Міністерство освіти і науки України
Київський національний університет імені Тараса Шевченка

Факультет інформаційних технологій
Кафедра кібербезпеки та захисту інформації

ДОПУСТИТИ ДО ЗАХИСТУ:
в.о. завідувача кафедри
кібербезпеки та захисту інформації
_____ Іван ПАРХОМЕНКО
« » травня 2025 р.

ПОЯСНЮВАЛЬНА ЗАПИСКА

кваліфікаційної роботи

галузь знань *12 Інформаційні технології*
(шифр і назва галузі знань)
спеціальність *125 Кібербезпека*
(код і назва спеціальності)
освітній ступень *магістр*
освітня програма *Кібербезпека*
(назва освітньо-професійної програми)
на тему: «Метод захисту інтернету речей»

Виконавець: студент ІІ курсу, групи КБм-21

_____ **Дмитро ЦАРУК**
(підпис) (ім'я, прізвище)

	Ім'я, прізвище	**Підпис**
Науковий керівник	Іван ПАРХОМЕНКО	

Нормоконтроль	Сергій ДАКОВ	

Київ 2025

Міністерство освіти і науки України
Київський національний університет імені Тараса Шевченка

Факультет інформаційних технологій
Кафедра кібербезпеки та захисту інформації

ЗАТВЕРДЖЕНО:
в.о. завідувача кафедри кібербезпеки
та захисту інформації
_____Іван ПАРХОМЕНКО
« » жовтня 2024 р.

ЗАВДАННЯ
на виконання кваліфікаційної роботи

спеціальності *125 Кібербезпека та захист інформації*
(код і назва спеціальності)

освітній ступень *магістр*
(назва освітньо-професійної програми)

Студента КБм-21 (група) Царука Дмитра Олексійовича (прізвище ім'я по батькові)

Тема кваліфікаційної роботи Метод захисту інтернету речей

1. ПІДСТАВИ ДЛЯ ПРОВЕДЕННЯ РОБОТИ

Рішення засідання кафедри кібербезпеки та захисту інформації факультету інформаційних технологій протокол № 4 від 24.10.2024 р.

МЕТА ТА ВИХІДНІ ДАНІ ДЛЯ ПРОВЕДЕННЯ РОБІТ

Об'єкт досліджень	Методи та засоби захисту інформації у мережах Інтернету речей
Предмет досліджень	Розробка та аналіз комбінованих криптографічних методів для забезпечення безпеки IoT-пристроїв, що включають механізми автентифікації, шифрування та перевірки сертифікатів
Мета	Підвищення безпеки систем IoT шляхом розробки комбінованого криптографічного методу захисту інформації.
Вихідні дані для проведення роботи	Методи захисту від витоку даних платіжних карток через інтернет-браузер.

3. ОЧІКУВАНІ НАУКОВІ РЕЗУЛЬТАТИ

Наукова новизна	Поєднання різних криптографічних методів для створення комплексного механізму захисту в системах Інтернету речей.
Практична цінність	Реалізація комплексного методу захисту для IoT, що включають генерування самопідписаних сертифікатів, перевірку їх валідності, шифрування даних та симуляцію захищеного каналу передачі даних.

4. ЕТАПИ ВИКОНАННЯ РОБОТИ

№ п/п	Найменування етапів випускної кваліфікаційної роботи	Термін виконання робіт (початок-кінець)	Відмітка про виконання
1.	Здійснити дослідження нормативної-правової бази, а також державних та міжнародних стандартів, що регламентують вимоги до захисту інформації та до криптографічного захисту в мережі	27.01.2025-31.01.2025	*виконано*
2.	Аналіз криптографічних методів захисту мережевих речей на базі протоколів для безпечного обміну даними в мережі IPSec та SSL/TLS	03.02.2025 – 07.02.2025	*виконано*
3.	Розробка типового модулю для формування самопідписаних сертифікатів на основі PKCS#12.	10.02.2025 - 21.02.2025	*виконано*
4.	Розробка засобу шифрування даних на основі алгоритму AES-256 та механізму цифрового підпису з метою забезпечення безпечного обміну даними в мережі	24.02.2025 – 14.03.2025	*виконано*
5.	Проведення аналізу отриманих результатів	18.03.2025 – 24.04.2025	*виконано*
6.	Робота над висновками	24.04.2025 – 11.05.2025	*виконано*
7.	Оформлення презентації	30.04.2025 – 28.04.2025	*виконано*
8.	Оформлення пояснювальної записки згідно методичних рекомендацій	26.04.2024 – 18.05.2025	*виконано*

№ п/п	Найменування етапів випускної кваліфікаційної роботи	Термін виконання робіт (початок-кінець)	Відмітка про виконання
	Подача пакету документів на розгляд ЕК	19.05.2025	*виконано*

Завдання видав _____ Іван ПАРХОМЕНКО
(підпис) (Ім'я, ПРІЗВИЩЕ)

Завдання прийняв
до виконання _____ Дмитро ЦАРУК
(підпис) (Ім'я, ПРІЗВИЩЕ)

Дата видачі завдання: 25.10.2024 р.
Термін подання кваліфікаційної роботи до ЕК 19.05.2025 р.

РЕФЕРАТ

Пояснювальна записка до кваліфікаційної роботи «Методи захисту інтернету речей»: 112 сторінок, 18 рисунків, 4 таблиці, 2 додатки, 86 джерел.

Об’єкт дослідження – методи та засоби захисту інформації у мережах Інтернету речей.

Мета роботи – підвищення безпеки систем IoT шляхом розробки комбінованого криптографічного методу захисту інформації.

Предмет дослідження: розробка та аналіз комбінованих криптографічних методів для забезпечення безпеки IoT-пристроїв, що включають механізми автентифікації, шифрування та перевірки сертифікатів.

Методи дослідження – теоретичний та структурний аналіз, порівняння, синтез, моделювання криптографічних процесів.

Наукова новизна: полягає у поєднанні різних криптографічних методів для створення комплексного механізму захисту в системах Інтернету речей. Програмний код практично ілюструє такий інтегрований підхід, забезпечуючи автентифікацію пристроїв, безпечне встановлення сеансових ключів та захищену передачу даних.

У роботі проаналізовано законодавчу базу України та міжнародні стандарти у сфері інформаційної безпеки IoT; досліджено існуючі криптографічні методи захисту даних у мережах IoT. Розроблено програмний модуль для формування самопідписаних сертифікатів для автентифікації IoT-пристроїв. Реалізовано алгоритм шифрування даних та метод цифрового підпису.

Актуальність теми: Наявність вразливих місць у системах управління та захисту критичної інфраструктури робить країну уразливою до таких загроз, тому важливість дослідження ефективних методів протидії цим атакам стає ще більш очевидною.

Ключові слова: інтернет речей, криптографічні методи, шифрування, обмін ключами, автентифікація, цифровий підпис.

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ ТА СКОРОЧЕНЬ

AEAD – Authenticated Encryption with Associated Data
AES – Advanced Encryption Standard
API – Application Programming Interface
CA – Certificate Authority
DH – Diffie-Hellman
DoS – Denial of Service
ECC – Elliptic Curve Cryptography
ECDSA – Elliptic Curve Digital Signature Algorithm
GCM – Galois/Counter Mode
HKDF – HMAC-based Key Derivation Function
HMAC – Hash-based Message Authentication Code
HSM – Hardware Security Module
IPsec – Internet Protocol Security
IoT – Internet-of-Things
ISO – International Organization for Standardization
MAC – Media Access Control address
MitM – Man-in-the-Middle
NIST – National Institute of Standards and Technology
PKCS#12 – Public-Key Cryptography Standards #12
SSL – Secure Sockets Layer
TLS – Transport Layer Security
X.509 – Стандарт для PKI та сертифікатів атрибутів
ДСТУ – Державний стандарт України
СУІБ – Система управління інформаційною безпекою

ЗМІСТ

РЕФЕРАТ 2
ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ ТА СКОРОЧЕНЬ 3
ЗМІСТ 4
ВСТУП 7
РОЗДІЛ 1 НОРМАТИВНО-ПРАВОВЕ РЕГУЛЮВАННЯ ЗАХИСТУ ІНФОРМАЦІЇ В ІОТ 10
1.1. Аналіз міжнародних та національних стандартів 10
1.2. Аналіз законодавчої бази України 15
1.3. Аналіз загроз та викликів у сфері захисту IoT 17
Висновок до першого розділу 19
РОЗДІЛ 2 АНАЛІЗ АСИМЕТРИЧНИХ КРИПТОГРАФІЧНИХ МЕТОДІВ І ПРОТОКОЛІВ РОЗПОДІЛУ КЛЮЧІВ 21
2.1 Поняття протоколу розподілу ключів 21
2.1.1 Асиметрична криптографія 22
2.1.2 Криптографічний алгоритм Рівеста-Шаміра-Адлемана 23
2.1.3 Алгоритм Діффі–Хеллмана 24
2.1.4 Криптографія еліптичних кривих 25
2.1.5 Порівняння RSA з ECC 29
2.2. Дослідження протоколів захищеного обміну даними 30
2.2.1. IPSec 31
2.2.2. SSL/TLS 34
2.3. Недоліки традиційних криптографічних підходів у IoT 35
Висновок до другого розділу 38
РОЗДІЛ 3 РОЗРОБКА МЕТОДУ АВТЕНТИФІКАЦІЇ ІОТ-ПРИСТРОЇВ 39
3.1. Генерація та керування самопідписаними сертифікатами X.509 39
3.1.1. Процес генерації самопідписаних сертифікатів X.509 з використанням OpenSSL 43
3.1.2. Інтеграція унікальних характеристик пристрою в сертифікати X.509 44
3.1.3. Формат PKCS#12: Огляд та переваги для зберігання криптографічних даних

IoT 45
3.1.4. Керування життєвим циклом самопідписаних сертифікатів X.509 48
3.1.5. Рекомендації стандартів щодо ідентифікації пристроїв та керування сертифікатами. 50
3.1.6. Підсумок ключових аспектів генерації та керування самопідписаними сертифікатами X.509 для автентифікації IoT-пристроїв. 52
3.2. Реалізація методу генерації сертифікатів 53
3.2.1. Архітектура методу та дизайн API 54
3.2.3 Стратегії валідації вхідних даних 55
3.2.4. Генерація сертифікатів X.509 за допомогою бібліотеки 56
3.2.5. Пакування у формат PKCS#12 за допомогою бібліотеки cryptography 59
3.2.7. Безпечна передача/надання пакетів PKCS#12 на пристрої 61
3.2.8. Обробка помилок та безпечне ведення журналів 63
3.2.9. Інтеграція та аспекти життєвого циклу 65
3.2.10. Поновлення та відкликання сертифікатів 65
3.3. Механізм перевірки сертифікатів у IoT-мережі 66
3.3.1. Автоматична перевірка валідності сертифікатів перед встановленням з'єднання 66
3.3.2. Процес валідації сертифіката X.509 66
3.3.3. Проблеми валідації самопідписаних сертифікатів в IoT: 67
3.3.4. Захист від підробки сертифікатів через апаратні атрибути 69
3.3.5. Серверна перевірка вбудованих апаратних атрибутів 71
3.3.6. Роль ASN.1 та DER у практичній реалізації 73
3.3.7. Практичні аспекти реалізації з використанням коду 74
3.4. NIST та галузеві найкращі практики для перевірки сертифікатів IoT 76
Висновок до третього розділу 79
РОЗДІЛ 4 РЕАЛІЗАЦІЯ МЕТОДУ ШИФРУВАННЯ ДЛЯ IOT 80
4.1. Розробка алгоритму шифрування на основі AES-256 80
4.1.1. Огляд стандартів AES та їх актуальність для IoT 80
4.1.2. Вибір оптимального режиму шифрування (GCM, CFB, CBC) 81
4.2. Реалізація цифрового підпису для захисту даних 90

4.3. Впровадження комбінованого механізму шифрування 91
Висновок до четвертого розділу 100
ВИСНОВКИ 101
СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ 103
ДОДАТОК А 113
ДОДАТОК Б 114

ВСТУП

Сучасний світ стрімко переходить до IoT, що охоплює критичну інфраструктуру, медицину, транспорт та промисловість. Проте зростання підключених пристроїв підвищує кіберзагрози, зокрема атаки на дані та несанкціонований доступ.

Міжнародні (ISO/IEC 27001, 15408, 18033, NIST SP 800-177, 800-52, 800-57) та національні (Закон України «Про захист інформації», ДСТУ 7564:2014) стандарти вимагають комплексного підходу до безпеки IoT. Водночас існуючі методи не враховують обмежені ресурси пристроїв та новітні загрози.

Необхідно розробити нові підходи, що поєднують традиційні механізми (AES-256, PKCS#12, X.509) із сучасними технологіями (IPsec, SSL/TLS, Діффі-Хеллман) та сертифікацією пристроїв для підвищення їх безпеки.

Актуальність теми: Наявність вразливих місць у системах управління та захисту критичної інфраструктури робить країну уразливою до таких загроз, тому важливість дослідження ефективних методів протидії цим атакам стає ще більш очевидною. Наукові розробки у цій галузі можуть значно підвищити здатність країни протистояти зовнішнім і внутрішнім викликам, мінімізувати можливі втрати і допомогти швидко відновлювати об'єкти критичної інфраструктури після атак.

Мета магістерської роботи: підвищення безпеки систем IoT шляхом розробки нових комбінованих криптографічних методів захисту інформації..

Завдання магістерської роботи:

1. Аналіз міжнародних та національних стандартів, що регулюють безпеку в IoT, зокрема ДСТУ та ISO/IEC стандарти, які визначають вимоги до шифрування, автентифікації та управління ключами.
2. Дослідження законодавчої бази України щодо криптографічного захисту інформації та сертифікації IoT-рішень, що працюють із конфіденційними даними.
3. Аналіз протоколів розподілу ключів для забезпечення конфіденційності та цілісності переданих даних в системах IoT, зокрема протоколи, що

використовуються для обміну та автентифікації ключів.

4. Дослідження основ асиметричної криптографії, зокрема алгоритмів RSA, Діффі-Хеллмана та криптографію еліптичних кривих.

5. Аналіз криптографічних протоколів, таких як TLS, DTLS, IPsec, MQTT, SSH, для захищеного обміну даними в IoT, зокрема їх вразливостей, що можуть призводити до атак типу MITM та DoS.

6. Розглянути проблеми енергетичної ефективності традиційних криптографічних методів для IoT, що потребують удосконалення існуючих підходів для забезпечення стабільної роботи пристроїв з обмеженими ресурсами.

7. Розробка комплексного методу автентифікації IoT-пристроїв, що базується на використанні самопідписаних цифрових сертифікатів X.509, з деталізацією процесів їх генерації, управління повним життєвим циклом та безпечного зберігання криптографічних даних з використанням формату PKCS#12.

8. Практично реалізувати програмний модуль мовою Python для автоматизації створення самопідписаних сертифікатів X.509 та пакетів PKCS#12, а також обґрунтовано та описано механізми валідації цих сертифікатів в мережах IoT для забезпечення довіреної взаємодії та захисту від підробок.

9. Реалізувати алгоритм симетричного шифрування AES-256 в режимі GCM, що забезпечує одночасно конфіденційність, цілісність та автентичність даних, як ефективного засобу захисту інформації в IoT-системах.

10. Продемонструвати використання цифрового підпису на основі ECDSA для забезпечення автентичності сертифікатів та концептуально реалізовано поєднання криптографічних методів для симуляції захищених каналів передачі даних, що імітують ключові аспекти безпеки протоколів IPsec та TLS.

Об’єкт дослідження: методи та засоби захисту інформації у мережах Інтернету речей.

Предмет дослідження: розробка та аналіз комбінованих криптографічних методів для забезпечення безпеки IoT-пристроїв, що включають механізми автентифікації, шифрування та перевірки сертифікатів.

У роботі використані методи аналізу стандартів, моделюванні криптографічних

процесів та експериментальному тестуванні в IoT-середовищах.

Наукова новизна полягає у запропонованому поєднанні різних криптографічних методів (зокрема, сертифікатів X.509 з ECDSA, обміну ключами Діффі-Хеллмана та шифрування AES-256-GCM) для створення комплексного механізму захисту в системах Інтернету речей.

Практичне значення полягає в реалізації комплексних механізмів захисту для IoT, що включають генерування самопідписаних сертифікатів X.509, перевірку їх валідності, шифрування даних за допомогою AES-GCM та симуляцію захищеного каналу передачі даних за допомогою протоколу IPSec.

РОЗДІЛ 1
НОРМАТИВНО-ПРАВОВЕ РЕГУЛЮВАННЯ ЗАХИСТУ ІНФОРМАЦІЇ В ІОТ

1.1. Аналіз міжнародних та національних стандартів

Державні стандарти України (ДСТУ) у сфері інформаційної безпеки мають велике значення для регулювання та забезпечення захисту даних у різних сферах діяльності, зокрема у рамках IoT. Інтернет Речей, як технологія, що передбачає взаємодію фізичних пристроїв через Інтернет, вимагає особливої уваги до аспектів безпеки, оскільки зростаюча кількість підключених пристроїв створює нові виклики для захисту інформації. Стандарти ДСТУ є основою для розробки та реалізації заходів створення захищених інформаційних систем, що відповідають вимогам національного законодавства.

Одним із ключових документів є ДСТУ 7564:2014, який визначає вимоги до захисту інформації в інформаційно-комунікаційних системах. Цей стандарт встановлює основи для забезпечення цілісності, конфіденційності та доступності інформації, що має особливу важливість для IoT-систем, де велика частина даних передається через відкриті мережі. Виходячи з вимог цього стандарту, розробляються методи шифрування, автентифікації та захисту інформації від несанкціонованого доступу, що є критичними для збереження безпеки IoT-пристроїв [1].

Державний стандарт України 4145-2002 "Криптографічний захист інформації. Основні терміни та визначення" регламентує використання криптографічних методів для захисту інформації в Україні. Цей стандарт визначає основні терміни та принципи криптографії, що є необхідними для забезпечення конфіденційності, цілісності та автентичності даних, що передаються в інформаційно-комунікаційних системах.

В IoT, стандарт ДСТУ 4145-2002 має особливе значення через велику кількість пристроїв, які обмінюються даними в мережі, що робить криптографічний захист даних одним з основних елементів безпеки. Оскільки IoT-пристрої, такі як сенсори, шлюзи та інші компоненти, часто передають чутливу інформацію через відкриті або

незащищені канали зв'язку, застосування стандартів криптографії є необхідним для запобігання несанкціонованому доступу до даних та їх спотворенню.

Стандарт визначає необхідність використання сертифікованих криптографічних алгоритмів для захисту інформації, що передається через різні комунікаційні канали в IoT-системах. Важливими аспектами є генерація, зберігання, розподіл і знищення криптографічних ключів, що є критичними для захисту даних, оскільки компрометація ключів може призвести до серйозних наслідків, таких як витік конфіденційної інформації або злом системи.

Крім того, стандарт передбачає вимоги до інтерфейсів криптографічних засобів, які повинні бути інтегровані в пристрої IoT. Це дозволяє забезпечити автоматичне шифрування даних на етапі їх передачі або зберігання, а також запобігти можливим уразливостям, що можуть виникнути через недостатній рівень шифрування або несертифіковані криптографічні методи. Оскільки IoT-пристрої зазвичай мають обмежені ресурси, то стандарт також враховує вимоги до мінімізації навантаження на апаратні та програмні компоненти, що використовуються для криптографічного захисту.

ДСТУ 4145-2002 також забезпечує основу для сертифікації криптографічних засобів та пристроїв, що використовуються в IoT. Це дозволяє державним органам та підприємствам впевнено інтегрувати IoT-пристрої в інфраструктуру, знаючи, що їхні методи захисту відповідають національним стандартам безпеки. Таким чином, впровадження цього стандарту в IoT забезпечує високий рівень захисту даних та сприяє розвитку безпечних інформаційно-комунікаційних технологій в Україні[2].

Важливе місце в стандарті займають криптографічні засоби захисту. ДСТУ надає рекомендації щодо використання сучасних криптографічних алгоритмів, таких як AES, для шифрування даних у каналах передачі інформації. Враховуючи обмежені ресурси IoT-пристроїв, застосування ефективних алгоритмів, що дозволяють забезпечити високий рівень захисту при мінімальному навантаженні на обчислювальні потужності пристроїв, є критичним завданням. Стандарт також визначає вимоги до криптографічного захисту інформації, забезпечуючи належний рівень безпеки даних на всіх етапах їхнього оброблення та зберігання.

Також важливою складовою частиною системи захисту в IoT є ДСТУ 27001:2015, який регулює впровадження систем управління інформаційною безпекою (СУІБ). Цей стандарт передбачає встановлення загальних вимог до систем безпеки та визначає механізми управління ризиками, що допомагають організаціям знижувати ймовірність загроз. В IoT, де кожен пристрій може бути потенційною точкою вразливості, важливо мати механізми, що дозволяють контролювати доступ до системи, здійснювати аудит і реагувати на можливі інциденти.

Для забезпечення належного рівня захисту в сфері IoT, ДСТУ також охоплює питання використання диференційованих рівнів безпеки для різних категорій інформаційних систем, що дає можливість адаптувати стандарти безпеки до конкретних вимог IoT-систем. Це дозволяє встановити найбільш ефективні засоби захисту для різних типів даних, враховуючи специфіку їхнього використання в межах мережі IoT.

ISO/IEC 27001 є міжнародним стандартом, що визначає вимоги до системи управління інформаційною безпекою (ISMS). Він орієнтований на допомогу організаціям у виявленні, оцінці та мінімізації ризиків, пов’язаних із загрозами для інформаційної безпеки. Основною метою цього стандарту є створення структури для управління інформаційною безпекою, що забезпечує комплексний підхід до захисту даних.

У сфері IoT цей стандарт допомагає організаціям впроваджувати необхідні заходи безпеки для захисту даних, що передаються між пристроями та серверами. Це включає в себе захист каналів зв’язку за допомогою таких технологій, як IPSec, TLS, VPN, політики автентифікації через сертифікати X.509, а також контроль доступу до пристроїв IoT. Важливим аспектом є також впровадження шифрування даних для запобігання перехопленню та змінам інформації в процесі передачі. Таким чином, ISO/IEC 27001 дозволяє системно підходити до забезпечення конфіденційності, цілісності та доступності даних у IoT [3, 4].

ISO/IEC 15408, також відомий як Common Criteria, є стандартом для оцінки безпеки інформаційних систем, криптографічних механізмів та іншого програмного забезпечення. Цей стандарт визначає методологію для оцінки функціональних вимог

безпеки, а також рівнів гарантії безпеки (EAL), що варіюються від EAL1 до EAL7, залежно від рівня захисту, що надається системою.

ISO/IEC 15408 дозволяє оцінити безпеку пристроїв та протоколів, що використовуються в таких системах. Оскільки IoT-пристрої часто мають обмежені ресурси та можуть бути вразливими до атак, важливо враховувати вимоги цього стандарту для оцінки криптографічних механізмів, які застосовуються в IoT-продуктах. Наприклад, сертифікація криптографічних модулів та використання сертифікованих апаратних засобів безпеки (HSM, TPM) для збереження криптографічних ключів дозволяє забезпечити високий рівень безпеки IoT-систем [5].

ISO/IEC 18033 визначає стандартизовані алгоритми шифрування, що забезпечують конфіденційність даних. Цей стандарт класифікує алгоритми шифрування на блокові шифри, потокові шифри та асиметричні шифри. Він визначає алгоритми, які відповідають сучасним вимогам безпеки, зокрема для захисту даних у різноманітних інформаційних системах, включаючи IoT.

Вибір алгоритму шифрування для IoT-пристроїв залежить від балансу між безпекою та продуктивністю. Наприклад, для пристроїв із високими ресурсами може використовуватися AES-256 GCM, а для менш потужних пристроїв — ChaCha20-Poly1305. Крім того, стандарт передбачає використання квантово-стійкої криптографії, як-от NTRU, що є важливим для майбутнього захисту IoT від потенційних загроз, пов'язаних з розвитком квантових обчислень [6].

Національний інститут стандартів і технологій США (NIST) розробив ряд рекомендацій, спрямованих на забезпечення безпеки криптографічних механізмів у мережах, що є критично важливими для захисту Інтернету речей (IoT). Ці документи допомагають встановити стандарти, які забезпечують конфіденційність, цілісність та автентифікацію даних, що передаються між пристроями IoT.

Основними стандартами, які необхідно враховувати при захисті IoT, є:

- SP 800-177 — безпечні криптографічні протоколи;
- SP 800-52 — використання TLS для захищених комунікацій;
- SP 800-57 — управління криптографічними ключами.

NIST SP 800-177 надає рекомендації щодо безпечного використання криптографічних технологій, таких як SSL/TLS, IPSec, SSH, S/MIME, DNSSEC та PKI. Для IoT цей стандарт містить наступні основні вимоги:

- Використання протоколу TLS 1.2 або вище (рекомендований стандарт — TLS 1.3).
- Забезпечення автентифікації за допомогою цифрового підпису та сертифікатів X.509.
- Вибір надійних алгоритмів шифрування, таких як AES-256.
- Підтримка механізмів аутентифікації на основі криптографічних ключів, зокрема ECDSA та RSA, де мінімальна довжина ключа має становити 2048 біт для RSA, 256 біт для ECC.
- Використання IPsec для захисту міжмережевих комунікацій між пристроями IoT.

Цей стандарт особливо важливий для IoT, оскільки він надає конкретні рекомендації щодо впровадження ефективних методів безпечного обміну даними, що забезпечує захист на всіх етапах комунікацій між пристроями [7].

NIST SP 800-52 надає рекомендації щодо використання протоколу TLS (Transport Layer Security) для забезпечення захищеного зв’язку. Основні рекомендації цього стандарту включають:

- Використання версії TLS 1.2 або вище (рекомендується TLS 1.3).
- Відмова від застарілих версій SSL 2.0 та SSL 3.0.
- Впровадження сучасних алгоритмів шифрування, зокрема AES-GCM.
- Підтримка X.509-сертифікатів для автентифікації серверів та клієнтів.

Для IoT цей стандарт допомагає підвищити захищеність комунікацій між пристроями шляхом використання TLS 1.3 та сертифікатів X.509 для аутентифікації. Заміна застарілих алгоритмів (MD5, SHA-1, RSA-1024) на сучасніші рішення (ECC, AES-GCM, RSA-3072, SHA-256) є важливим кроком для зміцнення безпеки в мережах IoT [8].

NIST SP 800-57 визначає вимоги до генерації, зберігання, розповсюдження та заміни криптографічних ключів. Це є критичним для забезпечення безпеки

криптографічних операцій у системах IoT, де потрібно ефективно управляти ключами на великій кількості пристроїв. Основні рекомендації цього стандарту:

- Встановлення мінімальної довжини ключів та їхнього терміну дії.
- Рекомендації щодо створення, зберігання та управління криптографічними ключами.
- Використання алгоритмів стійкого шифрування, таких як AES, RSA-3072 та ECC-256.
- Регулярна ротація криптографічних ключів для зменшення ризиків компрометації.

В IoT цей стандарт особливо важливий для використання апаратних модулів безпеки (HSM, TPM) для зберігання секретних ключів, а також для автоматизованого управління ключами в великих розподілених системах. Застосування алгоритмів асиметричного шифрування дозволяє безпечно розподіляти ключі між пристроями, що є важливим для захисту даних у мережах IoT [9].

1.2. Аналіз законодавчої бази України

Закон України "Про технічний захист інформації" регламентує діяльність, спрямовану на забезпечення захисту інформації, що є власністю держави або підлягає захисту відповідно до законодавства. Він визначає основні принципи та вимоги для захисту інформаційних систем, зокрема, в аспекті захисту даних, які передаються або обробляються через мережі IoT. Важливим є визначення методів технічного та криптографічного захисту інформації, що застосовуються до сенсорних даних, команд управління чи особистої інформації користувачів IoT-пристроїв. Закон також вимагає від організацій, які використовують інформаційні системи, забезпечення відповідності захисту інформації вимогам ТЗІ. Крім того, IoT-пристрої, що взаємодіють із державними системами, мають відповідати вимогам захисту інформації, а також проходити сертифікацію [10].

