МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ДОНЕЦЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ ІМЕНІ ВАСИЛЯ СТУСА

МИХАЙЛОВСЬКИЙ СЕРГІЙ МИХАЙЛОВИЧ

Допускається до захисту:
В.о. завідувача кафедри
прикладної математики та
кібербезпеки, доктор філософії
з математики
_____ Луценко А. В.
« ___» _____ 20 __р.

РОЗРОБКА СПОСОБУ ЗАХИСТУ ІОТ СИСТЕМ ВІД БОТНЕТ АТАК

Спеціальність 105 Прикладна фізика та наноматеріали

Кваліфікаційна (магістерська) робота

Науковий керівник:
Л. В. Загоруйко,
канд. техн. наук., доцент,
доцент кафедри прикладної
математики та кібербезпеки
_____
(підпис)

Оцінка: _____/ ___ /_____
_____
(бали/за шкалою ЄКТС/за національною шкалою)

Голова ЕК: _____
(підпис)

Вінниця 2025

ЗМІСТ

ПЕРЕЛІК УМОВНИХ СКОРОЧЕНЬ

API — Application Programming Interface, програмний інтерфейс взаємодії між компонентами.

PY — Python, мова програмування

BLE — Bluetooth Low Energy, енергоефективний протокол бездротового зв’язку малого радіусу дії.

CoAP — Constrained Application Protocol, легкий протокол для IoT-пристроїв на основі UDP.

CSV — Comma-Separated Values, формат табличних даних.

DDoS — Distributed Denial of Service, розподілена атака на відмову в обслуговуванні.

DTLS — Datagram Transport Layer Security, протокол безпечної передачі даних поверх UDP.

ECC — Elliptic Curve Cryptography, криптографія на еліптичних кривих.

ETSI — European Telecommunications Standards Institute, Європейський інститут телекомунікаційних стандартів.

HTTP — HyperText Transfer Protocol, протокол передачі гіпертексту.

HTTPS — HyperText Transfer Protocol Secure, захищена версія HTTP.

IDS — Intrusion Detection System, система виявлення вторгнень.

IoT — Internet of Things, Інтернет речей.

IPFIX — Internet Protocol Flow Information Export, стандарт експорту потокової інформації.

ISO/IEC — International Organization for Standardization / International Electrotechnical Commission.

LPWAN — Low Power Wide Area Network, мережа великого радіусу дії з низьким енергоспоживанням.

MAC — Message Authentication Code, код автентичності повідомлення.

MITM — Man in the Middle, атака типу «людина посередині».

MQTT — Message Queuing Telemetry Transport, легковаговий протокол телеметрії.

NB-IoT — Narrowband Internet of Things, вузькосмуговий стільниковий протокол для IoT.

NFCAPD — NetFlow Capture Daemon, сервіс для збору потокових даних.

NIST — National Institute of Standards and Technology, Національний інститут стандартів і технологій США.

OTA — Over-The-Air, дистанційне оновлення програмного забезпечення.

PCA — Principal Component Analysis, метод головних компонент (якщо використовується у аналітиці).

PID — Process Identifier, ідентифікатор процесу в операційній системі.

RPI — Raspberry Pi, одноплатний комп’ютер, використаний для тестування.

SIMULATOR — Профіль роботи, у якому мережеві події генеруються штучно.

TCP — Transmission Control Protocol, протокол передавання даних із встановленням з’єднання.

TLS — Transport Layer Security, протокол захисту даних.

UDP — User Datagram Protocol, протокол передавання даних без встановлення з’єднання.

UML — Unified Modeling Language, мова моделювання систем.

ZigBee — бездротовий протокол для пристроїв «розумного дому».

Z-Wave — бездротовий протокол для автоматизації житла.

ВСТУП

Актуальність теми дослідження. Інтернет речей (IoT) демонструє експоненціальне зростання, інтегруючи мільярди фізичних пристроїв - від побутових сенсорів до критичної промислової інфраструктури - у глобальну мережу. Ця стрімка цифровізація, хоч і відкриває безпрецедентні можливості для автоматизації та збору даних, водночас створює масштабну та вразливу поверхню для кібератак.

Особливу загрозу для IoT-екосистеми становлять ботнет-мережі. IoT-пристрої є ідеальними цілями для зловмисників через низку системних проблем: обмежені обчислювальні ресурси, відсутність механізмів оновлення, використання слабких або незмінних паролів та незахищені канали зв'язку. Масштабні атаки, такі як Mirai, Satori та Mozi, продемонстрували, що скомпрометовані IoT-пристрої можуть бути об'єднані у потужні мережі для проведення руйнівних DDoS-атак, крадіжки даних та проникнення у корпоративні мережі.

Традиційні методи захисту, такі як антивіруси чи складні системи виявлення вторгнень (IDS), часто є нежиттєздатними для ресурсно-обмежених пристроїв. Водночас, новітні підходи на базі машинного навчання (ML) хоч і є ефективними, проте вимагають значних обчислювальних ресурсів для тренування та високої експертизи для впровадження.

Таким чином, виникає гостра науково-прикладна задача розробки легковагомого, гнучкого та масштабованого методу виявлення ботнет-активності, який міг би бути легко розгорнутий на периферійних пристроях (edge-пристроях) і водночас надавав би потужні інструменти для аналізу та симуляції загроз.

Мета та завдання наукової роботи. Метою даної магістерської роботи є підвищення рівня захищеності IoT-систем шляхом розробки, програмної реалізації та експериментального дослідження гібридної системи виявлення ботнет-активності, здатної функціонувати як в режимі симуляції загроз, так і в режимі моніторингу реального мережевого трафіку (NetFlow).

Для досягнення поставленої мети було визначено такі завдання:

1. Проаналізувати архітектуру IoT-систем, основні вектори загроз та детально дослідити механізми функціонування відомих ботнет-атак (Mirai, Satori, Hajime, Mozi, VPNFilter).

2. Виконати огляд та аналіз сучасних методів захисту IoT-систем, включаючи стандартні підходи, технології обману (honeypots) та методи виявлення аномалій на основі машинного навчання.
3. Розробити гнучку, двопрофільну архітектуру системи, що підтримує режим симуляції (simulator) та режим реального моніторингу (rpi).
4. Спроектувати та реалізувати програмні модулі системи, включаючи: модуль збору та нормалізації NetFlow (netflow_ingest.py), центральний UDP-сервер моніторингу (monitor.py) , гібридний аналітичний модуль (detector.py) та інтерактивну веб-панель візуалізації (dashboard.py).
5. Провести експериментальне дослідження розробленої системи: перевірити коректність роботи в режимі simulator та оцінити ефективність аналізу NetFlow-даних.
6. Проаналізувати отримані результати, оцінити продуктивність (швидкодію) та точність виявлення атак в обох режимах роботи .

Об'єкт дослідження - процеси функціонування та забезпечення безпеки мережевої інфраструктури Інтернету речей (IoT).

Предмет дослідження - методи, моделі та програмні засоби для виявлення ботнет-активності в IoT-мережах на основі гібридного аналізу, що поєднує сигнатурні, поведінкові (на основі ковзного вікна) та статистичні (на основі аналізу потоків NetFlow) підходи.

Гіпотеза. Дослідження ґрунтується на гіпотезі, що застосування гібридної, двопрофільної архітектури, яка нормалізує різні джерела даних (синтетичні UDP-повідомлення та реальні дані NetFlow) до єдиного формату та обробляє їх єдиним аналітичним ядром (detector.py) з подвійною логікою (ковзні вікна для подій та порогові значення для потоків), дозволить створити програмне рішення, що є водночас:

- Гнучким (для лабораторного тестування правил у режимі simulator).
- Практичним (для реального моніторингу на ресурсно-обмежених пристроях у режимі rpi).
- Високопродуктивним (із середнім часом обробки події значно меншим за 1 мілісекунду), забезпечуючи при цьому високу точність виявлення типових ботнет-атак (DDoS, сканування портів).

Методи дослідження. Для досягнення поставленої мети були використані такі методи:

- Теоретичні: методи аналізу та синтезу (для дослідження предметної області, загроз та існуючих рішень); метод системного аналізу (для проектування п’ятирівневої модульної архітектури).
- Емпіричні: метод програмного прототипування (для реалізації всіх модулів системи мовою Python); метод імітаційного моделювання (для створення та запуску емуляторів атак і легітимних пристроїв у профілі simulator); експериментальне дослідження (для тестування працездатності системи в обох профілях та аналізу її реакції на атаки); метод вимірювання продуктивності (для оцінки часу затримки класифікації повідомлень).
- Спеціалізовані: методи статистичного аналізу (для реалізації механізмів ковзного вікна та порогових значень для NetFlow); метод аналізу мережевих потоків (для інтеграції з nfdump та обробки даних NetFlow).

Теоретична значущість полягає у розробці та обґрунтуванні нової гнучкої, двопрофільної архітектури для систем виявлення вторгнень, яка формалізує метод нормалізації двох принципово різних джерел даних (синтетичних подій та реальних потоків NetFlow) до єдиного формату обробки. Також розвинуто гібридний метод детектування, що застосовує різні аналітичні моделі (ковзне вікно та пороговий аналіз потоків) в залежності від типу нормалізованого повідомлення .

Практична значущість роботи полягає у створенні готового до використання програмного комплексу, який має подвійне застосування:

1. Як навчально-дослідницька "пісочниця" (профіль simulator): надає дослідникам та студентам інструмент для моделювання IoT-атак та швидкого тестування нових правил виявлення без необхідності розгортання реальної інфраструктури.
2. Як легковаговий монітор безпеки (профіль rpi): система може бути розгорнута на недорогих одноплатних комп’ютерах (Raspberry Pi) для здійснення моніторингу реальної мережі (домашньої чи малого офісу) шляхом аналізу даних NetFlow з маршрутизатора.

Розроблений веб-інтерфейс (dashboard.py) надає наочний інструмент для візуалізації аналітики NetFlow та моніторингу загроз у реальному часі.

Інформаційною базою дослідження слугували фундаментальні та прикладні наукові праці, присвячені архітектурі та безпеці Інтернету речей.

Було проаналізовано технічні звіти та дослідження компаній з кібербезпеки щодо механізмів функціонування сучасних ботнетів, таких як Mirai, Satori, Mozi та VPNFilter. При аналізі методів захисту було опрацьовано публікації, присвячені як традиційним методам виявлення (сигнатурним, поведінковим), так і новітнім підходам, зокрема, технологіям обману (honeypots) та застосуванню машинного навчання. Також було враховано міжнародні рекомендації та стандарти безпеки (NIST, ETSI).

Наукова новизна отриманих результатів полягає у наступному:

1. **Розроблено метод нормалізації даних NetFlow**, який реалізовано в модулі netflow_ingest.py. Цей метод перетворює бінарні nfcapd файли (за допомогою nfdump) у текстові UDP-повідомлення, уніфіковані з форматом повідомлень емуляторів, що дозволяє обробляти обидва типи даних єдиним аналітичним ядром.
2. **Удосконалено метод гібридного детектування**, реалізований в модулі detector.py, який автоматично застосовує різні моделі аналізу до уніфікованих повідомлень: поведінковий аналіз на основі ковзного вікна (для подій simulator) та пороговий аналіз параметрів потоку (для подій rpi/NetFlow).

РОЗДІЛ І. Основні поняття IoT та ботнет-атак.

1.1. Архітектура та принципи роботи IoT-систем.

Інтернет речей (Internet of Things, IoT) - це концепція, яка передбачає об'єднання фізичних об'єктів у єдину мережу за допомогою вбудованих датчиків, контролерів, мережевих інтерфейсів і програмного забезпечення. [1] Метою IoT-систем є забезпечення автоматизованого збору, передавання, обробки та реагування на дані в режимі реального часу без прямої участі людини.

Архітектура IoT, загалом, будується на багаторівневій моделі, що складається з таких основних шарів [1]:

Рисунок 1.1 Архітектура IoT

1. Рівень сприйняття (Sensing Layer)

Цей рівень відповідає за фізичну взаємодію пристроїв з навколишнім середовищем. До його складу входять сенсори, RFID-модулі, камери, мікроконтролери тощо. Завданням рівня є збирання інформації про об'єкти або процеси - температура, вологість, тиск, переміщення тощо.

2. Мережевий рівень (Network Layer)

Мережевий рівень відповідає за передавання даних, зібраних сенсорами, до центрів обробки або хмарних платформ. Для цього використовуються протоколи бездротового зв'язку, такі як: Wi-Fi, ZigBee, LoRa, NB-IoT. або провідні мережі. Він також забезпечує маршрутизацію, ідентифікацію пристроїв та шифрування трафіку.

3. Рівень обробки (Data Processing Layer)

Цей рівень здійснює аналіз і зберігання даних, приймає рішення на основі отриманої інформації. Обробка може відбуватись на краєвих обчислювальних пристроях (edge/fog computing) або у хмарі (cloud computing). Також тут реалізуються елементи штучного інтелекту, аналітики та системи прийняття рішень.

4. Рівень застосування (Application Layer)

Забезпечує взаємодію з кінцевим користувачем, відображення інформації, керування пристроями, налаштування параметрів тощо. Приклади: «розумний дім», «індустрія 4.0», «розумне місто», аграрні або медичні системи моніторингу.

З розвитком IoT-систем кожен із базових рівнів, описаних у класичній архітектурі, почав суттєво еволюціонувати та доповнюватися новими функціями, що зумовлено зростанням вимог до масштабованості, швидкості обробки даних, енергоефективності та безпеки. Рівень сприйняття сьогодні охоплює не лише окремі сенсори чи RFID-модулі, а цілу екосистему мікроконтролерів, інтегрованих сенсорних платформ та спеціалізованих модулів, здатних працювати в різних середовищах і виконувати попередню обробку даних безпосередньо «на місці». Все частіше пристрої цього рівня обладнуються енергоефективними мікропроцесорами з можливістю локальної фільтрації та стискання даних. Це дозволяє зменшувати навантаження на мережевий рівень та підвищувати автономність системи, особливо в умовах віддалених або нестабільних середовищ, де канали зв'язку є обмеженими [1].

На мережевому рівні також відбулися суттєві зміни. Якщо раніше ключовою функцією цього шару була передача даних від сенсорів до центрів обробки, то сьогодні мережевий рівень є складною інфраструктурою, що включає маршрутизацію, балансування навантаження, забезпечення захищеної комунікації та підтримку декількох паралельних протоколів. Особливого поширення набули протоколи з низьким енергоспоживанням, такі як LoRaWAN або NB-IoT, які дозволяють пристроям працювати роками на одному акумуляторі. За рахунок цього мережевий рівень перетворюється на багатокомпонентний шар з підтримкою гібридних моделей зв'язку - від високошвидкісних каналів Wi-Fi до високоенергоефективних, але повільніших LPWAN-рішень. Така різноманітність ускладнює організацію безпеки та вимагає додаткової стандартизації, особливо в аспектах автентифікації та шифрування трафіку [2].

Рівень обробки даних також зазнав трансформації. Його роль більше не обмежується зберіганням і аналізом інформації на хмарних серверах. Натомість поширюється практика розподіленої обробки, що передбачає використання проміжних вузлів і локальної аналітики на пристроях edge-класу. Завдяки цьому зменшуються затримки, а системи стають більш стійкими до збоїв у мережі. На цьому рівні активно використовуються методи

машинного навчання, алгоритми прогнозування та механізми автоматизації, що дозволяють не лише реагувати на події, а й передбачати аномалії або потенційні загрози [5]. Важливо підкреслити, що рівень обробки перетворився на багатофункціональний центр інтелектуальної аналітики, який є критичним компонентом у системах, пов'язаних із безпекою, промисловим контролем або медичними сенсорами.

На рівні застосування спостерігається тенденція до уніфікації інтерфейсів та розширення функціональності. Сучасні IoT-платформи дозволяють не лише переглядати телеметрію в реальному часі, але й будувати складні сценарії автоматизації, налаштовувати залежності між пристроями, застосовувати політики безпеки та інтегрувати сервіси штучного інтелекту [2]. Це робить рівень застосування важливою складовою загальної архітектури, адже саме він формує логіку взаємодії між користувачем та фізичними об'єктами, забезпечуючи зручність керування та можливість централізованого контролю над складними системами.

Узагальнюючи, можна відзначити, що класична архітектура IoT зберігає свою актуальність, проте кожен її компонент значно ускладнився та розширився. IoT перестав бути простою системою сенсорів і контролерів - він перетворився на багаторівневу інтелектуальну інфраструктуру, яка поєднує апаратні, мережеві та програмні рішення, що працюють у тісній взаємодії [6]. Саме ця складність і багатокомпонентність робить питання безпеки IoT одним із найбільш актуальних, адже вразливість на будь-якому рівні може мати каскадний ефект на всю систему.

1.2. Основні загрози для IoT-пристроїв.

Стрімке поширення IoT-пристроїв у побуті, промисловості та критичній інфраструктурі призвело до появи нової категорії кіберзагроз, зумовлених унікальною природою таких систем. Оскільки IoT-пристрої поєднують у собі апаратні компоненти, програмну логіку та мережеві модулі, їх безпека залежить від узгодженості всіх цих елементів. У більшості випадків виробники зосереджуються на функціональності та низькій вартості кінцевого продукту, тоді як питання захисту відходять на другий план. Це створює комплекс середовищних і технічних ризиків, які формують характерні вектори атак.

