym_key = encryption_aes_generate_key()
    encryption_hybrid_nonce, encryption_hybrid_ciphertext = encryption_aes_encrypt(encryption_hybrid_sym_key, encryption_aes_plaintext)
    encryption_hybrid_encrypted_key = encryption_rsa_encrypt_oaep(encryption_rsa_public_key, encryption_hybrid_sym_key)
    return {
        "encrypted_key": encryption_hybrid_encrypted_key,
        "nonce": encryption_hybrid_nonce,
        "ciphertext": encryption_hybrid_ciphertext,
    }
def encryption_hybrid_decrypt(encryption_rsa_private_key, encryption_hybrid_package: dict):
    encryption_hybrid_sym_key = encryption_rsa_decrypt_oaep(encryption_rsa_private_key, encryption_hybrid_package["encrypted_key"])
    plaintext = encryption_aes_decrypt(encryption_hybrid_sym_key, encryption_hybrid_package["nonce"], encryption_hybrid_package["ciphertext"])
    return plaintext
```

Гібридний підхід (лістинг коду 2.4) поєднує дві сильні сторони: симетричне шифрування яке ефективне для великих даних, та асиметричне – зручне для доставки ключа між сторонами. Спочатку генерується випадковий AES-ключ для шифрування повідомлення в режимі AES-GCM з унікальним nonce. Потім цей ключ зашифровується RSA-OAEP для безпечної передачі. Результат – пакет із зашифрованим ключем, nonce та ciphertext.

Ключова метрика тут це сумарна додаткова витрата ресурсів, яку створює операція RSA для шифрування симетричного ключа, разом із вартістю симетричного шифрування всього обсягу даних за допомогою AES.

Лістинг коду 2.4 – Гібридний режим (RSA для ключа, AES-GCM для даних)

```
def encryption_hybrid_encrypt(encryption_rsa_public_key, encryption_aes_plaintext:
bytes):
    encryption_hybrid_sym_key = encryption_aes_generate_key()
    encryption_hybrid_nonce, encryption_hybrid_ciphertext =
encryption_aes_encrypt(encryption_hybrid_sym_key, encryption_aes_plaintext)
    encryption_hybrid_encrypted_key =
encryption_rsa_encrypt_oaep(encryption_rsa_public_key, encryption_hybrid_sym_key)
    return {
        "encrypted_key": encryption_hybrid_encrypted_key,
        "nonce": encryption_hybrid_nonce,
        "ciphertext": encryption_hybrid_ciphertext,
    }

def encryption_hybrid_decrypt(encryption_rsa_private_key,
encryption_hybrid_package: dict):
    encryption_hybrid_sym_key =
encryption_rsa_decrypt_oaep(encryption_rsa_private_key,
encryption_hybrid_package["encrypted_key"])
    plaintext = encryption_aes_decrypt(encryption_hybrid_sym_key,
encryption_hybrid_package["nonce"], encryption_hybrid_package["ciphertext"])
    return plaintext
```

Остання частина коду цього тесту (лістинг коду 2.5) яку буде розібрано зосереджує увагу на візуалізації отриманих результатів. Функція summarize_and_plot виконує одразу кілька важливих завдань: вона агрегує зібрані під час експерименту дані, обчислює середні значення часу виконання та використання пам'яті, а також формує узагальнену таблицю результатів.

Лістинг коду 2.5 – Агрегація даних, виконання вимірювань (частина коду)

```
for encryption_size in encryption_test_sizes:
    n_iter = iterations_for_size(encryption_size)
    for i in range(n_iter):
        encryption_aes_plaintext = secrets.token_bytes(encryption_size)
        encryption_aes_key = encryption_aes_generate_key()
        # ... measure AES encrypt/decrypt ...
        if encryption_size <= encryption_rsa_max_plaintext:
            # ... measure RSA direct encrypt/decrypt ...
        else:
            # ... measure RSA direct encrypt/decrypt ...
        results.append({...})
```

Така таблиця необхідна для оцінки продуктивність кожного з методів і кожної операції. Всі дані зберігаються у вигляді *.csv файлу, та на основі них будуються графіки для кращого сприйняття.

Слідом за сформованими графіками код переходить до підсумкової інтерпретації отриманих результатів. Завершальний блок коду реалізує CLI-інтерфейс (консоль) за допомогою бібліотеки argparse, за допомогою якого можна задати кількість ітерацій, вказати шлях до файлу для збереження результатів або вимкнути генерацію графіків, якщо візуалізація не потрібна.

2.2.2. Тестування методів автентифікації

Наступний тест totp_vs_argon2_test.py для порівняльного аналізу методів автентифікації, доречних для IoT-екосистем. Метою тесту є кількісна оцінка двох класичних підходів: хешування паролів за допомогою Argon2 та генерація і верифікація одноразових кодів TOTP. Задача полягає в визначенні витрат часу та пам’яті кожної операції, оцінка надійності (це успішність операцій) та в завершення подібно до попередньої структури коду тесту – підготовці агрегованих даних, їх візуалізація. Далі розглянуті послідовно ключові фрагменти коду. Функція вимірювання часу та пам’яті – лістинг коду 2.6.

Лістинг коду 2.6 – Функція вимірювання часу та пам’яті

```
def measure_time_and_memory(func: Any, *args, **kwargs) -> tuple[Any, float, int]:
    gc.collect()
    mem_before: int = _process.memory_info().rss
    t_start: float = time.perf_counter()
    result = func(*args, **kwargs)
    t_end: float = time.perf_counter()
    gc.collect()
    mem_after: int = _process.memory_info().rss
    return result, (t_end - t_start), (mem_after - mem_before)
```

Було застосовано такий же уніфікований підхід до метричних вимірювань: замір RSS-пам’яті процесу memory_info() та фіксація часу через time.perf_counter(). Параметри elapsed_seconds і mem_delta_bytes є основними індикаторами продуктивності. Такий підхід мінімізує вплив непередбачуваної затримки (gc.collect()) та дозволяє отримати порівнювані між собою показники для різних операцій. Застосування RSS як метрики пам’яті відображає системний слід процесу, що важливо для пристроїв з обмеженими ресурсами.

Наступний фрагмент присвячений підготовці перед самим тестом та функції Argon2 – див. лістинг коду 2.7.

Лістинг коду 2.7 – Підготовка перед тестом, функції для Argon2

```
def auth_argon2_setup() -> PasswordHasher:
    ph = PasswordHasher(time_cost=auth_argon2_time_cost,
memory_cost=auth_argon2_memory_cost, parallelism=auth_argon2_parallelism)
    return ph


def auth_argon2_hash(auth_argon2_hasher: PasswordHasher, auth_argon2_password: str)
-> str:
    auth_argon2_hash = auth_argon2_hasher.hash(auth_argon2_password)
    return auth_argon2_hash


def auth_argon2_verify(auth_argon2_hasher: PasswordHasher, auth_argon2_hash_str:
str, auth_argon2_password: str) -> bool:
    try:
        return auth_argon2_hasher.verify(auth_argon2_hash_str,
auth_argon2_password)
    except Exception:
        return False
```

В даній реалізації використовується argon2-cffi через клас PasswordHasher, параметризований наступними трьома ключовими аргументами: time_cost (кількість ітерацій), memory_cost (споживання пам’яті в KiB, це кібібайт = 1024 байт) і parallelism (відповідно паралелізм). Аргумент auth_argon2_memory_cost у скрипті визначений як 64 MB, стандартна величина. Операція hash повертає рядок хешу, який зберігає параметри (це: salt, time/memory/parallelism) в одному рядку, verify перевіряє пароль за хешем та повертає булевий результат. В рамках

цього дослідження, тесту, важливо розділяти час хешування та час перевірки, оскільки під час експлуатації частота створення хешів (під час реєстрації/зміни пароля) значно нижча за кількість верифікацій.

Наступний фрагмент (лістинг коду 2.8) для TOTP. Застосовується бібліотека pyotp, яка реалізує алгоритм RFC 6238. Генерується секрет, в даному контексті секрет це набір байтів який утворює код відомий лише системі та користувачеві, детермінованим чином (Base32), після чого створюється сам об’єкт TOTP. Функції now() та verify() відповідають за оперативну генерацію коду і його валідацію відповідно. Оскільки TOTP – це легка криптографічна операція (HMAC та кілька арифметичних дій), її латентність зазвичай вимірюється в сотих мілісекундах, що значно менше за витрати Argon2. Для IoT-проєктів це має практичне значення – TOTP підходить як другий фактор через низьку обчислювальну вартість.

Лістинг коду 2.8 – Підготовка перед тестом, функції для TOTP

```
def auth_totp_setup() -> tuple[str, pyotp.TOTP]:
    auth_totp_secret: str = pyotp.random_base32()
    auth_totp_obj = pyotp.TOTP(auth_totp_secret)
    return auth_totp_secret, auth_totp_obj


def auth_totp_generate_code(auth_totp_obj: pyotp.TOTP) -> str:
    auth_totp_code: str = auth_totp_obj.now()
    return auth_totp_code


def auth_totp_verify_code(auth_totp_obj: pyotp.TOTP, auth_totp_code: str) -> bool:
    try:
        return auth_totp_obj.verify(auth_totp_code)
    except Exception:
        return False
```

Далі йде великий блок коду що відповідає за основну логіку тесту – частина наведена в лістингу коду 2.9. Узагальнююче пояснення далі. В ньому йде імітування незалежних автентифікаційних спроб, кількість визначається параметром iterations (ітерації). Для кожної ітерації генерується випадковий

пароль (функція gen_random_password()), виконується хешування Argon2 та його верифікація, а також генерація та перевірка TOTP-коду. Використано tqdm для індикатора прогресу під час тривалого процесу тестування. Усі результати (час, зміни пам'яті, успішність) накопичуються у списку словників та пізніше перетворюються у pandas.DataFrame для подальшого аналізу.

Лістинг коду 2.9 – Основний цикл тестування (частина коду)

```
for i in tqdm(range(iterations), desc="Auth attempts"):
    auth_password = gen_random_password()
    auth_hash_result, auth_argon2_time, auth_argon2_mem =
measure_time_and_memory(auth_argon2_hash, auth_argon2_hasher, auth_password)
    # ... verify ...
    auth_totp_code_result, auth_totp_gen_time, auth_totp_gen_mem =
measure_time_and_memory(auth_totp_generate_code, auth_totp_obj)
    auth_totp_verify_result, auth_totp_verify_time, auth_totp_verify_mem =
measure_time_and_memory(auth_totp_verify_code, auth_totp_obj, auth_totp_code)
    # ... results ...
```

Цей тест також має агрегацію даних, їх візуалізацію та обчислення метрик які реалізовані подібним чином до попереднього тесту. Агрегація для отримання зведених метрик здійснюється за формулою 2.2, а частка успішних операцій розраховується за формулою 2.3.

$$\bar{t} = \frac{1}{N} \sum_{i=1}^{N} t_i \tag{2.2}$$

де t_i час i-тої операції;

N число валідних спостережень.

$$success_rate = \frac{\sum_{i=1}^{N} 1(success_i)}{N} \tag{2.3}$$

де 1 – індикатор успіху.

Для наочності результати відображаються у вигляді стовпчикових графіків, де відображають середній час (в мс) і зміну використання пам’яті (в байтах). Всі необроблені («сирі») й агреговані дані зберігаються у CSV-файлах.

2.2.3. Тестування протоколів зв’язку

Останній тест реалізовано для порівняння захищених протоколів зв’язку. Мета тесту – виміряти латентність (RTT), надійність доставки та зміну системного використання пам’яті для двох підходів, це: MQTT поверх TLS (імітація через TLS TCP echo) та DTLS (UDP-імітація). Визначення термінів: RTT (Round-Trip Time) – час повного циклу, за який запит доходить до сервера і повертається назад; RSS (Resident set Size) – це об’єм пам’яті який процес реально використовує в даний момент. Далі будуть розглянуті послідовно ключові фрагментів коду з поясненням їхньої ролі в методиці дослідження та інтерпретації результатів.

Як і в попередніх двох тестах перший ключовий фрагмент це функція виміру часу та пам’яті – лістинг коду 2.10.

Лістинг коду 2.10 – Функція виміру часу та пам’яті

```
def measure_time_and_memory(func: Any, *args, **kwargs) -> tuple[Any, float, int]:
    gc.collect()
    mem_before = _process.memory_info().rss
    t_start = time.perf_counter()
    result = func(*args, **kwargs)
    t_end = time.perf_counter()
    gc.collect()
    mem_after = _process.memory_info().rss
    return result, (t_end - t_start), (mem_after - mem_before)
```

Подібно до попередніх відповідних функцій – ця також уніфіковано збирає три основні показники: результат виклику, час виконання (в секундах) і зміна RSS у байтах. Використання time.perf_counter() забезпечує високу роздільну здатність для коротких інтервалів, що важливо в рамках вимірювання RTT та

швидких криптографічних операцій. Значення ΔRSS слугує наближенням до системного сліду операції.

Необхідність генерації самопідписаних сертифікатів обумовлена можливістю розгорнути локальний TLS-сервер без зовнішніх залежностей (лістинг коду 2.11). Це важливо для контрольованих локальних експериментів, де необхідно уникнути мережевих факторів які очевидно що ускладнюють самовідтворення. Сертифікати підписано за допомогою SHA-256 в термін дії рік.

Лістинг 2.11 – Функція генерації самопідписаних сертифікатів (частина коду)

```
def generate_self_signed_cert(cert_file: Path, key_file: Path) -> None:
    key = rsa.generate_private_key(public_exponent=65537, key_size=2048)
    subject = issuer = x509.Name([
        #... The certificate is self-signed using a generated RSA key ...
    ])
    cert = (
        #... Both subject and issuer are set to a predefined identity containing
standard certificate fields ...
    )
    with open(key_file, "wb") as f:
        f.write(key.private_bytes(
            encoding=serialization.Encoding.PEM,
            format=serialization.PrivateFormat.TraditionalOpenSSL,
            encryption_algorithm=serialization.NoEncryption(),
        ))
    with open(cert_file, "wb") as f:
        f.write(cert.public_bytes(serialization.Encoding.PEM))
```

Цей сервер створює TLS-з’єднання поверх TCP і поводиться як echo-брокер, тобто: приймає повідомлення та відразу відправляє їх назад. Ця емуляція дозволяє виміряти RTT для зашифрованого з’єднання без реалізації повного MQTT-стеку. Запуск у фоновому потоці та обробка клієнтів у окремих потоках забезпечують паралелізм і робоче навантаження, яке наближене до реального (лістинг коду 2.12).

Лістинг коду 2.12 – TLS echo-сервер, що імітує MQTT-брокер поверх TLS з багатопотоковою обробкою клієнтів

```
class ProtocolMQTTTLS_EchoServer(threading.Thread):
    def __init__(self, host, port, certfile, keyfile):
        #...

    def run(self) -> None:
        ctx = ssl.create_default_context(ssl.Purpose.CLIENT_AUTH)
        ctx.load_cert_chain(certfile=self.certfile, keyfile=self.keyfile)
        bindsock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        bindsock.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
        bindsock.bind((self.host, self.port))
        bindsock.listen(5)
        self._sock = bindsock
        self._running.set()
        while self._running.is_set():
            newsock, addr = bindsock.accept()
            ssock = ctx.wrap_socket(newsock, server_side=True)
            newsock.close()
            threading.Thread(target=self._handle_client, args=(ssock, addr),
daemon=True).start()

    def _handle_client(self, ssock: ssl.SSLSocket, addr) -> None:
         data = ssock.recv(4096)
         data = ssock.recv(4096)
         if not data: break
```

Розгляд наступного фрагменту – див. лістинг коду 2.13. Клієнт встановлює захищене з’єднання, створюючи SSL-обгортку (TLS wrapper) навколо звичайного сокету. Обгортка – це спеціальний шар, який додає шифрування і безпеку до існуючого з’єднання без зміни основної логіки роботи сокета. В цьому випадку вона забезпечує захист даних під час передачі.

Для локального тестування перевірка сертифікатів вимкнена (verify_mode = CERT_NONE), що дозволяє працювати з самопідписаними сертифікатами без помилок.

Функція виконує відправку повідомлення і прийом відповіді, а потім повертає час затримки (латентність), зміну використання пам’яті або індикатор невдачі, якщо наприклад сталася помилка.

Лістинг коду 2.13 – MQTT/TLS клієнтський цикл з вимірюванням часу та пам’яті (один RTT)

