НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ ІМЕНІ ІГОРЯ СІКОРСЬКОГО»

Навчально-науковий інститут атомної та теплової енергетики

Кафедра цифрових технологій в енергетиці

«До захисту допущено»
Завідувач кафедри
_____ Наталія АУШЕВА
“___”_____2025 р.

Дипломна робота
на здобуття ступеня бакалавр

За освітньою програмою “Цифрові технології в енергетиці”

Спеціальності 122 “Комп’ютерні науки”

на тему: “Клієнт-серверний додаток моніторингу мережевого трафіку домашньої мережі”

Виконав: студент 4 курсу, групи ТР-12
Ковальов Олександр Олексійович _____
(прізвище, ім’я, по батькові) (підпис)

Керівник асистент каф. ЦТЕ Олександр КАРДАШОВ _____
(посада, науковий ступінь, вчене звання, ім’я, ПРІЗВИЩЕ) (підпис)

Рецензент доцент каф. ТАЕ, к.т.н., доц. Олександр СІРИЙ _____
(посада, науковий ступінь, вчене звання, ім’я, ПРІЗВИЩЕ) (підпис)

Н.контроль доцент каф. ЦТЕ Артем ГУРІН _____
(посада, ім’я, ПРІЗВИЩЕ) (підпис)

Засвідчую, що у цій дипломній роботі немає запозичень з праць інших авторів без відповідних посилань.
Студент _____

Київ – 2025

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
“КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО”

НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ АТОМНОЇ ТА ТЕПЛОВОЇ ЕНЕРГЕТИКИ

Кафедра ЦИФРОВИХ ТЕХНОЛОГІЙ В ЕНЕРГЕТИЦІ

Рівень вищої освіти – перший (бакалаврський)
спеціальність 122 “Комп’ютерні науки”
Освітньо-професійна програма “Цифрові технології в енергетиці”

ЗАТВЕРДЖУЮ
Завідувач кафедри ЦТЕ
_____ Наталія АУШЕВА
(підпис)
“___” _____2025 р.

ЗАВДАННЯ
на дипломну роботу студенту

Ковальову Олександру Олексійовичу
(прізвище, ім’я, по батькові)

1. Тема роботи “Клієнт-серверний додаток моніторингу мережевого трафіку домашньої мережі”
Науковий керівник Кардашов Олександр Вадимович
(прізвище, ім’я, по батькові, науковий ступінь, вчене звання)

затверджені наказом по університету від “02” червня 2025 року №1875-с

2. Термін подання студентом роботи 09.06.2025

3. Вихідні дані до роботи: мова програмування Rust, бібліотеки pcap, nom, egui, tungstenite, serde, crossbeam, csv, chrono, середовище розробки RustRover.

4. Перелік питань, які потрібно розробити:

1) провести аналіз існуючих рішень та підходів до моніторингу мережевого трафіку

2) визначити та аргументувати вимоги до функціональності та нефункціональних характеристик DPI-системи з урахуванням обмежених ресурсів роутера

3) обґрунтувати вибір інструментів і технологій для реалізації модуля розбору пакетів, сервера та клієнта.
4) розробити загальну архітектуру системи: компоненти захоплення та розбору пакетів, клієнтський та серверний застосунки
5) розробити та реалізувати модуль розбору протоколів
6) створити прикладний інтерфейс для візуалізації статистики, продемонструвати роботу системи
5. Орієнтований перелік ілюстративного матеріалу зображення інтерфейсів Wireshark, tcpdump, TShark, OPNsense, pfSense, структурні діаграми компонентів системи, приклади роботи з програмним продуктом.
6. Дата видачі завдання 14.09.2024

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів виконання дипломної роботи	Термін виконання етапів роботи	Примітка
1.	Вибір теми роботи	20.02.25	
2.	Аналіз методів та засобів розв'язання задачі	17.04.25 – 20.04.25	
3.	Розробка архітектури та загальної структури системи	21.04.25 – 25.04.25	
4.	Розробка окремих підсистем	26.04.25 – 02.05.25	
5.	Програмна реалізація системи	03.05.25 – 07.05.25	
6.	Оформлення пояснювальної записки	08.05.25 – 12.05.25	
7.	Захист програмного забезпечення	15.05.25	
8.	Передзахист	26.05.25	
9.	Захист		

Студент _____ Олександр КОВАЛЬОВ
(підпис) (прізвище та ініціали)

Керівник _____ Олександр КАРДАШОВ
(підпис) (прізвище та ініціали)

АНОТАЦІЯ

Дипломна робота виконана на 63 сторінках, містить 20 ілюстрацій, 2 додатки, 36 джерел у переліку посилань.

Мета роботи – розробка програмного комплексу для моніторингу домашньої мережі з урахуванням обмежених ресурсів домашнього маршрутизатора.

Методи та засоби: мова програмування Rust, бібліотека libpcap для захоплення Ethernet-кадрів; бібліотека nom для побудови парсерів заголовків протоколів; бібліотека для побудування графічних інтерфейсів egui.

Основний зміст роботи: аналіз існуючих рішень, формалізація вимог до модуля розбору на вбудованому пристрої, побудова компонентної архітектури (захоплювач пакетів, алгоритм ідентифікації протоколів, підсистема зберігання статистики, API комунікації), реалізація алгоритмів розпізнавання Ethernet, IPv4/IPv6, TCP, UDP, ICMP та підрахунку обсягів трафіку, впровадження інтерфейсу egui для візуалізації поточної та історичної інформації, проведення експериментального тестування на тестовій мережі з перевіркою продуктивності.

Рекомендації щодо використання: застосування в домашніх і малих офісних мережах для моніторингу та діагностики; подальше розширення бази сигнатур; інтеграція з системами сповіщення про аномалії.

Результат – програмний комплекс для моніторингу мережі.

Ключові слова: Deep Packet Inspection, Rust, libpnet, nom, egui, розбір мережевого трафіку, DPI.

ANNOTATION

The thesis is presented on 63 pages, and includes 20 illustrations, 2 appendices, and 36 bibliographic sources.

Objective: Development of a software suite for home-network monitoring, considering the limited resources of a consumer-grade router.

Methods and Tools: Rust programming language; libpcap library for Ethernet frame capture; nom library for building protocol-header parsers; egui library for constructing graphical user interfaces.

Main Content: Analysis of existing solutions; formalization of requirements for the parsing module on an embedded device; design of a component architecture (packet sniffer, protocol-identification algorithm, statistics-storage subsystem, communication API); implementation of algorithms to recognize Ethernet, IPv4/IPv6, TCP, UDP, and ICMP protocols and to calculate traffic volumes; integration of an egui-based interface for real-time and historical data visualization; experimental testing on a lab network with performance evaluation.

Recommendations for Use: Deployment in home and small-office networks for traffic monitoring and diagnostics; further expansion of the protocol-signature database; integration with anomaly-alerting systems.

Outcome: A complete software suite for network monitoring.

Keywords: Deep Packet Inspection; Rust; libpcap; nom; egui; network-traffic parsing; DPI

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ, СКОРОЧЕНЬ І ТЕРМІНІВ

DPI – Deep Packet Inspection – метод аналізу мережевого трафіку, який полягає в збиранні корисної інформації зі заголовків всіх протоколів кожного кадру, а також можливого подальшого дослідження корисного навантаження змісту.

DTO – Data Transfer Object – структурований об’єкт, призначений виключно для перенесення даних між певними різними компонентами системи без включення в нього певної логіки, що зменшує залежності та полегшує процес серіалізації.

IANA – Internet Assigned Numbers Authority – глобальна організація, яка відповідає за координацію й управління простором IP-адрес, реєстрацію доменних імен верхнього рівня, а також призначення номерів протоколів і портів у відповідності до стандартів Інтернету.

OSI – Open Systems Interconnection Basic Reference Model – базова еталонна модель для розробки комунікації між протоколами. Визначає рівні, за якими просто орієнтуватись при розробці власного протоколу, розуміти які функції вже реалізовані і покладаються на інше забезпечення, а які треба реалізувати самотужки.

OUI – Organizationally Unique Identifier – унікальний ідентифікатор виробника мережевого обладнання, вказаний на початку MAC-адреси пристрою.

RFC – Request for Comments – серія офіційних технічних документів, які містять стандарти, протоколи, рекомендації й опис інтернет-архітектури.

SoC – System-on-a-chip – дизайн електронної схеми, яка вміщує функціональні складові цілого пристрою (наприклад комп’ютера) на одній мікросхемі.

VPN – Virtual Private Network – це технологія створення захищеного та зашифрованого каналу передачі даних через публічні мережі, яка забезпечує конфіденційність, цілісність і анонімність обміну.

ВСТУП

У сучасних умовах зростаючої кількості пристроїв, підключених до домашніх мереж, та постійного зростання обсягів даних які циркулюють в мережі, актуальним постає питання ефективного моніторингу та аналізу мережевого трафіку. Через обмежені обчислювальні ресурси та малий об’єм пам’яті типових побутових маршрутизаторів доводиться балансувати між можливостями та продуктивністю рішень моніторингу мережі. Відсутність відкритих, простих у налаштуванні та супроводі інструментів, які б задовольняли специфіку домашніх та малих офісних мереж, обумовлює необхідність розробки власного програмного комплексу для моніторингу мережі на базі доступного обладнання.

Актуальність теми.

Аналіз мережевого трафіку дозволяє виявляти аномалії, оцінювати завантаженість мережі, розмежовувати трафік за типами застосунків та своєчасно реагувати на загрози безпеці. Більшість існуючих рішень орієнтовані на корпоративні середовища з потужними серверами й не можуть бути ефективно використані на побутових маршрутизаторах із 64 МБ оперативної пам’яті та 8 МБ флеш-пам’яті. Враховуючи зазначені обмеження, розробка оптимізованого комплексного рішення є своєчасною й має практичне значення.

Мета роботи.

Метою дипломної роботи є розробка програмного комплексу для моніторингу домашньої мережі з урахуванням обмежених ресурсів домашнього маршрутизатора, що забезпечує захоплення та аналіз Ethernet-кадрів, ідентифікацію мережевих протоколів, збір та зберігання статистики, а також візуалізацію результатів через GUI-клієнт на базі egui.

Завдання роботи.

Насамперед, провести огляд існуючих рішень для моніторингу домашніх мереж, знайти переваги та недоліки. Також треба визначити вимоги до функціональних і нефункціональних характеристик модуля розбору протоколів в умовах обмежених ресурсів. Є потреба в обґрунтуванні вибору інструментів і

технологій. Розробити компонентну архітектуру системи: захоплювач пакетів, модуль ідентифікації протоколів, підсистема зберігання статистики, API для клієнта. Реалізувати алгоритми розпізнавання Ethernet, IPv4/IPv6, TCP, UDP та інших протоколів з підрахунком обсягів трафіку. Створити додаток з графічним клієнтом для візуалізації поточної й історичної статистики. Випробувати інструмент на звичайному персональному комп’ютері.

Апробація результатів.

Результати роботи були представлені на студентській науковій конференції «Сучасні проблеми наукового забезпечення енергетики», секція №10 «Інформаційні технології та комп’ютерне моделювання» (доповідь «Порівняння технологій захоплення мережевого трафіку для розробки системи глибокого аналізу пакетів», квітень 2025 р.).

1 ЗАДАЧІ ПОБУДОВИ ТА АНАЛІЗ ІСНУЮЧИХ РІШЕНЬ

Розробка комплексу для моніторингу мережі полягає в проектуванні та реалізації програмного забезпечення, яке забезпечує перехват мережевих кадрів (фреймів), розбір необроблених даних та відображення на певному користувацькому інтерфейсі.

1.1 Постановка задачі

Вимоги до комплексу дещо специфічні, враховуючи деяку парадоксальність в знаннях потенціальних користувачів – вони повинні розуміти мережеву схему OSI, мати певне розуміння більшості з відомих протоколів, як мінімум основних для моделі TCP/IP, а саме Ethernet, IPv4, TCP та UDP. Також, бажано мати загальні знання з комп’ютерних мереж. Але, при цьому, можливий користувач має вдосталь інструментів для такого моніторингу – тому, бажано максимально спростити встановлення, налаштування, тощо. Всі функції комплексу мають бути максимально зрозумілими пересічному користувачу, і при цьому мати простий інтуїтивний інтерфейс.

Комплекс повинен працювати на різних девайсах, під різними операційними системами – тобто, бути кросплатформним. Орієнтуватись треба на найбільш обмежену платформу – таким видом пристроїв є домашній роутер (маршрутизатор) – мережевий пристрій, призначений для житлових приміщень, який забезпечує бездротове підключення до Інтернету для кількох пристроїв у домогосподарстві. Він служить центральним вузлом для розподілу доступу до Інтернету від широкосмугового з’єднання, що надається постачальником інтернет-послуг до різних пристроїв, таких як смартфони, планшети, ноутбуки тощо. Згідно дослідженню від «The Business Research Company», найбільш популярними маршрутизаторами для домогосподарств є пристрої від компаній TP-Link, Asus, Huawei [1]. Найбільш популярним в Україні станом на травень 2025 року є TP-Link

Archer A64. В специфікації вказано, що пристрій має 16 мегабайт флеш пам’яті, 128 МБ оперативної пам’яті та двоядерний процесор MediaTek MT7621DAT [2]. Відповідно, певну частку ресурсів займають основні функції девайсу – з цього уточнення та всього вищесказаного випливають доволі жорсткі апаратні обмеження.

Окрім цього, так як потенціальним користувачем може бути людина, яка тільки вивчає комп’ютерні мережі, або знайома з ними але без бажання розбиратись в незрозумілому перевантаженому інтерфейсі – з’являється вимога і до стабільності комплексу. Він повинен працювати стабільно, повинен бути протестованим, якнайбільше граничних випадків повинні оброблюватися.

До безпеки серйозних вимог немає – так як домашня мережа є локальною, то кількість користувачів зазвичай є невеликою й загалом обмеженою. Тому, якщо дані будуть передаватись від застосунку через мережу, вони не обов’язково повинні бути зашифрованими. Але, повинна забезпечуватись аутентифікація та авторизація користувачів – якщо взяти приклад батьківського контролю це є доволі логічним рішенням, доступ буде обмеженим, його зможуть отримати лише батьки. Так як суть застосунку полягає лише в моніторингу, такі рішення можуть здаватись суперечливими, так як навіщо потрібна аутентифікація, якщо трафік можна перехопити та прочитати. Але на це є відповідь – згідно абстрактної вимоги простоти, весь контроль над застосунком повинен проводитись в ньому самому, тому немає проблеми в перехопленні даних – але важливо, щоб не було змоги видати себе за іншого користувача.

Підсумовуючи, можна виділити функціональні та нефункціональні вимоги. Функціональними є підтримка захоплення трафіку у реальному часі, розбір даних в зрозумілий вигляд, отримання з них метаданих – тобто, «корисних даних про дані». Також, важливо підраховувати статистику, обчислювати обсяг трафіку, визначати швидкість входу та виходу пакетів з мережі, вести облік всіх подій, надати зручний інтерфейс для керування системою, можливість підписувати окремі пристрої для зручнішої ідентифікації. Дані, які можна подати в графічному вигляді, краще подавати саме так. Окрім цього, варто врахувати що користувацька

база може складатись не лише з українців – тому, повинна бути можливість змінити мову, як мінімум на англійську, і надати зручний спосіб внести зміни в систему та додати будь-яку іншу мову.

З нефункціональних вимог можна виділити продуктивність – система повинна добре працювати на малопотужних девайсах, мати систему підключень відразу декількох користувачів враховуючи мережеві обмеження, займати мало оперативної пам’яті (не більше 10 мегабайт). Затримка між отриманням та відображенням інформації повинна бути малою, визначення точного ліміту можна делегувати користувачеві, при цьому базовим значенням може бути 5 секунд, після чого повинне з’явитись певне попередження про неактуальність інформації в будь-якому вигляді. Система повинна мати змогу автоматично перезавантажуватись або бути перезавантаженою за запитом адміністратора, і бути достатньо стійкою щоб у клієнтів не було потреби розбиратись з проблемами – кажучи технічною мовою, розробник повинен достатньо протестувати комплекс, з великим покриттям всіх програмних одиниць, або, як мінімум, перекласти це на компілятор або інтерпретатор. Велика кількість отриманих даних повинна мати можливість відображатись повністю, без очистки застарілих, і при цьому не перенавантажувати систему. Конфігурація повинна бути достатньо простою для користувачів які не мають достатніх знань, або ж достатньо задокументованою (покроково). Архітектура пристроїв може бути різноманітною, тому треба обирати технічні засоби реалізації з урахуванням цього аспекту.

Якщо ж мова йде про пропускну здатність пристрою – застосунок не має права перенавантажувати, бо якщо виникнуть проблеми з мережею через нього, користувач залишиться зовсім незадоволеним і взагалі його сенс зникне. Через це бажано використовувати оптимізовані способи передачі, і загалом не навантажувати пристрій в плані процесорних тактів, якщо є спосіб розподілення задач, то краще скористуватись даною можливістю.

1.2 Аналіз існуючих рішень

Задача моніторингу мережі не є новою – скоріше навпаки, вже достатньо рішень різного спектру які так чи інакше її вирішують, з фокусом на ті чи інші аспекти. Багато з них мають відкритий код, і це відбувається вже багато років – через це треба порівняти існуючі аналоги, та знайти їхні недоліки, або ж просто слабкі місця, які не є частиною загального фокусу. Окрім цього, авжеж, треба врахувати загальну базу користувачів застосунку про який йдеться – у більшості інших рішень адміністраторами є досвідченими, перелік можливостей може бути набагато більшим ніж просто моніторинг. Через це варто провести конкурентний аналіз інших рішень, та знайти власну нішу в якій застосунок зможе продемонструвати певні переваги.

1.2.1 Аналізатори трафіку

Найбільш популярним аналізатором пакетів є Wireshark, який має графічний інтерфейс використовуючи технологію Qt і захоплює трафік за допомогою бібліотеки libpcap [3]. Цей застосунок підтримує десятки, якщо не сотні мережевих протоколів. В нього є декілька режимів, а саме захоплення та розбір даних в реальному часі та розбір вже раніше захоплених даних у форматі «.pcap». Можна застосовувати складні фільтри, використовувати кольорову підсвітку, тощо.

Режим захоплення в реальному часі починається з вибору мережевого інтерфейсу. В головному меню є вибір з усіх наявних на пристрої, також відображається міні-шкала мережевої активності. Після вибору, вмикається головне вікно, яке майже однакове в обох режимах (рисунок 1.1). Робочий простір займає перелік кадрів канального рівня, на які можна натиснути і нижче побачити повний розбір даних за полями протоколів. Збоку є можливість побачити вміст в шістнадцятковій формі. Панель управління складається з двох шарів – перший стосується глобального стану застосунку, а другий відповідає за маніпуляцію в

конкретному режимі. Як приклад, можна побачити кнопки запуску захоплення, зупинки, перезавантаження. Окрім цього, можна застосувати певні фільтри або знайти конкретний пакет.