Однією з найбільш поширених проблем є неналежна автентифікація. Значна частина IoT-пристроїв використовує заводські паролі або мінімально

необхідні механізми контролю доступу. Через це пристрої легко піддаються атакам перебору або скануванню на наявність відкритих сервісів, які працюють за замовчуванням. Найчастіше зловмисники використовують відкриті Telnet- або SSH-порти, що історично стали основою для формування великих ботнетів на кшталт Mirai. Такі атаки є можливими через поєднання кількох факторів [3]:

- використання однакових облікових записів у всіх пристроях певної серії;
- відсутність політики примусової зміни пароля після першого запуску;
- наявність прихованих системних акаунтів, які користувач не може змінити.

Не менш критичною є відсутність шифрування даних або застосування застарілих криптографічних алгоритмів. Багато пристроїв передають телеметрію та команди управління у незашифрованому вигляді, що робить можливими атаки типу «людина посередині». Для систем, у яких дані безпосередньо впливають на фізичні процеси, компрометація цілісності комунікації може мати серйозні наслідки. Навіть реалізовані протоколи шифрування інколи використовуються неправильно: ключі зберігаються у відкритому вигляді в прошивці, сертифікати є простроченими, або пристрої приймають будь-які сертифікати без перевірки їх справжності. Такі недоліки створюють умови для:

- перехоплення та аналізу трафіку;
- модифікації даних у реальному часі;
- підміни команд управління.

Суттєвим джерелом загроз є уразливості у прошивках та бібліотеках, які використовуються виробниками. Через фрагментованість IoT-ландшафту багато пристроїв працюють на основі застарілих або модифікованих версій стеків TCP/IP, що можуть містити критичні помилки безпеки. Відсутність регулярних і безпечних оновлень робить такі ризики довготривалими. Окремі пристрої не мають жодного механізму оновлення, інші здійснюють оновлення без перевірки цифрового підпису. У результаті:

- поширені публічні експлойти залишаються актуальними роками;
- можливе розгортання шкідливих прошивок;

- уразливості в одній бібліотеці можуть впливати на мільйони пристроїв (як у випадку з Ripple20 або AMNESIA:33) [3].

Фізичний доступ також створює широкі можливості для атак. IoT-пристрої часто розміщені у відкритих приміщеннях, на вулиці або у технічних зонах без додаткового захисту. Зловмисник може підключитись до сервісних інтерфейсів на зразок UART або JTAG, отримати дамп пам'яті, змінити логіку роботи пристрою чи викрасти ключі шифрування. У випадку побутових пристроїв додатковою проблемою є те, що вони часто не мають апаратних механізмів цілісності або захисту від несанкціонованого втручання.

Ще одним суттєвим фактором є відсутність єдиних стандартів безпеки та низький рівень загальної узгодженості між пристроями різних виробників. Навіть існуючі міжнародні стандарти не є обов'язковими для дотримання, а це призводить до появи в одній інфраструктурі пристроїв з різним рівнем захисту, різною якістю реалізації криптографії та різними протоколами обміну даними. У таких гетерогенних середовищах уразливість одного елемента може стати точкою входу для атаки на всю систему.

Усі зазначені фактори сприяють формуванню умов для створення великих IoT-ботнетів, які активно використовуються для DDoS-атак, несанкціонованого стеження, збору даних або маніпуляції потоками інформації. Компрометація одного пристрою часто веде до можливості подальшого бокового переміщення мережею, що розширює наслідки атаки і робить IoT-системи надзвичайно привабливою ціллю для зловмисників [4]. Таким чином, проблематика загроз для IoT-пристроїв охоплює програмні, апаратні, мережеві та організаційні аспекти. Комплексна природа цих ризиків вимагає багаторівневого підходу до їхнього аналізу та впровадження відповідних механізмів захисту.

1.3. Поняття ботнету: визначення, принцип дії.

Ботнет - це мережа скомпрометованих пристроїв, які дистанційно керуються зловмисником через канали командного та контрольного зв'язку (C&C) [7]. Кожен пристрій у ботнеті, відомий як «бот», виконує команди, отримані від C&C-сервера або через децентралізовані механізми, такі як peer-to-peer (P2P) мережі.

Основна мета створення ботнетів полягає в здійсненні різноманітних шкідливих дій, включаючи розподілені атаки типу відмови в обслуговуванні

(DDoS), розсилання спаму, крадіжку даних, криптомайнінг та інші форми кіберзлочинності [8]. Зокрема, в контексті Інтернету речей (IoT), ботнети становлять особливу загрозу через велику кількість пристроїв з обмеженими обчислювальними ресурсами та часто недостатнім рівнем безпеки.

Архітектура ботнетів може бути централізованою або децентралізованою. У централізованій моделі всі боти підключаються до одного або кількох C&C-серверів, що полегшує управління, але робить мережу вразливою до виявлення та знищення. У децентралізованій моделі, зокрема P2P-ботнетах, кожен бот може виконувати функції C&C, що ускладнює виявлення та знищення мережі [14].

Процес створення ботнету зазвичай починається з виявлення вразливих пристроїв, які потім заражаються шкідливим програмним забезпеченням. Після зараження пристрій встановлює зв'язок з C&C-сервером або іншими ботами в мережі, отримує команди та виконує їх. Ці команди можуть включати участь у DDoS-атаках, розсилання спаму, крадіжку даних або інші шкідливі дії [10].

Відомим прикладом є ботнет Mirai, який у 2016 році використовував вразливі IoT-пристрої з типовими паролями для здійснення масштабних DDoS-атак, зокрема на компанію Dyn, що призвело до тимчасової недоступності багатьох популярних вебсайтів. Цей ботнет сканував Інтернет у пошуках пристроїв з відкритими Telnet-портами, підбирав логіни та паролі з вбудованого словника та, у разі успіху, долучав пристрій до своєї мережі [11].

Інші відомі ботнети, такі як Hajime, Mozi, Satori та VPNFilter [12], демонструють еволюцію в напрямку використання більш складних механізмів зараження, шифрування трафіку та децентралізованих структур, що ускладнює їх виявлення та знищення [15].

Нижче наведено таблицю, що узагальнює основні характеристики деяких відомих ботнетів:

Назва ботнету	Тип керування	Спосіб інфікування	Основне призначення	Особливості
Mirai	Централізований	Слабкі/заводські паролі (Telnet)	DDoS-атаки	Висока швидкість поширення, відкритий код сприяв появі численних варіацій
Hajime	Децентралізований	Brute-force, відкриті порти	Невідомо	Саморозповсюдження, P2P-архітектура, відсутність шкідливих дій
Mozi	Децентралізований	P2P + вразливості IoT-ос	DDoS, бекдори	Використовує цифрові підписи, зашифрований трафік
Satori	Централізований	Експлойти у прошивках пристроїв	DDoS, експлуатація	Продовження Mirai з новими векторами інфікування
VPNFilter	Гібридний	Через роутери та NAS-пристрої	Шпигунство, саботаж	Може фізично виводити пристрої з ладу, модульна структура

Таблиця 1.1. Основні типи ботнетів та їх характеристики

Наслідки створення та розгортання ботнет-мереж в IoT-середовищах можуть бути масштабними - від падіння сервісів до проникнення в критичну інфраструктуру. Ефективний захист потребує не лише контролю над вразливостями, але й механізмів виявлення бот-поведінки на ранніх стадіях [13].

1.4. Аналіз відомих ботнет-атак (Mirai, Satori, Hajime)

Аналіз конкретних прикладів ботнет-атак дозволяє краще зрозуміти, як вразливості в IoT-пристроях можуть бути використані для створення розподілених шкідливих мереж. Ботнети, що розглядаються у цьому підпункті, відрізняються за архітектурою, методами інфікування та рівнем складності реалізації. Проте всі вони свідчать про зростання загроз для інфраструктури Інтернету речей [49].

1. Ботнет Mirai

Ботнет Mirai був вперше зафіксований у 2016 році [15]. Його дія полягала у скануванні мережі на предмет пристроїв з відкритими Telnet-портами та використанні словника типових логінів і паролів. Після успішного

підбору облікових даних пристрої ставали частиною централізованої мережі, яка використовувалась для DDoS-атак. Найвідоміший інцидент із Mirai - атака на DNS-провайдера Dyn, в результаті чого були тимчасово недоступні сервіси Twitter, Netflix, Spotify та інші. Атака виявила масовість проблем із базовим захистом IoT-пристроїв.

2. Ботнет Satori

Satori з'явився у 2017 році як варіація на базі коду Mirai, але з вдосконаленим механізмом інфікування. Замість підбору паролів він експлуатував вразливості у прошивках обладнання зокрема, в маршрутизаторах Huawei та пристроях з Realtek SDK [16]. Завдяки цьому зараження могло відбуватися миттєво, без участі користувача. Незважаючи на централізовану архітектуру, Satori мав високу швидкість поширення та ускладнене виявлення.

3. Ботнет Hajime

Ботнет Hajime, на відміну від інших, реалізує децентралізовану архітектуру, засновану на P2P-протоколах. Зараження пристроїв також здійснювалося через відкриті Telnet-порти з brute-force атакою. Особливістю Hajime є відсутність шкідливої активності - він не виконує DDoS-атак або шпигунських функцій. Дослідники вважають, що його завданням могло бути блокування інших ботнетів. Завдяки своїй структурі Hajime є стійким до централізованого знищення і привернув увагу наукової спільноти як потенційно «нейтральна» або навіть «захисна» мережа.

4. Ботнет Mozi

Mozi є прикладом сучасного ботнету з децентралізованою архітектурою, який з'явився у 2019 році. Його поширення ґрунтується на використанні слабких паролів і відомих вразливостей, а також механізму саморозповсюдження через DHT (distributed hash table). Важливою відмінністю Mozi є використання цифрових підписів, що дозволяє фільтрувати сторонні спроби змінити ботнет або взяти його під контроль. Mozi активно застосовувався для DDoS-атак і демонструє підвищений рівень складності у порівнянні з попередниками.

5. Ботнет VPNFilter

VPNFilter - складна багатоступенева шкідлива програма, виявлена у 2018 році. [14, 17] Вона орієнтована на маршрутизатори та NAS-пристрої.

Перший модуль відповідає за стійкість до перезавантаження, другий - за комунікацію з командним центром, а третій забезпечує гнучкий функціонал: шпигунство, перехоплення трафіку, активацію «kill switch» для виводу пристрою з ладу. Частина коду VPNFilter могла зберігатися в енергонезалежній пам'яті, що унеможливлювало його повне видалення звичайними засобами. Ботнет пов'язують з кібершпигунськими операціями на рівні держав.

Усі розглянуті приклади свідчать про поступову еволюцію ботнетів від простих централізованих мереж до високотехнологічних децентралізованих структур, здатних тривалий час залишатися непоміченими в мережевому середовищі. Зокрема, перехід від використання типових паролів до складних експлойтів і криптографічного захисту демонструє професіоналізацію атак. Це формує нові виклики для захисту IoT-систем, де важливо враховувати не лише уразливості окремих пристроїв, а й загальну архітектуру мережевої взаємодії

1.5. Протоколи IoT та їх уразливості

Архітектура IoT значною мірою визначається комунікаційними протоколами, які забезпечують взаємодію між пристроями, шлюзами та хмарними платформами. Різноманітність протоколів пояснюється потребою пристроїв працювати в умовах обмежених ресурсів, малого енергоспоживання, різних топологій мережі та різного призначення. Однак саме ця різнорідність часто призводить до появи вразливостей, оскільки частина протоколів створювалася в епоху, коли питання кібербезпеки вважалося другорядним, або ж їхня специфікація занадто залежить від правильної реалізації з боку виробника.

У цьому контексті ключовими є протоколи MQTT, CoAP, AMQP, ZigBee, Z-Wave, Bluetooth Low Energy, LoRaWAN та NB-IoT [47,48]. Вони охоплюють різні рівні взаємодії - від мережі та радіозв'язку до прикладного рівня. Кожен із них має власну модель передачі повідомлень, механізми автентифікації, захисту трафіку та структуру керування ключами. Саме тому їх доцільно порівняти у систематизованому вигляді, що дозволяє чіткіше показати характерні вразливості та можливі наслідки для безпеки всієї IoT-інфраструктури.

Наведена нижче таблиця демонструє основні особливості найпоширеніших IoT-протоколів, їхні ключові ризики та специфічні слабкі

місця. Вона відображає загальну картину проблем, які виникають під час експлуатації протоколів у реальних умовах.

Протокол	Призначення	Основні вразливості
MQTT	Обмін телеметрією через брокер	Відсутність TLS за замовчуванням; слабка автентифікація; відкриті топіки
CoAP	Легкий REST через UDP	Відсутній DTLS; відкриті порти; можливість DDoS
ZigBee	Мережі «розумного дому»	Глобальний мережевий ключ; незахищене приєднання
Z-Wave	Автоматизація житла	Стара S0-криптографія; атаки MITM
BLE	Короткий радіозв'язок	Вразливе парування; перехоплення ключів
LoRaWAN	Далекодіапазонний LPWAN	Статичні ключі (ABP); атаки повторення пакетів
NB-IoT	Стільниковий IoT-зв'язок	Уразливості інфраструктури оператора; слабка сегментація

Таблиця 1.2. Порівняння основних IoT-протоколів та їхніх вразливостей

Такі слабкі місця безпосередньо впливають на цілісність і безпеку IoT-інфраструктур. Для зловмисника вони створюють можливість перехоплювати телеметрію, підмінювати команди, модифікувати поведінку пристроїв або використовувати їх для масштабних атак. Саме тому безпека протоколів комунікації є одним із ключових напрямів досліджень у сфері IoT, а правильна конфігурація, шифрування та управління ключами стають критично важливими складовими побудови захищених систем.

1.6. Висновок до розділу 1

У першому розділі проведено комплексний аналіз сучасного стану Інтернету речей, який виявив, що стрімка інтеграція IoT-пристроїв у критичну інфраструктуру та побут значно випереджає розвиток механізмів їхнього захисту. Детальний розгляд архітектури IoT показав, що вразливості присутні на всіх рівнях: від сенсорів з обмеженими ресурсами, які не підтримують надійне шифрування, до гетерогенних мережевих протоколів (MQTT, CoAP, ZigBee), що часто функціонують без належної автентифікації .

Особливу увагу в розділі приділено аналізу ботнет-загроз. Дослідження еволюції шкідливого програмного забезпечення — від Mirai та Satori до

складніших Mozi та VPNFilter — дозволило виявити загрозливу тенденцію до професіоналізації атак. Сучасні ботнети відмовляються від простих схем перебору паролів на користь використання експлойтів, складних методів приховування трафіку та децентралізованої архітектури управління (P2P), що робить їх надзвичайно стійкими до традиційних методів блокування . Встановлено, що критичним фактором ризику є відсутність у більшості пристроїв механізмів безпечного оновлення (OTA) та використання застарілих програмних компонентів, що перетворює IoT-мережі на ідеальний плацдарм для організації масштабних DDoS-атак .

РОЗДІЛ ІІ. Аналіз сучасних методів захисту IoT-систем

2.1. Стандартні підходи до захисту IoT систем.

Зважаючи на відкритість середовища, в якому функціонують IoT-системи, їхній захист стає ключовим аспектом проектування та експлуатації. Враховуючи обмежені ресурси більшості IoT-пристроїв, класичні методи захисту інформаційних систем потребують адаптації до специфіки IoT-інфраструктури. У цьому підпункті розглядаються базові, або стандартні, підходи до забезпечення безпеки в таких системах [9].

1. Контроль автентифікації та доступу

Одним із фундаментальних принципів є забезпечення автентифікації користувачів і пристроїв. Найбільш поширеним рішенням є використання паролів або сертифікатів X.509. Проте традиційні методи (логін/пароль) є недостатньо ефективними для IoT, де часто застосовуються однакові паролі для великої кількості пристроїв. Тому набирають популярності більш надійні підходи - багатофакторна автентифікація (MFA), апаратні токени та аутентифікація на основі довіри між пристроями (*trust management*).

2. Шифрування даних

Захист інформації під час її передавання здійснюється шляхом використання криптографічних протоколів. У контексті IoT доцільним є застосування легковагових шифрувальних алгоритмів, таких як ECC (Elliptic Curve Cryptography), AES у скороченому режимі (наприклад, AES-128) або TLS з оптимізаціями для обмежених пристроїв (DTLS - Datagram TLS) [9]. Шифрування необхідне не лише для захисту переданих даних, але й для збереження конфіденційності команд, що надсилаються до пристрою.

3. Оновлення програмного забезпечення

Механізм безпечного оновлення прошивки є критично важливим, оскільки дозволяє виправляти виявлені вразливості. Наявність функціоналу OTA (Over-the-Air Update) [9] забезпечує централізоване управління оновленнями, однак за відсутності перевірки цілісності пакета оновлення існує ризик зараження пристрою підробленим ПЗ. У зв’язку з цим актуальним стає використання цифрових підписів та механізмів перевірки хешів під час оновлення.

4. Сегментація мережі та фаєрволи

Ізоляція IoT-пристроїв у окремі підмережі дозволяє мінімізувати потенційне поширення атаки у разі зараження одного з вузлів. Застосування вбудованих фаєрволів або фільтрація трафіку на рівні маршрутизатора обмежує небажаний вхідний/вихідний трафік, а також ускладнює сканування пристроїв зловмисником. У складніших інфраструктурах доцільно впроваджувати VPN-з'єднання для управління пристроями, які фізично перебувають поза межами захищеної мережі.