```
def protocol_mqtt_tls_client_round(host: str, port: int, certfile: Path, message: bytes, timeout: float = 5.0) -> tuple[Optional[float], Optional[int], bool]:
    def _do_round() -> bytes:
        sock = socket.create_connection((host, port), timeout=timeout)
        ctx = ssl.create_default_context(ssl.Purpose.SERVER_AUTH)
        ctx.check_hostname = False
        ctx.verify_mode = ssl.CERT_NONE
        ssock = ctx.wrap_socket(sock, server_hostname='localhost')
        ssock.settimeout(timeout)
        ssock.sendall(message)
        data = ssock.recv(65536)
        ssock.close()
        return data

    try:
        _, latency_sec, mem_delta = measure_time_and_memory(_do_round)
        success = True
    except Exception:
        latency_sec = None
        mem_delta = None
        success = False
    return latency_sec, mem_delta, success
```

Наступний фрагмент коду – лістинг коду 2.14. Для DTLS використовується бібліотека dtls, якщо ж її немає, застосовується UDP echo як симуляція поведінки DTLS, але без шифрування, бо сокет не реалізовує захищений канал самостійно. Тобто, відсутність бібліотеки призводить до передачі даних без шифрування, що вкотре підкреслює критичну залежність безпеки систем таких як розумний будинок від правильного налаштування криптографічних компонентів.

Основна логіка тесту наведена в лістингу коду 2.15. В ній застосовано ітераційний підхід: у кожному етапі формується тестове повідомлення (payload) фіксованого розміру, після чого виконуються два вимірювання – одне для MQTT/TLS інше для DTLS/UDP. Отримані «сирі» дані зберігаються у табличному вигляді, для зручності подальшої агрегації яка може включати обчислення середнього значення, медіани чи перцентилів (тобто аналіз розподілу значення в наборі даних); та візуалізацію результатів. Відповідно

даний тест також має функції агрегації даних та їх візуалізації з логікою подібною до попередніх тестів.

Лістинг коду 2.14 – UDP echo-сервер для імітації DTLS-брокера та клієнтський fallback-раунд без шифрування

```
def protocol_dtls_udp_echo_server(host, port, certfile, keyfile, stop_event):
    sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
    sock.bind((host, port))
    while not stop_event.is_set():
        try:
            data, addr = sock.recvfrom(65536)
        except socket.timeout:
            continue
        sock.sendto(data, addr)

def protocol_dtls_client_round(host, port, message, timeout=5.0):
    if _dtls_available:
        try dtls variant ...
    try:
        sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
        sock.settimeout(timeout)
        def _do_udp_round():
            sock.sendto(message, (host, port))
            data, _ = sock.recvfrom(65536)
            return data
        _, latency_sec, mem_delta = measure_time_and_memory(_do_udp_round)
        sock.close()
        return latency_sec, mem_delta, True
    except Exception:
        return None, None, False
```

Лістинг коду 2.15 – Основний цикл тесту та збереження результатів

