

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Національний університет «Запорізька політехніка»

Факультет комп'ютерних наук і технологій

(повне найменування факультету)

Кафедра програмних засобів

(повне найменування кафедри)

Пояснювальна записка

до дипломного проекту (роботи)

магістр

(ступінь вищої освіти)

на тему ДОСЛІДЖЕННЯ ТА ПРОГРАМНА РЕАЛІЗАЦІЯ

МОНІТОРИНГУ В LINUX НА ОСНОВІ eBPF

У РЕЖИМІ РЕАЛЬНОГО ЧАСУ

RESEARCH AND SOFTWARE IMPLEMENTATION OF

REAL-TIME MONITORING IN LINUX

USING eBPF

Виконав(ла): студент(ка) 2 курсу, групи
КНТ-124м

Спеціальності 121 Інженерія програмного
забезпечення

(код і найменування спеціальності)

Освітня програма (спеціалізація)

Інженерія програмного забезпечення

ГАНСЬКИЙ Д.А.

(ПРИЗВИЩЕ та ініціали)

Керівник СТЕПАНЕНКО О.О.

(ПРИЗВИЩЕ та ініціали)

Рецензент ДЬЯЧУК Т.С.

(ПРИЗВИЩЕ та ініціали)

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Національний університет «Запорізька політехніка»

Факультет КНТ
Кафедра програмних засобів
Ступінь вищої освіти магістр
Спеціальність 121 Інженерія програмного забезпечення
(код і найменування)
Освітня програма (спеціалізація) Інженерія програмного забезпечення
(назва освітньої програми (спеціалізації))

ЗАТВЕРДЖУЮ

Завідувач кафедри ПЗ, д.т.н, проф.
Сергій СУББОТІН
“ ” 2025 року

З А В Д А Н Н Я
НА ДИПЛОМНИЙ ПРОЄКТ (РОБОТУ) СТУДЕНТА(КИ)

ГАНСЬКОГО Данила Анатолійовича

(ПРІЗВИЩЕ, ім'я, по батькові)

1. Тема проєкту (роботи) Дослідження на програмна реалізація моніторингу в Linux на основі eBPF у режимі реального часу.

Research and Software Implementation of Real-time Monitoring in Linux Using eBPF

керівник проєкту (роботи) к.т.н., доцент, СТЕПАНЕНКО Олександр Олексійович,
(науковий ступінь, вчене звання, ПРІЗВИЩЕ, ім'я, по батькові)

затверджені наказом закладу вищої освіти від “ 30 ” вересня 2025 року № 447

2. Строк подання студентом проєкту (роботи) 08 грудня 2025 року

3. Вихідні дані до проєкту (роботи) рекомендована література, технічне завдання

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити) 1. Аналіз предметної області. 2. Обґрунтування вибору програмних засобів розробки. 3. Основні рішення щодо реалізації компонентів системи. 4. Експлуатація, тестування та експериментальне дослідження програми.

5. Перелік графічного матеріалу (з точним зазначенням обов'язкових креслень) Слайди презентації

6. Консультанти розділів проєкту (роботи)

Розділ	ПРИЗВИЩЕ, ініціали та посада консультанта	Підпис, дата	
		завдання видав	прийняв виконане завдання
1-4 Основна частина	СТЕПАНЕНКО О. О., доцент		
Нормоконтроль	ДЕЙНЕГА Л. Ю., старший викладач кафедри програмних засобів		

7. Дата видачі завдання «30» вересня 2025 року.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів дипломного проєкту (роботи)	Строк виконання етапів проєкту (роботи)	Примітка
1	Постановка завдання роботи.	1 тиждень	Завдання, ТЗ
2	Аналіз предметної області.	1 тиждень	Розділ 1
3	Вибір мови програмування та інших технологій розробки.	2 тиждень	Розділ 2
4	Розробка архітектури програми.	3 тиждень	Розділ 3
5	Розробка програми.	4-7 тижні	Розділ 3, 4
6	Тестування та експериментальне дослідження програмного забезпечення.	8 тиждень	Розділ 4
7	Оформлення пояснювальної записки та документів до неї.	9-10 тиждень	Додатки
8	Нормоконтроль та рецензування.	11 тиждень	
9	Захист роботи.	12 тиждень	

Студент

_____ Данило ГАНСЬКИЙ
(підпис) (Ім'я ПРИЗВИЩЕ)

Керівник проєкту (роботи)

_____ Олександр СТЕПАНЕНКО
(підпис) (Ім'я ПРИЗВИЩЕ)

РЕФЕРАТ

Пояснювальна записка до дипломної кваліфікаційної роботи магістра:
109 с., 2 табл., 57 рис., 2 дод., 41 джерело.

LINUX, EBPf, C, GO, МОНІТОРИНГ У РЕАЛЬНОМУ ЧАСІ, СИСТЕМНЕ АДМІНІСТРУВАННЯ, ПРОДУКТИВНІСТЬ, БЕЗПЕКА, BCC, BPFTRACE СИСТЕМНІ ВИКЛИКИ (SYSCALLS), МЕРЕЖЕВИЙ МОНІТОРИНГ, ХМАРНІ ОБЧИСЛЕННЯ.

Об'єкт дослідження – процес моніторингу роботи, розроблення та тестування програмного забезпечення операційних системах, побудованих на ядрі Linux.

Предмет дослідження – методи і засоби збору моніторингових даних про роботу систем на ядрі Linux та додатків у режимі реального часу.

Мета роботи – створити програмне рішення, яке забезпечує збір моніторингових даних в ОС Linux з використанням технології eBPF, для збільшення їх обсягу та спрощення процесу моніторингу в контейнеризованих середовищах порівняно зі стандартними підходами.

Матеріали, методи та технічні засоби: документація eBPF, приклади застосування з відкритих джерел, тестові сценарії навантаження, операційна система Linux (Ubuntu/Debian/інші), компілятор clang/LLVM, бібліотеки libbpf, мови програмування C та Go, інструменти bcc/bpftool, апаратні ресурси ПК/серверу.

Результати: виконано дослідження технології та механізмів eBPF для моніторингу у реальному часі та розроблено програмне рішення для збору та відображення даних про системні події та ресурси.

Наукова новизна дипломної роботи полягає у розробці програмного рішення для моніторингу систем на основі Linux з використанням технології eBPF, що дозволяє збирати детальні метрики, журнали та трасування як з ядра Linux, так і з користувацьких додатків, забезпечуючи глибший та точніший

аналіз стану системи порівняно з традиційними методами моніторингу. Використання eBPF розширює можливості збору даних, дозволяючи отримувати інформацію, яка раніше була недоступною, що відкриває нові можливості оптимізації роботи систем на основі Linux, а використання OpenTelemetry дозволяє експортувати зібрані дані в уніфікованому форматі, що робить додаток незалежним від системи, де обробляються моніторингові дані.

Висновки: досліджено та розроблено програмне рішення для високопродуктивного збору моніторингових даних за допомогою eBPF з операційних систем на основі ядра Linux.

Галузь використання – інформаційна безпека, адміністрування та експлуатація серверів і хмарних середовищ.

ABSTRACT

Explanatory note to the diploma qualifying work of the master: 109 pages, 2 tables, 57 figures, 2 appendixes, 41 sources.

LINUX, EBPF, C, GO, REAL-TIME MONITORING, SYSTEM ADMINISTRATION, PRODUCTIVITY, SECURITY, BCC, BPFTRACE SYSTEM CALLS (SYSCALLS), NETWORK MONITORING, CLOUD COMPUTING.

The object of the study is the process of monitoring the operation, development and testing of software in operating systems built on the Linux kernel.

The subject of the study is methods and means of collecting monitoring data on the operation of systems on the Linux kernel and applications in real time.

The purpose of the work is to create a software solution that provides the collection of monitoring data in the Linux OS using eBPF technology, to increase their volume and simplify the monitoring process in containerized environments compared to standard approaches.

Materials, methods and technical means: eBPF documentation, open source application examples, load test scenarios, Linux operating system (Ubuntu/Debian/others), clang/LLVM compiler, libbpf libraries, C and Go programming languages, bcc/bpftrace tools, PC/server hardware resources.

Results: a study of eBPF technology and mechanisms for real-time monitoring was performed and a software solution was developed for collecting and displaying data on system events and resources.

The scientific novelty of the thesis lies in the development of a software solution for monitoring Linux-based systems using eBPF technology, which allows collecting detailed metrics, logs, and traces from both the Linux kernel and user applications, providing a deeper and more accurate analysis of the system state compared to traditional monitoring methods. The use of eBPF expands the capabilities of data collection, allowing you to obtain information that was previously unavailable, which opens up new opportunities for optimizing the operation of Linux-based systems, and the use of OpenTelemetry allows you to export the

collected data in a unified format, which makes the application independent of the system where the monitoring data is processed.

Conclusions: a software solution for high-performance collection of monitoring data using eBPF from operating systems based on the Linux kernel was researched and developed.

The field of use is information security, administration and operation of servers and cloud environments.

ЗМІСТ

	С.
Перелік скорочень та умовних позначок.....	10
Вступ.....	11
1 Аналіз предметної області.....	13
1.1 Моніторинг і спостережливість.....	13
1.2 Огляд ключових задач моніторингу.....	14
1.3 Огляд ключових сфер моніторингу.....	15
1.4 Огляд існуючих програмних моніторингових рішень.....	23
1.5 Висновки за розділом 1.....	27
2 Обґрунтування вибору програмних засобів розробки.....	28
2.1 Огляд підходів та технологій моніторингу операційних систем.....	28
2.2 Обґрунтування вибору мов програмування.....	31
2.3 Обґрунтування вибору OpenTelemetry (OTel) як інструменту для експорту даних моніторингу.....	35
2.4 Обґрунтування вибору середовища розробки.....	36
2.5 Загальний опис архітектури розробленої програмної системи.....	38
2.6 Висновки за розділом 2.....	40
3 Основні рішення щодо реалізації компонентів системи.....	41
3.1 Опис механізму пошуку процесів для збору моніторингових даних.....	41
3.2 Опис механізму роботи eBPF та його можливостей для збору даних	42
3.3 Компіляція та завантаження eBPF програм у ядро Linux.....	50
3.4 Читання даних з eBPF програм на стороні Go (user-space).....	60
3.5 eBPF та контейнерезовані середовища.....	62
3.6 Опис OpenTelemetry для експорту моніторингових даних.....	64
3.7 Висновки за розділом 3.....	68
4 Експлуатація, тестування та експериментальне дослідження програми.....	69

4.1 Опис застосування.....	69
4.2 Умови виконання програми.....	69
4.3 Інструкція по експлуатації програми.....	70
4.4 Тестування програмного забезпечення.....	79
4.5 Висновки за розділом 4.....	80
Висновки.....	81
Перелік джерел посилання.....	83
Додаток А Текст програми.....	87
Додаток Б Слайди презентації.....	102

ПЕРЕЛІК СКОРОЧЕНЬ ТА УМОВНИХ ПОЗНАК

AI	– Artificial Intelligence;
API	– Application Programming Interface;
BCC	– BPF Compiler Collection;
BPF	– Berkeley Packet Filter;
CPU	– Central Processing Unit;
CO-RE	– Compile Once – Run Everywhere;
DPDK	– Data Plane Development Kit;
DNS	– Domain Name System;
eBPF	– extended Berkeley Packet Filter;
GC	– garbage collector;
HTTP	– HyperText Transfer Protocol;
HTTPS	– HyperText Transfer Protocol Secure;
I/O	– Input/Output;
IDE	– Integrated Development Environment;
IP	– Internet Protocol;
LLVM	– Low Level Virtual Machine;
ML	– Machine Learning;
OOM	– Out Of Memory
OS	– Operating System;
RAM	– Random Access Memory;
SSH	– Secure Shell;
SYSCALL	– System Call;
TC	– Traffic Control;
TCP	– Transmission Control Protocol;
UDP	– User Datagram Protocol;
VM	– Virtual Machine;
XDP	– eXpress Data Path;
ПЗ	– програмне забезпечення.