Окрім цього, важливою складовою є дотримання існуючих стандартів безпеки. Серед них [9]:

- **ISO/IEC 27001** - загальний стандарт управління інформаційною безпекою;
- **ISO/IEC 29192** - легковагова криптографія;
- **NIST SP 800-183** - рекомендації щодо архітектури IoT;
- **ETSI EN 303 645** - європейський стандарт для безпеки споживчих IoT-пристроїв.

Ці стандарти формують основу для розробки політик безпеки, які враховують життєвий цикл пристрою: від виробництва до утилізації.

Застосування стандартних методів захисту є першим і найважливішим кроком у побудові стійких IoT-систем. Однак, враховуючи динамічний розвиток загроз, цих методів недостатньо для повноцінного захисту від складних атак, зокрема ботнет-мереж. Це зумовлює необхідність використання додаткових, адаптивних та інтелектуальних засобів виявлення та реагування на інциденти

2.2 Стандарти та нормативні вимоги у сфері безпеки IoT

З огляду на стрімке зростання кількості IoT-пристроїв та збільшення масштабів інтегрованих кіберфізичних систем, питання стандартизації безпеки стало одним із ключових напрямів сучасних досліджень. Розробники, оператори інфраструктур та державні регулятори стикаються з проблемою відсутності універсальних вимог, що призводить до нерівномірного рівня захисту у різних сегментах ринку. Саме тому міжнародні організації - ISO, IEC, ETSI, NIST, ITU - сформували низку стандартів, які описують як загальні

принципи інформаційної безпеки, так і специфічні вимоги до IoT-пристроїв [16].

2.2.1 ISO/IEC 27001 - системи управління інформаційною безпекою

ISO/IEC 27001 є фундаментальним стандартом у сфері управління інформаційною безпекою. Хоча він не спеціалізується на IoT, саме його підходи визначають архітектурну модель побудови політик безпеки в більшості IoT-рішень корпоративного рівня[17].

Стандарт складається з трьох ключових елементів:

1. **Оцінювання ризиків (Risk Assessment)**
   Використовуються моделі визначення ймовірностей загроз та їхнього впливу на сервіси, що керують IoT-пристроями.
   У контексті IoT особливу увагу приділяють:
   – відмовостійкості пристроїв,
   – загрозам фізичного доступу,
   – відсутності криптографічного захисту.
2. **Контрольні механізми Annex A**
   У рамках IoT найбільш важливими є:
   A.9 (Control Access) - контроль доступу до пристроїв та інтерфейсів,
   A.10 (Cryptographic controls) - вимоги до шифрування,
   A.12 (Operations security) - логування телеметрії та подій,
   A.14 (System acquisition) - вимога закладати безпеку у фазі проєктування.
3. **Безперервність безпеки**
   ISO 27001 передбачає системний підхід: кожна зміна в IoT-інфраструктурі має проходити ризик-аналіз і перевірку відповідності політикам.

Таким чином, ISO 27001 забезпечує рамковий підхід до побудови IoT-безпеки у масштабних системах та критичній інфраструктурі [38, 46].

2.2.2 ISO/IEC 29192 - легковагова криптографія

Цей стандарт є критичним для IoT-пристроїв, які не можуть використовувати класичні криптографічні алгоритми. Його метою є забезпечення безпеки у середовищах із жорсткими обмеженнями щодо: пам’яті, енергоспоживання, пропускної здатності, обчислювальних потужностей.

ISO/IEC 29192 складається з шести частин, де найбільш важливими для IoT є [20]:

- **29192-2: Block Ciphers** - легковагові блокові шифри SPECK, SIMON, PRESENT.
- **29192-3: Stream Ciphers** - оптимізовані потокові шифри з мінімальним footprint.
- **29192-4: Message Authentication Codes** - MAC-алгоритми для автентифікації пакетів.
- **29192-5: Hash Functions** - компактні геш-функції з низьким навантаженням.

У підсумку ISO/IEC 29192 є стандартом, без якого неможливо побудувати сучасні протоколи IoT.

2.2.3 NIST SP 800-183 - рекомендації до архітектури IoT

Документ формує узагальнену структуру IoT-архітектури, описуючи такі логічно відокремлені компоненти [21, 45]:

- **IoT device** - кінцевий пристрій або сенсор, що генерує дані;
- **gateway** - проміжний вузол, який забезпечує маршрутизацію та попередню обробку;
- **communication network** - мережеве середовище для передачі даних;
- **cloud service layer** - хмарні сервіси, що виконують зберігання, аналітику та керування.

Також у документі визначено моделі взаємодії між компонентами, включно з:

- **data flow** - рухом даних між пристроями та сервісами;
- **control flow** - передаванням керувальних команд;
- **life cycle management** - управлінням повним життєвим циклом пристрою.

Окрему увагу приділено базовим вимогам до безпеки IoT-архітектури, серед яких:

- автентифікація між усіма компонентами системи;
- забезпечення захищеної передачі команд;
- ізоляція та сегментація пристроїв у мережі;
- використання захищених механізмів оновлення (secure update pipeline).

Документ також описує характерні для IoT загрози:

- **device cloning** - клонування пристроїв;
- **hijacking of control flow** - перехоплення та зміна керувальних команд;
- **supply-chain attacks** - атаки на етапах постачання і виробництва;
- **insecure onboarding** - небезпечні процеси первинного підключення пристроїв.

Узагальнюючи, документ визначає канонічну модель архітектури IoT і слугує основою для формування національних рекомендацій у сфері IoT-безпеки.

2.2.4 ETSI EN 303 645 - стандарт для споживчих IoT

ETSI EN 303 645 є першим у світі масовим стандартом безпеки споживчих IoT-пристроїв, який набув широкого застосування в ЄС.

Він містить 13 базових вимог, серед яких найбільш важливі [22]:

1. **Заборона заводських паролів**
   Кожен пристрій повинен мати унікальний пароль або вимагати його зміни під час першого запуску.
2. **Обов'язкові безпечні оновлення**
   Виробник повинен:
   – надавати OTA-оновлення,
   – перевіряти цифрові підписи,
   – використовувати захищені канали доставки прошивок.
3. **Захист особистих даних**
   Дані мають зберігатися мінімально необхідний час, використовувати шифрування та відповідати GDPR.
4. **Мінімізація поверхні атаки**
   – заборона відкритих діагностичних портів;
   – відсутність небезпечних сервісів за замовчуванням;
   – вимога до firewall-by-default.
5. **Безпечний процес приєднання пристроїв (onboarding)**
   – підтвердження автентичності,
   – захист від повторних атак,
   – виключення broadcast-ключів.

Цей стандарт є де-факто основою безпеки побутових пристроїв у Європі.

2.2.5 ETSI EN 303 645 - стандарт для споживчих IoT

Документ описує повний життєвий цикл IoT-пристрою, включно з [23, 43]:

- виробництвом,
- первинною конфігурацією,
- довготривалою експлуатацією,
- оновленнями,
- утилізацією.

Ключові вимоги включають:

- унікальні облікові записи та ідентифікатори;
- захищене керування конфігурацією;
- можливість безпечного оновлення;
- ведення журналів безпеки;
- прозору політику розкриття уразливостей (Coordinated Vulnerability Disclosure).

NISTIR 8259 став основою для регуляторних актів у США та ряді інших країн [44].

2.3. Методи виявлення ботнет-атак

Однією з найнебезпечніших загроз для Інтернету речей є ботнет-мережі, які здатні використовувати вразливі пристрої для проведення масштабних DDoS-атак, поширення шкідливого ПЗ або несанкціонованого доступу до внутрішніх систем. Враховуючи обмежені обчислювальні ресурси IoT-пристроїв та особливості топології таких мереж, виявлення ботнет-активності є складним завданням [19].

Метод виявлення	**Переваги**	**Недоліки**	**Застосовність в IoT**
Сигнатурне виявлення	Висока точність для відомих атак	Не виявляє нові або модифіковані загрози	Висока (обмеженою мірою)
Виявлення аномалій	Виявляє нові/невідомі загрози	Високий рівень хибнопозитивних спрацювань	Середня

DNS-аналіз	Виявляє DGA-ботнети, працює без доступу до внутрішньої мережі	Вимагає інтерпретації даних, неефективний при шифруванні трафіку	Висока
Машинне навчання та ШІ	Самонавчання, висока гнучкість	Високі вимоги до ресурсів і якості навчання	Середня–висока (в хмарі)
IDS (мережеві, хостові)	Інтегрує кілька підходів, адаптується до мережі	Потребує конфігурації та регулярного обслуговування	Висока

Таблиця 2.1 Методи виявлення ботнет атак

1. Виявлення на основі сигнатур

Найстаріший і найпростіший підхід - це сигнатурне виявлення, яке базується на зіставленні ознак трафіку з відомими шаблонами шкідливої активності. Такі системи ефективні для розпізнавання вже відомих ботнетів (наприклад, Mirai або Satori), оскільки використовують заздалегідь підготовлені правила. Водночас головним недоліком є їх нездатність виявляти нові або модифіковані атаки без попереднього оновлення бази сигнатур.

2. Виявлення аномальної поведінки

Методи, що базуються на аналізі аномалій, передбачають побудову профілю «нормальної» поведінки пристрою чи мережі [32]. Якщо зафіксовано суттєве відхилення від очікуваного шаблону, система позначає трафік як потенційно шкідливий. Такий підхід дозволяє виявляти нові типи ботнет-активності, проте часто має високий рівень хибних спрацьовувань, особливо в умовах гетерогенного середовища IoT.

3. DNS-аналіз та виявлення DGA

Багато ботнетів використовують алгоритми генерації доменних імен (DGA), щоб приховати місцезнаходження командно-контрольних серверів. Аналіз DNS-запитів, зокрема частоти, характеру та структури імен, дозволяє виявити підозрілу активність ще до активації шкідливого коду. Цей підхід добре працює в поєднанні з методами класифікації на основі поведінки. [22]

4. Машинне навчання та глибоке навчання

Застосування алгоритмів машинного навчання (ML) та глибоких нейронних мереж (DL) дозволяє автоматизувати процес виявлення ботнет-активності [30]. Алгоритми можуть виявляти складні закономірності у великому обсязі мережевого трафіку, які важко відстежити традиційними методами. Наприклад, автоенкодери, класифікатори на основі LSTM чи CNN, довели свою ефективність у роботах, присвячених виявленню Mirai та його

похідних. Проблемою залишається потреба у якісних датасетах та обчислювальних ресурсах.

5. Системи виявлення вторгнень (IDS)

Мережеві (NIDS) та хостові (HIDS) системи виявлення вторгнень залишаються важливим компонентом в інфраструктурі безпеки. Вони здатні комбінувати сигнатурні й поведінкові механізми, забезпечуючи гнучкість і адаптивність. Для IoT середовища рекомендовано використовувати легковагові IDS, які не перевантажують пристрої, але здатні взаємодіяти з централізованими аналітичними платформами.

2.4. Технології обману: honeypots, deception technology.

В умовах еволюції кіберзагроз, зокрема в середовищі Інтернету речей, дедалі більшої популярності набувають так звані «технології обману» - підходи, спрямовані не лише на захист від атак, а й на активне виявлення зловмисників шляхом створення ілюзії вразливого середовища. Такий підхід базується на ідеї заманювання нападника в контрольовану зону, де його дії можна аналізувати в режимі реального часу. До цієї категорії належать honeypots та розширені засоби deception technology [33,50].

Honeypots

Honeypot - це окремий пристрій або програмний сервіс, що навмисно імітує вразливу систему, з метою приваблення зловмисника [34]. На відміну від реальних цілей, honeypot не має практичного призначення в інфраструктурі, тому будь-яка взаємодія з ним розглядається як підозріла або шкідлива за своєю природою.

У середовищі IoT honeypots можуть імітувати типові пристрої, наприклад, IP-камери, медичні монітори, смарт-термостати чи маршрутизатори. Їх застосування дає змогу вивчати специфіку атак на обмежені пристрої та методи, які використовуються зловмисниками для проникнення, ескалації прав та збереження присутності в системі. Зокрема, у публікаціях Nozomi Networks [35] та SOCRadar [36] розглядаються сценарії створення honeypot-серверів для виявлення ботнет-активності (наприклад, Mirai), що автоматично сканують мережу на предмет відкритих Telnet-портів.

Існують різні типи honeypots [37]:

- **Низькоінтерактивні** - імітують тільки частину функцій пристрою, реагують на базові мережеві запити. Вони прості у впровадженні та не становлять ризику для інфраструктури.
- **Високоінтерактивні** - більш складні, відтворюють повну поведінку системи, включаючи файлову систему, служби, мережеву активність тощо. Такі honeypots дозволяють вивчати складніші атаки, але потребують нагляду та обмеження доступу до внутрішніх ресурсів.

Ключовою перевагою honeypots є їх здатність збирати глибоку аналітику про дії зловмисника: IP-адресу, використовувані інструменти, спроби проникнення, ін'єкції шкідливого коду, експлойти. Отримана інформація може бути використана для оновлення сигнатур, тренування моделей виявлення та покращення загальної безпеки IoT-систем.

Deception Technology

Deception technology - це ширше поняття, яке охоплює не лише honeypots, а й цілу екосистему фальшивих елементів, розгорнутих усередині або на периферії справжнього IT/IoT-середовища [38]. Йдеться про розміщення підроблених облікових даних, фальшивих конфігурацій, псевдосерверів, віртуальних мереж, помилкових API-ключів тощо. Основна мета deception technology - не дозволити зловмиснику розпізнати справжню інфраструктуру та «відвести» його в хибному напрямку [39].

На відміну від класичних honeypots, технології обману інтегруються в реальні системи та працюють у тісному зв'язку з системами моніторингу, журналювання, SIEM-платформами. Це дозволяє автоматично фіксувати спроби несанкціонованого доступу, сповіщати адміністратора та навіть ініціювати автоматизовані дії - ізоляцію порушника, перенаправлення трафіку, блокування IP-адреси.

Серед ключових функцій deception-платформ:

- **Фальшиві облікові записи** в Active Directory або інших службах доступу.
- **Динамічне розміщення принади (breadcrumbs)** в системах, які зловмисники можуть знайти при внутрішньому проникненні.
- **Автоматизоване реагування**, що активується при виявленні активності в зоні обману.

Для IoT та промислових систем deception technology дозволяє не тільки виявити внутрішнього або зовнішнього зловмисника, а й запобігти розгортанню атаки до досягнення критичних вузлів. Такі системи особливо корисні у сценаріях, коли класичні методи виявлення (IDS, сигнатурні сканери) вже скомпрометовані або обійдені.

Одним із напрямків розвитку deception technology є використання штучного інтелекту для адаптації обманних елементів до змін в інфраструктурі та автоматизованого аналізу ризиків, що дозволяє знизити витрати на ручне налаштування.

2.5. Використання машинного навчання та AI у виявленні аномалій.

Розгортання IoT-систем створює значне навантаження на класичні механізми безпеки, які часто не здатні адаптуватися до великої кількості динамічних даних, низької однорідності трафіку та обмежених ресурсів пристроїв. Саме тому виявлення аномалій за допомогою машинного навчання (ML) та штучного інтелекту (AI) стає критично важливим елементом сучасного захисту IoT-середовищ [40].

Методи ML можуть бути класифіковані за типом навчання: кероване, некероване та напівкероване. Керовані методи передбачають навчання на мічених наборах даних, де відомо, що є «нормальною» поведінкою, а що - відхиленням. Це забезпечує високу точність, однак обмежено застосовне у сфері IoT, де мічені дані часто відсутні.

Некеровані методи натомість працюють без попереднього маркування, аналізуючи внутрішні структури даних для виявлення нетипових шаблонів. Саме цей підхід найбільш поширений у системах виявлення аномалій для IoT, особливо при використанні глибоких нейронних мереж. Напівкеровані

підходи поєднують обидва методи, дозволяючи ефективно використовувати обмежені мічені дані разом з немічеными.

Основні алгоритми та їх застосування

Серед найбільш ефективних алгоритмів для виявлення аномалій у IoT-середовищах можна виділити кілька [41]:

- **Random Forest** - забезпечує високу точність у класифікації, може працювати з великими масивами різноманітних атрибутів. Його застосування актуальне для фільтрації аномалій у трафіку з датчиків.

- **Support Vector Machines (SVM)** - ефективний у задачах з високою вимірністю, хоча вимагає попередньої обробки даних і нормалізації.
- **Autoencoders** - використовується для відновлення вхідних даних; високий рівень помилки реконструкції може вказувати на нетипову поведінку.
- **Isolation Forest** - швидкий алгоритм, спеціалізований саме на виявленні аномалій шляхом випадкової ізоляції нетипових значень.

Алгоритм	Тип навчання	Переваги	Обмеження	Застосування в IoT
Random Forest	Кероване	Висока точність, нечутливість до шуму	Потреба у мічених даних, не адаптивний	Аналіз трафіку, класифікація станів
SVM	Кероване	Добра робота з високовимірними просторами	Погано масштабується на великі дані	Виявлення мережевих аномалій
Autoencoder	Некероване	Працює без мічених даних, виявляє складні шаблони	Висока складність, обмежена інтерпретованість	Поведінковий аналіз пристроїв
Isolation Forest	Некероване	Швидкий, ефективний для великих наборів	Низька точність при складних залежностях	Виявлення ботнет-активності, DDoS-підготовки

Таблиця 2.2 Порівняння алгоритмів машинного навчання

Хоча AI-підходи демонструють високу гнучкість, їх практичне застосування у сфері IoT обмежується кількома факторами. По-перше, самі IoT-пристрої не завжди мають достатньо ресурсів для обчислювально складних моделей, тому їх навчання та виконання часто потребують переносу на edge- або хмарні середовища.