```
for i in tqdm(range(rounds), desc="Protocol rounds"):
    protocol_payload = secrets.token_bytes(protocol_message_payload_size)
    mqtt_latency, mqtt_mem, mqtt_success = protocol_mqtt_tls_client_round(...)
    results.append({'method': 'MQTT-over-TLS', 'latency_sec': mqtt_latency,
'mem_delta_bytes': mqtt_mem, 'success': mqtt_success})
    dtls_latency, dtls_mem, dtls_success = protocol_dtls_client_round(...)
    results.append({'method': 'DTLS (or UDP-fallback)', 'latency_sec':
dtls_latency, 'mem_delta_bytes': dtls_mem, 'success': dtls_success})
df = pd.DataFrame(results)
df.to_csv(out_csv, index=False)
```

2.3. Аналіз результатів тестування

У попередніх підпунктах було детально розглянуто програмну реалізацію тестів, їх логіку функціонування та ключові функції. Цей підрозділ присвячений безпосередньому аналізу отриманих експериментальних даних. Як вже було згадано раніше результати зберігаються як *.csv файли (рис. 2.1) на основі яких будуються графіки. Далі подано послідовний, структурований опис який включає: ключову інформацію, виявлені закономірності а також практичні висновки в межах окремого тесту.

Рисунок 2.1 – Приклад збереження даних, наповнення файлу *.csv

Розглядаємо отримані дані з проведеного тесту методів шифрування AES-GCM та RSA-OAEP. Загальну картину найзручніше сприймати за допомогою графіків, зокрема для цього тесту було побудовані наступні: середній час шифрування (див. рис. 2.2), середній час дешифрування (див. рис. 2.3) та середні зміни RSS-пам’яті (див. рис. 2.4).

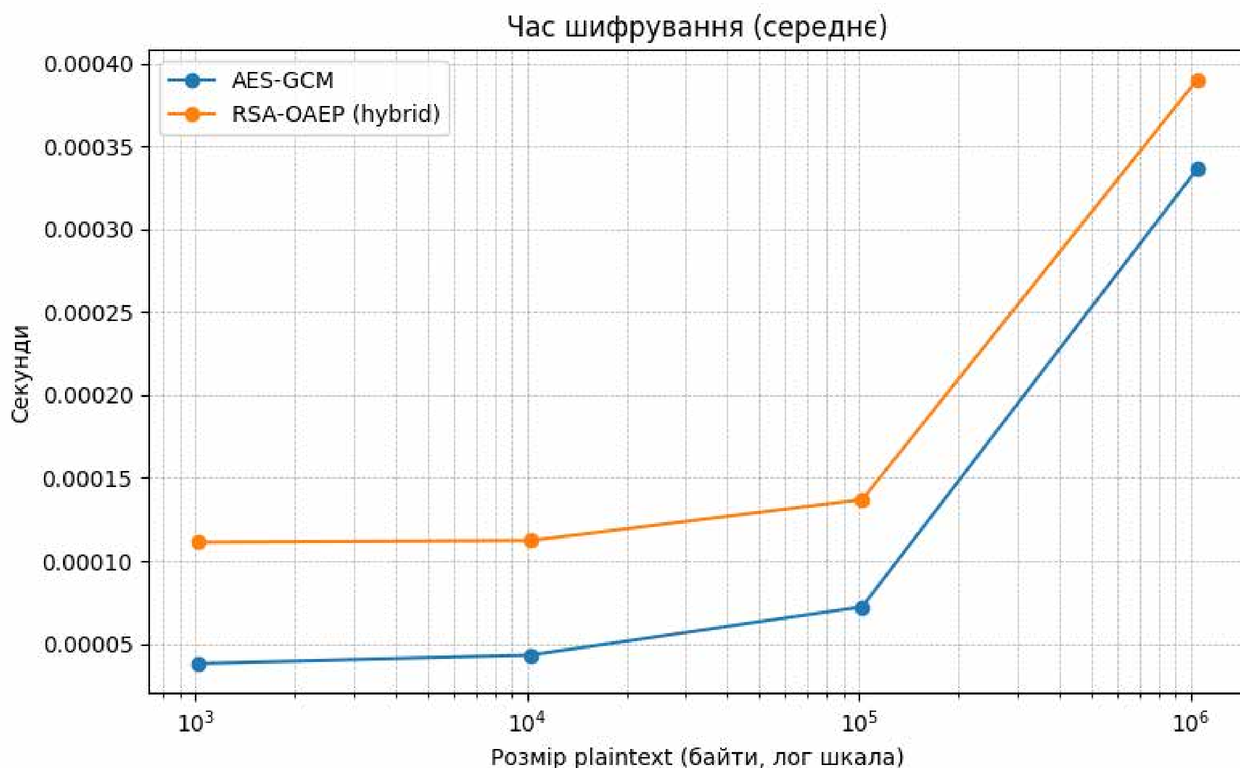

Рисунок 2.2 – Графік середнього часу шифрування

Графік середнього значення часу шифрування (рис. 2.2) наглядно демонструє, що AES-GCM забезпечує найнижчу латентність для всіх розмірів повідомлень. При малих блоках (в рамках 1 КБ) середній час шифрування AES знаходиться в діапазоні десятих тисячних мілісекунд, а при зростанні блока до 1 МБ підвищується до приблизно 0.33 – 0.34 мс. Гібридна схема має вищий початковий наклад, зокрема це через RSA-операцію для шифрування симетричного ключа, тому сумарний час гібридної операції знаходиться помітно вище за AES для всіх розмірів блоків і становить близько 0.11 мс для 1 КБ і ≈0.39 мс для 1 МБ. Така поведінка є очікуваною, оскільки симетричні алгоритми оптимізовані для потокового або блочного шифрування великих обсягів даних, тоді як асиметричні операції несуть фіксований обчислювальний наклад. Ця тенденція в отриманих даних цілком узгоджуються з іншими опублікованими порівняльними оцінками ефективності симетричних та асиметричних алгоритмів для обмежених платформ [14, с. 9-13].

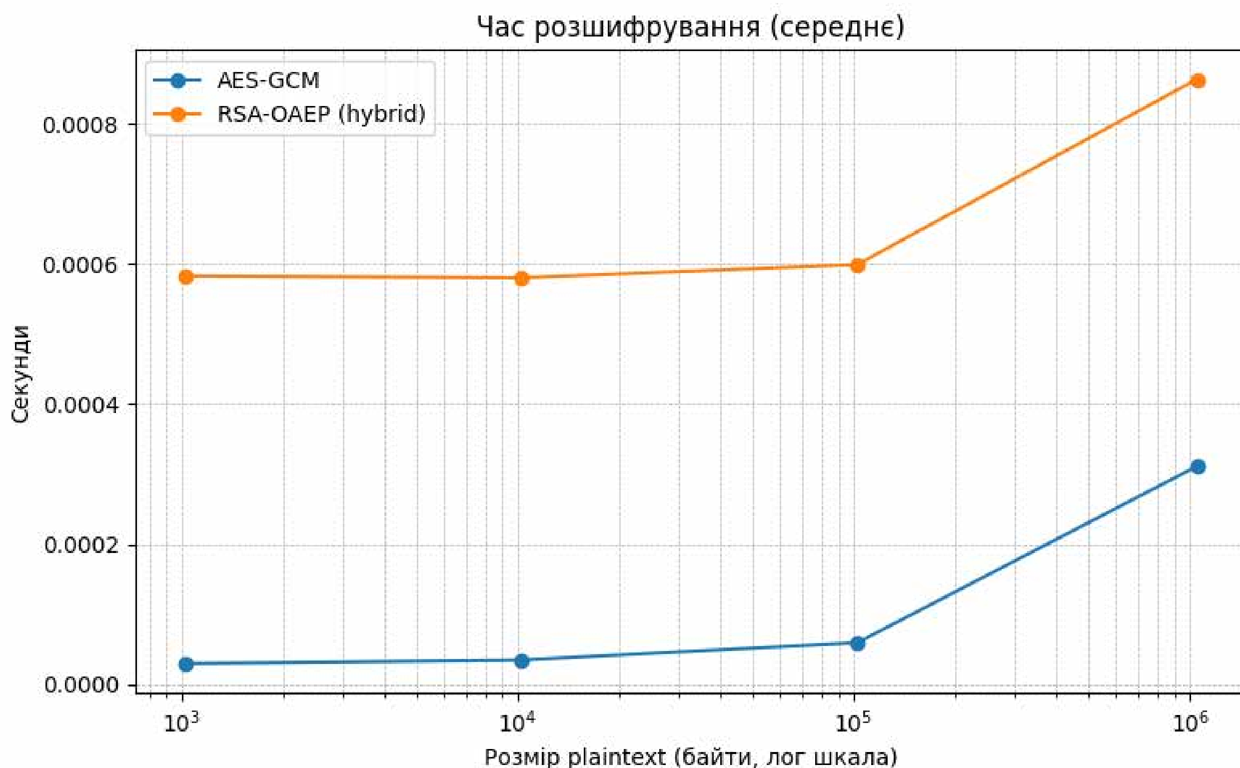

Рисунок 2.3 – Графік середнього часу дешифрування

В свою чергу графік середнього часу дешифрування (рис. 2.3) виявив ще більш помітні відмінності: AES-GCM досі зберігає низьку латентність і лінійну залежність від розміру даних (від десятків мікросекунд для малих блоків до ≈0.31 мс для 1 МБ), тоді як для гібридної схеми дешифрування приватним RSA-ключем формується значна стала витрати часу, яка домінує над зростанням обчислень з збільшенням розміру блоків. З того ж графіку видно, вітка графіку RSA-дешифрування займає кілька десятих мілісекунди навіть для невеликих блоків (≈0.58 мс для 1 КБ) і вже збільшується до ≈0.86 – 0.90 мс для 1 МБ. Така асиметрія пояснюється, алгоритмічною, складністю операцій приватного ключа (це великі множення над багатовимірними числами) і властивістю реалізацій самих криптографічних бібліотек, у яких операції приватного дешифрування є витратнішими щодо обчислень, ніж операції публічного шифрування. Тобто, у сценаріях, де дешифрування виконується часто на ресурсо-обмежених вузлах, це фактично означає що така гібридна схема може створювати проблемні місця в продуктивності.

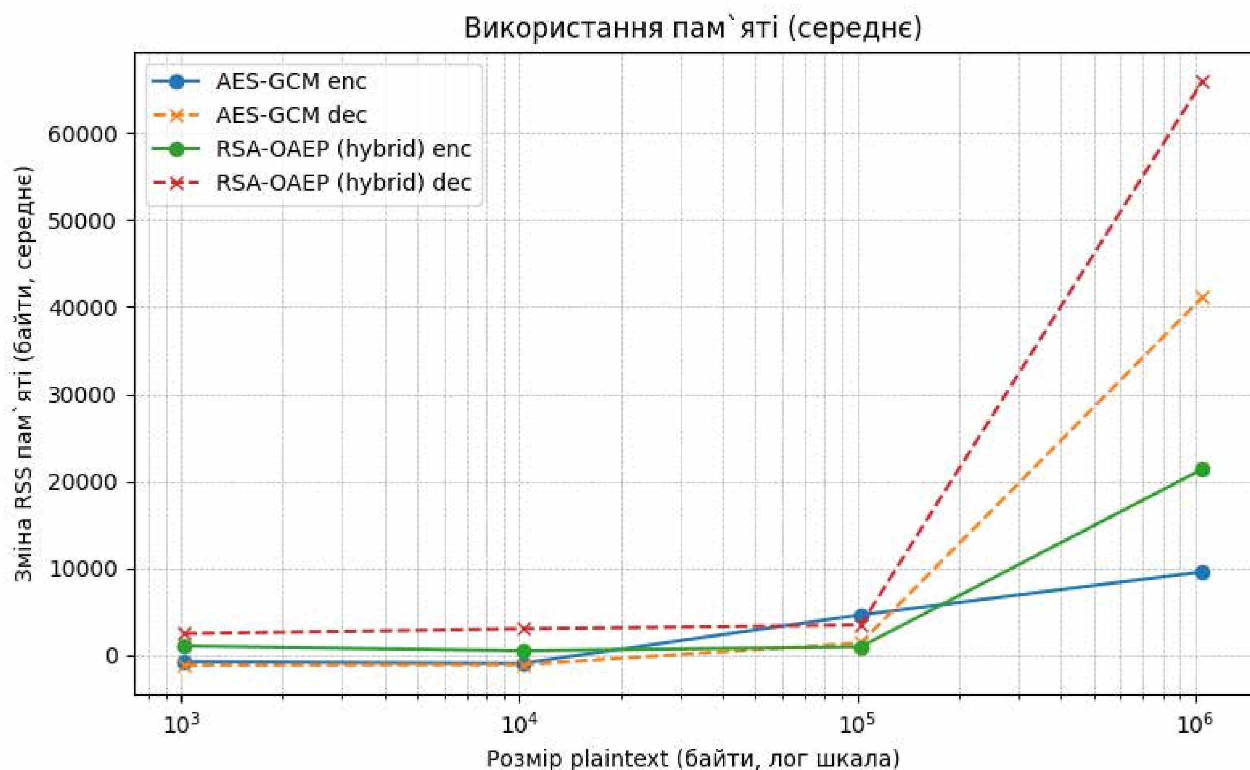

Рисунок 2.4 – Графік середнього використання пам’яті

Графік зміни використання пам’яті (рис. 2.4) відносно розміру даних доповнює данні спостереження: при малих і середніх розмірах даних ΔRSS для обох підходів спостерігається відносно невеликим, тоді як вже при розміру даних (plaintext) в 1 МБ помітна суттєва різниця – середні значення ΔRSS для AES-GCM зростають до кількох десятків кілобайт, а для RSA-OAEP у гібриді пікові значення при дешифруванні перевищують 60 кБ (див. рис. 2.4). Можна припустити що підвищене споживання пам’яті під час RSA-дешифрування ймовірно пов’язане зі створенням проміжних структур для обчислень над великими цілими числами та з роботою бібліотек реалізації приватних операцій. Однак варто наголосити на важливому моменті – ΔRSS піддається зовнішньому шуму (вплив ОС, менеджера пам’яті, збирача сміття «gc.collect») хоч і була спроба мінімізації його впливу, але попри це явна тенденція до більшого системного сліду RSA-дешифрування залишається очевидною в наведених даних.

Далі наведено аналіз отриманих результатів порівняння методів автентифікації Argon2 та TOTP. Метою тесту було кількісно оцінити часові витрати і зміну системної пам’яті для операцій хешування та перевірки (Argon2), а також для операцій генерації та верифікації (TOTP).

Оцінка часу виконання (див. рис. 2.5) показує суттєву різницю між Argon2 та TOTP: генерація й верифікація TOTP відбуваються в мілісекундних або навіть в коротших інтервалах, що є набагато швидше за опрацьовані параметри Argon2, де хешування займає більше часу через особливості роботи самого алгоритму.

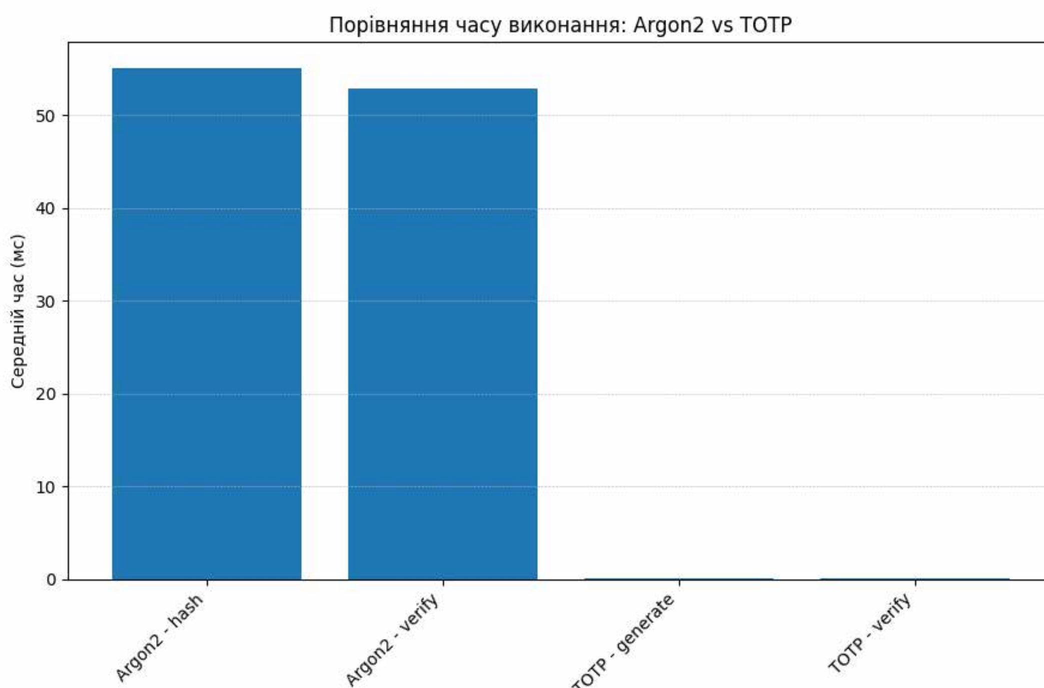

Рисунок 2.5 – Метрика часу виконання Argon2 та TOTP

На графіку часових метрик видно, що середній час виконання операцій Argon2 у конфігурації з помірними параметрами (time_cost, memory_cost) значно перевищує час TOTP – що саме і підтверджує базове припущення: Argon2 призначений як захист від швидких атак шляхом свідомого збільшення ресурсів, необхідних для перевірки та пошуку пароля. Для практичного розуміння варто врахувати, що TOTP реалізує HMAC-обчислення та просту арифметичну обробку часу, а отже він працює надзвичайно швидко – у межах мікро або

мілісекунд (що власне можна побачити на рис. 2.5), натомість Argon2 навмисно навантажує пам'ять і процесор, щоб ускладнити використання апаратних засобів спрямованих для пришвидшеного зламу. Алгоритму Argon2 притаманна така властивість як «memory-hard» функція [15, с. 2-4].

Розподіл операцій хешування та перевірки потребує уточнення: хешування створює стійкий рядок, що містить параметри та salt (це випадкові дані, що додаються до пароля перед хешуванням з метою уникнення повторів), тоді як перевірка виконує повторне відтворення обчислення для порівняння. У більшості реалізацій обидві операції мають схожу асимптотичну складність, однак оптимізації в коді або відмінності в налаштуваннях можуть спричинити помітні відхилення в часі виконання.

Графік зміни RSS-пам'яті демонструє, що операції Argon2 мають найбільший системний слід у порівнянні з TOTP, однак можна помітити що значення ΔRSS іноді мають від'ємний знак що заслуговує окремого пояснення (див. рис. 2.6).

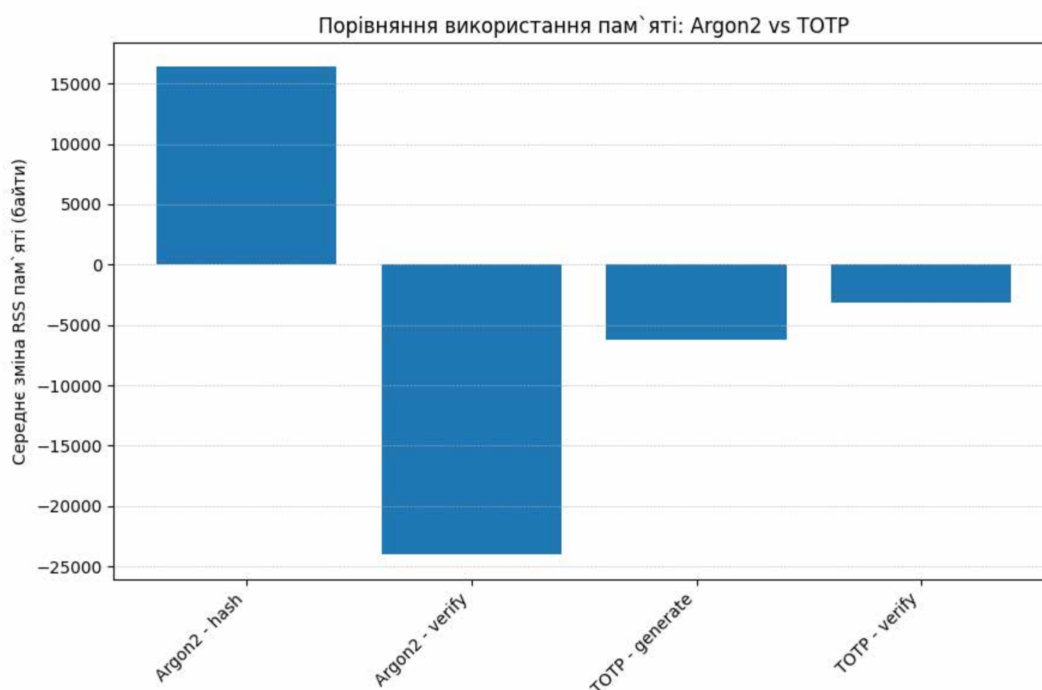

Рисунок 2.6 – Метрика використання пам'яті, Argon2 та TOTP

Такі результати (від'ємні показники) є цілком «природніми» в контексті обраного способу тестування і відображають реальну поведінку середовища виконання що вказує на звільнення пам'яті або артефакти поведінки менеджера пам'яті між вимірами тощо.

На прикладі середніх величин видно наступне: під час фази хешування Argon2 спостерігається помітне середнє додаткове навантаження (порядку десятків кілобайт), тоді як під час фази перевірки може фіксуватись негативна або зворотна зміна RSS, обумовленість якої вже було висвітлена, - наслідок викликів збирача сміття (gc.collect), кешування або повторного використання буферів у середовищі виконання. TOTP, навпаки, має невеликі абсолютні ΔRSS (у кілобайтах або менше) і демонструє меншу варіативність. Отримані ΔRSS слід сприймати з обережністю бо вони піддаються впливу фонового навантаження ОС, перерозподілів алокацій і звільнень, тощо. Тому якщо необхідно отримати більш точні показники до пам'яті цей напрямок може стати доповнення для даної магістерської роботи – в рамках цього тесту, потрібно доповнити RSS-аналіз іншими інструментами, як heap-профілюванням або зовнішнім системним моніторингом. Проте надані дані вже наочно показали, що Argon2 створює значно більший піковий слід у порівнянні з TOTP.

Останній тест був присвячений дослідженню продуктивності двох підходів до захищеної передачі повідомлень які використовуються у контексті IoT: це MQTT поверх TLS та DTLS (з механізмом повернення до UDP), його мета була кількісно оцінити латентності (час затримки), витрати на обробку відповідей і системного сліду по оперативній пам'яті. Далі наведені інтерпретації спостережень на основі отриманих даних, кількісні співвідношення та відповідно їх аналіз. З синтезованих даних, подібно до попередніх тестів, також були побудовані графікі для кращого сприйняття інформації, перший який буде розглянуто – графік латентності (див. рис. 2.7).

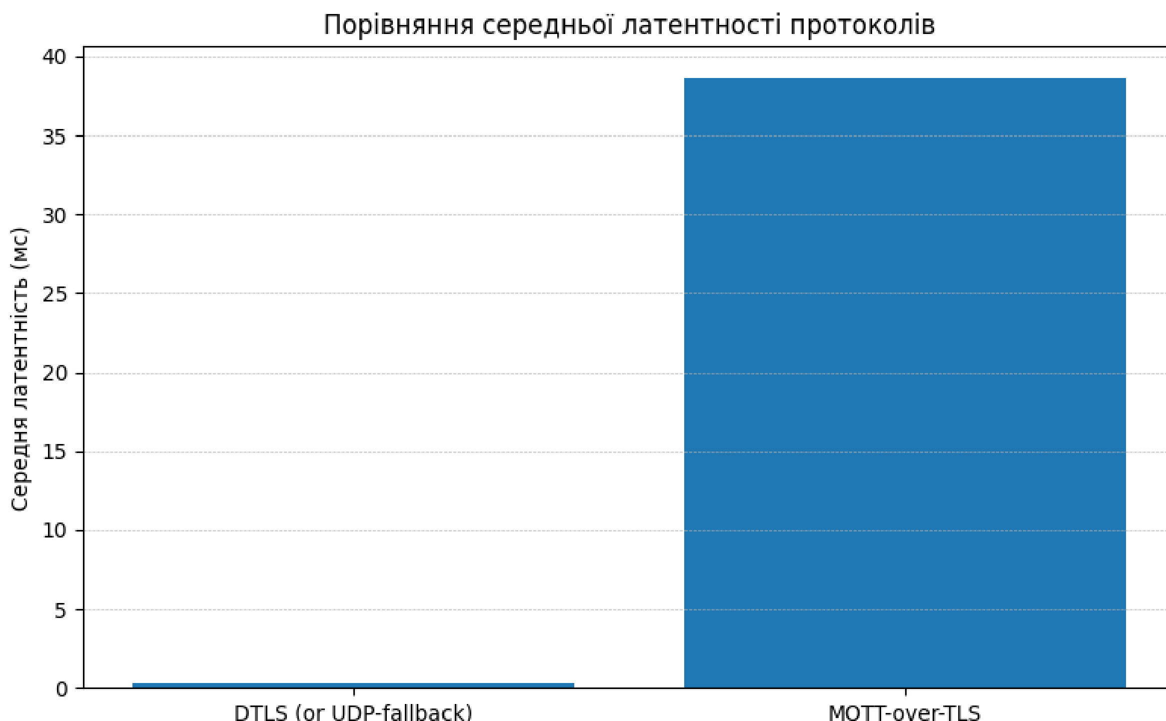

Рисунок 2.7 – Графік середньої латентності протоколів

Середній час затримки, одного раунду, передачі демонструє наглядну різницю між двома режимами: у DTLS/UDP-fallback вона становить приблизно 0,33 мс, тоді як у MQTT-over-TLS – близько 38,7 мс (див. рис. 2.7). Така сильна різниця має кілька технічних пояснень. По-перше, MQTT традиційно працює поверх TCP, а при використані на додачу TLS накладається криптографічний шар, що вимагає додаткових обчислень для шифрування і дешифрування, ведення сесій і управління буферами. По-друге, у TCP-орієнтованому сценарії присутні механізми підтверджень (ACK) і відновлення втрат, це додає затримки при кожному циклі запит – відповідь. В сумі ці фактори і дають суттєвий фіксований наклад, який виявляється особливо помітним для коротких повідомлень і частих циклів обміну. Щодо, DTLS який реалізований поверх UDP, у режимі fallback, який позбавлений TCP-верхнього шару й відповідних механізмів, саме тому латентність залишається низькою. Тому, в режимі fallback не відтворюється повністю захищена DTLS-функціональність, що слід враховувати.

Відмінності в поведінці під час обробки відповідей (умовно названі як «дешифрування» у контексті RTT-вимірів) посилюють попередні спостереження: у TLS-сценарії клієнт додатково навантажений перевіркою криптографічного з’єднання і, залежно від конфігурації cipher-suite та перевірок сертифікатів, ці операції можуть бути ресурсозатратним. Отримані дані показують, що для MQTT-over-TLS час обробки відповіді значно більший у порівнянні з DTLS/UDP (приблизно в два порядки величини у середньому для коротких передач). Виходячи з цього формується відповідно наступна думка – що у сценаріях із великою частотою обміну, або мікрофреймами, вибір на користь TLS може призвести до помітного зниження пропускної здатності та збільшення відгуку системи. З іншого ж боку DTLS забезпечує захищене шифрування поверх UDP і тим самим поєднує малу затримку із криптографічним захистом, при умові повноцінної реалізації. Відсутність повноцінної DTLS-підтримки (тобто це у випадку UDP-fallback) захист відсутній що призводить до компрометації вимог до конфіденційності та цілісності.

Аналіз системного сліду по пам’яті (графік – див. рис. 2.8). Відразу видно негативне значення, природу якого потрібно окреслити. Подібно до попереднього тесту, воно не свідчить про «негативне споживання» як таке, а просто вказує на те, що після виконання низки операцій процес повернув частину виділеної ОС пам’яті або відбулося звільнення кешів чи буферів, також це може бути наслідком поведінки менеджера пам’яті інтерпретатора або бібліотеки сокетів. Ще важливо наголосити на тому що, знову ж таки подібно до попереднього тесту, RSS вимірює resident set size у межах процесу й підлягає впливу зовнішніх факторів (фонові процеси, ОС, GC у Python), але абсолютна різниця між режимами вказує на те, що TLS-стек значно більше споживає пам’ять у порівнянні з UDP-реалізаціями. Якщо мислити з інженерної точки зору, це означає, що у вузлах із обмеженим об’ємом ОЗП застосування MQTT-over-TLS може вимагати додаткових ресурсів або застосування оптимізацій (наприклад використання TLS-PSK, спрощених cipher-suite або делегування TLS-обробки на шлюз).

Рисунок 2.8 – Графік використання пам'яті протоколів

Варто наголосити також на наступних трьох особливостях. По-перше, багато залежить від того, чи TLS-з'єднання встановлюється одноразово та утримується (тобто це persistent connection), чи створюється повторно для кожного повідомлення; у першому випадку наклад handshake розподіляється і вплив на час затримки операцій зменшується, у другому – витрати значно зростають. По-друге, вибір конкретного cipher-suite (наприклад AES-GCM чи ChaCha20-Poly1305) та наявність апаратної підтримки криптографії на пристрої можуть сильно впливати на споживання пам'яті та швидкодію. По-третє, негативна зміна RSS у DTLS-fallback підкреслює попередню думку, сформовану при аналізу отриманих результатів тесту методів автентифікації, щодо можливого розширення дослідження для магістерської роботи, та обмумовлює необхідність додаткового профілювання пам'яті для виключення подібних артефактів вимірювань. Інакше кажучи, варто здійснювати декілька незалежних ітерацій тестів і використовувати додаткові інструменти (як heap-профілювання, системний моніторинг тощо).

Отримані результати та проведений на основі них аналіз щодо цього тесту узгоджується з результатами досліджень публікацій, які теж аналізують вплив TLS на продуктивність MQTT. Зокрема, одна з таких робіт, у якій порівнюється продуктивність MQTT з використанням TLS та без нього, підтверджує наявність компромісу між рівнем захисту та затримкою і рекомендує підтримувати TLS-з'єднання у відкритому стані, щоб мінімізувати вплив на загальну продуктивність системи, що підтверджує доцільність сформованої, в попередньому абзаці, думки в ході аналізу результатів [16, с. 10-19].

Для кращого сприйняття загальної картини, будуть винесені оцінки для методів які базуються на результатах емпіричного дослідження (проведених тестів та окремих їх аналізів), у ході яких вимірювалися ключові показники продуктивності, такі як час виконання операцій, обсяг використаної пам'яті та стабільність роботи. На основі цих експериментальних даних було сформовано інтегральні оцінки за шкалою від 0 до 100 для кожної з шести критично важливих категорій: масштабованість, сумісність, рівень безпеки, енергоефективність, зручність у використанні та толерантність до помилок. Результати цієї комплексної оцінки наочно представлені на діаграмі (див. рис. 2.9).

Ця діаграма доповнить проведені окремі аналізи досліджуваних методів та надалі допоможе в синтезі практичних висновків та рекомендацій. Вона наочно ілюструє компроміси: наприклад, зазвичай методи з найвищим рівнем безпеки часто демонструють нижчу масштабованість та енергоефективність. Так це дозволить зробити обґрунтований вибір, спираючись не на теоретичну перевагу одного алгоритму, а на його відповідність специфічним вимогам IoT-систем. Наприклад, для мережі з сотень сенсорів, що живляться від акумуляторів, пріоритетом очевидно буде енергоефективність, тоді як для центрального шлюзу розумного будинку – високий рівень безпеки та сумісність. Таким чином, аналіз переходить від дещо абстрактних показників до контекстно-залежної оцінки придатності кожного методу.

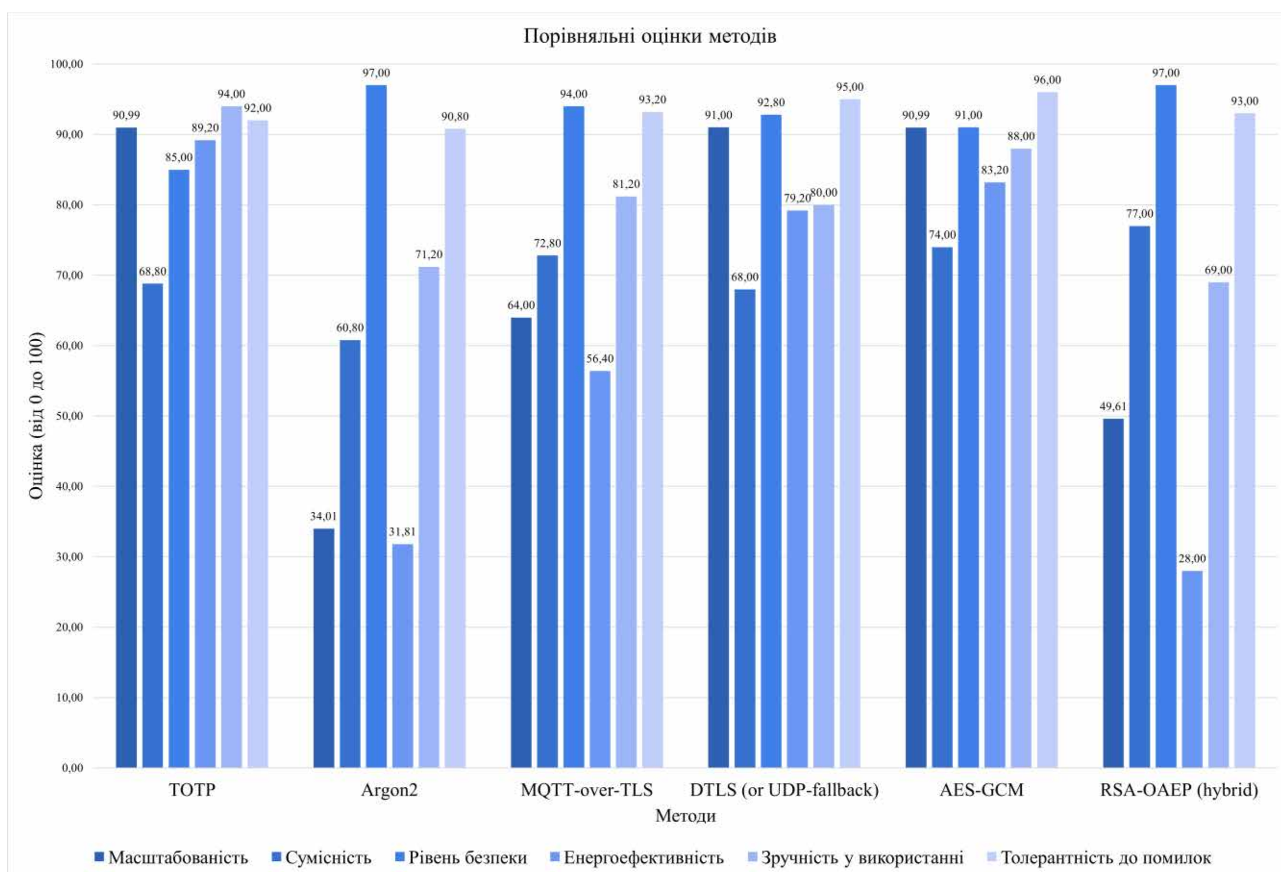

Рисунок 2.9 – Діаграма порівняльних оцінок методів за категоріями

Перше що відразу можна помітити з загального аналізу – всі досліджені методи демонструють високу толерантність до помилок, що відповідно підтверджує їхню надійність для розгортання в IoT-системах. Але водночас помітні значні відмінності у профілях їхньої ефективності, що дозволяє класифікувати їх за пріоритетними сценаріями використання. У категорії високопродуктивних рішень такими «безумовними лідерами» виявилися наступні методи: симетричний алгоритм шифрування AES-GCM, протокол захисту транспортного рівня DTLS та метод автентифікації TOTP. Окремо можна сказати що AES-GCM продемонстрував майже максимальні показники масштабованості та енергоефективності, а також найвищу відмовостійкість, що робить його ідеальним вибором для захисту потоків даних у реальному часі. Дещо схожі результати показав DTLS, що підтверджує його ефективність для забезпечення надійного та швидкого зв'язку у великих мережах пристроїв. Водночас TOTP абсолютно домінує за зручністю використання та показав

чудову енергоефективність, що робить його оптимальним для інтеграції вже в клієнтські застосунки, де важлива швидка та проста взаємодія.

З іншого боку знаходяться методи які орієнтовані на досягнення максимального рівня безпеки. Функція Argon2 та гібридна система шифрування RSA-OAEP отримали найвищі оцінки за рівнем безпеки, тобто це підтверджує їх можливість в протистоянні складним атакам. Однак такий рівень захисту досягається ціною значних обчислювальних ресурсів, що відобразилося в найнижчих показниках масштабованості та енергоефективності відповідно. Це робить Argon2 ідеальним для критично важливих офлайн-завдань, як-от наприклад хешування та зберігання паролів, але непридатним для інтенсивного обміну даними. RSA-OAEP, маючи схожий профіль продуктивності, вирізняється високою сумісністю, що наприклад може бути вирішальним фактором при інтеграції з існуючими криптографічними стандартами. Метод MQTT-over-TLS займає трохи проміжну позицію, пропонуючи збалансоване поєднання високої безпеки, відмовостійкості та сумісності, проте поступається «лідерам» у швидкодії та енергоефективності, що робить його таким собі «універсальним» рішенням для систем без екстремальних (високих) вимог до продуктивності.

2.4. Вихідні практичні висновки та рекомендації на основі тестів

В попередньому підрозділі було проведено комплексне дослідження ефективності програмних методів програмного захисту інформації які розглядаються. Підсумовуючи ключові результати проведених тестів, їх окремих аналізів та загального аналізу сформульовано практичні висновки та рекомендації. Нижче послідовно наведено ключові тези, що узагальнюють результати тестування з врахування особливостей систем IoT зокрема для систем типу розумний будинок.

Для передачі великих обсягів даних доцільно застосовувати симетричне шифрування в режимі AEAD, зокрема це AES-GCM. Вибір, зумовлений доволі

простою причиною – AES-GCM демонструє низьку затримку обробки як при шифруванні, так і при дешифруванні, а додатковий мережний наклад обмежується невеликим тегом автентифікації (приблизно 16 байт). Іншими словами, при потоковій передачі даних або наприклад в сценаріях, де важлива швидка реакція, AES-GCM дає найкращий баланс-компроміс між захистом і продуктивністю. Але при цьому слід пам'ятати технічну деталь-особливість: RSA-шифрування напряму не підходить для великих блоків через обмеження максимальної довжини повідомлення для OAEP-паддингу. Тому на практиці варто реалізовувати гібридну схему – симетричний ключ шифрується асиметрично, а сам контент (вміст) шифрується за допомогою AES-GCM.

Пряме виконання приватного дешифрування RSA виявилося ресурсно-інтенсивним – ця операція має фіксовані затрати часу і пам'яті (наклад), яка помітно перевищує витрати, пов'язані з AES. Такий результат є очікуваним а причина полягає в наступному – алгоритмічна складність операцій над великими цілими числами та аспекти самої реалізації криптографічних бібліотек. Як наслідок, приватне RSA-дешифрування на периферійних пристроях знижує загальну ефективність системи і разом з цим виникає проблема в її масштабуванні, що є критичним аспектом для таких систем. Через це практично випливає наступна рекомендація – виконувати приватні RSA-операції на більш потужних вузлах (сервери, IoT-шлюзи) або розглянути перехід на ефективніші асиметричні схеми (наприклад це може бути ECC), які при еквівалентній криптографічній стійкості забезпечують менші розміри ключів або і швидші приватні операції.

Важливість використання методів автентифікації безперечна, як вже було з'ясовано в першому розділі. З її боку доцільно розділити ролі між двома підходами: Argon2 має слугувати як механізм для довготривалого, стійкого зберігання паролів, тоді як TOTP – як швидкий та легкий другий фактор для кінцевих пристроїв. Просте пояснення-приклад: Argon2 – це «memory-hard» функція (вибаглива до пам'яті), її параметри (time_cost, memory_cost, parallelism) можна налаштовувати для підвищення захищеності проти апаратних атак, але ці

ж параметри роблять її дорогою для виконання на слабких вузлах у системі IoT. TOTP, що базується на HMAC та часовому факторі, виконується у діапазоні від мікро до мілісекунд і ідеально підходить для частих операцій на пристроях з обмеженими ресурсами. З огляду на вищесказане рекомендується наступне: хешування паролів виконувати на сервері (якщо передбачений в проєкті системи) або на ресурсному шлюзі, а на пристроях залишити лише генерацію та перевірку TOTP; у разі критичної необхідності у Argon2 на периферії – варто зменшити memory_cost і time_cost за обґрунтованої оцінки ризиків.

Результати порівняльного аналізу транспортних протоколів засвідчили про типовий компроміс, який є притаманним інженерним рішенням у сфері мережевої взаємодії. DTLS (повний DTLS поверх UDP) дає мінімальну латентність (затримку в часі) і малий системний слід (використання ресурсів, інтегральне навантаження), що робить його доволі привабливим для сценаріїв із жорстко визначними вимогами щодо відгуку. MQTT-over-TLS, натомість, забезпечує надійну доставку, порядок повідомлень і зручне управління сесіями, але відповідно накладає значні витрати за часом і пам’яттю – у проведених тестах середня латентність MQTT-over-TLS була в декілька десятків мілісекунд, тоді як DTLS/UDP-fallback продемонстрував мілісекундні або субмілісекундні затримки. Як практичний висновок щодо цього: вибір між DTLS і MQTT-TLS має визначатися пріоритетом системи, якщо наприклад критичною є швидкість реакції, перевагу слід надати DTLS; якщо пріоритетом є надійність доставки та простота інтеграції з брокером (в даному контексті мова про брокер-сервер), тоді MQTT-over-TLS є доречною опцією, але з необхідністю в резервуванні додаткових ресурсів.