ВСТУП

За даними журналу SQ Magazine, частка операційних систем, заснованих на ядрі Linux, які використовуються для хмарних обчислень, становила 49,2% у другому кварталі 2025 року. Також, за даними SQ Magazine, Linux-системи використовуються на 78,3% вебсерверів. Крім того, у 2023-2025 роках почала зростати потреба в серверах для запуску AI та ML. Linux-операційні системи і тут залишаються лідерами – 87,7% таких систем працюють на Linux [1].

Таким чином, можна зробити висновок, що більшість серверної інфраструктури ґрунтується на Linux. У зв'язку з цим питання забезпечення безпеки та організації моніторингу є одним із ключових у сучасних хмарних обчислювальних технологіях.

Наразі у сфері хмарних обчислень існує безліч рішень для захисту та моніторингу серверної інфраструктури, проте не кожне з них спроможне забезпечити повний захист інфраструктури та програмного забезпечення, що на ній функціонує. Саме тут на допомогу приходить технологія eBPF, яка є частиною ядра Linux і дозволяє здійснювати моніторинг операційної системи на будь-якому рівні: від мережевого стека, взаємодії з файловою системою чи запуску процесів – до контролю запитів на отримання певних прав. При цьому eBPF характеризується високою продуктивністю порівняно з альтернативними технологіями та підходами.

Виходячи з цього, дослідження та розробка програмного рішення, яке дозволить проводити ефективно моніторинг і збір даних з операційної системи з подальшим аналізом на предмет потенційних атак чи вразливостей, є актуальними.

Отже, основною метою цієї роботи є комплексне дослідження eBPF та інших технологій моніторингу операційних систем на основі Linux, їх порівняння для вирішення різних типів завдань, класифікація сфер моніторингу та розробка програмного рішення, орієнтованого на вирішення визначених проблем.

Досягнення поставленої мети передбачає виконання таких завдань:

- визначити ключові задачі і області моніторингу в операційних системах, які потенційно можуть стати об'єктом атак зловмисників або призвести до порушення коректної роботи програмних систем;
- здійснити порівняльний аналіз технології eBPF та інших наявних засобів і підходів до моніторингу й захисту визначених областей операційних систем;
- розробити програмне рішення для подолання зазначених проблем моніторингу та безпеки за допомогою технології eBPF.

1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ

1.1 Моніторинг і спостережливість

Моніторинг та спостережливість (observability) є ключовими складовими сучасних підходів до забезпечення надійності та ефективності інформаційних систем. Вони дозволяють не лише контролювати стан інфраструктури, а й отримувати глибоке розуміння процесів, що відбуваються всередині програмного та апаратного середовища.

Моніторинг зазвичай визначається як процес збору, аналізу та відображення даних про стан системи у режимі реального часу. Його основна мета – виявлення відхилень від нормальної роботи, своєчасне інформування адміністраторів та ініціювання автоматичних механізмів реагування. Традиційно моніторинг базується на заздалегідь визначених метриках (наприклад, завантаженість процесора, використання пам'яті, стан мережевих інтерфейсів), що дозволяє швидко оцінити працездатність системи, проте не завжди дає відповіді на питання "чому" виникла певна проблема.

Спостережливість (observability) є ширшим поняттям, яке розвиває і доповнює класичний моніторинг. Вона базується на концепції, що система є "спостережливою", якщо за її зовнішніми сигналами (логами, метриками, трасуваннями) можна відтворити внутрішній стан та причини виникнення відхилень. Іншими словами, спостережливість надає не лише статистичні дані, а й інструменти для діагностики першопричин інцидентів.

У практиці DevOps та SRE (Site Reliability Engineering) обидва підходи застосовуються спільно. Моніторинг забезпечує базовий контроль і своєчасне реагування на збої, тоді як спостережливість дозволяє глибше дослідити складні сценарії, пов'язані з продуктивністю, масштабованістю чи безпекою системи. Для цього використовуються комплексні інструменти та платформи, які інтегрують:

- метрики (кількісні показники роботи системи);
- логи (журнали подій для аналізу історії виконання операцій);

- трасування (відстеження життєвого циклу запитів у розподіленому середовищі).

Таким чином, моніторинг і спостережливість формують взаємодоповнюючі механізми, що забезпечують цілісне уявлення про стан системи, сприяють швидкому виявленню проблем і допомагають у довгостроковому вдосконаленні архітектури та оптимізації використання ресурсів.

1.2 Огляд ключових задач моніторингу

Серед основних задач моніторингу можна виділити наступні:

- попередження про потенційні вразливості в системі;
- попередження про помилки в налаштуванні в компонентах системи;
- виявлення потенційних або вже існуючих “вузьких місць” в системі, також відомих як “bottlenecks”, які спричиняють затримки (latencies) при обробці користувацьких запитів або іншої активності в системі;
- визначення оптимальних налаштувань та обмежень по ресурсам для різних компонентів в системі.

Як можна бачити, кожна з задач яку вирішує моніторинг, є важливою або навіть критичною для будь-яких програмних систем. Маючи наскрізний моніторинг в програмній системі, дозволяє покращувати стабільність таких систем, їх швидкодію, пришвидшувати реагування та вирішення критичних інцидентів в продакшені, і запобігання таких інцидентів в майбутньому.

Щоб вирішити цей список задач, інструменти моніторингу мають мати широкий спектр доступів до різних компонентів системи, щоб збирати великі об’єми різноманітних подій на кожному з етапів системи, аналізувати їх, і повідомляти розробників цієї системи про них.

Відповідно, виділимо ключові сфери, на яких зосереджується моніторинг.

1.3 Огляд ключових сфер моніторингу

Серед ключових сфер моніторингу, здатних забезпечити виконання завдань, зазначених у попередньому розділі, можна виділити наступні:

- моніторинг продуктивності системи;
- моніторинг мережевої активності;
- моніторинг безпеки та доступу;
- моніторинг журналів та подій.

Розглянемо кожен сферу окремо, визначимо її значення та критичність, проаналізуємо роботу залучених компонентів, можливі вразливості та підходи до їх моніторингу в системах Linux.

1.3.1 Моніторинг продуктивності системи

Моніторинг продуктивності системи на основі ядра Linux визначається наступними чинниками:

- навантаження на процесорні ядра (CPU), кількість контекстних перемикачів та робота планувальника;
- використання пам'яті (heap, stack, кеші, swap);
- ввід/вивід (I/O) – навантаження на дискові пристрої та файлові системи.

Коли система характеризується перевантаженням, насамперед аналізу підлягає навантаження на процесор. Навантаження на процесор визначається рівнем використання процесорних ресурсів його ядер. Воно відображає, скільки часу ядро витрачає на виконання процесів користувача, системних процесів, обробку переривань та очікування операцій вводу/виводу. Аналіз навантаження дозволяє оцінити ефективність розподілу процесів між ядрами та виявити можливі “вузькі місця”, коли окремі ядра перевантажені, а інші залишаються мало задіяними.

Основні характеристики навантаження на процесор:

- User CPU – час процесора, витрачений на виконання користувацьких процесів (не системних), включно з усіма програмами користувача, скриптами та демонами;
- System CPU – час, витрачений на виконання системних викликів ядра (kernel space), таких як обробка файлових операцій, мережових запитів та робота планувальника задач;
- Nice CPU – час процесора, витрачений на процеси яким надано нижчий або вищий пріоритет;
- Idle CPU – час, коли CPU простоює і не виконує жодної роботи;
- I/O Wait – час, коли процесор чекає завершення операцій вводу/виводу (диск, мережа), або неактивний час CPU, але процеси заблоковані на I/O [2].

Не найменшу роль в навантаженні на процесор в Linux і розподіленні цього навантаження по ядрам є планувальник. У Linux планувальник визначає пріоритети завдань, обирає наступний процес для виконання та забезпечує баланс навантаження між ядрами. Аналіз його роботи важливий для розуміння, наскільки справедливо та ефективно відбувається розподіл ресурсів між користувацькими і системними процесами. При моніторингу функціонування планувальника ключове значення має визначення інтенсивності контекстних перемикань.

Контекстні перемикання в Linux – це процес збереження та відновлення стану виконання потоків чи процесів під час їхнього перемикання планувальником. Висока частота контекстних перемикань може свідчити про значну конкуренцію між процесами за процесорний час або про надмірне використання механізмів синхронізації. Моніторинг цього показника дозволяє виявити проблеми з продуктивністю, викликані частими переходами між задачами.

Наступним не менш важливим параметром моніторингу системи є оперативна пам'ять (RAM). Недостатній обсяг оперативної пам'яті може

спричиняти неочікувані ефекти, такі як блокування запуску нових процесів, збільшення навантаження на процесор, що зазвичай пов'язано з роботою збирача сміття (Garbage Collector, GC), який при великій кількості об'єктів у пам'яті повинен просканувати всі її ділянки для виявлення невикористаних об'єктів та їх очищення. У крайніх випадках дефіцит пам'яті може призводити до повної відмови системи (downtime), унеможливаючи користувачам взаємодію з програмним забезпеченням та значно ускладнюючи відновлення доступу до машини, оскільки у перевантажених системах навіть створення SSH-з'єднання для перезапуску ресурсоємних процесів часто є неможливим.

Також якщо для процесу встановлено обмеження на використання оперативної пам'яті і він досягає цього ліміту, у ядрі Linux активується механізм OOM Killer, який примусово зупиняє відповідні процеси відправкою сигналу SIGKILL [3]. Це може бути корисним для відновлення доступу до системи, що досягла ліміту по пам'яті, проте є небажаним у випадку зупинки критичних процесів. Можливість моніторити стан пам'яті процесів до активації OOM Killer є критичною для підтримки стабільності системи. Це дозволяє своєчасно виявляти та усувати витoki пам'яті (memory leaks), переконфігурувати обмеження на використання ресурсів процесами або здійснювати вертикальне масштабування серверів.

А отже, ефективний моніторинг використання оперативної пам'яті в системах Linux охоплює наступні області:

- heap (купа) – динамічно виділена область пам'яті, що використовується програмами для створення об'єктів довільного розміру;
- stack (стек) – пам'ять для зберігання локальних змінних, адрес повернення та керування викликами функцій, або об'єктів, що мають фіксованих розмір;
- кеші сторінок – використовуються ядром для прискорення доступу до файлів і зменшення кількості звернень до диска;
- swp (підкачка) – область на диску, що використовується для збереження вмісту оперативної пам'яті у випадку її нестачі [4].

Тримаючи ці параметри під контролем, можна уникнути небажаних ситуацій при роботі з пам'яттю, які були розглянуті вище.

Якщо система включає компоненти, що використовують інтенсивно дискові пристрої для збереження та обробки даних, як наприклад бази даних, брокери повідомлень (message brokers), або програми, що обробляють великі масиви даних, то моніторинг вводу/виводу (I/O) також набуває важливого значення. Моніторинг I/O охоплює операції доступу до дисків, файлових систем та інших пристроїв вводу/виводу. Ефективність цих операцій напряму впливає на продуктивність системи, адже доступ до дисків це зазвичай блокуюча операція, і довгий простій програми із-за повільного диску може призводити до каскадного зниження продуктивності або збою, якщо компоненти програмної системи є взаємозалежними.