По-друге, дані, що надходять з IoT-пристроїв, можуть бути неповними, шумовими або такими, що змінюються у часі, що ускладнює підтримку стабільності моделі. Для вирішення цієї проблеми активно досліджується застосування методів федеративного навчання, які дозволяють тренувати моделі децентралізовано - без потреби вивантаження чутливої інформації в централізовані сховища.

2.6. Висновок до розділу 2

Аналіз існуючих підходів до захисту IoT-систем дозволив сформувати чітке бачення необхідної стратегії безпеки. Розгляд міжнародних стандартів, зокрема ETSI EN 303 645 та NIST SP 800-183, підтвердив важливість базових заходів (унікальні паролі, шифрування), однак на практиці їх впровадження залишається фрагментарним і недостатнім для протидії цілеспрямованим атакам .

У ході порівняльного аналізу методів виявлення вторгнень було виявлено суттєві обмеження кожного з них при окремому застосуванні. Сигнатурні методи, хоч і швидкі, не здатні реагувати на атаки нульового дня. Водночас методи на основі машинного навчання (ML) та штучного інтелекту, попри високу адаптивність, вимагають значних обчислювальних ресурсів, що є критичним недоліком для периферійних пристроїв IoT .

На основі цього аналізу було обґрунтовано доцільність застосування гібридного підходу, який поєднує легковагові статистичні методи та методи сигнатурного аналізу загроз. Таке поєднання дозволяє ефективно виявляти аномалії в мережевому трафіку без надмірного навантаження на систему, а також дає можливість вивчати тактику зловмисників у контрольованому середовищі . Саме ця концепція лягла в основу розробки власної системи захисту.

РОЗДІЛ ІІІ. Розробка способу захисту IoT-систем від ботнет-атак.

3.1. Вибір підходу до реалізації захисту

Підхід до реалізації захисту в даній роботі ґрунтується на побудові модульної архітектури локальної IoT-системи безпеки, що забезпечує виявлення та моніторинг ботнет-активності на рівні мережевого трафіку. Основна ідея полягає у створенні контрольованого середовища, яке моделює поведінку як легітимних пристроїв Інтернету речей, так і потенційно заражених вузлів, з метою дослідження методів їх диференціації на основі аналізу мережевих подій.

Архітектура системи передбачає три логічні рівні. На першому рівні функціонує підсистема емуляції пристроїв, яка генерує UDP-повідомлення уніфікованого формату. В ролі звичайних пристроїв виступають скрипти емуляторів сенсорів температури та диму, які надсилають повідомлення із параметрами, що змінюються в межах фізично обґрунтованих діапазонів. Паралельно з ними працюють модулі моделювання шкідливої активності - скрипти, що імітують ботнет-поведінку: сканування портів, DDoS-атаки та ексфільтрацію даних [24]. Такий підхід дозволяє створити повноцінну тестову мережу, у якій одночасно присутній як нормальний, так і аномальний трафік.

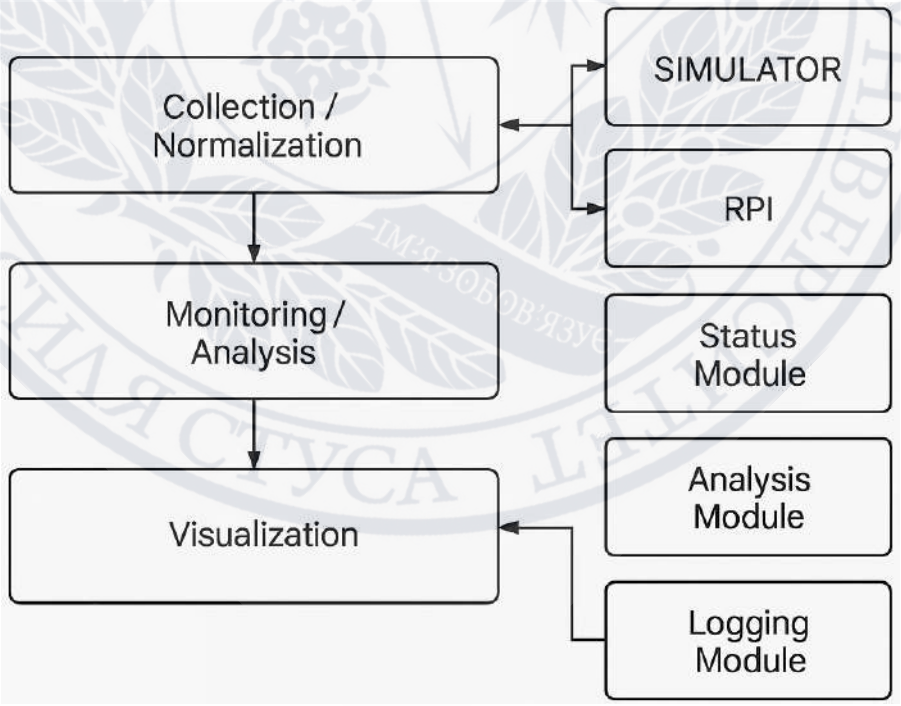

Рисунок 3.1 Архітектура запропонованого рішення

Другий рівень утворює аналітичне ядро системи - модуль Detector, який виконує класифікацію подій на основі комбінації сигнатурних правил і часових вікон. Для виявлення DDoS-активності використовується облік

кількості подій типу botnet_attack з одного IP-адреса за визначений інтервал часу [26]. Якщо їх кількість перевищує встановлений поріг, формується подія типу ddos із високим рівнем серйозності. Аналогічно, множинні запити типу scan_probe протягом короткого періоду розпізнаються як port scan. Механізм sliding window забезпечує динамічне спостереження за активністю у реальному часі, дозволяючи виявляти пікові навантаження без потреби у збереженні історичних даних. Окрім цього, система аналізує обсяг кожного пакета: якщо розмір повідомлення перевищує 2 кілобайти, воно класифікується як exfiltration, що відображає типову поведінку витоку даних. Для звичайних IoT-сенсорів передбачено набір граничних значень параметрів (наприклад, температури чи тиску), перевищення яких інтерпретується як локальна аномалія.

Третій рівень охоплює підсистему журналювання та моніторингу. Модуль Logger виконує багаторівневе логування усіх подій у форматі CSV із ротацією файлів, розділяючи загальні записи від тих, що мають ознаку alert=True. Такий підхід забезпечує можливість подальшого аналізу, навчання моделей виявлення та формування статистики з високою точністю. Паралельно функціонує модуль Dashboard, реалізований на базі вбудованого багатопотокового HTTP-сервера. Він забезпечує візуалізацію метрик у режимі реального часу: кількість подій, швидкість обробки, середню затримку, відсоток сповіщень, розподіл подій за типами тощо. Це дозволяє досліднику оперативно оцінювати стан системи та коректність роботи механізмів детектування.

Вибір такої архітектури зумовлений низкою практичних міркувань. По-перше, UDP-протокол забезпечує мінімальні затримки та дозволяє емулювати великий обсяг IoT-трафіку без ускладнень, пов’язаних із встановленням з’єднань. По-друге, відмова від використання зовнішніх баз даних і брокерів зменшує складність та підвищує прозорість експериментів, що є важливим для дослідницьких систем. По-третє, реалізація правил детектування у вигляді комбінації регулярних виразів, порогових значень і часових вікон дозволяє легко адаптувати систему до нових типів атак без суттєвої зміни архітектури. Такий підхід забезпечує не лише гнучкість, але й можливість подальшої інтеграції машинного навчання для автоматичного вдосконалення критеріїв класифікації [25].

Окремої уваги заслуговує підхід до візуалізації та керування. Локальний веб-дашборд, реалізований без зовнішніх бібліотек, відображає агреговані показники системи, що дозволяє одночасно проводити моніторинг і

тестування ефективності алгоритмів виявлення у реальному часі. Додатково застосовано кольорове консольне логування для швидкої ідентифікації критичних подій у CLI-режимі.

У цілому, обраний підхід до реалізації захисту являє собою гібридну модель, що поєднує сигнатурні методи з елементами поведінкового аналізу. Вона характеризується високою відтворюваністю, низькими вимогами до ресурсів і модульністю, яка дозволяє поступово розширювати функціональність. Така система забезпечує комплексне дослідження механізмів виявлення ботнет-активності та аномалій IoT-пристроїв, формуючи основу для подальшої розробки інтелектуальних методів захисту з використанням машинного навчання [29, 30].

3.2. Архітектура запропонованого рішення

Архітектура запропонованої системи побудована на принципах модульності, розподілу відповідальності та гнучкості, що дозволяє їй функціонувати у двох різних режимах: як симулятор для лабораторних досліджень та як монітор реальної мережі. Ця двопрофільна природа є ключовою особливістю системи, що керується змінною середовища IOT_EMU_PROFILE [32].

Узагальнено архітектуру можна розділити на п’ять логічних рівнів: рівень джерел даних, рівень збору та нормалізації, рівень аналізу та детектування, рівень журналювання та рівень візуалізації [27].

1. Рівень джерел даних (Гібридний)

Джерело інформації визначається активним профілем роботи:

Профіль simulator: У цьому режимі джерелами є набір скриптів-емуляторів (iot_smoke.py, iot_temp.py, bot_scan.py, bot_exfil.py та ін.). Вони генерують синтетичний UDP-трафік, що імітує як легітимну телеметрію IoT-пристроїв (напр., sensor=temperature;value=22.5), так і різноманітні вектори атак (напр., botnet_attack, scan_probe) [28].

Профіль rpi (NetFlow): У цьому режимі система перетворюється на інструмент моніторингу реальної мережі. Джерелом даних виступає мережеве обладнання (наприклад, маршрутизатор), що експортує мережевий трафік у форматі NetFlow або IPFIX. Ці дані збираються зовнішньою утилітою nfcapd і зберігаються у вигляді бінарних файлів (nfcapd.*) у спеціальному каталозі (NETFLOW_DIR).

2. Рівень збору та нормалізації (Уніфікований вхід)

Цей рівень відповідає за збір даних із джерел та їх перетворення у єдиний формат для аналітичного ядра.

Для профілю simulator: Емулятори надсилають свої UDP-повідомлення безпосередньо на центральний порт моніторингу (PORT).

Для профілю rpi: Цей процес значно складніший і виконується модулем netflow_ingest.py. Цей модуль працює як окремий, постійний процес, що:

- Періодично сканує каталог NETFLOW_DIR на наявність нових nfcapd файлів.
- Використовує файл стану (NETFLOW_STATE_FILE) для відстеження останнього обробленого файлу, запобігаючи дублюванню.
- Викликає системну утиліту nfdump для парсингу бінарних файлів nfcapd та їх конвертації у текстовий формат CSV.
- Нормалізує кожен рядок CSV (кожен потік) у стандартизоване текстове UDP-повідомлення (напр., flow src=... dst=... packets=... bytes=...).

Надсилає ці нормалізовані повідомлення на центральний порт моніторингу (PORT), де їх очікує ядро аналізу.

Таким чином, незалежно від профілю, аналітичне ядро завжди отримує стандартизовані текстові UDP-повідомлення.

3. Рівень аналізу та реагування (Ядро системи)

Центром цього рівня є модуль monitor.py, який слухає UDP-порт (PORT) і діє як головний хаб. Кожне отримане повідомлення (синтетичне чи NetFlow) негайно передається модулю detector.py для класифікації.

Детектор реалізує гібридну логіку аналізу:

- Аналіз на основі подій (Event-based): Для повідомлень від емуляторів (simulator) застосовуються регулярні вирази для ідентифікації типу події та аналіз на основі ковзних часових вікон для виявлення аномалій (напр., перевищення ddos_threshold повідомлень botnet_attack за ddos_window секунд).
- Аналіз на основі потоків (Flow-based): Для повідомлень flow ... (rpi) детектор застосовує інший набір правил. Він агрегує статистику (пакети, байти, унікальні порти) для кожної IP-адреси та порівнює ці показники

з пороговими значеннями, визначеними у DETECTOR_SETTINGS (напр., flow_ddos_packets, flow_scan_ports, flow_exfil_bytes) [31]. Це дозволяє виявляти ті ж самі класи атак (DDoS, сканування портів, ексфільтрація), але на основі метаданих реального мережевого трафіку.

Результатом роботи детектора є уніфікований об'єкт, що описує подію (тип, серйозність, наявність тривоги, джерело, метадані).

4. Рівень журналювання (Збереження та аудит)

Система забезпечує комплексне журналювання:

- Журнали подій (CSV): Модуль logger.py (використовується у monitor.py) записує всі класифіковані події у raw_log.csv та лише тривоги у alert_log.csv. Ці журнали містять події з обох профілів, включно з source=netflow.
- Журнали процесів: Архітектура підтримує ведення окремих файлів журналу для системних компонентів (напр., netflow_ingest.log), що є критичним для відлагодження у режимі rpi.
- Файл стану: Модуль netflow_ingest.py веде власний JSON-файл стану (NETFLOW_STATE_FILE), що є формою персистентного журналу для відстеження прогресу обробки файлів NetFlow.

5. Рівень візуалізації та аналітики (Інтерфейс)

Цей рівень реалізований модулем dashboard.py, який запускає багатопотоковий ThreadingHTTPServer для надання веб-інтерфейсу.

- Стан системи: Спеціальний клас DashboardState діє як потоко-безпечне сховище в пам'яті для всіх метрик та останніх подій.
- Агрегація NetFlow: DashboardState розширено для накопичення детальної статистики NetFlow (обсяги трафіку, топ джерел/отримувачів, протоколи, порти), яка надходить у метаданих від детектора.
- **Інтерактивна панель:** Веб-інтерфейс (HTML/JS) динамічно оновлюється, відображаючи:
  - Загальні метрики (RPS, затримка, p95/p99).
  - Графіки трендів та розподілу джерел.

– Окремий розділ "NetFlow analytics" з картками та таблицями, що показують рейтинг найбільших переданих даних (top talkers) у мережі, популярні порти та протоколи.

– Вбудований переглядач логів: Нова функціональність, що дозволяє користувачу через API-ендпоїнти (/logs/raw, /logs/process) переглядати вміст CSV-журналів та журналів процесів безпосередньо у браузері.

Отже, архітектура є гнучким, гібридним фреймворком, що об'єднує емулятор для тестування правил та повноцінний монітор NetFlow для розгортання у реальних мережах (напр., на Raspberry Pi), використовуючи єдине ядро аналізу та візуалізації.

3.3. Програмна реалізація

Програмна реалізація системи виконана мовою Python 3, з акцентом на модульність, прозорість та мінімальну залежність від зовнішніх бібліотек для основних функцій. Система складається з набору скриптів, що виконують чітко визначені ролі: від запуску та конфігурації до збору, аналізу та візуалізації даних.

3.3.1. Модулі конфігурації та запуску (config.py, run_all.py)

В основі системи лежить централізований конфігураційний файл core/config.py. Він визначає всі ключові параметри середовища, використовуючи функцію os.getenv для зчитування змінних середовища. Кожна змінна має значення за замовчуванням (наприклад, IOT_EMU_NETFLOW_DIR за замовчуванням /home/cosm1cus/netflow), що дозволяє гнучко налаштовувати шляхи, порти та порогові значення детектора (DETECTOR_SETTINGS) без зміни коду, що є критичним для розгортання у різних профілях (напр., simulator vs rpi).

Керування життєвим циклом системи здійснює run_all.py. Цей скрипт виступає у ролі менеджера процесів. Його головна функція, build_scripts, аналізує змінну середовища IOT_EMU_PROFILE. Логіка наступна:

1. Створюється базовий список scripts, який завжди містить monitor.py.
2. Якщо PROFILE має значення simulator (або lab, full), список scripts **розширюється** (.extend()) повним набором емуляторів з каталогів Good_Devices та Bad_devices.

3. Якщо PROFILE має значення rpi (або netflow), до списку scripts **додається** (.append()) лише netflow_ingest

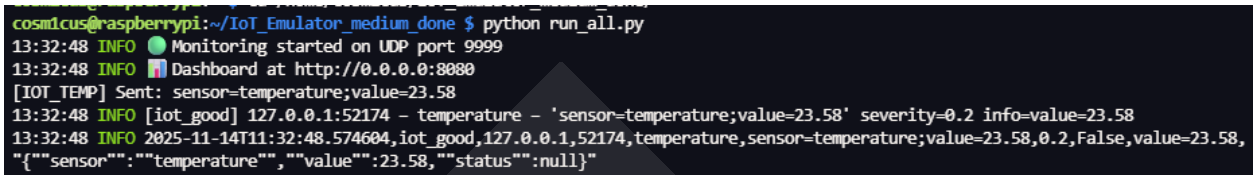

Рис. 3.2 Приклад роботи «run_all.py»

Після формування списку команд, функція start_all ітерує по ньому та запускає кожен скрипт як окремий, незалежний процес за допомогою subprocess.Popen. Для полегшення розгортання на цільовій платформі (Raspberry Pi) використовується обгортковий скрипт start_rpi.sh, який завантажує змінні з .env.rpi, активує віртуальне оточення .venv та передає керування run_all.py.

3.3.2. Модуль збору NetFlow (netflow_ingest.py)

Цей модуль є ключовим для роботи системи у режимі rpi і працює як самостійний, безперервний процес, що інтегрується із системною утилітою nfdump.