Варто окреслити наступні уточнення та обмеження інтерпретації які мають важливе значення. Частина експериментального накладу залежить від того, чи з’єднання TLS утримується відкритим (persistent connection) чи створюється заново на кожне повідомлення, в останньому випадку handshake створює суттєві одноразові витрати. Також вибір набору шифрування (cipher-suite) та наявність апаратного прискорення криптографії (наприклад AES-NI) можуть значно

вплинути на часові витрати; а отже, для остаточного проєктування рекомендується попередньо провести тести на цільовому обладнанні і з тими конфігураціями протоколів, які плануються у розгортанні системи. Негативні або змінні значення ΔRSS слід розглядати ретельно: resident set size (RSS) підпадає під вплив ОС і поведінки менеджера пам’яті, тому для точного визначення пікового споживання було б корисно застосовувати додаткові інструменти профілювання пам’яті.

2.5. Постановка завдань для практичної реалізації

Завершуючи аналітичну частину, після аналізу отриманих результатів тестів було визначено набір технічних й організаційних вимог, які візьмуться за основу для практичної реалізації програмного забезпечення для системи розумного будинку, для якого відведено окремий розділ – практична частина в даній магістерській кваліфікаційній роботі. Цей підрозділ ставить за мету формалізацію функціональних та нефункціональних завдань (вимог), для реалізації додатку а також відповідних фреймворків і допоміжних інструментів.

Виходячи з аналізу існуючих підходів до забезпечення безпеки програмних засобів систем розумного будинку, було сформовано наступну основну вимогу до практичної частини дослідження: розробка власного модульного програмного продукту. Цей продукт повинен мати на меті інтегрувати та продемонструвати роботу ключових механізмів захисту, зберігаючи при цьому концептуальну відповідність системам, призначеним для побутового використання.

Відповідно до цієї мети, було визначено наступні ключові завдання для реалізації. Необхідно буде створити модуль автентифікації користувачів, який би включав надійне хешування паролів за допомогою Argon2id та реалізацію двофакторної автентифікації на основі TOTP, забезпечивши при цьому безпечне зберігання секретних ключів, наприклад через Windows DPAPI. Також важливим завданням має бути реалізація модуля взаємодії з MQTT-брокером, що

підтримує захищений обмін даними через TLS з автентифікацією за клієнтськими сертифікатами та налаштуванням прав доступу на брокері.

Для захисту безпосередньо даних, що передаються, випливає таке завдання як реалізувати гібридну схему шифрування: симетричне шифрування AES повідомлень з використанням сесійних ключів, які, у свою чергу, шифруються за допомогою асиметричного алгоритму RSA. Додатково, для зберігання стану системи та логів подій, передбачено створення модуля роботи з локальною реляційною базою даних, наприклад використовуючи можливості SQLite. Додатково варто зазначити що невід'ємною частиною має стати підсистема логуванням подій безпеки з їх збереженням у базі даних та можливістю перегляду, а також відповідно обов'язково і для механізму збору та відображення даних про продуктивність реалізованих криптографічних операцій.

3 ПРАКТИЧНА ЧАСТИНА

Переходячи від аналізу методів захисту інформації в системах розумного будинку, цей розділ буде присвячений безпосередньо практичній реалізації програмного засобу. Основна мета розробки полягає у створенні, в першу чергу демонстраційної платформи, яка дозволяє дослідити та візуалізувати роботу ключових механізмів безпеки, обґрунтованих у попередніх розділах як оптимальних та доцільних в реалізації, але тим не менш, вона може бути розширена для взаємодії з реальною «фізичною» системою в разі необхідності. Розроблений додаток моделює основні функції керування пристроями розумного будинку та зосереджується саме на практичному впровадженні процесів автентифікації користувачів, захисту каналу передачі даних за допомогою TLS, шифрування повідомлень за гібридною схемою AES+RSA та безпечного зберігання конфігураційних даних.

Сам програмний засіб створено з використанням мови C# та платформи Windows Forms (див. рис. 3.1), що забезпечило швидку розробку інтерфейсу для наочної демонстрації, цей вибір обґрунтовується ще й наявністю значного практичного досвіду який апробований в подібних проєктах на базі цих засобів. Взаємодія компонентів системи імітується через протокол MQTT з використанням локального брокера та захищеного з'єднання. Отримані результати функціонування та вимірювання продуктивності криптографічних операцій слугуватимуть основою для аналізу ефективності застосованих рішень безпеки. Підсумовуючи, ця розробка є програмним прототипом, що ілюструє застосування теоретичних засад та проведеного в попередньому розділі аналізу, у конкретному програмному продукті.

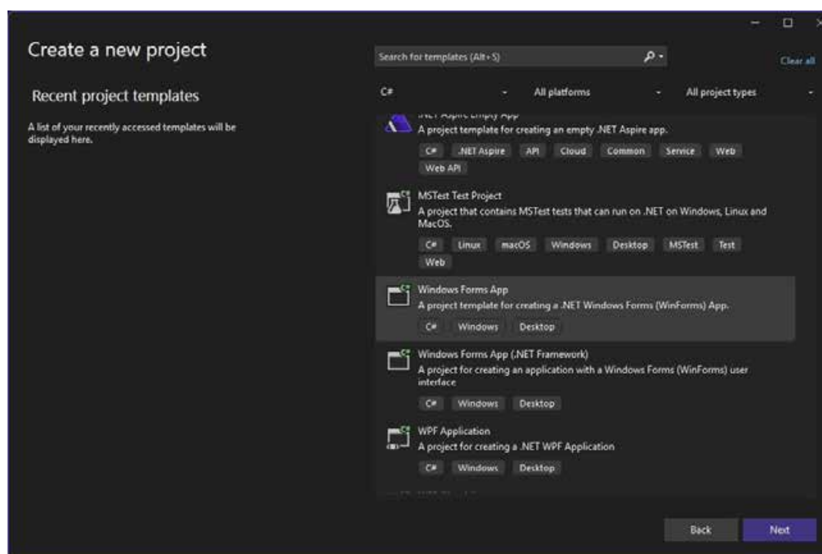

Рисунок 3.1 – Створення нового проєкту Windows Forms App

3.1. Архітектура програми

Ефективність та надійність програмного засобу, особливо у контексті безпеки, значною мірою визначаються обраними його архітектурними рішеннями та технічними рішеннями які лягли в основу. Для розробленої програми було обрано архітектуру, що базується на чіткому розділенні відповідальності між логічними компонентами (сервісами). Такий підхід сприяє модульності, полегшить тестування окремих частин системи та створить підґрунтя для подальшого розширення функціоналу. Наступні підпункти наводять загальну структуру проєкту, організацію бази даних та модель взаємодії через протокол MQTT.

3.1.1. Загальна структура проєкту

Програмний проєкт (засіб), побудовано за принципом багатошарової архітектури (модульної) з виділенням окремих сервісів для кожної специфічної задачі. Така декомпозиція створює інкапсуляційну логіку та зменшити загалом зв'язність між компонентами системи, а в разі необхідності внесення змін це спросить відповідну задачу.

Для програм на базі Windows Forms типовим та ключовим класом є MainForm, який виступає у ролі шару представлення (UI Layer) та точки взаємодії з користувачем. Він відповідає за відображення інтерфейсу, обробку дій користувача (натискання кнопок, вибір елементів тощо) та ініціацію відповідних операцій у бекенд-сервісах. Структура вікон та елементів керування визначена переважно в окремому файлі «MainForm.Designer.cs», що дозволяє візуальне редагування макету (див. рис. 3.2). Логіка керування видимістю панелей («loginPanel», «dashboardPanel» тощо) та обробники подій реалізовані відповідно у самому файлі «MainForm.cs».

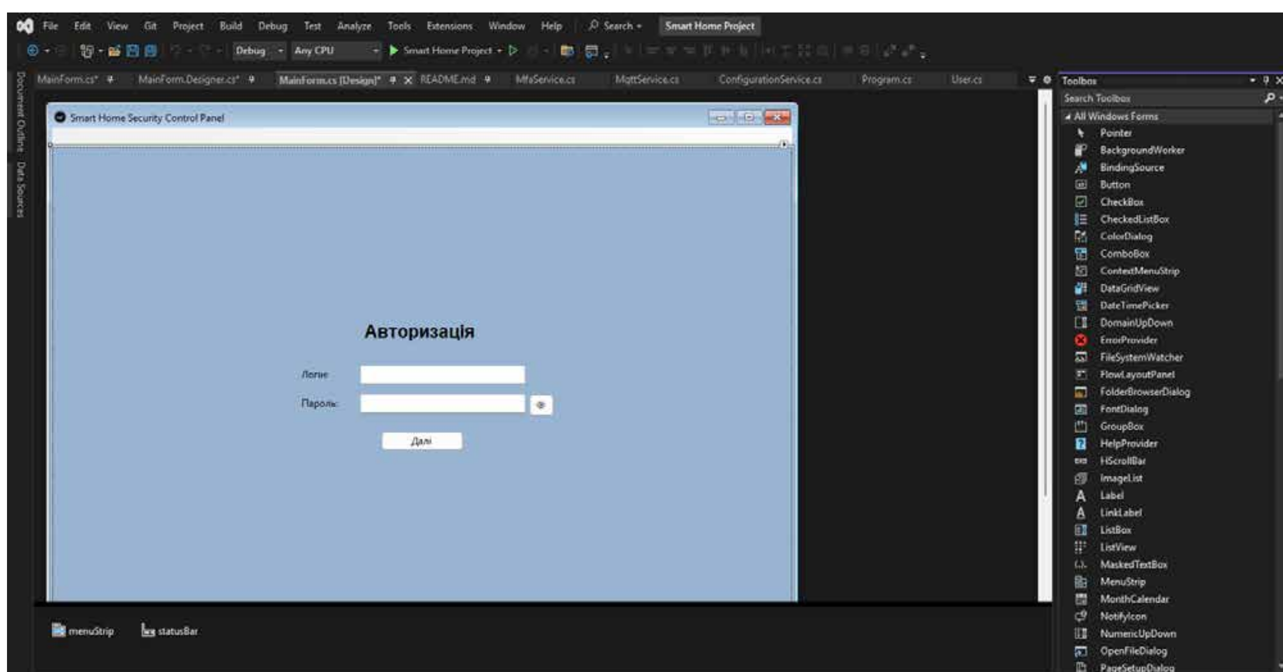

Рисунок 3.2 – Візуальне редагування та налаштування інтерфейсу

Ядром системи є набір сервісних класів, кожен з яких виконує чітко визначену функцію, нижче послідовно про кожен з них:

1. «DatabaseService» - забезпечує взаємодію з локальною базою даних SQLite. Відповідає за ініціалізацію бази даних та її структури при першому запуску, виконання CRUD-операцій (тобто Create, Read, Update, Delete) для сутностей (в рамках проєкту це користувачі, кімнати, пристрої) та запис подій до логів (звітів) безпеки. Цей сервіс відповідно інкапсулює всю логіку роботи з

SQL-запитами та об'єктами SQLiteConnection, SQLiteCommand. Більш детальніше про БД в наступному пункті.

2. «SecurityService» - централізує функції, пов'язані з автентифікацією на основі паролів. Основне завдання це забезпечити безпечне хешування паролів за допомогою алгоритму Argon2id під час створення користувача (у поточній реалізації – користувача за замовчуванням) та валідація введеного користувачем пароля шляхом порівняння його хешу зі збереженим у базі даних. Для отримання даних користувача (це хешу та солі) він взаємодіє з «DatabaseService».

3. «ConfigurationService» - реалізує механізм безпечного зберігання конфігураційних даних програми, зокрема секретного ключа для TOTP. Використовує Windows Data Protection API (DPAPI) для шифрування даних, прив'язуючи їх до поточного користувача операційної системи. Зашифровані дані зберігаються у локальному файлі config.dat в директорії проєкту.

4. «MfaService» - відповідає за функціонал двофакторної автентифікації на основі TOTP (Time-based One-Time Password). Він генерує секретний ключ при першому запуску (якщо він відсутній у конфігурації), зберігає його за допомогою «ConfigurationService», генерує поточні 6-значні коди для відображення користувачу (це зроблено для демонстраційної версії ще й спрощує процес налагодження), та перевіряє валідність коду, введеного користувачем, з урахуванням можливих часових розбіжностей.

5. «CryptographyService» - інкапсулює всю логіку симетричного (AES) та асиметричного (RSA) шифрування, а також їх комбінацію у гібридній схемі. Відповідає за завантаження RSA-ключів з файлів PEM, генерацію сесійних AES-ключів, виконання операцій шифрування (EncryptHybrid) та дешифрування (DecryptHybrid), а також за вимірювання часу виконання цих операцій за допомогою класу Stopwatch.

6. «MqttService» - керує взаємодією з MQTT-брокером. Відповідає за встановлення захищеного TLS-з'єднання з автентифікацією за клієнтським сертифікатом, обробку подій підключення та відключення, підписку на вказані топіки для отримання повідомлень (статусів пристроїв) та публікацію

повідомлень (команд для пристроїв) у відповідні топіки. Детальніше про взаємодію з MQTT в пункті 3.1.3.

Загальна взаємодія між компонентами побудована таким чином, що «MainForm» використовує екземпляри всіх вище згаданих сервісів для виконання необхідних операцій. Деякі сервіси також залежать від інших (наприклад, «SecurityService» від «DatabaseService», «MfaService» від «ConfigurationService»), що реалізовано через передачу залежностей у конструкторах. Як вже було сказано, така архітектура забезпечує належний рівень абстракції та полегшує потенційну майбутню заміну, розширення чи модифікацію окремих сервісів. Для кращого сприйняття архітектури програмного проєкту наведено UML діаграму компонентів – див. рис. 3.3.

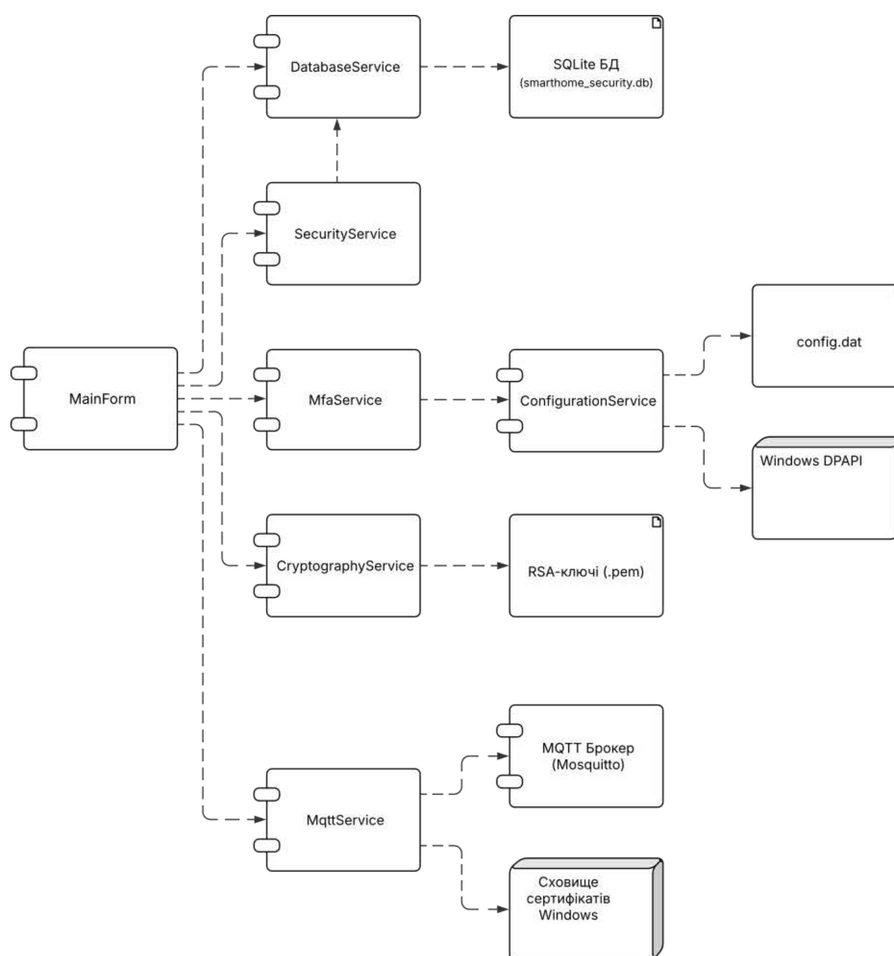

Рисунок 3.3 – UML діаграма компонентів «Загальна архітектура програми»

3.1.2. Структура бази даних

Для зберігання даних програми, а це включаючи інформацію про користувачів, конфігурацію розумного будинку та логи подій, було створено локальну реляційну базу даних (БД) використовуючи SQLite. Файл бази даних «smarthome_security.db» створюється автоматично при першому запуску програми у її робочій директорії, визначається файлом на мові SQL. Структура бази даних складається з п'яти основних таблиць, які розроблені для забезпечення нормалізації даних та підтримки всього необхідного функціоналу. Детальна структура БД наведена у вигляді ER-моделі – див. рис. 3.4.

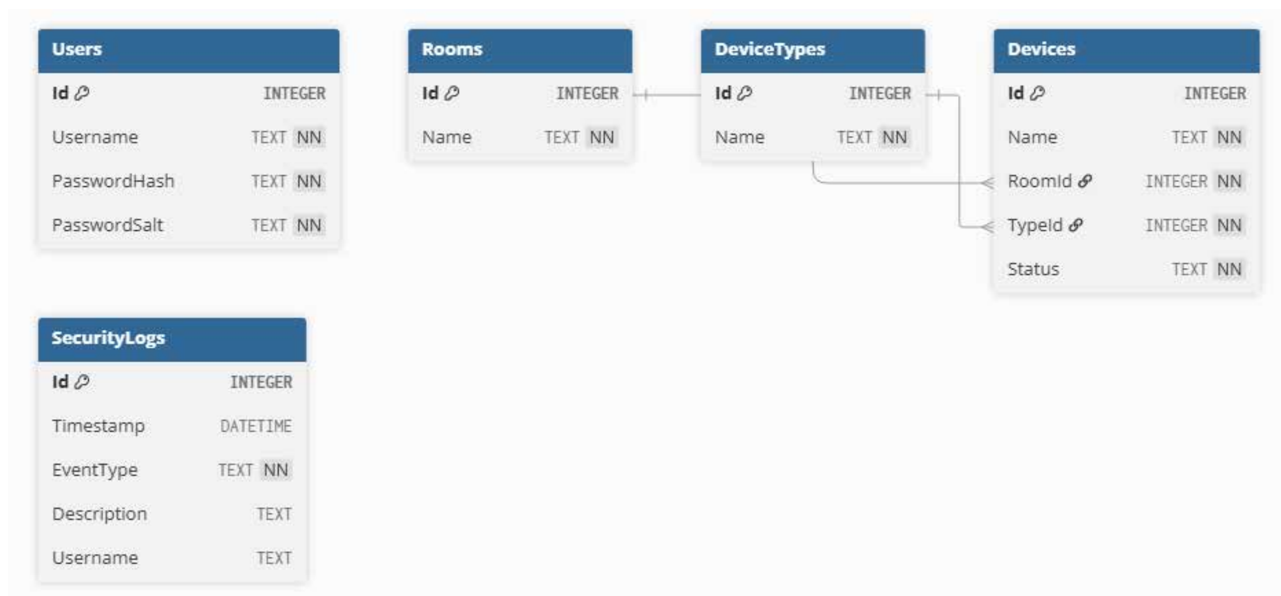

Рисунок 3.4 – ER-модель БД

Таблиця «Users» призначена для зберігання облікових даних користувачів. Включає унікальний ідентифікатор (Id), ім'я користувача (Username), хеш пароля (PasswordHash) та сіль (PasswordSalt), що використовувалася при хешуванні. Зберігання в БД саме хешу та солі замість відкритого пароля є фундаментальним загальноприйнятим принципом безпеки, адже при сценарії атаки, наслідком якої може бути компрометація БД, це вбереже дані користувачів. Приклад вмісту цієї таблиці БД наведено на рис. 3.5.

Рисунок 3.5 – Вміст таблиці «Users» (приклад зберігання паролів)

Таблиця «Rooms» містить відповідно просто перелік кімнат у будинку. Спрощена структура включає лише унікальний ідентифікатор (Id) та назву кімнати (Name), яка до речі також має бути унікальною.

Таблиця «DeviceTypes» це такий собі «довідник типів пристроїв», які підтримуються системою (наприклад щось загальне, «Лампа», «Термостат» тощо). Вона містить унікальний ідентифікатор (Id) та назву типу (Name). Цю окрему таблицю було сформовано під час нормалізації БД, і вона дозволяє легко додавати нові типи пристроїв у майбутньому.

Таблиця «Devices» є центральною для опису конфігурації розумного будинку. Вона містить інформацію про кожен окремий пристрій. Зв'язки «RoomId» та «TypeId» реалізовані як зовнішні ключі (Foreign KEY), що посилаються на відповідні ідентифікатори в таблицях «Rooms» та «DeviceTypes» відповідно. Поле Status зберігає поточний стан пристрою у вигляді рядка (наприклад, «Ввімкнено», «Вимкнено» чи «22°C»). Може здатися що зберігання стану у текстовому вигляді є менш структурованим, але це спрощує демонстраційну реалізацію для різних типів пристроїв без необхідності створювати окремі таблиці станів – тобто проведення наступних етапів нормалізації БД, що може бути кропітким процесом та не є темою даної роботи.

Таблиця «SecurityLogs» призначена для журналювання (логування, звітування) важливих подій. Це може бути події пов'язані з безпекою та функціонуванням системи. Кожен запис містить мітку часу (Timestamp), тип події (EventType), детальний опис (Description) та ім'я користувача (Username), якщо подія пов'язана з діями конкретного користувача. Загалом ця таблиця дозволяє відстежувати спроби входу, зміни станів пристроїв, помилки автентифікації тощо.

Взаємодія з БД, як вже було сказано, здійснюється через клас «DatabaseService», який використовує бібліотеку «System.Data.SQLite.Core» для виконання SQL-запитів. При ініціалізації сервіс перевіряє наявність таблиць та створює їх за потреби, а також заповнює таблиці («Rooms», «DeviceTypes») та таблицю «Devices» початковими даними при першому запуску.

3.1.3. Архітектура потоків даних MQTT

Взаємодія між програмою (тобто клієнтом) та імітованими пристроями розумного будинку реалізована за допомогою протоколу MQTT (Message Queuing Telemetry Transport), який є стандартом для комунікацій в сфері інтернету речей. Використовується архітектура «клієнт-брокер», де центральним вузлом виступає сам MQTT-брокер (в рамках даної роботи – локально встановлений Mosquitto). Програма підключається до брокера як клієнт і може як публікувати повідомлення (іншими словами – надсилати команди), так і підписуватися на топіки для отримання повідомлень (отримувати статуси).

З'єднання між клієнтом та брокером захищене за допомогою протоколу TLS 1.2. Автентифікація клієнта відбувається на основі X.509 сертифікатів: клієнт надає свій сертифікат, підписаний тим самим кореневим центром сертифікації (CA), що й сертифікат сервера. А брокер Mosquitto налаштований таким чином щоб вимагати клієнтський сертифікат (require_certificate true) та використовувати поле Common Name (CN) з сертифіката клієнта як ім'я користувача для подальшої авторизації (use_identity_as_username true).

Авторизація (мається на увазі в контексті визначення прав доступу до топіків) реалізована за допомогою файлу контролю доступу (ACL) на брокері, який надає користувачу smart-home-client права на читання та запис у відповідні топіки.

Для обміну повідомленнями використовуються наступні структури топіків:

1. Команди від клієнта до пристрою: marthome/devices/{deviceId}/command_hybrid
2. Статуси від пристрою до клієнта: smarthome/devices/+/status_hybrid

Всі повідомлення, що передаються, мають формат JSON та відповідно до реалізованих методів підлягають гібридному шифруванню. Повідомлення до шифрування містять ідентифікатор пристрою та команду (приклад, {"deviceId": 1, "status": "Ввімкнено"} або {"deviceId": 2, "temperature": 22.5}).

Після шифрування, повідомлення вже перетворюється на уніфікований JSON-контейнер, який забезпечує його конфіденційність. Забігаючи наперед, для наочності, буде наведена практична демонстрація цієї моделі взаємодії та реальний вигляд зашифрованого повідомлення, перехопленого за допомогою інструменту MQTT Explorer, яку можна побачити на рис. 3.6.

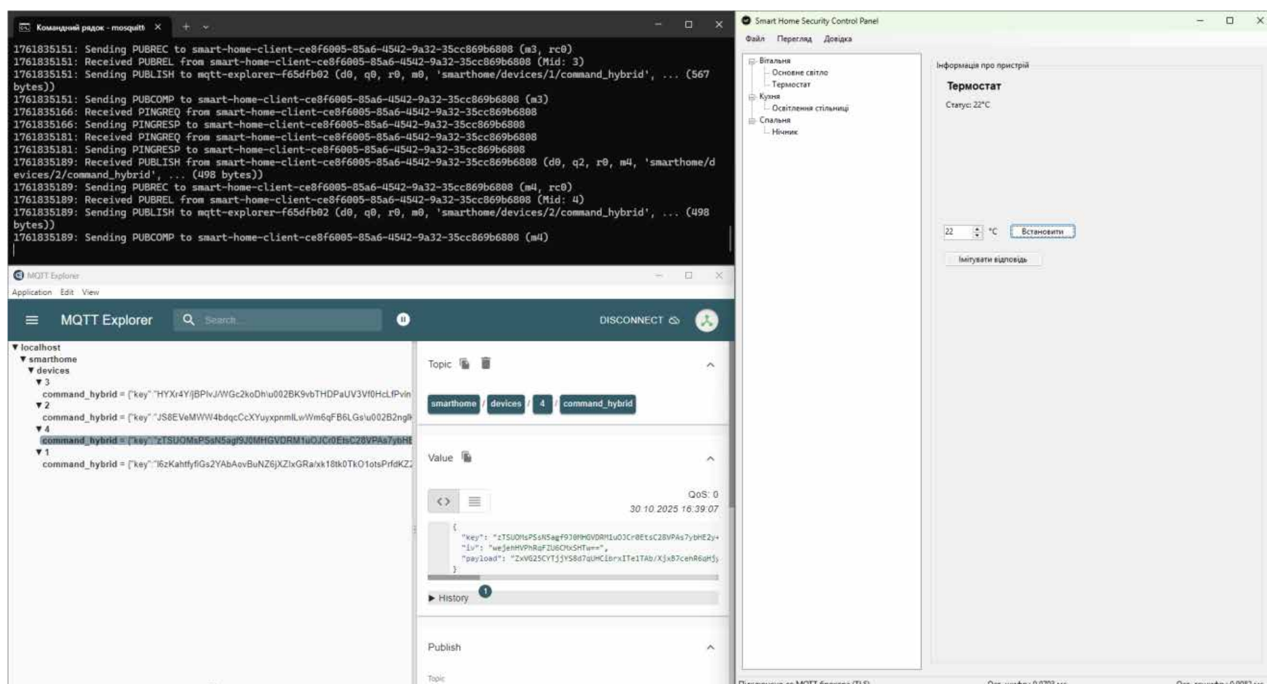

Рисунок 3.6 – Демонстрація передачі гібридного зашифрованого повідомлення

З рисунку 3.6, видно інтерфейс програми (вікно праворуч) який надсилає команду для пристроїв IoT. Лог брокера Mosquitto (вікно ліворуч зверху) фіксує публікацію повідомлення (PUBLISH). В цей же час інструмент MQTT Explorer (вікно ліворуч знизу), підписаний на топік, перехоплює це повідомлення. А його вміст є нечитабельним JSON-контейнером, що складається з трьох полів: key (це зашифрований сесійний ключ), iv (це вектор ініціалізації) та payload (це зашифровані дані команди). Уточнення – MQTT Explorer попередньо був налаштований та підписаний на топік з використанням TLS, а також надані всі необхідні ключи та сертифікати. Отже, результати наведені на рис. 3.6 підтверджують, що оригінальний вміст команди (deviceId та status) є скриті під час передачі.

3.2. Реалізація основних функціональних модулів програми

На основі розробленої архітектури було реалізовано ключові функціональні модулі. Вони забезпечують логіку роботи демонстраційної платформи. Кожен з модулів інкапсульовано в окремому сервісному класі, що відповідає за свою частину функціоналу, тоді як клас «MainForm» виступає координатором програми та шаром представлення. В цьому підрозділі буде детально описано реалізацію модулів інтерфейсу, роботи з базою даних та взаємодії з MQTT-брокером.

3.2.1. Модуль інтерфейсу користувача

Інтерфейс користувача є основною точкою взаємодії з програмним засобом та відповідає за візуалізацію даних і керування станом програми. Реалізований за допомогою технології Windows Forms, структура визначається у файлі «MainForm.Designer.cs» і це ж дає можливість візуально редагувати інтерфейс. Головний клас «MainForm.cs» можна сказати виступає у ролі

«контролера», та містить виключно логіку обробки подій та керування іншими сервісами.

При створенні інтерфейсу користувача було обрано наступний підхід - інтерфейс у вигляді єдиного вікна, тобто, замість відкриття щоразу нових вікон відбувається просто перемикання видимості панелей, що займають усю клієнтську область форми. Навігація між панелями керується набором приватних методів відображення (наприклад, «ShowLoginView», «ShowDashboardView», «ShowLogsView» тощо), кожен з яких приховує всі панелі, окрім цільової. На рис. 3.7 наведено вигляд 4 вікон програми: вікно авторизації, вікно двофакторної автентифікації, вікно панелі керування пристроями, вікно про автора.

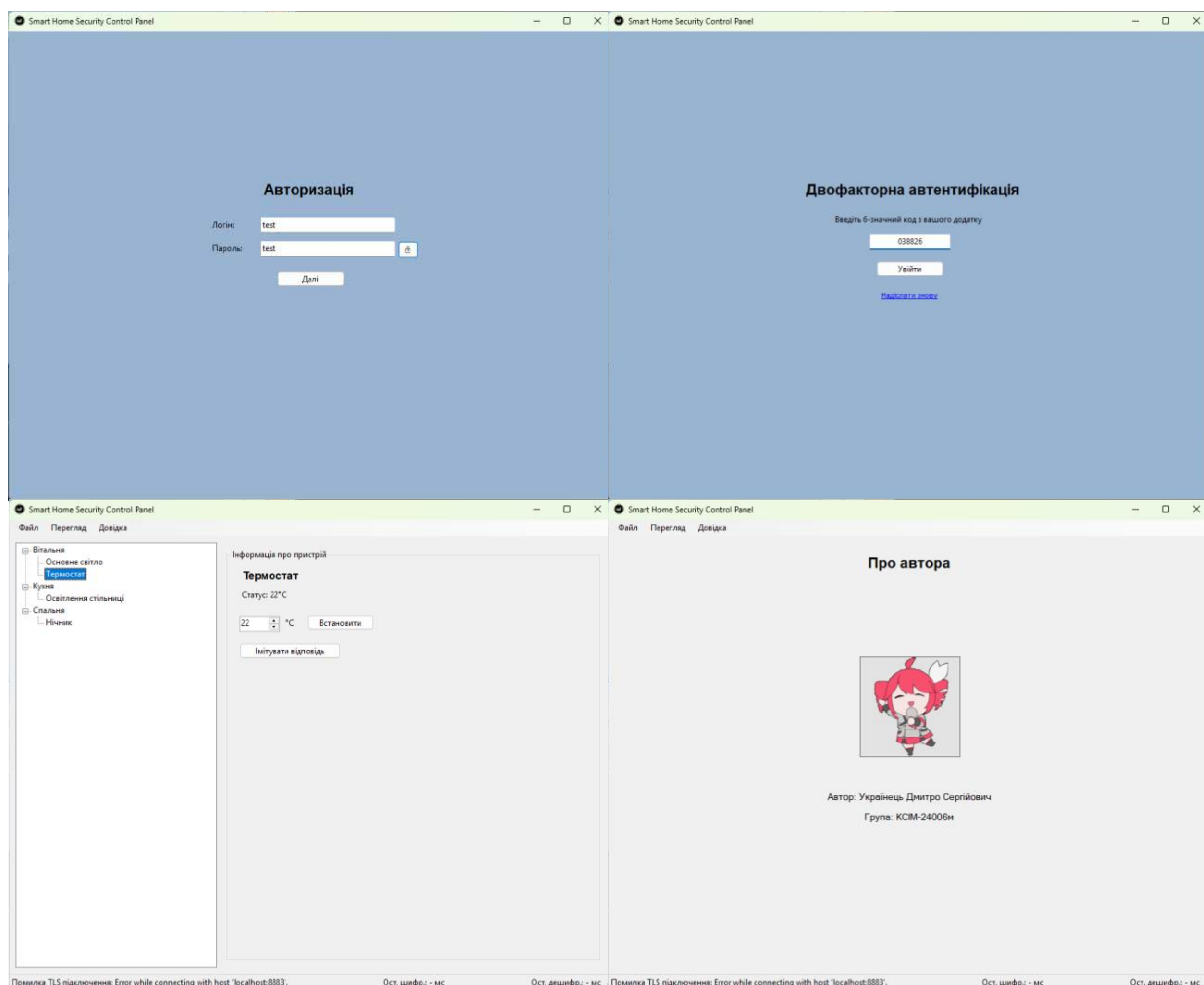

Рисунок 3.7 – Вигляд вікон програми

Управління життєвим циклом програми та сервісів відбувається у конструкторі «MainForm» та обробнику події «FormClosing». При запуску ініціалізуються всі інші необхідні сервіси та передаються залежності (наприклад, «SecurityService» отримує «DatabaseService»). Прив'язка всіх обробників подій до елементів керування відбувається централізовано окремо у методі «InitializeCustomComponents».

Можна виділити ключовою особливістю реалізації – повна асинхронна взаємодія з сервісами, що вимагають часу, зокрема з «MqttService». Виклик підключення («mqttService.ConnectAsync») та публікація повідомлень («mqttService.PublishAsync») виконуються асинхронно. Це необхідно щоб не блокувати потік інтерфейсу користувача. Отримання повідомлень від MQTT відбувається у фоновому потоці та безпечно оновлює інтерфейс користувача (UI) за допомогою механізму «InvokeRequired/Invoke», це є критично важливим для багатопотокових додатків Windows Forms. Для наочної демонстрації роботи системи також було реалізовано додаткові функції, такі як кнопка «Імітувати відповідь» для пристроїв, що демонструє повний цикл дешифрування та оновлення UI.

3.2.2. Модуль роботи з базою даних

Модуль роботи з даними реалізований у класі «DatabaseService», який забезпечує повну абстракцію від фізичного сховища, тобто локальної бази даних SQLite. Вибір SQLite зумовлений його простотою, серверною незалежністю та зберіганням всієї бази в єдиному файлі що є зручним рішенням та ідеально підходить для створеного програмного засобу. Сервіс використовує бібліотеку «System.Data.SQLite.Core» для прямого виконання SQL-запитів, що відповідно надає повний контроль над структурою та даними, оминаючи рівень ORM (наприклад, релевантний для мови C# це Entity Framework), який був би надлишковим а також затратним по часу для поставлених завдань.

При ініціалізації сервісу у конструкторі викликається метод «InitializeDatabase». Цей метод перевіряє наявність файлу БД. Якщо він відсутній – створюється, після чого виконується набір відповідних SQL-команд для створення всієї необхідної структури (це таблиці «Users», «Rooms», «DeviceTypes», «Devices», «SecurityLogs»). Інакше кажучи, це гарантує що програма завжди працюватиме з коректною схемою. Разом з першим створенням БД також викликається метод «AddDefaultData», який заповнює таблиці початковими типовими даними (кімнатами та пристроями). Ці початкові дані необхідні для демонстрації.

Для забезпечення безпеки та запобігання SQL-ін’єкціям (тип ін’єкційних атаках), для всіх методів, що приймають дані від користувача (наприклад це «GetUserByUsername», «UpdateDeviceStatus»), було реалізовано параметризовані запити. Тобто, дані передаються у запит через об'єкти «SQLiteParameter», а не шляхом об’єднання рядків. Це є стандартною та обов'язковою практикою при роботі з базами даних, яка корелює з даною темою дослідницької роботи, тому доречна, навіть обов’язкова, до реалізації. Приклад реалізації такого підходу наведено в лістингу коду 3.1 методу «GetUserByUsername» з коментарями-поясненнями.

Лістинг коду 3.1 – Фрагмент коду методу «GetUserByUsername», реалізація запобіганню SQL-ін’єкціям