Основні аспекти моніторингу I/O включають:

- пропускна здатність дисків – кількість даних, що можуть бути прочитані або записані за одиницю часу;
- затримки (latency) – середній час відповіді дискової підсистеми на запит;
- кількість операцій вводу/виводу (IOPS) – частота виконання операцій читання та запису;
- черги запитів – показують, скільки процесів очікує доступу до диска чи файлової системи;
- рівень використання файлових систем – обсяг зайнятого та вільного місця, фрагментація, а також кешування метаданих [5].

Таким чином, моніторинг використання процесора, оперативної пам'яті та навантаження на дискові пристрої є одним із ключових і первинних етапів налаштування системи моніторингу, оскільки ці показники впливають на працездатність не лише окремих компонентів, а й всієї системи, що функціонує на відповідній інфраструктурі.

1.3.2 Моніторинг мережевої активності

До активності пристроїв вводу/виводу можна віднести також роботу мережевої карти та обробку мережевого трафіку, що охоплює ширшу сферу як з точки зору потенційних загроз безпеці, так і проблем із продуктивністю, ніж моніторинг дискових пристроїв.

Виділимо наступні основні методи моніторингу мережевої активності:

- моніторинг мережевих інтерфейсів (трафік, затримки, пропускна здатність);
- моніторинг мережевих з'єднань (TCP/UDP сесії, пакети, помилки, “дроппи”);
- моніторинг мережевих атак або підозрілої активності (сканування портів, DoS/DDoS-атаки, підробка пакетів, DNS-запити до невідомих або підозрілих серверів);
- глибокий аналіз трафіку.

Моніторинг мережевих інтерфейсів у Linux передбачає відстеження стану інтерфейсів, обсягу переданих та отриманих даних, швидкості пропуску, обробки пакетів та випадків їх втрати. Це дозволяє оцінити ефективність роботи мережевих підсистем, виявляти аномалії у трафіку та забезпечувати стабільність передачі даних у системі [6].

Моніторинг мережевих з'єднань у Linux включає відстеження активних TCP/UDP сесій, відкритих портів та віддалених хостів. Це дозволяє оцінити навантаження на мережеву підсистему, виявити незвичні або небажані з'єднання та забезпечити контроль за ресурсами системи. Регулярний аналіз мережевих з'єднань допомагає своєчасно реагувати на потенційні загрози та оптимізувати продуктивність передачі даних.

Моніторинг мережевих атак та підозрілої активності у Linux передбачає виявлення аномальних патернів трафіку, таких як сканування портів, DoS/DDoS-атаки, підробка пакетів або DNS-запити до невідомих серверів. Це забезпечує підвищений рівень безпеки системи, дозволяє виявляти потенційні

загрози на ранніх стадіях і реагувати на них до того, як вони призведуть до порушення стабільності роботи або втрати даних.

Глибокий аналіз трафіку є складним напрямом моніторингу, оскільки передбачає дослідження різноманітних мережевих протоколів на прикладному рівні, таких як HTTP/1.1, HTTP/2, DNS, FTP та інші, на транспортному рівні — TCP і UDP, а також на мережевому рівні — IPv4 та IPv6. Такий підхід може розглядатися як наступний крок для підвищення ефективності моніторингу, забезпечуючи більш просунутий контроль за переданими даними. Глибокий аналіз трафіку дозволяє визначати, як клієнти використовують публічні API, виявляти підозрілу активність, а також ідентифікувати випадки розкриття конфіденційної інформації без належної аутентифікації користувачів [7].

Для ефективного аналізу змісту трафіку необхідно не лише збирати та обробляти кожен пакет, переданий мережею, а й формувати на їх основі окремі запити та відповіді, наприклад, у контексті HTTP-протоколів, де кожен запит і відповідь має власну структуру та механізми комунікації.

Отже, застосування всіх зазначених підходів на практиці забезпечує своєчасне виявлення аномалій, можливість оптимізувати використання мережевих ресурсів, запобігати потенційним атакам на систему через мережу та забезпечувати стабільну і захищену роботу інформаційної інфраструктури.

1.3.3 Моніторинг безпеки та доступу

Забезпечення безпеки інформаційних систем вимагає постійного контролю за діями користувачів та процесів на рівні операційної системи. У середовищі Linux важливим аспектом є моніторинг системних викликів (syscalls), які можуть свідчити про аномальну або підозрілу активність. Їх моніторинг дозволяє відслідковувати:

- відкриття чи зміну критичних файлів (наприклад, /etc/passwd, /etc/shadow);
- створення або завершення процесів;

- виконання операцій із мережею (підозрілі з'єднання, сканування портів).

Також важливим аспектом є моніторинг дій з файлами, що допомагає виявляти спроби несанкціонованого доступу до конфіденційних даних. Наприклад:

- читання захищених системних директорій;
- модифікація двійкових файлів, що можуть призвести до встановлення бекдорів;
- звернення до файлів журналів з метою їх очищення або підробки.

Ще одним із ключових напрямів безпекового моніторингу є виявлення дій, спрямованих на отримання прав адміністратора. До таких подій належать:

- виконання команд через `setuid` або запуск процесів із підвищеними правами;
- використання експлойтів, що викликають аномальні системні виклики;
- підозрілі виклики `execve`, які запускають оболонку з `root`-доступом [8].

1.3.4 Моніторинг журналів та подій

Журнали та події є ключовим джерелом інформації для виявлення інцидентів, аналізу проблем та забезпечення стабільності системи. У середовищі Linux велике значення має збір і кореляція логів на різних рівнях – від ядра до прикладних програм.

Kernel logs (ядро Linux) – містять інформацію про драйвери, апаратні збої, помилки ядра та події на низькому рівні. Їх можна переглядати за допомогою `dmesg` або через системний демон `journald` [9].

Audit logs – записи з підсистеми `auditd`, що фіксують доступ до файлів, системні виклики, зміну користувачів та інші критично важливі події безпеки. Вони дозволяють відстежувати аномалії та спроби порушення політик доступу.

Також кожен застосунок може вести власний журнал, у якому фіксуються помилки, попередження, стан сервісів та інші події. Ці логи зазвичай зберігаються у директорії `/var/log/` (наприклад, `/var/log/nginx/access.log`, `/var/log/mysql/error.log`) [10]. Їх аналіз дозволяє виявляти як технічні несправності, так і потенційні атаки (наприклад, масові спроби входу).

SELinux та AppArmor генерують журнали щодо блокування доступу до ресурсів або спроб обходу політик безпеки. Аналіз таких подій допомагає ідентифікувати шкідливу активність [11].

Типові приклади: спроба процесу отримати доступ до файлу чи мережевого порту, що заборонений політикою.

Також в Linux ведуться наступні журнали, які можна відслідковувати та аналізувати:

- входи/виходи користувачів у систему (`/var/log/auth.log`) [12];
- запуск і завершення служб (`systemd`, `cron`);
- спроби автентифікації та ескалації привілеїв.

Виявлення аномальної активності (наприклад, багаторазові невдалі логіни) є важливою складовою моніторингу безпеки.

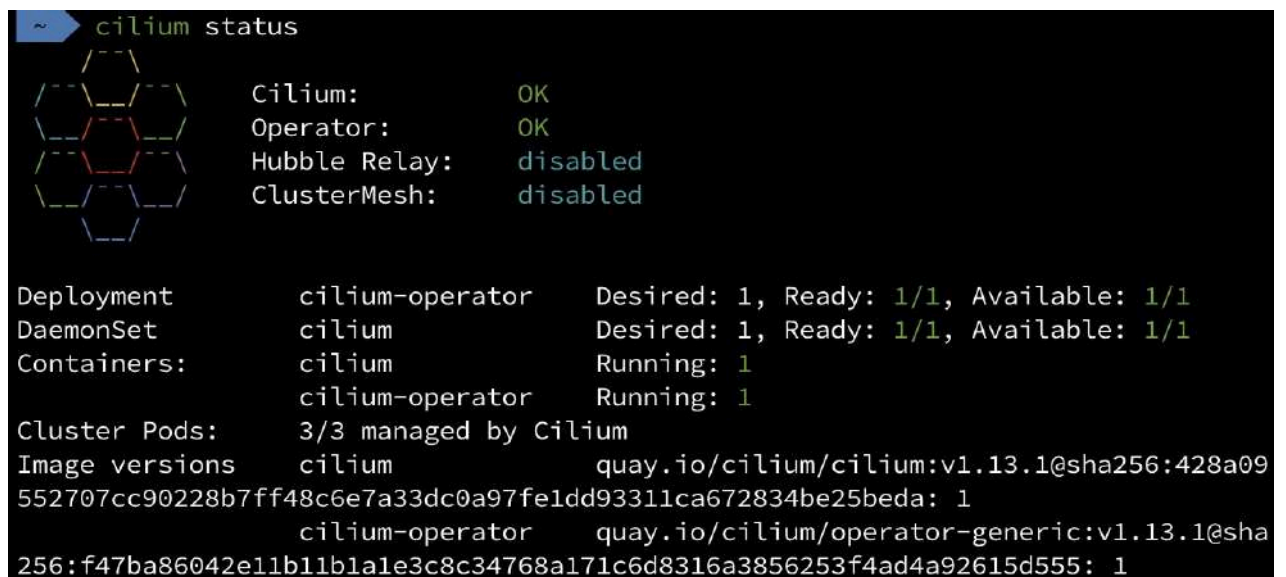
1.4 Огляд існуючих програмних моніторингових рішень

1.4.1 Cilium

Cilium – це платформа для мережевої безпеки та моніторингу, орієнтована на Kubernetes і хмарні середовища та розробляється компанією Isovalent. Вона базується на eBPF (Extended Berkeley Packet Filter) технології, що дозволяє здійснювати глибокий аналіз мережевого трафіку на рівні ядра Linux без значних накладних витрат. Cilium забезпечує:

- мікросегментацію мережевого трафіку;
- детальний моніторинг TCP/HTTP/gRPC-запитів між сервісами;
- інтеграцію з Prometheus та Grafana для збору та візуалізації метрик;
- підтримка мережевих політик Kubernetes і аналіз мережевих подій у реальному часі;
- консольну утиліту для управління системою Cilium (рис. 1.1).

Також команда яка працює на продуктах Cilium, розробила модулі cilium/ebpf та cilium/bpf2go які є основою розробки програмних рішень на основі технології eBPF та мові програмування Go (Golang).



```

cilium status

Cilium: OK
Operator: OK
Hubble Relay: disabled
ClusterMesh: disabled

Deployment      cilium-operator   Desired: 1, Ready: 1/1, Available: 1/1
DaemonSet      cilium            Desired: 1, Ready: 1/1, Available: 1/1
Containers:    cilium            Running: 1
               cilium-operator Running: 1
Cluster Pods:  3/3 managed by Cilium
Image versions  cilium            quay.io/cilium/cilium:v1.13.1@sha256:428a09552707cc90228b7ff48c6e7a33dc0a97fe1dd93311ca672834be25beda: 1
               cilium-operator quay.io/cilium/operator-generic:v1.13.1@sha256:f47ba86042e11b11b1a1e3c8c34768a171c6d8316a3856253f4ad4a92615d555: 1
  
```

Рисунок 1.1 – Сеанс командного рядка, що показує стан інсталяції Cilium у локальному кластері Kind [13]

1.4.2 Pixie

Pixie – це інструмент для автоматичного збору телеметрії та трасування мікросервісів, який розробляє компанія New Relic. Він інтегрується з Kubernetes і дозволяє отримувати дані без необхідності розгортання агентів на кожному вузлі (рис. 1.2). Основні характеристики Pixie включають:

- безагентне інтегрування в кластери Kubernetes;
- автоматичний збір логів, метрик та трасування додатків;
- підтримку високочастотного збирання даних із мінімальним впливом на продуктивність системи;
- інтеграцію з New Relic для централізованої обробки та аналізу даних.