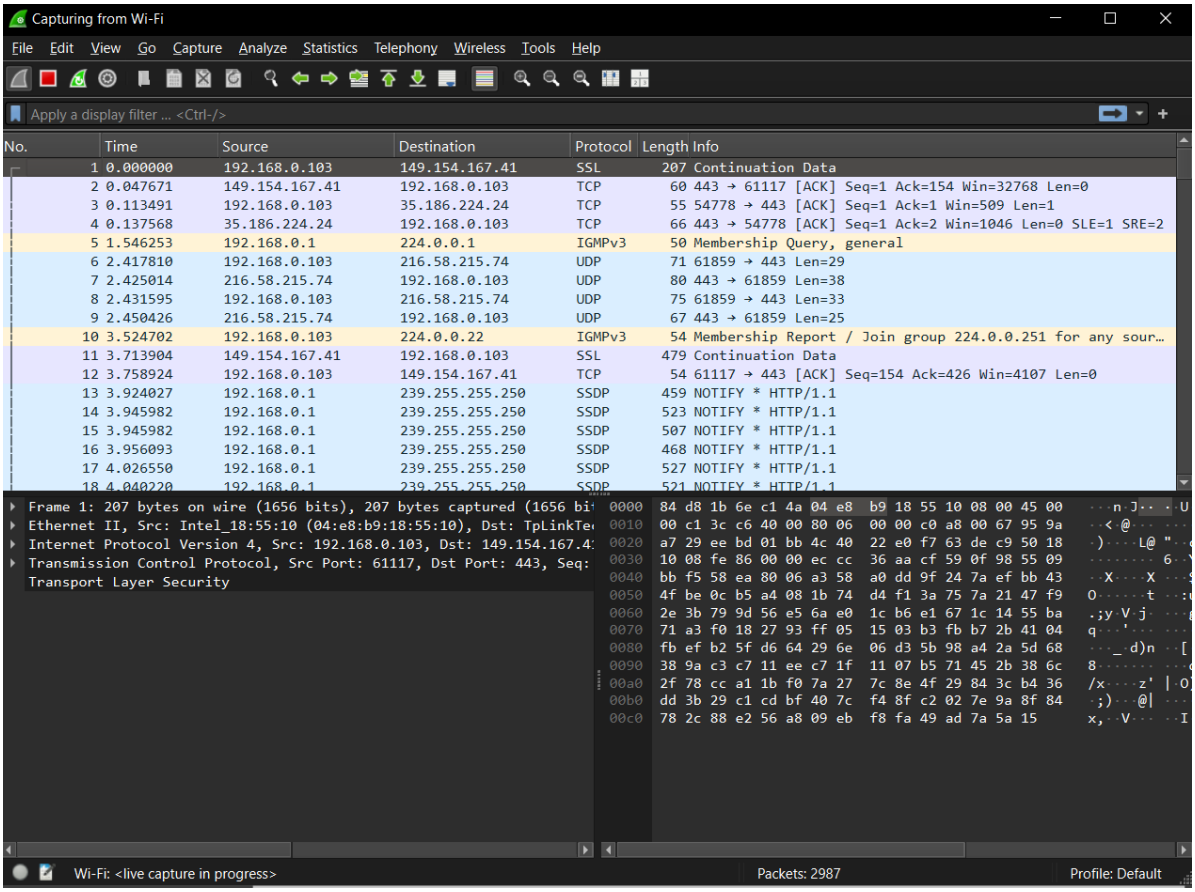

Рисунок 1.1 – Інтерфейс застосунку Wireshark

Інший режим дозволяє відкривати вже готові файли з даними. Це можна робити для доналаштування власного застосунку – як приклад, якщо будуть реалізовані певні механізми розбору протоколів, вони можуть не працювати відразу справно – невдалі спроби можуть фіксуватись і згодом зберігатись у файл відповідного формату. Після цього, можна завантажити цей файл, і переглянути, в якому саме місці алгоритм не спрацював правильно.

Існують і консольні аналоги – найбільш відомим, напевно, є утиліта під назвою «tcpdump». Вона знайома кожному адміністратору комп'ютерних мереж. Також дозволяє захоплювати трафік, переглянути заголовки пакетів, і загалом

відстежувати мережеву активність. Але інтерфейс не є інтуїтивним – загалом, програма використовується скоріше для вирішення певних проблем, які є тут і зараз (рисунок 1.2). Проте, існують користувачі які вміло уміють користуватись утилітами текстового пошуку, такими як «grep» – для них такий варіант був би зручнішим. Tcpdump не є кросплатформним, підтримується лише Linux, хоча на Windows є аналог – «pktmon».

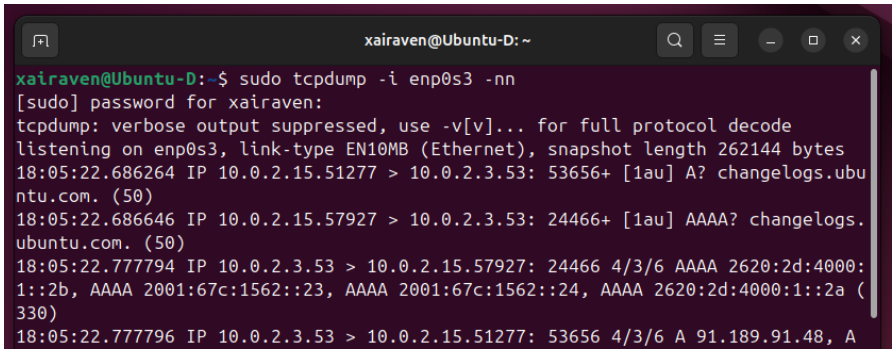

Рисунок 1.2 – Інтерфейс застосунку tcpdump

Якщо ж є бажання отримати більш детальний розбір даних, і це все в консольному вигляді – є консольна версія Wireshark [4], яка має назву «TShark». Інтерфейс, знову ж, не настільки зручний, як звичайний (рисунок 1.3).

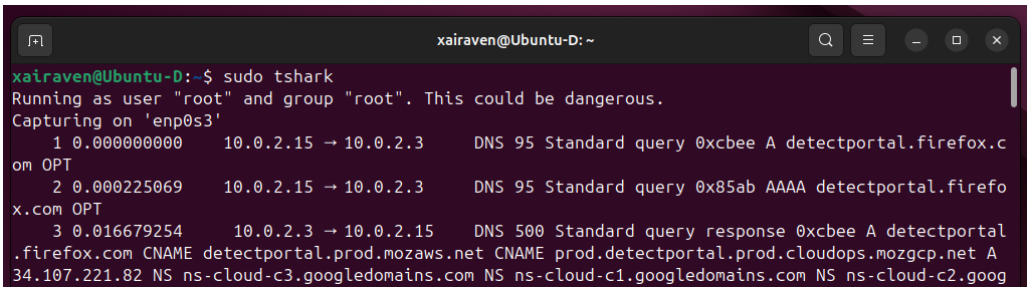

Рисунок 1.3 – Інтерфейс застосунку TShark

Але, на відміну від tcpdump, є широка підтримка протоколів. Власне, використовується та ж база розбору що й в Wireshark, що дозволяє більш глибоко інспектувати дані, ціною швидкості перехоплення.

У доповнення до класичних аналізаторів пакетів варто згадати систему під назвою «ntopng», яка поєднує функції збору інформації та веб-інтерфейсу для їх візуалізації. Основою також є бібліотека libpcap для безпосереднього перехоплення трафіку, але при цьому інтегрується з такими механізмами як PF_RING та eBPF для збільшення швидкості перехоплення пакетів, уникнення копіювання в користувацький простір. Також, використовується бібліотека nDPI для ідентифікації протоколів на рівні додатків. Зібрані дані зберігаються у базі даних Redis (іноді – у MongoDB), що дозволяє масштабувати систему та зберігати історію потоків для побудови графіків, звітів тощо. Однак така архітектура має суттєві недоліки: по-перше, високе навантаження – рішення потребує значної кількості оперативної пам’яті та процесорної потужності [5]. По-друге, залежність від зовнішньої бази даних ускладнює розгортання на вбудованих пристроях із обмеженими ресурсами та вимагає додаткової конфігурації і підтримки. Нарешті, навіть при з пристроями з високими апаратними можливостями, фільтрація в реальному часі та користувацькі звіти вимагають тонкого налаштування. Через ці причини ntopng складно використовувати як легкий компонент у домашніх маршрутизаторах без суттєвого перегляду апаратних вимог.

Підсумовуючи, можна виділити один глибокий недолік – у цих застосунків відсутня гнучкість. Їх можна використовувати лише на одному пристрої – а якщо хочеться на іншому, то треба, наприклад, налаштовувати доступ за протоколом ssh, тощо. Це все не є частиною одного спільного механізму, який би давав безпосередню можливість аналізу одразу після налаштування, і підходить вже більш досвідченим користувачам. Але все одно, якусь статистику теж довелось би робити окремо – у вищезазначених варіантах така опція взагалі відсутня. Але є й переваги – наприклад, зручний інтерфейс у Wireshark, у них є велика база протоколів разом з TShark, а tcpdump є чудовим засобом для знаходження помилок в мережі.

1.2.2 Платформи для маршрутизаторів

Для маршрутизаторів існує багато рішень у вигляді Linux-дистрибутивів і не тільки. Вони слугують не лише в цілях моніторингу, а й як брандмауер, тощо. Одним з таких рішень є операційна система під назвою «OPNsense». В її основі лежить інша UNIX-подібна операційна система – FreeBSD. І хоча й це рішення не є призначеним лише для моніторингу (насамперед, це ще й брандмауер та VPN), в ній є модуль NetFlow, розроблений Cisco [6]. Воно дозволяє відслідковувати пакети на мережевому рівні схеми OSI – тобто, захоплює та розбирає IP-пакети. Всі дані відображаються в інтуїтивному інтерфейсі та досить зручному вигляді, від текстового до діаграм (рисунок 1.4).

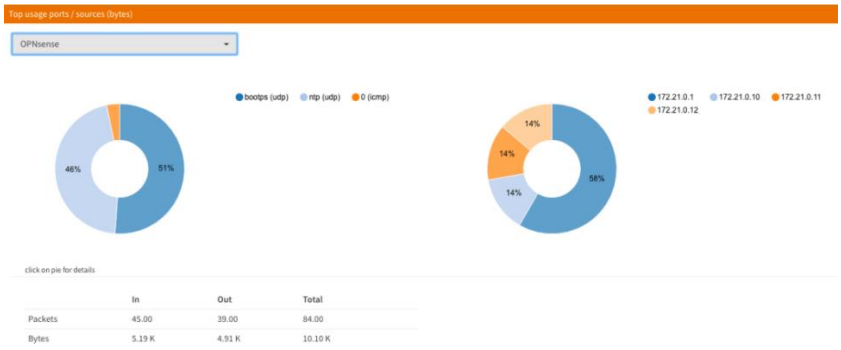

Рисунок 1.4 – Веб-інтерфейс OPNsense

Окрім цього, існує ще й доповнення до нього – NetFlow Analyzer (або Insight), яке дозволяє детально аналізувати мережеві потоки, включаючи й дані з вищих рівнів схеми OSI – наприклад, транспортний, з визначенням джерел та призначення (протоколи IP та TCP/UDP), а також конкретні сервіси, які прив’язані до певних портів. Модуль має купу різних функцій, а також можливості експорту в різні формати, такі як, наприклад, csv. Також, дані можна візуалізувати за допомогою Grafana. Наразі, це лідер на ринку вбудованих рішень, в яких все працює «з коробки». Але, є й вагомий недолік – підтримувані архітектури це i386 та x86_64, які рідко використовуються на домашніх маршрутизаторах. Зазвичай, з такою

архітектурою випускаються пристрої з більш високого цінового сегменту. Але, якщо казати про моніторинг – все ж, конкурентів немає. Хоча, що стосується розбору протоколів – він обмежений. Якщо це важливо, то варіант не підходить.

Також, варто зазначити високі апаратні вимоги, а саме як мінімум 2 гігабайти оперативної пам'яті. Тобто, система, як і написано вище, призначена для роутерів іншого цінового сегменту. Зовсім не підходить для звичайних, домашніх пристроїв.

OPNSense почали розробляти вже з готового продукту під назвою pfSense, який в свою чергу теж був створений не з нуля. В нього вимоги до апаратної частини набагато менші – потрібен процесор з тактовою частотою 100 МГц або більше, і мінімум 1 ГБ оперативної пам'яті [7]. Система із самого початку була розроблена під архітектуру x86_64, тому це також не є кандидатом на вирішення поставленої задачі. Але, треба відзначити деталізований інтерфейс (рисунок 1.5), можливість відслідковувати статус мережі, та ще багато опцій.

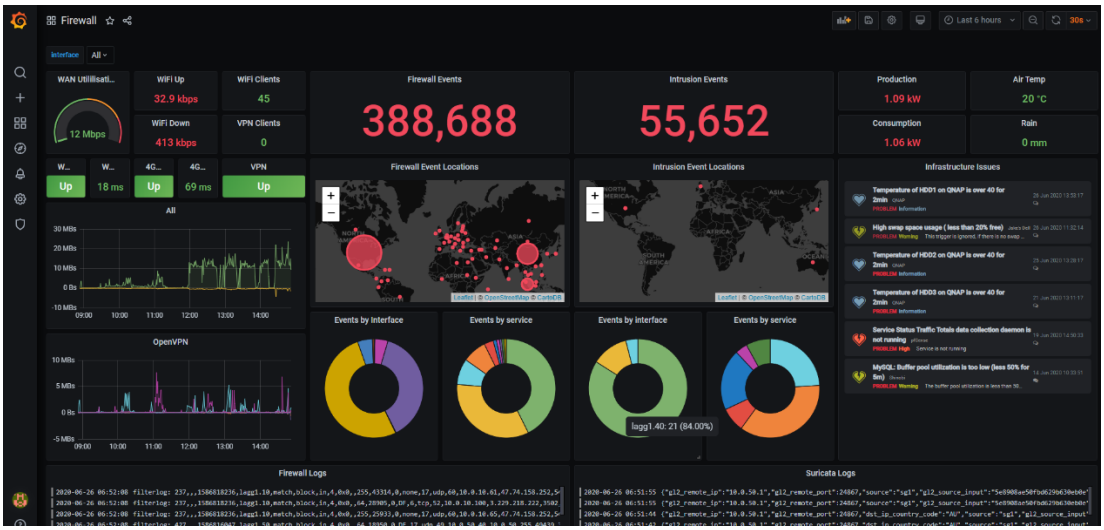

Рисунок 1.5 – Інтерфейс pfSense

Є можливість додавати користувацькі модулі, і як з OPNsense, використовувати рішення як міжмережевий екран або VPN. Підтримує інтеграцію з зовнішніми системами моніторингу, такими як Zabbix та Nagios, що дозволяє налаштовувати сповіщення, генерувати звіти та проводити моніторинг мережевої активності.

Щодо більш простих рішень з даного спектру, існує Linux дистрибутив IPFire, який зосереджений на простоті використання та базових можливостях моніторингу [8]. Він не має вбудованих інструментів для глибокого аналізу трафіку або детальної візуалізації даних, але модульний дизайн дозволяє налаштовувати систему як завгодно – в теорії, модуль обробки даних можна написати власноруч, якщо такого ще не існує. Рекомендовані апаратні вимоги це процесор з 1 ГГц тактової частоти, 1 ГБ оперативної пам'яті, 4 ГБ пам'яті та два мережевих інтерфейси. Основними функціями є DHCP-сервер, брандмауер та VPN.

Зовсім іншим за призначенням продуктом є базована на Linux операційна система OpenWRT. Вона є гнучкою платформою з можливістю встановлення різноманітних модулів, як для моніторингу (наприклад, darkstat), так і для інших потреб. Проте, налаштування та інтеграція цих інструментів вимагає додаткових зусиль та технічних знань – є збірником різних речей, які треба власноруч зібрати в цілу працюючу систему. Це дещо важче, ніж встановити на пристрій перелічені вище системи. Але, значною перевагою, яка нівелює мінуси відносно інших варіантів є апаратні вимоги. Для того щоб завантажити цю систему на роутер, треба від 8 МБ (мінімум) до 16 МБ (оптимально) флеш-пам'яті та від 64 МБ (мінімум) до 128 МБ (оптимально) оперативної пам'яті [9], що в цілому може задовольнити велику частку домогосподарств.

У висновку можна сказати, що з таких рішень як OPNsense та pfSense можна взяти лише переваги, та далі адаптувати під поставлені вимоги. Їх можливо використовувати якщо запит поставити інакше – моніторинг або аналіз на потужних серверах, в промислових системах. Так як зі зрозумілих причин не розглядається реверс-інжиніринг прошивок інших роутерів або написання власної, єдиним варіантом є використання OpenWRT з певними модифікаціями. Це дозволить виконати всі функціональні та нефункціональні вимоги.

2 МЕТОДИ ТА ЗАСОБИ РОЗРОБКИ

Враховуючи доволі нестандартну задачу, доведеться приймати нестандартні рішення в плані архітектури, підбору мови програмування, тощо. Можливо, є стандартизоване рішення для вирішення саме поставленої проблеми, але, напевно, воно не досить популярне. Також треба зазначити, що рішення не вийде ідеальним – у аналогів була присутня або підтримка спільноти протягом років, або це було (чи є) комерційним рішенням, яке цілеспрямовано тестували, бо підтвердити виконання замовлення можна лише з розробленим готовим продуктом, на відміну від відкритого коду, де його можна написати силами всієї спільноти, але все ж продукт не буде настільки відточеним. Тому, до вимог додається ще одна – технології не повинні бути нішевими, бо інакше проект приречений на забуття, і ніколи не стане в ряд з аналогами за якістю (за умови, що не стане комерційним).

2.1 Апаратна архітектура

В основі будь-якої обчислювальної платформи лежить її апаратна архітектура – набір інструкцій процесора, модель пам’яті, тощо. Найбільш відомими з архітектур загального призначення зараз є x86_64, ARM, MIPS та новітня RISC-V із відкритою специфікацією.