```
public User? GetUserByUsername(string username)
{
    // Використання using гарантує звільнення ресурсів з'єднання
    using var connection = new SQLiteConnection(connectionString);
    connection.Open();
    string query = "SELECT Id, Username, PasswordHash, PasswordSalt FROM Users
WHERE Username = @Username LIMIT 1;"; // це SQL-запит
    using var command = new SQLiteCommand(query, connection);
    // Пряме додавання значення @Username до запиту
    // Це ↓ запобігає можливості SQL-ін'єкції
    command.Parameters.AddWithValue("@Username", username);
    using var reader = command.ExecuteReader();
    if (reader.Read()) // Увома якщо користувача знайдено
    {
```

Продовження лістингу коду 3.1

```
        return new User
        {
            Id = reader.GetInt32(0),
            Username = reader.GetString(1),
            PasswordHash = reader.GetString(2),
            PasswordSalt = reader.GetString(3)
        };
    }
    return null; // Коли користувача не знайдено
}
```

Методи для вибірки даних (наприклад, «GetRooms», «GetDevicesInRoom», «GetSecurityLogs») зчитують результати за допомогою «SQLiteDataReader» та вручну перетворюють (цей процес називається «mapping») отримані дані на об'єкти C# (класи-моделі Room, Device, SecurityLog), цим самим забезпечуючи чітке розділення між даними БД та об'єктною моделлю програми, ще одна здорова практика реалізації.

3.2.3. Модуль взаємодії з MQTT

Як вже було зазначено, комунікація з віртуальними пристроями реалізована через «MqttService», який інкапсулює всю логіку роботи з протоколом MQTT за допомогою бібліотеки MQTTnet (використовується версії 3.1.2). Цей модуль відповідає за встановлення з'єднання, публікацію команд та підписку на статуси.

Найважливішою частиною реалізації є метод «ConnectAsync», який налаштовує захищене з'єднання з брокером Mosquitto через протокол TLS 1.2. При налаштуванні з'єднання виконується двостороння автентифікація: програма не лише перевіряє сертифікат сервера, але й надає свій клієнтський сертифікат. Клієнтський сертифікат завантажується не з файлу, а безпосередньо зі сховища сертифікатів поточного користувача Windows за його унікальним ідентифікатором («Thumbprint»). Це є більш безпечним методом зберігання.

Ключовим елементом є реалізація власного обробника валідації (перевірки, затвердження) сертифіката сервера («CertificateValidationCallback»). Оскільки в демонстраційному середовищі використовується самопідписаний кореневий сертифікат, тому стандартна перевірка буде зазнавати невдачі. Лістинг коду 3.2 демонструє як обробник перехоплює цю помилку та виконує валідацію вручну, код наведений з коментарями-поясненнями до нього.

Лістинг коду 3.2 – Фрагмент коду методу «ConnectAsync» з «MqttService» (налаштування TLS)