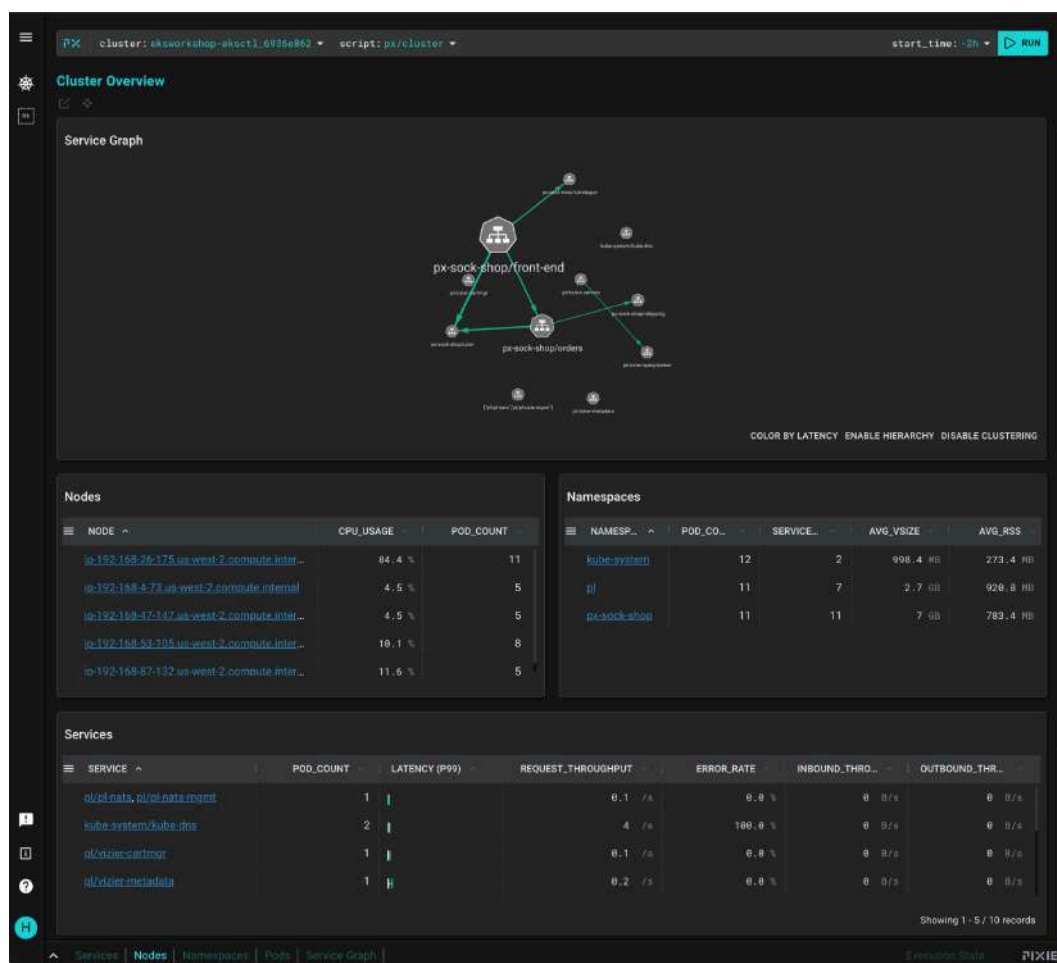


Рисунок 1.2 – Інформація про кластер, де встановлено Pixie [14]

Основні можливості Falco:

- моніторинг системних викликів у реальному часі;
- визначення потенційних загроз і небажаних дій через політики безпеки;
- інтеграція з Kubernetes для забезпечення контролю за контейнерами;
- підтримка Sysdig для глибшого аналізу подій у системі.

1.4.4 Parca / Parca Agent

Parca – це проект з відкритим програмним кодом для безперервного профілювання продуктивності додатків, який розробляє незалежна спільнота розробників. Parca дозволяє отримувати дані про використання CPU та пам'яті додатками у реальному часі (рис. 1.4), що допомагає оптимізувати ресурси в хмарних середовищах.

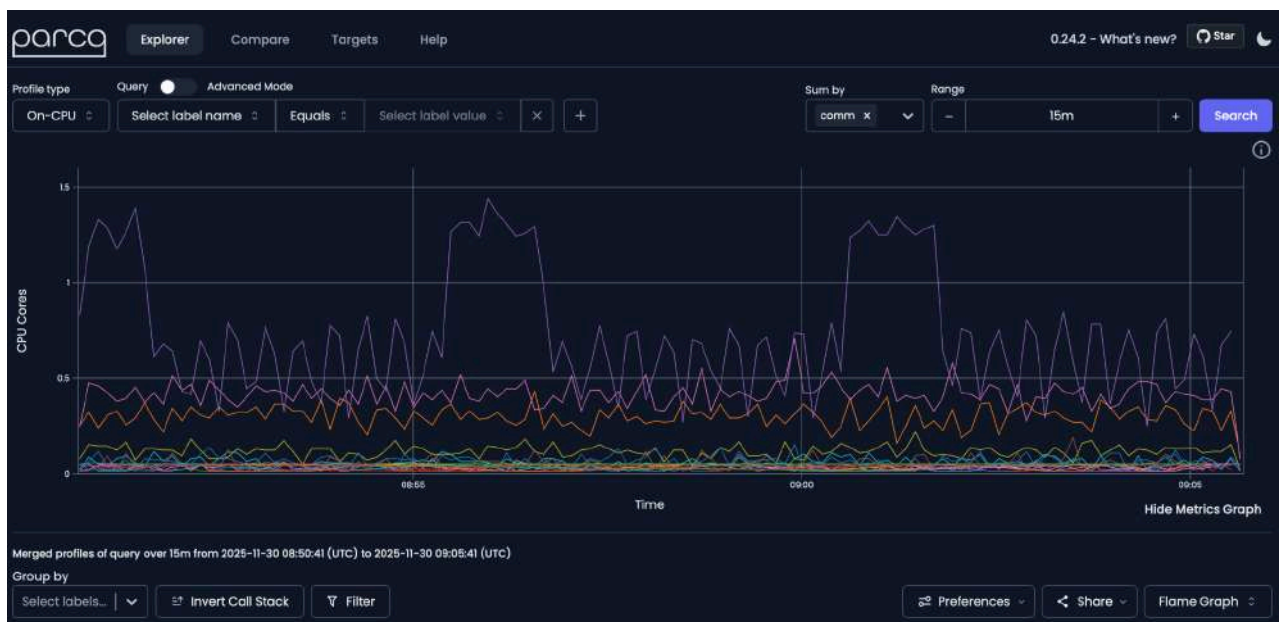


Рисунок 1.4 – Графік завантаження CPU в демо-кластері, дані для якого зібрані за допомогою Parca Agent [16]

Основні функції Parca включають:

- безперервне профілювання CPU та пам'яті;
- збирання і агрегування профілів на рівні контейнерів та вузлів;
- інтеграція з Prometheus та Grafana для візуалізації продуктивності;
- Parca Agent відповідає за локальний збір профілів і передачу їх на сервер Parca.

1.5 Висновки за розділом 1

Як показує аналіз технологій та програмних продуктів для збору даних, сфера моніторингу надзвичайно різноманітна та характеризується високою конкуренцією. Це свідчить про її ключову роль у світі хмарних і серверних обчислень. Водночас залишається простір для нових рішень, які зможуть охоплювати ширший спектр завдань та виступати повноцінною альтернативою кільком окремим, а подекуди й несумісним інструментам моніторингу.

Створене ПЗ має покривати якнаймога більший діапазон сфер де може бути задіяний моніторинг і де можна передчасно запобігти появам вразливостей для атак зловмисниками або несправностей самого ПЗ, що дозволить використовуючи лише один програмний продукт замість кількох спеціалізованих

2 ОБҐРУНТУВАННЯ ВИБОРУ ПРОГРАМНИХ ЗАСОБІВ РОЗРОБКИ

2.1 Огляд підходів та технологій моніторингу операційних систем

Моніторинг операційних систем є ключовим елементом забезпечення продуктивності, безпеки та стабільності сучасних серверних та мережесистем. Існує широкий спектр технологій і підходів, які дозволяють перехоплювати події ядра, мережевий трафік, системні виклики та активність користувацьких процесів. У даному розділі розглянуто основні інструменти та технології для моніторингу Linux-систем.

Далі наведено перелік основних технологій, які використовуються в сфері моніторингу систем на основі ядра Linux:

- eBPF (Extended Berkeley Packet Filter) – eBPF дозволяє виконувати користувацький код всередині ядра Linux у вигляді скомпільованого байт-коду без необхідності модифікації програмного коду ядра. Перед запуском такий код проходить перевірку (верифікацію), яка гарантує безпеку й не дозволяє програмам виконувати небезпечні операції (наприклад, прямий доступ до пам'яті ядра). Його використовують у таких інструментах як bcc, bpftrace, Falco, а також у комерційних рішеннях (наприклад, Cilium, Sysdig Secure);

- DPDK (Data Plane Development Kit) – працює у просторі користувача (user-space), але обходить стандартний мережевий стек ядра Linux, взаємодіючи напряду з мережею через спеціальні драйвери (poll mode drivers). Це дозволяє досягати надзвичайно високої пропускної здатності. Використовується для створення мережесистемних приладів, DPI (Deep Packet Inspection), IDS/IPS систем, балансувальників навантаження. Підходить у випадках, коли потрібно обробляти мільйони пакетів за секунду;

- perf – інструмент що є частиною ядра Linux і дозволяє профілювати продуктивність за допомогою апаратних лічильників (hardware performance counters) і програмних подій. Використовується для аналізу CPU-завантаження, виявлення "гарячих точок" (hot spots) у програмному коді, дослідження кеш-промахів, викликів функцій та часу виконання;

- `ftrace` – фреймворк, що дозволяє трасувати виклики функцій у ядрі Linux. Це низькорівневий інструмент для розробників і адміністраторів, які аналізують роботу ядра;
- `strace` – інструмент, що перехоплює та виводить всі системні виклики, які виконує процес, а також їхні аргументи і результати;
- `DTrace` – інструмент для динамічного трасування, що дозволяє відстежувати як ядро, так і `user-space` код у реальному часі. Має власну мову сценаріїв, але орієнтований більш на Solaris та BSD;
- `iptables/nftables` – інструменти для керування правилами Netfilter, що є частиною мережевого стеку Linux, що дозволяє фільтрувати пакети;
- `Auditd` – демони для аудиту системних подій та безпеки;
- Класичні `kprobes/uprobes` – механізми вставки точок моніторингу у функції ядра (`kprobes`) або користувацьких процесів (`uprobes`).

Нижче в табл. 2.1 наведено порівняльну характеристику основних технологій з точки зору застосування для моніторингу системних подій.