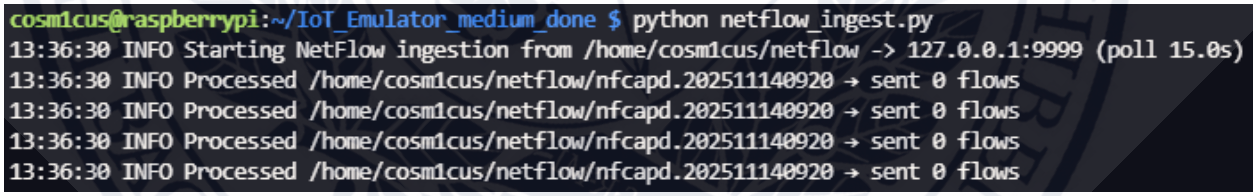

Рис. 3.3 Приклад роботи «netflow_ingest.py»

1. **Керування станом:** При запуску ініціалізується клас NetflowState. Його конструктор викликає _load, що намагається прочитати JSON-файл NETFLOW_STATE_FILE. Звідти завантажується last_processed (ім'я файлу) та last_mtime (час його модифікації). Це дозволяє процесу відновити роботу з точки, на якій він зупинився, навіть після перезапуску.
2. **Виявлення файлів:** Головний цикл run періодично (раз на NETFLOW_POLL_INTERVAL секунд) викликає _discover_files. Ця функція рекурсивно сканує NETFLOW_DIR (через rglob(f'{FLOW_PATTERN}*')), знаходить усі файли nfcapd.*, сортує їх за часом модифікації та повертає список лише тих файлів, що є новішими за збережений last_mtime.

3. **Парсинг (_read_flows):** Для кожного нового файлу зі списку викликається _read_flows. Ця функція динамічно формує командний рядок у вигляді списку cmd, що містить шлях до nfdump (з NETFLOW_NFDUMP_PATH), прапор -r з шляхом до файлу nfcapd, та прапор -o csv для вказання формату виводу. Ця команда виконується через subprocess.run, що дозволяє захопити stdout (текстові дані CSV) та stderr (повідомлення про помилки). Реалізована обробка помилок, зокрема FileNotFoundError, на випадок, якщо nfdump не встановлено або вказано невірний шлях.

4. **Форматування та відправка (_emit, _format_message):** Функція _emit ітерує по словниках потоків, отриманих з _parse_csv. Для кожного потоку _format_message витягує ключові поля (наприклад, sa, da, sp, dp, pr, ipkt, ibyt, td), проводить необхідні перетворення типів (в int або float, обробляючи можливі помилки ValueError) та формує єдиний, стандартизований текстовий рядок, де поля розділені пробілами (напр., flow src=... dst=... packets=...). Цей нормалізований рядок надсилається (sock.sendto) на UDP-порт (NETFLOW_SEND_HOST:NETFLOW_SEND_PORT), де його очікує monitor.py.

5. **Оновлення стану:** Після успішної обробки файлу викликається state.mark_processed(str(path)), що перезаписує NETFLOW_STATE_FILE з новим last_mtime та іменем файлу.

3.3.3. Головний модуль моніторингу (monitor.py)

Цей модуль є центральним вузлом системи, який виконує роль UDP-сервера, аналітичного шлюзу та агрегатора метрик.

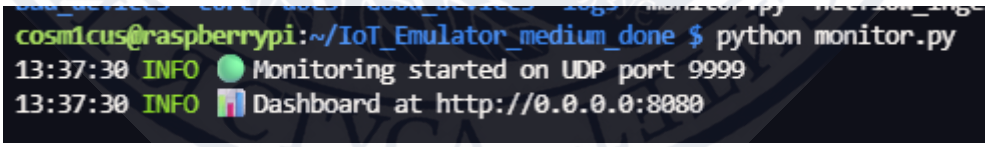

Рис. 3.4 Приклад роботи «monitor.py»

1. **Ініціалізація:** Створюються екземпляри Logger (для CSV-файлів) та Detector (для аналізу), ініціалізується DashboardState та запускається start_dashboard (веб-сервер). UDP-сокет прив'язується до 0.0.0.0 та PORT.

2. **Головний цикл:** Процес входить у while цикл, що працює, доки не спрацює прапор зупинки stop['flag'] (який встановлюється обробниками

сигналів SIGINT/SIGTERM). Основна логіка знаходиться всередині блоку try...except socket.timeout. Процес блокується на sock.recvfrom(65535), очікуючи на дані. Таймаут 0.5 секунди гарантує, що цикл не "зависне" назавжди і зможе перевірити stop['flag'].

Коли дані отримано, фіксується час recv_ts, повідомлення декодується з utf-8. Для вимірювання продуктивності детектора, час фіксується (time.perf_counter()) безпосередньо перед викликом result = detector.classify(msg, addr) та одразу після, обчислюючи dt_ms (час обробки в мілісекундах). Отриманий result (словник з даними аналізу) та dt_ms передаються до file_logger.log_detailed та dashboard_state.add_event для журналювання та візуалізації.

Кожні 5 секунд monitor.py обчислює середню затримку (avg_ms) та RPS за останній період. Він викликає detector.get_metrics() для отримання максимальних значень вікон (напр., ddos_max_window_size, flow_max_packets). Ці дані, разом з обчисленими перцентилями (p95, p99), передаються до dashboard_state.update_rates(...) та dashboard_state.update_extras(...).

3.3.4. Модуль аналітики (detector.py)

Це головний модуль системи, його екземпляр створюється у monitor.py.

1. **Ініціалізація:** Конструктор Detector приймає DETECTOR_SETTINGS (словник з пороговими значеннями з config.py).
2. **Метод classify:** Це основний метод, що викликається для кожного UDP-повідомлення. Він реалізує гібридну логіку:
    - **Шлях 1 (Емуляція):** Повідомлення перевіряється на відповідність сигнатурам (напр., botnet_attack, scan_probe). Якщо збіг знайдено, детектор оновлює внутрішні stateful-об'єкти (ковзні вікна), підраховуючи кількість подій за IP. Якщо лічильник перевищує ddos_threshold за ddos_window секунд, повертається результат з alert=True.
    - **Шлях 2 (NetFlow):** Повідомлення перевіряється на сигнатуру flow .... Якщо збіг є, рядок парситься. Детектор, також веде stateful-агрегацію (напр., сума packets та bytes за IP) та просто порівнює значення окремого потоку з пороговими (напр., if packets > flow_ddos_packets: alert=True). Він також відстежує унікальні порти для flow_scan_ports.

3. **Метод get_metrics:** Цей метод повертає словник з максимальними зафіксованими значеннями (напр., ddos_max_window_size, flow_max_packets), які monitor.py використовує для звітів.

3.3.5. Модуль візуалізації (dashboard.py)

Модуль реалізує повноцінний веб-інтерфейс на базі вбудованого ThreadingHTTPServer.

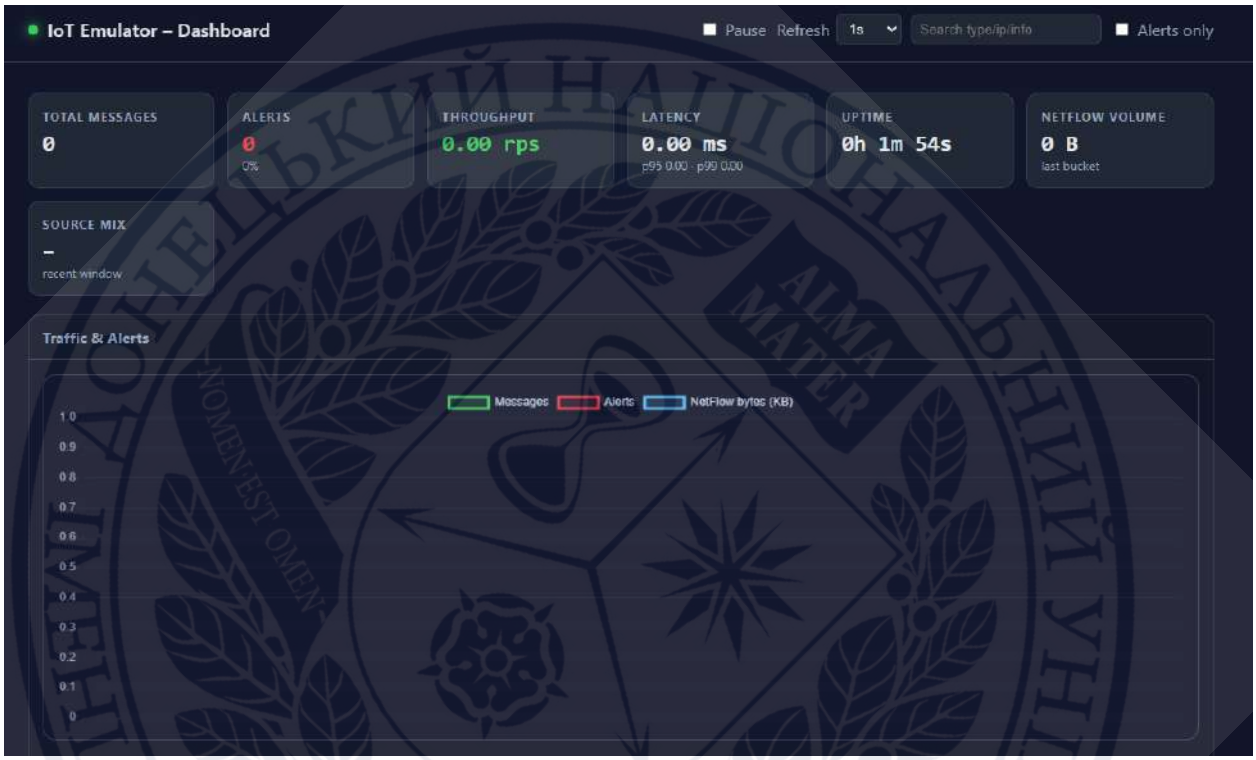

Рис. 3.5 Вигляд інформаційної панелі зі статистикою

1. **DashboardState:** Це потоко-безпечний клас, що зберігає стан. Він використовує threading.Lock() для захисту доступу до своїх полів (напр., recent_events, per_type_counts, netflow_src_bytes).
2. **add_event та _update_history:** Метод add_event викликає _update_history. Ця функція є критично важливою для агрегації. Вона перевіряє поле source події. Якщо source == 'netflow', вона безпечно витягує meta об'єкт. З метаданих вона зчитує bytes (з фолбеком на window_bytes), src, та dst. Потім вона оновлює словники-агрегатори, наприклад, self.netflow_src_bytes[src] += bytes_value та self.netflow_dst_bytes[dst] += bytes_value, забезпечуючи накопичення довгострокової статистики для NetFlow.

3. **snapshot:** Цей метод, що викликається DashboardHandler, блокує self.lock, фільтрує recent_events за часовим вікном (window_seconds) та обчислює всі похідні метрики (напр., alert_rate, netflow_alert_rate_recent), а також формує топ-10 списки з агрегатів NetFlow.
4. **API Endpoints (DashboardHandler):**
    - / (головна): Рендерить HTML/JS/CSS інтерфейс методом _render_html.
    - /metrics: Повертає JSON-результат методу state.snapshot().
    - /logs/raw, /logs/alerts: Використовують допоміжну функцію _tail_csv. Ця функція ефективно читає останні N рядків з CSV-файлу. Вона відкриває файл за допомогою csv.DictReader і читає його рядок за рядком, але додає рядки до collections.deque з фіксованим maxlen=limit. deque автоматично видаляє старі елементи при додаванні нових, що дозволяє отримати N останніх рядків, обробивши файл один раз, без завантаження його повністю в пам'ять.
    - /logs/process: Дозволяє переглядати файли з PROCESS_LOG_DIR. Якщо параметр name не вказано, повертає список .log файлів; інакше викликає _tail_text (аналог _tail_csv для звичайних текстових файлів) для читання конкретного файлу.
5. **Frontend (JavaScript):** Вбудований у HTML-код JavaScript виконує fetch('/metrics') з інтервалом, що задається (intervalSel). Отримавши JSON, він оновлює DOM-елементи (document.getElementById(...)), заповнює таблиці NetFlow (populateTable) та оновлює графіки Chart.js (trendChart, sourceChart).

3.3.6. Модулі емуляції (Профіль simulator)

Скрипти в каталогах Good_Devices та Bad_devices (iot_smoke.py, bot_scan.py тощо) є найпростішими компонентами. Кожен з них є UDP-клієнтом, що працює у нескінченному циклі (while True), формує специфічний для нього рядок повідомлення та надсилає його (sock.sendto) на HOST:PORT з невеликою випадковою затримкою. Їхня єдина мета - генерація трафіку для тестування детектора у профілі simulator.

3.4. Аналіз отриманих результатів

Після запуску системи було проведено серію експериментів із симуляцією одночасної роботи легітимних IoT-пристроїв та ботнет-активності. До системи надходили UDP-пакети від чотирьох джерел: двох сенсорів (температури та диму) та двох емуляторів атак (сканування портів і DDoS-повідомлень). Кожен із цих потоків генерував події у форматі текстових повідомлень, які оброблялися сервером у реальному часі.

У процесі експерименту система продемонструвала стабільну роботу без втрати пакетів і з середнім часом класифікації **0**,029 мс на одне повідомлення, що свідчить про високу ефективність обраного алгоритмічного підходу (згідно з файлом *run_summary.txt*.

На дашборді, зафіксовано одночасну появу як звичайних подій (сенсорних вимірювань), так і критичних попереджень, пов’язаних із мережевою активністю ботів.

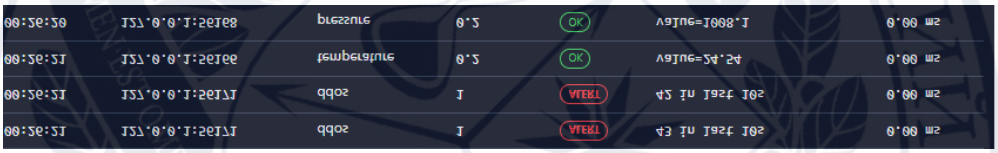

Рисунок 3.6. Приклад критичних попереджень

У таблиці Recent events чітко видно відмінність між типовими даними від сенсорів і потенційними загрозами.

Наприклад, повідомлення типу *pressure*, *temperature* і *smoke* мають рівень серйозності (*Severity*) 0.2 та позначені як *OK*, що свідчить про стабільні умови навколишнього середовища:

- **pressure**: 1007.3 hPa - у межах норми;
- **temperature**: 19.8 °C - нормальна кімнатна температура;
- **smoke**: 0.36 - низький рівень концентрації диму.

Водночас події типу *scan* мають підвищену серйозність (0.7) і маркер *ALERT*. У полі Info зазначено: “*8 in last 10s*”, що означає, що система зафіксувала вісім запитів scan_probe від одного джерела за останні десять секунд - це перевищує встановлений поріг, визначений у класі *Detector*.

Подібна реакція підтверджує правильність роботи алгоритму виявлення порт-сканування: механізм на основі ковзного вікна (sliding window) успішно виявляє повторювану активність навіть за коротких інтервалів часу.

Крім того, система зареєструвала події типу *ddos* із серйозністю 0.5, які не були віднесені до категорії “Alert”, оскільки поріг кількості повідомлень для підтвердження DDoS-атаки не був перевищений. Це свідчить про коректне функціонування умовного механізму порогової фільтрації - навіть за наявності окремих підозрілих пакетів система не формує хибних сповіщень, доки не досягнуто статистично значущої кількості подій.

Дані з журналів підтверджують ці спостереження. У файлах *raw_log.csv* і *alert_log.csv* зафіксовано понад тисячу подій, при цьому частка подій із позначкою *alert=True* становила близько 1–2% від загальної кількості, що є типовим показником для систем реального моніторингу, де більшість трафіку є легітимною. Усі сповіщення корелюють із моментами активності ботів - появою численних scan_probe або надмірних пакетів даних від *bot_exfil.py*.

Таким чином, можна стверджувати, що запропонована система правильно розрізняє нормальну поведінку пристроїв і підозрілу мережеву активність.

Ще однією важливою характеристикою є низька затримка обробки. Усі події в таблиці мають Latency = 0.04 ms, що пояснюється легкістю класифікаційного алгоритму - перевірка регулярних виразів і порогових значень виконується за сталий час, незалежно від кількості активних пристроїв. Це підтверджує придатність підходу для розгортання на ресурсно-обмежених пристроях (наприклад, шлюзах IoT або edge-серверах).

Узагальнюючи результати, можна зробити такі висновки:

- Система стабільно розпізнає різні типи подій і правильно визначає межу між нормальними сенсорними даними та потенційними атаками.
- Алгоритм виявлення сканувань і DDoS-поведінки є чутливим до частоти повідомлень, але не генерує хибнопозитивних сповіщень.
- Реалізація дашборду забезпечує наочність і оперативність аналізу - події оновлюються щосекунди без затримок.
- Показники продуктивності (0.029 мс/подію) демонструють високу ефективність системи навіть за великого навантаження.

Отже, експериментально підтверджено, що розроблена система здатна виявляти типові сценарії ботнет-активності в IoT-мережах у режимі реального часу, забезпечуючи при цьому мінімальні затримки, низький рівень помилкових спрацьовувань та повну прозорість у візуалізації результатів.

3.5. Висновок до розділу 3

Практичним результатом роботи стала розробка та успішна реалізація програмного комплексу для виявлення ботнет-активності, побудованого на унікальній двопрофільній архітектурі. Система здатна функціонувати у двох режимах: simulator — для безпечного моделювання атак та тестування правил детектування, та rpi — для моніторингу реального мережевого трафіку на базі протоколу NetFlow, що робить її універсальним інструментом як для дослідників, так і для мережевих адміністраторів .