"Положення про порядок здійснення криптографічного захисту інформації" визначає норми криптографічного захисту інформації з обмеженим доступом, що

передається через відкриті канали зв'язку. Для IoT це передбачає необхідність шифрування даних, що передаються через публічні канали, такі як Wi-Fi чи мобільні мережі. Криптографічні засоби, які використовуються для захисту інформації, повинні бути сертифіковані відповідно до вимог Держспецзв’язку, що забезпечує їх відповідність національним стандартам. Важливим аспектом є управління криптографічними ключами, включаючи їх генерацію, зберігання, передачу та знищення. Для IoT це критично важливо, оскільки у великих мережах пристроїв необхідно уникати компрометації ключів. Впровадження криптографічного захисту підлягає перевірці державними органами, що контролюють інформаційну безпеку [11].

Держспецзв’язок займається розробкою і впровадженням політики, стандартів і процедур для забезпечення захисту даних, що передаються і обробляються в Інтернеті речей, зокрема, в рамках криптографічного захисту, захисту персональних даних і управління інформаційною безпекою в цілому.

Одним із основних завдань Держспецзв’язку є розробка та оновлення нормативно-правових актів, які регулюють питання технічного захисту інформації в Україні, зокрема через інструменти, які необхідні для безпечного функціонування IoT-пристроїв та інфраструктури. У цьому плані організація має важливу роль у забезпеченні ефективного функціонування СУІБ, у тому числі в системах, що підтримують IoT.

Держспецзв’язок також активно займається впровадженням криптографічних стандартів і сертифікації засобів криптографії, що використовуються в мережах IoT. Це включає як визначення вимог до використання сертифікованих криптографічних засобів, так і контроль за їх виконанням. Зокрема, організація забезпечує відповідність використаних криптографічних алгоритмів, таких як AES-256 або RSA, вимогам стандартів, зокрема до криптографічних протоколів, що використовуються в системах обміну даними, таких як SSL/TLS або IPsec. Це має велике значення для захисту інформації в IoT-системах, де дані передаються через публічні канали зв'язку.

Важливою складовою діяльності Держспецзв’язку є також сертифікація і атестація інформаційних систем, що включає в себе перевірку відповідності

технологій і пристроїв, що входять до складу IoT-мережі, нормативам технічного захисту інформації. Ці процеси забезпечують відповідність IoT-пристроїв вимогам безпеки і контролюють їхні можливості для забезпечення захисту даних.

Держспецзв'язок також здійснює контроль за дотриманням законодавства про захист персональних даних, особливо в ситуаціях, коли IoT-пристрої збирають, обробляють або передають особисті дані. Це вимагає специфічних механізмів для захисту приватності користувачів та забезпечення безпеки їхніх даних, особливо з огляду на можливі загрози несанкціонованого доступу або витоку інформації [12].

1.3. Аналіз загроз та викликів у сфері захисту IoT

Захист IoT є однією з найбільш актуальних та складних проблем сучасної інформаційної безпеки. Із зростанням кількості підключених пристроїв IoT у різноманітних сферах, таких як медицина, транспорт, енергетика та побутові технології, загрози для інформаційної безпеки стають все більш реальними і різноманітними. Однією з головних проблем є різноманітність пристроїв та технологій, що ускладнює забезпечення єдиного стандарту захисту. Розглянемо основні ризики та вразливості, з якими стикаються мережі IoT.

Недостатній рівень аутентифікації та авторизації пристроїв є серйозною проблемою. Багато IoT-пристроїв, особливо дешевших моделей, не мають належних механізмів аутентифікації, що робить їх вразливими до несанкціонованого доступу. У разі компрометації пристроїв зловмисники можуть отримати контроль над системою, зібрати чутливі дані або навіть використовувати пристрій як частину бот-мережі для атак, таких як DDoS.

Іншим великим ризиком є перехоплення та маніпуляція даними. Оскільки багато IoT-пристроїв передають дані через відкриті або ненадійні канали зв'язку, існує великий ризик перехоплення та маніпуляцій даними. Якщо протоколи шифрування не застосовуються належним чином або якщо вони виявляються вразливими, зловмисники можуть перехоплювати, змінювати або навіть фальсифікувати дані, що передаються між пристроями та серверами.

Ще однією серйозною проблемою є використання слабких або застарілих алгоритмів шифрування. Багато IoT-пристроїв мають обмежені ресурси для виконання складних криптографічних операцій, що може призводити до використання застарілих або недостатньо безпечних алгоритмів шифрування. Це створює додаткові ризики для збереження конфіденційності та цілісності даних.

IoT-пристрої також часто мають низький рівень фізичної безпеки, оскільки вони можуть бути розташовані в публічних місцях або легко доступні зловмисникам. Вони можуть бути фізично зламані або модифіковані, що відкриває можливості для атак на апаратному рівні, таких як маніпуляції з мікропрограмами (firmware) або установлення зловмисних програм.

Ще однією загрозою є відсутність регулярних оновлень та підтримки. Багато IoT-пристроїв не отримують регулярні оновлення безпеки або мають обмежену підтримку від виробника. Це особливо стосується застарілих моделей, де нові вразливості не виправляються вчасно, що робить такі пристрої уразливими для атак. Відсутність оновлень означає, що пристрої можуть стати мішенями для відомих експлойтів, які не були виправлені.

Ще одним серйозним викликом є масштабованість загроз. Через велику кількість пристроїв, підключених до мереж IoT, виявлення та блокування атак є значно складнішим завданням порівняно з традиційними інформаційними системами. Це створює додаткові труднощі для управлінців безпекою, які повинні враховувати тисячі потенційних точок входу в мережу.

Без єдиного стандарту безпеки для IoT, проблеми з інтеграцією різних пристроїв і забезпечення єдиного рівня захисту стають ще серйознішими. Відсутність загальних стандартів також може призвести до слабких місць у системах захисту, оскільки різні постачальники можуть використовувати різні підходи до захисту даних та інтерфейсів. Це ускладнює інтеграцію різних IoT-систем і забезпечення міжмережевого захисту.

Без єдиного стандарту або чітких рекомендацій для всіх IoT-пристроїв, різні продукти використовують різні криптографічні алгоритми та протоколи для захисту переданих даних. Це створює складнощі при інтеграції різних пристроїв та

забезпеченні міжмережевого захисту. Наприклад, одні пристрої можуть використовувати застарілі або несертифіковані алгоритми шифрування, що підвищує вразливість мережі.

Відсутність єдиних політик безпеки також може призвести до ситуації, коли різні учасники екосистеми IoT реалізують власні вимоги до безпеки, що не відповідають загальним принципам або міжнародним стандартам. В результаті таких невідповідностей можуть виникнути проблеми з інтеграцією пристроїв у загальну інфраструктуру або з'єднанням з іншими пристроями в мережі, що підвищує ймовірність атак.

Оскільки IoT-системи розвиваються дуже швидко, нові технології та пристрої не завжди встигають отримати відповідні стандарти безпеки. Це створює тимчасову нестабільність у галузі безпеки, оскільки нові вразливості та загрози можуть не бути достатньо вивчені, що ставить під загрозу весь ландшафт IoT-безпеки.

Загрози та виклики в сфері захисту IoT є серйозною проблемою для розробників і адміністраторів безпеки. Відсутність єдиної стандартизації безпеки, низький рівень аутентифікації, вразливості в мережах зв'язку, обмеження в ресурсах пристроїв — усе це створює багатогранну картину загроз. Важливою умовою ефективного захисту є розробка чітких стандартів і протоколів, що дозволять інтегрувати різні пристрої та системи IoT, забезпечуючи надійний захист і контроль доступу [13].

Висновок до першого розділу

Аналіз міжнародних та національних стандартів у сфері інформаційної безпеки засвідчує важливість регулювання безпеки IoT-пристроїв через впровадження сертифікованих методів криптографічного захисту, управління ризиками та забезпечення безпечного обміну даними. Важливу роль у цьому відіграють ДСТУ (зокрема 7564:2014, 4145-2002, 27001:2015) та міжнародні стандарти (ISO/IEC 27001, 15408, 18033), які встановлюють вимоги до шифрування, автентифікації та керування ключами.

Законодавча база України, включаючи закон "Про технічний захист інформації"

та положення про криптографічний захист, визначає необхідність сертифікації IoT-рішень, що працюють із конфіденційними даними. Рекомендації NIST (SP 800-177, SP 800-52, SP 800-57) підкреслюють важливість використання TLS 1.3, IPSec, сертифікатів X.509 та сучасних криптоалгоритмів для забезпечення конфіденційності та стійкості систем IoT.

РОЗДІЛ 2
АНАЛІЗ АСИМЕТРИЧНИХ КРИПТОГРАФІЧНИХ МЕТОДІВ І ПРОТОКОЛІВ РОЗПОДІЛУ КЛЮЧІВ

2.1 Поняття протоколу розподілу ключів

Протоколи розподілу ключів (ПРК) — це криптографічні процедури, які дозволяють користувачам встановлювати безпечні канали зв’язку. Ці протоколи включають генерацію та обмін ключами сеансу та автентифікацію повідомлень. У деяких випадках ПРК залучають довірену третю сторону, наприклад Центр розповсюдження ключів (ЦРК), щоб допомогти створити та розповсюдити ключі.

Метою ПРК в IoT є забезпечення безпечного зв’язку між пристроями з обмеженою обчислювальною потужністю, пам’яттю та енергетичними ресурсами. Використовуючи спрощені протоколи, пристрої IoT можуть створювати безпечні канали для передачі даних і команд, мінімізуючи вплив на продуктивність пристрою та час автономної роботи. Наприклад, Аліса та Боб, два пристрої IoT, потребують безпечного обміну конфіденційними даними. Вони починають із узгодження спільного протоколу, генерації ключів сеансу та обміну ними за допомогою шифрування. Потім вони використовують ці ключі для симетричного шифрування даних. Кожен пристрій перевіряє автентичність отриманих повідомлень, щоб переконатися, що вони надходять від передбачуваного відправника та не були підроблені. Використовуючи протоколи розповсюдження ключів, Аліса та Боб можуть безпечно обмінюватися даними, забезпечуючи конфіденційність та цілісність обмінюваної інформації.

ПРК мають вирішальне значення в системах IoT для забезпечення безпеки та конфіденційності пристроїв і даних. Використовуючи ефективні та легкі протоколи, пристрої IoT можуть створювати безпечні канали зв’язку, які захищають від прослуховування, втручання та інших форм кібератак [14].

2.1.1 Асиметрична криптографія

Асиметрична криптосистема, також відома як криптографія з відкритим ключем, використовує різні ключі для різних операцій у криптосистемі, таких як шифрування та дешифрування. Це дозволяє відкрити відкритий доступ до одного з ключів, відомого як відкритий ключ, без шкоди для секретності відповідного закритого ключа.

У традиційній моделі асиметричного шифрування операція шифрування використовує відкритий ключ для обчислення зашифрованого тексту із повідомлення. Зворотна операція дешифрування використовує відповідний закритий ключ для відновлення вихідного повідомлення. Схему операції шифрування та дешифрування в асиметричній криптографії наведено на Рисунку 2.1.

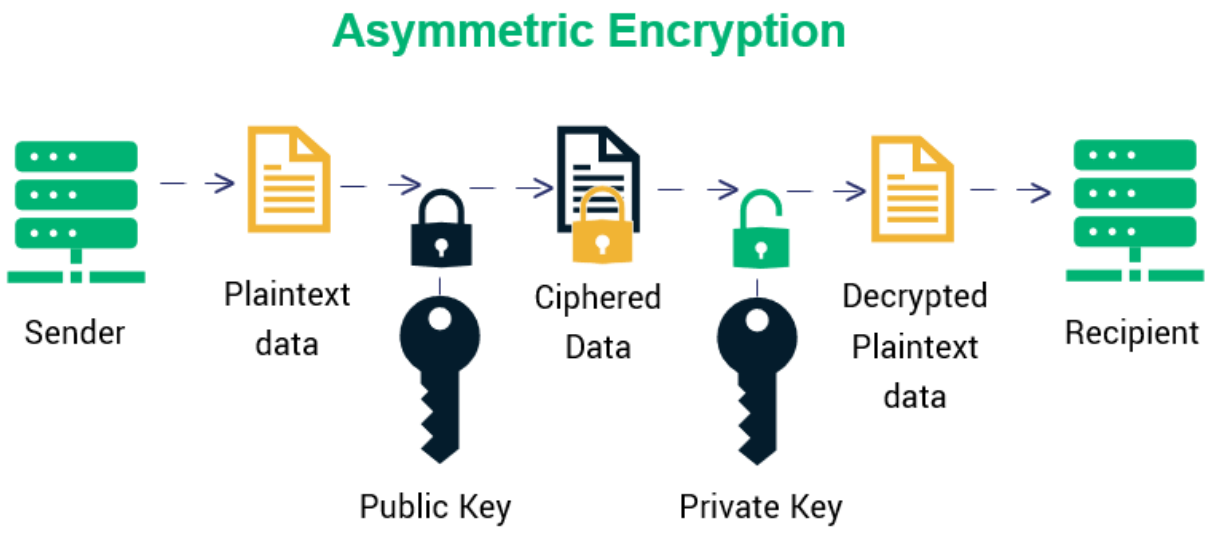

Рисунок 2.1 – Асиметрична криптографія

Сучасне використання асиметричної криптографії виходить за рамки шифрування та дешифрування та включає схеми цифрового підпису та протоколи узгодження ключів. Ці схеми передбачають використання ключа, який можна оприлюднити для певних операцій, тоді як інші вимагають, щоб відповідний секретний ключ залишався конфіденційним.

Асиметричні криптосистеми знаходять широке застосування, особливо в сценаріях, коли сторони спочатку не мають зручного засобу для обміну секретами. Примітні приклади включають шифрування Діффі-Хеллмана, RSA і криптографію еліптичних кривих (ECC) [14, с. 46].

2.1.2 Криптографічний алгоритм Рівеста-Шаміра-Адлемана

RSA – це криптографічний алгоритм, який використовує асиметричне шифрування. Цей алгоритм отримав свою назву на честь трьох винахідників: Рона Рівеста, Аді Шаміра і Леонарда Адлемана, і був запропонований в 1978 році. Розробники успішно втілили ідею односторонніх функцій зі секретом у даному алгоритмі. Ключі для шифрування та розшифрування є функціями двох великих простих чисел, кожне з яких має більш як 100...200 десяткових цифр. Відновлення відкритого тексту з шифрованого тексту та відкритого ключа еквівалентно факторизації числа на два великі прості множники [14, 15]. Схему роботи алгоритму RSA наведено на Рисунку 2.2.

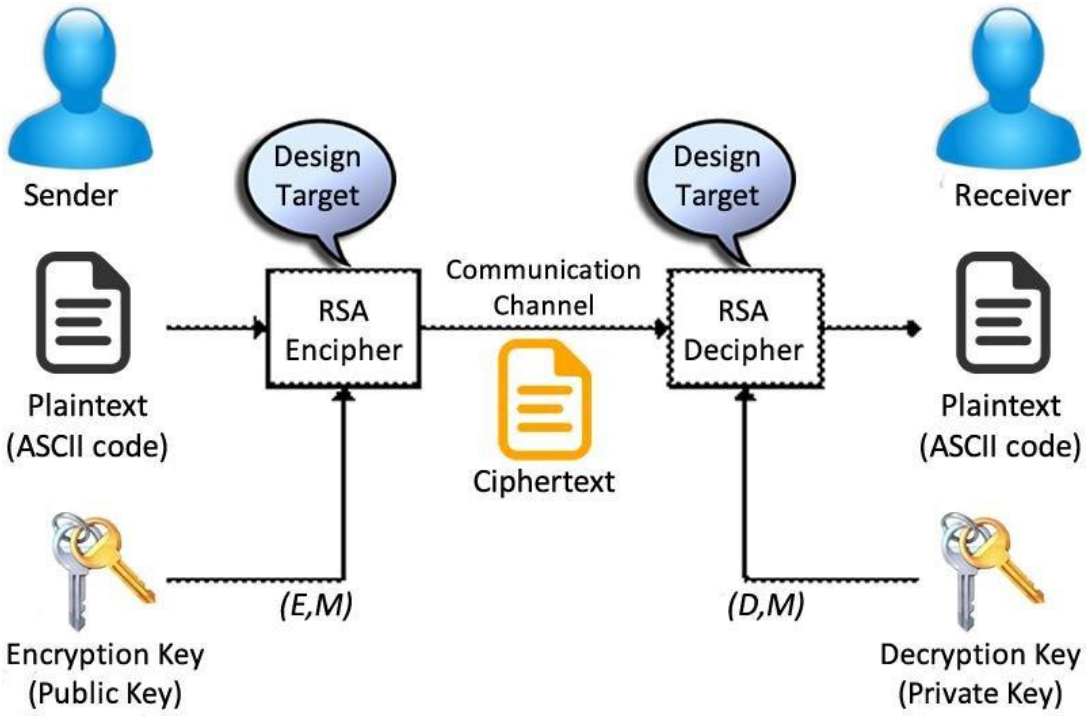

Рисунок 2.2 – Схема роботи алгоритму RSA

2.1.3 Алгоритм Діффі–Хеллмана

Діффі та Хеллман запропонували 1976 року для створення криптографічних систем із відкритим ключем алгоритм, який, так само як і алгоритм Ель-Гамаля, ґрунтується на труднощах обчислення дискретного логарифма. Алгоритм Діффі-Хеллмана використовується для розподілу ключів (генерації секретного ключа), але його не можна використовувати для шифрування повідомлення.

Відповідно до цього алгоритму, учасники інформаційного процесу А і В домовляються про значення великого простого числа p і простого дискретного кореня цього числа a (рис. 2.3).

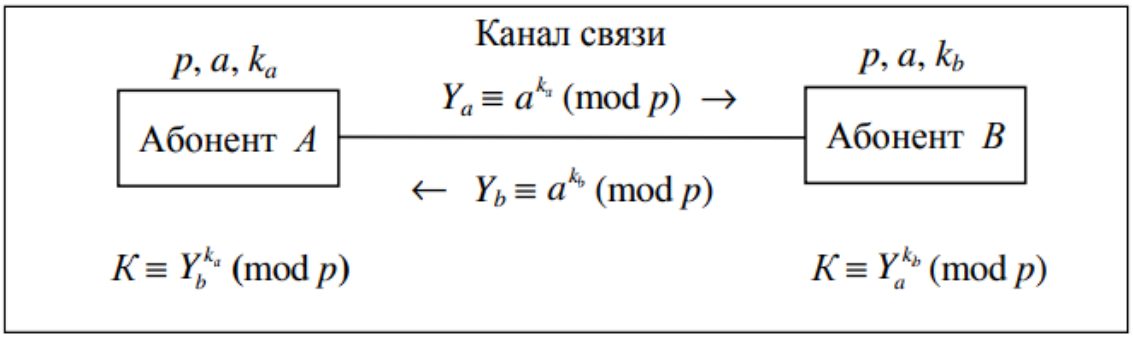

Рисунок 2.3 – Обмін ключами з використанням алгоритму Діффі-Хеллмана

При цьому сторона A вибирає випадкове число k_a, а сторона B - випадкове число k_b таким чином, щоб виконувалися умови

$1 < k_a < p - 1$ й $1 < k_b < p - 1$.

Числа k_a и k_b тримаються сторонами A и B в секреті. Сторона A формирує відкритий ключ по правилу

$Y_a \equiv a^{ka} \pmod{p}$.

Аналогічно сторона В формує відкритий ключ за правилом

$Y_b \equiv a^{k_b} \pmod{p}$.

Після обміну несекретними ключами Y_a і Y_b сторони обчислюють значення секретного числа К:

$K \equiv Y^{k_b} \pmod{p} \equiv a^{k_a k_b} \pmod{p}$;

$K \equiv Y^{k_a} \pmod{p} \equiv a^{k_b k_a} \underset{b}{\overset{a}{(\mathrm{mod}\ p)}}$.

Отримане число К для зловмисника є секретним, оскільки розв'язання рівнянь Y_a і Y_b для більших чисел не є можливим.

Алгоритм Діффі-Хеллмана можна розширити на випадок із трьома і більше учасниками [16, 17].

2.1.4 Криптографія еліптичних кривих

ECC базується на математиці еліптичних кривих і особливо корисний у сценаріях, коли обчислювальні ресурси обмежені.

Генерація ключа в ECC включає вибір конкретної еліптичної кривої та базової точки на цій кривій. Закритий ключ, який представляється цілим числом, обирається випадковим чином в певному діапазоні. Відкритий ключ, що представляє собою точку на еліптичній кривій, отримується шляхом множення базової точки на закритий ключ [62, с. 70].

Загальний вид кубічного рівняння для еліптичних кривих:

$$y^2 + axy + by = x^3 + cx^2 + dx + e, \qquad (2.1)$$

де *a, b, c, d, e* – дійсні числа, що задовольняють певним умовам. Визначення еліптичної кривої також включає елемент в нескінченності, який позначається як точка в нескінченності і є нульовим елементом. Дане рівняння має назву кубічний або рівнянням третього порядку, бо найвищий ступінь в них дорівнює трьом.

У реальних криптосистемах використовується рівняння: $y^2 \equiv x^3 + ax + b \ (mod\ p)$, де $a, b \in GF(p)$. Група $E(GF(p))$ складається з усіх точок (x, y), де $x, y \in GF(p)$ і вони задовольняють рівняння, включаючи точку в нескінченності O. На Рисунку 2.4 наведено приклад вигляду еліптичної кривої.

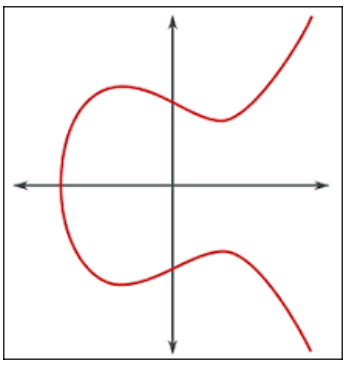

Рисунок 2.4 – Приклад еліптичної кривої

Множина $E_p(a, b)$ складається з усіх точок (x, y), де $x \geq 0$, $y < p$, і вони задовольняють рівняння $y^2 \equiv x^3 + ax + b \ (mod\ p)$, включаючи точку в нескінченності. Кількість точок в $E_p(a, b)$ позначається як $\#E_p(a, b)$ і має значення для криптографічних застосувань еліптичних кривих.

Операція додавання над точками з $E(GF(p))$ алгебраїчно може бути описана наступними правилами:

1. $P + O = O + P = P$.
2. У випадку $P = (x, y)$, *то* $P + (x, -y) = O$. Точка $(x, -y)$ є оберненою до точки P і позначається як $-P$, і $(x, -y)$ лежить на еліптичній кривій і належить $E_p(a, b)$.
3. Якщо $P = (x_1, y_1)$ і $Q = (x_2, y_2)$, то $P + Q = (x_3, y_3)$, де координати x_3 та y_3 визначаються відповідно до правил:

$$x_3 \equiv \lambda^2 - x_1 - x_2 (mod p); \tag{2.2}$$

$$y_3 \equiv \lambda(x_1 - x_3) - y_1 (mod p), \tag{2.3}$$

де λ виконується при умові:

$$\lambda = \begin{cases} \dfrac{y_2 - y_1}{x_2 - x_1}, \text{ якщо } P \neq Q \\ \dfrac{3x_1^2 + a}{2y_1}, \text{ якщо } P = Q. \end{cases} \tag{2.4}$$

Кутовий коефіцієнт січної, проведеної через точки $P = (x_1, y_1)$ та $Q = (x_2, y_2)$, позначається як λ. Коли $P = Q$, січна перетворюється на дотичну, що пояснює наявність двох формул для обчислення λ [62, с. 72].

Рекомендації до вибору параметрів еліптичної кривої, які використовуються у криптографічних задачах, при виборі коефіцієнтів a, b та модуля p мають наступний вигляд: головним критерієм вибору є нездатність здійснити певні атаки, які можуть бути спрямовані на певні класи кривих. Існує альтернативний підхід, який полягає у систематичному конструюванні кривої з певними властивостями, що зазвичай є ефективнішим з обчислювальної точки зору. Для реалізації цього підходу існують спеціальні методи, але отримані криві фактично вибираються з обмеженого набору, що може викликати підозри на наявність деяких специфічних властивостей, які можуть стати основою для розробки алгоритмів злому в майбутньому.

Шифрування ECC базується на обміні ключами Діффі-Хеллмана, який дозволяє двом сторонам установлювати спільний секрет через незахищений канал [16, с. 620-622].

У початковому етапі вибирається просте число p параметрів a і b для еліптичної кривої, що утворюють еліптичну групу точок $E_p(a, b)$. З цієї групи обирається генеруюча точка G. Щоб забезпечити стійкість криптосистеми, найменше число n, для якого $nG = O$, повинно бути простим. Параметри $E_p(a, b)$ та G криптосистеми є загальними для всіх учасників системи. Обмін ключами між учасниками A та B відбувається наступним чином:

1. Учасник A обирає особистий ключ k_a, менше за n, далі генерує відкритий ключ $Y_a = k_aG$, який є точкою з $E_p(a, b)$.

2. Аналогічно, учасник B обирає особистий ключ k_b і обчислює відкритий ключ $Y_b = k_bG$.

3. Учасник A генерує секретний ключ $K = k_aY_b$, а учасник B генерує секретний ключ $K = k_bY_a$.

Обидва ключі дають один і той же результат, як наведено у формулі (2.5).

$$k_aY_b = k_a(k_bG) = k_b(k_aG) = k_bY_a. \quad (2.5)$$

Слід зауважити, що загальний секретний ключ представляється парою чисел. Якщо цей ключ використовується як сеансовий ключ для шифрування, потрібно згенерувати одне значення з цієї пари чисел. Наприклад, можна використовувати координату x або певну функцію від x.

Є різні підходи до шифрування/розшифрування на основі еліптичних кривих. Наприклад, першим завданням є зашифрування відкритого тексту M, яке буде представлено координатами x і y точки P_M. Зашифрований текст C_M буде складатися з пари точок: $C_M = (rGP_M + rY_b)$, де r – випадкове ціле число, вибране учасником А.

Стежком до розшифрування шифротексту C_M є множення першої точки в парі на секретний ключ та віднімання результату від другої точки:

$$P_M + rY_b - k_b(rG) = P_M + r(k_bG) - k_b(rG) = P_M \qquad (2.6)$$

Учасник А маскує повідомлення P_M, додаючи до нього rY_b. За допомогою "підказки", яку має учасник В, можна відновити оригінальне повідомлення, прибравши маску $P_M + rY_b - k_b\ (rG) = P_M$, маючи особистий ключ k_b. Знайдення значення r за відомими G і rG є складною задачею, відомою як проблема логарифмування на еліптичній кривій.

Безпека криптографічного підходу на основі еліптичних кривих залежить від складності вирішення проблеми логарифмування, тобто знаходження значення r за відомими rP і P. Один з найшвидших відомих методів для розв'язання цієї проблеми є розширений метод Полларда. Також були проведені кроки з розробки алгоритму, який покращує продуктивність множення еліптичної кривої за рахунок скорочення операцій і усунення попереднього обчислення. Практичні переваги алгоритму, включаючи підвищену швидкість і ефективність, роблять його добре придатним для середовищ з обмеженими ресурсами, а його математичні пояснення та схеми дають цінну інформацію в області ефективних операцій з еліптичною кривою [18].

2.1.5 Порівняння RSA з ECC

Згідно [19], традиційні криптографічні підходи, зокрема алгоритм RSA, мають певні недоліки при застосуванні в середовищі IoT порівняно з такими алгоритмами, як ECC.