Таблиця 2.1 – Порівняльна таблиця основних технологій для моніторингу в системах на основі ядра Linux

Технологія	Переваги	Недоліки
eBPF	Висока продуктивність, безпека, інтеграція з ядром, віддалене трасування, один з найпопулярніших інструментів в сфері моніторингу	Вимагає сучасного ядра Linux (≥ 4.1)
DPDK	Максимальна продуктивність збору пакетів з мінімальними затримкам	Обмежений винятково мережевими задачами, вимагає спеціальної конфігурації та складніший у розгортанні

Продовження таблиці 2.1

Технологія	Переваги	Недоліки
perf	Можливості профілювання CPU та подій ядра	Непридатний для моніторингу безпеки чи мережевого трафіку, сфокусований лише на продуктивності
ftrace	Можливість детального трасування функцій в ядрі	Вимагає доступу до ядра, складний синтаксис, важко інтегрувати в користувацький код
strace	Простий та зручний інструмент для перехоплення системних викликів	Високе навантаження, не підходить для реального часу та великого обсягу подій
DTrace	Потужні можливості трасування	Погана підтримка Linux, складне розгортання
iptables/nftables	Фільтрація пакетів та логування мережі	Тільки мережеві події, не підходить для моніторингу системних викликів
Auditd	Аудит подій та безпека	Високе навантаження при великому обсязі подій, обмежена аналітика
kprobes/uprobes	Гнучке трасування функцій ядра та user-space	Складність написання, низький рівень безпеки у порівнянні з eBPF

Як видно з таблиці 2.1, eBPF поєднує унікальні властивості: високу продуктивність, безпечне виконання коду у просторі ядра, яка надається верифікатором коду eBPF програм, та можливість інтеграції з користувацькою частиною програми. Інші технології або обмежені у сфері застосування (мережевий трафік, аудит безпеки), або потребують складних налаштувань та додаткових знань. Тому у рамках цього проєкту вибір зупинено саме на eBPF, як на оптимальному інструменті для моніторингу операційної системи Linux у реальному часі з можливістю подальшої інтеграції з Go-кодом користувацької частини.

2.2 Обґрунтування вибору мов програмування

Оскільки програмний продукт складається з користувацької (user-space) та низькорівневої (kernel-space) частин, для їх реалізації застосовуються різні мови програмування. Зокрема, у user-space використовується Go для обробки даних, перехоплених у kernel-space, тоді як сама логіка в kernel-space реалізується мовою C, проте не в її класичному вигляді, а у модифікованому варіанті, адаптованому до вимог і обмежень розробки програм у просторі ядра.

2.2.1 Go (Golang) як мова програмування користувацької (user-space) частини застосунку

Для реалізації користувацької (user-space) частини програмного продукту було обрано мову програмування Go (Golang). Вибір цієї мови обумовлений низкою технічних переваг, які роблять її оптимальною для задач взаємодії з низькорівневими компонентами та обробки даних, що надходять від eBPF програм запущених у просторі ядра (kernel-space).

Go поєднує простоту синтаксису з високою продуктивністю, наближеною до мов низького рівня (наприклад, C). Це дозволяє створювати

ефективні користувацькі додатки з відносно невеликими витратами на розробку та підтримку.

Вбудовані механізми паралельного програмування (goroutines, канали) забезпечують зручну роботу з потоками даних у реальному часі. У межах даного проєкту це особливо важливо для обробки великого обсягу мережевого трафіку та системних подій, що передаються з kernel-space. Такі дані можуть накопичуватися у буфері, що дозволяє уникнути блокування читання з циклічного буферу (ring buffer), який слугує посередником між користувацькою частиною та компонентами, що виконуються у просторі ядра.

Також Go має розвинену стандартну бібліотеку та широкую екосистему відкритих пакетів, зокрема для роботи з мережею, протоколами, системними викликами та інструментами моніторингу. Це дає змогу ефективно інтегрувати користувацьку частину із зібраними eBPF даними для їх обробки. Додатковою перевагою застосування Go у даному проєкті є можливість використання бібліотеки cilium/ebpf. Це сучасний та активно підтримуваний модуль, який забезпечує високорівневий інтерфейс для взаємодії з eBPF програмами без необхідності прямого використання низькорівневих системних викликів. Завдяки цій бібліотеці можна завантажувати, прикріплювати та керувати eBPF програмами безпосередньо з Go-коду, а також зручно працювати з BPF-мапами та кільцевими буферами [17].

Крім того, Go характеризується кросплатформеністю та підтримкою статичної компіляції, що дозволяє створювати невеликі за розміром та самодостатні виконувані файли без складних залежностей. Така властивість значно спрощує розгортання застосунку у середовищах на основі Linux.

Для написання користувацької частини додатку можна використовувати й інші мови програмування, наприклад Python або Rust. Так як Rust стає дедалі популярнішим, збільшується й кількість відповідних бібліотек. Для eBPF у Rust вже існує подібний інструмент для завантаження eBPF програм, який перебуває в активній розробці. Python не має спеціалізованого фреймворку для роботи з eBPF, проте надає можливість завантажувати eBPF програми через BCC.

Далі в табл. 2.2 наведена порівняльна таблиця мов програмування для реалізації користувацької частини додатку.

Таблиця 2.2 – Порівняльна таблиця основних мов програмування для написання користувацької частини додатку для роботи з eBPF

Характеристика	Go (Golang)	Rust	Python
Продуктивність	Висока	Дуже висока	Середня
Безпека пам'яті	Середня	Висока	Середня
Простота мови	Висока	Середня	Дуже висока
Модулі для інтеграції з ядром	cilium/ebpf	aya-rs/aya	ВСС (зв'язка з C) [18]
Використання для eBPF проєктів	Є основною мовою для eBPF проєктів (Cilium, Falco, Pixie)	Поки немає високої популярності, але поступово зростає	Використовується в основному для прототипів
Можливості роботи з моніторинговими даними	Доступні модулі для роботи з OpenTelemetry	Доступні модулі для роботи з OpenTelemetry	Доступні модулі для роботи з OpenTelemetry

Як можна бачити з таблиці 2.2, використання Go для розробки користувацької (user-space) частини забезпечує оптимальний баланс між продуктивністю, простотою розробки та підтримкою сучасних підходів до обробки даних та розробки eBPF програм, що робить цю мову ефективним інструментом для побудови користувацької логіки у системах, які взаємодіють з eBPF програмами.

2.2.2 Мова програмування C (eBPF) для розробки низькорівневої (kernel-space) частини застосунку

Низькорівнева (kernel-space) частина програмного продукту реалізується з використанням модифікованої мови C у контексті eBPF (Extended Berkeley Packet Filter). Вибір C обумовлений тим, що більшість BPF-програм розробляються саме на цій мові, оскільки вона дозволяє безпосередньо взаємодіяти з ядром операційної системи Linux та забезпечує контроль над низькорівневими ресурсами системи.

Програми eBPF виконуються у захищеному середовищі ядра, що гарантує безпеку роботи та стабільність системи. Для розробки використовується обмежена в можливостях мова C, адаптована під вимоги ядра: наприклад, відсутність динамічного виділення пам'яті та обмежена робота з вказівниками. Це дозволяє уникнути потенційних аварійних ситуацій і забезпечує детерміновану поведінку програм у просторі ядра [19].

Використання C для розробки eBPF програм забезпечує високу продуктивність та низьку затримку обробки даних, що є критично важливим для моніторингу системних подій у реальному часі та обробки великого обсягу мережевого трафіку.

Також у сучасній розробці kernel-space програм розглядають мову Rust як альтернативу C. Rust забезпечує підвищену безпеку пам'яті та захист від помилок, пов'язаних з доступом до вказівників, що зменшує ймовірність аварійних збоїв. Проте на даний момент підтримка eBPF програм у Rust все ще менш розвинена, і велика частина прикладів та бібліотек у спільноті Linux створена для C. Крім того, використання Rust потребує додаткового навчання та адаптації інструментів, тоді як C має довгу історію застосування для розробки низькорівневих компонентів Linux, що забезпечує більшу сумісність та перевірену практику.

Таким чином, мова C у поєднанні з eBPF залишається оптимальним вибором для реалізації низькорівневої частини програмного продукту,

забезпечуючи баланс між продуктивністю, стабільністю та сумісністю з Linux ядром, хоча Rust поступово стає перспективною альтернативою для безпечнішого програмування kernel-space.

2.3 Обґрунтування вибору OpenTelemetry (OTel) як інструменту для експорту даних моніторингу

OpenTelemetry є сучасним і єдиним уніфікованим стандартом для збирання, обробки та експорту телеметричних даних, що охоплюють метрики, логи та трасування. Його ключова перевага полягає в уніфікованому підході до роботи з даними спостереження, що дозволяє уникнути залежності від конкретних інструментів або постачальників. Це робить OpenTelemetry універсальним рішенням для побудови систем моніторингу у масштабованих та гетерогенних інфраструктурах [20].

OpenTelemetry є відкритим стандартом, який підтримується спільнотою і є проектом з відкритим кодом під егідою CNCF (Cloud Native Computing Foundation), що забезпечує його активний розвиток, сумісність з великою кількістю інструментів та відсутністю прив'язки до постачальника (vendor lock-in).

Також OpenTelemetry характеризується як універсальний інструмент, що підтримує різні типи моніторингових даних (метрики, логи, трасування), дозволяючи формувати єдину екосистему спостережливості без необхідності інтеграції окремих специфічних агентів.

OpenTelemetry сумісний з популярними системами моніторингу та візуалізації, такими як Prometheus, Grafana, Jaeger, ELK чи Datadog (рис. 2.1). Це спрощує інтеграцію у вже існуючу інфраструктуру.



Рисунок 2.1 – Системи моніторингу, що підтримують стандарт OpenTelemetry [21]

Серед додаткових переваг варто відзначити гнучкість та розширюваність архітектури, яка дозволяє налаштовувати експортери, фільтри та процесори даних відповідно до вимог конкретної системи. Також OpenTelemetry відзначається масштабованістю, оскільки завдяки ефективному збору та оптимізованим форматам передачі даних воно підходить для використання у великих розподілених системах.

Підсумовуючи, використання OpenTelemetry як інструменту експорту даних моніторингу є обґрунтованим з точки зору стандартизації, гнучкості, сумісності та перспективи розвитку. Це рішення дозволяє забезпечити єдиний підхід до збору та передачі телеметричних даних, що є критично важливим для побудови ефективних систем спостережливості у сучасних програмних продуктах і можливості їх легкого інтегрування з іншими програмними продуктами для агрегування або візуалізації моніторингових даних.

2.4 Обґрунтування вибору середовища розробки

Для реалізації даного проєкту основним середовищем розробки було обрано Visual Studio Code (VS Code), що обумовлено його широкими можливостями для роботи з мовами програмування Go та C, які застосовуються для написання eBPF програм.

VS Code має офіційну підтримку Go завдяки розширенню Go extension for VS Code, яке надає засоби автодоповнення, статичного аналізу коду, інтеграцію з gopls (Go Language Server), запуск юніт-тестів та автоматичне форматування відповідно до стандартів мови. Це значно пришвидшує розробку та знижує кількість синтаксичних і логічних помилок [22].

Для роботи з eBPF програмами на мові C VS Code забезпечує інтеграцію з компіляторами (clang/llvm) та підтримку системи збірки (наприклад, make, CMake). Завдяки цьому стає можливим зручне редагування коду, його збірка, а також використання інструментів статичного аналізу для підвищення надійності розроблених BPF-програм.