У світі маршрутизаторів домінують насамперед RISC-процесори через їхню енергоефективність і низьку вартість. Протягом багатьох років це були чіпи на базі MIPS (Atheros AR7xx, Qualcomm Atheros QCA95xx, MediaTek MT76xx), а останнім часом усе частіше з’являються SoC на ARM-ядрах (Broadcom із серії BCM, Marvell, Qualcomm IPQ). MIPS відіграє особливо важливу роль завдяки своїм компактним потребам у кристалі та широкій підтримці в прошивках, таких як OpenWRT.

Якщо ж вважати що еталонним для вирішення задачі є роутер TP-Link Archer A64, то він має саме MIPS-ядро [2]. Тобто, обрана мова програмування повинна мати змогу компілюватись під дану архітектуру.

2.1 Мова програмування

В бажаному програмному комплексі обов’язково має бути складова, яка відповідає за захоплення пакетів. Враховуючи апаратні вимоги та доменну область, не можна допустити затримок, які відбуваються через роботу збирача сміття (GC, Garbage Collector), тому на вибір залишаються такі мови як C, C++ та Rust. Інші мови можуть вільно використовуватись, наприклад, в частині візуалізації, тощо, а ось саме в захопленні та подальшій обробці пакетів затримки неприпустимі.

Враховуючи вимогу до безпеки та стабільності, а також рекомендації Агенства з кібербезпеки та захисту інфраструктури США, для систем з переліченими вище вимогами краще обирати мови де помилки керування пам’яттю вважаються неможливими ще на етапі проектування [10]. Мовою, де поєднується відсутність збирача сміття (висока продуктивність) та безпечність, є лише Rust.

Звичайно, можна використовувати інші мови, наприклад C/C++, але це ризиковано – насамперед, виникне потреба в досвідчених програмістах, які мають досвід достатній для уникнення різноманітних помилок, які можуть призвести до вразливостей. Розробка проекту дипломної роботи не передбачає коштів на утримання такого штату, як і наявності коштів на обширне подальше тестування для знаходження дірок в безпеці або витоків пам’яті. Так, є програмні продукти які спрямовані на нівелювання цих проблем, наприклад, санітайзери, або утиліта Valgrind, яка потрібна для відлагодження пам’яті, профілювання та знаходження витоків [11]. Але це не надійний шлях.

Rust – мова програмування, яка поєднує високу продуктивність із безпекою пам’яті за рахунок відсутності збирача сміття (що є критичним в умовах обмеженої потужності роутера) та системі володіння пам’яттю (ownership) і перевірці запозичень (borrow checker). Завдяки ним Rust дозволяє уникнути типових помилок, пов’язаних із доступом до невизначеної або звільненої пам’яті. За статистикою, можна уникнути близько 1/5 від взагалі всіх зареєстрованих вразливостей, а саме таких як buffer overflow, use-after-free та інші [12]. Крім того, Rust забезпечує безпеку в багатопотокових середовищах, використовуючи певні

обмеження та власні механізми забезпечення надійної передачі пам’яті між потоками. Це дозволяє ефективно обробляти великий обсяг мережевого трафіку без втрат у продуктивності. Ці переваги доволі значні, якщо розглядати конкурентів для системних застосунків: мови C та C++, які спроектовані таким чином, що уникати цих проблем повинен розробник.

Система перевірки запозичень Rust базується на суворих правилах володіння та життєвих циклів об’єктів (lifetimes), які на етапі компіляції виключають цілий клас помилок, властивих традиційним мовам із ручним управлінням пам’яттю [13]. По-перше, модель володіння гарантує, що кожен фрагмент даних має одного власника, а всі інші посилання на нього – це або посилання для читання (&T), або єдине запозичення з можливістю зміни об’єкту (&mut T). Це автоматично виключає сценарії використання змінних після очищення: якщо власник об’єкта виходить за межі області видимості, компілятор перевіряє, чи жодне з запозичень не виходить за цей термін життя, і не дозволяє згенерувати код із вказівником на неіснуючу або вже використану кимось іншим ділянку пам’яті.

По-друге, компілятор не дозволяє повертати з функції посилання на локальні змінні стеку: життєві цикли запозичень строго прив’язані до областей видимості, тому будь-яка спроба передати посилання назовні миттєво відкидається на етапі компіляції. Аналогічно, механізм володіння запобігає порушенню стану ітераторів: якщо контейнер або ітератор змінює свій внутрішній стан, всі запозичення його елементів стають недійсними, але компілятор виявить такі ситуації й не допустить збірки програми з невизначеною поведінкою (undefined behaviour).

Система перевірки життєвих циклів Rust масштабована на десятки й сотні тисяч рядків, модульну архітектуру і навіть на взаємодію між різними бібліотеками. Перевірка життєвих циклів працює через всю ієрархію викликів: якщо функція з одного модуля повертає запозичення з певним життєвим циклом, а інший модуль неправильно його використовує, помилка буде виявлена ще до запуску програми. Саме ця поведінка дозволяє писати великі, складні та високопродуктивні системи без класичних помилок роботи з пам’яттю. Наявність таких механізмів і є унікальною перевагою мови.

Підсумовуючи, можна сказати, що згідно вимог більш оптимальним є використання саме мови програмування Rust. Вона має більш новітній синтаксис, тому поріг входу доволі малий, і це без ризиків створити небезпечне програмне забезпечення (мова йде про помилки, які можна обробити заздалегідь за допомогою компілятора). Якщо ж в наявності було б достатньо коштів для тестування або штат досвідчених програмістів – тоді можна було б зробити вибір на користь C або C++.

2.2 Захоплення пакетів

Однією з компонент на шляху до виконання поставленої задачі є визначення методу захоплення та подальшої обробки фреймів. Так як йдеться про домашню локальну мережу, можна вважати що потік трафіку досягає максимум 100 МБ/с. Є потреба в виборі такого методу, який би захоплював кадри з мінімальною затримкою і безпечно оброблював все отримане корисне навантаження. Також, важливим аспектом є помилки управління пам’яттю, яких треба в будь-якому випадку запобігти.

Протягом останніх років сформувалося кілька основних підходів заснованих на використанні певних технологій захоплення пакетів. Технологія pf_ring, реалізована на рівні ядра та написана на мові програмування C, використовує спеціалізований драйвер для прискореного доступу до мережевого адаптера, що дозволяє обробляти великі обсяги трафіку з мінімальною затримкою [14]. Проте через відсутність гарантій безпечного доступу до пам’яті її застосування може бути ризикованим у системах із підвищеними вимогами до безпеки. Інший підхід – використання eBPF (Extended Berkeley Packet Filter) – ця технологія вбудовує байткод прямо в ядро Linux, перевіряючи його коректність перед виконанням і таким чином знижуючи ризик експлуатації вразливостей. Хоча eBPF забезпечує компроміс між швидкістю й безпекою, розробникам необхідно писати код мовою програмування C і вивчити нову модель виконання, а значна частина коду на мові програмування Rust (бібліотека aya-rs) залишається небезпечною (блок unsafe) [15].

Окрім цих підходів, існує й інший – більш безпечний, але повільний. Він полягає у використанні бібліотеки pcap, написаної мовою C. Вона використовується в програмному забезпеченні Wireshark, яке було описане вище. Безпечність тут забезпечується використанням мови програмування Rust. В її екосистемі існують бібліотеки обгортки під назвою libpnet та libpcap. Перша додає низку власного функціоналу, а друга є реалізацією підмножини функцій оригінальної бібліотеки. Є й мінуси – вони працюють в просторі користувача, копіюючи пакети з ядра до програми [16]. Така модель додає затримки через I/O, але повністю гарантує безпечний доступ до пам’яті завдяки властивостям Rust: відсутності збирача сміття, статичній перевірці типів і володінню ресурсами. Це робить дані варіанти привабливими для побудови модульних та масштабованих рішень аналізування заголовків трафіку.

Жодне рішення не є універсальним: pf_ring забезпечує максимальну пропускну спроможність, eBPF – баланс швидкості й безпеки в Linux-середовищі, а libpnet або libpcap враховуючи Rust-реалізацію гарантує відсутність помилок пов’язаних з помилок управління пам’яті ціною деякої затримки. Вибір оптимального підходу для системи аналізу заголовків залежить від пріоритетів: якщо ключовим є захист від помилок – перевага за libpnet або libpcap. Якщо ж вирішальним фактором є пікова продуктивність – варто обирати pf_ring чи eBPF. Як було зазначено на початку, типові домашні роутери не потребують обробки з великим потоком даних – тому, згідно вимоги до стабільності, більш оптимальним (бо надійним) рішенням є використання libpcap.

2.3 Програмна архітектура

Існує досить багато варіантів, яким чином організувати програмну архітектуру. Найбільш популярними є клієнт-серверна та шарова архітектура (фронтенд, бекенд і аналогічна взаємодія між браузером та сервером) [17]. Враховуючи обмеження, є декілька шляхів, як організувати проект.

Першим підходом є найбільш класичний – веб-сайт написаний з використанням мови програмування JavaScript, мови розмітки HTML та оформлення за допомогою каскадних таблиць стилів CSS. Ці сторінки може обслуговувати HTTP-сервер, який би віддавав зміст згідно запиту. Сервер може бути як і вбудований, якщо системою є OpenWRT (uHTTPd) так і будь-який інший, наприклад Apache або nginx. Така архітектура має перевагу в тому, що складність мінімальна – не потрібні ніякі додаткові інструменти збірки, а веб-сервер і сервер обробки пакетів можуть працювати як окремі сервіси, легко перезапускаючись без взаємозалежності. Однак через те, що весь інтерфейс завантажується заново при кожному переході, користувацький досвід може здаватися трохи застарілим, а обсяг статичних файлів обмежується флеш-пам’яттю пристрою. Якщо ж ця частина знаходиться на маршрутизаторі, це може викликати навантаження процесору, так як потрібно і передавати дані на користувацький комп’ютер, так і в реальному часі візуалізувати зміни. Якщо цю частину перенести на персональний комп’ютер, виходить клієнт-серверна архітектура, але все ж, кодова база не буде консистентною. Бажано використовувати одну мову програмування для всіх компонентів – але при цьому дотримуватись саме клієнт-серверної, а не шарової архітектури.

Якщо покращити минулий варіант, отримуємо SPA (Single-Page Application) використовуючи бібліотеку React (JavaScript) з тією ж клієнт-серверною архітектурою. React-додаток забезпечує плавну взаємодію без перезавантажень, відкладене завантаження компонентів та вбудовані інструменти для керування станом, що робить UI більш сучасним і зручним. Однак збірка React-пакету (webpack, babel) генерує об’ємніші артефакти, які довелось би зберігати в зберігати у 8-16 МБ флеш-пам’яті роутера. Відтак довелось би ретельно оптимізувати збірки, мінімізовувати залежності й, можливо, задатися питанням, чи вистачить місця для всіх компонентів. Утім, якщо флеш-пам’ять дозволяє, React забезпечив би гарантовано кращий користувацький досвід на стороні клієнта.

Для розміщення перехоплювача пакетів на маршрутизаторі, треба в будь-якому випадку змінювати стандартну прошивку, тому що базова в багатьох

випадках дає чіткий та обмежений перелік можливостей, а програмний комплекс повинен бути стандартизованим. Тому, оптимальним є використання Linux-дистрибутиву OpenWRT. А відтак, можна скористатись модульним веб-інтерфейсом системи під назвою «LuCI». У такому випадку виключається клієнт-серверна архітектура – виходить лише шарова, що має зазначені вище недоліки. Все одно код зберігається на маршрутизаторі, і незважаючи на невеликий розмір файлів, бажано все навантаження і файли перенести на користувацьке обладнання. Окрім цього, незважаючи на зручність та можливу консистентність в плані інтерфейсу – вона зникла б з кодової бази, так як система LuCI вимагає використовувати не лише JavaScript, а ще й мову програмування Lua. Від такого підходу вже відмовляються, але на даний момент довелось би використовувати мінімум три мови програмування без значної на те потреби, ще й з додатковим навантаженням на процесор через процеси візуалізації повністю на маршрутизаторі, через що даний підхід не є оптимальним.

Є можливість інтегрувати сучасні технології, тобто використати WebAssembly (WASM) за допомогою бібліотеки для побудови графічних інтерфейсів egui та системи збірки Trunk, і таким чином поєднати найсильніші сторони Rust: і візуальна частина, і логічна пишуться однією мовою, а графічна бібліотека допомагає транкомпілювати код в WebAssembly, у результаті візуалізація працює прямо в браузері [18]. Результатом є статично зібраний wasm-модуль, який за обсягом може бути навіть меншим за великі JavaScript-збірки, а також отримані гарантії безпеки Rust відносно пам’яті. Мінус цього підходу вже був неодноразово озвучений – без клієнт-серверної архітектури буде зайве навантаження через візуалізацію даних, й загалом непотрібні файли будуть займати вкрай обмежене місце.

Нарешті, рознесена архітектура «клієнт на пристрої, легкий сервер написаний на Rust на маршрутизаторі» дає оптимальний баланс: обчислювальне навантаження інтерфейсу переноситься на потужніший пристрій, а на самому маршрутизаторі залишається тільки сервер з модулем обробки пакетів і мінімальними залежностями та статичним бінарним файлом. Таким чином, будуть

спрощені вимоги до пам’яті та процесору роутера, усунуті будь-які обмеження флеш-пам’яті для статичних файлів (окрім, звичайно, обмеження для самого сервера), а клієнтський застосунок може періодично отримувати дані через певний вид комунікації із власного зручного інтерфейсу. Єдиний мінус – необхідність керувати двома середовищами розгортання (сервер і клієнт), але це компенсується використанням додаткових скриптів, і системою збірки від GitHub під назвою GitHub Actions.

2.4 Комунікація між компонентами

При побудові клієнт-серверної взаємодії для бажаного комплексу одним із ключових питань стає вибір протоколу зв’язку між клієнтським застосунком та серверною логікою. Традиційний підхід базується на HTTP-запитах: наприклад, фреймворк Axum у Rust простий у налаштуванні й дозволяє використовувати знайомі REST-контракти [19], однак пов’язаний із суттєвим надлишком функцій, і таким чином бінарний файл стає більшим у розмірі. До того ж, кожен HTTP-запит передбачає встановлення й закриття TCP-з’єднання (якщо не використовується keep-alive), формування та передачу громіздких заголовків, обробку статусних кодів і метаданих, що спричиняє додаткове навантаження на процесор та мережеву підсистему при частих викликах. Крім того, у випадку моніторингу в реальному часі, коли споживачу необхідно отримувати нові дані про пакети майже терміново, REST-модель з концепції циклічності «запит-відповідь» виявляється малоефективною, не уникнути так званого «шаблону опитування». Клієнту доводиться або опитувати сервер на наявність оновлень із заданим інтервалом, або ризикувати отримати застарілі дані.

На противагу цьому WebSocket-з’єднання, реалізоване за допомогою бібліотеки tungstenite (якщо використовувати мову програмування Rust) або схожих, встановлює єдину довготривалу сесію поверх протоколу HTTP/1.1 із встановленням з’єднання лише один раз – на початку [20]. Після відкриття каналу

обидві сторони мають можливість миттєво обмінюватися повідомленнями без повторної генерації HTTP-заголовків, що значно знижує об’єм трафіку та затримки. Сервер може негайно передавати оновлення клієнту, як тільки з’являються нові дані для відправки, а клієнт у будь-який момент ініціювати команди або запити. Такий двосторонній, подійно-орієнтований зв’язок ідеально підходить для додатків які працюють у реальному часі, оскільки виключає необхідність періодичного опитування і дозволяє більш ефективно використовувати ресурси маршрутизатора.

Інші методи, зокрема gRPC над HTTP/2 або навіть чисті TCP-сокети, теж мають свої переваги й недоліки. gRPC забезпечує контрактну взаємодію з автоматично згенерованими клієнтськими обгортками та підтримкою потокової передачі інформації, але вимагає використання додаткових бібліотек і протоколів, подекуди не настільки легких щоб використовувати їх на обмежених ресурсах домашнього маршрутизатора. Звичайні ж TCP-з’єднання мінімізують накладні витрати на заголовки, проте натомість перекладають на розробника необхідність реалізації власного протоколу обміну, контрольних повідомлень та обробки відновлення з’єднання, що суттєво ускладнює розробку й тестування.

Підсумовуючи, можна сказати що проста інтеграція та висока продуктивність в контексті системи моніторингу є саме WebSocket-канали. Вони дозволяють серверу передавати інформацію про події негайно, без зайвого коду для рукостискання і безперервного опитування, а клієнтському інтерфейсу не треба підтримувати актуальний стан мережі (хоча, для візуальної складової це можна реалізувати, і продемонструвати, чи наявний зв’язок). Сам протокол працює на стандартному HTTP-порті – 80, тому проблем із міжмережевими екранами чи проксі не виникає. Завдяки цьому WebSocket-зв’язок забезпечує мінімальні затримки та оптимальне використання обмежених ресурсів пристрою, тому цей варіант є оптимальним.

2.5 Вибір моделі багатозадачності для серверу

У Rust існують дві основні концепції паралелізму: асинхронна модель (async/await із використанням додаткових програм, бібліотек або процесів, які контролюють виконання – так званих рантаймів, наприклад Tokio) та класичні потоки операційної системи (модуль thread зі стандартної бібліотеки Rust під назвою std). В асинхронній моделі один потік (хоча, кількість можна регулювати) обробляє велику кількість задач, які по черзі виконуються у неблокуючому режимі. Такий підхід дає змогу ефективно масштабуватися на тисячі одночасних підключень, а перед розробником стоїть задача – як реалізувати підключення відразу декількох адміністраторів до одного серверу, та отримання ними однакової інформації? Також цей підхід дозволяє уникати великого числа блокуючих потоків і зменшувати витрати пам’яті на стеки потоків. Але, кількість підключень в бажаному програмному комплексі не є достатньо великою – тому, це може бути лише зайвим навантаженням, оскільки асинхронне програмування в Rust вимагає використання вищевказаних процесів, які б регулювали цей механізм (Tokio або іншого), явного очікування виконаних задач (.await) і складнішої обробки помилок (через механізм під назвою futures), що може ускладнювати розробку [21].

Натомість, модель з класичними потоками припускає виділення окремого потоку операційної системи під кожне завдання (наприклад, під кожне підключення). Такий підхід є інтуїтивно зрозумілим і простим: функція може виконувати блокуючі операції при роботі з файловою системою, а операційна система сама керує плануванням потоків на ядрах процесора, без використання зайвих проміжних посередників. Взагалі, така модель з використанням «рантайму» саме на рівні програми а не операційної системи називається «зелені потоки», або з англійської «green threads», і це не завжди є перевагою – для малих за потужністю систем це є, скоріше, значним мінусом, через зайве перевантаження. Класична модель набагато більше підходить у даному випадку, через конкретно окреслені для них задачі та невелику кількість паралельних з’єднань.