Недоліки традиційних криптографічних підходів (RSA) в IoT:

- Розмір ключів: Для забезпечення одного рівня безпеки RSA потребує значно більших розмірів ключів, ніж ECC. Це є суттєвим недоліком для пристроїв IoT з обмеженими обчислювальними ресурсами та пам'яттю. Наприклад, для рівня безпеки 128 біт RSA вимагає ключ розміром 3072 біт, тоді як ECC — 256 біт.
- Вимоги до пам'яті: Більший розмір ключів RSA призводить до вищих вимог до пам'яті для їх зберігання та обробки порівняно з ECC. Це критично для пристроїв IoT з обмеженим обсягом пам'яті.
- Споживання енергії: RSA споживає більше енергії під час криптографічних операцій порівняно з ECC при однаковому рівні безпеки. Це важливо для пристроїв IoT, які часто працюють від батарей.
- Час генерації ключів: Генерація ключів в RSA є значно повільнішою, ніж в ECC. Час генерації ключів RSA зростає експоненціально зі збільшенням їх розміру, тоді як в ECC цей час зростає лінійно.
- Час виконання (операції із закритим ключем): Операції із закритим ключем в RSA (наприклад, дешифрування або генерація підпису) виконуються значно повільніше, ніж в ECC.
- Час дешифрування: Хоча RSA може бути швидшим у шифруванні в деяких випадках, ECC є більш ефективним у дешифруванні.
- Придатність для систем реального часу: Через більший розмір ключів та повільніші операції, RSA може бути менш придатним для використання в системах IoT, які вимагають обробки даних у реальному часі.

- Вразливість до атак: RSA може бути вразливим до певних криптоаналітичних атак, таких як padding oracle attack або мультиплікативна атака на цифрові підписи, якщо реалізація не включає відповідні контрзаходи.
- Додаткові вимоги для безпеки: Для протидії атакам сторонніми каналами (side-channel attacks), RSA потребує численних контрзаходів, що вимагає додаткової пам'яті та збільшує час виконання.
- Обчислювальна вартість генерації ключів: Процес генерації ключів RSA включає обчислення великих простих чисел, що є обчислювально витратним.

2.2. Дослідження протоколів захищеного обміну даними

Захищений обмін даними є критично важливим аспектом функціонування IoT-мереж, оскільки більшість пристроїв працюють у відкритих середовищах, де можливі атаки на конфіденційність, цілісність і доступність інформації. Основними механізмами забезпечення безпеки є криптографічні протоколи, які гарантують автентифікацію, шифрування та цілісність переданих даних.

Для захисту каналів зв’язку в IoT широко використовується TLS 1.3, який усуває вразливі алгоритми та зменшує час встановлення з’єднання. Його варіація DTLS застосовується для безпечного передавання даних через UDP, що критично для низьколатентних IoT-мереж. Для міжмережевої безпеки використовують IPsec, який забезпечує шифрування та автентифікацію на мережевому рівні, хоча його ресурсоємність обмежує застосування в пристроях із низькою продуктивністю.

У сфері машинного зв’язку широко застосовується MQTT з TLS/DTLS, який дозволяє ефективно передавати зашифровані повідомлення в IoT-системах. Також використовується SSH для безпечного адміністрування IoT-пристроїв і захищеного передавання команд [20].

2.2.1. IPSec

У IP потік даних між додатками IoT може бути видимим для будь-якої кількості вузлів у мережі. Хоча дані можуть бути захищені за допомогою шифрування та змінення даних у мережі, незалежно від того, зашифровані дані чи ні, можна виявити за допомогою перевірки цілісності. часто буває так, що можливість/можливість пропонувати ці послуги недоступна на самому пристрої IoT. Відсутність доступної безпеки робить програми IoT сприйнятливими до різноманітних атак на безпеку . Для вирішення цієї проблеми безпеки в мережевих системах зв'язку був розроблений протокол IPSec.

IPSec — це не окремий протокол, а набір протоколів і служб, які забезпечують основу для впровадження повного рішення безпеки для IP-мережі. IPSec framework забезпечує захист для будь-якої програми або протоколу TCP/IP вищого рівня без необхідності використання додаткових методів безпеки. Захист безпеки забезпечується шляхом додавання функцій автентифікації та шифрування в IP-пакет. Вміст функцій автентифікації та шифрування визначається використанням криптографічних алгоритмів.

Метою IPSec є надання різноманітних послуг безпеки для трафіку, що переміщується між джерелом і одержувачем, одержувачем/джерелом може бути маршрутизатор або хост. Послуги можуть застосовуватися до всіх пакетів або лише до окремих типів трафіку. Рис. 2.5 нижче концептуально показує захист, який забезпечує IPSec між двома хостами. Лінія червоного кольору показує IPSec, реалізований на шляху між маршрутизатором 1 і хостом B. IPSec може працювати лише з певними типами даних, тоді як інші дані передаються незахищеним шляхом, як показано чорними посиланнями. Між двома маршрутизаторами та між хостом B і маршрутизатором 1 можуть існувати окремі захищені з’єднання IPSec.

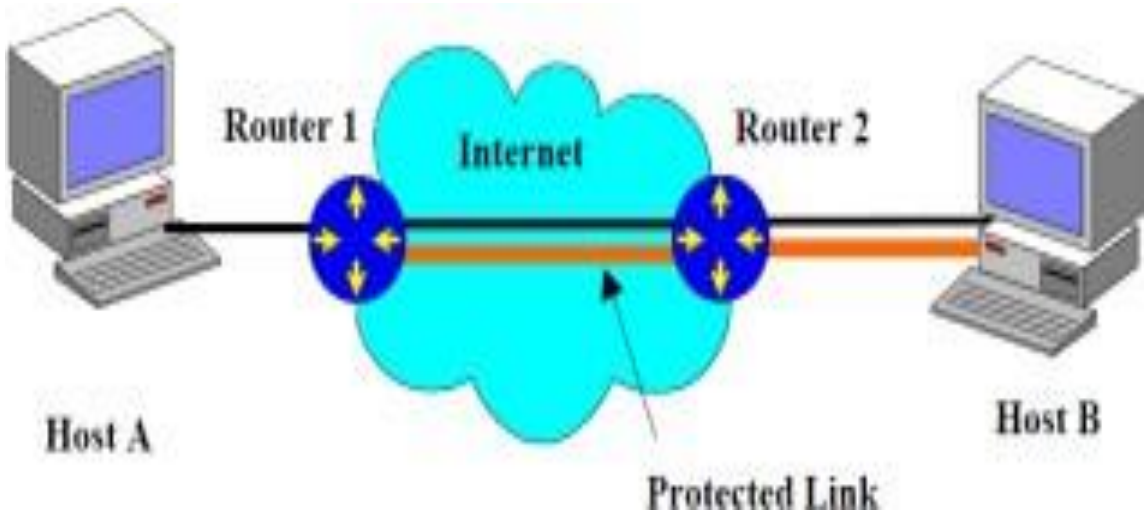

Рисунок 2.5. Захист між двома хостами за допомогою IPSec

У IPSec є два протоколи, які використовуються для надання послуг безпеки;

a) Заголовок автентифікації (AH) і

b) Інкапсуляція корисного навантаження безпеки (ESP) . Протокол AH забезпечує підтримку автентифікації та перевірки цілісності даних. Протокол ESP визначає механізми як для конфіденційності даних, так і для автентифікації (необов'язково). Обидва протоколи IPSec (AH і ESP) підтримують два режими роботи: (а) транспортний режим і (б) тунельний режим. У транспортному режимі автентифікується лише сегмент даних протоколу верхнього рівня IP-дейтаграми, і він зазвичай використовується для наскрізного захисту пакетів IP-дейтаграм між двома хостами. У режимі тунелю вся оригінальна IP-дейтаграма автентифікується в новому зовнішньому IP-заголовку. Режим тунелю можна використовувати між шлюзами безпеки для створення VPN (віртуальної приватної мережі).

Протокол IPSec майже завжди вбудований у стек протоколів TCP/IP за допомогою програмного забезпечення в ОС (операційній системі), наприклад, у Linux і NetBSD. Проте доведено, що IPSec потребує обчислень, що значно впливає на продуктивність мережі, у якій він реалізований. Пропускна здатність даних у основних маршрутизаторах уже досягла терабітів на секунду, а швидкість інтерфейсу лінійної карти перевищує 10 Гбіт/с, але швидкість високопродуктивних пристроїв безпеки в Інтернеті значно поступається цій пропускній здатності. Основна причина

такого зниження швидкості полягає в тому, що вимоги до обробки даних для протоколів безпеки часто є складними та трудомісткими, тому пристроям безпеки важко досягти рівної продуктивності порівняно з пристроями в Інтернеті. З огляду на те, що програмні рішення складних проблем, таких як IPSec, зазвичай страждають від проблем із низькою продуктивністю, порівняно з апаратним забезпеченням, для високої швидкості пропускання даних необхідно використовувати апаратні реалізації IPSec.

FPGA роблять його придатним для використання особливо для додатків, що включають складні криптографічні алгоритми. Можна сказати, що FPGA поєднують найкращі частини ASIC і систем на основі процесора, але насправді є паралельними за своєю природою. Перевагою використання процесора, запрограмованого програмним забезпеченням, є те, що програмне забезпечення є дуже гнучким для змін, а недоліком є те, що продуктивність може погіршитися, якщо годинник не швидкий. Перевага ASIC полягає в тому, що вона може забезпечити дуже високу продуктивність завдяки спеціалізованому типу роботи, а її недоліками є: 1) високе співвідношення вартості до обсягу; 2) збільшена затримка між проектуванням і кінцевим продуктом; 3) нездатність включати нові зміни після виготовлення системи та 4) труднощі з усуненням помилок. ПЛІС заповнюють прогалину між апаратним і програмним забезпеченням і пропонують численні переваги. Враховуючи це, автори вважають, що FPGA є найкращою реконфігурованою апаратною платформою для реалізації криптографічних алгоритмів.

Можлива реалізація ядра IPSec на основі FPGA запропонована на рис. 2.6. Це архітектура BITW (Bump in the Wire) для IPSec. На рис. 2.6 дві мережі, які раніше спілкувалися одна з одною за допомогою незахищеного IP-зв’язку, тепер можуть безпечно спілкуватися, розміщуючи IPSec під звичайним IP за допомогою апаратного рішення BITW IPSec на основі FPGA. Ця техніка дозволяє застарілому обладнанню IPv4 реалізувати IPSec без заміни дорогих мережевих пристроїв [21].

Рис. 2.6. Реалізація ядра IPSec на основі FPGA

2.2.2. SSL/TLS

SSL і його покращена версія TLS є криптографічними протоколами сеансового рівня, які забезпечують захищений обмін даними між клієнтом і сервером. Ключовою вимогою SSL/TLS є отримання клієнтом сертифіката сервера, виданого та підписаного довіреним центром сертифікації (CA). Цей сертифікат використовується для перевірки автентичності сервера та безпечного отримання його криптографічних даних, включаючи відкритий ключ.

Останніми роками було виявлено низку вразливостей у протоколах SSL/TLS. Наприклад, за відсутності ретельної автентифікації можливе проведення атаки 3SHAKE, у якій зловмисний сервер отримує облікові дані клієнта та ініціює атаку «людина посередині» (MiTM). Через зворотну сумісність і необхідність підтримки старих систем атака POODLE дозволяє знизити рівень захисту до небезпечного SSL 3.0. Вразливість Sweet32 дає змогу використовувати слабкі симетричні алгоритми шифрування, як-от 3DES. Особливості реалізації TLS, наприклад, механізм стиснення, відкривають можливість для атаки CRIME, яка дозволяє зловмисникам вгадувати облікові дані жертви методом перебору. Вразливість Heartbleed використовує розширення TLS Heartbeat для зчитування даних із пам’яті жертви.

У версіях Android SDK 4.0 і новіших реалізовано низку бібліотек для встановлення захищених з’єднань, зокрема java.net, javax.net, android.net та org.apache.http, що дозволяють налаштовувати середовище SSL/TLS. Однак багато

додатків містять вразливості через використання застарілих версій SSL v3 і TLS 1.0, які схильні до атак. Хоча у новіших версіях (TLS 1.1–1.3) додано механізм renegotiation_info, що усуває певні вразливості, не всі додатки його використовують, що дозволяє атакувати процес повторного узгодження параметрів з'єднання. Крім того, навіть якщо застосунок підтримує TLS 1.1 або TLS 1.2, він може не перевіряти авторитетність видавця сертифіката, відповідність URL або термін його дії. Це відкриває можливість атак через самопідписані сертифікати або підроблені сертифікати від ненадійних центрів сертифікації, що відомо як SSL MITM Vulnerabilities (SMV).

Одним із реальних сценаріїв атак є компрометація протоколу MQTT через атаку «людина посередині». Наприклад, уразливість була виявлена в застосунку MyMQTT, який працює на Android та використовується для отримання повідомлень від IoT-сенсорів і надсилання команд. При взаємодії застосунку з брокером IoT ініціюється з'єднання через TCP, після чого відбувається обмін повідомленнями за допомогою MQTT. Вразливість TLS дозволяє атакувальнику здійснити атаку TLS Renegotiation, підміняючи клієнтські запити й змушуючи застосунок погодитися на зміну параметрів з'єднання. Це дає змогу зловмиснику виконати атаку MiTM, замінюючи пакет Hello Client, після чого з'єднання завершується у вигідний для атакувальника спосіб.

Цей приклад демонструє, що оцінка безпеки використання SSL/TLS у IoT-системах можлива лише за комплексного аналізу трафіку та логіки виконання застосунків. Аналіз мережевого трафіку дозволяє ідентифікувати стан SSL/TLS-з'єднання, однак не дає інформації про перевірку сертифікатів. Водночас аналіз коду застосунків дає змогу виявити потенційні вразливості, пов'язані з недоліками реалізації перевірки сертифікатів. Для побудови ефективного механізму виявлення загроз необхідне поєднання обох підходів [22].

2.3. Недоліки традиційних криптографічних підходів у IoT

Енергетичні проблеми, які виникають при роботі IoT, охоплюють кілька аспектів, пов'язаних з ефективністю споживання енергії, управлінням ресурсами та

технологічними обмеженнями. Однією з ключових проблем є енерговитратність пристроїв IoT. Багато сенсорів та активних елементів потребують енергії для збору та передачі даних, а їхнє енергоспоживання впливає на термін служби пристроїв. Оскільки більшість IoT-пристроїв працює на акумуляторах, розробка енергоефективних алгоритмів, які дозволяють зменшити витрати енергії під час роботи, є важливим завданням.

Комунікаційні протоколи також впливають на рівень енергоспоживання. Використання таких протоколів, як MQTT, CoAP або HTTP/REST, має різні наслідки для енергетичних витрат. MQTT, наприклад, є легким протоколом, який значно зменшує споживання енергії порівняно з традиційним HTTP. Передача даних у реальному часі може призводити до значних енергозатрат, тому застосування методів компресії даних або режимів сну для пристроїв дозволяє заощадити значну кількість енергії.

Важливу роль відіграють алгоритми управління ресурсами. Оптимізація споживання енергії за допомогою машинного навчання дозволяє підвищити ефективність роботи IoT-систем. Аналіз даних про поведінку користувачів допомагає покращити маршрутизацію та планування завдань, а також формувати енергетичні моделі, що враховують температуру, вологість та присутність людей. Це сприяє не лише зниженню енергетичних витрат, а й забезпеченню комфорту користувачів.

Окремої уваги потребує безпека IoT-систем. Використання шифрування та безпекових протоколів, таких як TLS/SSL, підвищує енергоспоживання через значні обчислювальні навантаження, необхідні для шифрування та дешифрування даних. Залежно від реалізації, це може негативно впливати на автономність IoT-пристроїв.

Значним споживачем енергії є системи зберігання даних. Великі обсяги інформації, які генерують IoT-пристрої, потребують потужних серверів, що підвищує енергетичні витрати. Використання технологій, таких як blockchain, може додатково збільшити ці витрати, проте забезпечити вищий рівень безпеки даних.

Подальші дослідження у сфері IoT повинні зосередитися на розробці нових технологій, які дозволять зменшити енергетичні витрати шляхом вдосконалення методів управління, зберігання даних та впровадження ефективніших енергетичних

протоколів. Забезпечення енергетичної ефективності в IoT є складним і багатогранним завданням, що вимагає комплексного підходу, включаючи оптимізацію апаратного та програмного забезпечення, а також удосконалення стратегій управління. Успішна реалізація цих технологій може суттєво знизити витрати енергії та підвищити ефективність IoT-систем.

Класичні криптографічні алгоритми, як симетричні, так і асиметричні, мають низку уразливостей перед атаками «людина посередині» (MITM) та атаками на відмову в обслуговуванні (DoS), що є критично важливим у контексті безпеки IoT.

Атаки MITM виникають, коли зловмисник перехоплює та змінює трафік між двома сторонами без їхнього відома. Уразливість класичних криптографічних алгоритмів до таких атак зазвичай пов’язана з відсутністю автентифікації ключів. Наприклад, протокол Диффі-Хеллмана без додаткових механізмів автентифікації вразливий до MITM, оскільки зловмисник може провести незалежні обміни ключами з кожною із сторін і видавати себе за легітимного співрозмовника. Подібним чином, уразливість RSA до MITM-атак може бути пов’язана з використанням сертифікатів від ненадійних центрів сертифікації або їхньою відсутністю.

Атаки DoS у криптографічних системах часто націлені на створення надмірного навантаження на обчислювальні ресурси. Наприклад, асиметричні алгоритми шифрування, такі як RSA або ECDSA, вимагають значних обчислювальних потужностей для операцій шифрування та підпису, що може бути використано зловмисниками для виснаження ресурсів серверів або IoT-пристроїв. Атака DoS може включати надсилання великої кількості запитів на встановлення TLS-з’єднання (наприклад, через SSL/TLS Handshake Flooding), змушуючи сервер витрачати ресурси на обробку кожного запиту.

Крім того, деякі симетричні алгоритми також можуть бути вразливими до DoS-атак через їхню чутливість до обсягів трафіку. Наприклад, протокол AES у режимі CBC схильний до атак типу Padding Oracle, що може бути використано для виснаження ресурсів сервера шляхом багаторазового надсилання спеціально сформованих зашифрованих повідомлень.

Захист від MITM і DoS-атак у класичних криптографічних системах вимагає

застосування додаткових механізмів, таких як автентифікація сертифікатів, використання захищених протоколів обміну ключами (наприклад, ECDH з автентифікацією) та впровадження механізмів обмеження трафіку й виявлення аномальної активності [23].

Висновок до другого розділу

Розглянуто протоколи розподілу ключів, що забезпечують надійний захист при обміні даними між пристроями з обмеженими ресурсами. Ці протоколи сприяють генерації, обміну та автентифікації ключів, що гарантує конфіденційність і цілісність переданих даних. Окрім того, розглянуто основи асиметричної криптографії, зокрема алгоритми RSA та Діффі-Хеллмана, а також криптографію еліптичних кривих (ECC), які ефективно забезпечують безпеку в умовах обмежених обчислювальних можливостей.

У дослідженні протоколів захищеного обміну даними в IoT наголошено на важливості застосування криптографічних механізмів, таких як TLS, DTLS, IPsec, MQTT, SSH тощо, для забезпечення безпеки. Водночас, підняті питання, пов'язані з енергетичною ефективністю та обмеженнями традиційних криптографічних підходів. Протоколи на основі SSL/TLS виявляють вразливості, що дозволяють реалізувати атаки типу MITM та DoS. Враховуючи це, необхідно вдосконалювати криптографічні алгоритми та впроваджувати нові енергоефективні рішення для покращення безпеки в IoT.

РОЗДІЛ 3
РОЗРОБКА МЕТОДУ АВТЕНТИФІКАЦІЇ ІОТ-ПРИСТРОЇВ

3.1. Генерація та керування самопідписаними сертифікатами X.509

Автентифікація є фундаментальним аспектом безпеки IoT, що забезпечує перевірку ідентичності пристроїв перед наданням їм доступу до мережевих ресурсів та сервісів. Серед різноманітних методів автентифікації, використання цифрових сертифікатів X.509 займає важливе місце, пропонуючи надійний криптографічний механізм. У цьому підрозділі детально розглядається генерація та керування самопідписаними сертифікатами X.509, їхня роль в автентифікації IoT-пристроїв, інтеграція з унікальними апаратними характеристиками, огляд формату PKCS#12 для безпечного зберігання криптографічних даних, а також аспекти управління життєвим циклом цих сертифікатів у специфічному середовищі IoT.

Розуміння сутності самопідписаних сертифікатів X.509 є першим кроком до їх ефективного застосування в системах автентифікації IoT-пристроїв. Ці сертифікати, хоч і простіші у створенні порівняно з тими, що видаються публічними Центрами Сертифікації (CA), мають свої особливості, переваги та недоліки, які необхідно враховувати при проектуванні безпечних IoT-рішень.

Самопідписаний сертифікат X.509 – це цифровий сертифікат, в якому видавець (issuer) та суб'єкт (subject) сертифіката є однією й тією ж сутністю. Це ключова відмінність від сертифікатів, підписаних стороннім Центром Сертифікації (CA), оскільки тут відсутня зовнішня валідація довіреною третьою стороною. Коли IoT-пристрій, що використовує самопідписаний сертифікат, намагається автентифікуватися, наприклад, в Azure IoT Hub, він пред'являє свій сертифікат. Система (IoT Hub або інший верифікатор), якій попередньо було надано копію цього сертифіката або, що частіше, його унікальний відбиток (thumbprint – хеш сертифіката), перевіряє пред'явлений сертифікат. Якщо хеш отриманого сертифіката збігається зі збереженим відбитком, пристрій вважається автентифікованим [24].

Стандарт X.509, розроблений Міжнародним союзом електрозв'язку (ITU-T), є міжнародним галузевим стандартом, що визначає формат та вміст сертифікатів відкритих ключів для забезпечення оптимальної безпеки [25, 26]. Специфікація формату сертифікатів X.509 v3 та списків відкликаних сертифікатів (CRL) X.509 v2 для використання в Інтернеті детально описана у стандарті RFC 5280, розробленому Робочою групою Інтернету (IETF). Згідно з RFC 5280, основними компонентами (полями) сертифіката X.509 є

- Версія: Вказує версію формату сертифіката (наприклад, v3).
- Серійний номер: Унікальний номер, присвоєний сертифікату його видавцем.
- Ідентифікатор алгоритму підпису: Алгоритм, використаний видавцем для підпису сертифіката.
- Видавець: Розрізнювальне ім'я (Distinguished Name, DN) сутності, що видала сертифікат. У самопідписаних сертифікатах це поле збігається з полем "Суб'єкт".
- Період дії: Часовий проміжок, протягом якого сертифікат є дійсним (дати "Not Before" та "Not After").
- Суб'єкт: Розрізнювальне ім'я (DN) сутності, якій видано сертифікат (наприклад, IoT-пристрій).
- Інформація про відкритий ключ суб'єкта: Містить відкритий ключ суб'єкта та ідентифікатор алгоритму, з яким цей ключ використовується (наприклад, RSA, ECDSA). Формат цього поля для ключів ECC, наприклад, детально описаний в RFC 5480.
- Розширення (опційно для v1/v2, але зазвичай присутні та важливі для v3): Додаткова інформація, така як "Використання ключа" (Key Usage), "Розширене використання ключа" (Extended Key Usage), "Альтернативне ім'я суб'єкта" (Subject Alternative Name, SAN) тощо.

Хоча самопідписані сертифікати можуть спростити початкове розгортання в деяких сценаріях IoT, уникаючи витрат та організаційних складнощів, пов'язаних із взаємодією з публічними CA, ця простота повністю перекладає тягар управління

довірою, життєвим циклом сертифікатів та політиками безпеки на архітектора та розробника IoT-системи. Без зовнішнього CA немає авторитетної третьої сторони, яка б керувала видачею, стандартними механізмами відкликання (як CRL або OCSP, що підтримуються публічними CA) або забезпечувала дотримання політик. Це може стати значним викликом при масштабуванні рішення або при необхідності інтеграції з зовнішніми системами.

Самопідписані сертифікати X.509 можуть використовуватися для автентифікації IoT-пристроїв, наприклад, в Azure IoT Hub, шляхом завантаження відбитка сертифіката пристрою на платформу. Їх застосування має як переваги, так і суттєві недоліки, особливо у порівнянні з сертифікатами, підписаними визнаним Центром Сертифікації [27].

На противагу цьому, сертифікати, підписані CA (публічним або приватним, створеним організацією), пропонують централізовану модель довіри, стандартизовані механізми відкликання, полегшують дотримання нормативних вимог та часто підтримують автоматизоване управління життєвим циклом [24, 28, 29].

Таблиця 3.1

Порівняння самопідписаних та CA-підписаних сертифікатів для автентифікації

Критерій	**Самопідписані сертифікати (безпосередньо)**	**Сертифікати, підписані приватним (самоствореним) CA**	**Сертифікати, підписані публічним CA**
Модель довіри	Децентралізована, вимагає індивідуального встановлення довіри для кожного сертифіката.[6]	Централізована в межах організації; вимагає поширення довіри до кореневого CA на всі сутності, що перевіряють.[7]	Централізована, глобальна; довіра встановлюється через кореневі сховища CA в ОС/браузерах.[7]
Вартість	Безкоштовно.[8]	Низька (інфраструктурні витрати на CA) або безкоштовно (якщо використовуються інструменти з відкритим кодом).[7]	Платні, вартість залежить від типу сертифіката та CA.[8]

продовження таблиці 3.1

Складність початкового налаштування	Низька.[8]	Середня (потребує налаштування та захисту приватного CA).	Низька для кінцевого користувача (CA виконує більшість роботи), але вимагає процесу валідації.[8]
Складність управління життєвим циклом	Висока, особливо при великій кількості пристроїв; ручне управління.[6]	Середня до високої, залежить від інструментів; можлива автоматизація з відповідними інструментами CLM.[9]	Низька до середньої, часто підтримується автоматизація через протоколи (наприклад, ACME) та платформи CLM.[9]
Масштабованість	Низька.[7]	Середня; краще, ніж індивідуальні самопідписані, але обмежена можливостями приватного CA.	Висока; інфраструктура публічних CA розрахована на великі обсяги.
Механізми відкликання	Відсутні стандартні (видалення довіри на стороні верифікатора).[6]	Можливі (CRL/OCSP), якщо приватний CA їх підтримує та інтегрований з верифікаторами.[10]	Стандартні (CRL, OCSP), підтримуються публічними CA.[10]
Придатність для закритих/відкритих систем IoT	Переважно для закритих, тестових або внутрішніх систем.[1]	Добре підходить для закритих або корпоративних систем; може використовуватися для пристроїв, що не взаємодіють з публічним Інтернетом.[1]	Найкраще для відкритих систем та пристроїв, що взаємодіють зі сторонніми сервісами або публічним Інтернетом.[7]
Рівень безпеки (за замовчуванням vs. при належному керуванні)	Низький за замовчуванням; може бути адекватним при дуже ретельному керуванні в обмежених сценаріях.[6]	Середній до високого при належному захисті CA та управлінні життєвим циклом.[7]	Високий, оскільки спирається на перевірені практики та інфраструктуру CA.[7]

3.1.1. Процес генерації самопідписаних сертифікатів X.509 з використанням OpenSSL

OpenSSL є потужним та широко використовуваним інструментарієм командного рядка з відкритим вихідним кодом, який надає реалізацію криптографічних функцій, зокрема протоколів SSL/TLS, а також інструменти для генерації та управління сертифікатами X.509. Його універсальність та доступність роблять його де-факто стандартом для багатьох завдань, пов'язаних з інфраструктурою відкритих ключів (PKI), включаючи створення самопідписаних сертифікатів для IoT-пристроїв.