```
// ... (відбувається пошук та завантаження clientCert зі сховища Windows) ...
var clientCertificates = new List<X509Certificate> { clientCert };
// Налаштування параметрів TLS
var tlsParameters = new MqttClientOptionsBuilderTlsParameters
{
    UseTls = true,
    SslProtocol = System.Security.Authentication.SslProtocols.Tls12,
    Certificates = clientCertificates, // Надання клієнтського сертифіката
    // Кастомний обробник валідації сертифіката сервера
    CertificateValidationCallback = (cert, chain, errors, options) =>
    {
        if (caCert == null) return false; // Якщо немає CA для перевірки
        if (errors == SslPolicyErrors.None) return true; // Якщо сертифікат валідний
        // Обробка помилки самопідписаного сертифіката
        if (errors == SslPolicyErrors.RemoteCertificateChainErrors)
        {
            // Налаштування політики перевірки
            chain.ChainPolicy.RevocationMode = X509RevocationMode.NoCheck;
            chain.ChainPolicy.VerificationFlags =
X509VerificationFlags.AllowUnknownCertificateAuthority;
            chain.ChainPolicy.ExtraStore.Add(caCert); // Додаємо наш CA до довірених
            var serverCert = new X509Certificate2(cert);
            var isValid = chain.Build(serverCert);
            if (isValid)
            {
                var rootCert = chain.ChainElements[chain.ChainElements.Count -
1].Certificate; // Перевіряємо чи це наш CA
                return rootCert.Thumbprint == caCert.Thumbprint;
            }
        }
        return false; // Відхиляємо сертифікат
    }
};
// Створення опцій підключення MQTT клієнта.
var options = new MqttClientOptionsBuilder()
    .WithTcpServer("localhost", 8883) // Захищений порт
    .WithClientId($"smart-home-client-{Guid.NewGuid()}")
    .WithTls(tlsParameters) // Застосування налаштувань TLS
    .Build();
// ... (далі виклик await mqttClient.ConnectAsync()) ...
```

3.3. Реалізація основних механізмів безпеки та їх аналіз

Попередні підрозділи описували архітектуру та основні функціональні модулі розробленої програми. Цей підрозділ фокусується на деяких деталях практичної реалізації та аналізі ключових механізмів безпеки, чому присвячена дана магістерська робота. Далі розглядається впровадження багатофакторної автентифікації, захист даних на рівні каналу та повідомлень, а також аналіз продуктивності реалізованих криптографічних алгоритмів.

3.3.1. Реалізація механізмів автентифікації та контролю доступу

Було реалізовано багатоетапний процес автентифікації для ідентифікації користувачів. На першому рівні, при перевірці пароля, система уникає зберігання будь-яких зворотних облікових даних. Замість цього, «SecurityService» натомість використовує рекомендований для хешування паролів алгоритм Argon2id. При створенні користувача генерується криптографічний рядок (сіль, PasswordSalt), яка разом з хешем пароля (PasswordHash) зберігається у БД, як було наведено раніше в рис. 3.5. Процес валідації (ValidatePassword) відтворює хеш із введеного пароля та збереженої солі, використовуючи ті ж самі параметри (це: ступінь паралелізму, обсяг пам'яті, кількість ітерацій), та порівнює результат із збереженим хешем. В результаті це унеможливлює відновлення оригінального пароля.

Другим рівнем захисту виступає двофакторна автентифікація на основі одноразових паролів (TOTP), реалізована через «MfaService» та бібліотеку «OtpNet» (див. лістинг коду 3.3). Для кожного користувача (в рамках демонстрації створено один типовий обліковий запис «admin» та пароль «password», в реальних умовах варто слідувати рекомендаціям визначеними в попередньому розділі) генерується унікальний секретний ключ. Цей ключ не зберігається у відкритому вигляді. Натомість, він передається сервісу «ConfigurationService», який відповідає за його безпечне зберігання. Далі

«ConfigurationService» використовує вбудований механізм «Windows Data Protection API» (DPAPI) для його шифрування. Ці дані зашифровані таким чином що прив’язуються до облікового запису користувача Windows і можуть бути дешифровані лише тим самим користувачем на тому ж комп'ютері, що забезпечує надійний захист ключа TOTP від несанкціонованого доступу, навіть розглядаючи випадок компрометації файлової системи.

Лістинг коду 3.3 – Використання Windows DPAPI для захисту секрету TOTP