2.6 Концепції багатопотоковості в Rust

Мова програмування Rust гарантує безпеку багатопотоковості на рівні типів, використовуючи систему володіння та спеціальні інтерфейси, які тут називаються «трейти».

Одною з основних конструкцій є володіння та запозичення. Якщо раніше в роботі це розглядалось в контексті витоку пам’яті, то в контексті багатозадачності це гарантує відсутність одночасних конфліктних звернень до однієї й тієї ж ділянки пам’яті. Наприклад, спроба взяти два посилання з можливістю змінювати данні призведе до помилки. Така перевірка не дозволяє згенерувати код, де два потоки одночасно записували б в один і той самий об’єкт без синхронізації.

Також, існують інтерфейси «Send» і «Sync», які повідомляють компілятору про безпечність передачі типів між потоками, тобто, це так звані «трейти-маркери» [22]. Send означає, що володіння значенням T можна перемістити (концепція move) в інший потік без можливості отримати невизначену поведінку. Майже всі базові типи і власні структури є Send, окрім деяких (наприклад, лічильник посилань не є Send, оскільки його реалізація не є безпечною для передачі в різних потоках).

Якщо тип містить тільки Send-поля, він автоматично теж стає Send. Трейт Sync означає, що доступ за незмінним посиланням до типу T є безпечним з кількох потоків одночасно; формально, тип T є Sync тоді й тільки тоді, коли посилання на нього T імплементує Send.

Наприклад, тип Arc (атомарний лічильник) виконує і Send і Sync, а тип Mutex реалізує Sync (тобто посилання на м’ютекс можна ділити між потоками). Таким чином, трейти Send/Sync разом із моделлю володіння на етапі компіляції гарантують, що будуть виявлені порушення цих правил, запобігаючи «гонкам даних» – стану, коли один потік змінює пам’ять, а інший намагається її отримати, і згодом отримує не до кінця змінену інформацію, яку можна назвати пошкодженою, і це є невизначеною поведінкою (undefined behaviour).

Але ж не всі дані можна просто так безпечно передати між потоками, через що існують вищевказані типи Arc (атомарний лічильник посилань) та м’ютекс.

Вони використовуються для безпечного спільного доступу у кількох потоках. Атомарний лічильник посилань дозволяє кільком потокам мати власні «копії» одного вказівника на дані. Його можна безпечно клонувати в кожний потік з повними гарантіями безпеки. Якщо ж дані треба не лише читати, а й змінювати, ці дані окрім Arc обгортають ще й в тип Mutex. М'ютекс гарантує, що одночасний доступ відбувається лише одним потоком. При цьому Rust гарантує, що сам процес блокування/розблокування м'ютексу не призведе до помилок (забув розблокувати, тощо), адже він не просто блокує, як в інших мовах програмування – а володіє змінною, і при вдалій спробі блокування повертає тип MutexGuard, за яким можна отримати доступ до змінної через змінюване посилання. І згодом, через так званий шаблон «RAII», змінна звільнить м'ютекс при виході з області видимості [23]. З приводу багатопотоковості з м'ютексами варто зазначити: механізм м'ютексів досить простий, але вимагає обережності (можливі взаємні блокування), проте компілятор усуває низку помилок за рахунок типобезпечності.

До того ж, для простих атомарних операцій (наприклад, інкремента лічильника) можна використати атомарні типи, які дозволяють оновлювати значення без блокувань. Усі вони надають операції load, store, fetch_add/fetch_sub, compare_exchange тощо. Кожна така операція вимагає вказати порядок пам'яті (тип Ordering). Наприклад, найсуворіший порядок SeqCst гарантує послідовну узгодженість всіх атомарних операцій у програмі, тоді як Acquire/Release встановлюють більш слабкі локальні бар'єри, а Relaxed ігнорує порядок між потоками [23]. У більшості випадків для простоти безпеки рекомендують використовувати SeqCst, якщо немає жорстких вимог до продуктивності. Якщо ж вони є, оптимальним вважається використання Acquire/Release.

Таким чином, Rust забезпечує безпеку роботи з потоками на рівні системи типів. Поєднання обмежень володіння і інтерфейсів Send/Sync гарантує, що будь-яка спільна змінна або передається між потоками тільки через безпечні примітиви (Arc, Mutex, атоміки), або взагалі не доступна двом потокам одночасно. Якщо код порушує ці правила, компілятор видає помилку на етапі збірки. При виконанні звичайних задач, отримати гонку даних доволі важко.

2.7 Спосіб розбору даних

Бібліотека pcap повертає тип «Packet», якщо їй вдалося отримати мережевий кадр. Структура складається з заголовку, в якому вказані дані про довжину фрейму та час отримання, і даних – байтового зрізу.

2.7.1 Зрізи

Зріз (або слайс) у Rust – це тип, який представляє послідовність елементів у пам’яті, що вже десь існують. Він не зберігає самі дані, а лише посилання на частину масиву, вектора або іншої колекції разом із її довжиною.

У контексті розбору даних зрізи дозволяють ефективно оперувати частинами байтових буферів – наприклад, розділяти пакет на заголовок і тіло, просто передаючи зрізи з різними діапазонами індексації. Це дуже ефективно і безпечно, оскільки не потребує копіювання пам’яті й не допускає некоректного доступу.

У Rust пряме індексування зрізів (&[T]) захищене від некоректного доступу – під час виконання компілятор додатково перевіряє, що індекс лежить в межах його довжини. Якщо це не так, програма видає необроблювану помилку, яка називається панікою, замість доступу до неї. В мові програмування C або C++, наприклад, інколи це викликає помилку без зайвих деталей, яка називається segmentation fault [24], а інколи дає без перешкод змінити або прочитати цю пам’ять, що знову ж є невизначеною поведінкою. Ці перевірки автоматично додаються компілятором, і виконуються під час роботи програми, коли її виконання доходить до моменту отримання даних за індексом. Якщо ж компілятор на етапі компіляції може довести, що вираз точно виходить за межі, він просто не скомпілює такий код – розробник побачить деталізовану помилку. І навпаки, якщо точно відомо, що індекс не перевищить довжину масиву (таке може бути якщо наприклад йде доступ за циклом з визначеним діапазоном, і при цьому точно відома довжина масиву – компілятор може оптимізувати такий код [25], і не додавати

зайвих перевірок. Таким чином у безпечному коді Rust неможливо зчитати дані за межами буфера і отримати невідомі дані – замість цього програма припиняє виконання з панікою. Якщо ж користуватися методом отримання даних, то можна отримати тип Option, який є замінником концепції null в Rust – шаблон «Result». Це усуває непередбачуваний кінець роботи програми [26], але змушує додати зайву обробку випадку, коли дані не вийшло отримати. Звичайно ж, це ефективно і безпечно – але візуально ускладнює код, бо у випадку з перебором даних на протоколи доводиться таку перевірку чимало разів, і це в одному модулі, якщо ж не функції.

2.7.2 Бібліотека для створення «парсерів-комбінаторів»

Бібліотека під назвою «nom» пропонує декларативний підхід до парсингу (процес аналізу вхідних даних з метою виділення смислової, заздалегідь встановленої структури згідно з певним набором правил), базуючись на функціональному шаблоні «комбінатор». Мета бібліотеки – забезпечити інструменти для побудови безпечних парсерів, без зменшення швидкості та збільшення споживання пам’яті. Для цього використовуються строгі типи Rust та гарантії безпеки пам’яті. Іншими словами, вирішується проблема, що треба вказувати певні індекси вручну, і потім обробляти можливі помилки – за допомогою цієї бібліотеки можна вказати бажану кількість байтів (наприклад, 2), і отримати кортеж зі залишком (теж слайс) і отриманим новим значенням, якщо ж його вийшло отримати. Якщо ні, то можна повернути результат функції (помилку) за допомогою оператора «знак питання», і тим самим винести перевірку назовні. Тобто, вона б відбувалась лише один раз, і в самому механізмі перебору цих перевірок не було б. А ззовні є можливість налаштувати перебір так, щоб при отриманні помилки можна було припинити розбір, або ж скористатись «комбінаторною» складовою бібліотеки – спробувати інший алгоритм розбору, що є дуже корисним при розборі кадрів на протокольні дані.

Схема повернення залишку має назву «zero-copy», бо парсер не копіює байти, а повертає посилання на початковий зріз (залишок).

Як приклад роботи цього механізму, можна привести алгоритм парсингу кадру з самого початку, тобто канального рівня. Парсер Ethernet-кадра може спочатку викликати парсер MAC-адреси на перші 6 байтів, потім ще одного парсеру MAC-адреси на наступні 6, а потім запитати 2 байти для поля під назвою EtherType, з'єднуючи їх у єдину структуру [27]. Бібліотека дозволяє виразити це компактно без зайвих перевірок, і розробник взагалі не має турбуватись щодо помилок індексування. А залишок після Ethernet-парсера потім передається далі до IP-парсера, а далі – до TCP тощо. Такий ланцюжок обробляється крок за кроком, і при помилці на будь-якому рівні nom поверне опис помилки, і залежно від нього можна або обрати інший варіант, або припинити виконання.

Завдяки декларативному стилю (комбінатори, замикання, кортежі тощо) код залишається модульним і масштабованим: кожен рівень протоколу описується своїм функціональним парсером, який потім об'єднується з іншими. Примітиви на зразок методі «be_u8», «be_u16» спрощують витягування полів з двійкових структур, а різноманітні контейнери дозволяють скласти їх у єдину воронку, з якої потім можна отримати результат. Таким чином дуже просто написати так звані «диссектори» – програмні компоненти, призначені для аналізу, інтерпретації та розбору мережевого пакету на складові частини відповідно до специфікації певного протоколу [28], безпечно і без використання індексації, при цьому зберігаючи високу продуктивність.

2.8 Структури даних для зберігання префіксних сутностей

У мережевих систем схожих на ту що є ціллю роботи, часто виникає потреба організувати ефективне зберігання та швидкий пошук сутностей, представлених префіксами бітових або байтових послідовностей. Прикладом є IP-адреси, що мають довжину 32 або 128 бітів у випадку IPv4 та IPv6, або MAC-адреси з

фіксованою довжиною 48 бітів. MAC-адреса зазвичай записується шістнадцятковим рядком з шести байтів, три з яких (не завжди) позначають організацію-виробника обладнання (OUI, Organizationally Unique Identifier), і ще три є індивідуальним ідентифікатором пристрою. Відповідність між MAC та OUI є прикладом задачі знаходження відповідностей між послідовностями байтів для подальшого отримання інформації, у даному випадку – отримання назви фірми яка є виробником обладнання.

Для вирішення цієї задачі використовують різні структури даних, а саме бітові дерева, хеш-таблиці з розбиттям по діапазонах префіксів, інтервальні дерева та стиснені префіксні структури, такі як Patricia-trie або Radix-tree. Бітове дерево забезпечує очевидну відповідність кожного біта ключа вкладенню вузлів, але витрачає багато пам’яті на вузли з двома вказівниками та неефективно працює при великій кількості порожніх гілок. Інтервальні дерева дозволяють оперувати діапазонами префіксів і знаходити найбільш пригожий діапазон для заданого ключа, але складність вставки та видалення може бути вищою, а логіка пошуку вимагає додаткових перевірок на перетин інтервалів. Хеш-таблиці, попри швидкий доступ за фіксованим індексом, погано підтримують пошук за префіксом, оскільки для кожного запиту доводиться виконувати послідовність хешувань або обходів кількох можливих контейнерів.

Patricia-trie оптимізує бітове дерево за рахунок злиття вузлів, що мають лише одного нащадка, зберігаючи при цьому властивість швидкого префіксного пошуку. Однак, ця реалізація потребує підтримки складної логіки злиття та розбиття вузлів, а також механізмів обробки випадків, коли ключі мають довгі спільні префікси. Radix-tree, у свою чергу, розв’язує ці проблеми за допомогою збереження в кожному вузлі не по одному біту, а одразу групи бітів або байтів, відповідних певному префіксу. Це дозволяє істотно зменшити глибину дерева, знизити кількість проміжних вузлів та оптимізувати збереження кешу при обході. Для кожного вузла зберігаються лише ті гілки, які справді існують у наборі ключів, що ще більше зменшує витрати пам’яті [29].

Саме радиксне дерево є найбільш вдалим вибором для вирішення даної задачі. Типова довжина префікса ідентифікатора організації – 24 біти, але є й ідентифікатори по 28 та 36 бітів в одній послідовності, що можна розглядати як один або кілька кроків у Radix-дереві з радіусом розбиття на байти. Замість того щоб крок за кроком аналізувати кожен біт, Radix-tree дозволяє переходити відразу за трьома байтами, після чого зберігати й шукати відповідні значення OUI. Такий підхід мінімізує глибину структури і кількість порівнянь, а також спрощує підтримку алгоритмів вставки і видалення. Саме тому для реалізації співставлення MAC-адрес до їх організаційного ідентифікатору у системах, де існує потреба шукати виробника пристроїв найбільш доцільно застосовувати структуру Radix-tree – вона гарантує високу швидкодію пошуку за префіксом, ефективне використання пам’яті та простоту розширення структури під майбутні потреби.

2.9 Зберігання коду, збірка

Усі етапи розробки сучасного програмного забезпечення неможливо уявити без системи контролю версій, і в більшості випадків розробники обирають систему під назвою «Git» як найбільш популярний інструмент з цього спектру, який має розподілену модель зберігання історії змін, підтримку гілок і злиттів, гарантовану цілісність даних та можливість роботи без з’єднання з інтернетом.

Серед сервісів для розміщення Git-сховищ найпоширенішими є GitHub та GitLab. GitLab пропонує вбудовані механізми інтеграції та доставки програмного забезпечення, тощо. В свою чергу, GitHub характеризується масовою залученістю розробників, великою кількістю відкритих проєктів із прикладами інтеграцій та високою стабільністю платформи. Це теж вирішує проблему, сформулювану раніше – треба якомога спростити можливість зробити зміни до системного коду, бо інакше проект приречений на велику кількість проблем. Особливою перевагою GitHub є система Actions, яка дозволяє описати в автоматизовані робочі процеси збірки комплексу під різні платформи та створення вихідних бінарних файлів.

3 ОПИС ПРОГРАМНОЇ РЕАЛІЗАЦІЇ

Програмна реалізація комплексу для моніторингу домашньої мережі включає в себе використання трьох компонентів – серверу, модулю парсингу протоколів та клієнтський застосунок. Загалом складається з чотирьох проектів, що зроблено для розділення програмної логіки та повторного використання певних функцій.

3.1 Спільний функціонал

Багато логіки є спільною між проектами серверу та клієнтського застосунку, а саме аутентифікаційні заголовки, логіка стискання, криптографічні функції, налаштування конфігураційних файлів, журналювання і насамкінець спільне API для комунікації між ними (рисунок 3.1).

Проект під назвою «common» є бібліотечною одиницею, які в мові Rust називаються «крейт». Він має лише необхідні залежності, щоб зменшити розмір вихідних файлів.

Модуль «auth» визначає заголовки, які присутні у першому повідомленні WebSocket – потрібні для аутентифікації та подальшої авторизації. Саме їх присутність перевіряють відповідні алгоритми в проектах серверу та клієнту. Також, присутній заголовок наявності стиснення – якщо в застосунках різні налаштування компресії, то підключення повинно вважатись неуспішним. Окрім цього, присутні описи помилок, які передаються, якщо не вдалося підключитись.

Інший модуль під назвою «compression» містить функції, які потрібні для стискання та розтискання повідомлень за допомогою компресору «ZLIB». Це може бути потрібно, якщо пакети проходять скрізь сервер доволі рідко, і при цьому немає змоги сильно завантажувати мережу передачею повідомлень. Також, якщо на пристрої з сервером є потужний процесор, використання цієї опції є рекомендованим. Відповідні опції є як на клієнті, так і на сервері. Для правильного функціонування треба злагодженість на обох компонентах.

Криптографічний модуль містить лише одну функцію, яка потрібна для того щоб зашифрувати пароль. Якщо його потрібно передати по мережі, а це відбувається під час спроби під'єднання, то обов'язково використовується хешування за допомогою алгоритму SHA256.

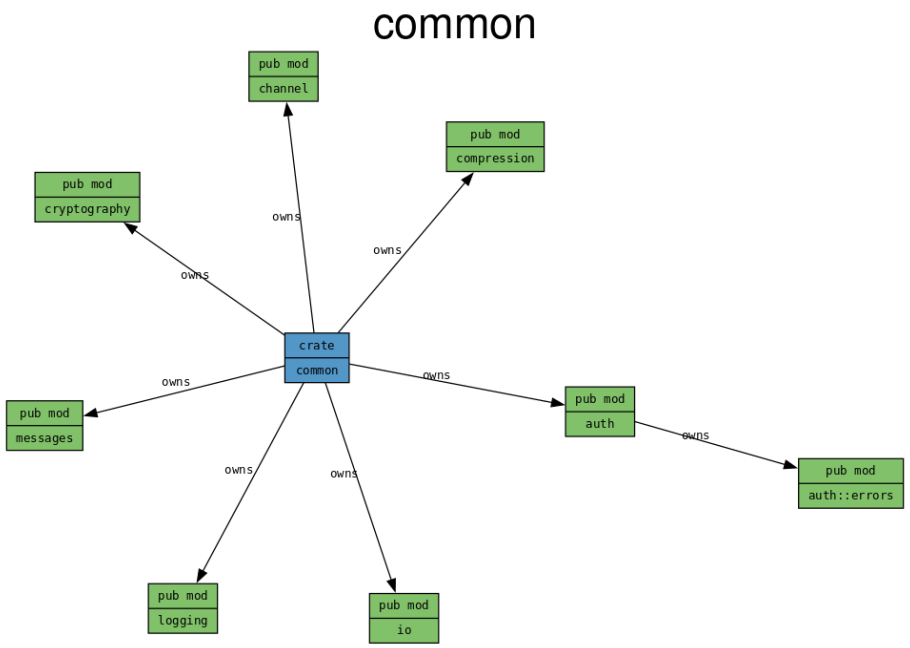

Рисунок 3.1 – Структура бібліотеки «common»

Модуль, який називається «io» – з англійської input/output, тобто операції з файловою системою – використовується для визначення шляху конфігурації клієнтського застосунку, тому що він знаходиться за замовчуванням в директорії %APPDATA% на Windows та /home/user/.config на Linux. Окремо визначені опції для файлових даних – наприклад, профілі підключення, список псевдонімів пристроїв, тощо.