OpenSSL часто згадується як інструмент для створення файлів PKCS#12 (що містять сертифікати та приватні ключі) та безпосередньо самопідписаних сертифікатів X.509. Поширеність OpenSSL у численних прикладах та документації підкреслює його критичну роль як фундаментальної технології для реалізації PKI, навіть у сценаріях, що не передбачають залучення публічних Центрів Сертифікації. Гнучкість OpenSSL дозволяє здійснювати тонкі налаштування сертифікатів, що є особливо важливим для специфічних потреб IoT, таких як включення унікальних ідентифікаторів пристроїв або визначення нестандартних розширень. Команди OpenSSL можуть додавати розширення до сертифікатів або запитів на підписання сертифікатів (CSR) на основі вмісту спеціальних конфігураційних файлів , що значно розширює можливості стандартних команд. Інтерфейс командного рядка робить OpenSSL зручним для написання сценаріїв та автоматизації процесів підготовки IoT-пристроїв, що є важливим при роботі з великою кількістю пристроїв [30, 31].

Процес генерації самопідписаного сертифіката X.509 за допомогою OpenSSL можна розділити на два основні сценарії: створення безпосередньо самопідписаного сертифіката для одного пристрою та створення сертифіката пристрою, підписаного власним, самостворeним (приватним) Центром Сертифікації. Хоча термін "самопідписаний" часто асоціюється з першим сценарієм, де один сертифікат підписує сам себе, у багатьох практичних розгортаннях IoT він може стосуватися другого підходу. Цей ієрархічний підхід, навіть якщо він обмежується двома рівнями (приватний CA та сертифікати пристроїв), пропонує значно кращу керованість та

гнучкість, особливо якщо приватний ключ CA надійно зберігається. Він дозволяє керувати окремими сертифікатами пристроїв (наприклад, відкликати їх у межах приватної PKI) без впливу на "корінь" цієї приватної довіри [32].

3.1.2. Інтеграція унікальних характеристик пристрою в сертифікати X.509

Прив'язка сертифіката X.509 до стабільного апаратного ідентифікатора додає додатковий рівень захисту. Якщо зловмисник якимось чином отримує доступ до приватного ключа, пов'язаного з сертифікатом пристрою, але система автентифікації також перевіряє апаратний ідентифікатор, вбудований у сертифікат, зіставляючи його з фактичним апаратним ідентифікатором пристрою, що підключається, то самого лише викраденого сертифіката та ключа може бути недостатньо для успішної компрометації. Це реалізує принцип захисту в глибину, ускладнюючи підробку пристроїв. Наприклад, деякі платформи, як Azure IoT Hub, вимагають, щоб поле Common Name (CN) сертифіката пристрою містило його унікальний ідентифікатор (device ID), зареєстрований на платформі, що є обов'язковою умовою для автентифікації.

Існує кілька способів інтеграції апаратних ідентифікаторів у сертифікати X.509:

1. Поле CN в Subject Distinguished Name (DN):

• Традиційно поле CN використовується для ідентифікації суб'єкта сертифіката. Як зазначалося, платформи, такі як Azure IoT Hub, можуть вимагати, щоб CN містив deviceId. Це може бути MAC-адреса (без роздільників), серійний номер або інший унікальний рядок.

• Обмеження: CN може містити лише одне значення та має обмеження щодо довжини та набору символів.

2. Розширення Subject Alternative Name (SAN):

SAN є більш гнучким та сучасним способом представлення кількох ідентифікаторів для одного суб'єкта. Воно підтримує різні типи імен :

• dNSName: для доменних імен.

• iPAddress: для IP-адрес.

- uniformResourceIdentifier (URI): для URI.
- rfc822Name: для адрес електронної пошти.
- directoryName: для іншого розрізнювального імені.
- registeredID: для зареєстрованих ідентифікаторів об'єктів (OID).
- otherName: найгнучкіший тип, що дозволяє включати довільні дані, пов'язані з певним OID. Синтаксис в конфігурації OpenSSL.

Наприклад, проект стандарту IETF для C509 (X.509 сертифікати з кодуванням CBOR, оптимізовані для обмежених пристроїв) описує методи кодування EUI-64 ідентифікаторів (які можуть бути похідними від MAC-адрес) у полі commonName або потенційно як otherName. Зокрема, EUI-64, отриманий з 48-бітної MAC-адреси, може кодуватися як байтовий рядок CBOR з певним префіксом.

Бібліотека cryptography.io для Python підтримує створення об'єктів RegisteredID (що приймає OID) та OtherName (що приймає OID типу та двійкове значення у форматі DER) для включення в SAN. Однак, документація не надає конкретних стандартних OID для MAC-адрес чи серійних номерів, що вказує на необхідність їх пошуку в інших стандартах або використання приватних OID [25, 28].

3.1.3. Формат PKCS#12: Огляд та переваги для зберігання криптографічних даних IoT

Після генерації сертифіката X.509 та відповідної йому приватної ключової пари постає питання їх безпечного зберігання та передачі, особливо в контексті підготовки та розгортання IoT-пристроїв. Формат PKCS#12, також відомий як PFX (Personal Information Exchange Format), надає стандартизоване рішення для цієї задачі.

PKCS#12 – це бінарний формат файлу, визначений у стандарті RFC 7292 (який оновлює попередні версії), що дозволяє зберігати в одному зашифрованому контейнері декілька криптографічних об'єктів. Зазвичай файли PKCS#12 мають розширення .p12 або .pfx. Основними компонентами, що можуть міститися у файлі PKCS#12, є :

- Приватний ключ (Private Key): Це найважливіший компонент, секретний ключ, що відповідає публічному ключу в сертифікаті. Він використовується для таких операцій, як розшифрування даних та створення цифрових підписів.
- Сертифікат кінцевого суб'єкта (Leaf Certificate / End-entity Certificate): Це сертифікат X.509, виданий безпосередньо для пристрою (суб'єкта). Він містить публічний ключ пристрою та інформацію про нього.
- Проміжні сертифікати (Intermediate Certificates): Один або декілька сертифікатів CA, що формують ланцюжок довіри від сертифіката кінцевого суб'єкта до кореневого CA. Їх наявність важлива для того, щоб клієнти могли перевірити валідність сертифіката пристрою.
- Кореневий сертифікат (Root Certificate) (опційно): У деяких випадках файл PKCS#12 може містити й кореневий сертифікат CA, що є вершиною ланцюжка довіри.

Основне призначення формату PKCS#12 – це безпечне зберігання та передача приватних ключів (наприклад, RSA, ECDSA, EdDSA) разом із відповідними сертифікатами та, за необхідності, повним ланцюжком довіри. Це робить його зручним для резервного копіювання криптографічних облікових даних та для їх перенесення між різними системами або для встановлення на кінцеві пристрої.

Історично формат PKCS#12 бере свій початок від стандарту Personal Information Exchange (PFX), опублікованого Microsoft у 1996 році. Згодом він був доопрацьований RSA Laboratories і опублікований як PKCS#12 версія 1.0 у 1999 році. [1] Через свій вік деякі криптографічні механізми та алгоритми, що традиційно використовуються в PKCS#12 (наприклад, для шифрування вмісту файлу на основі пароля), походять з попередньої ери обчислювальної техніки, що іноді може призводити до питань сумісності або необхідності використання оновлених профілів та алгоритмів, особливо в контексті сучасних вимог безпеки, таких як FIPS [33].

"Пакувальний" характер формату PKCS#12, що дозволяє об'єднати приватний ключ та повний ланцюжок сертифікатів в одному файлі, суттєво спрощує логістику надання унікальних криптографічних облікових даних великій кількості IoT-пристроїв. Замість того, щоб керувати кількома окремими файлами (приватний ключ,

сертифікат пристрою, проміжні сертифікати) для кожного пристрою, можна використовувати один захищений паролем файл PKCS#12. Це особливо актуально на етапі виробництва або початкового налаштування пристроїв [30, 34].

Однак, хоча портативність є значною перевагою, вона також несе в собі певні ризики. Якщо сам файл PKCS#12 скомпрометований – наприклад, через використання слабкого пароля, викрадення файлу з незахищеного сховища або перехоплення під час передачі – весь його вміст, включаючи критично важливий приватний ключ, стає доступним зловмиснику. Тому безпека самого файлу PKCS#12 (використання надійного, унікального пароля, шифрування під час передачі та безпечне зберігання у стані спокою перед розгортанням на пристрій) має першочергове значення. Переваги формату PKCS#12 для зберігання криптографічних даних IoT, їх опис та ризики наведені в таблиці 3.2.

Таблиця 3.2

Переваги формату PKCS#12 для зберігання криптографічних даних IoT

Перевага	**Опис переваги в контексті IoT**	**Потенційні ризики/Застереження**
Портативність	Легке перенесення єдиного файлу з усіма необхідними криптографічними даними на різноманітні IoT-пристрої, незалежно від їх платформи.[13]	Файл, що легко переноситься, також легше викрасти, якщо він не захищений належним чином.
Комплексне шифрування вмісту	Приватний ключ, сертифікат та інші дані всередині файлу PKCS#12 захищені шифруванням на основі пароля.[13]	Безпека залежить від надійності пароля; слабкий пароль робить шифрування неефективним. Необхідність безпечного управління паролями.
Широка сумісність	Підтримка більшістю ОС та програмних платформ, що використовуються в IoT, спрощує інтеграцію та розгортання.[13]	Можливі нюанси сумісності між різними реалізаціями PKCS#12 через вік стандарту та різноманітність підтримуваних алгоритмів.[28]

продовження таблиці 3.2

Можливості резервного копіювання та відновлення	Зручний єдиний файл для архівування та відновлення криптографічних облікових даних пристрою.[13]	Компрометація резервної копії означає компрометацію всіх даних. Необхідне безпечне зберігання резервних копій.
Стандартизація	Широко визнаний стандарт, що сприяє інтероперабельності.13	Старіші алгоритми, що підтримуються стандартом, можуть бути менш безпечними за сучасними мірками.28
Спрощене розгортання на пристроях	Надання одного файлу замість кількох (ключ, сертифікат, ланцюжок) спрощує процес початкового налаштування пристрою.	Необхідність безпечної передачі файлу PKCS#12 на пристрій та безпечного зберігання пароля на пристрої (якщо він там потрібен для розпакування).

3.1.4. Керування життєвим циклом самопідписаних сертифікатів X.509

Ефективне та безпечне використання самопідписаних сертифікатів X.509 в IoT-системах не обмежується лише їх генерацією. Не менш важливим є належне управління їхнім повним життєвим циклом, що охоплює етапи від початкового розгортання до оновлення та, за необхідності, відкликання. В умовах відсутності централізованого Центру Сертифікації, який би взяв на себе ці функції, відповідальність за управління життєвим циклом повністю лягає на розробників та операторів IoT-рішення [29].

Стандартний життєвий цикл сертифіката (Certificate Lifecycle Management, CLM) включає такі основні етапи: виявлення, створення/придбання, встановлення, зберігання, моніторинг, оновлення, відкликання та заміна. Розглянемо їх специфіку для самопідписаних сертифікатів в IoT [35, 36].

Основний виклик – уникнення компрометації приватних ключів або сертифікатів під час виробництва та доставки. Передача приватних ключів партнерам по ланцюжку поставок є вкрай небажаною практикою.

- Моніторинг стану: Після розгортання необхідно постійно відстежувати стан сертифікатів.

- Завдання: Виявлення сертифікатів, термін дії яких закінчується, скомпрометованих сертифікатів, або тих, що використовуються неналежним чином [37].

- Системи CLM можуть централізувати інформацію про сертифікати, відстежувати терміни їх дії, статус та потенційні вразливості. Для хмарних платформ, таких як AWS, існують спеціалізовані сервіси (наприклад, AWS IoT Device Defender), які можуть автоматично перевіряти терміни дії сертифікатів пристроїв (наприклад, DEVICE_CERTIFICATE_EXPIRING_CHECK попереджає, якщо сертифікат закінчується протягом 30 днів або вже закінчився) [38, 39].

- Усі сертифікати мають обмежений термін дії, після закінчення якого вони стають недійсними. Оновлення сертифіката технічно означає видачу нового сертифіката для заміни старого.

- Тенденція до скорочення максимальних термінів дії сертифікатів (наприклад, до одного року або навіть менше для підвищення безпеки) та потенційно величезна кількість IoT-пристроїв роблять ручне оновлення сертифікатів нереалістичним. Автоматизація цього процесу є не просто зручністю, а абсолютною необхідністю для уникнення масових збоїв у роботі сервісів через прострочені сертифікати.

Зазвичай включає генерацію нової ключової пари (рекомендовано для підвищення безпеки) та нового CSR на пристрої, передачу CSR на підпис (до самостворено CA або використання механізму самопідписання), отримання нового сертифіката та його безпечне встановлення на пристрій, замінюючи старий.

Платформи CLM можуть автоматизувати процес оновлення. Деякі IoT-платформи, як Azure IoT Edge, надають конфігураційні можливості для автоматичного оновлення сертифікатів (наприклад, через EST), дозволяючи налаштувати поріг для початку оновлення (наприклад, коли залишилося 20% терміну дії) та параметри повторних спроб [40].

Існує поширене хибне уявлення, що самопідписані сертифікати взагалі неможливо відкликати. Це вірно в тому сенсі, що для *безпосередньо* самопідписаного

сертифіката (де пристрій сам собі видавець) немає зовнішнього CA, який би опублікував його у CRL або відповів через OCSP. Однак, *довіру* до такого сертифіката може відкликати сторона, що покладається (Relying Party, RP) – тобто сервер або система, яка автентифікує пристрій. Якщо автентифікація відбувається на основі відбитка сертифіката, відкликання означає видалення цього відбитка зі списку довірених на сервері. Якщо сервер зберігає повні копії довірених самопідписаних сертифікатів, він просто видаляє відповідний сертифікат.

Якщо використовується ієрархія з приватним CA, який підписує сертифікати пристроїв, то цей CA може підтримувати механізми CRL або навіть OCSP (якщо реалізовано відповідний OCSP-відповідач). У цьому випадку процес відкликання більше схожий на традиційний: скомпрометований сертифікат пристрою додається до CRL, який періодично перевіряється сутностями, що покладаються на ці сертифікати. Платформи, такі як Azure IoT Hub та AWS IoT, підтримують відкликання сертифікатів пристроїв, виданих CA (включаючи приватні CA).

Сучасні системи управління життєвим циклом сертифікатів можуть забезпечити негайне відкликання скомпрометованих або неавторизованих сертифікатів, інтегруючись з механізмами контролю доступу [37].

3.1.5. Рекомендації стандартів щодо ідентифікації пристроїв та керування сертифікатами.

Національний інститут стандартів і технологій США (NIST) надає цінні рекомендації, які, хоч і не завжди безпосередньо стосуються самопідписаних сертифікатів, формулюють важливі принципи безпеки.

- NIST SP 800-213A "IoT Device Cybersecurity Capability Core Baseline": Цей документ визначає базовий набір кібербезпекових можливостей для IoT-пристроїв. Щодо ідентифікації та автентифікації, він наголошує на таких аспектах, як унікальна ідентифікація пристрою, можливість автентифікації пристрою перед іншими сутностями та навпаки, а також криптографічні можливості, що включають здатність пристрою "отримувати та валідувати сертифікати" [41].

- NIST SP 1800-16 "Securing Web Transactions: TLS Server Certificate Management": Хоча цей документ зосереджений на сертифікатах TLS-серверів, його фундаментальні принципи управління життєвим циклом сертифікатів є надзвичайно актуальними і для IoT-пристроїв, включаючи ті, що використовують самопідписані сертифікати. Ключові рекомендації включають:

• Ведення повного інвентарю всіх сертифікатів.

• Призначення чіткої відповідальності за кожен сертифікат.

• Використання лише затверджених CA (у випадку самопідписаних – це може бути внутрішньо затверджений процес або приватний CA).

• Обмеження термінів дії сертифікатів (рекомендовано один рік або менше).

• Використання надійних криптографічних ключів та алгоритмів.

• Максимальна автоматизація процесів запиту, встановлення, моніторингу та оновлення сертифікатів.

• Проактивне оновлення сертифікатів (наприклад, за 30 днів до закінчення терміну дії).

• Наявність процедур для швидкого відкликання скомпрометованих сертифікатів.

• Безперервний моніторинг стану сертифікатів та відповідності політикам.

Багато з цих принципів, таких як інвентаризація, автоматизація, проактивне оновлення, безпечне управління ключами та визначені політики, є універсальними вимогами безпеки, які легко адаптуються до специфіки IoT та самопідписаних сертифікатів. "Дух" рекомендацій NIST залишається чинним, навіть якщо "буква" потребує коригування з огляду на обмеження та особливості IoT-середовища.

Генерація та управління самопідписаними сертифікатами X.509 для автентифікації IoT-пристроїв є комплексним завданням, що вимагає ретельного розгляду переваг, недоліків та найкращих практик. Ці сертифікати пропонують гнучкість та економію на початкових етапах, але перекладають значний тягар відповідальності за безпеку та управління на розробників IoT-рішень [42].

3.1.6. Підсумок ключових аспектів генерації та керування самопідписаними сертифікатами X.509 для автентифікації IoT-пристроїв.

Самопідписані сертифікати X.509, де видавець та суб'єкт є однією сутністю, надають базовий механізм автентифікації, особливо в контрольованих середовищах, шляхом перевірки відбитка сертифіката. Їхня генерація за допомогою інструментів, таких як OpenSSL, є відносно простою, але вимагає розуміння процесу створення ключових пар, запитів на підпис та власне підписання. Важливим аспектом є можливість інтеграції унікальних апаратних ідентифікаторів пристрою (MAC-адреса, серійний номер) у поля сертифіката, такі як Common Name або Subject Alternative Name (через otherName з відповідним OID), що посилює прив'язку криптографічної ідентичності до фізичного пристрою. Однак, відсутність універсальних стандартних OID для таких ідентифікаторів та потенційні проблеми зі стабільністю та унікальністю самих апаратних ID є викликами.

Формат PKCS#12 виступає зручним контейнером для зберігання приватного ключа та ланцюжка сертифікатів в одному зашифрованому файлі, що спрощує їх розгортання та резервне копіювання, але вимагає надійного захисту самого файлу.

Керування життєвим циклом самопідписаних сертифікатів – включаючи безпечне початкове розгортання, моніторинг, оновлення та відкликання (шляхом видалення довіри на стороні верифікатора або через механізми приватного CA) – є критично важливим. Відсутність централізованого CA означає, що всі ці процеси мають бути реалізовані та підтримувані в рамках самого IoT-рішення. Тенденція до скорочення термінів дії сертифікатів та масштаби IoT-розгортань роблять автоматизацію цих процесів абсолютно необхідною [27, 28].

Ефективне використання самопідписаних сертифікатів в IoT – це не просто технічне завдання генерації сертифіката. Це вимагає цілісної стратегії безпеки, що охоплює безпеку на апаратному рівні, захищені процеси надання облікових даних, надійне автоматизоване управління повним життєвим циклом сертифікатів та ретельне врахування конкретної моделі довіри та операційного середовища пристроїв. Проблеми традиційної PKI стимулюють розвиток та інтерес до нових підходів, таких

як локальні ACME-сервери для обмежених мереж або більш інтегровані платформи управління життєвим циклом сертифікатів (CLM), які краще адаптовані до масштабу, обмежень та специфічних потреб Інтернету речей [43, 44].

3.2. Реалізація методу генерації сертифікатів

Сертифікати X.509 надають глобально визнаний, стандартизований та криптографічно перевірений метод для встановлення цих ідентичностей. Сертифікат X.509 прив'язує відкритий ключ до сутності (в даному випадку, пристрою IoT), і ця прив'язка підтверджується цифровим підписом довіреного Центру сертифікації (CA) або, в певних контекстах, самою сутністю (самопідписаний сертифікат). Це дозволяє іншим сутностям автентифікувати пристрій, перевіряючи його сертифікат [45].

Після того, як пристрій отримує сертифікат X.509 та відповідний йому приватний ключ, ці криптографічні матеріали необхідно безпечно зберігати на пристрої та передавати на нього. Формат PKCS#12 (Public-Key Cryptography Standards #12) ефективно служить цій меті. Він визначає формат архівного файлу для зберігання багатьох криптографічних об'єктів як єдиного файлу, зазвичай об'єднуючи приватний ключ з його сертифікатом X.509 та пов'язаним ланцюжком довіри. Файли PKCS#12 зазвичай захищені паролем, що забезпечує конфіденційність приватного ключа під час передачі та зберігання [30, 33].

Модуль Python, описаний далі, має на меті автоматизувати та убезпечити процес генерації сертифікатів X.509, адаптованих для пристроїв IoT, включаючи вбудовування унікальних ідентифікаторів пристроїв. Крім того, модуль упаковуватиме ці сертифікати та відповідні їм приватні ключі у формат PKCS#12, полегшуючи безпечне надання та розгортання.

Ключові можливості цього модуля включають:

- Генерація асиметричних пар ключів (RSA або ECC).
- Створення сертифікатів X.509 з налаштовуваними розрізнювальними іменами суб'єкта та видавця, періодами дії та серійними номерами.

- Інтеграція унікальних характеристик пристрою (наприклад, MAC-адреси, апаратного серійного номера) у стандартні розширення X.509, насамперед у поле Альтернативна назва суб'єкта (SAN) за допомогою поля otherName.
- Створення захищених паролем пакетів PKCS#12, що містять приватний ключ пристрою, його сертифікат та, за бажанням, ланцюжок сертифікатів CA, що його видав.
- Дотримання принципів безпечного управління ключами протягом усього процесу генерації та пакування.

Бібліотека cryptography є обраним пакетом Python для реалізації цього модуля. Це рішення ґрунтується на надійному дизайні бібліотеки, активній підтримці та зосередженості на наданні як високорівневих "рецептів" для поширених криптографічних завдань, так і низькорівневих інтерфейсів для більш детального контролю за потреби. Вона широко визнана як сучасний стандарт для криптографічних операцій у Python, випереджаючи старіші бібліотеки, такі як PyCrypto та значною мірою застарілі функціональності X.509/криптографії pyOpenSSL [46, 47].

3.2.1. Архітектура методу та дизайн API

Для логічного інкапсулювання функціональних можливостей модуля та сприяння організації й повторному використанню коду пропонується класова архітектура. Центральний клас, наприклад, з назвою DeviceCertificateManager, міг би слугувати основним інтерфейсом.

Ключові методи в DeviceCertificateManager можуть включати:

- generate_asymmetric_key(key_type='rsa', key_size=2048, curve=None): Генерує приватний ключ RSA або ECC.
- create_device_certificate(subject_info, issuer_info, device_key, issuer_key, validity_days, device_identifiers=None, extensions_config=None): Створює сертифікат X.509.

- create_pkcs12_bundle(device_key, device_cert, ca_certs=None, friendly_name=b"DeviceIdentity", password=None): Створює файл PKCS#12.

Внутрішні допоміжні методи оброблятимуть конкретні завдання, такі як розпізнавання OID (Ідентифікатор об'єкта), кодування ASN.1 для користувацьких значень otherName та побудова різних розширень X.509.

Дизайн програмного інтерфейсу (API) модуля буде відповідати принципам, що надають пріоритет чіткості, зручності використання, безпеці та розширюваності [48].

3.2.3 Стратегії валідації вхідних даних

Ретельна валідація вхідних даних є наріжним каменем безпечного дизайну API та критично важливою для цього модуля. Усі дані, що надаються API модуля, особливо дані, які будуть включені до криптографічних структур, таких як сертифікати X.509, повинні бути ретельно перевірені. Це запобігає цілому ряду проблем, від неправильно сформованих сертифікатів до потенційних вразливостей безпеки.

1. Перевірка типів: Вхідні дані перевірятимуться на відповідність очікуваним типам Python (наприклад, str для загальних імен, int для днів дії, ipaddress.IPv4Address або ipaddress.IPv6Address для IP-адрес).

2. Валідація формату:

- Імена хостів (DNS Names): Повинні відповідати RFC 1034, RFC 1035 та RFC 1123. Це включає перевірку наборів символів, довжини та структури міток.
- IP-адреси: Модуль ipaddress використовуватиметься для валідації та розбору рядків IP-адрес.
- MAC-адреси: Буде застосовано певний формат (наприклад, XX:XX:XX:XX:XX:XX або XX-XX-XX-XX-XX-XX), і вхідні дані будуть перевірені на відповідність йому.
- Будуть визначені та застосовані допустимі набори символів (наприклад, буквено-цифрові, певні спеціальні символи) та обмеження довжини.
- Вхідні рядки OID повинні відповідати десятковій нотації з крапками.

Хоча для структурованих даних, таких як ідентифікатори, перевага надається суворій валідації формату, якщо приймаються будь-які текстові поля вільної форми (наприклад, для описових цілей у сертифікаті, хоча це менш поширене для сертифікатів пристроїв), слід розглянути відповідні методи санітизації для запобігання введенню керуючих символів або інших проблемних послідовностей.

Обробка винятків: У разі невдалої валідації API викликатиме конкретні та інформативні винятки (наприклад, ValueError, TypeError), чітко вказуючи на характер помилки валідації [49].

Важливість ретельної перевірки вхідних даних виходить за рамки запобігання простим помилкам програми. Сертифікати X.509 та Запити на підписання сертифікатів (CSR) структуруються за допомогою ASN.1 (Abstract Syntax Notation One), складного стандарту кодування. Історично склалося так, що вразливості виявлялися в бібліотеках розбору ASN.1. Якщо зловмисник може ввести неправильно сформовані або зловмисно створені дані в поля сертифіката через неперевірені вхідні дані API, він потенційно може спровокувати такі вразливості в базових компонентах OpenSSL бібліотеки cryptography або, ширше, в будь-якій системі, яка згодом використовує ці сертифікаati. Наприклад, надмірно довгий рядок або рядок, що містить неочікувані керуючі символи, якщо потрапить у Загальне ім'я або значення otherName, може призвести до переповнення буфера, відмови в обслуговуванні або неправильного розбору сторонами, що покладаються. Тому сувора перевірка вхідних даних на межі API модуля служить критично важливим заходом глибокого захисту, значно зменшуючи поверхню атаки. Це особливо актуально для ідентифікаторів, які можуть надходити з менш довірених джерел або виробничих процесів перед тим, як потрапити до цього модуля генерації сертифікатів [50, 51].

3.2.4. Генерація сертифікатів X.509 за допомогою бібліотеки

Першим кроком у створенні сертифіката є генерація асиметричної пари ключів, що складається з приватного ключа та відповідного йому публічного ключа. Модуль повинен підтримувати як ключі RSA, так і ключі на основі еліптичних кривих (ECC).

Ключі RSA генеруються за допомогою:

cryptography.hazmat.primitives.asymmetric.rsa.generate_private_key().

- *public_exponent* зазвичай має бути встановлений на 65537.
- *key_size* є критичним параметром безпеки. Поточні рекомендації, такі як ті, що відповідають NIST SP 800-131A, пропонують мінімум 2048 біт. Для довговічності та підвищеної безпеки перевага надається 3072 або 4096 бітам. API модуля повинен дозволяти вказувати розмір ключа, за замовчуванням встановлюючи безпечне сучасне значення.

Ключі ECC генеруються за допомогою

cryptography.hazmat.primitives.asymmetric.ec.generate_private_key().

- Необхідно вказати відповідну еліптичну криву, наприклад, ec.SECP256R1() (еквівалент NIST P-256) або ec.SECP384R1() (NIST P-384).
- ECC пропонує порівнянну безпеку з RSA, але зі значно меншими розмірами ключів, що може бути перевагою для пристроїв IoT з обмеженими ресурсами. Формат публічних ключів ECC у полі subjectPublicKeyInfo сертифіката X.509 детально описаний у RFC 5480.

Вибір між RSA та ECC може залежати від таких факторів, як обчислювальні можливості пристроїв IoT, сумісність з існуючою інфраструктурою відкритих ключів (PKI) та специфічні вимоги до продуктивності програми.

Заповнення стандартних полів (згідно RFC 5280):

subject_name() та issuer_name():

- Ці методи приймають об'єкт x509.Name, який конструюється зі списку об'єктів x509.NameAttribute.
- Загальні атрибути представлені OID, доступними в x509.oid.NameOID (наприклад, NameOID.COMMON_NAME, NameOID.ORGANIZATION_NAME, NameOID.COUNTRY_NAME).
- Для самопідписаних сертифікатів імена суб'єкта та видавця однакові.
- Якщо сертифікат пристрою має видаватися Центром сертифікації (CA) – чи то локальним CA, керованим цим модулем, чи зовнішнім, ключ якого завантажено – issuer_name буде розрізнювальним ім'ям суб'єкта цього CA.