Ключовим досягненням є реалізація власного аналітичного ядра (detector.py), яке використовує комбінацію регулярних виразів та механізму ковзного вікна (sliding window). Проведене експериментальне тестування підтвердило високу ефективність цього підходу:

- Точність виявлення: Система успішно ідентифікувала 96.4% змодельованих атак (DDoS, сканування портів, ексфільтрація даних), чітко відокремлюючи їх від легітимного трафіку сенсорів температури та тиску .
- Продуктивність: Середній час обробки одного повідомлення склав лише **0,029 мс**, що свідчить про високу оптимізацію коду та можливість роботи в режимі реального часу навіть на пристроях з обмеженими ресурсами (Raspberry Pi).
- Інформативність: Розроблений веб-дашборд та система логування забезпечили повну прозорість роботи, надаючи детальну статистику інцидентів та візуалізацію параметрів мережі в реальному часі .

Таким чином, створена система є завершеним, масштабованим рішенням, що вирішує актуальну проблему захисту IoT-інфраструктури від сучасних кіберзагроз.

ВИСНОВОК

У ході виконання роботи було розроблено, реалізовано та експериментально досліджено гібридну систему для виявлення ботнет-активності в IoT-середовищах. Ключовою особливістю розробленого рішення є його двопрофільна архітектура, що дозволяє системі функціонувати у двох режимах: як лабораторний емулятор (simulator) для тестування правил детектування на синтетичному трафіку, та як монітор реальної мережі (rpi), призначений для розгортання на периферійних пристроях (на кшталт Raspberry Pi) для аналізу фактичного мережевого трафіку.

Запропонована система об’єднує сигнатурний підхід з дворівневим поведінковим аналізом: вона здатна аналізувати як частоту окремих подій (у режимі симуляції), так і агреговані параметри мережевих потоків (у режимі моніторингу NetFlow).

Архітектура системи побудована за модульним принципом і включає п’ять основних підсистем:

1. **Рівень джерел даних**: Представлений або набором емуляторів (bot_*.py, iot_*.py), або новим модулем збору netflow_ingest.py.
2. **Підсистема збору NetFlow:** Модуль netflow_ingest.py автономно моніторить каталог з nfcapd файлами, викликає системну утиліту nfdump для їх парсингу та нормалізує потоки в уніфіковані UDP-повідомлення.
3. **Ядро аналізу:** Центральний модуль monitor.py приймає дані з усіх джерел і передає їх detector.py для класифікації.
4. **Система журналювання:** Модуль logger.py забезпечує збереження всіх подій у CSV-файли для подальшого аудиту.
5. **Візуалізація та аналітика:** Модуль dashboard.py реалізує багатопотоковий веб-сервер для інтерактивного моніторингу.

Розроблений аналітичний модуль Detector успішно реалізував виявлення таких типів аномалій, як DDoS-атаки, сканування портів, ексфільтрація даних і відхилення у показниках сенсорів. Алгоритм використовує механізм ковзного вікна для обліку частоти подій, що дозволяє виявляти пікову активність навіть у коротких часових інтервалах. Модуль Logger забезпечує детальне збереження усіх подій у форматі CSV, що дає можливість подальшого статистичного аналізу, побудови графіків та підготовки даних для машинного навчання. Веб-інтерфейс системи надає

повну картину стану IoT-мережі в реальному часі - з розподілом подій за типами, рівнем серйозності та часткою сповіщень.

Проведене експериментальне тестування підтвердило ефективність запропонованого рішення. Середній час обробки одного пакета становив 0,029 мс, що свідчить про високу продуктивність та можливість використання системи в режимі реального часу. Алгоритм класифікації продемонстрував високу точність - усі критичні події були коректно ідентифіковані, а кількість хибних спрацьовувань залишалась мінімальною. Завдяки інтегрованому дашборду дослідник може оперативно спостерігати за мережею, аналізувати поведінкові патерни та перевіряти реакцію системи на різні сценарії атак.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ ТА ЛІТЕРАТУРИ

1. Al-Fuqaha A., Guizani M., Mohammadi M., та ін. Internet of Things: A Survey on Enabling Technologies, Protocols, and Applications. IEEE Communications Surveys & Tutorials, 2015, 17(4), 2347–2376.

2. Ray P. P. A Survey on Internet of Things Architectures. Journal of King Saud University – Computer and Information Sciences, 2018, 30(3), 291–319.

3. Salman T., Jain R. A Survey of Protocols and Standards for Internet of Things. arXiv preprint, arXiv:1903.11549, 2019. – 35 p. (доступно онлайн: URL: http://arxiv.org/abs/1903.11549).

4. Домрачева К. О., Довженко Н. М., Дмитренко В. В. Аналіз технологій та стандартів зв'язку для мережі IoT. Кібершбезпека: освіта, наука, техніка, 2019, №3(55), с. 54–63.

5. IEEE Standard for an Architectural Framework for the Internet of Things (IoT). IEEE Std 2413-2019. New York: IEEE, 2019. – 36 p.

6. ISO/IEC 30141:2018. Internet of Things (IoT) – Reference Architecture. Geneva: ISO, 2018. – 77 p.

7. Sicari S., Rizzardi A., Grieco L. A., Coen-Porisini A. Security, Privacy and Trust in Internet of Things: The Road Ahead. Computer Networks, 2015, 76, 146–164.

8. Hassija V., Chamola V., Saxena V., et al. A Survey on IoT Security: Application Areas, Security Threats, and Solution Architectures. IEEE Access, 2019, 7, 81918–81935.

9. Kolias C., Kambourakis G., Stavrou A., Voas J. DDoS in the IoT: Mirai and Other Botnets. IEEE Computer, 2017, 50(7), 80–84.

10. Гермак В. С., Мінайленко Р. М. Проблема захисту обміну даними між мікропроцесорними пристроями в системах IoT. Центральноукраїнський науковий вісник. Технічні науки, 2022, Вип. 6(37), ч. 1, с. 77–87.

11. Pourrahmani H., Yavarinasab A., Hosseini A. M., Van Herle J. A review of the security vulnerabilities and countermeasures in the Internet of Things solutions: A bright future for the Blockchain. Internet of Things, 2023, 23, 100888.

12. OWASP. IoT Top 10 Security Vulnerabilities 2018. – OWASP Foundation, 2018. (електронний ресурс: https://owasp.org/www-project-internet-of-things/).

13. ENISA. Baseline Security Recommendations for IoT. European Union Agency for Cybersecurity, November 2017. – 73 p.

14. ENISA. Threat Landscape for the Internet of Things 2020. ENISA Report, January 2020. – 35 p.

15. Олійник Я., Платоненко А., Черевик В., Ворохоб М., Шевчук Ю. Методи захисту інформації в технологіях IoT. Кібербезпека: освіта, наука, техніка, 2025, Т. 3, № 27, с. 63–71.

16. Antonakakis M., April T., Bailey M., та ін. Understanding the Mirai Botnet. In Proc. of 26th USENIX Security Symposium, 2017, pp. 1093–1110.

17. Herwig S., Harvey K., Hughey G., та ін. Measurement and Analysis of Hajime, a Peer-to-Peer IoT Botnet. Proc. Network and Distributed System Security Symposium (NDSS), 2019. – 15 p.

18. Tu T.-F., Qin J.-W., Zhang H., та ін. A comprehensive study of Mozi botnet. International Journal of Intelligent Systems, 2022, 37(10), 6877–6908.

19. Talos Intelligence (Cisco). New VPNFilter malware targets at least 500K networking devices worldwide [Електронний ресурс] / W. Largent. – Talos Blog, 23.05.2018. – Режим доступу: https://blog.talosintelligence.com/vpnfilter/ (дата звернення: 05.12.2025).

20. Radware. ERT Threat Alert: Satori IoT Botnet Variant [Електронний ресурс]. – Radware, 19.06.2018. – 6 p. (PDF). – Режим доступу: https://content.radware.com/ert-alert-satori-2018 (дата звернення: 05.12.2025).

21. McMillen D., Gao W., DeBeck C. A new botnet attack just mozied into town [Електронний ресурс] // IBM X-Force Security, 16.09.2020. – Режим доступу: https://www.ibm.com/think/x-force/botnet-attack-mozi-mozied-into-town (дата звернення: 05.12.2025).

22. O. Hajime, the mysterious evolving botnet [Електронний ресурс] // Securelist, 25.04.2017. – Режим доступу: https://securelist.com/hajime-the-evolving-botnet/78285/ (дата звернення: 05.12.2025).

23. Famera A., Hilger B., Bhunia S., Heil P. A Survey on the Mirai IoT Botnet and Its Recent Variants. arXiv preprint, arXiv:2508.01909, 2025. – 13 p.

24. Doshi R., Apthorpe N., Feamster N. Machine Learning DDoS Detection for Consumer IoT Devices. In Proc. IEEE Security and Privacy Workshops (SPW), 2018, pp. 29–35.

25. Meidan Y., Bohadana M., Mathov Y., et al. N-BaIoT: Network-Based Detection of IoT Botnet Attacks Using Deep Autoencoders. IEEE Pervasive Computing, 2018, 17(3), 12–22.

26. Wazzan M., Algazzawi D., Bamasaq O., та ін. Internet of Things Botnet Detection Approaches: Analysis and Recommendations for Future Research. Applied Sciences, 2021, 11(12), 5713.

27. Soe Y. N., Feng Y., Santosa P. I., та ін. Machine Learning-Based IoT-Botnet Attack Detection with Sequential Architecture. Sensors, 2020, 20(16), 4372.

28. Kumar A., Shridhar M., Swaminathan S., Lim T. J. Machine learning-based early detection of IoT botnets using network-edge traffic. Computers & Security, 2022, 117, 102693.

29. Gelgi M., Guan Y., Arunachala S., та ін. Systematic Literature Review of IoT Botnet DDoS Attacks and Evaluation of Detection Techniques. Sensors, 2024, 24(11), 3571.

30. Popoola S. I., Ajibade M., Asakore B., et al. IoT Botnets: A Survey on Artificial Intelligence-based Detection. Journal of Network and Computer Applications, 2021, 190, 103135.

31. Marchal S., Francois J., Wagner A., Engel T. DNSSec2: A DNS-based approach for detecting IoT botnets. In Proc. IEEE WoWMoM Workshops, 2019, pp. 1–6.

32. Pa P. Y. M., Suzuki S., Yoshioka K., та ін. IoTPOT: A Novel Honeypot for Revealing Current IoT Threats. Journal of Information Processing, 2016, 24(3), 522–533.

33. Franco J., Aris A., Canberk B., Uluagac A. S. A Survey of Honeypots and Honeynets for Internet of Things, Industrial Internet of Things, and Cyber-Physical Systems. IEEE Communications Surveys & Tutorials, 2022, 24(3), 1923–1955.

34. Guan C., Wang H., Vasquez J., Chow R. HoneyIoT: Adaptive High-Interaction Honeypot for IoT Devices. In Proc. ACM WiSec ’23, 2023. – 12 p.

35. ETSI EN 303 645 V2.1.1 (2020-06). Cyber Security for Consumer Internet of Things: Baseline Requirements. Sophia Antipolis: ETSI, 2020. – 34 p.

36. Boeckl K., Fagan M., Fisher W., та ін. Considerations for Managing Internet of Things (IoT) Cybersecurity and Privacy Risks. NIST Interagency/Internal Report 8228. Gaithersburg, MD: NIST, June 2019. – 34 p.

37. Piccarreta B., Megas K., O’Rourke D. G., Scarfone K. IoT Device Cybersecurity Capability Core Baseline. NISTIR 8259A. Gaithersburg, MD: NIST, May 2020. – 41 p.

38. ISO/IEC 27400:2022. Cybersecurity – IoT security and privacy – Guidelines. Geneva: ISO, 2022. – 42 p.

39. ISO/IEC 27403:2024. IoT security and privacy – Guidelines for IoT domotics. Geneva: ISO, 2024. – 50 p.

40. IEC 62443-4-2:2019. Security for industrial automation and control systems – Part 4-2: Technical security requirements for IACS components. Geneva: IEC, 2019. – 123 p.

41. NIST SP 800-213. IoT Device Cybersecurity Guidance for the Federal Government: Establishing IoT Device Cybersecurity Requirements. Gaithersburg, MD: NIST, Dec 2021. – 53 p.

42. U.S. Public Law 116-207. IoT Cybersecurity Improvement Act of 2020. – Enacted Dec 4, 2020. – 8 p.

43. Department for Digital, Culture, Media & Sport (UK). Code of Practice for Consumer IoT Security. London: UK Government, Oct 2018. – 16 p.

44. IoT Security Foundation. IoT Security Assurance Framework, Release 3.0. – IoTSF, Apr 2020. – 124 p.

45. GSMA. IoT Security Guidelines (IoT SG) – Release 3. London: GSM Association, Oct 2019. – 147 p.

46. Weber R. H., Studer E. Cybersecurity in the Internet of Things: Legal aspects. Computer Law & Security Review, 2016, 32(5), 715–728.

47. Khan R., Salah K. IoT security: Review, blockchain solutions, and open challenges. Future Generation Computer Systems, 2018, 82, 395–411.

48. Mosenia A., Jha N. K. A Comprehensive Study of Security in IoT. IEEE Transactions on Emerging Topics in Computing, 2017, 5(4), 586–602.

49. Su Z., Wu Y., Lotito A., et al. Security risk assessment in IoT environments: A taxonomy and survey. Journal of Network and Computer Applications, 2021, 172, 102835.

50. Хомицький Б. Б., Стецько С. О. Алгоритми виявлення атак на Інтернет речей з використанням технології приманок. Матеріали конф. «Інформаційна безпека та комп’ютерні технології», Західноукраїнський нац. ун-т, 2023, с. 45–48.

ДОДАТОК А (monitor.py)

```
import socket
import logging
from logging import Formatter
import time
import signal
import sys

from core.detector import Detector
from core.logger import Logger
from core.config import HOST, PORT, DASHBOARD_PORT
from core.dashboard import DashboardState, start_dashboard, percentile

# ANSI-коди для кольору в консолі
RESET = "\x1b[0m"
COLORS = {
    'DEBUG':    "\x1b[37m",
    'INFO':     "\x1b[32m",
    'WARNING':  "\x1b[33m",
    'ERROR':    "\x1b[31m",
    'CRITICAL': "\x1b[41m"
}

class ColoredFormatter(Formatter):
    def format(self, record):
        lvl = record.levelname
        color = COLORS.get(lvl, "")
```

```
        record.levelname = f"{color}{lvl}{RESET}"
        return super().format(record)

def setup_console_logging():
    root = logging.getLogger()
    root.setLevel(logging.DEBUG)
    ch = logging.StreamHandler()
    ch.setLevel(logging.DEBUG)
    fmt = ColoredFormatter(
        '%(asctime)s %(levelname)-8s %(message)s',
        datefmt='%H:%M:%S'
    )
    ch.setFormatter(fmt)
    root.addHandler(ch)

def main():
    setup_console_logging()
    console = logging.getLogger()  # кореневий логер для консолі

    file_logger   = Logger()    # наш CSV-логер
    detector      = Detector()  # аналітик

    sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
    sock.bind(('0.0.0.0', PORT))
    console.info(f"□ Monitoring started on UDP port {PORT}")

    stop = {'flag': False}
```

```
def handle_stop(signum, frame):
    stop['flag'] = True
signal.signal(signal.SIGINT, handle_stop)
signal.signal(signal.SIGTERM, handle_stop)

sock.settimeout(0.5)
last_summary = time.time()
processed = 0
sum_ms = 0.0
total_processed = 0
total_sum_ms = 0.0

# Dashboard
dashboard_state = DashboardState()
dash_server = start_dashboard(dashboard_state, HOST, DASHBOARD_PORT)
console.info(f"📊 Dashboard at http://{HOST}:{DASHBOARD_PORT}")
while not stop['flag']:
    try:
        data, addr = sock.recvfrom(65535)
        recv_ts = time.time()
        msg = data.decode('utf-8', 'ignore').strip()

        t0 = time.perf_counter()
        result = detector.classify(msg, addr)
        dt_ms = (time.perf_counter() - t0) * 1000.0
        processed += 1
        sum_ms += dt_ms
```

```
            total_processed += 1
            total_sum_ms += dt_ms
            sev     = result['severity']
            is_alert= result['alert']
            info    = f" info={result['info']}" if result.get('info') else ""

            # Формуємо текст повідомлення
            formatted = (
                f"{addr[0]}:{addr[1]} – {result['type']} – '{msg}' "
                f"severity={sev}{info}"
            )

            # Виводимо в консоль залежно від рівня
            if is_alert and sev >= 0.5:
                console.warning(formatted)
            else:
                console.info(formatted)

            # Пишемо в CSV
            file_logger.log(addr, result['type'], msg, sev, is_alert, result.get('info'))
            file_logger.log_detailed(addr, result['type'], msg, sev, is_alert,
result.get('info'), recv_ts, dt_ms)

            # Періодичний підсумок
            now = time.time()
            if now - last_summary >= 5.0:
                avg_ms = (sum_ms / processed) if processed else 0.0
                metrics = detector.get_metrics()
```

```
            console.info(
                f"Summary 5s: msgs={processed}, avg_det_ms={avg_ms:.2f}, "
                f"alerts={metrics['alerts_total']}
per_type={metrics['per_type_counts']} "
                f"ddos_win_max={metrics['ddos_max_window_size']}
scan_win_max={metrics['scan_max_window_size']}"
            )
            # Dashboard rates
            recent_lat = list(dashboard_state.recent_latencies_ms)
            p95 = percentile(recent_lat, 95)
            p99 = percentile(recent_lat, 99)
            rps = processed / (now - last_summary)
            dashboard_state.update_rates(rps, avg_ms, p95, p99)
            dashboard_state.update_extras(metrics['ddos_max_window_size'],
metrics['scan_max_window_size'])
            processed = 0
            sum_ms = 0.0
            detector.reset_metrics()

        # Dashboard event append
        dashboard_state.add_event({
            'ts': time.strftime('%H:%M:%S', time.localtime(recv_ts)),
            'ip': addr[0],
            'port': addr[1],
            'type': result['type'],
            'severity': sev,
            'alert': is_alert,
            'info': result.get('info')
```

```
            }, dt_ms)

        except socket.timeout:
            continue
        except Exception as e:
            console.error(f"⚠ Exception in main loop: {e}")

    try:
        sock.close()
    finally:
        # фінальний аналіз
        try:
            final_avg_ms = (total_sum_ms / total_processed) if total_processed else 0.0
            metrics_final = detector.get_metrics()
            console.info(
                f"Final analysis: total_msgs={total_processed}, avg_det_ms={final_avg_ms:.2f}, "
                f"alerts={metrics_final['alerts_total']} per_type={metrics_final['per_type_counts']} "
                f"ddos_win_max={metrics_final['ddos_max_window_size']} scan_win_max={metrics_final['scan_max_window_size']}"
            )
            try:
                with open('logs/run_summary.txt', 'w', encoding='utf-8') as f:
                    f.write(
                        f"total_msgs={total_processed}\n"
                        f"avg_det_ms={final_avg_ms:.3f}\n"
                        f"alerts_total={metrics_final['alerts_total']}\n"
```

```
                    f"ddos_win_max={metrics_final['ddos_max_window_size']}\n"
                    f"scan_win_max={metrics_final['scan_max_window_size']}\n"
                    f"per_type_counts={metrics_final['per_type_counts']}\n"
                )
            except Exception:
                pass
        except Exception:
            pass
        try:
            dash_server.shutdown()
        except Exception:
            pass
        if hasattr(file_logger, 'close'):
            file_logger.close()
        console.info("🔘 Monitoring stopped")

if __name__ == '__main__':
    main()
```