Журналювання відбувається за допомогою функції, яка заміняє відповідні ідентифікатори з рядку формату на відповідні дані, які отримуються за допомогою шаблону «фасад» – бібліотеки під назвою «log» [30]. Цей підхід дозволяє уніфікувати запис всіх подій, і в подальшому дати користувачу можливість змінювати налаштування на свій смак.

Насамкінець, модуль «messages» описує API, яке використовується при комунікації застосунків. За допомогою перелічень мови програмування Rust, які є «вичерпними», можна визначити точний перелік команд, які можна приймати і передавати. Якщо ж в цей список додається новий варіант, обидва проекти не скомпілюються, на відміну від перелічень з інших мов, де варіант може бути відсутнім. Це допомогає розробнику не забути додати обробку того чи іншого варіанту. Клієнтський застосунок може запросити зміну паролю, перезавантаження серверу, збереження серверної конфігурації, зміну опції стиснення, вибрати мережевий інтерфейс та змінити налаштування відправлення необроблених кадрів. У свою чергу, сервер може підтвердити синхронізацію з клієнтом, надіслати оброблені (або ж ні) дані, поточні налаштування, або підтвердження виконання будь-якої з описаних вище операцій. Окрім цього, повністю описана структура налаштувань, які можуть бути відправлені сервером – абсолютно всі повинні сприйматись клієнтом, інакше знову ж, застосунок не скомпілюється. Також, описані помилки, які можуть виникнути під час процесу обробки запитів, вони теж можуть бути відправлені для повної синхронізації виконання дій.

3.2 Графічний клієнт

Клієнтський застосунок використовує бібліотеку для створення графічних інтерфейсів під назвою «egui». Її особливістю є «негайний» режим, при якому прорахунок графічних кадрів відбувається постійно, на відміну від бібліотек зі звичним «збереженим» режимом, де візуальна частина оновлюється лише якщо змінюється певний компонент, або за певними інструкціями [31]. Можливо цей вибір не є найкращим, тому що зазвичай кожен кадр прораховувати треба в таких застосунках як, наприклад, ігри. Це викликає набагато більше навантаження процесора. Але зважаючи на природу застосунку, де задачею є відображення поточних даних – це можна вважати і перевагою, користувач зможе побачити дані негайно.

Виконання починається зі завантаження конфігурації, в якій описані всі опції, що стосуються застосунку. Якщо файлу немає – він створюється, а якщо ж є, але пошкоджений – виконання завершується з помилкою.

Після цього відбувається завантаження локалізації зі встановленої в конфігурації мови. Застосунок повністю перекладений на англійську та українську мови, і загалом система інтернаціоналізації працює досить просто – там де потрібно відобразити текст, використовуються заздалегідь встановлені ідентифікатори, за якими текст береться з внутрішнього сховища бібліотеки «rust-i18n». Це сховище завантажується з файлів в директорії «locales», через що можливість додавання власної мови зводиться до створення нового файлу, копіювання змісту з, наприклад, англійського перекладу, і згодом перекладу всіх зазначених в ньому фраз. Після чого мова повинна бути зареєстрована в переліченні – на цьому все, можна використовувати застосунок на, як приклад, японській або будь-якій іншій мові, що може привабити користувачів з інших країн.

Надалі відбувається налаштування глобального об’єкту, який займається журналюванням. Формат та рівень також завантажуються з конфігураційного файлу. Якщо ж на цьому етапі виникає помилка – застосунок припиняє роботу.

Нарешті, якщо минулі етапи пройшли успішно – створюється глобальний контекст застосунку, в якому зберігаються канали бібліотеки «crossbeam» для передачі даних в режимі черги скрізь весь застосунок або різні потоки. Також, там зберігається флаг вимкнення застосунку (який використовується для того, щоб коректно завершити роботу всіх потоків), сховище профілів підключення, зміст конфігураційного файлу, структури з поточними налаштуваннями застосунку та серверу (можуть відрізнятись від конфігураційних під час роботи), сховище мережевих метаданих і структура, яка відповідає за синхронізацію між клієнтом та сервером.

В меню аутентифікації є можливість обрати профіль підключення або створити новий – це потрібно для швидкого доступу до серверу. Якщо ж була натиснута кнопка «підключення», відправляється WebSocket запит на з’єднання. Як написано вище, він має містити заголовки про те, що це запит аутентифікації та

містити захешований пароль. Якщо підключення не було успішним – користувач побачить відповідь від серверу, або повідомлення про те, що виникла певна помилка. Якщо все пройшло успішно – запитуються серверні налаштування.

Відразу створюється окремий потік, який відповідає за отримання повідомлень від серверу. Загалом, підключення створюється лише один раз, що є перевагою через обрану технологію. Між потоками дані передаються через відповідні структури – канали. Залежно від налаштувань стискання, прийняті дані можуть бути або в текстовому, або бінарному вигляді – оброблюються вони по різному. Для текстових даних розрахована десеріалізація з JSON формату (і навпаки, при відправленні – всі дані серіалізуються за допомогою бібліотеки «serde»). Якщо ж дані в двійковому вигляді, це означає, що вони стиснуті – після чого вони розтискаються, і згодом оброблюються як і звичайні текстові дані.

Далі, якщо була отримана команда – вони обробляються в окремій черзі, ніж корисне навантаження. Помилка теж є командою. Якщо ж були отримані дані, то вони проходять скрізь власний алгоритм обробки. Спочатку, з них отримуються метадані про розібраний кадр, і додаються у відповідне сховище всього, що стосується швидкості мережі.

В цій структурі відбувається створення масивів, які відповідають за візуалізацію графіку швидкості мережі. З окремих налаштувань береться «вікно» в секундах, за який час повинні відображатись відомості – після чого в масив додаються ці дані, і видаляється все, що застаріло. Підраховується пікова швидкість загалом, отримання та відправки.

Далі, якщо пакет прийшов в розібраному вигляді, він підлягає обробці для відображення в інспекторі. Кожен шар за схемою OSI потрапляє у власний вектор записів, який потім відображається певним чином відносно протоколу у відповідній вкладці.

Якщо ж пакет не був розібраний і прийшов у вигляді байтів (таким чином налаштована взаємодія) – то він потрапляє у вектор нерозібраних кадрів, на довжину якого можна накласти обмеження. Надалі, всі зібрані дані можна зберегти у форматі «.pcap» – це може знадобитись для відлагодження механізму розбору.

Окремо шари Ethernet та IP потрапляють у функцію визначення, чи вони стосуються пристрою з локальної мережі, чи ні. Це потрібно для відображення домашніх девайсів. Якщо пристрій вже є в локальному сховищі – відбувається пошук призначеного для нього імені в структурі HashMap і відображення на головній вкладці і в інспекторі. У випадку відсутності – ім’я відображається з підписом «Пристрій» і відповідним номером.

Окрім цього, MAC-адреса перевіряється у радиксному дереві ідентифікаторів виробників пристроїв. Використовується заздалегідь завантажена текстова база даних від Wireshark, яка зчитується при кожному вході в застосунок. За рахунок особливостей структури, вхід в програму може відбутись зі затримкою, але пошук можна вважати взагалі непомітним.

TCP та UDP шари теж проходять перевірку на співпадіння з іншою текстовою базою даних у форматі «.csv» від організації IANA [32]. Вона теж завантажена заздалегідь і знаходиться в каталозі ресурсів програмного комплексу. Тут вже не потрібне радиксне дерево – використовується звичайний HashMap. Якщо порт джерела або отримання співпадає з певним записом – ця інформація відобразиться в інспекторі як можливий сервіс, з яким працює хтось з мережі.

Окремо можна виділити механізм синхронізації. Користувач на головній панелі може побачити останній час, коли була отримана відповідь щодо того що зв’язок в справному стані, та відправити новий запит. Він може сам встановити це значення в секундах. Якщо ж відповідь не отримана, зв’язок вважається порушеним. Його можна відновити, якщо перепідключитись.

3.3 Сервер

Сервер, як і клієнт, є «бінарним крейтом», але на відміну від нього має консольний інтерфейс, в якому відображаються записані події. Вимкнути його можна за допомогою сигналу SIGINT – тобто, використовуючи комбінацію клавіш «Ctrl-C».

Стартові інструкції застосунку такі ж, як і у клієнта – хіба що, за відсутністю локалізації. Далі створюється глобальний контекст, в якому зберігається початкова конфігурація, параметр стискання повідомлень, зашифрований пароль, «лінк-тайп» (значення, яке визначає тип з'єднання – наприклад, Ethernet) [33], активний мережевий інтерфейс та параметр відправки необроблених до кінця кадрів.

Окремим моментом є створення сховища каналів – BroadcastPool. Це власноруч написана структура даних, що зберігає в собі канали для передачі даних. Потрібна через те, що отримані пакети треба передавати в кожен потік зв'язку. Була також розроблена структура BroadcastChannel – для автоматичної відправки копій даних в ці потоки. Але, копія повинна бути в потоці TCP, який відповідає за перевірку спроб з'єднань, і перевіряти чи вони з'явились – якщо так, то блокувати цю структуру, дістати отримувача даних, передати його в потік. В той же час, при отриманні пакетів канал теж повинен блокуватись для відправки, що створює постійну чергу з блокуванням. Для часткового вирішення цієї проблеми був розроблений так званий «пул каналів», який міститься в обгортці з атомарного лічильника повідомлень і не простого ексклюзивного м'ютекса, а RWLock – який дозволяє блокувати на читання відразу декільком джерелам, а на запис лише одному [23]. Таким чином, якщо створений новий потік для з'єднання, лише тоді і тільки тоді читач пакетів блокує на запис це сховище, забирає канал «відправник», і передає його в широкомовний канал. А в потоці який простіше називати «TCP-Thread», це сховище теж блокується лише при створенні з'єднання. Таким чином вирішується проблема частих ексклюзивних блокувань, які за якийсь час створюють значну затримку.

Разом зі створенням пулу каналів та контексту створюється ще й флаг для коректного завершення всіх потоків (атомарний булева змінна), і атомарна цілочисельна змінна для підрахунку поточних з'єднань. Вона потрібна, щоб не завантажувати зайвий раз мережеву плату – якщо клієнтів немає, то сервер буде лише очікувати вхідні спроби, без прослуховування мережі.

Копії цих змінних, обгорнуті в атомарний лічильник, передаються в два нові потоки – вже вище згаданий «TCP-Thread», метою якого є очікування спроб

з’єднання, та «Sniffer-Thread» – відповідає за перехват кадрів. Основний же залишається на блокуюче очікування їх завершення, при цьому не навантажуючи процесор – через реалізацію в Rust, потік «засипає», не витрачаючи процесорні такти.

Прослуховувач TCP-з’єднань, відповідно, очікує надходження повідомлень за вказаною в конфігурації IP-адресою та портом в неблокуючому режимі. Якнайбільше операцій в застосунку намагаються використовувати неблокуючий режим – це пов’язано з тим, що флаг вимкнення застосунку треба періодично перевіряти за допомогою Acquire завантаження. Якщо ж була спроба під’єднатися, лічильник потоків збільшується на 1, та копії контексту, флагу і каналу «отримувача» даних передаються в новостворений потік для WebSocket-з’єднання. Цикл продовжується далі з паузами в 10 мілісекунд «сну».

Обробка та відправка повідомлень теж працюють в циклі – допоки не зміниться значення атомарного флагу вимкнення застосунку. Запит на це відстежується в основному потоці – якщо був надісланий сигнал SIGINT, значення змінюється. В середині циклу зі затримкою 10 мілісекунд перевіряється, чи щось надійшло в канал «отримувач» даних. Якщо так – ці дані додаються у відповідну чергу, і далі йде спроба отримати дані ще раз у розмірі 100 пакетів – вважаючи, що пакет зазвичай не приходить в одному екземплярі, у випадку з, наприклад, TCP. Все що було отримане, оброблюється по тому ж алгоритму, що й на клієнті, і згодом відправляється. Після цього йде перевірка, чи надійшли певні команди на сервер – якщо так, то в окремому модулі розписані алгоритми їх обробки, після чого відповідь теж додається до черги відправки.

Що ж до «Sniffer-Thread», алгоритм теж працює циклічно. Окрім перевірки флагу вимкнення, також повинна виконуватись умова, що кількість з’єднань більше нуля – для уникнення безцільної роботи мережевої карти та розбору протоколів. Якщо ж всі умови виконуються, виконується спроба отримання кадру та його розбір за алгоритмом парсингу, що знаходиться в окремій бібліотеці. Копії результату відправляються в потоки з’єднань, після чого потрапляють на клієнт (рисунок 3.2). У випадку відсутності результату, що може статися якщо парсер

зовсім не зміг обробити фрейм, і при цьому в налаштуваннях встановлено, що відправляти необроблені кадри не потрібно – цикл просто продовжує працювати далі.

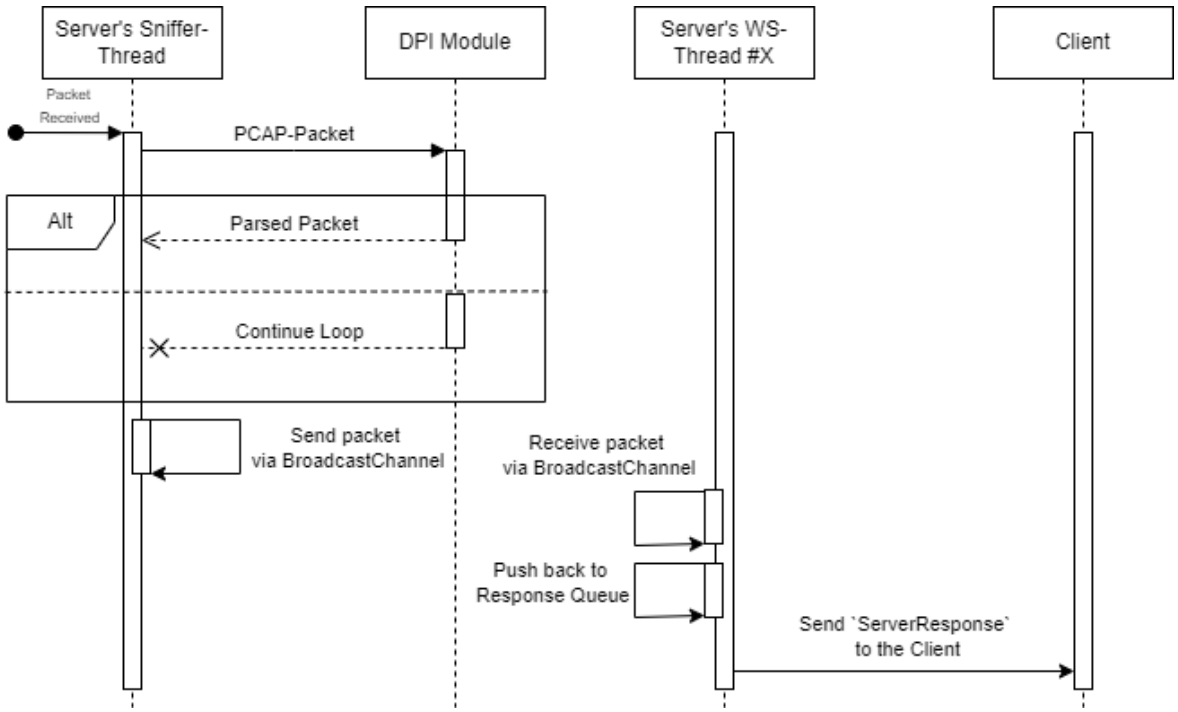

Рисунок 3.2 – Шлях отриманого пакету

Саме таким чином забезпечується робота сервера, задачею якого є підтримання декількох з'єднань і негайна відправка даних клієнтським застосункам. Злагоджена робота двох потоків та зв'язок між ними за допомогою самописних структур «широкомовний канал» та «широкомовне сховище» дозволяють відмінно виконати задачу, без зайвих ексклюзивних блокувань.

3.4 Бібліотека розбору даних

Проект з назвою «dpi» складається з двох частин (рисунок 3.3): «аналіз» та механізми для відтворення процесу «Deep Packet Inspection», або просто DPI, суть якого полягає в отриманні корисної інформації з кожної мережевої одиниці.

Наприклад, це може бути не тільки розбір заголовків – може аналізуватись зміст пакетів HTTP, тощо.

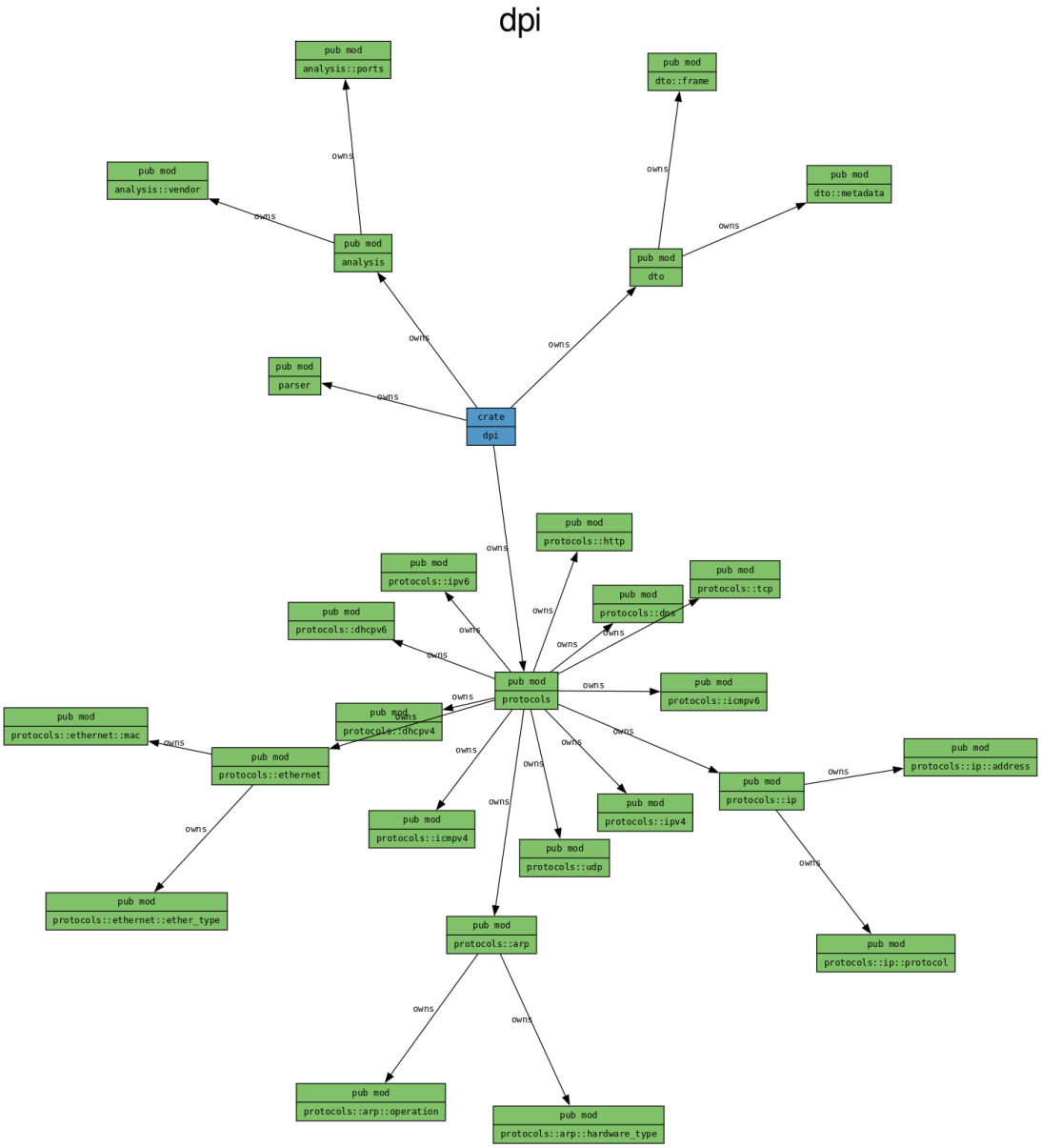

Рисунок 3.3 – Структура бібліотеки DPI

В частині «аналізу» знаходяться два модулі, які відповідають за завантаження в систему баз даних сервісів, закріплених за портами, і розробників пристроїв відповідно OUI. Також, є додаткові структури, в яких зручно передавати отриману інформацію – DTO (Data Transfer Object).