Також у VS Code реалізована інтеграція з системою контролю версій Git та передбачені зручні засоби налагодження, що є важливими у процесі розробки як користувацької логіки на Go, так і низькорівневих eBPF програм.

Завдяки широкому набору плагінів (наприклад, для Docker, Remote-SSH чи Kubernetes) це середовище розробки дає змогу безпосередньо працювати з контейнеризованими або віддаленими системами, що є особливо актуальним для тестування та розгортання eBPF програм, оскільки їхня розробка має здійснюватися виключно у середовищі Linux та на відповідній архітектурі процесора.

Не менш важливим фактором є те, що у VS Code можна працювати з кількома мовами програмування одночасно в межах одного редактора, тоді як, наприклад, JetBrains GoLand орієнтований виключно на розробку мовою Go. Для роботи з C необхідно використовувати інше середовище, яке, до того ж, є платним.

Таким чином, Visual Studio Code було обрано як основне середовище розробки, оскільки воно забезпечує повноцінну підтримку Go та C, має інструменти для статичного аналізу, тестування та налагодження, а також дозволяє ефективно організувати робочий процес у проєкті, де поєднуються високорівнева логіка на Go та низькорівневе програмування з eBPF.

2.5 Загальний опис архітектури розробленої програмної системи

Архітектура запропонованої системи моніторингу побудована за модульним принципом, що дозволяє інтегрувати різні інструменти збору, зберігання та візуалізації даних. Основними її складовими є компоненти для експорту метрик, трасування та агрегації логів, що забезпечують комплексний огляд роботи операційної системи та прикладних процесів.

Програмна система реалізує інтерфейс OpenTelemetry (Otel), що дозволяє інтегрувати розроблений продукт з іншими популярними інструментами візуалізації та агрегації даних.

Збір даних із ядра Linux здійснюватиметься за допомогою eBPF програм, таких як kprobe, TC, XDP, а також uprobe для користувацьких (user-space) програм. Зібрані дані будуть оброблятися на стороні Go і експортуватися за допомогою OpenTelemetry (рис. 2.2).



Рисунок 2.2 – Технології й інструменти, що були використані при розробці програмної системи збору моніторингових даних

Структура та взаємодія основних компонентів програмної системи зображені на рис. 2.3.

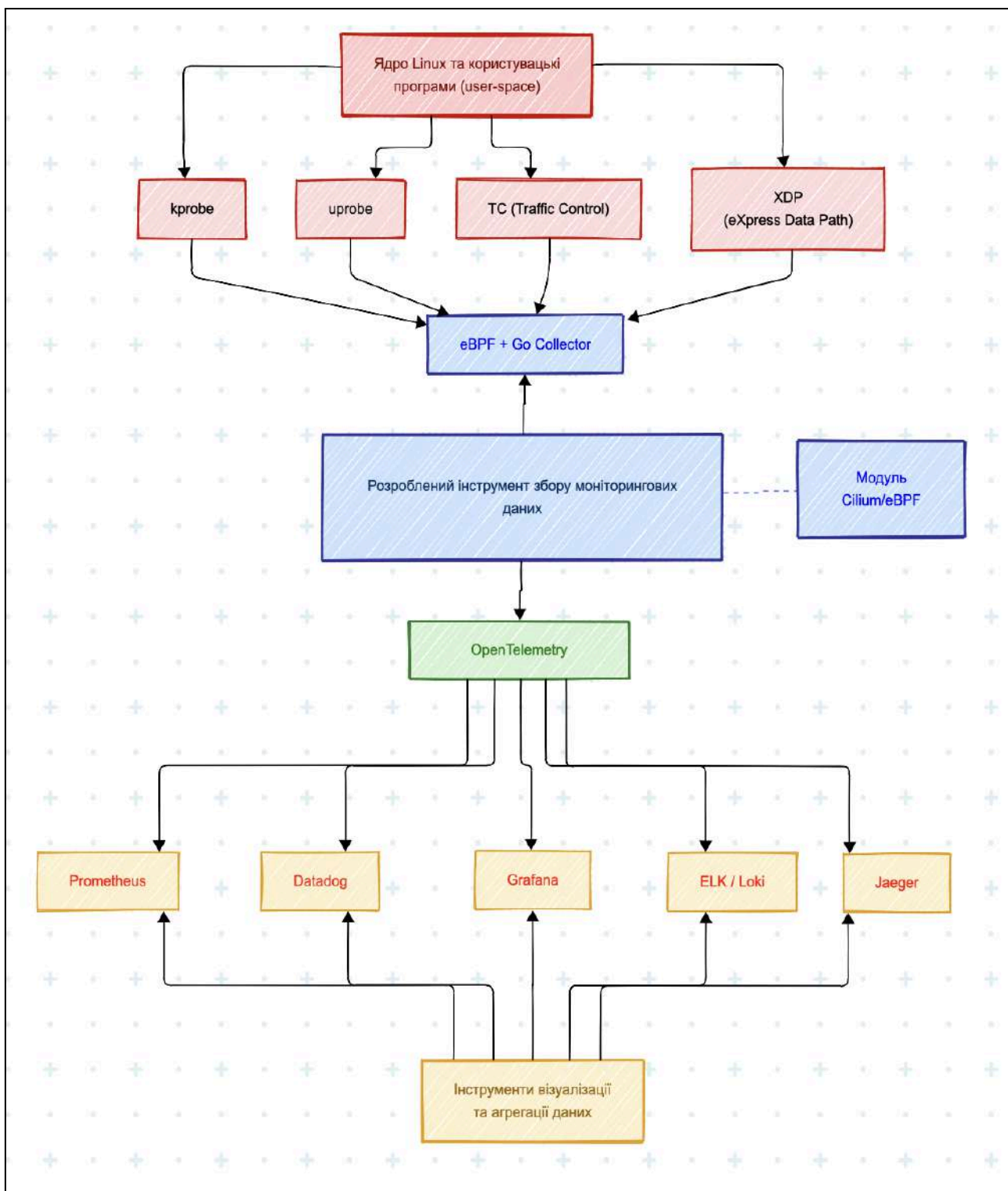


Рисунок 2.3 – Загальна діаграма компонентів програмної системи

2.6 Висновки за розділом 2

У другому розділі було розглянуто мови програмування та інструменти для реалізації користувацької (user-space) і низькорівневої (kernel-space) частин програмного продукту. Визначено вибір: eBPF – як надійний та безпечний інструмент для перехоплення даних на рівні ядра; Go (Golang) – для реалізації користувацької частини; C – для написання низькорівневого коду, що виконується в середовищі ядра Linux; OpenTelemetry (OTel) – для уніфікованого та стандартизованого експорту метрик, логів та даних трасування в інші системи візуалізації та агрегації.

Також наведено опис архітектури програмного продукту та перелік основних компонентів системи з їхнім описом і призначенням.

3 ОСНОВНІ РІШЕННЯ ЩОДО РЕАЛІЗАЦІЇ КОМПОНЕНТІВ СИСТЕМИ

3.1 Опис механізму пошуку процесів для збору моніторингових даних

При старті, перше, що має зробити колектор – це знайти програмні процеси, з яких потрібно буде збирати моніторингові дані. Щоб здійснити пошук по користувацьким процесам запущених на машині, можна просканувати всі директорії в /proc (рис. 3.1). /proc це віртуальна файлова система в Linux системах де зберігаються всі дані процесів. Так як процеси ідентифікуються в Linux системах по PID, що розшифровується як Process Identifier, всі директорії, як призначені для зберігання даних під окремі процеси називаються номером цього процесу.



```

$ ls /proc
1      1234649 1403410 1558894 1655259 1758740 2025705 248686 2881292 3193341 3441832 3898827 413739 506070 68 831133 982575
10     1235710 1403442 1559341 1657762 1758755 2026754 2486938 2885013 3193848 344919 3899868 4137570 506071 685340 831567 986366
1010445 1242116 1404332 1559841 1660274 1758766 2027628 2487371 2885513 3196211 3450979 39 4138059 506075 685869 831691 987263
1011812 1245654 1407699 1560374 1660714 1758796 2029159 2488004 2886126 3202599 3451436 3902324 4138556 506081 686735 832183 991861
1015893 1259654 1407910 1562751 1661183 1758804 2032907 2492654 2892989 3203689 3451473 3902649 4139477 506088 691563 835084 997587
1016345 1260112 1412621 1563352 1662457 1758904 2033527 2499756 29 3204685 3451660 3921420 4140548 506108 692077 835519 acpi
1017370 1260321 14286 1563843 1664272 1758907 2038204 25 291711 321276 3455251 3922371 4155507 506249 692625 835953 bootconfig
1024801 1261243 1428946 1564866 166547 1758969 2039695 2516847 2937615 3214078 3455284 3932271 4161871 508998 693271 837523 buddyinfo
1026262 1262619 1429413 1568395 1669483 1759333 2043868 251819 2941447 3214579 3459725 3932720 4181878 51 7 840868 bus
1027651 126320 1429556 1568825 1679986 1759399 2050035 253143 2941450 3215114 3460475 3933292 4191525 517509 701856 841350 cgroups
1040266 1263364 1430390 1569332 1680842 1759430 2052402 2534356 2942404 3216527 3461105 3933796 42 518781 713257 841415 cmdline
1041330 1263804 1433363 1569612 1684261 1765606 2052903 2544 2943565 3217022 3464580 3933827 42327 52 72108 84320 consoles
1041892 1264243 1435589 1569970 1684755 1766732 2055763 256469 2948912 3217644 347637 3934478 426258 521035 725217 843866 cpuinfo
1052699 1264684 1435825 1576270 1685723 1769758 2058054 257826 2953753 3218138 3486896 3936742 426752 522175 725814 84464 crypto
1053174 1265625 1436315 1587079 1686041 1770253 2061946 26 295904 3218645 3487138 3937426 42815 522610 726604 849285 devices
1053990 1270464 1436775 1588093 1686215 1770723 2067866 2601565 2959851 3218682 3495806 3937915 429397 53 727109 849909 diskstats
1055305 127359 1437341 1590153 1687212 1770812 2069389 2601566 2965601 3229808 3496379 3938584 429882 530239 727881 851916 dma
1070522 1274930 1437372 1590674 1687951 1771964 2070785 2603700 2975856 3229839 3497138 3939111 43 531066 728109 855684 driver
1071315 1275149 1439340 1590772 1688445 1772620 207658 2607075 2977476 3229860 35 393985 433221 532181 728559 856676 dynamic_debug
1091328 1275932 1439408 1591205 1689365 1772653 2077624 261367 2981041 3230778 3530498 3944054 43331 532918 729141 857298 execdomains
1093220 1281519 1441408 1591212 1690025 1775 2087473 2620007 2986793 3230802 3545687 3972198 44 54 729640 858283 fb
1098833 1281975 1449030 1591679 1691924 18 209285 2620528 2987818 3237936 3546218 3972713 440216 55 730202 874713 filesystems
1100496 1282193 1450102 1591699 1692463 1800359 2095548 2620529 3 3237970 3546815 3977622 442801 551 730674 875203 fs
1104127 1283001 1450546 1592121 1694180 1800390 209795 2620808 30 3237988 3561594 3978529 446751 556 731232 876072 interrupts
1104543 1283549 1450600 1592308 1694712 1813144 21 2623896 300333 3238909 3568872 3979702 447893 56 732254 876517 iomem
1105125 1284871 1450648 1592870 1695406 1814796 2100265 263104 301 3238933 3582906 3980189 448343 561364 736558 877387 ioports
1105146 1286211 1451716 1596729 1695909 1815292 210887 2639365 3022961 3245167 36 3980281 448636 561834 74048 881171 irq
1105996 1286651 1453775 1597336 1696527 1815763 2109843 2651803 3023395 3245189 3616219 3981120 449289 562353 743455 890809 kallsyms
1115604 1287090 1455824 16 17 1816324 2123679 2652625 3023474 3245237 3617297 3982165 45 564 747990 891283 kcore