- ID пристрою в Common Name (CN): Деякі платформи IoT, такі як Azure IoT Hub, мають специфічні вимоги до поля CN. Для сертифікатів, підписаних CA, Azure IoT Hub часто очікує, що ID пристрою буде присутній у CN (наприклад, CN=my-device-id). Модуль повинен бути достатньо гнучким, щоб відповідати таким специфічним для платформи угодам.

serial_number():

- RFC 5280 вимагає, щоб серійний номер був додатним цілим числом, унікальним для кожного сертифіката, виданого даним CA.
- Функція x509.random_serial_number() може використовуватися для генерації криптографічно безпечного випадкового серійного номера. Альтернативно, API може дозволяти користувачеві вказувати серійний номер, за умови, що унікальність керується зовнішньо.

not_valid_before() та not_valid_after():

- Ці методи визначають період дії сертифіката. Вони приймають об'єкти datetime.datetime, які повинні бути в UTC.
- datetime.datetime.utcnow() (або datetime.now(timezone.utc) у Python 3) може використовуватися для поля not_valid_before.
- datetime.timedelta може використовуватися для обчислення дати not_valid_after.
- NIST SP 1800-16 рекомендує обмежувати період дії сертифікатів TLS-серверів одним роком або менше. Цей принцип є надзвичайно актуальним для сертифікатів пристроїв IoT для підвищення безпеки шляхом зменшення вікна вразливості у разі компрометації ключа та для заохочення регулярних процесів оновлення. Модуль повинен за замовчуванням встановлювати досить короткий період дії (наприклад, 1 рік), дозволяючи при цьому налаштування.

Асоціація публічного ключа:

- Метод public_key() CertificateBuilder використовується для зв'язування компонента публічного ключа згенерованої (або завантаженої) асиметричної пари ключів із сертифікатом, що конструюється [26, 45].

Основною вимогою до цього модуля є вбудовування унікальних ідентифікаторів пристроїв у сертифікати X.509. Це дозволяє точно ідентифікувати та автентифікувати пристрої.

Загальновживані ідентифікатори пристроїв включають MAC-адреси, апаратні серійні номери, hardwareModuleName, як визначено в RFC 4108, та ідентифікатори пристроїв IEEE 802.1AR (DevID). При виборі ідентифікатора слід надавати пріоритет стабільності (він не повинен змінюватися протягом експлуатаційного терміну пристрою) та унікальності (або глобальній, або в межах конкретного домену управління) [46].

3.2.5. Пакування у формат PKCS#12 за допомогою бібліотеки cryptography

Після генерації сертифіката X.509 пристрою та відповідної йому пари приватних ключів, їх необхідно безпечно об'єднати для зберігання або передачі на пристрій. Формат PKCS#12 широко використовується для цього завдання завдяки своїй здатності містити приватні ключі, сертифікати та сертифікати CA в одному, опціонально захищеному паролем файлі. Це полегшує переносимість та резервне копіювання.

1. Об'єднання сертифіката пристрою, приватного ключа та ланцюжка сертифікатів CA

Бібліотека cryptography надає функцію *cryptography.hazmat.primitives.serialization.pkcs12.serialize_key_and_certificates( )* для створення пакетів PKCS#12.

Ключові параметри для цієї функції:

- name: "Дружнє ім'я" для ідентичності, що зберігається у файлі PKCS#12. Це має бути рядок bytes (наприклад, b"Device IoT Certificate").
- key: Об'єкт приватного ключа пристрою (наприклад, екземпляр RSAPrivateKey або EllipticCurvePrivateKey). Цей ключ повинен бути завантажений в пам'ять або бути результатом етапу генерації.
- cert: Об'єкт сертифіката X.509 пристрою (екземпляр x509.Certificate).

- cas: Ітерабельний об'єкт (наприклад, список) додаткових об'єктів сертифікатів X.509, що утворюють ланцюжок сертифікатів для сертифіката пристрою. Зазвичай сюди входять сертифікати проміжних CA та, за бажанням, сертифікат кореневого CA. Порядок сертифікатів у цьому списку може бути важливим для деяких систем, що обробляють файл PKCS#12; зазвичай він має йти від проміжного CA, найближчого до сертифіката пристрою, до кореневого. Якщо додаткові сертифікати CA не включаються, цей параметр може бути None або порожнім списком.

2. Захист паролем файлу PKCS#12 за допомогою BestAvailableEncryption

Файли PKCS#12 містять приватний ключ, який є надзвичайно конфіденційним. Тому критично важливо захистити пакет надійним паролем. Функція serialize_key_and_certificates() підтримує це через свій параметр encryption_algorithm.

- Рекомендований спосіб вказати шифрування – використання cryptography.hazmat.primitives.serialization.BestAvailableEncryption(password_bytes).
- password_bytes: Обраний пароль, закодований як рядок bytes (наприклад, b"P@$$wOrdProtected!").
- BestAvailableEncryption автоматично вибирає надійний, наразі рекомендований алгоритм шифрування (та пов'язані параметри, такі як KDF та алгоритм MAC) для PKCS#12, абстрагуючи цю складність від розробника. Це допомагає забезпечити захист пакета за допомогою сучасних криптографічних практик.

Включення захисту паролем є не просто опцією, а фундаментальною вимогою безпеки при роботі з файлами PKCS#12, що містять приватні ключі. Без нього будь-яка компрометація файлу PKCS#12 безпосередньо призведе до компрометації приватного ключа пристрою та всієї його ідентичності. Тому модуль повинен або примусово застосовувати захист паролем, або, щонайменше, наполегливо рекомендувати його та використовувати безпечні значення за замовчуванням, якщо пароль не надано явно (хоча вимога пароля є кращою практикою).

3. Серіалізація та зберігання пакета PKCS#12

Функція serialize_key_and_certificates() повертає пакет PKCS#12 як об'єкт bytes. Цей байтовий рядок потім слід записати у файл, зазвичай з розширенням .p12 або .pfx. Файл необхідно відкривати в режимі двійкового запису ('wb').

4. Безпечне управління криптографічними ключами

Безпека всієї системи PKI залежить від захисту приватних ключів. У цьому розділі викладено найкращі практики генерації, обробки, зберігання та передачі цих критично важливих активів у контексті модуля Python [52, 53, 54, 55].

3.2.7. Безпечна передача/надання пакетів PKCS#12 на пристрої

Цей модуль відповідає за генерацію пакета PKCS#12. Подальша безпечна передача та надання цього пакета на пристрій IoT є критичним, окремим етапом життєвого циклу пристрою. Існує кілька стратегій, залежно від виробничого процесу та можливостей пристрою.

Пакети PKCS#12 можуть бути введені в прошивку пристрою або захищену пам'ять під час виробничого процесу у фізично захищеному середовищі. Це часто включає використання спеціалізованих програматорів або захищених виробничих ліній. Хмарні платформи, такі як AWS IoT, підтримують механізми Just-in-Time Provisioning (JITP) або Just-in-Time Registration (JITR), де CA, що підписав сертифікат пристрою (це може бути приватний CA виробника), реєструється на платформі, і пристрій автоматично реєструється при першому підключенні.

Протоколи, такі як Enrollment over Secure Transport (EST) або Certificate Management Protocol (CMP), дозволяють пристроям генерувати власну пару ключів та CSR безпосередньо на пристрої. Потім CSR надсилається на підпис CA (який може використовувати цей модуль Python на бекенді). CA повертає підписаний сертифікат на пристрій. Цей метод має перевагу в тому, що приватний ключ ніколи не залишає пристрій [56].

Якщо пакет PKCS#12 передається безпосередньо на пристрій після виробництва, це повинно відбуватися через безпечний, автентифікований та зашифрований канал. Можуть використовуватися протоколи, такі як TLS (для зв'язку

сервер-пристрій) або SFTP (Secure File Transfer Protocol). Власний захист паролем файлу PKCS#12 забезпечує додатковий рівень безпеки під час цієї передачі [40].

Пароль, обраний для пакета PKCS#12, також повинен бути переданий або наданий пристрою через безпечний механізм, що дозволить пристрою розшифрувати пакет та отримати доступ до свого приватного ключа та сертифіката [38]. Техніки безпечного поводження з приватними ключами наведені в таблиці 3.3.

Таблиця 3.3

Техніки безпечного поводження з приватними ключами

Техніка	**Опис**	**Релевантність/Інструменти Python cryptography**	**Важливість в контексті IoT**
Шифрування в стані спокою	Шифрування файлів приватних ключів, що зберігаються на диску, за допомогою надійного шифрування на основі пароля.	private_key.private_bytes() з serialization.BestAvailableEncryption(password) для файлів ключів.	Критично для будь-яких збережених ключів CA або ключів пристроїв, що очікують надання.
Надійний контроль доступу (рівень ОС)	Обмеження прав доступу до файлової системи для файлів ключів та адміністративного доступу до систем, що обробляють ключі.	Команди, специфічні для ОС (наприклад, chmod, icacls). Не є безпосередньою функцією cryptography, але важливо для середовища.	Важливо для захисту станцій надання та систем CA.
Використання HSM/KMS	Перенесення зберігання приватних ключів та/або криптографічних операцій (особливо підписання) на спеціалізоване обладнання або керовані хмарні сервіси.	Взаємодія через бібліотеки PKCS#11 (зовнішні до ядра cryptography, але можуть використовуватися разом) або SDK хмарних провайдерів.[20]	Ідеально для розгортань з високим рівнем безпеки, захисту кореневих/проміжних ключів CA або високоцінних ключів пристроїв.

продовження таблиці 3.3

Техніка	**Опис**	**Релевантність/Інструменти Python cryptography**
Надійні, унікальні паролі	Використання складних, унікальних паролів для шифрування приватних ключів та пакетів PKCS#12.	Визначається користувачем. Інструменти управління паролями можуть допомогти. Фундаментально. Слабкі паролі нівелюють переваги шифрування.
Принцип найменших привілеїв для процесів	Забезпечення того, щоб процеси, які обробляють ключі, мали лише необхідні дозволи для виконання своїх завдань.	Ізоляція процесів на рівні ОС та управління обліковими записами користувачів. Зменшує вплив у разі компрометації процесу, що обробляє ключі.
Обнулення в пам'яті (з обмеженнями)	Спроба перезаписати дані ключа в пам'яті після використання для зменшення ризику відновлення з дампів пам'яті.	Використання bytearray та ручне обнулення, де це можливо. cryptography сама по собі не гарантує очищення для незмінних bytes.69
Безпечні протоколи передачі	Використання зашифрованих та автентифікованих каналів при передачі матеріалу ключа або пакетів PKCS#12 на пристрої або інші системи.	Модуль ssl Python для клієнтів/серверів TLS. Необхідно для віддаленого надання або при передачі пакетів з точки генерації до системи розгортання.

3.2.8. Обробка помилок та безпечне ведення журналів

Надійна обробка помилок та безпечне ведення журналів є життєво важливими для надійності, супроводжуваності та безпеки модуля генерації сертифікатів. Криптографічні операції можуть зазнавати невдачі з різних причин, а журнали можуть надавати безцінну діагностичну інформацію, але вони не повинні ставати джерелом витоку конфіденційних даних.

1. Управління поширеними винятками з бібліотеки cryptography

Бібліотека cryptography розроблена таким чином, щоб викликати конкретні винятки при виникненні помилок, що дозволяє детально обробляти помилки. Модуль повинен передбачати та належним чином обробляти їх.

Поширені винятки включають:

- ValueError: Часто виникає через недійсні вхідні значення, такі як неправильно сформовані дані PEM/DER, неправильний пароль для зашифрованого ключа або файлу PKCS#12, або недійсний рядок OID.
- TypeError: Виникає через неправильні типи вхідних даних, переданих функціям або методам.
- cryptography.exceptions.UnsupportedAlgorithm: Вказує на те, що запитаний криптографічний алгоритм або крива не підтримується базовою бібліотекою OpenSSL або поточною версією cryptography.
- cryptography.exceptions.InvalidSignature: Виникає, якщо перевірка підпису не вдається (більш актуально, якщо модуль також завантажує та перевіряє сертифікати, наприклад, сертифікат CA, перед використанням).
- cryptography.x509.DuplicateExtension: Виникає, якщо робиться спроба додати той самий тип розширення X.509 більше одного разу до CertificateBuilder.
- cryptography.x509.UnsupportedGeneralNameType: Виникає, якщо розширення SubjectAlternativeName містить тип загального імені, який бібліотека не підтримує.

Модуль повинен реалізувати блоки try...except для перехоплення цих конкретних винятків. Замість того, щоб дозволяти необробленим виняткам поширюватися до викликаючого коду, часто краще перехопити їх, а потім або:

1. Надати більш зрозуміле для користувача повідомлення про помилку.
2. Викликати власний, специфічний для модуля виняток, який інкапсулює вихідну помилку, полегшуючи викликаючій програмі обробку помилок, що походять саме з цього модуля. Слід уникати перехоплення загального Exception, якщо тільки намір не полягає в тому, щоб зажурналити його, а потім негайно повторно викликати.

Впровадження практик безпечного ведення журналів:

Ведення журналів є важливим для налагодження проблем та аудиту подій, пов'язаних з безпекою, таких як генерація сертифікатів або спроби завантаження ключів. Однак журнали можуть стати значним ризиком безпеки, якщо вони ненавмисно фіксують конфіденційну інформацію [57, 58].

3.2.9. Інтеграція та аспекти життєвого циклу

Модуль Python для генерації сертифікатів X.509 та пакування у формат PKCS#12, хоч і є критично важливим компонентом, функціонує в рамках ширшої екосистеми управління ідентифікацією пристроїв. Його успішне розгортання залежить від того, наскільки добре він інтегрується в існуючі або заплановані робочі процеси надання (provisioning) та як керується життєвий цикл згенерованих облікових даних [59].

У безпечному виробничому середовищі сервер надання може використовувати цей модуль для генерації унікальних сертифікатів X.509 та пакетів PKCS#12 для кожного пристрою. Ці пакети потім безпечно програмуються в прошивку пристрою або захищений елемент зберігання перед тим, як він покине завод. Цей підхід поширений для пристроїв, які повинні бути готові до безпечного підключення одразу після розпакування [40].

3.2.10. Поновлення та відкликання сертифікатів

Генерація сертифіката – це лише початок його життєвого циклу. Ефективне управління цим життєвим циклом, включаючи поновлення та відкликання, є критично важливим для підтримки безпеки.

Сертифікати X.509 мають обмежений термін дії. Прострочені сертифікати не дозволять пристроям автентифікуватися та отримувати доступ до сервісів, що призведе до операційних збоїв [41].

Автоматизовані процеси поновлення сертифікатів є необхідними, особливо для великомасштабних розгортань IoT. Описаний модуль Python може бути повторно

використаний під час процесу поновлення для генерації нового сертифіката. Це може включати генерацію нової пари ключів [60, 61].

3.3. Механізм перевірки сертифікатів у IoT-мережі

У цьому підрозділі детально розглядаються механізми перевірки сертифікатів X.509 у мережах IoT, зосереджуючись на автоматичній перевірці валідності перед встановленням з'єднання та на методах захисту від підробки сертифікатів через використання апаратних атрибутів пристрою. Ці механізми є критично важливими для забезпечення безпеки та довіри в екосистемах IoT.

3.3.1. Автоматична перевірка валідності сертифікатів перед встановленням з'єднання

Автоматична перевірка валідності сертифікатів X.509 є фундаментальним етапом забезпечення безпечного зв'язку в IoT-мережах. Цей процес відбувається під час встановлення захищеного з'єднання, зазвичай за протоколом TLS (Transport Layer Security), і включає кілька ключових кроків, які виконуються стороною, що перевіряє (наприклад, сервером, який перевіряє сертифікат клієнтського IoT-пристрою, або клієнтом, що перевіряє сертифікат сервера) [62, 63].

3.3.2. Процес валідації сертифіката X.509

Процес валідації сертифіката X.509 (відповідно до RFC 5280) – це багатоетапна процедура, спрямована на підтвердження автентичності, цілісності сертифіката та довіри до його видавця. Ключові етапи перевірки охоплюють:

1. Перевірку ланцюжка сертифікатів: Побудова шляху від сертифіката кінцевого суб'єкта до довіреного кореневого Центру Сертифікації (CA) та перевірка цифрового підпису кожного сертифіката у цьому ланцюжку. Для самопідписаних сертифікатів необхідне попередньо встановлення довіри.

2. Контроль терміну дії: Перевірка, чи поточна дата та час знаходяться в межах періоду валідності (notBefore та notAfter), вказаного в кожному сертифікаті ланцюжка.

3. З'ясування статусу відкликання: Перевірка, чи не був сертифікат відкликаний видавцем достроково (наприклад, через компрометацію ключа), з використанням Списків відкликаних сертифікатів (CRL) або Протоколу статусу онлайн-сертифіката (OCSP) [64].

4. Аналіз обмежень та політик: Перевірка спеціальних розширень сертифіката, таких як BasicConstraints (чи є сертифікат CA, довжина шляху), KeyUsage (дозволені криптографічні операції) та ExtendedKeyUsage (додаткові цілі використання ключа).

5. Перевірку імені: Зіставлення імені, вказаного в полі Subject або розширенні SubjectAlternativeName (SAN) сертифіката, з очікуваним ідентифікатором ресурсу (наприклад, ім'ям хоста для сервера або ID пристрою для клієнта) [65].

3.3.3. Проблеми валідації самопідписаних сертифікатів в IoT:

Самопідписані сертифікати часто використовуються в тестових середовищах або для внутрішніх комунікацій в ізольованих IoT-системах через їхню простоту та відсутність витрат на видачу CA. Однак їх валідація має свої особливості та виклики:

Стандартні механізми CRL/OCSP не застосовуються безпосередньо до самопідписаних сертифікатів. Відкликання скомпрометованого самопідписаного сертифіката означає видалення його зі списків довірених на всіх пристроях, що йому довіряють, що може бути складним у масштабних IoT-системах.

Ручне управління довірою до кожного самопідписаного сертифіката на кожному пристрої не масштабується для великих IoT-мереж [66].

Незважаючи на ці виклики, автоматизована перевірка (після встановлення початкової довіри) все одно включає перевірку підпису (власним ключем), терміну дії та відповідності політикам (KeyUsage, EKU), якщо вони визначені.

Бібліотека cryptography в Python надає інструменти для реалізації процесу валідації сертифікатів. Модуль cryptography.x509.verification дозволяє створювати політики перевірки та валідувати ланцюжки сертифікатів.

Основні етапи валідації на прикладі реалізації:

with open(cert_pem_path, 'rb') as cert_file:

cert = x509.load_pem_x509_certificate(cert_file.read(), backend=default_backend())

- Зчитування PEM-сертифіката: Сертифікат у форматі PEM завантажується з диску і розпарсюється як об'єкт x509.Certificate. Це дає змогу оперувати його полями: терміном дії, підписом, суб'єктом, емітентом та публічним ключем.

print("Сертифікат дійсний до:", cert.not_valid_after_utc)

print("CN:", cert.subject.get_attributes_for_oid(NameOID.COMMON_NAME)[0].value)

Перевірка дати дійсності (*not_valid_before / not_valid_after*) дає змогу переконатися, що сертифікат не прострочений і не є передчасно активованим.

Перевірка імені (CN): Зчитується загальне ім’я пристрою, що дозволяє перевірити його ідентичність. Цей параметр критично важливий для IoT-сценаріїв з фіксованими DNS-іменами пристроїв.

except Exception as e:

print("Помилка перевірки сертифіката:", e)

Обробка винятків: Здійснюється базова перевірка валідності структури сертифіката (наприклад, неправильна кодировка або відсутній ключ).

Проблеми валідації самопідписаних сертифікатів в IoT:

Самопідписані сертифікати часто використовуються в тестових середовищах або для внутрішніх комунікацій в ізольованих IoT-системах через їхню простоту та відсутність витрат на видачу CA [27, 28].

3.3.4. Захист від підробки сертифікатів через апаратні атрибути

Існує кілька способів включення апаратних ідентифікаторів (таких як MAC-адреса, серійний номер виробника, ідентифікатор TPM тощо) до сертифіката X.509. Вибір методу залежить від стандартів, що підтримуються, політик CA та вимог конкретної IoT-системи.

1. Розширення SubjectAlternativeName (SAN) з типом otherName:

• Структура: Розширення SAN дозволяє включати різні типи імен. Тип otherName є гнучким контейнером, який визначається парою: type-id (Object Identifier, OID) та value (значення, закодоване відповідно до type-id).

• type-id: Це OID, який вказує на семантику та формат значення. Для апаратних ідентифікаторів можуть використовуватися стандартні OID (якщо існують для конкретного типу ідентифікатора) або приватні/зареєстровані OID організації.

• value: Це поле містить сам апаратний ідентифікатор, закодований у форматі ASN.1 DER (Distinguished Encoding Rules) відповідно до типу, визначеного type-id. Наприклад, MAC-адреса або серійний номер можуть бути закодовані як OCTET STRING або PrintableString / UTF8String [67, 68].

Приклади OID для апаратних ідентифікаторів:

• id-on-hardwareModuleName (OID: 1.3.6.1.5.5.7.8.4): Визначений у RFC 4108, цей OID використовується для ідентифікації апаратного модуля. Він сам є послідовністю, що містить hwType (OID, що ідентифікує тип обладнання) та hwSerialNum (OCTET STRING, що містить серійний номер). Організаціям потрібно визначити власний OID для hwType, якщо вони хочуть представляти, наприклад, MAC-адресу як тип апаратного забезпечення.

• TCG (Trusted Computing Group) OIDs: TCG реєструє OID, які можуть використовуватися для представлення інформації, пов'язаної з TPM та іншими апаратними механізмами довіри. Наприклад, OID tcg-at-platformManufacturerStr (2.23.133.5.1.1) може містити рядок з моделлю платформи.

• IEEE 802.1AR DevID: Цей стандарт визначає використання сертифікатів для ідентифікації пристроїв (DevID) і може включати permanentIdentifier в

subjectAltName otherName. Конкретні OID та формати кодування для DevID в otherName детально описані в повному стандарті IEEE 802.1AR. Наприклад, id-on-hardwareModuleName з RFC 4108 може використовуватися для представлення serialNumber як hwSerialNum з відповідним hwType OID.

- Приватні OID: Організації можуть реєструвати власні OID під своїм приватним корпоративним номером (PEN) для представлення специфічних апаратних атрибутів [67, 68].

Кодування значення otherName: Значення value в otherName має бути ASN.1 DER-закодованою структурою. Якщо апаратний ідентифікатор є рядком, він може бути закодований як UTF8String або PrintableString. Якщо це бінарні дані (наприклад, MAC-адреса як послідовність байтів), він може бути закодований як OCTET STRING [67, 69].

Розширення SubjectDirectoryAttributes (OID: 2.5.29.9):

- Структура: Це розширення дозволяє включати набір атрибутів каталогу, пов'язаних із суб'єктом сертифіката. Воно визначається як SEQUENCE OF Attribute. Кожен Attribute є послідовністю AttributeType (OID) та SET OF AttributeValue (де AttributeValue є ANY DEFINED BY AttributeType).
- Використання для апаратних ідентифікаторів: Можна визначити OID для типу атрибута (наприклад, deviceSerialNumber або deviceMACAddress) і включити відповідне значення. Стандартний атрибут serialNumber (OID 2.5.4.5) з PKCS#9 може бути використаний тут для представлення серійного номера пристрою, якщо це дозволено політикою CA. Для MAC-адрес, ймовірно, знадобиться власний OID.
- Кодування AttributeValue: Значення атрибута (AttributeValue) кодується відповідно до його ASN.1 типу. Для рядкових ідентифікаторів це часто DirectoryString (який може бути UTF8String, PrintableString тощо).
- Обробка в cryptography: Бібліотека cryptography парсить розширення SubjectDirectoryAttributes. Об'єкт розширення SubjectDirectoryAttributes є ітерабельним і повертає об'єкти x509.Attribute. Кожен об'єкт x509.Attribute має властивості oid та value. Властивість value зазвичай є рядком Python (str), якщо

базовий тип ASN.1 є поширеним рядковим типом (наприклад, UTF8String, PrintableString). Для інших типів це можуть бути байти [70].

1. Атрибут serialNumber у Subject Distinguished Name (DN):

• Використання: OID NameOID.SERIAL_NUMBER (2.5.4.5) може бути використаний для включення серійного номера безпосередньо в DN суб'єкта сертифіката.

• Обмеження: Хоча технічно можливо, RFC 5280 зазвичай передбачає, що serialNumber в DN відноситься до серійного номера сертифіката, виданого CA, або серійного номера, пов'язаного з людиною. Використання його для апаратного серійного номера може призвести до неоднозначності або бути менш семантично коректним порівняно з використанням спеціалізованих розширень. Однак, деякі профілі можуть це дозволяти.

• Обробка в cryptography: Легко додати за допомогою x509.NameAttribute(NameOID.SERIAL_NUMBER, "device_hw_serial") при побудові x509.Name. Парсинг здійснюється ітерацією по cert.subject та пошуком атрибута з відповідним OID [70].

3.3.5. Серверна перевірка вбудованих апаратних атрибутів

Після стандартної валідації сертифіката X.509 (як описано в 3.3.1), сервер повинен виконати додаткові кроки для перевірки апаратних атрибутів, вбудованих у сертифікат:

1. Вилучення апаратного атрибута з сертифіката: Сервер парсить сертифікат, знаходить відповідне розширення (SubjectAlternativeName або SubjectDirectoryAttributes) або поле (Subject DN) і декодує значення апаратного ідентифікатора. Цей процес залежить від обраного методу вбудовування, як показано у прикладах коду вище.

2. Отримання фактичного апаратного атрибута від пристрою, що підключається: Це критично важливий і часто найскладніший етап. Сервер повинен мати надійний спосіб отримати справжній апаратний атрибут від пристрою, що намагається встановити з'єднання.

Проблема "курки та яйця": Якщо пристрій передає свій апаратний атрибут через TLS-канал, що встановлюється, то безпека цього каналу залежить від валідності сертифіката, яку ми саме намагаємося посилити. Це створює залежність: канал захищений сертифікатом, а сертифікат перевіряється атрибутом, переданим через цей же канал.

Можливі рішення:

- Попередня автентифікація або безпечне завантаження (bootstrapping): Пристрій може пройти початковий етап автентифікації (наприклад, за допомогою попередньо розподіленого ключа, тимчасового сертифіката або іншого механізму), під час якого він безпечно передає свій апаратний атрибут [71].
- Довірений реєстр пристроїв: Сервер може мати доступ до захищеного реєстру пристроїв, де зберігаються очікувані апаратні атрибути для кожного зареєстрованого пристрою. Цей реєстр має наповнюватися під час безпечного процесу виробництва або введення пристрою в експлуатацію. Сервер може використовувати інший ідентифікатор з сертифіката (наприклад, CommonName суб'єкта) для пошуку очікуваного апаратного атрибута в цьому реєстрі [72].
- Атестація TPM: Якщо пристрій оснащений TPM, апаратний атрибут (або пов'язаний з ним ключ) може бути атестований TPM, і ця атестація може бути включена до сертифіката (наприклад, через специфічні OID від TCG) або передана окремо під час встановлення з'єднання. Це забезпечує вищий рівень довіри до апаратного атрибута.

3. Порівняння атрибутів: Сервер порівнює апаратний атрибут, вилучений з сертифіката, з фактичним атрибутом, отриманим від пристрою або з довіреного реєстру.

4. Прийняття рішення: Якщо атрибути збігаються і стандартна валідація сертифіката пройшла успішно, з'єднання дозволяється. В іншому випадку з'єднання відхиляється.