В іншій частині містяться, перш за все, структури-обгортки пакету від бібліотеки pcap – це потрібно через те, що зазвичай інформація отримується за посиланням, через що нею не зручно оперувати, і неможливо відправити по мережі згідно поточної реалізації. Потрібні власні структури, які «володіють» цією інформацією згідно концепції володіння в Rust, і вони повинні реалізовувати інтерфейси з бібліотеки serde для серіалізації даних.

Модуль «парсер» містить структуру, яка складається з поля-параметру чи потрібно віддавати початкові байти, якщо їх не вийшло обробити, і наявності «корінного» протоколу, яким в даному випадку є Ethernet. Протокол визначається через вищезгаданий «лінк-тайп» – це значення задається згідно мережевого інтерфейсу. Далі викликається метод обробки, в якому перш за все створюється об’єкт збору всього, що вдасться отримати. Як мінімум, це буде інформація з заголовку об’єкту пакету – кількість байтів та час, коли він був створений. Після цього, якщо є корінний протокол, відтворюється рекурсивний алгоритм визначення протоколів. В функцію передається поточний протокол, рівень рекурсії (максимальний – 16), зріз байтів та зібрана інформація. Для поточного протоколу викликається відповідний заздалегідь написаний метод-парсер. Якщо байтів після цього не залишилось – повертається значення «Завершено», протокол не підтвердився – «Помилка». У випадку, якщо результат успішний, але ще залишились байти, обирається «найкращий нащадок» згідно отриманого шару. Наприклад, якщо на поточному рівні був отриманий заголовок Ethernet, наступний протокол однозначно визначається за полем EtherType, а для протоколу TCP можна зробити припущення за номером порту, умовно 53 – це скоріше за все DNS. Якщо ж найкращого немає, циклічно перебираються всі можливі нащадки, які також вказані заздалегідь. Тобто, після умовного HTTP не може відбутися такого, що алгоритм пробує відтворити IPv4. Але тоді виникає питання – навіщо ж рекурсія? Відповідь проста – можуть виникати різні випадки, найбільш простим прикладом яких є тунелювання. Корисним навантаженням протоколу IP може бути вкладений IP – і так декілька разів. Взагалі, різні протоколи можуть мати неочікувані вкладені дані – тому, список можливих наступників має бути якомога розлогим.

Якщо після перебору нащадків позитивний результат не був отриманий, це означає що або відповідного протоколу і алгоритму його розбору немає в кодовій базі, або дані взагалі не можна розібрати. Таке також іноді буває, і для уникнення треба відтворювати певні механізми, такі як TCP Reassembling – перезбірку TCP фрагментів. Так як цей протокол є потоковим, корисне навантаження може прийти в неповному вигляді [34], і більш того – в невірному порядку. В операційній системі вирішення таких ситуацій займається абстракція TCP – механізм розбору, який повністю реалізований згідно стандарту RFC. Використання таких механізмів є недоцільним при DPI, або як мінімум має бути опціональним через велике споживання пам'яті. У Wireshark є реалізація цього алгоритму, але її використання безпечне при розборі .pcap файлів, а при потоковій передачі, чим і користується розроблюваний комплекс, реалізація не рекомендується.

Далі, згідно налаштування парсеру, назад до процесу серверу передається один з видів готової інформації. Це або вектор байтів, або просто дані про пакет, або ж ті ж дані з вектором розібраних шарів.

Щоб додати власну реалізацію відтворення структури заголовку протоколу в кодову базу, треба створити підмодуль в модулі «protocols», додати назву в перелічення ProtocolId, вказати яка функція з новоствореного модулю відповідає за парсинг (сигнатура заздалегідь визначена), і за наявності вказати відповідник портів, найкращого нащадка і їх список. Наразі присутні протоколи ARP, DHCPv4, DHCPv6, DNS, Ethernet, HTTP, ICMPv4, ICMPv6, IPv4, IPv6, TCP та UDP.

Також, згідно есе Едгера Декстри «Скромний програміст» [35], тестування програм може бути ефективним шляхом показати присутність помилок, але зовсім неадекватним шляхом показати їх відсутність». Але при цьому, за його словами, це не означає, що не треба тестувати якомога більше – а навпаки. Коли можна перекласти це на плечі компілятору тощо, то краще зробити так, це і є перевагою мови Rust, як приклад. Але у випадку логічного співставлення байтових даних зі структурами тих чи інших протокольних заголовків, компілятор не зможе допомогти уникнути помилок, так як це бізнес-логіка. Саме для цього в кожному модулі написані тести, як мінімум 3 на кожен протокол. Завдяки жорсткій

зв’язності механізму розбору, якщо існують помилки, скоріше за все вони відобразяться й на інших з поточних 12 алгоритмів. Зазвичай, це є мінусом, але в цьому випадку це є значною перевагою. Помилки відразу виявляються і не зможуть нашкодити й згодом фальсифікувати результат.

Підсумовуючи, можна сказати, що був розроблений ефективний алгоритм для глибокого аналізу пакетів. Він був протестований та відлагоджений, і певні рішення, такі як використання бібліотеки nom лише відточили загальну структуру. Код є чистим і зрозумілим, як і обробка помилок, при цьому алгоритм не викликає значне зростання розміру фінального бінарного файлу. За допомогою концепції «переходу» власності змінних Rust, зменшується копіювання даних, а ця проблема може продовжити час виконання алгоритму в 2.8 разів [36]. Також, можна зазначити, що модульна структура дозволяє в майбутньому й це навантаження зняти з маршрутизатору – він може надсилати лише отримані дані, а подальший аналіз став би зоною відповідальності клієнтського застосунку.

4 РОБОТА КОРИСТУВАЧА З ПРОГРАМНИМ КОМПЛЕКСОМ

Взаємодія користувача з системою відбувається за допомогою кросплатформного застосунку (рисунок 4.1), який включає низку функціональних інструментів та вкладок з інформацією. Цей підхід дозволяє користувачам ефективно використовувати користуватись системою, не маючи глибоких знань в комп’ютерних мережах.

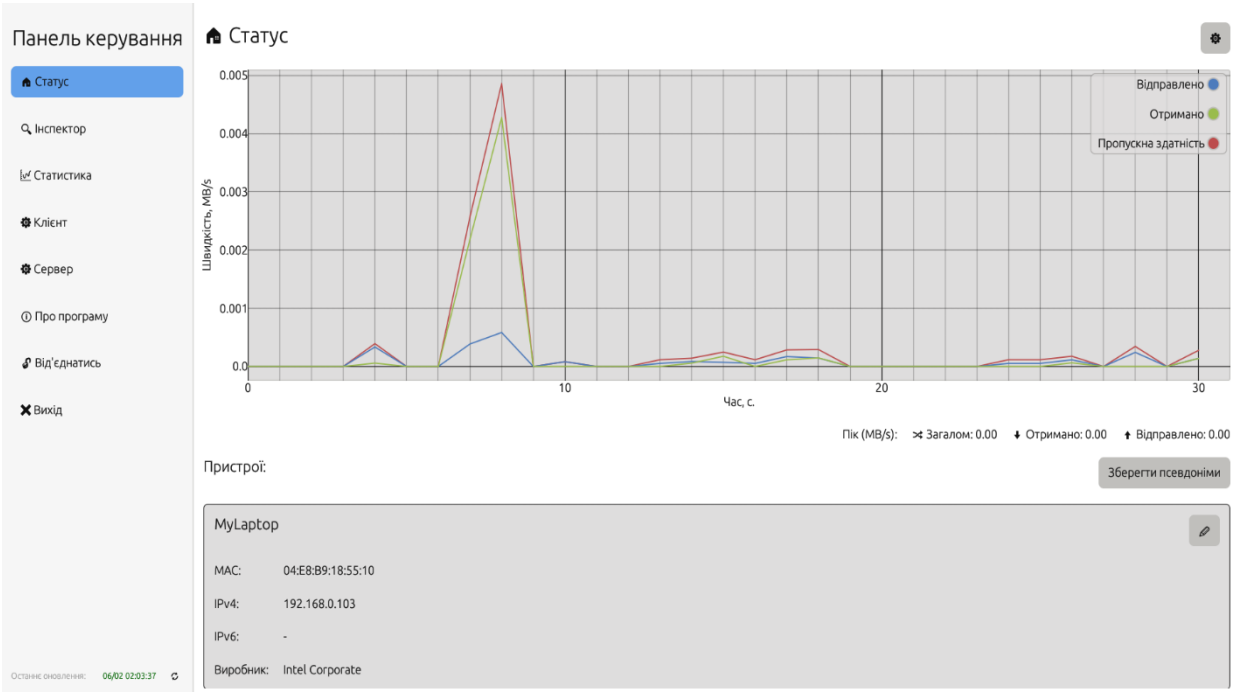

Рисунок 4.1 – Інтерфейс застосунку

Головними елементами інтерфейсу є панель керування, яка знаходиться зліва, і робочий простір, який займає всю іншу частину екрану. На ньому відображається зміст вкладки, обраної в панелі керування. Всього в наявності 8 вкладок, а саме «Статус», «Інспектор», «Статистика», «Клієнт», «Сервер», «Про програму», «Від’єднання», «Вихід».

4.1 Інструкції зі встановлення та запуску

Застосунок та сервер можна використовувати на операційних системах Windows, Unix, MacOS. На GitHub сторінці проекту є заздалегідь скомпільовані бінарні файли. Загалом, у кожної версії застосунку є в наявності 6 архівів: 3 клієнти і 3 сервери для Windows, Ubuntu та MacOS. Наявність заздалегідь скомпільованих архівів гарантована – вони автоматично генеруються при створенні версії за допомогою Github Actions.

Для роботи потрібно завантажити бажані версії компонентів та розпакувати їх. У випадку з сервером, в архіві присутній скриптовий файл (залежно від цільової системи – bash shell або powershell скрипт). Він необхідний для автоматичного перезапуску застосунку, якщо за певних причин він вимкнеться. Це може статись не лише через помилку – функція «Перезапустити сервер» на клієнті працює саме за допомогою скрипту.

Щодо клієнта – в архіві знаходиться бінарний файл «client.exe», та 3 каталоги які необхідні для його роботи – ресурси, зображення та локалізаційні файли. Розпакований каталог можна перемістити в бажане місце. Конфігураційні файли клієнту створяться після першого запуску, і будуть знаходитись або за шляхом «%APPDATA%/xairaven/xailyser», або безпосередньо в каталозі зі застосунком. Конфігураційний файл серверу буде створений в каталозі з ним.

4.2 Опис інтерфейсу

Сервер має консольний інтерфейс, в якому відображаються всі події, зафіксовані під час роботи. Рівень журналювання можна змінити в конфігураційному файлі застосунку, вказавши бажаний рівень деталізації (TRACE, DEBUG, INFO, WARN, ERROR). Але, використання рівнів нижче «INFO» не рекомендується – це може забрати значну частку ресурсів лише на відображення тексту, і загалом, завантажити застосунок консолі.

Якщо ж запустити клієнтський додаток, то користувача зустрічає сторінка авторизації (рисунок 4.2):

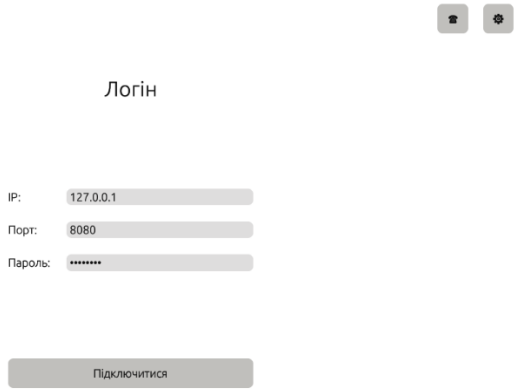

Рисунок 4.2 – Сторінка авторизації

Для підключення треба заповнити відповідні відповідні поля: IP-адресу, порт (який «слухає» сервер), та пароль з конфігураційного файлу серверу. Вони повністю валідуються – ввести дані не за форматом не вийде.

Також, прямо з цієї сторінки можна перейти в клієнтські налаштування або в меню швидкого вводу (рисунок 4.3):

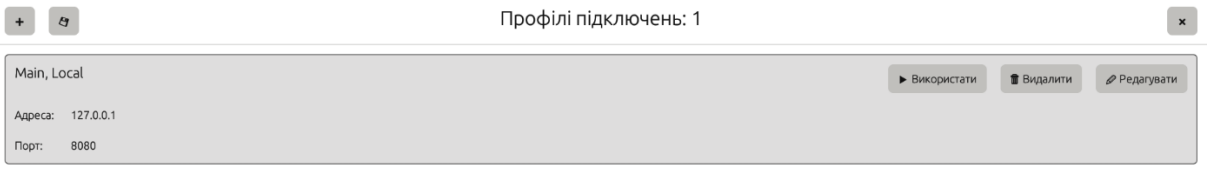

Рисунок 4.3 – Налаштування профілів входу

Воно використовується для збереження варіантів входу. Можна використати вже наявний профіль (поля з авторизаційної сторінки заповнюються автоматично), зберегти профілі, переглянути їх, видалити або відредагувати. Редагування відбувається за допомогою модального вікна з потрібними для цього полями.

Після авторизації користувача зустрічає вкладка «Статус». На ній можна перевірити швидкість мережі в графічному вигляді (загальний потік, надходження та відправлення), і найвищі значення цих параметрів за певний проміжок часу (рисунок 4.4):

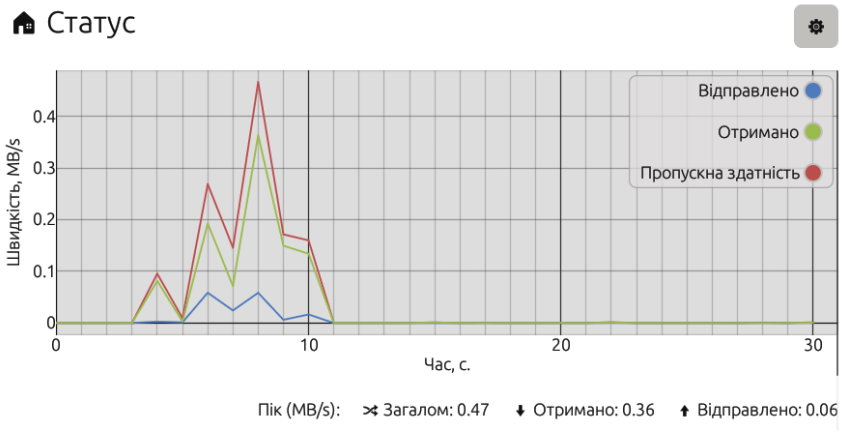

Рисунок 4.4 – Графік швидкості мережі

Трохи нижче знаходиться перелік відстежених пристроїв з локальної мережі (рисунок 4.5). Відображається основна інформація: MAC-адреса, IP-адреси 4 та 6 версій, та виробник, який визначається за допомогою ідентифікатору.

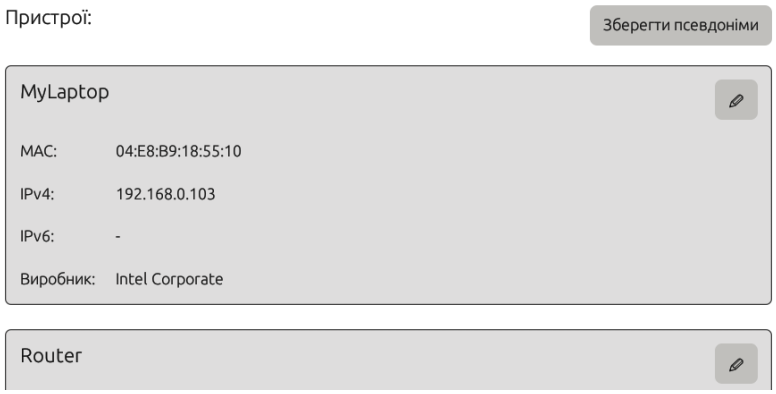

Рисунок 4.5 – Список пристроїв

Їм можна надати псевдоніми, якщо натиснути на кнопку «редагувати» – але потім треба обов’язково зберегти зміни.

Є можливість змінити налаштування графіка швидкості мережі. Для цього знадобиться кнопка «Налаштування», яка знаходиться справа від нього. В меню можна обрати період показу (зміняться відповідно і пікові значення) та одиниці швидкості (рисунок 4.6):

Рисунок 4.6 – Налаштування графіку швидкості мережі

Наступною вкладкою є «Інспектор» (рисунок 4.7), за допомогою якого можна переглянути дані за різними протоколами. Можна обрати протокол, очистити дані лише за обраним або за всіма, а також переходити по сторінкам (розмір однієї – 100 записів).