```

Рисунок 3.1 – Приклад змісту директорії /proc

В кожній з цих директорій можна знайти дані процесів, а саме:

- cmdline – містить командний рядок, яким був запущений процес;
- comm – коротка назва процесу (ім'я виконуваного файлу);
- cwd – символічне посилання на поточну робочу директорію процесу;

- `environ` – список змінних середовища (ENV), з якими стартував процес;
- `exe` – символічне посилання на виконуваний файл, з якого стартував процес;
- `chroot` – посилання на кореневий каталог процесу (`chroot`, якщо застосовувалось);
- `status` – файл з основною інформацією про стан процесу: UID, GID, потоки, використання пам'яті, тощо;
- `stat` – великий рядок числових полів зі статистикою процесу: часи, пріоритети, PID/PPID, використання CPU;
- `statm` інформація про використання пам'яті: розмір, RSS, shared pages;
- `maps` – усі memory mappings: бібліотеки, сегменти стека, heap, анонімні області;
- `task` – каталог зі списком потоків процесу. Дані кожного потоку розділені по схожим директоріям названими по TID (TID розшифровується як Thread Identifier);
- `fd` – каталог з дескрипторами файлів, де перші три дескриптори відповідають за `stdin` (потік введення), `stdout` (потік виведення), `stderr` (потік виведення помилок) й інші відкриті файли і сокети (рис. 3.2).

arch_status	clear_refs	cpuset	fdinfo	latency	mem	ns	pagemap	sched	smaps_rollop	syscall	uid_map
attr	cmdline	cwd	gid_map	limits	mountinfo	numa_maps	patch_state	schedstat	stack	task	wchan
autogroup	comm	environ	io	loginuid	mounts	oom_adj	personality	sessionid	stat	timens_offsets	
auxv	coredump_filter	exe	kvm_merging_pages	map_files	mountstats	oom_score	projid_map	setgroups	statm	timers	
cgroup	cpu_resctrl_groups	fd	kvm_stat	maps	net	oom_score_adj	root	smaps	status	timerslack_ns	

Рисунок 3.2 – Приклад змісту директорії `/proc/<pid>`

Для фільтрування процесів по назві запущеної програми колектор використовує файл `comm`, але пізніше знадобляться й інші файли в `/proc`.

Звичайний користувач не може отримати доступ до всього змісту файлової системи /proc, для цього потрібно бути root користувачем, або встановити так звані capabilities.

Capabilities – це “розбиті на частини” root-права. Замість того щоб процесу давати повний root, можна дати тільки окремі можливості: читати чужу пам’ять, керувати мережевою конфігурацією, змінювати UID/GID, керувати файлами /proc тощо. У випадку для моніторингу файлової системи /proc, потрібно надати CAP_SYS_PTRACE.

Також, так як колектор можна буде запускати в контейнері, ми маємо розширити доступ контейнера до PID простору імен хоста, тобто, щоб процес колектору міг бачити усі процеси на машині, а не тільки в його контейнері. По замовчуванню, колектор побачить тільки процеси, що запущені в цьому ж контейнері, так як одна з основних задач контейнерів це ізоляція програм запущених в них від хоста.

В Docker розширити PID простір імен можна за допомогою команди `docker run --pid=host collector-image:latest`, або в Kubernetes встановивши `hostPID: true` для контейнера.

3.2 Опис механізму роботи eBPF та його можливостей для збору даних

Як вже було зазначено раніше, розроблюваний колектор моніторингових даних використовує технологію eBPF для перехоплення цих даних з різних точок в ядрі Linux або в користувацьких додатках.

Extended Berkeley Packet Filter (eBPF) – це сучасний механізм ядра Linux, який дозволяє виконувати безпечний, ізольований байткод у просторі ядра без необхідності створювати або модифікувати модулі ядра. Завдяки цьому eBPF забезпечує гнучкий, високопродуктивний та безпечний спосіб збору системних даних, спостереження, мережевого контролю та профілювання.

Основою механізму eBPF є програми eBPF, які компілюються у спеціальний байткод і завантажуються в ядро. Після завантаження програми прив'язуються до певних точок у системі (подій ядра або користувацьких програм) і виконуються при спрацьовуванні цих подій. Для кращого розуміння можна порівняти з принципом роботи точок зупинки для відладки коду.

Серед основних категорій eBPF програм налічуються наступні:

- kprobe/kretprobe – динамічні точки входу та виходу з функцій ядра;
- uprobe/uretprobe – аналогічні точки в користувацькому просторі (наприклад, у бібліотеках або бінарних файлах);
- trampoline – статичні точки трасування ядра з гарантованим стабільним ABI;
- fentry/fexit – високопродуктивні гачки, що виконуються при вході або виході з будь-якої функції ядра;
- tc/XDP – мережеві точки, використовувані для обробки пакетів на ранніх стадіях мережевого стека.

Розроблений колектор використовує програми типу uprobe та uretprobe для збору трасувальних даних користувацьких застосунків, що дозволяє перехоплювати виклики та завершення функцій у просторі користувача. Для збирання метрик та журналів системного рівня колектор застосовує eBPF програми типу trampoline.

3.2.1 ABI ядра та вимоги до сумісності

Для коректної роботи eBPF програми повинні бути сумісними з ABI (Application Binary Interface) ядра – бінарним інтерфейсом між ядром операційної системи та програмами, який визначає, як користувацькі застосунки взаємодіють з ядром на рівні машинного коду, а саме:

- структурами ядра;
- сигнатурами функцій;
- зміщеннями полів в структурах;

- доступними допоміжними функціями.

Для цього використовуються:

- `vmlinux` – бінарне представлення ядра з вбудованими BTF типами;
- `Linux headers` – заголовки ядра, необхідні під час компіляції eBPF

коду. BTF інформація (BPF Type Format) дозволяє забезпечувати сумісність програм із різними версіями ядра та мінімізує потребу в ручному описі структур.

3.2.2 eBPF maps – структури даних та передача даних до користувацької частини додатку

eBPF maps – це ключовий механізм обміну даними між eBPF програмою та користувацьким простором. Карти працюють як узагальнені структури даних, якими можна безпечно керувати через файл дескриптор.

Серед основних і загальноновживаних типів карт можна виділити наступні:

- `BPF_MAP_TYPE_HASH` – хеш-таблиця обмеженого розміру;
- `BPF_MAP_TYPE_ARRAY` – масив обмеженого розміру;
- `BPF_MAP_TYPE_PERCPU_ARRAY/BPF_MAP_TYPE_PERCPU_HASH` – карти, у яких кожне ядро має власну локальну копію значення, що дозволяє виконувати записи без блокувань і без ризику перезапису даних потоками, що працюють на інших ядрах;
- `BPF_MAP_TYPE_RINGBUF` – кільцевий буфер обмеженого розміру, що дозволяє читання на користувацькій частині додатку;
- `BPF_MAP_TYPE_LRU_HASH` – хеш-таблиця, яка працює з використанням алгоритму LRU;
- `BPF_MAP_TYPE_STACK/BPF_MAP_TYPE_QUEUE` – стек та черга фіксованого розміру.

За допомогою eBPF карт застосунки можуть накопичувати статистику, зберігати стан або відправляти дані до користувацького простору.

Особливої уваги заслуговує `ring buffer`, який забезпечує передачу подій з мінімальними накладними витратами та без блокувань, що робить його важливим механізмом для високопродуктивного збору моніторингових даних. Саме цей тип карти використовується в колекторі для передачі зібраних метрик, журналів, даних трасування додатків.

3.2.3 Допоміжні функції в eBPF та заголовки де вони оголошені

Для взаємодії eBPF програм із ядром Linux використовуються спеціальні допоміжні функції, які ядро експортує для віртуальної машини eBPF. Ці функції забезпечують доступ до системних ресурсів, таймерів, мережних стеків, карт, трасувальних механізмів та низки внутрішніх API ядра. Використання допоміжних функцій є єдиним дозволеним способом взаємодії eBPF коду з ядром, оскільки прямий доступ до пам'яті та структур ядра заборонений верифікатором.

Варто зазначити, що використовувати стандартну бібліотеку мови C теж заборонено, тому якщо потрібні функції для обробки строк, наприклад, то треба розробнику самому їх реалізовувати і таким чином, щоб цей код пройшов перевірку верифікатором.

Розглянемо заголовкові файли для eBPF програм. eBPF програми використовують спеціальний набір заголовків, які оголошують допоміжні функції, типи eBPF, макроси та допоміжні структури, а саме:

- `bpf_helpers.h`, що містить оголошення деяких допоміжних функцій, надає макроси `SEC(...)` для визначення секцій ELF, у яких розміщуються програми та карти, описує атрибути `__u64`, `__u32`, `__always_inline` та інші специфічні типи;

- `vmlinux.h`, який є автоматично згенерованим файлом, що містить типи поточного ядра Linux, забезпечує доступ до структур ядра без необхідності вручну підтримувати відповідність версій та зміщень, дає змогу

використовувати реальні структури ядра (наприклад, `task_struct`, `sock`, `file`) у eBPF коді;

- `helpers.h/linux/bpf.h` файл включає основні визначення eBPF допоміжних типів, констант, перерахунку, ключів карт, кодів програм тощо. Містить визначення для `bpf_map_def` (у старих версіях), типів програм (`BPF_PROG_TYPE_*`), а також системного ABI для `bpf()`.

Тепер розглянемо основні категорії допоміжних функцій в eBPF. Серед них це функції для роботи з картами, що забезпечують взаємодію eBPF програми зі структурами даних у ядрі:

- `bpf_map_lookup_elem()`;
- `bpf_map_update_elem()`;
- `bpf_map_delete_elem()`.

Наступна категорія функцій використовується для роботи з таймерами та часом для виконання профілювання, latency трасування, побудові метрик:

- `bpf_ktime_get_ns()`;
- `bpf_jiffies64()`.

Категорія функцій для читання та запис пам'яті, які використовуються переважно у `kprobe/uprobe` програмах:

- `bpf_probe_read()`;
- `bpf_probe_read_kernel()`;
- `bpf_probe_read_user()`;
- `bpf_probe_write_user()`.

Функції передавання подій у користувацький простір (фактично ці функції є розширенням до функцій для роботи з картами, але ці працюють лише з деякими типами карт):

- `bpf_ringbuf_reserve()`;
- `bpf_ringbuf_submit()`;
- `bpf_ringbuf_discard()`;
- `bpf_ringbuf_output()`;
- `bpf_perf_event_output()`.

Категорія допоміжних функцій для роботи з мережею (застосовуються у програмах типу XDP/TC):

- `bpf_redirect()`;
- `bpf_clone_redirect()`;
- `bpf_xdp_adjust_head()`;
- `bpf_skb_store_bytes()`.

Це лише основні категорії допоміжних функцій, доступних у eBPF. Загальна їх кількість значно більша, однак більшість із них має спеціалізоване призначення й використовується у доволі вузьких сценаріях [23]. Також з кожною новою версією ядра Linux перелік доступних допоміжних функцій розширюється, оскільки eBPF інфраструктура продовжує активно розвиватися.