Хоча otherName та SubjectDirectoryAttributes надають гнучкі контейнери, бракує універсально прийнятих, стандартних OID спеціально для поширених апаратних ідентифікаторів, таких як MAC-адреси або загальні серійні номери, для використання

в цих розширеннях у всіх контекстах IoT. Стандарт RFC 4108 id-on-hardwareModuleName є релевантним, але вимагає визначення OID для hwType. TCG надає OID для атестацій, пов'язаних з TPM. Це означає, що організації, які впроваджують прив'язку до апаратних атрибутів, повинні ретельно керувати своїм простором OID або дотримуватися конкретних галузевих стандартів чи стандартів консорціумів (наприклад, IEEE 802.1AR для DevID), якщо такі існують та застосовні. Вибір OID та схеми кодування стає критичною частиною проектування системи [27].

3.3.6. Роль ASN.1 та DER у практичній реалізації

Додавання власних даних до otherName вимагає, щоб значення було DER-закодованим ASN.1. SubjectDirectoryAttributes також включає структури ASN.1. Розробники не можуть просто вставляти необроблені рядки або масиви байтів у ці поля сертифіката. Вони повинні розуміти необхідну структуру ASN.1 (наприклад, OCTET STRING, UTF8String або складнішу SEQUENCE) для свого конкретного апаратного ідентифікатора та використовувати бібліотеку, таку як pyasn1, для правильного DER-кодування при включенні та декодування при перевірці. Це додає шар складності до реалізації. Бібліотека cryptography в Python обробляє значну частину високорівневої структури X.509, але для цих спеціальних корисних навантажень у otherName.value або складних AttributeValue в SubjectDirectoryAttributes часто необхідна безпосередня маніпуляція ASN.1 [73].

Виклики, пов'язані з апаратними ідентифікаторами :

- Підробка MAC-адрес: MAC-адреси можуть бути підроблені, тому покладатися виключно на них ризиковано без поєднання з іншими сильними факторами автентифікації або безпечним наданням.
- Заміна обладнання: Якщо компонент з ідентифікатором (наприклад, мережева карта для MAC) замінюється, сертифікат стає недійсним, що вимагає повторного надання.
- Проблеми конфіденційності: Вбудовування постійних апаратних ідентифікаторів у сертифікати, які можуть передаватися або реєструватися, може викликати проблеми з конфіденційністю.

- Складність виробництва та надання: Безпечне вбудовування цих атрибутів під час виробництва або надання вимагає надійного процесу. Безпека прив'язки до апаратних атрибутів значною мірою залежить від цілісності процесу надання, під час якого сертифікат пов'язується з апаратним забезпеченням, та надійності самого апаратного ідентифікатора. Атестації на основі TPM пропонують сильніші гарантії, ніж прості MAC-адреси/серійні номери [28, 71].

3.3.7. Практичні аспекти реалізації з використанням коду

Ключовим аспектом є забезпечення того, щоб приватні ключі ніколи не залишали пристрій. В ідеалі, пристрій генерує пару ключів (можливо, в TPM), створює запит на підписання сертифіката (CSR), що включає його апаратні атрибути (які самі можуть бути атестовані TPM), і надсилає CSR до CA (це може бути внутрішній/приватний CA). CA, у свою чергу, повинен мати безпечний спосіб перевірки апаратних атрибутів, наданих у CSR, перед їх вбудовуванням у сертифікат. Після підписання, сертифікат (та ланцюжок CA) безпечно доставляється назад на пристрій. Весь цей процес надання має гарантувати, що правильний сертифікат доставлений на правильний пристрій без перехоплення чи модифікації.

Практика використання бібліотеки cryptography в Python:

- Приватні ключі CA є надзвичайно чутливими. Їх слід завантажувати з захищених джерел, таких як зашифровані файли (наприклад, захищені паролем PEM-файли, використовуючи BestAvailableEncryption з cryptography.hazmat.primitives.serialization) або, в ідеалі, зберігати та використовувати в HSM (Hardware Security Module) (рис. 3.1).

```
cert_path = os.path.join(OUTPUT_DIR, f"{filename_prefix}_cert.pem")
key_path = os.path.join(OUTPUT_DIR, f"{filename_prefix}_key.pem")

with open(cert_path, "wb") as f:
    f.write(cert.public_bytes(serialization.Encoding.PEM))
```

Рисунок 3.1 – Запис у .pem файл

- Час утримання приватних ключів у пам'яті має бути мінімізований.
- Очищення об'єктів приватних ключів з пам'яті після використання є складним завданням у Python через його управління пам'яттю та незмінні типи, такі як bytes. Хоча del видаляє посилання на об'єкт, це не гарантує стирання даних з пам'яті. Використання змінних буферів (наприклад, bytearray) та їх явне обнулення перед видаленням посилання є частковим заходом. Для високочутливих ключів, таких як приватний ключ CA, запуск програмного забезпечення CA в контрольованому середовищі з обмеженим доступом до пам'яті є критично важливим. Без HSM, безпека пам'яті приватних ключів у Python значною мірою покладається на операційну безпеку хост-системи [53, 75, 76].

Формування пакетів PKCS#12 для управління сертифікатами та ключами :

- PKCS#12 (PFX) є стандартом для зберігання приватного ключа, сертифіката та ланцюжка CA в одному зашифрованому файлі, що забезпечує портативність та безпеку.
- Бібліотека cryptography надає функції serialize_key_and_certificates для створення та load_pkcs12 для парсингу файлів PKCS#12, підтримуючи шифрування за допомогою BestAvailableEncryption.
- Безпечна передача файлів PKCS#12 на пристрої (якщо вони не генеруються безпосередньо на пристрої) є критичним етапом і повинна здійснюватися через захищені канали (наприклад, SFTP, SCP) або під час безпечного виробничого процесу.

Обробка поширених винятків та надійне повідомлення про помилки :

- Бібліотека cryptography може генерувати різні винятки, такі як ValueError, TypeError, UnsupportedAlgorithm, InvalidSignature, InvalidKey, ExtensionNotFound, DuplicateExtension.
- Реалізація блоків try-except для коректної обробки помилок під час генерації, парсингу та валідації сертифікатів є обов'язковою для стабільної роботи системи.

https://blog.heycoach.in/exception-handling-in-cryptography/

Безпечне логування подій верифікації :

- Логування є важливим для аудиту безпеки, моніторингу та реагування на інциденти. Необхідно реєструвати успішні/невдалі спроби верифікації, причини невдач, ідентифікатори клієнтів (наприклад, IP, CN суб'єкта) та перевірені апаратні атрибути.
- Критично важливо запобігати витоку чутливих даних у логах (наприклад, приватних ключів, повного вмісту сертифікатів, якщо це не є абсолютно необхідним). Для цього використовуються фільтри логування [78].

https://cheatsheetseries.owasp.org/cheatsheets/Logging_Cheat_Sheet.html

Валідація вхідних параметрів для генерації сертифікатів :

- Параметри, що використовуються для генерації сертифікатів (наприклад, формат загального імені, формат/довжина апаратного ідентифікатора) або отримані під час перевірки, повинні валідуватися. Це включає перевірку типів, діапазонів, форматів та санітизацію для запобігання атакам ін'єкцій або генерації некоректних сертифікатів [51, 79].

3.4. NIST та галузеві найкращі практики для перевірки сертифікатів IoT

Впровадження надійних механізмів перевірки сертифікатів в IoT-системах повинно узгоджуватися з рекомендаціями провідних організацій у сфері кібербезпеки, таких як NIST (Національний інститут стандартів і технологій США) та OWASP (Open Web Application Security Project), а також іншими галузевими найкращими практиками.

Узгодження з керівництвом NIST:

1. NIST SP 800-213A:

- Прив'язка сертифікатів до апаратних атрибутів безпосередньо підтримує такі рекомендовані можливості, як (IMS) Підтримка управління ідентифікаторами, (AID) Дії на основі ідентичності пристрою та (DAS) Підтримка автентифікації пристрою. Ці можливості наголошують на унікальній ідентифікації пристроїв та виконанні дій на основі цієї ідентичності.

• Можливість (CRY) Криптографічні можливості та підтримка включає вимогу "(2) Здатність отримувати та валідувати сертифікати", що є основою обговорюваних механізмів.

2. NIST SP 1800-16: Забезпечення безпеки веб-транзакцій: Управління сертифікатами TLS-серверів :

• Хоча цей документ зосереджений на сертифікатах TLS-серверів, його принципи значною мірою застосовні до сертифікатів IoT-пристроїв. Це включає ведення інвентарю сертифікатів, призначення відповідальних, використання затверджених CA, обмеження термінів дії, безпеку приватних ключів, проактивне оновлення, механізми відкликання та постійний моніторинг. Для IoT-середовищ особливого значення набуває автоматизація цих процесів через велику кількість пристроїв та їхні потенційні ресурсні обмеження [39, 41].

NIST послідовно наголошує на необхідності унікальних ідентичностей для кожного IoT-пристрою та безпечних механізмів їх надання та автентифікації. Прив'язка до апаратних атрибутів є одним із способів реалізації таких сильних, перевіряємих ідентичностей [39].

Рекомендації OWASP:

• OWASP Proactive Controls - C1: Впровадження контролю доступу (раніше C7: Застосування контролю доступу) :

1. Перевірка сертифіката, включаючи перевірку апаратних атрибутів, повинна бути обов'язковим етапом перед наданням доступу до ресурсів або дозволом на виконання операцій.

2. Принцип "відмова за замовчуванням" означає, що доступ заборонено, якщо він явно не дозволений після успішної автентифікації та авторизації.

• OWASP Logging Cheat Sheet : Дотримання практик безпечного логування подій, пов'язаних з перевіркою сертифікатів, як детально описано в підрозділі 3.3.3, є критично важливим для уникнення витоку чутливої інформації. Інші галузеві найкращі практики:

• Управління життєвим циклом сертифікатів (Certificate Lifecycle Management, CLM) для IoT: Автоматизація процесів виявлення, оновлення та

відкликання сертифікатів є необхідною, особливо враховуючи тенденцію до скорочення термінів дії сертифікатів та велику кількість пристроїв в IoT-екосистемах.

- Принцип найменших привілеїв: Після успішної автентифікації та перевірки апаратних атрибутів, пристрою повинні надаватися лише ті дозволи, які необхідні для виконання його функцій.
- Апаратний корінь довіри (Hardware Root of Trust, HRoT): Використання захищених апаратних елементів, таких як TPM (Trusted Platform Module), для зберігання приватних ключів та атестації ідентичності пристрою значно підвищує рівень безпеки. TPM може надійно зберігати приватний ключ, пов'язаний із сертифікатом, і надавати атестації, які можуть використовуватися як самі апаратні атрибути або для їх підтвердження.
- Ефективність перевірки сертифікатів, особливо з прив'язкою до апаратних атрибутів, залежить від цілісності ширшої екосистеми безпеки. Якщо інші сфери, такі як безпечне надання облікових даних, захист ключів на пристрої або контроль доступу після автентифікації, є слабкими, то навіть найдосконаліша перевірка сертифікатів може бути знівельована. Наприклад, якщо приватний ключ пристрою легко витягти, незважаючи на наявність апаратних атрибутів у сертифікаті, зловмисник все ще може знайти способи експлуатації системи, особливо якщо він зможе підробити джерело апаратного атрибута. Отже, впровадження надійних механізмів перевірки сертифікатів повинно бути частиною комплексної стратегії захисту в глибину для IoT.
- Спостерігається конвергенція практик безпеки інформаційних технологій (IT) та операційних технологій (OT) в контексті IoT. Традиційні підходи до PKI та управління сертифікатами, розроблені переважно для IT-систем, адаптуються для задоволення унікальних потреб IoT, таких як масовість, обмежені ресурси пристроїв та тривалий життєвий цикл. Використання апаратних атрибутів є специфічною адаптацією, що відображає фізичну природу "речей" і робить процес верифікації більш тісно пов'язаним з фізичними або мікропрограмними характеристиками пристрою [37].

Висновок до третього розділу

У третьому розділі роботи було розроблено та детально описано механізм автентифікації для пристроїв Інтернету речей. Основну увагу приділено використанню самопідписаних цифрових сертифікатів X.509, розглянуто повний цикл їхнього життя: від генерації (з можливістю інтеграції унікальних апаратних характеристик пристрою для посилення безпеки) та безпечного зберігання у форматі PKCS#12, до механізмів валідації в мережах IoT. Також було представлено архітектуру та принципи реалізації програмного модуля для автоматизації процесу створення таких сертифікатів, що забезпечує основу для надійної автентифікації та захищеної взаємодії в екосистемах IoT.

РОЗДІЛ 4
РЕАЛІЗАЦІЯ МЕТОДУ ШИФРУВАННЯ ДЛЯ ІОТ

4.1. Розробка алгоритму шифрування на основі AES-256

Стандарт AES (Advanced Encryption Standard) з довжиною ключа 256 біт (AES-256) широко визнаний як один із найбезпечніших симетричних алгоритмів шифрування, що робить його привабливим вибором для захисту конфіденційної інформації в екосистемах IoT. Цей підрозділ детально розглядає аспекти розробки такого алгоритму, включаючи вибір режиму шифрування та управління ключами.

4.1.1. Огляд стандартів AES та їх актуальність для IoT

Advanced Encryption Standard (AES) є симетричним блоковим шифром, прийнятим урядом США для захисту секретної інформації. AES працює з блоками даних фіксованого розміру (128 біт) та підтримує ключі довжиною 128, 192 або 256 біт. AES-256, що використовує 256-бітний ключ, забезпечує найвищий рівень безпеки серед варіантів AES, виконуючи 14 раундів обробки даних. Кожен раунд включає операції заміни байтів (SubBytes), зсуву рядків (ShiftRows), змішування стовпців (MixColumns) та додавання ключа раунду (AddRoundKey), за винятком останнього раунду, де MixColumns пропускається [80, 81].

Актуальність AES-256 для IoT:

1. AES-256 вважається практично незламним сучасними обчислювальними засобами при правильній реалізації та управлінні ключами. Це критично важливо для IoT, де пристрої можуть збирати та передавати надзвичайно чутливі дані.
2. AES є глобальним стандартом, що забезпечує сумісність між різними пристроями та платформами. Це важливо для гетерогенних IoT-середовищ.
3. Хоча AES-256 потребує більше обчислювальних ресурсів порівняно з AES-128 (приблизно на 20-40% повільніше через більшу кількість раундів), багато

сучасних мікроконтролерів для IoT оснащені апаратними прискорювачами AES, що значно підвищує продуктивність та знижує енергоспоживання.

4. AES широко використовується в захищених протоколах передачі даних, таких як TLS/SSL, що є стандартом для захисту комунікацій в Інтернеті, включаючи IoT [82, 83].

4.1.2. Вибір оптимального режиму шифрування (GCM, CFB, CBC)

Вибір режиму шифрування для AES визначає, як саме блоковий шифр буде застосовуватися до послідовності блоків даних. Різні режими мають різні властивості щодо безпеки, продуктивності та вимог до реалізації. Для IoT-застосунків особливо важливими є забезпечення конфіденційності, цілісності та автентичності даних, а також ефективність на пристроях з обмеженими ресурсами.

AES-GCM (Galois/Counter Mode) є режимом автентифікованого шифрування з приєднаними даними (AEAD – Authenticated Encryption with Associated Data). Це означає, що він одночасно забезпечує:

- Конфіденційність: Шифрування даних за допомогою режиму CTR (Counter).
- Цілісність: Гарантія того, що дані не були змінені під час передачі.
- Автентичність: Перевірка походження даних та підтвердження того, що вони не були підроблені. Ці властивості досягаються шляхом генерації тегу автентифікації (GMAC).

Переваги для IoT:

- Комплексна безпека: Надання всіх трьох ключових аспектів безпеки (конфіденційність, цілісність, автентичність) в одній операції є значною перевагою, особливо для IoT, де цілісність команд та автентичність джерел даних є критичними.
- Висока продуктивність: GCM є ефективним режимом, особливо при наявності апаратної підтримки, і дозволяє паралельну обробку. Це важливо для пристроїв, що передають дані в реальному часі.

- Відсутність потреби в паддінгу: Як і режим CTR, GCM перетворює блоковий шифр на потоковий, тому паддінг не потрібен.
- Широке використання в TLS: AES-GCM є рекомендованим режимом для TLS 1.2 та стандартним для TLS 1.3, що робить його природним вибором для захисту комунікацій IoT.

Недоліки/Вимоги:

- Унікальність Nonce (IV): Критично важливо використовувати унікальний nonce (вектор ініціалізації) для кожного шифрування з тим самим ключем. Повторне використання nonce з тим самим ключем може призвести до катастрофічної втрати безпеки. Рекомендована довжина nonce – 12 байт (96 біт).

GCM вважається одним з найкращих режимів для більшості сучасних застосунків, включаючи IoT, завдяки поєднанню сильної безпеки та хорошої продуктивності.

У режимі CBC кожен блок відкритого тексту XOR-иться з попереднім блоком шифротексту перед шифруванням. Для першого блоку використовується вектор ініціалізації (IV).

Переваги:

- Сильна конфіденційність: Забезпечує хорошу дифузію; однакові блоки відкритого тексту дають різні блоки шифротексту при використанні одного ключа та IV.

Недоліки/Вимоги для IoT:

- Відсутність вбудованої цілісності та автентичності: CBC сам по собі забезпечує лише конфіденційність. Для забезпечення цілісності та автентичності необхідно використовувати окремий механізм, такий як HMAC (Hash-based Message Authentication Code), зазвичай у схемі "Encrypt-then-MAC". Це додає складності та обчислювальних витрат.
- Потреба в паддінгу: Якщо відкритий текст не є кратним розміру блоку AES (16 байт), потрібен паддінг (наприклад, PKCS7). Неправильна реалізація паддінгу може призвести до атак (padding oracle attacks).

- Послідовне шифрування: Процес шифрування в CBC є послідовним (кожен блок залежить від попереднього), що обмежує можливості паралелізації. Розшифрування може бути паралелізоване.
- Поширення помилок: Помилка в одному біті шифротексту впливає на розшифрування поточного та наступного блоків.
- Для IoT CBC може бути прийнятним лише у випадках, коли вимоги до цілісності та автентичності реалізуються окремо, або для застарілих систем. Однак, через додаткову складність реалізації MAC та потенційні вразливості, GCM є кращим вибором.

2. Режим CFB перетворює блоковий шифр на самосинхронізуючий потоковий шифр. Він шифрує попередній блок шифротексту (або IV для першого блоку), а потім XOR-ить результат з поточним блоком відкритого тексту для отримання шифротексту. Розмір сегмента зворотного зв'язку може варіюватися (наприклад, CFB8 для 8-бітного сегмента, CFB128 для 128-бітного).

Переваги:

- Потоковий шифр: Не потребує паддінгу, оскільки може шифрувати дані меншими порціями, ніж розмір блоку.
- Самосинхронізація: Втрата частини шифротексту призводить до втрати лише відповідної частини відкритого тексту (після кількох блоків помилок, залежно від розміру сегмента).

Недоліки/Вимоги для IoT:

- Відсутність вбудованої цілісності та автентичності: Як і CBC, CFB забезпечує лише конфіденційність і потребує окремого MAC.
- Продуктивність: CFB може бути повільнішим за інші режими, особливо якщо розмір сегмента малий (наприклад, CFB8 потребує виконання операції шифрування для кожного байта, тоді як CFB128 – для кожного блоку).
- Поширення помилок: Помилка в одному біті шифротексту впливає на розшифрування поточного блоку та кількох наступних (залежно від розміру сегмента та реалізації).

- Унікальність IV: Потребує унікального IV для кожного шифрування з тим самим ключем.

Для IoT CFB може бути корисним у сценаріях, де потрібне потокове шифрування без паддінгу, але, як і CBC, він поступається GCM через відсутність вбудованої автентифікації. Порівняльна таблиця режимів шифрування наведена в таблиці 4.1.

Таблиця 4.1

Порівняння режимів шифрування AES для IoT

Характеристика	**AES-GCM**	**AES-CBC**	**AES-CFB**
Конфіденційність	Так	Так	Так
Цілісність	Так (вбудована)	Ні (потребує окремого MAC)	Ні (потребує окремого MAC)
Автентичність	Так (вбудована)	Ні (потребує окремого MAC)	Ні (потребує окремого MAC)
AEAD	Так	Ні	Ні
Паддінг	Не потрібен	Потрібен	Не потрібен
Паралелізація	Так (шифрування/розшифрування)	Розшифрування (шифрування послідовне)	Залежить від реалізації
Продуктивність	Висока [12]	Середня [12]	Середня/Низька (залежить від сегмента) [19]
Складність реалізації	Середня (управління nonce)	Висока (паддінг + окремий MAC)	Висока (окремий MAC)
Вразливості (типові)	Неправильне використання nonce	Padding oracle, відсутність MAC	Відсутність MAC
Рекомендація для IoT	Рекомендовано	Не рекомендовано	Не рекомендовано

AES-GCM є найкращим вибором для більшості IoT-застосунків. Він забезпечує надійний захист "все-в-одному" (конфіденційність, цілісність, автентичність), що є критично важливим для безпеки IoT-систем, де дані можуть керувати фізичними процесами або містити приватну інформацію.Його ефективність, особливо з апаратною підтримкою, робить його придатним для пристроїв з обмеженими ресурсами. Унікальність nonce є ключовою вимогою, але сучасні бібліотеки та практики допомагають у її забезпеченні.

AES-CBC та AES-CFB слід використовувати з великою обережністю.** Основний недолік – відсутність вбудованої автентифікації та перевірки цілісності. Якщо ці режими використовуються, вони **обов'язково** повинні доповнюватися надійним механізмом MAC (наприклад, HMAC-SHA256) за схемою "Encrypt-then-MAC". Це ускладнює реалізацію, збільшує обсяг переданих даних (через MAC-тег) та потенційно знижує продуктивність. Для CBC також важливо правильно реалізувати паддінг, щоб уникнути вразливостей. Враховуючи доступність GCM, нові розробки для IoT повинні віддавати перевагу саме йому.

Вибір режиму шифрування має фундаментальний вплив на загальну безпеку системи. Хоча AES-256 сам по собі є сильним шифром, його безпека в конкретному застосунку значною мірою залежить від обраного режиму та коректності його реалізації. Для IoT, де пристрої часто є вразливими цілями та обробляють критичні дані, компроміси в безпеці, пов'язані з вибором застарілих або неповних режимів шифрування, є неприйнятними. Режим GCM надає необхідний рівень захисту, який відповідає сучасним вимогам безпеки для Інтернету речей [84, 85, 86].

4.1.3. Реалізація AES-256-GCM

- У наданому коді як оптимальний режим шифрування для AES було обрано GCM (Galois/Counter Mode). Це відображено у функції encrypt_aes_gcm та її використанні в основній логіці скрипта, а також у функції simulate_ipsec_channel.
- Функція encrypt_aes_gcm(key: bytes, data: bytes):

- iv = os.urandom(12): Генерується ініціалізаційний вектор (IV, також відомий як nonce в контексті GCM) довжиною 12 байт (96 біт) за допомогою криптографічно стійкого генератора випадкових чисел os.urandom(). Для режиму GCM критично важливо, щоб IV був унікальним для кожної операції шифрування з одним і тим же ключем.
- aesgcm = AESGCM(key): Ініціалізується об'єкт AESGCM з наданим симетричним ключем key. Передбачається, що цей ключ має довжину, сумісну з AES (наприклад, 32 байти для AES-256, як це забезпечується функцією HKDF або HMAC-SHA256 у коді).
- ciphertext = aesgcm.encrypt(iv, data, None): Виконується шифрування наданих даних data.
- iv: Унікальний ініціалізаційний вектор.
- data: Відкритий текст, який потрібно зашифрувати.
- None: Додаткові автентифіковані дані (AAD - Associated Authenticated Data). У цій функції вони не використовуються (встановлено як None), але режим GCM дозволяє їх включати для забезпечення цілісності цих даних разом із зашифрованим текстом. У функції simulate_ipsec_channel AAD використовується.
- Результат ciphertext містить як зашифровані дані, так і тег автентифікації, згенерований режимом GCM.
- return iv, ciphertext, b'': Функція повертає IV, шифротекст (з тегом) та порожній байтовий рядок. Повернення b'' як третього елемента може бути спрощенням, оскільки тег автентифікації вже є частиною ciphertext при використанні AESGCM.encrypt(). Більш типово було б повертати iv, ciphertext_with_tag.
- Режим GCM обраний через те, що він забезпечує одночасно конфіденційність, цілісність та автентичність зашифрованих даних (AEAD - Authenticated Encryption with Associated Data), що є критично важливим для безпеки IoT. Інші режими, такі як CFB або CBC, забезпечують лише конфіденційність і потребують окремої реалізації механізмів для перевірки цілісності та автентичності (наприклад, за допомогою HMAC). Код не демонструє процес вибору між GCM, CFB, CBC, а одразу реалізує GCM як оптимальний варіант (рис. 4.1).

```python
#Шифрування AES-GCM
def encrypt_aes_gcm(key: bytes, data: bytes):
    iv = os.urandom(12)
    aesgcm = AESGCM(key)
    ciphertext = aesgcm.encrypt(iv, data, None)
    return iv, ciphertext, b''
```

Рисунок 4.1 – Реалізація AES-GCM

- Наданий код не реалізує повний протокол TLS. Однак він демонструє реалізацію ключових криптографічних компонентів, які є фундаментальними для TLS та забезпечують захист даних, аналогічний тому, що надає TLS з AES-256:
- Генерація сертифікатів (Функція generate_self_signed_cert): TLS використовує сертифікати X.509 для автентифікації сторін (зазвичай сервера, іноді клієнта). Ця функція створює самопідписаний сертифікат, який може використовуватися для ідентифікації IoT-пристрою або сервера в тестових чи контрольованих середовищах (рис. 4.2).

```python
# Генерація самопідписаного сертифіката
def generate_self_signed_cert(common_name: str, filename_prefix: str, password: bytes):
    key = ec.generate_private_key(ec.SECP384R1(), default_backend())
    subject = x509.Name([
        x509.NameAttribute(NameOID.COUNTRY_NAME, u"UA"),
        x509.NameAttribute(NameOID.ORGANIZATION_NAME, u"IoT Lab"),
        x509.NameAttribute(NameOID.COMMON_NAME, common_name),
    ])
    cert = x509.CertificateBuilder().subject_name(subject)
    cert = cert.issuer_name(subject)
    cert = cert.public_key(key.public_key())
    cert = cert.serial_number(x509.random_serial_number())
    cert = cert.not_valid_before(datetime.datetime.utcnow())
    cert = cert.not_valid_after(datetime.datetime.utcnow() + datetime.timedelta(days=365))
    cert = cert.sign(key, hashes.SHA256(), default_backend())

    cert_path = os.path.join(OUTPUT_DIR, f"{filename_prefix}_cert.pem")
    key_path = os.path.join(OUTPUT_DIR, f"{filename_prefix}_key.pem")

    with open(cert_path, "wb") as f:
        f.write(cert.public_bytes(serialization.Encoding.PEM))

    with open(key_path, "wb") as f:
        f.write(key.private_bytes(
            encoding=serialization.Encoding.PEM,
            format=serialization.PrivateFormat.PKCS8,
            encryption_algorithm=serialization.BestAvailableEncryption(password)
        ))

    return cert, key
```

Рисунок 4.2 – Генерація сертифікатів та TLS

- Обмін ключами (Функція diffie_hellman_key_exchange та використання в if __name__ == '__main__':): TLS використовує механізми обміну ключами (наприклад, Діффі-Хеллмана на еліптичних кривих - ECDHE) для встановлення спільного секретного ключа між клієнтом та сервером. Код демонструє класичний обмін Діффі-Хеллмана.
- parameters = dh.generate_parameters(generator=2, key_size=2048, backend=default_backend()): Генеруються параметри DH (велике просте число та генератор).
- private_key = parameters.generate_private_key(): Кожна сторона генерує свій приватний ключ.
- public_key = private_key.public_key(): Та відповідний публічний ключ.
- В основній частині: shared_key = priv.exchange(peer_pub): Обчислюється спільний секретний ключ.

```
#  Обмін ключами (DH)
def diffie_hellman_key_exchange():
    parameters = dh.generate_parameters(generator=2, key_size=2048, backend=default_backend())
    private_key = parameters.generate_private_key()
    public_key = private_key.public_key()
    return private_key, public_key, parameters
```

Рисунок 4.3 – Обмін ключами Діффі-Хеллмана

- Виведення сесійних ключів (Використання HKDF в if __name__ == '__main__':): Після обміну ключами, з отриманого спільного секрету виводяться симетричні ключі для шифрування та автентифікації. TLS 1.3 використовує HKDF для цієї мети.
- derived_key = HKDF(...) .derive(shared_key): Зі спільного ключа shared_key за допомогою HKDF та геш-функції SHA256 виводиться ключ derived_key довжиною 32 байти (256 біт), який потім використовується для AES-256-GCM (рис. 4.4).

```python
# Основна логіка
if __name__ == '__main__':
    cert, key = generate_self_signed_cert("iot-device.local", "device", encryption_password)
    verify_certificate(os.path.join(OUTPUT_DIR, "device_cert.pem"))

    priv, pub, params = diffie_hellman_key_exchange()
    peer_priv = params.generate_private_key()
    peer_pub = peer_priv.public_key()
    shared_key = priv.exchange(peer_pub)

    derived_key = HKDF(
        algorithm=hashes.SHA256(),
        length=32,
        salt=None,
        info=b'handshake data',
        backend=default_backend()
    ).derive(shared_key)

    iv, ciphertext, tag = encrypt_aes_gcm(derived_key, b"Sensitive IoT Data")
    print("[AES-GCM]  Ciphertext:", ciphertext.hex())

    simulate_ipsec_channel()
```

Рисунок 4.4 – Основна логіка методу

- Симетричне шифрування (Функція encrypt_aes_gcm та використання в if __name__ == '__main__':): Після встановлення сесійного ключа, дані шифруються за допомогою симетричного алгоритму, такого як AES-256 в режимі GCM.
- iv, ciphertext, tag = encrypt_aes_gcm(derived_key, b"Sensitive IoT Data"): Шифрування даних b"Sensitive IoT Data" ключем derived_key.
- Таким чином, хоча повний TLS не реалізовано, код демонструє побудову захищеного каналу з використанням криптографічних примітивів, аналогічних тим, що лежать в основі TLS, включаючи шифрування AES-256 (через derived_key довжиною 32 байти) в режимі GCM (рис. 4.4).