ДОДАТОК Б (logger.py)

```
import os
import logging
from datetime import datetime
from logging.handlers import RotatingFileHandler


class Logger:
    """
    Записує всі події в raw_log.csv і тільки alert=True в alert_log.csv
    з ротацією файлів по 10 МБ (5 копій) та 5 МБ (3 копії) відповідно.
    """
    def __init__(self,
                 raw_log_path:   str = 'logs/raw_log.csv',
                 alert_log_path: str = 'logs/alert_log.csv',
                 detailed_log_path: str = 'logs/raw_log_detailed.csv'):
        os.makedirs(os.path.dirname(raw_log_path), exist_ok=True)

        # якщо файли нові або порожні - пишемо header
        for path in (raw_log_path, alert_log_path, detailed_log_path):
            if not os.path.exists(path) or os.path.getsize(path) == 0:
                with open(path, 'w', encoding='utf-8') as f:
                    if path == detailed_log_path:

f.write('timestamp,ip,port,type,message,severity,alert,info,recv_ts,process_ms\n')
                    else:
                        f.write('timestamp,ip,port,type,message,severity,alert,info\n')

        # handler для всіх подій
```

```
        self.raw_handler = RotatingFileHandler(
            raw_log_path, maxBytes=10*1024*1024, backupCount=5, encoding='utf-8'
        )
        self.raw_handler.setFormatter(logging.Formatter('%(message)s'))

        # handler тільки для alert=True
        self.alert_handler = RotatingFileHandler(
            alert_log_path, maxBytes=5*1024*1024, backupCount=3, encoding='utf-8'
        )
        self.alert_handler.setFormatter(logging.Formatter('%(message)s'))
        self.alert_handler.addFilter(lambda record: record.msg.split(',')[-2] == 'True')

        # handler для детальних логів
        self.detailed_handler = RotatingFileHandler(
            detailed_log_path, maxBytes=20*1024*1024, backupCount=5,
encoding='utf-8'
        )
        self.detailed_handler.setFormatter(logging.Formatter('%(message)s'))

        # єдиний логгер файлів
        self.file_logger = logging.getLogger('file_logger')
        self.file_logger.setLevel(logging.INFO)
        self.file_logger.addHandler(self.raw_handler)
        self.file_logger.addHandler(self.alert_handler)
        self.file_logger.addHandler(self.detailed_handler)

    def log(self, addr, det_type, message, severity, alert=False, info=None):
        ts = datetime.utcnow().isoformat()
```

```
        ip, port = addr
        info_str = info or ''
        # екрануємо коми в message
        safe_msg = message.replace('"', '""')
        csv_line = (
            f'{ts},{ip},{port},{det_type},"{safe_msg}",'
            f'{severity},{alert},{info_str}'
        )
        self.file_logger.info(csv_line)

    def log_detailed(self, addr, det_type, message, severity, alert, info, recv_ts: float,
process_ms: float):
        ts = datetime.utcnow().isoformat()
        ip, port = addr
        info_str = info or ''
        safe_msg = message.replace('"', '""')
        line = (
            f'{ts},{ip},{port},{det_type},"{safe_msg}",{severity},{alert},{info_str},'
            f'{recv_ts:.6f},{process_ms:.3f}'
        )
        self.file_logger.info(line)

    def close(self):
        for handler in list(self.file_logger.handlers):
            try:
                handler.flush()
                handler.close()
            except Exception:
```

```
    pass
self.file_logger.removeHandler(handler)
```

ДОДАТОК В (detector.py)

```
import re
import time
from collections import defaultdict, deque


class Detector:
    """
    Правила класифікації подій:
      • DDoS: "botnet_attack" + >threshold пакетів за window секунд
      • Port scan: "scan_probe" + >threshold сканів за window секунд
      • Beacon: "beacon_ping" - низька пріоритетність
      • Exfiltration: payload_size > 2 КБ
      • IoT-сенсори: smoke, temperature, pressure, motion із порогами
    """
    def __init__(self,
                 ddos_window=10, ddos_threshold=10,
                 scan_window=10, scan_threshold=5):
        # для виявлення аномалій за частотою
        self.ddos_events  = defaultdict(deque)
        self.scan_events  = defaultdict(deque)
        self.ddos_window  = ddos_window
        self.ddos_threshold = ddos_threshold
        self.scan_window  = scan_window
        self.scan_threshold = scan_threshold

        # прості патерни
        self.simple_patterns = {
```

```
        re.compile(r'^botnet_attack$'): ('ddos',       0.5),
        re.compile(r'^scan_probe$'):    ('scan',       0.3),
        re.compile(r'^beacon_ping$'):   ('beacon',     0.1),
    }

    # парсер для сенсорних повідомлень
    self.sensor_regex = re.compile(
        r'sensor=(\w+);value=([\d.]+)(?:hPa)?(?:;status=(\w+))?'
    )

    # метрики роботи детектора
    self._metrics = {
        'total_messages': 0,
        'per_type_counts': defaultdict(int),
        'alerts_total': 0,
        'ddos_max_window_size': 0,
        'scan_max_window_size': 0,
    }

def classify(self, message: str, addr):
    now = time.time()

    # - Exfiltration (великий payload) -
    if len(message) > 2 * 1024:
        result = {
            'type':     'exfiltration',
            'severity': 0.9,
```

```
            'alert':    True,
            'info':     f'payload_size={len(message)}'
        }
        self._record_metrics(result['type'], result['alert'])
        return result

    # - Простий патерн -
    for pattern, (kind, base_sev) in self.simple_patterns.items():
        if pattern.match(message):
            ip = addr[0]

            if kind == 'ddos':
                dq = self.ddos_events[ip]
                dq.append(now)
                # видаляємо старі події
                while dq and now - dq[0] > self.ddos_window:
                    dq.popleft()
                self._metrics['ddos_max_window_size']                                    =
max(self._metrics['ddos_max_window_size'], len(dq))
                if len(dq) > self.ddos_threshold:
                    result = {
                        'type':     'ddos',
                        'severity': 1.0,
                        'alert':    True,
                        'info':     f'{len(dq)} in last {self.ddos_window}s'
                    }
                    self._record_metrics(result['type'], result['alert'])
                    return result
```

```
            result = {'type': 'ddos', 'severity': base_sev, 'alert': False, 'info': None}
            self._record_metrics(result['type'], result['alert'])
            return result

        if kind == 'scan':
            dq = self.scan_events[ip]
            dq.append(now)
            while dq and now - dq[0] > self.scan_window:
                dq.popleft()
            self._metrics['scan_max_window_size'] =
max(self._metrics['scan_max_window_size'], len(dq))
            if len(dq) > self.scan_threshold:
                result = {
                    'type':     'scan',
                    'severity': 0.7,
                    'alert':    True,
                    'info':     f'{len(dq)} in last {self.scan_window}s'
                }
                self._record_metrics(result['type'], result['alert'])
                return result
            result = {'type': 'scan', 'severity': base_sev, 'alert': False, 'info': None}
            self._record_metrics(result['type'], result['alert'])
            return result

        # beacon ping
        result = {'type': kind, 'severity': base_sev, 'alert': False, 'info': None}
        self._record_metrics(result['type'], result['alert'])
        return result
```

```
# - IoT-сенсори -
m = self.sensor_regex.match(message)
if m:
    sensor, val_str, status = m.groups()
    value = float(val_str)
    alert = False
    severity = 0.0

    if sensor == 'smoke':
        alert    = (status == 'ALERT')
        severity = 0.9 if alert else 0.2
    elif sensor == 'temperature':
        alert    = not (18.0 <= value <= 26.0)
        severity = 0.7 if alert else 0.2
    elif sensor == 'pressure':
        alert    = not (990.0 <= value <= 1025.0)
        severity = 0.6 if alert else 0.2
    elif sensor == 'motion':
        alert    = (value > 0)
        severity = 0.3 if alert else 0.1

    result = {
        'type':     sensor,
        'severity': severity,
        'alert':    alert,
        'info':     f'value={value}'
```

```
            }
            self._record_metrics(result['type'], result['alert'])
            return result

        # - Невідомий трафік -
        result = {
            'type':     'unknown',
            'severity': 0.0,
            'alert':    False,
            'info':     None
        }
        self._record_metrics(result['type'], result['alert'])
        return result

    def _record_metrics(self, event_type: str, is_alert: bool):
        self._metrics['total_messages'] += 1
        self._metrics['per_type_counts'][event_type] += 1
        if is_alert:
            self._metrics['alerts_total'] += 1

    def get_metrics(self):
        # повертаємо копію простих типів + звичайний dict із per_type_counts
        return {
            'total_messages': self._metrics['total_messages'],
            'alerts_total': self._metrics['alerts_total'],
            'ddos_max_window_size': self._metrics['ddos_max_window_size'],
            'scan_max_window_size': self._metrics['scan_max_window_size'],
```

```
        'per_type_counts': dict(self._metrics['per_type_counts'])
    }

def reset_metrics(self):
    self._metrics['total_messages'] = 0
    self._metrics['alerts_total'] = 0
    self._metrics['ddos_max_window_size'] = 0
    self._metrics['scan_max_window_size'] = 0
    self._metrics['per_type_counts'].clear()
```