```
// Це процес Шифрування
byte[] encryptedData = ProtectedData.Protect(data, entropy,
DataProtectionScope.CurrentUser
);
return Convert.ToBase64String(encryptedData);
// Це Дешифрування
byte[] encryptedData = Convert.FromBase64String(encryptedText);
byte[] decryptedData = ProtectedData.Unprotect(encryptedData, entropy,
DataProtectionScope.CurrentUser
);
return Encoding.UTF8.GetString(decryptedData);
```

Контроль доступу реалізовано на рівні MQTT-брокера. Завдяки використанню TLS-автентифікації за клієнтськими сертифікатами, брокер Mosquitto ідентифікує клієнта за полем Common Name (CN) його сертифіката. Подальший доступ до топіків чітко слідує правилам файлу списку контролю доступу (acl.conf) на сервері, що дозволяє визначати права на публікацію та підписку лише для довірених, тобто автентифікованих клієнтів.

Окрім автентифікації користувача, ключовим елементом є автентифікація самого програмного клієнта на MQTT-брокері. Цей процес поєднує перевірку TLS-сертифіката з подальшою авторизацією на основі списку контролю доступу (ACL). Для кращого розуміння було створено діаграму послідовності яка наведена на рис. 3.8 та демонструє цей механізм взаємодії покроково.

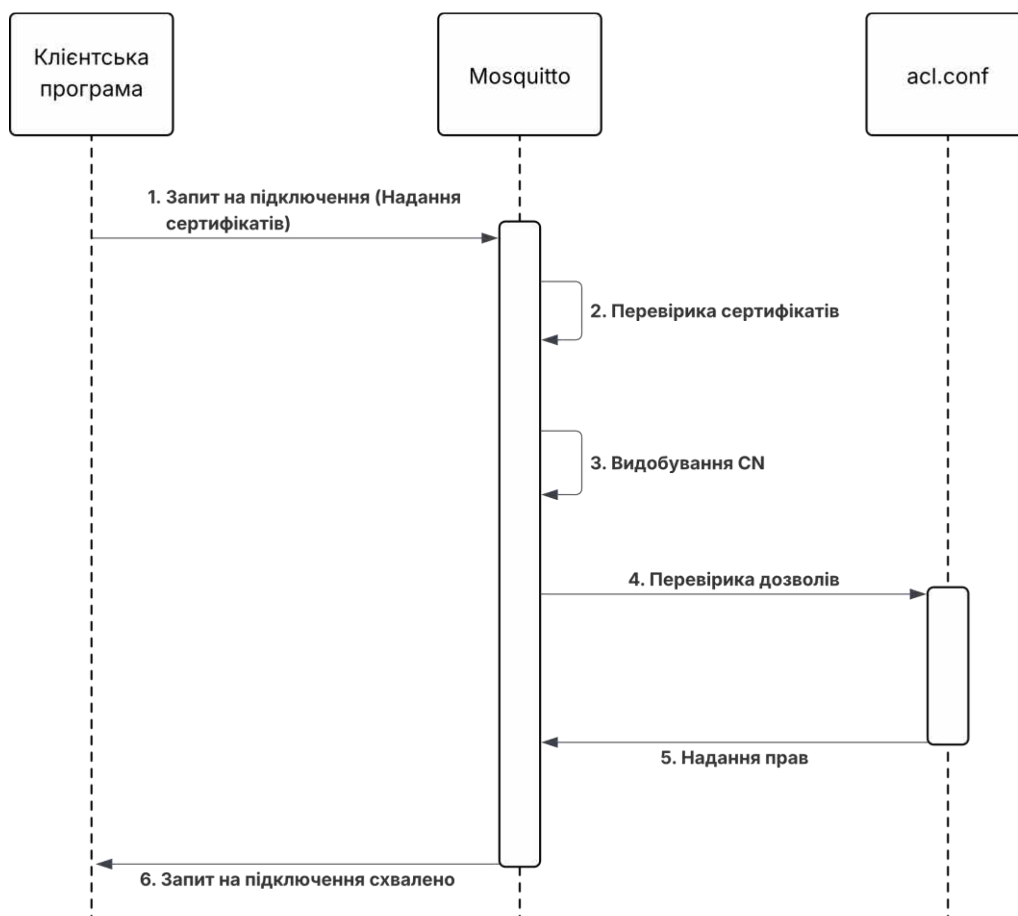

Рисунок 3.8 – UML діаграма послідовності автентифікації MQTT-клієнта та перевірки прав доступу

3.3.2. Реалізація гібридної схеми шифрування даних

Для захисту вмісту MQTT-повідомлень від перехоплення, навіть всередині захищеного TLS-каналу, було впроваджено другий, додатковий, рівень шифрування безпосередньо самих даних. Обрано гібридну схему, про яку було згадано в попередньому розділі, яка поєднує швидкість симетричного шифрування (AES) з надійністю асиметричного (RSA) для обміну ключами. Ця логіка інкапсульована відповідно у «CryptographyService».

Процес шифрування повідомлення (наприклад, команди для пристрою) відбувається наступним чином (див. лістинг коду 3.4): спочатку генерується випадковий одноразовий (сесійний) 32-байтний ключ та 16-байтний вектор ініціалізації (IV) для алгоритму AES-256.

Лістинг коду 3.4 – Фрагмент коду гібридного шифрування