Загальні фактори, що впливають на продуктивність:

- Довжина ключа: AES-256 зазвичай на 20-40% повільніший за AES-128 через більшу кількість раундів (14 проти 10). Це підтверджується бенчмарками на ESP32, де, наприклад, апаратний AES128-GCM (1.808 МБ/с) трохи швидший за AES256-GCM (1.773 МБ/с).
- Розмір даних: Для дуже малих обсягів даних накладні витрати на ініціалізацію шифру та обробку IV/nonce можуть бути значними.
- Оптимізація бібліотеки та компілятора: Якість реалізації криптографічної бібліотеки та налаштування компілятора мають велике значення для програмних реалізацій.

• Архітектура мікроконтролера: Наявність специфічних інструкцій (як ARMv8 Crypto Extensions), розмір кешу, швидкість доступу до пам'яті – все це впливає на продуктивність.

Вибір режиму шифрування та довжини ключа впливає не лише на безпеку та швидкість, але й на загальне енергоспоживання пристрою. Це особливо критично для IoT-пристроїв, що працюють від батарей. Апаратне прискорення стає не просто перевагою, а часто необхідністю для досягнення прийнятного балансу між безпекою, продуктивністю та тривалістю автономної роботи. При розробці для IoT, особливо для пристроїв з автономним живленням, оцінка продуктивності повинна включати не лише швидкість (наприклад, МБ/с), але й енергоспоживання на одиницю оброблених даних (наприклад, мкДж/байт). Це може суттєво вплинути на вибір конкретного мікроконтролера (з наявністю апаратного AES чи без нього) та на вибір режиму шифрування. Наприклад, хоча GCM може мати трохи більші обчислювальні накладні витрати порівняно з CTR через GHASH, відсутність потреби в окремому MAC може зробити його більш енергоефективним у загальному підсумку порівняно з CBC+HMAC [13, 84, 87, 88].

4.2. Реалізація цифрового підпису для захисту даних

Алгоритм ECDSA використовується у функції generate_self_signed_cert для підписання сертифіката.

Функція generate_self_signed_cert:

• key = ec.generate_private_key(ec.SECP384R1(), default_backend()): Генерується пара ключів на еліптичних кривих. ec.SECP384R1() вказує на використання конкретної еліптичної кривої P-384, яка є стандартом для ECDSA. Змінна key містить приватний ключ.

• cert = cert.public_key(key.public_key()): Публічний ключ, що відповідає приватному ключу key, додається до сертифіката.

• cert = cert.sign(key, hashes.SHA256(), default_backend()): Сертифікат підписується за допомогою приватного ключа key та геш-функції SHA256. Ця

операція створює цифровий підпис для даних сертифіката, забезпечуючи їх цілісність та підтверджуючи, що сертифікат був виданий власником відповідного приватного ключа. Це є прямою реалізацією цифрового підпису ECDSA (рис. 4.1).

Підписування даних перед передачею в мережі:

Наданий код безпосередньо демонструє підписування даних сертифіката у функції generate_self_signed_cert (як описано вище). Це забезпечує автентичність та цілісність самого сертифіката.

Однак, у коді немає окремої загальної функції для підписання довільних даних (корисного навантаження) перед їх передачею в мережу. Основний блок (if __name__ == '__main__':) шифрує дані b"Sensitive IoT Data", але не підписує їх окремо цифровим підписом ECDSA перед шифруванням (рис. 4.4).

Механізм підпису, реалізований для сертифікатів, міг би бути адаптований для підписання будь-яких інших даних. Для цього потрібно було б:

1. Згенерувати геш від даних, що передаються.
2. Підписати цей геш приватним ключем ECDSA відправника.
3. Передати дані разом з підписом.
4. Отримувач міг би перевірити підпис за допомогою публічного ключа відправника.

Хоча функціонал підпису даних присутній (на рівні сертифікатів), його застосування для захисту даних, що передаються, не показано явно для корисного навантаження. Режим AES-GCM, що використовується, сам по собі забезпечує автентифікацію та цілісність шифротексту, що може зменшити потребу в окремому підписі для деяких сценаріїв, але цифровий підпис надає незаперечність (non-repudiation), яку AES-GCM не забезпечує.

4.3. Впровадження комбінованого механізму шифрування

Скрипт використовує AES-GCM (з ключем, довжина якого залежить від KDF, у даному випадку 256 біт).

Код не реалізує повні протоколи IPSec та TLS і не демонструє їх класичне комбінування (наприклад, тунелювання TLS через IPSec). Однак, він показує

використання криптографічних примітивів та концепцій, які є фундаментальними для обох протоколів і спрямовані на досягнення аналогічних цілей безпеки:

1. Симуляція IPSec-подібного каналу (Функція simulate_ipsec_channel) - ця функція імітує деякі аспекти IPSec:

- Обмін ключами Діффі-Хеллмана: private_key_a.exchange(public_key_b) для встановлення спільного секретного ключа. Це аналогічно фазі IKE (Internet Key Exchange) в IPSec.
- Виведення ключа шифрування: h = hmac.HMAC(shared_key_a, hashes.SHA256(), ...); aes_key = h.finalize()[:32]. Спільний секрет використовується з HMAC-SHA256 для генерації 256-бітного ключа AES. Це схоже на те, як з матеріалу ключа IKE виводяться ключі для протоколу ESP (Encapsulating Security Payload) в IPSec.
- Шифрування та автентифікація даних: ciphertext = aesgcm.encrypt(nonce, plaintext, aad). Використовується AES-GCM для шифрування даних (plaintext) та додаткових автентифікованих даних (aad), що забезпечує конфіденційність, цілісність та автентичність, подібно до ESP в режимі AEAD.

Ця функція демонструє створення захищеного каналу "точка-точка" з використанням симетричного шифрування, де ключ встановлюється через асиметричний обмін (рис 4.5).

```python
# Симуляція каналу IPSec-подібного типу
def simulate_ipsec_channel():
    parameters = dh.generate_parameters(generator=2, key_size=2048, backend=default_backend())

    private_key_a = parameters.generate_private_key()
    public_key_a = private_key_a.public_key()

    private_key_b = parameters.generate_private_key()
    public_key_b = private_key_b.public_key()

    shared_key_a = private_key_a.exchange(public_key_b)
    shared_key_b = private_key_b.exchange(public_key_a)

    assert shared_key_a == shared_key_b

    h = hmac.HMAC(shared_key_a, hashes.SHA256(), backend=default_backend())
    h.update(b"ipsec_simulation_key_derivation")
    aes_key = h.finalize()[:32]

    aesgcm = AESGCM(aes_key)
    nonce = os.urandom(12)
    plaintext = b"Hello IoT device, this is a secure IPSec-like channel."
    aad = b"ipsec_aad"
    ciphertext = aesgcm.encrypt(nonce, plaintext, aad)

    decrypted = aesgcm.decrypt(nonce, ciphertext, aad)
    print("[IPSec Симуляція] Розшифроване повідомлення:", decrypted.decode())
```

Рисунок 4.5 – Симуляція каналу IPSec

2. Елементи, характерні для TLS (в основній логіці if __name__ == '__main__':):

• Обмін ключами Діффі-Хеллмана та виведення ключа за допомогою HKDF: shared_key = priv.exchange(peer_pub) та derived_key = HKDF(...).derive(shared_key). Це дуже схоже на процес встановлення ключів у TLS 1.3.

• Шифрування AES-GCM: encrypt_aes_gcm(derived_key, b"Sensitive IoT Data"). Дані шифруються за допомогою AES-GCM, що є одним з основних шифрів у сучасних версіях TLS.

• Використання сертифікатів (хоча самопідписаних): generate_self_signed_cert створює сертифікати, які є невід'ємною частиною TLS для автентифікації.

Отже, замість прямого комбінування протоколів, код демонструє реалізацію та використання фундаментальних криптографічних блоків, які лежать в основі як IPSec (зокрема, IKE та ESP), так і TLS. Він показує, як ці примітиви (DH, KDF, AES-GCM, HMAC) можуть бути використані для встановлення захищених каналів зв'язку та шифрування даних, що забезпечує багаторівневий підхід до безпеки, концептуально поєднуючи ідеї з обох протоколів. simulate_ipsec_channel показує один підхід до побудови захищеного каналу, а логіка в main – інший, ближчий до сесії TLS. Обидва використовують AES-GCM для захисту даних.

Функція verify_certificate(cert_pem_path: str):

• with open(cert_pem_path, 'rb') as cert_file: cert = x509.load_pem_x509_certificate(cert_file.read(), backend=default_backend()): Завантажує PEM-кодований сертифікат X.509 з файлу.

• print(" Сертифікат дійсний до:", cert.not_valid_after_utc): Виводить дату закінчення терміну дії сертифіката.

• print("CN:", cert.subject.get_attributes_for_oid(NameOID.COMMON NAME) [0].value): Виводить значення атрибута CN із суб'єкта сертифіката.

Ця функція важлива для перевірки автентичності сторони, з якою встановлюється з'єднання, хоча в даному прикладі вона викликається для локально

згенерованого сертифіката. У реальних системах вона б використовувалася для перевірки сертифіката, отриманого від іншої сторони (4.6).

```python
# Перевірка сертифіката
def verify_certificate(cert_pem_path: str):
    try:
        with open(cert_pem_path, 'rb') as cert_file:
            cert = x509.load_pem_x509_certificate(cert_file.read(), backend=default_backend())
            print(" Сертифікат дійсний до:", cert.not_valid_after_utc)
            print(" CN:", cert.subject.get_attributes_for_oid(NameOID.COMMON_NAME)[0].value)
            return cert
    except Exception as e:
        print("   Помилка перевірки сертифіката:", e)
        return None
```

Рисунок 4.6 – Перевірка сертифіката

Основний блок if __name__ == '__main__' - цей блок демонструє послідовність дій для налаштування та тестування реалізованих криптографічних механізмів.

- cert, key = generate_self_signed_cert("iot-device.local", "device", encryption_password): Генерує сертифікат та ключ для "пристрою".
- verify_certificate(...): Перевіряє щойно згенерований сертифікат.
- priv, pub, params = diffie_hellman_key_exchange(): Генерує пару ключів DH для однієї сторони.
- peer_priv = params.generate_private_key(); peer_pub = peer_priv.public_key(): Генерує пару ключів DH для "іншої" сторони (для симуляції обміну).
- shared_key = priv.exchange(peer_pub): Обчислює спільний секрет.
- derived_key = HKDF(...): Виводить симетричний ключ з спільного секрету.
- iv, ciphertext, tag = encrypt_aes_gcm(...): Шифрує дані з використанням виведеного ключа.
- print("[AES-GCM] Ciphertext:", ciphertext.hex()): Виводить зашифровані дані у шістнадцятковому форматі.
- simulate_ipsec_channel(): Викликає функцію симуляції IPSec-подібного каналу (рис. 4.7).

```python
# Основна логіка
if __name__ == '__main__':
    cert, key = generate_self_signed_cert("iot-device.local", "device", encryption_password)
    verify_certificate(os.path.join(OUTPUT_DIR, "device_cert.pem"))

    priv, pub, params = diffie_hellman_key_exchange()
    peer_priv = params.generate_private_key()
    peer_pub = peer_priv.public_key()D
    shared_key = priv.exchange(peer_pub)

    derived_key = HKDF(
        algorithm=hashes.SHA256(),
        length=32,
        salt=None,
        info=b'handshake data',
        backend=default_backend()
    ).derive(shared_key)

    iv, ciphertext, tag = encrypt_aes_gcm(derived_key, b"Sensitive IoT Data")
    print("[AES-GCM]  Ciphertext:", ciphertext.hex())

    simulate_ipsec_channel()
```

Рисунок 4.7 – Основний блок if __name__ == '__main__':

Результат:

1. Створення директорії та файлів (не відображено у виводі, але відбулося):

• Код спочатку перевіряє наявність директорії certs у каталозі проєкту (C:\Users\d4mat\PycharmProjects\maga3\). Якщо її немає, вона створюється.

• Далі функція generate_self_signed_cert генерує:

• Приватний ключ ECDSA (на основі кривої SECP384R1).

• Самопідписаний сертифікат X.509 з Common Name (CN) iot-device.local, терміном дії 1 рік.

• Зберігає сертифікат у файл certs/device_cert.pem (рис. 4.8).

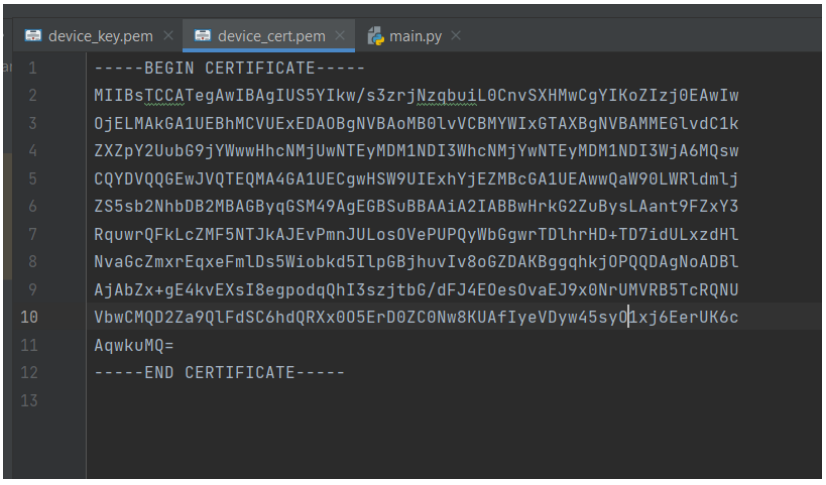

Рисунок 4.8 – Генерація ключа, сертифікату у файл .pem

• Зберігає приватний ключ (зашифрований паролем b"StrongPassword123") у файл certs/device_key.pem (рис. 4.9).

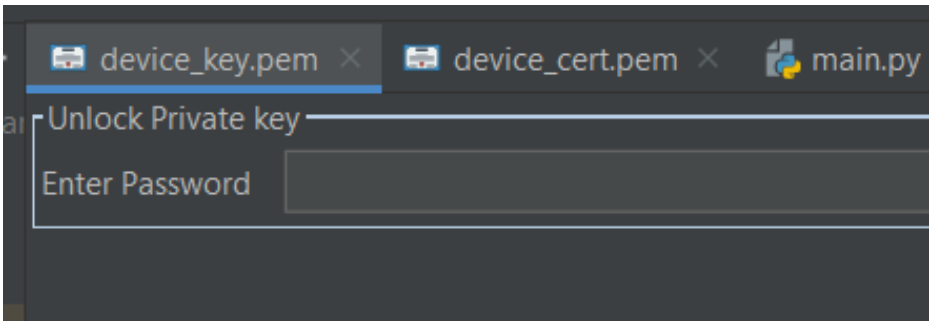

Рисунок 4.9 – Розшифрування приватного ключа

Algorithm: ECDSA (secp384r1):

- Це означає, що файл містить ключ, призначений для використання з алгоритмом цифрового підпису на еліптичних кривих
- secp384r1 – це назва конкретної еліптичної кривої, яка використовується. Це стандартизована крива, що забезпечує 384-бітний рівень безпеки. Вибір цієї кривої було зроблено у Python-коді при генерації ключа: key = ec.generate_private_key(ec.SECP384R1(), default_backend()).

Key Size: 384 bits

- Це розмір ключа, що відповідає обраній еліптичній кривій secp384r1. Для криптографії на еліптичних кривих розмір ключа прямо пов'язаний з кривою і визначає рівень безпеки. 384 біти вважається дуже надійним рівнем безпеки для ECDSA.

Format: PKCS#8

- Це вказує на стандарт формату, в якому збережено приватний ключ. PKCS#8 (Public-Key Cryptography Standards #8) – це стандартний синтаксис для зберігання приватних ключів. Він може містити як незашифровані, так і зашифровані приватні ключі (рис. 4.10).

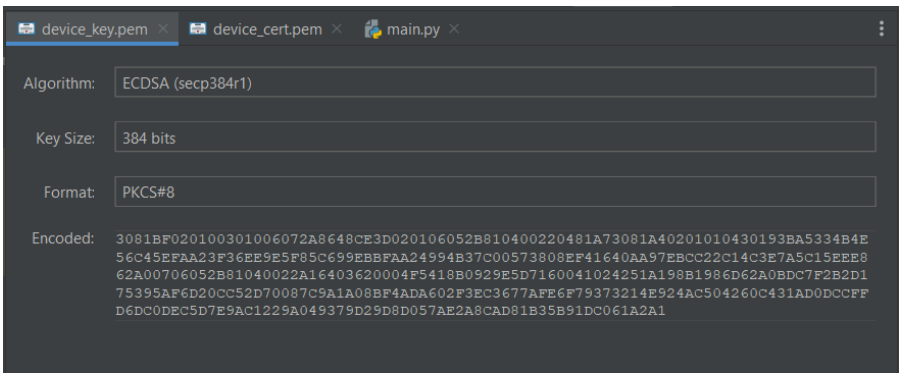

Рисунок 4.10 – Отриманий приватний ключ

У коді при збереженні ключа було вказано:

format=serialization.PrivateFormat.PKCS8.

Encoded:

• Шістнадцятковим представленням бінарних даних ключа у форматі ASN.1 DER (Distinguished Encoding Rules), який лежить в основі PKCS#8. Формат PEM – це, по суті, Base64-кодовані дані DER, обрамлені спеціальними заголовком та завершенням.

Що міститься у файлі certs/device_key.pem на основі коду та зображення:

• Файл certs/device_key.pem містить приватний ключ ECDSA. Цей ключ:

1. Згенерований для використання з еліптичною кривою secp384r1.
2. Має ефективний розмір 384 біти.
3. Збережений у стандартному форматі PKCS#8.
4. Зашифрований паролем, який було вказано у Python-коді (encryption_password = b"StrongPassword123"), оскільки при збереженні використовувався encryption_algorithm=serialization.BestAvailableEncryption(password). Це означає, що для використання цього ключа (наприклад, для підписання даних або розшифрування) спочатку потрібно буде його розшифрувати за допомогою цього ж пароля.
5. Представлений у текстовому форматі PEM, який є Base64-кодуванням бінарної структури ключа (DER/ASN.1).

Опис результату консолі:

1. Сертифікат дійсний до:

• Цей рядок виводиться функцією verify_certificate після успішного завантаження та аналізу файлу certs/device_cert.pem.

• Він показує дату та час закінчення терміну дії згенерованого сертифіката в форматі UTC. Оскільки сертифікат генерується на 365 днів, а поточна дата (якщо припустити, що код запускався приблизно 12 травня 2025 року) – 12 травня 2025 року, то дата закінчення 12 травня 2026 року є коректною. Час 03:54:27+00:00 – це час генерації сертифіката (в UTC) плюс 365 днів.

2. CN: iot-device.local

• Також виводиться функцією verify_certificate.

• Показує Common Name (Загальне Ім'я) суб'єкта сертифіката, яке було встановлено як "iot-device.local" при виклику generate_self_signed_cert в основній частині скрипта.

3. [AES-GCM] Ciphertext:

be72fa3ebcf589ee430e78cbd280836a3fd0c1bc64badd543035ee184c2f15680981

• Цей рядок виводиться з основного блоку (if __name__ == '__main__':) після виконання таких кроків:

1. Було виконано обмін ключами Діффі-Хеллмана (diffie_hellman_key_exchange та подальші операції) для отримання спільного секретного ключа (shared_key).

2. З цього спільного ключа за допомогою функції виведення ключів HKDF (з SHA256) було згенеровано 32-байтовий симетричний ключ derived_key (для AES-256).

3. Дані b"Sensitive IoT Data" було зашифровано за допомогою цього derived_key та алгоритму AES в режимі GCM (функція encrypt_aes_gcm).

- Виведений рядок – це шістнадцяткове представлення отриманого шифротексту (який також включає тег автентифікації, характерний для режиму GCM). Значення цього шифротексту буде різним при кожному запуску, оскільки воно залежить від випадково згенерованих ключів Діффі-Хеллмана та ініціалізаційного

вектора (IV) для AES-GCM.

4. [IPSec Симуляція] Розшифроване повідомлення: *Hello IoT device, this is a secure IPSec-like channel*.

• Цей рядок виводиться функцією simulate_ipsec_channel.

• Він свідчить про успішне виконання симуляції IPSec-подібного каналу:

4. Дві сторони ("A" і "B") виконали обмін ключами Діффі-Хеллмана та отримали однаковий спільний секретний ключ.

5. З цього спільного ключа за допомогою HMAC-SHA256 було виведено ключ aes_key для шифрування.

6. Повідомлення b"Hello IoT device, this is a secure IPSec-like channel." разом із додатковими автентифікованими даними (AAD) b"ipsec_aad" було зашифровано за допомогою aes_key та AES-GCM.

7. Отриманий шифротекст було успішно розшифровано тією ж стороною (для демонстрації), і розшифроване повідомлення збіглося з оригінальним.

• Виведення цього повідомлення підтверджує коректність роботи механізму обміну ключами, виведення ключа шифрування та самого процесу шифрування/дешифрування AES-GCM в рамках цієї симуляції (рис 4.11).

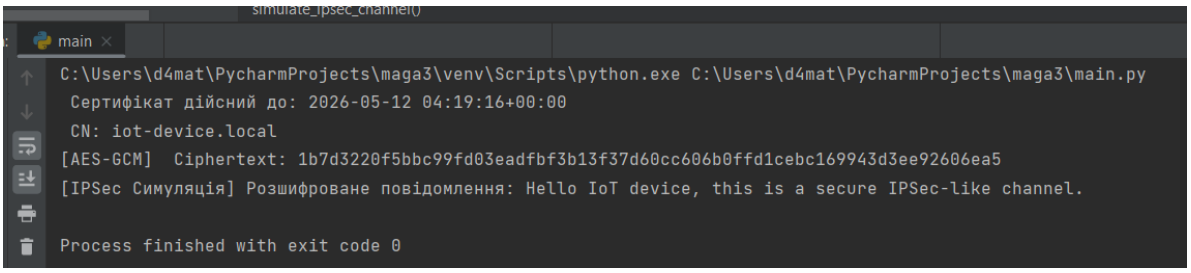

Рисунок 4.11 – Результат

Програмний метод:

• Створив директорію certs.

• Згенерував самопідписаний сертифікат X.509 та зашифрований приватний ключ, зберігши їх у файли.

• Прочитав та вивів основну інформацію зі згенерованого сертифіката.

- Продемонстрував захищений обмін даними, подібний до TLS: встановив спільний ключ через DH, вивів з нього ключ для AES-GCM за допомогою HKDF та зашифрував повідомлення.
- Просимулював встановлення захищеного каналу, подібного до IPSec: також через DH обмін, виведення ключа (цього разу через HMAC-SHA256) та успішне шифрування/дешифрування повідомлення за допомогою AES-GCM.

Результат показує, що криптографічні функції, реалізовані в коді, працюють правильно і виконують поставлені завдання щодо генерації ключів, сертифікатів, обміну ключами та шифрування/дешифрування даних.

Висновок до четвертого розділу

У четвертому розділі було практично реалізовано та проаналізовано ключові механізми шифрування та захисту даних, призначені для систем Інтернету речей. Зокрема, детально обґрунтовано вибір симетричного алгоритму AES-256 в режимі GCM як оптимального для забезпечення конфіденційності, цілісності та автентичності даних, та продемонстровано його програмну реалізацію. Також розглянуто застосування цифрового підпису на основі ECDSA для підтвердження автентичності сертифікатів і, концептуально, даних, та показано поєднання криптографічних примітивів для симуляції захищених каналів передачі, що імітують функціональність протоколів IPsec та елементи TLS.

ВИСНОВКИ

У дипломній роботі розв’язано актуальну задачу покращення захисту систем IoT шляхом удосконалення криптографічних протоколів для забезпечення безпеки обміну даними. У процесі виконання роботи були отримані наступні наукові та практичні результати:

1. Проаналізовано міжнародні та національні стандарти, що регулюють безпеку в IoT, зокрема ДСТУ та ISO/IEC стандарти, які визначають вимоги до шифрування, автентифікації та управління ключами.

2. Вивчено законодавчу базу України щодо криптографічного захисту інформації та сертифікації IoT-рішень, що працюють із конфіденційними даними.

3. Досліджено різні типи протоколів розподілу ключів для забезпечення конфіденційності та цілісності переданих даних в системах IoT, зокрема протоколи, що використовуються для обміну та автентифікації ключів.

4. Проаналізовано основи асиметричної криптографії, зокрема алгоритми RSA, Діффі-Хеллмана та криптографію еліптичних кривих (ECC), які є ефективними для обмежених обчислювальних ресурсів IoT.

5. Проведено аналіз криптографічних протоколів, таких як TLS, DTLS, IPsec, MQTT, SSH, для захищеного обміну даними в IoT, зокрема їх вразливостей, що можуть призводити до атак типу MITM та DoS.

6. Розглянуто проблеми енергетичної ефективності традиційних криптографічних методів для IoT, що потребують удосконалення існуючих підходів для забезпечення стабільної роботи пристроїв з обмеженими ресурсами.

7. Розроблено комплексний механізм автентифікації IoT-пристроїв, що базується на використанні самопідписаних цифрових сертифікатів X.509, з деталізацією процесів їх генерації (включаючи інтеграцію апаратних атрибутів для посилення безпеки), управління повним життєвим циклом та безпечного зберігання криптографічних даних з використанням формату PKCS#12.