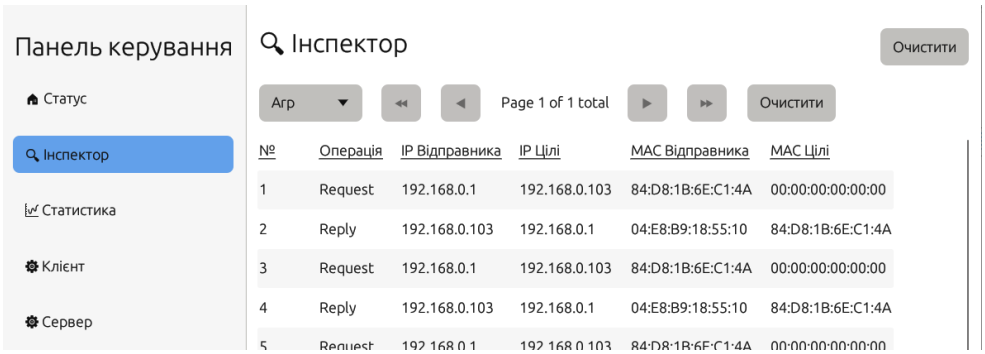

Рисунок 4.7 – Інспектор

Поля таблиці відрізняються згідно формату протоколу. Загалом в наявності 12 протоколів. DNS та HTTP мають формат у вигляді вкладок, а не таблиці. Це пов’язано з тим, що дані мають комплексний зміст.

На вкладці «Статистика» (рисунок 4.8) відображається статистична інформація про застосунок, а саме кількість записів, захоплених фреймів та інші дані. Також, можна переглянути кількість записів відповідно кожному з протоколів.

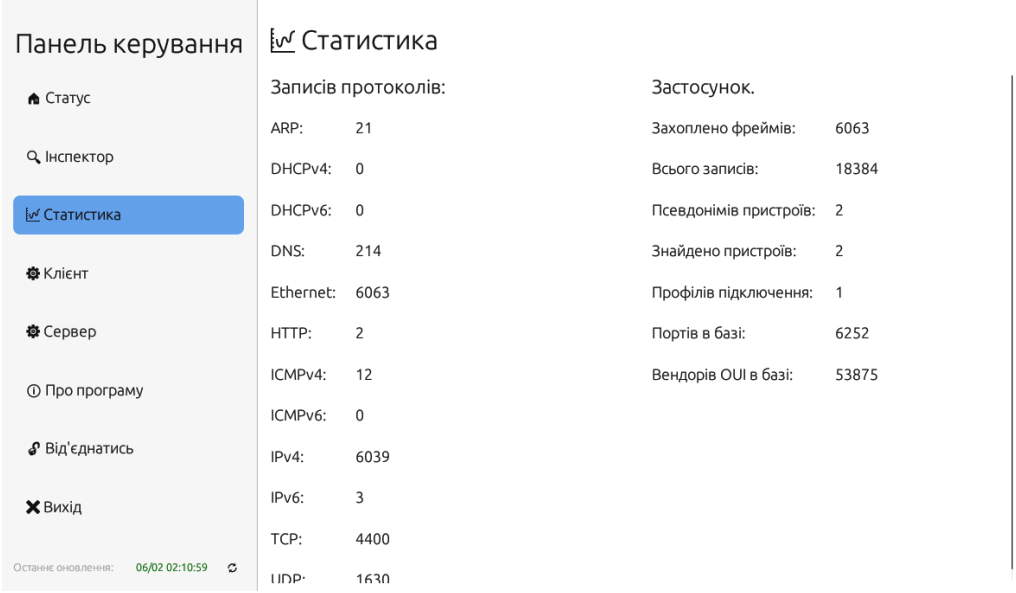

Рисунок 4.8 – Статистика

В додатку багато речей можна налаштувати, основні опції знаходяться на вкладці «Клієнт»:

– можна відкинути нерозібрані пакети, тобто якщо опція увімкнена, застосунок не буде зберігати кадри, які не були повністю розібрані на рівні схеми OSI, а розмір зарахується в загальну пропускну здатність;

– є можливість обрати час в секундах, після якого буде відправлятись запит на синхронізацію (пінг);

– якщо були змінені певні налаштування, потрібно натиснути на відповідну кнопку, щоб вони були збережені у файл;

– можливо увімкнути ліміт збереження нерозібраних фреймів, щоб вони не займали багато місця в оперативній пам’яті;

– є окрема опція для ліміту розібраних фреймів, щоб зберігати обмежену кількість записів з інспектору;

– додаток можна використовувати двома мовами, а саме українською та англійською, і надалі можливе додавання й інших мов, для цього треба створити лише один файл та перекласти всі рядки в ньому;

– в наявності окремі опції які пов’язані з журналюванням, а саме регулювання рівню (Вимкнено, Помилки, Попередження, Інформація, Дебаг, Трейс) та формату, бо такі параметри як дата, рівень, модуль та повідомлення є опціональними;

– якщо повідомлення приходять рідко, а пропускна полоса доволі мала, доцільно увімкнути стиснення повідомлень, але це також треба зробити і в серверних налаштуваннях;

– згідно уподобань користувача є 7 тем для інтерфейсу.

Окремо виділені налаштування серверу (рисунок 4.9). Їх можна змінювати як в конфігураційному файлі, так і прямо з клієнтського застосунку.

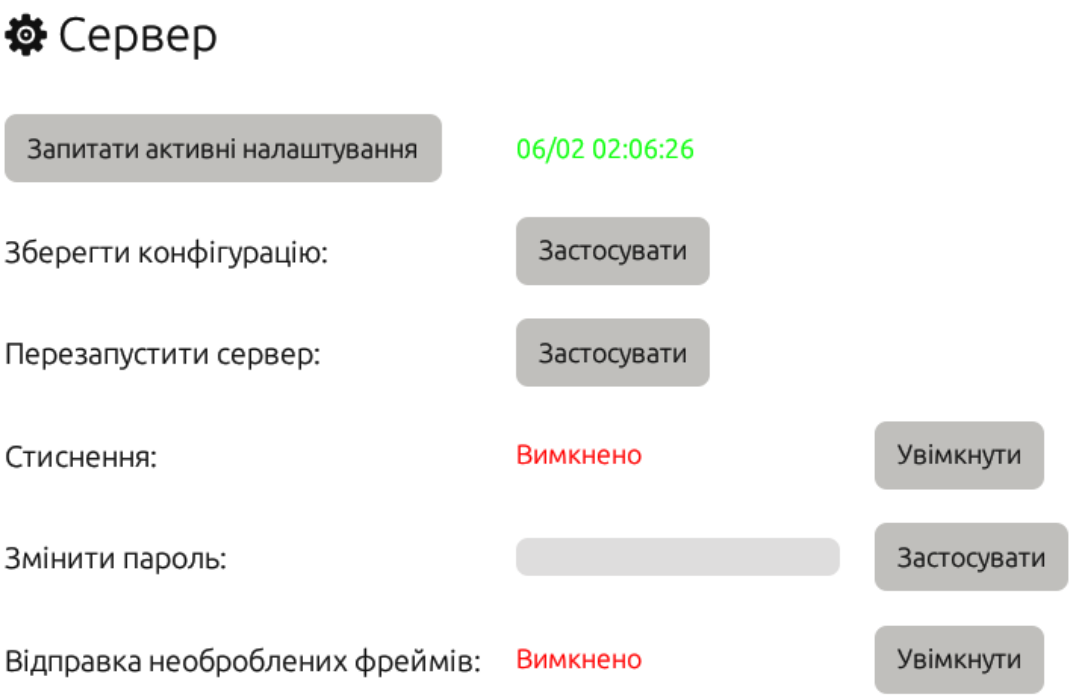

Рисунок 4.9 – Налаштування серверу

Можна запитувати активні налаштування (збоку позначається час та успішність спроби), зберігати конфігурацію, перезапускати сервер, змінювати пароль, вмикати або вимикати стиснення та відправку необроблених кадрів.

Окрім цього, є можливість зміни мережевого інтерфейсу (рисунок 4.10). Після запиту налаштувань відображаються активний, встановлений та доступні інтерфейси.

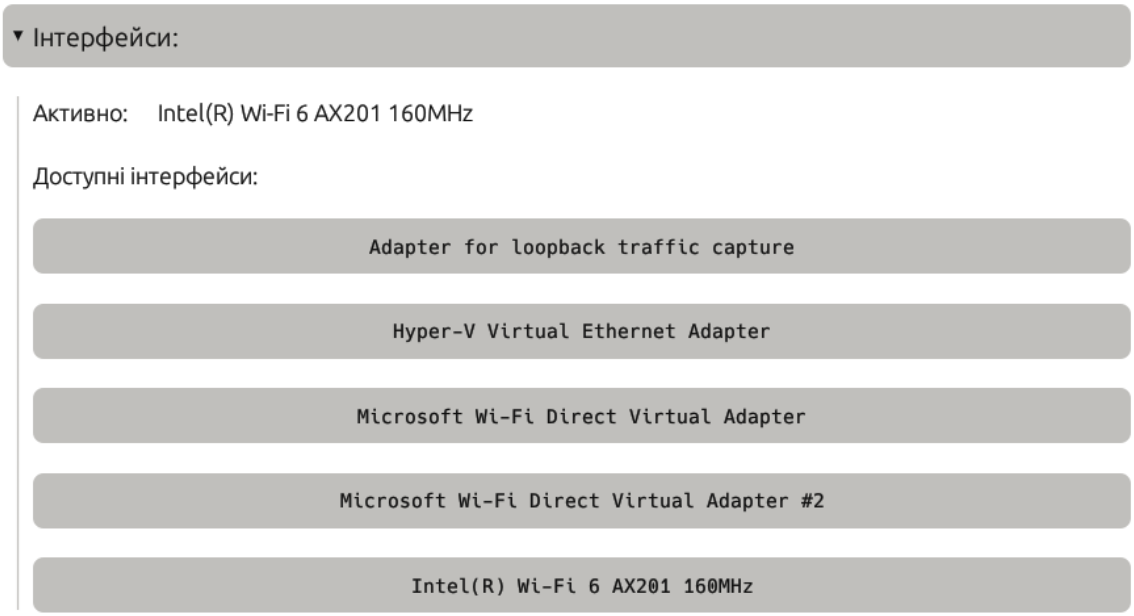

Рисунок 4.10 – Вибір мережевого інтерфейсу

Якщо інтерфейс був змінений, то для того щоб зміни вступили в силу, треба зберегти конфігурацію та перезапустити сервер з використанням відповідних налаштувань. Після цих дій, у комірці «Активно» повинен відображатись обраний адаптер.

4.3 Тестування системи

Як приклад, продемонстровано використанні при потоці трафіку близько 4 мегабайт в секунду.

В «debug» режимі збірки, при зібраних 100000 кадрах (418932 записи) – споживання оперативної пам’яті застосунку складає 99 мегабайт. Серверу – 2 МБ, що повністю задовольняє вимоги.

В збірці «release» застосовуються оптимізації компілятору, через що споживання повинно стати набагато меншим. Але, після тестування, результати виявились ідентичними (рисунок 4.11).

Name	Status	42% CPU	81% Memory	0% Disk	0% Network
client.exe		28.1%	98.1 MB	0 MB/s	0 Mbps
server.exe		0%	2.3 MB	0 MB/s	0 Mbps

Рисунок 4.11 – Демонстрація отриманих результатів

Перед тестуванням кінцевої версії, були проведені тести бібліотеки dpi. Всі тести виявились успішними (рисунок 4.12).

Рисунок 4.12 – Модульні тести

Відповідно, система повністю виконує всі поставлені нефункціональні вимоги. Для запуску на маршрутизаторі, треба скомпілювати проект під бажану архітектуру, та перенести на пристрій бінарний файл і відповідний скрипт. Це можна зробити за допомогою ssh, або іншого способу за бажанням користувача.

ВИСНОВКИ

У процесі формулювання мети даного дипломного проекту було поставлено завдання розробити інструмент для моніторингу домашньої мережі з глибокою інспекцією пакетів і максимально простим інтерфейсом для кінцевого користувача. Потенційний оператор системи повинен мати базові знання моделі OSI та ключових протоколів TCP/IP, але без глибокого занурення у низькорівневі аспекти роботи комп'ютерних мереж. Інтерфейс покликаний відображати результати розбору заголовків у вигляді зрозумілих графіків і метрик, а також забезпечувати можливість маркування пристроїв для зручної ідентифікації.

Основним апаратним орієнтиром стала модель домашнього маршрутизатора з обмеженими ресурсами: TP-Link Archer A64 із 16 МБ флеш-пам'яті, 128 МБ оперативної пам'яті та двоядерним процесором MediaTek MT7621. Саме ці характеристики лягли в основу суворих лімітів на обсяг використовуваної пам'яті (не більше 10 МБ) та навантаження на процесор, що вимагало ретельної оптимізації серверної складової та модулю інспекції трафіку.

Особлива увага приділялася забезпеченню стабільності: обробка виняткових ситуацій передбачена на кожному етапі, а критичні модулі тестувалися з максимально можливим охопленням граничних випадків. Найбільш відомі вразливості були виключені за допомогою використання мови програмування Rust.

Завдання було успішно виконане, продукт готовий до використання та є фундаментом для можливої подальшої розробки. Система була розроблена так, що додавати нові модулі доволі просто. Окрім цього, проаналізовані підходи до створення систем такого типу. Обґрунтовано та досліджено використання різноманітних побічних інструментів та здійснено апробацію на конференції «Сучасні проблеми наукового забезпечення енергетики».

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Home use wi-fi router global market report 2025. The Business Research Company. URL: https://www.thebusinessresearchcompany.com/report/home-use-wi-fi-router-global-market-report (дата звернення: 10.04.2025).

2. TP-Link archer A6 specs. OpenWRT Docs. URL: https://openwrt.org/toh/hwdata/tp-link/tp-link_archer_a6_v3 (дата звернення: 26.04.2025).

3. Wireshark internals. Wireshark Docs. URL: https://wiki.wireshark.org/libpcap (дата звернення: 07.03.2025).

4. TShark overview. Tshark Documentation. URL: https://tshark.dev/setup/about/ (дата звернення: 28.03.2025).

5. Hardware sizing. ntop. URL: https://www.ntop.org/guides/ntopng/performances/hardware_sizing.html (дата звернення: 28.04.2025).

6. Netflow export & analyses. OPNsense documentation. URL: https://docs.opnsense.org/manual/netflow.html (дата звернення: 13.05.2025).

7. PfSense minimum hardware requirements. Netgate Documentation. URL: https://docs.netgate.com/pfsense/en/latest/hardware/minimum-requirements.html (дата звернення: 07.05.2025).

8. IPFire documentation. IPFire.org. URL: https://www.ipfire.org/docs/what-is-ipfire (дата звернення: 22.04.2025).

9. OpenWRT supported devices. OpenWrt Documentation. URL: https://openwrt.org/supported_devices (дата звернення: 05.05.2025).

10. Cybersecurity and Infrastructure Security Agency. Exploring memory safety in critical open source projects. www.cisa.gov. URL: https://www.cisa.gov/sites/default/files/2024-06/joint-guidance-exploring-memory-safety-in-critical-open-source-projects-508c.pdf (дата звернення: 27.02.2025).

11. About valgrind. Valgrind Documentation. URL: https://valgrind.org/docs/manual/manual-intro.html#manual-intro.overview (дата звернення: 19.04.2025).

12. Rust won’t save us: an analysis of 2023’s known exploited vulnerabilities. Horizon3.ai. URL: https://horizon3.ai/attack-research/attack-blogs/analysis-of-2023s-known-exploited-vulnerabilities/#:~:text=Rust%20Won't%20Save%20Us,%20But%20It%20Will%20Help%20Us,0-days%20by%20threat%20actors (дата звернення: 14.05.2025).

13. Lifetimes - the rustonomicon. Rust Documentation. URL: https://doc.rust-lang.org/nomicon/lifetimes.html (дата звернення: 08.02.2025).

14. The Improvement of Network Performance by Using the Technique of PF-RING Zero-copy to Optimize Spark Streaming / F. LIU et al. International Conference on Computer Networks and Communication Technology (CNCT 2016), Xiamen, China, 16–18 December 2016. Paris, France, 2017. URL: https://doi.org/10.2991/cnct-16.2017.9 (дата звернення: 25.02.2025).

15. eBPF Foundation. EBPF documentation. URL: https://ebpf.io/what-is-ebpf/ (дата звернення: 22.02.2025).

16. Capturing packets in C program using libpcap. Open Source For You. URL: https://www.opensourceforu.com/2011/02/capturing-packets-c-program-libpcap/ (дата звернення: 24.03.2025).

17. 5 essential patterns of software architecture. Red Hat. URL: https://www.redhat.com/en/blog/5-essential-patterns-software-architecture (дата звернення: 20.02.2025).

18. WebAssembly documentation. MDN Web Docs. URL: https://developer.mozilla.org/en-US/docs/WebAssembly (дата звернення: 21.03.2025).

19. Axum - rust. Docs.rs. URL: https://docs.rs/axum/latest/axum/ (дата звернення: 22.04.2025).

20. RFC 6455: the websocket protocol. IETF Datatracker. URL: https://datatracker.ietf.org/doc/html/rfc6455 (дата звернення: 20.03.2025).

21. Tokio - An asynchronous Rust runtime. Tokio Docs. URL: https://tokio.rs (дата звернення: 22.04.2025).

22. An introduction to advanced Rust traits and generics. Shuttle. URL: https://www.shuttle.dev/blog/2024/04/18/using-traits-generics-rust (дата звернення:

23.05.2025).

23. Rust atomics and locks: low-level concurrency in practice. O'Reilly Media, Incorporated, 2022 (дата звернення: 01.05.2025).

24. Identify what's causing segmentation faults (segfaults). Indiana University. URL: https://servicenow.iu.edu/kb?id=kb_article_view&sysparm_article=KB0022598 (дата звернення: 03.05.2025).

25. Missing optimizations with slices. Rust Internals. URL: https://internals.rust-lang.org/t/missing-optimizations-with-slices/8684/3 (дата звернення: 05.05.2025).

26. Hoare T. Null references: the billion dollar mistake. InfoQ. URL: https://www.infoq.com/presentations/Null-References-The-Billion-Dollar-Mistake-Tony-Hoare/ (дата звернення: 08.05.2025).

27. Ieee 802.3 ethernet. IEEE802. URL: https://www.ieee802.org/3/ (дата звернення: 11.04.2025).

28. Applied network security monitoring. Elsevier, 2014. URL: https://doi.org/10.1016/c2013-0-00546-4 (дата звернення: 09.05.2025).

29. Deri L. FOSDEM 2022 - network traffic classification for cybersecurity and monitoring. FOSDEM. URL: https://archive.fosdem.org/2022/schedule/event/using_ndpi_to_efficiently_classify_network_traffic/ (дата звернення: 09.02.2025).

30. Facade. Refactoring and Design Patterns. URL: https://refactoring.guru/design-patterns/facade (дата звернення: 13.03.2025).