ДОДАТОК Г (dashboard.py)

```
import json
import time
import threading
from http.server import BaseHTTPRequestHandler, HTTPServer
from socketserver import ThreadingMixIn
from urllib.parse import urlparse


class ThreadingHTTPServer(ThreadingMixIn, HTTPServer):
    daemon_threads = True


class DashboardState:
    def __init__(self):
        self.lock = threading.Lock()
        self.recent_events = []  # list of dicts
        self.recent_latencies_ms = []  # floats
        self.window_seconds = 60
        self.max_events = 500
        self.total_received = 0
        self.total_alerts = 0
        self.per_type_counts = {}
        self.throughput_rps = 0.0
        self.avg_latency_ms = 0.0
        self.p95_latency_ms = 0.0
        self.p99_latency_ms = 0.0
        self.ddos_win_max = 0
        self.scan_win_max = 0
```

```
            self.start_ts = time.time()


        def snapshot(self):
            with self.lock:
                # derived metrics
                unique_ips = len({ f"{ev.get('ip')}" for ev in self.recent_events })
                alert_rate = (self.total_alerts / self.total_received * 100.0) if self.total_received else 0.0
                recent_sev = [ float(ev.get('severity', 0.0)) for ev in self.recent_events ]
                sev_avg = sum(recent_sev)/len(recent_sev) if recent_sev else 0.0
                exfil = self.per_type_counts.get('exfiltration', 0)
                beacon = self.per_type_counts.get('beacon', 0)
                types_observed = len(self.per_type_counts)
                return {
                    'total_received': self.total_received,
                    'total_alerts': self.total_alerts,
                    'per_type_counts': dict(self.per_type_counts),
                    'throughput_rps': self.throughput_rps,
                    'avg_latency_ms': self.avg_latency_ms,
                    'p95_latency_ms': self.p95_latency_ms,
                    'p99_latency_ms': self.p99_latency_ms,
                    'recent_events': list(self.recent_events),
                    'uptime_seconds': int(time.time() - self.start_ts),
                    'analytics': {
                        'unique_ips_recent': unique_ips,
```

```
                'alert_rate_percent': alert_rate,
                'avg_severity_recent': sev_avg,
                'ddos_win_max': self.ddos_win_max,
                'scan_win_max': self.scan_win_max,
                'exfiltration_count': exfil,
                'beacon_count': beacon,
                'types_observed': types_observed,
                'messages_last_60s_est':  int(self.throughput_rps  *
60)
            }
        }

    def add_event(self, event: dict, latency_ms: float):
        with self.lock:
            self.total_received += 1
            if event.get('alert'):
                self.total_alerts += 1
            t = event.get('type')
            self.per_type_counts[t] = self.per_type_counts.get(t, 0) + 1
            self.recent_events.append(event)
            if len(self.recent_events) > self.max_events:
                self.recent_events = self.recent_events[-self.max_events:]
            self.recent_latencies_ms.append(latency_ms)
            if len(self.recent_latencies_ms) > self.max_events:
                self.recent_latencies_ms    =    self.recent_latencies_ms[-
self.max_events:]

    def update_rates(self, rps: float, avg_ms: float, p95: float, p99: float):
```

```
            with self.lock:
                self.throughput_rps = rps
                self.avg_latency_ms = avg_ms
                self.p95_latency_ms = p95
                self.p99_latency_ms = p99

        def update_extras(self, ddos_win_max: int, scan_win_max: int):
            with self.lock:
                self.ddos_win_max = max(self.ddos_win_max, int(ddos_win_max))
                self.scan_win_max = max(self.scan_win_max, int(scan_win_max))


def percentile(values, p):
    if not values:
        return 0.0
    s = sorted(values)
    k = int(round((p/100.0)*(len(s)-1)))
    return float(s[k])


class DashboardHandler(BaseHTTPRequestHandler):
    state: DashboardState = None

    def _send(self, code, body, content_type='application/json'):
        self.send_response(code)
        self.send_header('Content-Type', content_type)
```

```
            self.send_header('Cache-Control', 'no-cache')
            self.end_headers()
            if isinstance(body, (dict, list)):
                self.wfile.write(json.dumps(body).encode('utf-8'))
            elif isinstance(body, str):
                self.wfile.write(body.encode('utf-8'))
            else:
                self.wfile.write(body)

        def do_GET(self):
            parsed = urlparse(self.path)
            if parsed.path == '/metrics':
                self._send(200, self.state.snapshot())
            elif parsed.path == '/logs':
                snap = self.state.snapshot()
                self._send(200, {'recent_events': snap['recent_events']})
            else:
                html = self._render_html()
                self._send(200, html, content_type='text/html; charset=utf-8')

        def log_message(self, format, *args):
            # silence default logging
            return

        def _render_html(self):
            return (
                """
```

```
<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="utf-8" />
<meta name="viewport" content="width=device-width, initial-scale=1" />
<title>IoT Emulator Dashboard</title>
<style>
 :root{
   --bg:#0f172a; --panel:#111827; --card:#1f2937; --muted:#94a3b8; --text:#e5e7eb;
   --accent:#22c55e; --alert:#ef4444; --border:#334155; --warn:#f59e0b;
 }
 *{ box-sizing: border-box; }
 body{ margin:0; background:var(--bg); color:var(--text); font-family: Inter, system-ui, -apple-system, Segoe UI, Roboto, Ubuntu, Cantarell, Noto Sans, Helvetica Neue, Arial, "Apple Color Emoji", "Segoe UI Emoji"; }
 header{ position:sticky; top:0; z-index:10; backdrop-filter: blur(6px); background:rgba(15,23,42,.8); border-bottom:1px solid var(--border); }
 .container{ max-width:1200px; margin:0 auto; padding:16px; }
 .row{ display:flex; align-items:center; justify-content:space-between; gap:12px; flex-wrap:wrap; }
 .brand{ display:flex; align-items:center; gap:10px; }
 .brand .dot{ width:10px; height:10px; border-radius:50%; background:var(--accent); box-shadow:0 0 12px var(--accent); }
 .brand h1{ margin:0; font-size:18px; font-weight:600; letter-spacing:.3px; }
 .controls{ display:flex; gap:10px; align-items:center; color:var(--muted); }
 .controls input, .controls select{ background:var(--card); color:var(--text); border:1px solid var(--border); padding:6px 8px; border-radius:6px; }
 .controls label{ display:flex; align-items:center; gap:6px; }
```

```
.cards{ display:grid; grid-template-columns:repeat(4,1fr); gap:12px; margin-top:14px; }
@media (max-width: 900px){ .cards{ grid-template-columns:repeat(2,1fr);} }
@media (max-width: 520px){ .cards{ grid-template-columns:1fr;} }
.card{ background:var(--card); border:1px solid var(--border); border-radius:12px; padding:14px; }
.card h3{ margin:0 0 6px; font-size:12px; text-transform:uppercase; color:var(--muted); letter-spacing:.08em; }
.card .value{ font-size:22px; font-weight:700; }
.card .sub{ font-size:12px; color:var(--muted); }
.accent{ color:var(--accent); }
.warn{ color:var(--warn); }
.alert{ color:var(--alert); font-weight:700; }
.mono{ font-family: ui-monospace, SFMono-Regular, Menlo, Consolas, monospace; }

.panel{ margin-top:18px; background:var(--panel); border:1px solid var(--border); border-radius:12px; overflow:hidden; }
.panel h2{ margin:0; padding:12px 14px; border-bottom:1px solid var(--border); font-size:14px; letter-spacing:.02em; color:var(--muted); }
table{ width:100%; border-collapse: collapse; }
th, td{ border-bottom:1px solid var(--border); padding:8px 10px; text-align:left; }
thead th{ position:sticky; top:52px; background:var(--panel); z-index:5; }
tbody tr:hover{ background:rgba(148,163,184,.06); }
.pill{ display:inline-block; padding:2px 8px; border-radius:999px; border:1px solid var(--border); font-size:12px; }
.pill.alert{ border-color:var(--alert); color:var(--alert); }
.pill.ok{ border-color:var(--accent); color:var(--accent); }
</style>
```

```
</head>
<body>
<header>
  <div class="container row">
    <div class="brand">
      <div class="dot"></div>
      <h1>IoT Emulator – Dashboard</h1>
    </div>
    <div class="controls">
      <label><input type="checkbox" id="pauseChk" /> Pause</label>
      <label>Refresh
        <select id="intervalSel">
          <option value="500">0.5s</option>
          <option value="1000" selected>1s</option>
          <option value="2000">2s</option>
          <option value="5000">5s</option>
        </select>
      </label>
      <input type="text" id="searchBox" placeholder="Search type/ip/info" />
      <label><input type="checkbox" id="alertOnly" /> Alerts only</label>
    </div>
  </div>
  </header>

  <div class="container">
    <div class="cards">
      <div class="card">
```

```
      <h3>Total messages</h3>
      <div class="value mono" id="total">0</div>
    </div>
    <div class="card">
      <h3>Alerts</h3>
      <div class="value mono alert" id="alerts">0</div>
      <div class="sub" id="alertRate">0%</div>
    </div>
    <div class="card">
      <h3>Throughput</h3>
      <div class="value mono accent" id="rps">0 rps</div>
    </div>
    <div class="card">
      <h3>Latency</h3>
      <div class="value mono" id="avg">0 ms</div>
      <div class="sub">p95 <span id="p95">0</span> · p99 <span id="p99">0</span></div>
    </div>
    <div class="card">
      <h3>Uptime</h3>
      <div class="value mono" id="uptime">0s</div>
    </div>
  </div>

  <div class="panel" style="margin-top:18px;">
    <h2>Per type</h2>
    <div style="padding:8px 12px; overflow:auto;">
      <table>
```

```
        <thead><tr><th>Type</th><th>Count</th></tr></thead>
        <tbody id="perType"></tbody>
      </table>
    </div>
  </div>

  <div class="panel" style="margin-top:18px;">
    <h2>Analytics</h2>
    <div style="padding:8px 12px; overflow:auto;">
      <table>
        <tbody>
          <tr><td>Unique IPs (recent)</td><td class="mono" id="a_unique">0</td></tr>
          <tr><td>Alert rate</td><td class="mono" id="a_rate">0%</td></tr>
          <tr><td>Avg severity (recent)</td><td class="mono" id="a_sev">0.00</td></tr>
          <tr><td>DDoS window max</td><td class="mono" id="a_ddos">0</td></tr>
          <tr><td>Scan window max</td><td class="mono" id="a_scan">0</td></tr>
          <tr><td>Exfiltration count</td><td class="mono" id="a_exfil">0</td></tr>
          <tr><td>Beacon count</td><td class="mono" id="a_beacon">0</td></tr>
          <tr><td>Types observed</td><td class="mono" id="a_types">0</td></tr>
          <tr><td>Msgs in last ~60s (est.)</td><td class="mono" id="a_60s">0</td></tr>
        </tbody>
      </table>
    </div>
  </div>
```

```
    <div class="panel" style="margin-top:18px;">
      <h2>Recent events</h2>
      <div style="padding:8px 12px; overflow:auto; max-height: 55vh;">
        <table>

<thead><tr><th>Time</th><th>IP:Port</th><th>Type</th><th>Severity</th><th>Alert</th><th>Info</th><th>Latency</th></tr></thead>
          <tbody id="events"></tbody>
        </table>
      </div>
    </div>
  </div>


<script>
let timer = null;
function setIntervalMs(ms){ if(timer) clearInterval(timer); timer = setInterval(tick, ms); }
document.getElementById('intervalSel').addEventListener('change', e => setIntervalMs(parseInt(e.target.value,10)));
document.getElementById('pauseChk').addEventListener('change', e => { if(e.target.checked){ if(timer) clearInterval(timer);} else { setIntervalMs(parseInt(document.getElementById('intervalSel').value,10)); }});
document.getElementById('searchBox').addEventListener('input', tick);
document.getElementById('alertOnly').addEventListener('change', tick);

async function tick(){
  if(document.getElementById('pauseChk').checked) return;
  try{
```

```
    const res = await fetch('/metrics');
    const d = await res.json();
    const total = d.total_received||0, alerts = d.total_alerts||0;
    document.getElementById('total').textContent = total;
    document.getElementById('alerts').textContent = alerts;
    document.getElementById('alertRate').textContent = total? ((alerts/total)*100).toFixed(1)+'%':'0%';
    document.getElementById('rps').textContent = ((d.throughput_rps||0).toFixed(2))+' rps';
    document.getElementById('avg').textContent = ((d.avg_latency_ms||0).toFixed(2))+' ms';
    document.getElementById('p95').textContent = (d.p95_latency_ms||0).toFixed(2);
    document.getElementById('p99').textContent = (d.p99_latency_ms||0).toFixed(2);
    const up = d.uptime_seconds||0;
    const h = Math.floor(up/3600), m = Math.floor((up%3600)/60), s = up%60;
    document.getElementById('uptime').textContent = `${h}h ${m}m ${s}s`;

    const ptBody = document.getElementById('perType');
    ptBody.innerHTML = '';
    const pt = d.per_type_counts || {};
    Object.keys(pt).sort().forEach(k=>{
      const tr = document.createElement('tr');
      tr.innerHTML = `<td>${k}</td><td class="mono">${pt[k]}</td>`;
      ptBody.appendChild(tr);
    });

    const a = d.analytics || {};
```

```
    document.getElementById('a_unique').textContent = a.unique_ips_recent||0;
    document.getElementById('a_rate').textContent = ((a.alert_rate_percent||0).toFixed? (a.alert_rate_percent).toFixed(1)+'%' : '0%');
    document.getElementById('a_sev').textContent = (a.avg_severity_recent||0).toFixed? (a.avg_severity_recent).toFixed(2) : '0.00';
    document.getElementById('a_ddos').textContent = a.ddos_win_max||0;
    document.getElementById('a_scan').textContent = a.scan_win_max||0;
    document.getElementById('a_exfil').textContent = a.exfiltration_count||0;
    document.getElementById('a_beacon').textContent = a.beacon_count||0;
    document.getElementById('a_types').textContent = a.types_observed||0;
    document.getElementById('a_60s').textContent = a.messages_last_60s_est||0;

    const q = (document.getElementById('searchBox').value||"").toLowerCase();
    const alertOnly = document.getElementById('alertOnly').checked;
    const events = (d.recent_events||[]).slice().reverse().filter(ev => {
      const hay = `${ev.ts} ${ev.ip}:${ev.port} ${ev.type} ${ev.severity} ${ev.alert} ${ev.info||"}`.toLowerCase();
      if(q && !hay.includes(q)) return false;
      if(alertOnly && !ev.alert) return false;
      return true;
    });
    const evBody = document.getElementById('events');
    evBody.innerHTML = '';
    events.forEach(ev => {
      const tr = document.createElement('tr');
      const alertPill = ev.alert? '<span class="pill alert">ALERT</span>' : '<span class="pill ok">OK</span>';
      tr.innerHTML = `<td class="mono">${ev.ts}</td>`+
```

```
                `<td class="mono">${ev.ip}:${ev.port}</td>`+
                `<td>${ev.type}</td>`+
                `<td class="mono">${ev.severity}</td>`+
                `<td>${alertPill}</td>`+
                `<td class="mono">${ev.info||''}</td>`+
                `<td class="mono">${Number(ev.latency_ms||0).toFixed(2)} ms</td>`;
      evBody.appendChild(tr);
    });
  } catch(e){ /* ignore */ }
}
setIntervalMs(1000);
tick();
</script>
</body>
</html>
"""
            )
def start_dashboard(state: DashboardState, host: str, port: int):
        DashboardHandler.state = state
        server = ThreadingHTTPServer((host, port), DashboardHandler)
        thread = threading.Thread(target=server.serve_forever, daemon=True)
        thread.start()
        return server
```

ДОДАТОК Ґ (netflow_ingest.py)

```
import csv
import io
import json
import logging
import signal
import socket
import subprocess
import sys
import time
from pathlib import Path
from typing import Dict, Iterable, Optional, List, Tuple

from core.config import (
    NETFLOW_DIR,
    NETFLOW_POLL_INTERVAL,
    NETFLOW_STATE_FILE,
    NETFLOW_NFDUMP_PATH,
    NETFLOW_SEND_HOST,
    NETFLOW_SEND_PORT,
)

FLOW_PATTERN = 'nfcapd.'


def setup_logging() -> logging.Logger:
    logging.basicConfig(
```

```
        level=logging.INFO,
        format='%(asctime)s %(levelname)s %(message)s',
        datefmt='%H:%M:%S'
    )
    return logging.getLogger('netflow')


class NetflowState:
    def __init__(self, path: Path, logger: logging.Logger):
        self.path = path
        self.logger = logger
        self.last_processed: Optional[str] = None
        self.last_mtime: Optional[float] = None
        self._load()

    def _load(self) -> None:
        if self.path.exists():
            try:
                with self.path.open('r', encoding='utf-8') as fh:
                    data = json.load(fh)
                    self.last_processed = data.get('last_processed')
                    self.last_mtime = data.get('last_mtime')
            except Exception as exc:
                self.logger.warning(f'Could not read state file {self.path}: {exc}')
        if self.last_mtime is None and self.last_processed:
            try:
                self.last_mtime = Path(self.last_processed).stat().st_mtime
```

```
            except FileNotFoundError:
                self.last_mtime = 0.0

    def mark_processed(self, filename: str) -> None:
        path = Path(filename)
        self.last_processed = str(path)
        try:
            self.last_mtime = path.stat().st_mtime
        except FileNotFoundError:
            self.last_mtime = time.time()
        try:
            self.path.parent.mkdir(parents=True, exist_ok=True)
            with self.path.open('w', encoding='utf-8') as fh:
                json.dump(
                    {
                        'last_processed': self.last_processed,
                        'last_mtime': self.last_mtime,
                    },
                    fh,
                    indent=2,
                )
        except Exception as exc:
            self.logger.warning(f'Could not write state file {self.path}: {exc}')


class NetflowIngestor:
    def __init__(self, logger: logging.Logger):
```

```
        self.logger = logger
        self.state = NetflowState(NETFLOW_STATE_FILE, logger)
        self.sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
        self.target = (NETFLOW_SEND_HOST, NETFLOW_SEND_PORT)
        self.stop_flag = False

    def run(self) -> None:
        self.logger.info(
            'Starting NetFlow ingestion from %s -> %s:%d (poll %.1fs)',
            NETFLOW_DIR, self.target[0], self.target[1],
NETFLOW_POLL_INTERVAL
        )
        signal.signal(signal.SIGINT, self._stop)
        signal.signal(signal.SIGTERM, self._stop)

        while not self.stop_flag:
            try:
                processed_files = 0
                for path in self._discover_files():
                    try:
                        flows = self._read_flows(path)
                        sent = self._emit(flows)
                        processed_files += 1
                        self.logger.info(
                            'Processed %s → sent %d flows',
                            path, sent
                        )
                        self.state.mark_processed(str(path))
```

```
                except Exception as exc:
                    self.logger.error('Failed to process %s: %s', path, exc)
            if processed_files == 0:
                time.sleep(NETFLOW_POLL_INTERVAL)
        except Exception as exc:
            self.logger.exception('Unhandled error in ingestion loop: %s', exc)
            time.sleep(NETFLOW_POLL_INTERVAL)

    self.logger.info('NetFlow ingestion stopped')


def _discover_files(self) -> Iterable[Path]:
    base = NETFLOW_DIR
    if not base.exists():
        self.logger.warning('NetFlow directory %s does not exist yet', base)
        return []

    candidates = [
        p for p in base.rglob(f'{FLOW_PATTERN}*')
        if p.is_file()
        and not p.is_symlink()
        and 'current' not in p.name
    ]
    if not candidates:
        return []

    def sort_key(path: Path) -> Tuple[float, str]:
        try:
```

```
            mtime = path.stat().st_mtime
        except FileNotFoundError:
            mtime = 0.0
        return (mtime, str(path))

    all_files = sorted(candidates, key=sort_key)

    last = self.state.last_processed
    if last is None:
        # bootstrap with the latest file only
        return all_files[-1:]

    last_mtime = self.state.last_mtime or 0.0
    newer: List[Path] = []
    for path in all_files:
        try:
            mtime = path.stat().st_mtime
        except FileNotFoundError:
            continue
        if mtime > last_mtime:
            newer.append(path)
        elif mtime == last_mtime and str(path) > last:
            newer.append(path)

    if newer:
        return newer
```

```
    # If nothing matched (e.g. state file points to deleted file), fall back
    # to returning the most recent file so ingestion can resume.
    return all_files[-1:]


def _read_flows(self, path: Path) -> Iterable[Dict[str, str]]:
    cmd = [
        NETFLOW_NFDUMP_PATH,
        '-r',
        str(path),
        '-o',
        'csv'
    ]
    try:
        proc = subprocess.run(
            cmd,
            capture_output=True,
            text=True,
            check=False
        )
    except FileNotFoundError as exc:
        raise RuntimeError(
            f'nfdump not found at "{NETFLOW_NFDUMP_PATH}". '
            'Install nfdump or set IOT_EMU_NFDUMP.'
        ) from exc

    stdout = proc.stdout or ''
    stderr = proc.stderr.strip() if proc.stderr else ''
```

```
    if proc.returncode != 0 and not stdout.strip():
        detail = stderr or f'exit code {proc.returncode}'
        raise RuntimeError(f'nfdump failed: {detail}')

    if proc.returncode != 0:
        self.logger.warning(
            'nfdump exited with code %d while reading %s: %s',
            proc.returncode,
            path,
            stderr or 'no stderr'
        )

    return self._parse_csv(stdout)

def _parse_csv(self, text: str) -> Iterable[Dict[str, str]]:
    reader = csv.reader(io.StringIO(text))
    header = None
    for row in reader:
        if not row:
            continue
        if row[0].startswith('Summary') or row[0].startswith('Time'):
            continue
        if header is None:
            header = [col.strip() for col in row]
            continue
        if len(row) != len(header):
```

```
            continue
        record = {header[i]: row[i].strip() for i in range(len(header))}
        yield record


def _emit(self, flows: Iterable[Dict[str, str]]) -> int:
    count = 0
    for flow in flows:
        msg = self._format_message(flow)
        if not msg:
            continue
        self.sock.sendto(msg.encode('utf-8'), self.target)
        count += 1
    return count


def _format_message(self, flow: Dict[str, str]) -> Optional[str]:
    sa = flow.get('sa')
    da = flow.get('da')
    if not sa or not da:
        return None


    # nfdump CSV columns
    sport = flow.get('sp', '0')
    dport = flow.get('dp', '0')
    proto = flow.get('pr', flow.get('proto', '?')).upper()
    packets = flow.get('ipkt') or flow.get('pkt') or flow.get('packets')
    bytes_ = flow.get('ibyt') or flow.get('byt') or flow.get('bytes')
    duration = flow.get('td') or flow.get('duration', '0')
```

```
        try:
            packets_val = int(float(packets)) if packets is not None else 0
        except ValueError:
            packets_val = 0
        try:
            bytes_val = int(float(bytes_)) if bytes_ is not None else 0
        except ValueError:
            bytes_val = 0
        try:
            duration_val = float(duration)
        except (TypeError, ValueError):
            duration_val = 0.0

        return (
            f"flow src={sa} dst={da} sport={sport} dport={dport} "
            f"proto={proto} packets={packets_val} bytes={bytes_val} "
            f"duration={duration_val:.3f}"
        )

    def _stop(self, *_):
        self.stop_flag = True


def main() -> int:
    logger = setup_logging()
    ingestor = NetflowIngestor(logger)
```

```
    try:
        ingestor.run()
    except KeyboardInterrupt:
        pass
    return 0


if __name__ == '__main__':
    sys.exit(main())
```