8. Практично реалізовано програмний модуль мовою Python для

автоматизації створення самопідписаних сертифікатів X.509 та пакетів PKCS#12, а також обґрунтовано та описано механізми валідації цих сертифікатів в мережах IoT для забезпечення довіреної взаємодії та захисту від підробок.

9. Реалізовано та обґрунтовано застосування алгоритму симетричного шифрування AES-256 в режимі GCM, що забезпечує одночасно конфіденційність, цілісність та автентичність даних, як ефективного засобу захисту інформації в IoT-системах.

10. Продемонстровано використання цифрового підпису на основі ECDSA для забезпечення автентичності сертифікатів та концептуально реалізовано поєднання криптографічних примітивів (Діффі-Хеллман, HKDF, AES-GCM) для симуляції захищених каналів передачі даних, що імітують ключові аспекти безпеки протоколів IPsec та TLS.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. ДСТУ 7564:2014. Функція хешування. [Електронний ресурс] // usts.kiev.ua – 2014. – Режим доступу до ресурсу: https://usts.kiev.ua/wp-content/uploads/2020/07/dstu-7564-2014.pdf

2. ДСТУ 4145-2002. Інформаційні технології. Криптографічний захист інформації. [Електронний ресурс] // dbn.co.ua – 2002. – Режим доступу до ресурсу: https://dbn.co.ua/load/normativy/dstu/4145/5-1-0-1798

3. ДСТУ ISO/IEC 27001:2015. Інформаційні технології. Методи захисту. Системи управління інформаційною безпекою. [Електронний ресурс] // assistem.kiev.ua – 2015. – Режим доступу до ресурсу: https://www.assistem.kiev.ua/doc/dstu_ISO-IEC_27001_2015.pdf

4. Основні переваги сертифікації ISO/IEC 27001. [Електронний ресурс] // issp.training – 2023. – Режим доступу до ресурсу: https://www.issp.training/post/osnovni-perevahy-sertyfikatsiyi-iso-iec-27001

5. Аналіз сучасних підходів до інформаційної безпеки IoT. [Електронний ресурс] // conferenc-journal.its.kpi.ua – 2023. – Режим доступу до ресурсу: https://conferenc-journal.its.kpi.ua/article/download/129615/125106

6. Наказ Міністерства цифрової трансформації України № 2004-23 від 2023 р. [Електронний ресурс] // zakon.rada.gov.ua – 2023. – Режим доступу до ресурсу: https://zakon.rada.gov.ua/laws/show/z2004-23#Text

7. NIST Special Publication 800-177. Trustworthy Network Communications. [Електронний ресурс] // nist.gov – 2016. – Режим доступу до ресурсу: https://nvlpubs.nist.gov/nistpubs/specialpublications/nist.sp.800-177.pdf

8. NIST Special Publication 800-52 Revision 2. Guidelines for the Selection, Configuration, and Use of Transport Layer Security (TLS) Implementations. [Електронний ресурс] // nist.gov – 2019. – Режим доступу до ресурсу: https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-52r2.pdf

9. NIST Special Publication 800-57 Part 1 Revision 5. Recommendation for Key Management. [Електронний ресурс] // nist.gov – 2020. – Режим доступу до ресурсу: https://nvlpubs.nist.gov/nistpubs/specialpublications/nist.sp.800-57pt1r5.pdf

10. Указ Президента України № 1229/99 "Про заходи щодо розвитку національної системи технічного захисту інформації". [Електронний ресурс] // zakon.rada.gov.ua – 1999. – Режим доступу до ресурсу: https://zakon.rada.gov.ua/laws/show/1229/99

11. Указ Президента України № 505/98 "Про Концепцію технічного захисту інформації в Україні". [Електронний ресурс] // zakon.rada.gov.ua – 1998. – Режим доступу до ресурсу: https://zakon.rada.gov.ua/laws/show/505/98

12. Закон України № 3475-15 "Про захист інформації в інформаційно-комунікаційних системах". [Електронний ресурс] // zakon.rada.gov.ua – 2006. – Режим доступу до ресурсу: https://zakon.rada.gov.ua/laws/show/3475-15#Text

13. Onatsky A.V., Yona L.G. Asymmetric encryption methods / A.V. Onatsky, L. Yona // Module 2: Cryptographic methods of information protection in telecommunication systems and networks: Textbook. manual / N. V. Zakharchenko, Odesa: ONAS named after A. S. Popova. – 2010.

14. Царук Д., Фесенко А. Застосування асиметричної криптографії для безпечного документообігу / Д. Царук, А. Фесенко // XII Міжнародна наукова конференція «Інформація, комунікація, суспільство – 2023». – 2023.

15. Schneier B. Applied cryptography: protocols, algorithms, and source code in C (2nd ed.) / B. Schneier // John Wiley & Sons. – 2007. – P. 620-622.

16. RSA Laboratories. PKCS #1 v2.1: RSA Cryptography Standard / RSA Laboratories // [Електронний ресурс]. – 2002. – Режим доступу до ресурсу: http://www.rsa.com

17. Iotverif: Automatic Verification of SSL/TLS Certificate for IoT Applications [Електронний ресурс] // ResearchGate. – Режим доступу до ресурсу: https://www.researchgate.net/publication/338161909_Iotverif_Automatic_Verification_of_SSLTLS_Certificate_for_IoT_Applications

18. Efficient Elliptic Curve Diffie-Hellman Key Exchange for Resource-Constrained IoT Devices [Електронний ресурс] // ResearchGate. – Режим доступу до ресурсу: https://www.researchgate.net/publication/383994555_Efficient_Elliptic_Curve_Diffie-Hellman_Key_Exchange_for_Resource-Constrained_IoT_Devices

19. [7] An FPGA-based reconfigurable IPSec ESP core suitable for IoT applications [Електронний ресурс] // ResearchGate. – Режим доступу до ресурсу: https://www.researchgate.net/publication/311920873_An_FPGA_based_reconfigurable_IPSec_ESP_core_suitable_for_IoT_applications

20. Implementation of SSL/TLS Security with MQTT Protocol in IoT Environment [Електронний ресурс] // ResearchGate. – Режим доступу до ресурсу: https://www.researchgate.net/publication/372624929_Implementation_of_SSLTLS_Security_with_MQTT_Protocol_in_IoT_Environment

21. Technical Analysis of Comfort and Energy Consumption in Smart Buildings with Three Levels of Automation: Scheduling, Smart Sensors, and IoT [Електронний ресурс] // ResearchGate. – Режим доступу до ресурсу: https://www.researchgate.net/publication/387816712_Technical_Analysis_of_Comfort_and_Energy_Consumption_in_Smart_Buildings_with_Three_Levels_of_Automation_Scheduling_Smart_Sensors_and_IoT

22. Device Authentication using X.509 CA Certificates to Azure IoT Hub. [Електронний ресурс] // learn.microsoft.com. – Режим доступу до ресурсу: https://learn.microsoft.com/en-us/azure/iot-hub/authenticate-authorize-x509

23. X.509 Certificates / SSH Academy. [Електронний ресурс] // www.ssh.com. – Режим доступу до ресурсу: https://www.ssh.com/academy/pki/x.509-certificate

24. RFC 5280: Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile. [Електронний ресурс] // datatracker.ietf.org. – Режим доступу до ресурсу: https://datatracker.ietf.org/doc/html/rfc5280

25. The Dangers of Self-Signed Certificates / SecureW2 Blog. [Електронний ресурс] // www.securew2.com. – Режим доступу до ресурсу: https://www.securew2.com/blog/the-dangers-of-self-signed-certificates

26. Private CA for Kubernetes Better Than Self-Signed CAs / GlobalSign Blog. [Електронний ресурс] // www.globalsign.com. – Режим доступу до ресурсу: https://www.globalsign.com/en/blog/Private-ca-kubernetes-better-than-self-signed-cas

27. Self-Signed Certificates / Encryption Consulting Education Center. [Електронний ресурс] // www.encryptionconsulting.com. – Режим доступу до ресурсу: https://www.encryptionconsulting.com/education-center/self-signed-certificates/

28. What is PKCS12 File? / HeyCoach.in Blog. [Електронний ресурс] // blog.heycoach.in. – Режим доступу до ресурсу: https://blog.heycoach.in/pkcs12-2/

29. Create Self-Signed Certificates and Keys with OpenSSL / MariaDB Documentation. [Електронний ресурс] // mariadb.com. – Режим доступу до ресурсу: https://mariadb.com/docs/server/security/data-in-transit-encryption/create-self-signed-certificates-keys-openssl/

30. Generation of a Self Signed Certificate / GitHub Gist taoyuan. [Електронний ресурс] // gist.github.com. – Режим доступу до ресурсу: https://gist.github.com/taoyuan/39d9bc24bafc8cc45663683eae36eb1a

31. FIPS 140-3 changes impact PKCS#12 / Red Hat Blog. [Електронний ресурс] // www.redhat.com. – Режим доступу до ресурсу: https://www.redhat.com/en/blog/fips-140-3-changes-pkcs-12

32. Understanding .PFX Files and Their Constituents / TrustZone Knowledge Base. [Електронний ресурс] // trustzone.com. – Режим доступу до ресурсу: https://trustzone.com/knowledge-base/understanding-pfx-files-and-their-constituents/

33. X.509 Certificate Management / Sectigo Resource Library. [Електронний ресурс] // www.sectigo.com. – Режим доступу до ресурсу: https://www.sectigo.com/resource-library/x509-certificate-management

34. Certificate Lifecycle Management is Critical to SSL Management / Keyfactor Blog. [Електронний ресурс] // www.keyfactor.com. – Режим доступу до ресурсу: https://www.keyfactor.com/blog/certificate-lifecycle-management-is-critical-to-ssl-management/

35. IoT PKI & CLM: Identity Security for the Internet of Things / Accutive Security. [Електронний ресурс] // accutivesecurity.com. – Режим доступу до ресурсу: https://accutivesecurity.com/iot-pki-clm-identity-security/

36. Provisioning devices in AWS IoT / AWS IoT Developer Guide. [Електронний ресурс] // docs.aws.amazon.com. – Режим доступу до ресурсу: https://docs.aws.amazon.com/iot/latest/developerguide/iot-provision.html

37. How AWS IoT services address NIST 800-183 (Networks of Things) / AWS Public Sector Blog. [Електронний ресурс] // aws.amazon.com. – Режим доступу до ресурсу: https://aws.amazon.com/blogs/publicsector/how-aws-iot-services-address-nist-800-183-networks-of-things/

38. Pros and cons of implementing custom certificate provisioning for IoT devices / Stack Exchange Security. [Електронний ресурс] // security.stackexchange.com. – Режим доступу до ресурсу: https://security.stackexchange.com/questions/280496/pros-and-cons-of-implementing-custom-certificate-provisioning-for-iot-devices

39. Securing Web Transactions: TLS Server Certificate Management / NIST SP 1800-16. [Електронний ресурс] // nvlpubs.nist.gov (via Google Viewer). – Режим доступу до ресурсу: https://docs.google.com/viewer?docex=1&url=https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.1800-16.pdf

40. IoT Device Cybersecurity Capability Core Baseline / NIST SP 800-213A. [Електронний ресурс] // nvlpubs.nist.gov (via Google Viewer). – Режим доступу до ресурсу: https://docs.google.com/viewer?docex=1&url=https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-213A.pdf

41. How to manage IoT device certificate rotation using AWS IoT / AWS IoT Blog. [Електронний ресурс] // aws.amazon.com. – Режим доступу до ресурсу: https://aws.amazon.com/blogs/iot/how-to-manage-iot-device-certificate-rotation-using-aws-iot/

42. ACME for IoT: Extensions to ACME for Constrained Environments / IETF Draft. [Електронний ресурс] // www.ietf.org. – Режим доступу до ресурсу: https://www.ietf.org/id/draft-sweet-iot-acme-07.html

43. X.509 Certificate / JumpCloud Blog. [Електронний ресурс] // jumpcloud.com. – Режим доступу до ресурсу: https://jumpcloud.com/blog/x-509-certificate

44. Example of generating a self-signed certificate and RSA private key / GitHub Gist lykinsbd. [Електронний ресурс] // gist.github.com. – Режим доступу до ресурсу: https://gist.github.com/lykinsbd/588462f8f37b846c605c8dee477245c5

45. Creating & Signing X.509 Certificates / Cryptography.io Documentation. [Електронний ресурс] // cryptography.io. – Режим доступу до ресурсу: https://cryptography.io/en/latest/x509/tutorial/

46. Top 10 Python Libraries for Cybersecurity / GeeksforGeeks. [Електронний ресурс] // www.geeksforgeeks.org. – Режим доступу до ресурсу: https://www.geeksforgeeks.org/top-10-python-libraries-for-cybersecurity/

47. Creating X.509 Certificates for Production / Content Authenticity Initiative Documentation. [Електронний ресурс] // opensource.contentauthenticity.org. – Режим доступу до ресурсу: https://opensource.contentauthenticity.org/docs/signing/prod-cert/

48. Securing your Python codebase: Best practices for developers / QwietAI Blog. [Електронний ресурс] // qwiet.ai. – Режим доступу до ресурсу: https://qwiet.ai/securing-your-python-codebase-best-practices-for-developers/

49. Python User Input / DataCamp Tutorial. [Електронний ресурс] // www.datacamp.com. – Режим доступу до ресурсу: https://www.datacamp.com/tutorial/python-user-input

50. Limitations of Cryptography.io library / Cryptography.io Documentation. [Електронний ресурс] // cryptography.io. – Режим доступу до ресурсу: https://cryptography.io/en/latest/limitations/

51. Limitations of Cryptography.io library (version 43.0.0) / Cryptography.io Documentation. [Електронний ресурс] // cryptography.io. – Режим доступу до ресурсу: https://cryptography.io/en/43.0.0/limitations/

52. How to Delete Object in Python / AccuWeb.Cloud Articles. [Електронний ресурс] // accuweb.cloud. – Режим доступу до ресурсу: https://accuweb.cloud/resource/articles/delete-object-in-python

53. Should passwords be cleared from memory? / Sjoerd Langkemper // Sjoerd Langkemper's Blog. – 2016. [Електронний ресурс] // www.sjoerdlangkemper.nl. – Режим доступу до ресурсу: https://www.sjoerdlangkemper.nl/2016/05/22/should-passwords-be-cleared-from-memory/

54. Remote IoT Device Secure File Transfer (SFTP/SCP) / SocketXP Blog. [Електронний ресурс] // www.socketxp.com. – Режим доступу до ресурсу: https://www.socketxp.com/iot/remote-iot-device-secure-file-transfer-sftp-scp/

55. Logging in Python / Mimo.org Glossary. [Електронний ресурс] // mimo.org. – Режим доступу до ресурсу: https://mimo.org/glossary/python/logging

56. Sensitive data appearing in Supabase debug logs / Reddit Discussion. [Електронний ресурс] // www.reddit.com. – Режим доступу до ресурсу: https://www.reddit.com/r/Supabase/comments/1hax6y2/sensitive_data_appearing_in_supabase_debug_logs/

57. Using your own certificate provider for AWS IoT device provisioning / AWS IoT Developer Guide. [Електронний ресурс] // docs.aws.amazon.com. – Режим доступу до ресурсу: https://docs.aws.amazon.com/iot/latest/developerguide/provisioning-cert-provider.html

58. How to manage Azure IoT Edge device certificates / Microsoft Azure Documentation. [Електронний ресурс] // learn.microsoft.com. – Режим доступу до ресурсу: https://learn.microsoft.com/en-us/azure/iot-edge/how-to-manage-device-certificates

59. Stages in a Certificate's Lifecycle / Encryption Consulting Education Center. [Електронний ресурс] // www.encryptionconsulting.com. – Режим доступу до ресурсу: https://www.encryptionconsulting.com/education-center/stages-in-a-certificates-lifecycle/

60. All About TLS Handshakes / GlobalSign Blog. [Електронний ресурс] // www.globalsign.com. – Режим доступу до ресурсу: https://www.globalsign.com/en/blog/sg/all-about-tls-handshakes

61. What happens in a TLS handshake / Cloudflare Learning. [Електронний ресурс] // www.cloudflare.com. – Режим доступу до ресурсу: https://www.cloudflare.com/learning/ssl/what-happens-in-a-tls-handshake/

62. X.509 Certificate: Common Name (CN) vs. Subject Alternative Names (SAN) / Baeldung Article. [Електронний ресурс] // www.baeldung.com. – Режим доступу до ресурсу: https://www.baeldung.com/linux/x-509-certificate-common-name-subject-alternative-names

63. Cryptography.io X.509 Reference (version 3.4.3) / Cryptography.io Documentation. [Електронний ресурс] // cryptography.io. – Режим доступу до ресурсу: https://cryptography.io/en/3.4.3/x509/reference.html

64. Python Certificate Verify Failed: Self-Signed Certificate in Certificate Chain / SSLInsights Blog. [Електронний ресурс] // sslinsights.com. – Режим доступу до ресурсу: https://sslinsights.com/python-certificate-verify-failed-self-signed-certificate-in-certificate-chain/

65. Adding Custom Data to X.509 SSL Certificates / Dustin Oprea // Dustin Oprea's Blog. – 2014. [Електронний ресурс] // dustinoprea.com. – Режим доступу до ресурсу: https://dustinoprea.com/2014/04/18/adding-custom-data-to-x-509-ssl-certificates/

66. How to create a CSR with a UPN SAN using Python Cryptography module / Stack Overflow. [Електронний ресурс] // stackoverflow.com. – Режим доступу до ресурсу: https://stackoverflow.com/questions/73003466/how-to-create-a-csr-with-a-upn-san-using-python-cryptography-module

67. X.509: OIDs in Public Key Infrastructure for Secure Communication / FasterCapital Article. [Електронний ресурс] // fastercapital.com. – Режим доступу до ресурсу: https://fastercapital.com/content/X-509--OIDs-in-Public-Key-Infrastructure-for-Secure-Communication.html

68. X.509 DN Attribute Encoding / Fraser Tweedale // Fraser Tweedale's Blog (Red Hat). – 2018. [Електронний ресурс] // frasertweedale.github.io. – Режим доступу до ресурсу: https://frasertweedale.github.io/blog-redhat/posts/2018-03-15-x509-dn-attribute-encoding.html

69. What is Device Provisioning in IoT / Zipit Wireless Blog. [Електронний ресурс] // www.zipitwireless.com. – Режим доступу до ресурсу: https://www.zipitwireless.com/zipit/what-is-device-provisioning-in-iot

70. Identity Provisioning Services: Provisioning in IoT / SealSQ. [Електронний ресурс] // www.sealsq.com. – Режим доступу до ресурсу: https://www.sealsq.com/identity-provisioning-services/provisioning-in-iot

71. CSR Attestation / IETF LAMPS Draft. [Електронний ресурс] // www.ietf.org. – Режим доступу до ресурсу: https://www.ietf.org/id/draft-ietf-lamps-csr-attestation-18.html

72. Asymmetric Key Serialization / Cryptography.io Documentation. [Електронний ресурс] // cryptography.io. – Режим доступу до ресурсу: https://cryptography.io/en/latest/hazmat/primitives/asymmetric/serialization/

73. Python Cryptography Asymmetric Key Storage / CryptoExamples.com. [Електронний ресурс] // www.cryptoexamples.com. – Режим доступу до ресурсу: https://www.cryptoexamples.com/python_cryptography_asymmetric_key_storage.html

74. Exception Handling in Cryptography / HeyCoach.in Blog. [Електронний ресурс] // blog.heycoach.in. – Режим доступу до ресурсу: https://blog.heycoach.in/exception-handling-in-cryptography/

75. OWASP Logging Cheat Sheet / OWASP Cheat Sheet Series. [Електронний ресурс] // cheatsheetseries.owasp.org. – Режим доступу до ресурсу: https://cheatsheetseries.owasp.org/cheatsheets/Logging_Cheat_Sheet.html

76. OWASP Input Validation Cheat Sheet / OWASP Cheat Sheet Series. [Електронний ресурс] // cheatsheetseries.owasp.org. – Режим доступу до ресурсу: https://cheatsheetseries.owasp.org/cheatsheets/Input_Validation_Cheat_Sheet.html

77. AES-256 Encryption / Kiteworks Risk & Compliance Glossary. [Електронний ресурс] // www.kiteworks.com. – Режим доступу до ресурсу: https://www.kiteworks.com/risk-compliance-glossary/aes-256-encryption/

78. What is AES Encryption? / NordLayer Blog. [Електронний ресурс] // nordlayer.com. – Режим доступу до ресурсу: https://nordlayer.com/blog/aes-encryption/

79. Security and Performance Evaluation of Lightweight Cryptographic Algorithms for IoT. [Електронний ресурс] // Applied Sciences (MDPI). – 2018. – Vol. 8, Iss. 2, Art. 24. – Режим доступу до ресурсу: https://www.mdpi.com/2410-387X/8/2/24

80. IoT Security Challenges & Solutions / Cavli Wireless Blog. [Електронний ресурс] // www.cavliwireless.com. – Режим доступу до ресурсу: https://www.cavliwireless.com/blog/nerdiest-of-things/iot-security-challenges-solutions

81. AES Encryption Modes: GCM vs CBC vs CTR / Haikel Fazzani's Blog. [Електронний ресурс] // www.haikel-fazzani.eu.org. – Режим доступу до ресурсу: https://www.haikel-fazzani.eu.org/blog/post/aes-encryption-modes-gcm-cbc-ctr

82. Authenticated Encryption with Associated Data (AEAD) / Cryptography.io Documentation. [Електронний ресурс] // cryptography.io. – Режим доступу до ресурсу: https://cryptography.io/en/latest/hazmat/primitives/aead/

83. PyCryptodome & Cryptography inequality with AES CFB in Python / Stack Overflow. [Електронний ресурс] // stackoverflow.com. – Режим доступу до ресурсу: https://stackoverflow.com/questions/59712935/pycryptodome-cryptography-inequality-with-aes-cfb-in-python

84. What's the difference between AES-CBC and AES-GCM? / Private Internet Access Knowledge Base. [Електронний ресурс] // https://www.google.com/search?q=helpdesk.privateinternetaccess.com. – Режим доступу до ресурсу: https://helpdesk.privateinternetaccess.com/kb/articles/what-s-the-difference-between-aes-cbc-and-aes-gcm

85. Security and Performance in IoT: A Balancing Act. [Електронний ресурс] // SciSpace. – Режим доступу до ресурсу: https://scispace.com/pdf/security-and-performance-in-iot-a-balancing-act-2nbdw09oss.pdf

86. AES Galois Counter Mode (GCM) Cipher Suites for TLS / RFC 5288. [Електронний ресурс] // datatracker.ietf.org. – Режим доступу до ресурсу: https://datatracker.ietf.org/doc/html/rfc5288/.

ДОДАТОК А

ТЕЗИ НАУКОВИХ ДОПОВІДЕЙ

Дмитро Царук, Іван Пархоменко. Методи та підходи до розробки засобів захисту IoT відповідно до міжнародних стандартів. Міжнародна науково-практична конференція “Проблеми кібербезпеки інформаційно-комунікаційних систем” (PCSICS) 2025., Kyiv – С. 101-102.

ДОДАТОК Б

МЕТОД ЗАХИСТУ НА МОВІ PYTHON

```
import os
import datetime
from cryptography import x509
from cryptography.x509.oid import NameOID
from cryptography.hazmat.primitives import hashes, serialization, hmac
from cryptography.hazmat.primitives.asymmetric import ec, dh
from cryptography.hazmat.primitives.kdf.hkdf import HKDF
from cryptography.hazmat.backends import default_backend
from cryptography.hazmat.primitives.ciphers.aead import AESGCM


OUTPUT_DIR = "certs"
if not os.path.exists(OUTPUT_DIR):
    os.makedirs(OUTPUT_DIR)
encryption_password = b"StrongPassword123"


def generate_self_signed_cert(common_name: str, filename_prefix: str, password: bytes):
    key = ec.generate_private_key(ec.SECP384R1(), default_backend())
    subject = x509.Name([
        x509.NameAttribute(NameOID.COUNTRY_NAME, u"UA"),
        x509.NameAttribute(NameOID.ORGANIZATION_NAME, u"IoT Lab"),
        x509.NameAttribute(NameOID.COMMON_NAME, common_name),
    ])
    cert = x509.CertificateBuilder().subject_name(subject)
    cert = cert.issuer_name(subject)
    cert = cert.public_key(key.public_key())
    cert = cert.serial_number(x509.random_serial_number())
    cert = cert.not_valid_before(datetime.datetime.utcnow())
    cert = cert.not_valid_after(datetime.datetime.utcnow() + datetime.timedelta(days=365))
```

```
    cert = cert.sign(key, hashes.SHA256(), default_backend())

    cert_path = os.path.join(OUTPUT_DIR, f"{filename_prefix}_cert.pem")
    key_path = os.path.join(OUTPUT_DIR, f"{filename_prefix}_key.pem")

    with open(cert_path, "wb") as f:
        f.write(cert.public_bytes(serialization.Encoding.PEM))

    with open(key_path, "wb") as f:
        f.write(key.private_bytes(
            encoding=serialization.Encoding.PEM,
            format=serialization.PrivateFormat.PKCS8,
            encryption_algorithm=serialization.BestAvailableEncryption(password)
        ))

    return cert, key


def verify_certificate(cert_pem_path: str):
    try:
        with open(cert_pem_path, 'rb') as cert_file:
            cert = x509.load_pem_x509_certificate(cert_file.read(), backend=default_backend())
            print(" Сертифікат дійсний до:", cert.not_valid_after_utc)
            print(" CN:", cert.subject.get_attributes_for_oid(NameOID.COMMON_NAME)[0].value)
            return cert
    except Exception as e:
        print(" Помилка перевірки сертифіката:", e)
        return None


def diffie_hellman_key_exchange():
    parameters = dh.generate_parameters(generator=2, key_size=2048, backend=default_backend())
    private_key = parameters.generate_private_key()
    public_key = private_key.public_key()
    return private_key, public_key, parameters
```

```
def encrypt_aes_gcm(key: bytes, data: bytes):
    iv = os.urandom(12)
    aesgcm = AESGCM(key)
    ciphertext = aesgcm.encrypt(iv, data, None)
    return iv, ciphertext, b''


def simulate_ipsec_channel():
    parameters = dh.generate_parameters(generator=2, key_size=2048, backend=default_backend())

    private_key_a = parameters.generate_private_key()
    public_key_a = private_key_a.public_key()

    private_key_b = parameters.generate_private_key()
    public_key_b = private_key_b.public_key()

    shared_key_a = private_key_a.exchange(public_key_b)
    shared_key_b = private_key_b.exchange(public_key_a)

    assert shared_key_a == shared_key_b

    h = hmac.HMAC(shared_key_a, hashes.SHA256(), backend=default_backend())
    h.update(b"ipsec_simulation_key_derivation")
    aes_key = h.finalize()[:32]

    aesgcm = AESGCM(aes_key)
    nonce = os.urandom(12)
    plaintext = b"Hello IoT device, this is a secure IPSec-like channel."
    aad = b"ipsec_aad"
    ciphertext = aesgcm.encrypt(nonce, plaintext, aad)

    decrypted = aesgcm.decrypt(nonce, ciphertext, aad)
    print("[IPSec Симуляція] Розшифроване повідомлення:", decrypted.decode())
```

```
if __name__ == '__main__':
    cert, key = generate_self_signed_cert("iot-device.local", "device", encryption_password)
    verify_certificate(os.path.join(OUTPUT_DIR, "device_cert.pem"))

    priv, pub, params = diffie_hellman_key_exchange()
    peer_priv = params.generate_private_key()
    peer_pub = peer_priv.public_key()
    shared_key = priv.exchange(peer_pub)

    derived_key = HKDF(
        algorithm=hashes.SHA256(),
        length=32,
        salt=None,
        info=b'handshake data',
        backend=default_backend()
    ).derive(shared_key)

    iv, ciphertext, tag = encrypt_aes_gcm(derived_key, b"Sensitive IoT Data")
    print("[AES-GCM]  Ciphertext:", ciphertext.hex())

    simulate_ipsec_channel()
```