31. Immediate mode vs retained mode. oandre.gal. URL: https://oandre.gal/concepts/immediate-mode-vs-retained-mode/ (дата звернення: 02.05.2025).

32. Service name and transport protocol port number registry. Internet Assigned Numbers Authority. URL: https://www.iana.org/assignments/service-names-port-numbers/service-names-port-numbers.xhtml (дата звернення: 01.05.2025).

33. Link-layer header types. TCPDUMP. URL: https://www.tcpdump.org/linktypes.html (дата звернення: 15.01.2025).

34. Segmentation explained with TCP and UDP header. ComputerNetworkingNotes.

URL: https://www.computernetworkingnotes.com/ccna-study-guide/segmentation-explained-with-tcp-and-udp-header.html (дата звернення: 01.05.2025).

35. E.W.Dijkstra archive: the humble programmer (EWD 340). Department of Computer Science. URL: https://www.cs.utexas.edu/~EWD/transcriptions/EWD03xx/EWD340.html (дата звернення: 17.04.2025).

36. Tanenbaum A. S. Computer networks. Pearson Education, Limited, 2010. 960 с.

ДОДАТОК А

Реалізація головного алгоритму розбору протоколів

Програмні засоби:

– мова програмування – Rust;

– середовище розробки – RustRover.

Програмний код файлу dto::parser.rs:

```
use crate::dto::frame::{FrameMetadata, FrameType, OwnedFrame};
use crate::protocols::{ProtocolData, ProtocolId};
use nom::IResult;
use nom::Parser;
use nom::bytes::take;
use nom::number::be_u8;

pub struct ProtocolParser {
    raw_needed: bool,
    root: Option<ProtocolId>,
}

impl ProtocolParser {
    pub fn new(link_type: &pcap::Linktype, raw_needed: bool) -> Self {
        Self {
            raw_needed,
            root: ProtocolId::root(link_type),
        }
    }

    pub fn process(&self, packet: pcap::Packet) -> Option<FrameType> {
```

```
            let mut metadata = FrameMetadata::from_header(packet.header);

            if let Some(root_protocol) = &self.root {
                let result = traversal(root_protocol, &packet, &mut metadata, 0);
                return match result {
                    ProcessResult::Complete                                    =>
Some(FrameType::Metadata(metadata.into())),
                    ProcessResult::Incomplete => match self.raw_needed {
                        true => Some(FrameType::Raw(OwnedFrame::from(packet))),
                        false => Some(FrameType::Metadata(metadata.into())),
                    },
                    ProcessResult::Failed => match self.raw_needed {
                        true => Some(FrameType::Raw(OwnedFrame::from(packet))),
                        false => Some(FrameType::Header(metadata.header)),
                    },
                };
            }

            None
        }
    }

    fn traversal(
        id: &ProtocolId, bytes: &[u8], metadata: &mut FrameMetadata, depth: usize,
    ) -> ProcessResult {
        const MAX_DEPTH: usize = 16;
        if depth > MAX_DEPTH {
            return ProcessResult::Failed;
        }
```

```
let result = id.parse()(bytes);

match result {
    Ok(([], layer)) => {
        metadata.layers.push(layer);
        ProcessResult::Complete
    },
    Ok((rest, layer)) => {
        metadata.layers.push(layer);

        if let Some(best) = id.best_children(metadata) {
            return match depth.checked_add(1) {
                Some(new_depth) => traversal(&best, rest, metadata, new_depth),
                None => ProcessResult::Failed,
            };
        }

        let children = match id.children() {
            Some(value) => value,
            None => {
                return ProcessResult::Incomplete;
            },
        };

        for id in children {
            let result = match depth.checked_add(1) {
                Some(new_depth) => traversal(&id, rest, metadata, new_depth),
                None => return ProcessResult::Failed,
            };
```

```
                match result {
                    ProcessResult::Complete | ProcessResult::Incomplete => {
                        return result;
                    },
                    ProcessResult::Failed => continue,
                }
            }

            ProcessResult::Incomplete
        },
        Err(_) => ProcessResult::Failed,
    }
}

pub fn wire_format(input: &[u8]) -> IResult<&[u8], String> {
    let mut labels = Vec::new();
    let mut rest_buffer = input;
    while !rest_buffer.is_empty() {
        let (rest, len_byte) = be_u8().parse(rest_buffer)?;
        // Null-terminator
        if len_byte == 0 {
            debug_assert!(rest.is_empty());
            rest_buffer = rest;
            break;
        }

        // Unexpected length
        if len_byte as usize > rest.len() {
            return Err(ParserError::ErrorVerify.to_nom(input));
        }
```

```
        // Creating label
        let (rest, label): (&[u8], &[u8]) = take(len_byte).parse(rest)?;
        let label = String::from_utf8(label.to_vec())
            .map_err(|_| ParserError::ErrorVerify.to_nom(input))?;
        labels.push(label);

        rest_buffer = rest;
    }

    Ok((rest_buffer, labels.join(".")))
}

pub fn cast_to_bool(bit: u8) -> Result<bool, ParserError> {
    match bit {
        0 => Ok(false),
        1 => Ok(true),
        _ => Err(ParserError::ErrorVerify),
    }
}

pub type ParseFn = fn(&[u8]) -> IResult<&[u8], ProtocolData>;
pub type PortFn = fn(u16, u16) -> bool;

pub enum ParserError {
    ErrorVerify,
    FailureVerify,
}

impl ParserError {
```

```
            pub fn to_nom<T>(&self, input: T) -> nom::Err<nom::error::Error<T>> {
                match self {
                    Self::ErrorVerify => nom::Err::Error(nom::error::Error::new(
                        input,
                        nom::error::ErrorKind::Verify,
                    )),
                    Self::FailureVerify => nom::Err::Failure(nom::error::Error::new(
                        input,
                        nom::error::ErrorKind::Verify,
                    )),
                }
            }
        }

        #[derive(Clone, Debug)]
        pub enum ProcessResult {
            // Fully parsed
            Complete,

            // Some protocols parsed (we are going into the deep), but some in the deepness
are not
            Incomplete,

            // Not matched
            Failed,
        }

        #[cfg(test)]
        pub(crate) mod tests {
            use crate::dto::frame::FrameMetadata;
```

```
use crate::parser::ProcessResult;
use crate::protocols::ProtocolId;

pub enum FrameType {
    Metadata(FrameMetadata),
    Header(()),
    Raw(()),
}

pub struct ProtocolParser {
    raw_needed: bool,
    root: Option<ProtocolId>,
}

impl ProtocolParser {
    pub fn new(link_type: &pcap::Linktype, raw_needed: bool) -> Self {
        Self {
            raw_needed,
            root: ProtocolId::root(link_type),
        }
    }

    pub fn process(&self, packet: pcap::Packet) -> Option<FrameType> {
        let mut metadata = FrameMetadata::from_header(packet.header);

        if let Some(root_protocol) = &self.root {
            let result = super::traversal(root_protocol, &packet, &mut metadata, 0);
            return match result {
                ProcessResult::Complete => Some(FrameType::Metadata(metadata)),
```

```
                ProcessResult::Incomplete => match self.raw_needed {
                    true => Some(FrameType::Raw(())),
                    false => Some(FrameType::Metadata(metadata)),
                },
                ProcessResult::Failed => match self.raw_needed {
                    true => Some(FrameType::Raw(())),
                    false => Some(FrameType::Header(())),
                },
            };
        }

        None
    }
  }
}
```

ДОДАТОК Б

Апробація

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
ІМЕНІ ІГОРЯ СІКОРСЬКОГО»

СУЧАСНІ ПРОБЛЕМИ НАУКОВОГО ЗАБЕЗПЕЧЕННЯ ЕНЕРГЕТИКИ

Матеріали XXII Міжнародної
науково-практичної конференції
молодих вчених і студентів
м. Київ, 22–25 квітня 2025 року

ТОМ 2

НТУУ "КПІ" 1931 ННІАТЕ

Київ- 2025

УДК 004.94

[1] Бакалаврант 4 курсу Ковальов О.О.
[1] Асист. Кардашов О.В.
https://scholar.google.com.ua/citations?user=gtnZz4EAAAAJ&hl=uk

[1] КПІ ім. Ігоря Сікорського

ПОРІВНЯННЯ ТЕХНОЛОГІЙ ЗАХОПЛЕННЯ МЕРЕЖЕВОГО ТРАФІКУ ДЛЯ РОЗРОБКИ СИСТЕМИ ГЛИБОКОГО АНАЛІЗУ ПАКЕТІВ

Постановка проблеми та її актуальність. У сучасному світі інформаційних технологій зростає потреба у високопродуктивних та безпечних системах аналізу мережевого трафіку, особливо коли йдеться про застосування технологій deep packet inspection (DPI, глибокий аналіз або інспекція пакетів, де пакет є будь яким блоком даних відносно рівнів системи OSI) для забезпечення кібербезпеки, моніторингу мереж і оперативного виявлення аномалій. Використання DPI полягає не лише в обробці заголовків, а й самих даних, payload (корисне навантаження) та визначення протоколів.

Основною задачею є захоплення потоків пакетів із високою швидкістю та їх подальша обробка, що вимагає не лише високої продуктивності, але й надійності та відсутності вразливостей, пов'язаних з помилками управління пам'яттю. Особливо актуальною стає проблема безпеки доступу до пам'яті, оскільки традиційні реалізації на мовах, таких як C або C++, часто стикаються з ризиком переповнення буфера, витоків пам'яті та інших помилок, які можуть бути використані зловмисниками. Сучасні вимоги до програмного забезпечення також передбачають відсутність залежності від збору сміття (відсутністю Garbage Collector) з метою мінімізації накладних витрат у реальному часі, що є особливо важливим при обробці великого обсягу даних у мережевих застосунках. Згідно аналізу [1] Агентства з кібербезпеки та захисту інфраструктури США, рекомендується використовувати «безпечні» мови, такі як Rust, Java, C#, Go і так далі. Лише мова Rust з цього переліку не має Garbage Collector. Таким чином, при виборі технології треба орієнтуватись на два основних фактори: швидкодія та можливість використовувати мову Rust.

Аналіз останніх досліджень. За останнє десятиліття було розроблено кілька підходів до захоплення пакетів, серед яких найбільш відомими є використання pf_ring, eBPF та бібліотек для роботи з pcap, а саме реалізації на основі Rust (libpnet).

pf_ring – це технологія, яка використовує спеціалізований драйвер для забезпечення високошвидкісного захоплення пакетів, дозволяючи обробляти величезні об'єми мережевого трафіку. Її продуктивність підтверджується широким застосуванням у високонавантажених мережах [2], проте реалізація на базі традиційних мов програмування не гарантує повної безпеки доступу до пам'яті, що може створювати ризики при експлуатації системи.

eBPF, або Extended Berkeley Packet Filter, інтегрований безпосередньо в ядро сучасних Linux-систем, відкриває нові можливості для безпечного та гнучкого аналізу мережевого трафіку. Завдяки вбудованій віртуальній машині, яка перевіряє виконуваний байткод перед його запуском, eBPF забезпечує високу ступінь захищеності, що є критично важливим при розгортанні систем глибокого аналізу пакетів. Проте, незважаючи на безпечність у виконанні, eBPF може вимагати певного часу на адаптацію та навчання розробників, оскільки синтаксис та моделі виконання відрізняються від традиційних підходів. Завдяки даній технології можна ефективно обробляти фрейми – читати їх, відхиляти, тощо. [3]

Бібліотека pcap (libpcap [4]), зокрема її реалізація libpnet (написана на Rust), поєднує популярність класичного підходу з перевагами сучасної мови програмування Rust:

володіння пам'яттю, перевірка типів під час компіляції та відсутність залежності від збирача сміття. Це означає, що навіть у високонавантажених умовах система може ефективно працювати без ризику витоків пам'яті або інших помилок, характерних для реалізацій на C. Ці переваги особливо актуальні з огляду на сучасні вимоги до безпеки програмного забезпечення, де будь-яка вразливість може призвести до серйозних наслідків. Аналіз останніх досліджень демонструє, що інтеграція «memory-safe» бібліотек у системи DPI може не тільки підвищити надійність, але й забезпечити модульність та масштабованість рішень.

Формулювання мети. Метою даної роботи є порівняння технологій захоплення мережевого трафіку – pf_ring, eBPF та pcap із реалізацією через Rust libpnet – з акцентом на пошук оптимального балансу між високою продуктивністю та забезпеченням безпеки доступу до пам'яті. Основною задачею є порівняння підходів до перехоплення мережевих фреймів для подальшої обробки, у даному випадку – для розробки системи глибокого аналізу пакетів, що накладає певні обмеження, а саме здатність обробляти великі обсяги даних без компромісів у безпеці та стабільності роботи програмного забезпечення.

Основна частина. Розглядаючи pf_ring, відразу слід зазначити, що ця технологія реалізована виключно мовою C та працює у ядерному режимі. Такий підхід дозволяє забезпечити максимальну швидкодію завдяки прямому доступу до мережевого інтерфейсу та мінімальним накладним витратам. Однак, використання мови програмування C, як було зазначено вище, створює безпекові ризики, які можуть бути неприпустимими в певних проектах.

pf_ring опитує фрейми від мережевих адаптерів за допомогою Linux NAPI (механізм обробки подій). NAPI копіює пакети з мережевого адаптера в циклічний буфер, а потім програма користувача зчитує пакети з кільця. У цьому сценарії є два клієнти які займаються полінгом (опитуванням), це програма розробника та NAPI (Рис. 2.)[5]. Перевага полягає в тому, що дана технологія може розподіляти вхідні пакети на кілька кілець (отже, на кілька програм) одночасно. Таким чином, досягається висока швидкодія – дану технологію можна використовувати в промислових масштабах, якщо немає серйозних вимог до кібербезпеки, наприклад, в локальних мережах. Хоча, загалом, технологія перевірена часом, але все ж, з огляду до попередніх вимог вона не може вважатись безпечною в плані доступу до пам'яті. Реалізація на Rust відсутня.

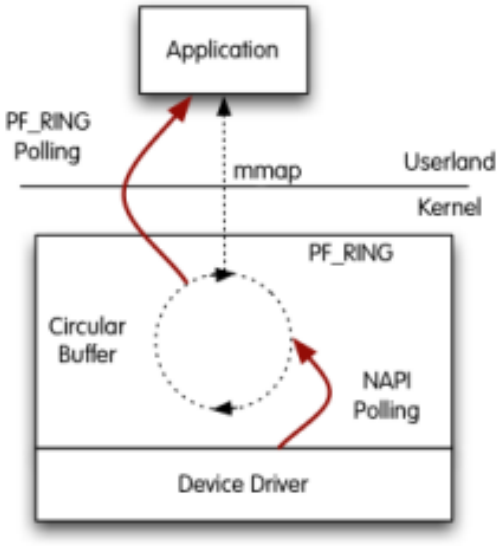

Рисунок 1 – Загальна архітектура технології PF_RING

В eBPF, який також функціонує в ядерному режимі, надає розробникам можливість завантажувати програми безпосередньо в ядро Linux. Завдяки верифікації байткоду перед виконанням, eBPF забезпечує високий рівень безпеки при обробці мережевого трафіку. Він працює у привілейованому контексті та використовує набір допоміжних функцій для ефективної обробки мережевих подій і трасування системних процесів у реальному часі. Пісочниця використовується для безпечного й ефективного розширення можливостей ядра без необхідності змінювати вихідний код ядра чи завантажувати модулі ядра [6]. Бібліотека aya-rs дозволяє використовувати eBPF у Rust, проте значна частина коду позначена як unsafe (небезпечний код, не має гарантій від компілятору), що створює потенційні ризики роботи з пам'яттю, хоч і забезпечує високу швидкодію. eBPF можна вважати компромісом між продуктивністю та гарантіями безпеки, однак технологія працює лише на Linux.

У свою чергу, бібліотека pcap з реалізацією на Rust (libpnet) працює в просторі користувача. На відміну від інших вищевказаних технологій, робота з фреймами в libpnet не може вестись «на льоту» – тому, умовний брандмауер написати з нею неможливо. Робота ведеться зі скопійованими даними, тобто, їх можна лише читати. Це, звичайно ж, впливає на швидкодію – до схожих механізмів роботи як у інших технологій, додається input/output затримка. Також, можна формувати власні фрейми, пакети, сегменти тощо, і відправляти їх, але це не стосується поставленої задачі. Є значна перевага – бібліотека може використовуватись на різних платформах завдяки чітко визначеному API. Наприклад, можна використовувати WinPCap або npcap для роботи на Windows.

Висновки. Порівняння цих технологій дозволяє зробити висновок, що вибір платформи для захоплення мережевого трафіку має враховувати компроміс між продуктивністю та гарантіями безпеки. У сучасних умовах, коли безпека доступу до пам'яті є критичною вимогою для запобігання експлуатації вразливостей, рішення на базі libpnet можуть бути привабливими для розробки систем глибокої інспекції пакетів, незважаючи на деяке зниження швидкодії. Хоча, якщо планується використовувати систему в умовах високого навантаження, варто розглянути інші варіанти, або компромісний – eBPF, або загалом безпечний для використання pf_ring, але не рекомендований до використання в системах, де є жорсткі вимоги до безпеки даних.

Перелік посилань:

1. Cybersecurity and Infrastructure Security Agency. Exploring memory safety in critical open source projects. www.cisa.gov. URL: https://www.cisa.gov/sites/default/files/2024-06/joint-guidance-exploring-memory-safety-in-critical-open-source-projects-508c.pdf (date of access: 27.02.2025).

2. The Improvement of Network Performance by Using the Technique of PF-RING Zero-copy to Optimize Spark Streaming / F. LIU et al. International Conference on Computer Networks and Communication Technology (CNCT 2016), Xiamen, China, 16–18 December 2016. Paris, France, 2017. URL: https://doi.org/10.2991/cnct-16.2017.9 (date of access: 25.02.2025).

3. Zhuravchak D., Kiiko E., Dudykevych V. Using EBPF to identify ransomware that use DGA DNS queries. Collection "Information Technology and Security". 2023. Vol. 11, no. 2. P. 166–174. URL: https://doi.org/10.20535/2411-1031.2023.11.2.293760 (date of access: 02.03.2025).

4. Snort 2.0 intrusion detection / J. Faircloth et al. Syngress, 2003. 550 p.

5. ntop. PF_RING - high-speed packet capture, filtering and analysis. https://www.ntop.org. URL: https://www.ntop.org/products/packet-capture/pf_ring/ (date of access: 19.02.2025).

6. eBPF Foundation. EBPF documentation. URL: https://ebpf.io/what-is-ebpf/ (date of access: 22.02.2025).