```
// Генерація випадкового сесійного AES-ключ та вектора ініціалізації (IV)
using var aes = Aes.Create();
aes.KeySize = 256;
aes.GenerateKey();
aes.GenerateIV();
byte[] sessionKey = aes.Key;
byte[] iv = aes.IV;
// Шифрація даних (plainText) за допомогою сесійного AES-ключа
byte[] encryptedDataBytes = EncryptAes(Encoding.UTF8.GetBytes(plainText),
sessionKey, iv);
// Шифрація сесійного AES-ключ за допомогою публічного RSA-ключа
byte[] encryptedSessionKeyBytes = rsaPublicKeyProvider.Encrypt(
    sessionKey,
    RSAEncryptionPadding.OaepSHA256
);
// Та повертаємо всі компоненти у вигляді Base64-рядків
return (
    Convert.ToBase64String(encryptedSessionKeyBytes),
    Convert.ToBase64String(iv),
    Convert.ToBase64String(encryptedDataBytes)
);
```

Вхідне повідомлення (у вигляді JSON-рядка) шифрується цим сесійним ключем. Після цього вже сам сесійний ключ шифрується заздалегідь відомим публічним RSA-ключем пристрою (який завантажується з файлу «device_public.pem»). Фінальне повідомлення, що відправляється в MQTT, є JSON-контейнером, який містить три компоненти у форматі Base64 (згідно з рис. 3.6, вікно MQTT Explorer): зашифрований сесійний ключ, публічний вектор ініціалізації та зашифрований основний текст повідомлення.

Процес дешифрування відбувається відповідно у зворотному порядку. Як тільки сервіс отримує повідомлення він спочатку розшифровує сесійний ключ, використовуючи приватний RSA-ключ програми (знову з файлу «device_private.pem»). Отримавши оригінальний сесійний ключ та вектор, програма використовує їх для дешифрування основного тіла повідомлення (payload) за допомогою AES, відновлюючи таким чином початкову JSON-команду. В результаті реалізоване наскрізне шифрування, в якому лише кінцевий отримувач, що володіє відповідним приватним ключем, може прочитати вміст повідомлення.

3.3.3. Тестування та аналіз продуктивності

Піж час перевірки коректності роботи всієї системи було використано комбінований підхід. Працездатність та коректність функціонування двосторонньої комунікації та дешифрування перевірялися за допомогою кнопки «Імітувати відповідь», яка генерувала, шифрувала та передавала повідомлення безпосередньо в обробник «MqttService_MessageReceived» що дозволило протестувати всі етапи дешифрування та оновлення інтерфейсу. Для моніторингу реальної передачі даних через брокер використовувався зовнішній інструмент MQTT Explorer, який, будучи підключеним з використанням клієнтських сертифікатів, підтверджував отримання зашифрованих повідомлень у відповідних топіках, рис. 3.6.

Однією з нефункціональних вимог, раніше визначених, був аналіз продуктивності реалізованих криптографічних методів. Тому для цього у «CryptographyService» було вбудовано механізм вимірювання часу на основі класу Stopwatch. Вимірювання часу запускається на початку кожного виклику EncryptHybrid та DecryptHybrid і відповідно зупиняється одразу після завершення всіх криптографічних операцій.

Результати вимірювань зберігаються у двох списках, що містять 10 останніх значень для шифрування та дешифрування відповідно. На основі цих даних обчислюється середній час виконання, а також відображається останній час. Всі ці дані виводяться на окрему панель «Статистика продуктивності», див. рис. 3.9. Проведені вимірювання показали стабільні результати: час гібридного шифрування (AES та RSA-Encrypt) та дешифрування (RSA-Decrypt та AES-Decrypt) стабільно знаходиться, приблизно, в діапазоні від 0.3 до 0.8 мілісекунд. Такі низькі показники свідчать про те, що накладні витрати на реалізацію надійної гібридної схеми шифрування є мінімальними і не створюють суттєвого впливу на продуктивність системи керування розумним будинком.

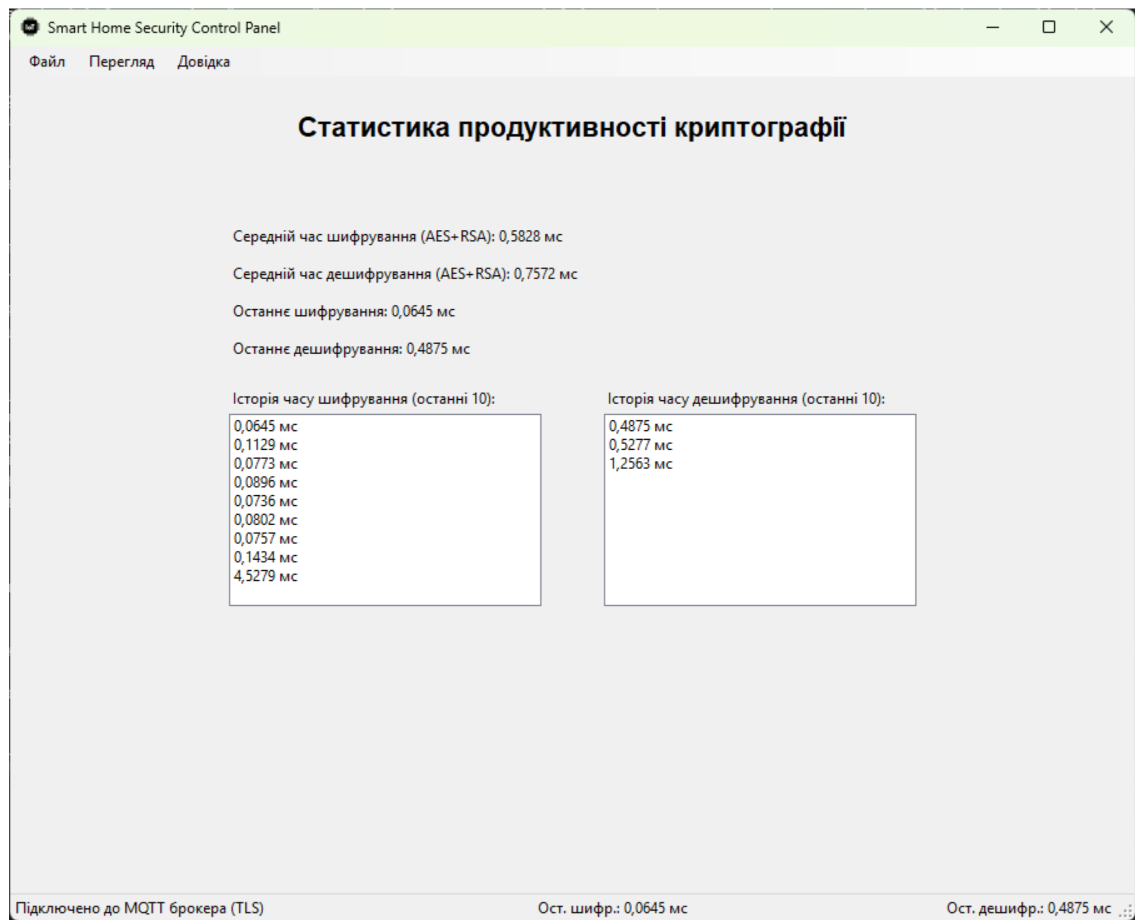

Рисунок 3.9 – Вигляд інтерфейсу статистика продуктивності криптографії

ВИСНОВКИ

В рамках даної магістерської кваліфікаційної роботи були отримані наступні головні теоретичні та практичні результати:

1. Проведено аналіз сучасних систем типу розумний будинок, їх архітектур, програмних засобів та встановлено ключові проблеми інформаційної безпеки, характерні для IoT-пристроїв з обмеженими апаратними ресурсами.
2. Розроблено методологію для проведення емпіричного дослідження продуктивності програмних засобів захисту, що включає тести на часову затримку, використання пам'яті та обчислювальну складність в обраному контрольованому програмному середовищі.
3. Здійснено кількісний аналіз захищених транспортних протоколів (зокрема це MQTT з TLS/DTLS), встановлення обчислювальну «ціну» початкового рукостискання (процес handshake) та переваги сучасних механізмів.
4. Проведено порівняльний аналіз продуктивності симетричних (AES-GCM) та асиметричних (RSA/ECC) алгоритмів шифрування, і також механізмів автентифікації (TOTP) та стійкого хешування (Argon2id).
5. На основі аналізу було доведено, що ефективна безпека для IoT-систем обмежених в ресурсах досягається не одним універсальним методом, а збалансованим (гібридним) підходом, та сформовано практичні рекомендації щодо їх оптимального поєднання.
6. Запропоновано та обґрунтовано багаторівневу архітектуру безпеки для програмного комплексу, який поєднує захист на транспортному, прикладному та аутентифікаційному рівнях.
7. Розроблено власне програмне забезпечення (на мові C# з використанням Windows Forms) для демонстрації та випробування запропонованої архітектури. Яке також інтегрує досліджені збалансовані механізми безпеки.

8. Запропонований гібридний підхід та результати емпіричного дослідження можуть бути використані для майбутніх розробок безпечних та водночас продуктивних програмних засобів для систем типу розумний будинок.

9. За рахунок гнучкої архітектуру створеного програмного забезпечення, його можна адаптувати в майбутньому для інтеграції та керування реальними фізичними компонентами системи розумного будинку.

10. По завершенню емпіричного дослідження та розділу 3 роботи, було покращено скрипт присвячений тестування протоколу MQTT з врахуванням отриманого нового практичного досвіду в ході роботи. Отримані уточнені графіки, що деталізують метрики протоколу MQTT та підкріплюють зроблені висновки, винесено у Додаток А, рис. А.1 та А.2.

11. Для можливості відтворення результатів дослідження та практичної частини даної магістерської кваліфікаційної роботи, вихідний код використаних тестових скриптів та розробленого програмного забезпечення було розміщено у відкритих репозиторіях GitHub. Доступ до них можна отримати за відповідним QR-кодом які наведено у Додатку Б, рис. Б.1 та Б.2.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. A review of Internet of Things: qualifying technologies and boundless horizon / Goel S. S та ін. *Journal of Reliable Intelligent Environments*. 2021. Т. 7. С. 23–33. URL: https://doi.org/10.1007/s40860-020-00127-w (дата звернення: 13.11.2024).

2. A Common Architecture-Based Smart Home Tools and Applications Forensics for Scalable Investigations / Kim S та ін. *Computers, Materials & Continua*. 2025. Т. 83, № 1. С. 661–683. URL: https://doi.org/10.32604/cmc.2025.063687 (дата звернення: 13.11.2024).

3. Zenarmor. Best IoT Device Management Tools - zenarmor.com. *Zenarmor - Agile Service Edge Security*. URL: https://www.zenarmor.com/docs/network-basics/best-iot-device-management-tools (дата звернення: 05.02.2025).

4. A Guide to IoT Gateways | Particle. *Particle*. URL: https://www.particle.io/iot-guides-and-resources/iot-gateway (дата звернення: 10.02.2025).

5. Internet of Things (IoT) – Gateways. *GeeksforGeeks*. URL: https://www.geeksforgeeks.org/computer-networks/internet-of-things-iot-gateways (дата звернення: 10.02.2025).

6. What is Azure Internet of Things (IoT). *Microsoft Learn*. URL: https://learn.microsoft.com/en-us/azure/iot/iot-introduction (дата звернення: 12.11.2025).

7. Annual number of IoT attacks global 2022| Statista. *Statista*. URL: https://www.statista.com/statistics/1377569/worldwide-annual-internet-of-things-attacks/ (дата звернення: 15.10.2024).

8. Botto-Tobar M, Rehan S, Prakash R. Protecting Smart Home from Cybersecurity Threats Strategies for Homeowners. *Journal of Cybersecurity and Information Management*. 2023. Т. 12, № 2. С. 83–98. URL: https://doi.org/10.54216/JCIM.120206 (дата звернення: 12.11.2024).

9. Information technology — Security techniques — Lightweight cryptography : ISO/IEC 29192 / International Organization for Standardization. 2012.

10. DDoS in the IoT: Mirai and Other Botnets / Kolias C та ін. *Computer*. 2017. Т. 50, № 7. С. 80–84. URL: https://doi.org/10.1109/mc.2017.201 (дата звернення: 12.11.2024).

11. Scarfone K, Mell P. Guide to Intrusion Detection and Prevention Systems (IDPS). *NIST Special Publication (SP)*. 2007. № 800-94. С. 75. URL: https://csrc.nist.gov/pubs/sp/800/94/final (дата звернення: 13.11.2024).

12. Abomhara M, Køien G. Security and privacy in the Internet of Things: Current status and open issues. *2014 International Conference on Privacy and Security in Mobile Systems (PRISMS)*. 2014. С. 80–99. URL: https://doi.org/10.1109/PRISMS.2014.6970594 (дата звернення: 13.11.2024).

13. Kumar Y, Kumar A. Securing Internet of Things Using RSA-OAEP encryption scheme. *Journal of Emerging Technologies and Innovative Research (JETIR)*. 2018. Т. 5, № 12. С. 37–40. URL: https://www.jetir.org/papers/JETIR1812407.pdf (дата звернення: 18.11.2024).

14. Suárez-Albela M, Suárez-Albela M, Fernández-Caramés T. M. A Practical Evaluation on RSA and ECC-Based Cipher Suites for IoT High-Security Energy-Efficient Fog and Mist Computing Devices. *Sensors*. 2018. Т. 18, № 11. С. 3868. URL: https://www.mdpi.com/1424-8220/18/11/3868 (дата звернення: 10.03.2025).

15. Biryukov A, Dinu D, Khovratovich D. Argon2: New Generation of Memory-Hard Functions for Password Hashing and Other Applications. *2016 IEEE European Symposium on Security and Privacy (EuroS&P)*. 2016. С. 1–16. URL: https://ieeexplore.ieee.org/document/7467361 (дата звернення: 10.03.2025).

16. Gavriilidis N. O., Halkidis S. T., Petridou S. Empirical Evaluation of TLS-Enhanced MQTT on IoT Devices for V2X Use Cases. *Applied Sciences*. 2025. Т. 15, № 15. С. 8398. URL: https://doi.org/10.3390/app15158398 (дата звернення: 16.09.2025).

17. .NET cryptography model - .NET. *Microsoft Learn: Build skills that open doors in your career*. URL: https://learn.microsoft.com/en-us/dotnet/standard/security/cryptography-model (дата звернення: 16.09.2025).

18. TOTP: Time-Based One-Time Password Algorithm / M'Raihi D та ін. *Request for Comments (RFC)*. 2011. № 6238. С. 1–14. URL: https://doi.org/10.17487/RFC6238 (дата звернення: 17.09.2025).

19. About SQLite. *SQLite Home Page*. URL: https://www.sqlite.org/about.html (дата звернення: 17.09.2025).

20. GitHub - dotnet/MQTTnet: MQTTnet is a high performance .NET library for MQTT based communication. *GitHub*. URL: https://github.com/dotnet/MQTTnet (дата звернення: 17.09.2025).

21. Digital Identity Guidelines: Authentication and Authenticator Management / Temoshok D та ін. *NIST Special Publication (SP)*. 2025. № 800-63B-4. С. 129. URL: https://doi.org/10.6028/NIST.SP.800-63B-4 (дата звернення: 17.09.2025).

22. Tran N. K, Babar M. A, Boan J. Integrating blockchain and Internet of Things systems: A systematic review on objectives and designs. *Journal of Network and Computer Applications*. 2021. Т. 173, № 102844. URL: https://doi.org/10.1016/j.jnca.2020.102844 (дата звернення: 17.09.2025).

23. Network-based multidimensional moving target defense against false data injection attack in power system / Hu Y та ін. *Computers & Security*. 2021. Т. 107, № 102283. URL: https://doi.org/10.1016/j.cose.2021.102283 (дата звернення: 17.09.2025).

24. Introduction to Public Key Technology and the Federal PKI Infrastructure / Kuhn R та ін. *NIST Special Publication (SP)*. 2001. № 800-32. С. 59. URL: https://doi.org/10.6028/NIST.SP.800-32 (дата звернення: 17.09.2025).

ДОДАТОК А
ДОДАТКОВІ РЕЗУЛЬТАТИ ТЕСТУВАННЯ ПРОДУКТИВНОСТІ ПРОТОКОЛУ MQTT

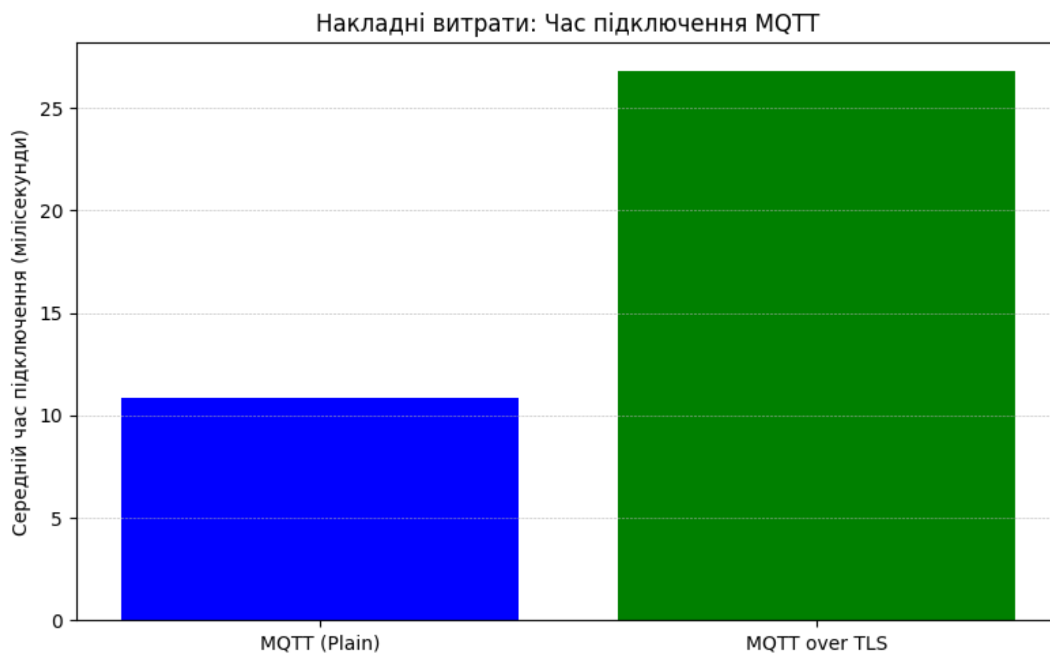

Рисунок А.1 – Графік середнього часу встановленого з'єднання

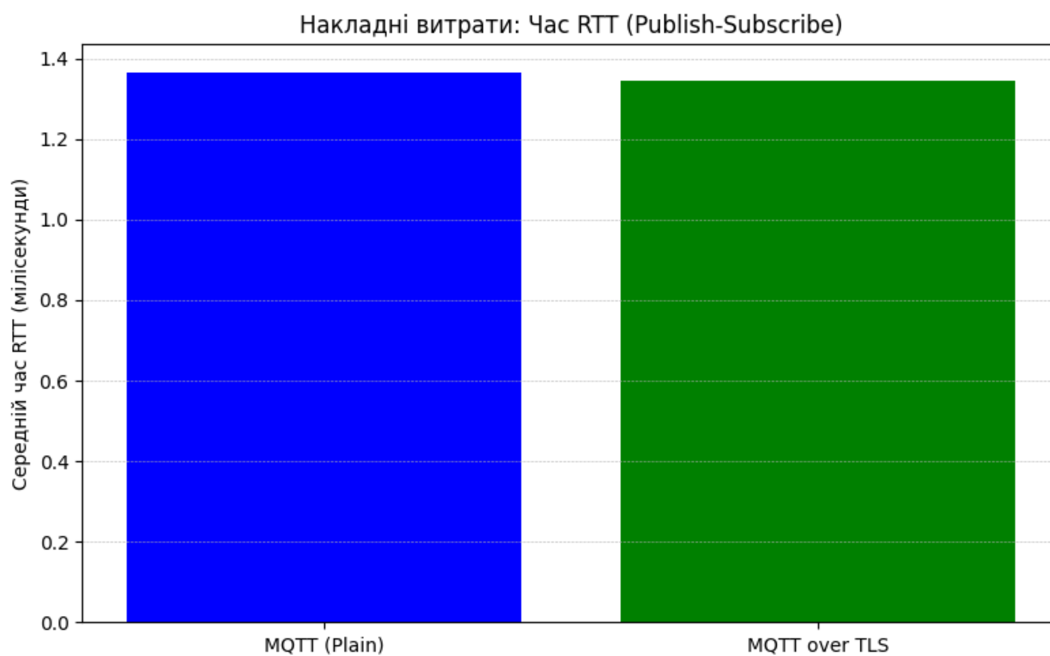

Рисунок А.2 – Вплив TLS на час RTT передачі повідомлень MQTT

ДОДАТОК Б
ПОСИЛАННЯ НА РЕПОЗИТОРІЇ (QR-КОДИ)

Рисунок Б.1 – Посилання на репозиторій GitHub з тестами програмних методів захисту у системах IoT

Рисунок Б.2 – Посилання на репозиторій GitHub з програмним проєктом системи розумного будинку

ДОДАТОК В
ДОПОМІЖНИЙ ІЛЮСТРАТИВНИЙ МАТЕРІАЛ

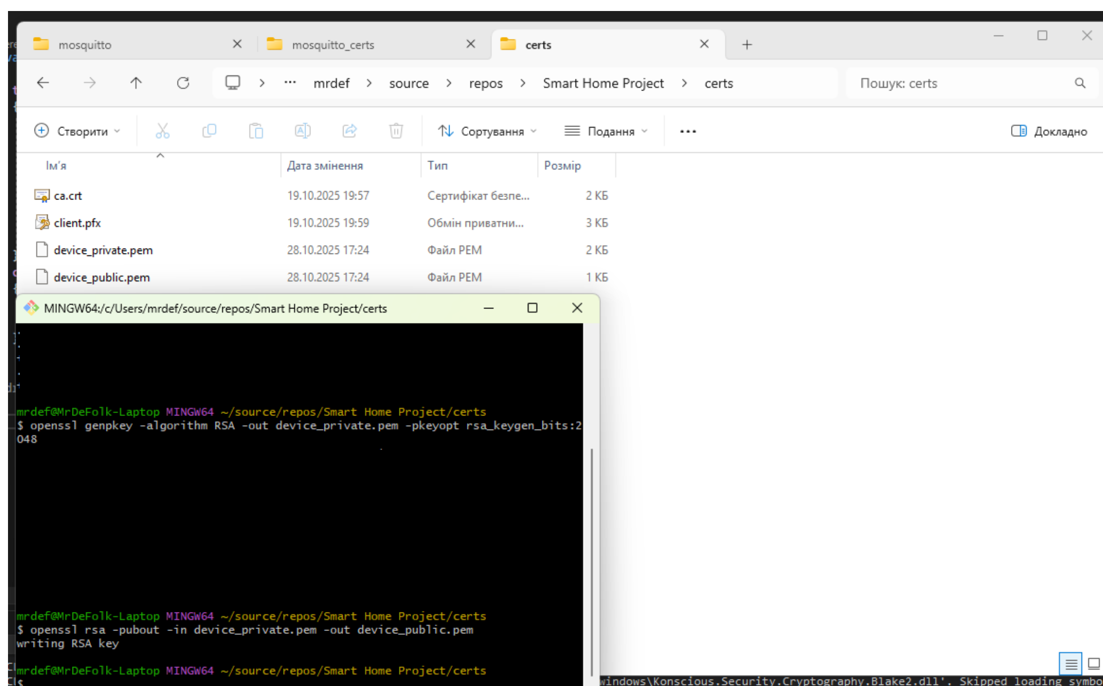

Рисунок В.1 – Створення сертифікатів та ключів

Smart Home Security Control Panel

Файл Перегляд Довідка

Логи безпеки

Timestamp	Тип події	Опис	Користувач
2025-11-02 13:46:03	2FA Success	User successfully entered TOTP.	admin
2025-11-02 13:46:00	Login Success	User successfully entered password.	admin
2025-11-02 12:15:14	2FA Success	User successfully entered TOTP.	admin
2025-11-02 12:15:11	Login Success	User successfully entered password.	admin
2025-11-02 11:46:05	Device Toggled (Hybrid)	Device 'Основне світло' status change...	admin
2025-11-02 11:46:04	Device Toggled (Hybrid)	Device 'Основне світло' status change...	admin
2025-11-02 11:45:57	2FA Success	User successfully entered TOTP.	admin
2025-11-02 11:45:52	Login Success	User successfully entered password.	admin
2025-11-02 10:09:51	2FA Success	User successfully entered TOTP.	admin
2025-11-02 10:09:47	Login Success	User successfully entered password.	admin
2025-11-02 10:07:16	2FA Success	User successfully entered TOTP.	admin
2025-11-02 10:07:12	Login Success	User successfully entered password.	admin
2025-11-02 10:07:07	Login Failed	Invalid username or password.	mrdefolk
2025-10-31 16:45:34	Temperature Set (Hybrid)	Device 'Термостат' temperature set to '...	admin
2025-10-31 16:45:31	Status Update Received	Received hybrid encrypted status updat...	System
2025-10-31 16:45:30	Status Update Received	Received hybrid encrypted status updat...	System
2025-10-31 16:45:28	Device Toggled (Hybrid)	Device 'Нічник' status changed to 'Вві...	admin
2025-10-31 16:45:28	Device Toggled (Hybrid)	Device 'Нічник' status changed to 'Вим...	admin
2025-10-31 16:45:26	Device Toggled (Hybrid)	Device 'Освітлення стільниці' status c...	admin
2025-10-31 16:45:26	Device Toggled (Hybrid)	Device 'Освітлення стільниці' status c...	admin
2025-10-31 16:45:24	Status Update Received	Received hybrid encrypted status updat...	System
2025-10-31 16:45:23	Temperature Set (Hybrid)	Device 'Термостат' temperature set to '...	admin
2025-10-31 16:45:16	2FA Success	User successfully entered TOTP.	admin
2025-10-31 16:45:13	Login Success	User successfully entered password.	admin
2025-10-30 22:38:57	2FA Success	User successfully entered TOTP.	admin
2025-10-30 22:38:50	Login Success	User successfully entered password.	admin
2025-10-30 14:39:49	Temperature Set (Hybrid)	Device 'Термостат' temperature set to '...	admin
2025-10-30 14:39:14	Status Update Received	Received hybrid encrypted status updat...	System

Помилка TLS підключення: Error while connecting with host 'localhost:8883'. Ост. шифр.: - мс Ост. дешифр.: - мс

Рисунок В.2 – Вигляд вікна «Логи безпеки» програми Система розумного будинку