У межах розробленого програмного застосунку найбільш затребуваними були допоміжні функції для роботи з загальними типами карт, з картою типу `ringbuf`, а також функції, які забезпечують безпечне читання та запис у пам'ять.

3.2.4 Інструментування процесів і бібліотек: `/proc/<pid>/maps` та СИМВОЛИ

Як вже було зазначено, колектор використовує `uprobe/uretprobe` eBPF програми, щоб збирати дані з користувачьких додатків. Для підключення `uprobe` програм необхідно визначити фактичне розташування бібліотек або виконуваних файлів у пам'яті. Це здійснюється через `/proc/<pid>/maps` – файл, що містить карту пам'яті процесу, включаючи шляхи до завантажених ELF бібліотек, наприклад, `libssl.so`, та таблицю символів ELF бібліотек, що дозволяє визначити зміщення функцій для `uprobe`. Комбінація цих механізмів дозволяє точно прив'язувати eBPF програми до необхідних функцій у користувачькому просторі.

Розглянемо приклад процесу `nginx` – одного з найпоширеніших вебсерверів. Як видно з рис. 3.3, процес `nginx` підключає кілька загальносистемних бібліотек, таких як `libssl.so`, `libcrypto.so` та інші. Для кожного

сегмента пам'яті наведено його діапазон віртуальних адрес, права доступу (r/w/x/p/s), зсув у файлі, ідентифікатор пристрою, номер inode та відповідний шлях до файлу або службову позначку.

Інструментувати можна не лише підключені динамічні бібліотеки, але й самі виконувані файли, з яких запущено додатки. Наприклад, у випадку перехоплення викликів до функцій libssl.so у застосунках, написаних на платформі Node.js, слід враховувати, що Node.js компілюється зі статично вбудованою libssl. Відповідно, символи libssl.so потрібно шукати не у зовнішній бібліотеці, а безпосередньо у бінарному файлі node.

```
f0e000-581562f3e000 r--p 00000000 00:22 587124 /usr/sbin/nginx
f3e000-581563038000 r-xp 00030000 00:22 587124 /usr/sbin/nginx
038000-581563078000 r--p 0012a000 00:22 587124 /usr/sbin/nginx
078000-58156307b000 r--p 0016a000 00:22 587124 /usr/sbin/nginx
07b000-58156309a000 rw-p 0016d000 00:22 587124 /usr/sbin/nginx
09a000-58156315c000 rw-p 00000000 00:00 0
97d000-58158da63000 rw-p 00000000 00:00 0 [heap]
6be000-7096576c0000 rw-p 00000000 00:00 0
6c0000-7096576c5000 r--p 00000000 00:22 564555 /usr/lib/x86_64-linux-gnu/libzstd.so.1.5.7
6c5000-709657774000 r-xp 00005000 00:22 564555 /usr/lib/x86_64-linux-gnu/libzstd.so.1.5.7
774000-709657788000 r--p 000b4000 00:22 564555 /usr/lib/x86_64-linux-gnu/libzstd.so.1.5.7
788000-709657789000 r--p 000c8000 00:22 564555 /usr/lib/x86_64-linux-gnu/libzstd.so.1.5.7
789000-70965778a000 rw-p 000c9000 00:22 564555 /usr/lib/x86_64-linux-gnu/libzstd.so.1.5.7
78a000-7096577b2000 r--p 00000000 00:22 564472 /usr/lib/x86_64-linux-gnu/libc.so.6
7b2000-709657917000 r-xp 00028000 00:22 564472 /usr/lib/x86_64-linux-gnu/libc.so.6
917000-70965796d000 r--p 0018d000 00:22 564472 /usr/lib/x86_64-linux-gnu/libc.so.6
96d000-709657971000 r--p 001e2000 00:22 564472 /usr/lib/x86_64-linux-gnu/libc.so.6
971000-709657973000 rw-p 001e6000 00:22 564472 /usr/lib/x86_64-linux-gnu/libc.so.6
973000-709657982000 rw-p 00000000 00:00 0
982000-709657985000 r--p 00000000 00:22 564553 /usr/lib/x86_64-linux-gnu/libz.so.1.3.1
985000-709657999000 r-xp 00003000 00:22 564553 /usr/lib/x86_64-linux-gnu/libz.so.1.3.1
999000-7096579a0000 r--p 00017000 00:22 564553 /usr/lib/x86_64-linux-gnu/libz.so.1.3.1
9a0000-7096579a1000 r--p 0001d000 00:22 564553 /usr/lib/x86_64-linux-gnu/libz.so.1.3.1
9a1000-7096579a2000 rw-p 0001e000 00:22 564553 /usr/lib/x86_64-linux-gnu/libz.so.1.3.1
9a2000-709657a99000 r--p 00000000 00:22 564480 /usr/lib/x86_64-linux-gnu/libcrypto.so.3
a99000-709657e1a000 r-xp 000f7000 00:22 564480 /usr/lib/x86_64-linux-gnu/libcrypto.so.3
e1a000-709657f51000 r--p 00478000 00:22 564480 /usr/lib/x86_64-linux-gnu/libcrypto.so.3
f51000-709657fd4000 r--p 005af000 00:22 564480 /usr/lib/x86_64-linux-gnu/libcrypto.so.3
fd4000-709657fd7000 rw-p 00632000 00:22 564480 /usr/lib/x86_64-linux-gnu/libcrypto.so.3
fd7000-709657fda000 rw-p 00000000 00:00 0
fda000-709657ffd000 r--p 00000000 00:22 564535 /usr/lib/x86_64-linux-gnu/libssl.so.3
ffd000-7096580a6000 r-xp 00023000 00:22 564535 /usr/lib/x86_64-linux-gnu/libssl.so.3
0a6000-7096580d9000 r--p 000cc000 00:22 564535 /usr/lib/x86_64-linux-gnu/libssl.so.3
0d9000-7096580e4000 r--p 000fe000 00:22 564535 /usr/lib/x86_64-linux-gnu/libssl.so.3
0e4000-7096580e8000 rw-p 00109000 00:22 564535 /usr/lib/x86_64-linux-gnu/libssl.so.3
0e8000-7096580eb000 r--p 00000000 00:22 564520 /usr/lib/x86_64-linux-gnu/libpcre2-8.so.0.
0eb000-709658166000 r-xp 00003000 00:22 564520 /usr/lib/x86_64-linux-gnu/libpcre2-8.so.0.
166000-709658195000 r--p 0007e000 00:22 564520 /usr/lib/x86_64-linux-gnu/libpcre2-8.so.0.
195000-709658196000 r--p 000ac000 00:22 564520 /usr/lib/x86_64-linux-gnu/libpcre2-8.so.0.
196000-709658197000 rw-p 000ad000 00:22 564520 /usr/lib/x86_64-linux-gnu/libpcre2-8.so.0.
197000-709658199000 r--p 00000000 00:22 564479 /usr/lib/x86_64-linux-gnu/libcrypt.so.1.1.
199000-7096581af000 r-xp 00002000 00:22 564479 /usr/lib/x86_64-linux-gnu/libcrypt.so.1.1.
```

Рисунок 3.3 – Приклад вмісту /proc/<pid>/maps на прикладі процесу nginx

Підсумовуючи, після того як колектор визначив список процесів, які відповідають заданому фільтру та можуть бути джерелами даних, наступним кроком є підготовка до завантаження програм типу `uprobe/uretprobe`. Для цього колектор аналізує вміст файлу `/proc/<pid>/maps`, щоб виявити необхідні виконувані файли та завантажені бібліотеки, визначити їхні базові адреси та, відповідно, коректно прикріпити eBPF програми до потрібних символів у цільовому процесі.

3.3 Компіляція та завантаження eBPF програм у ядро Linux

Після того, як колектор знайшов список процесів по заданому фільтру, виявив потрібні бінарні файли і бібліотеки, колектор має завантажити скомпільовані eBPF програми для цих процесів, тому розглянемо детальніше, як відбувається процес розробки, компіляція і завантаження в ядро eBPF програм.

Реалізація eBPF програм складається з кількох основних етапів:

- написання вихідного коду (зазвичай мовою C або за допомогою високорівневих обгортки);
- компіляції у спеціальний формат eBPF байткоду;
- завантаження цього байткоду в ядро Linux;
- перевірка байткоду та подальше прикріплення до відповідних точок виконання (`kprobe`, `uprobe`, `tracpoint`, кельця `fentry/fexit` тощо).

У цьому процесі задіяні низка інструментів і бібліотек, серед яких `clang/LLVM`, `libbpf`, Go модуль `cilium/ebpf` та `bpf2go`. Розглянемо їх детальніше.

3.3.1 Компіляція eBPF коду за допомогою clang та LLVM

Вихідний код eBPF в колекторі написаний на мові C, тому для компіляції цього коду використовується компілятор clang, який у свою чергу використовує бекенд LLVM для генерації eBPF байткоду.

Розглянемо детально процес компіляції eBPF коду через clang:

- clang компілює вихідні .c файли з використанням спеціальної архітектури -target bpf;
- LLVM генерує проміжне представлення IR і компілює його у платформонезалежний eBPF байткод;
- результатом є ELF-файл, який містить: секції програм eBPF (наприклад, .text), карти eBPF (.maps), метадані, необхідні для CO-RE (Compile Once – Run Everywhere), інформацію про точки приєднання.

ELF виступає контейнером, який зберігає всі частини eBPF програми в структурованому вигляді і може бути інтерпретований завантажувачами на рівні користувацького простору (рис. 3.4).

```
$ readelf -s pkg/ebpf/bpf_bpfeb.o
```

Symbol table '.symtab' contains 8 entries:

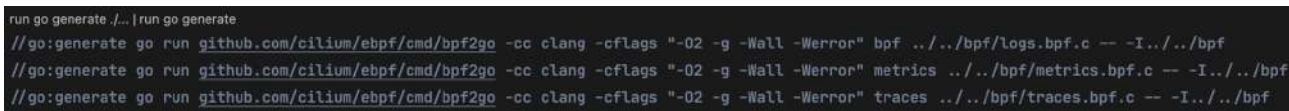
Num:	Value	Size	Type	Bind	Vis	Ndx	Name
0:	0000000000000000	0	NOTYPE	LOCAL	DEFAULT	UND	
1:	0000000000000000	0	SECTION	LOCAL	DEFAULT	3	tracepoint/sysca[...]
2:	00000000000000190	0	NOTYPE	LOCAL	DEFAULT	3	LBB0_7
3:	00000000000000150	0	NOTYPE	LOCAL	DEFAULT	3	LBB0_4
4:	00000000000000178	0	NOTYPE	LOCAL	DEFAULT	3	LBB0_6
5:	0000000000000000	416	FUNC	GLOBAL	DEFAULT	3	trace_write_enter
6:	0000000000000000	16	OBJECT	GLOBAL	DEFAULT	6	log_events
7:	0000000000000000	13	OBJECT	GLOBAL	DEFAULT	5	LICENSE

Рисунок 3.4 – Структура скомпільованого ELF файлу для eBPF програми для збору журналів в колекторі

3.3.2 bpf2go та генерація обгортки для Go

Для проєктів, написаних на Go, часто використовується інструмент bpf2go (частина пакета cilium/ebpf). Його завдання це автоматично викликати Clang/LLVM для компіляції .c файлів, генерувати Go-код, який інкапсулює всі необхідні структури, мапи та програми, вбудовувати eBPF об'єкт прямо у бінарний файл Go у вигляді байтових масивів, забезпечувати зручний API для завантаження і управління eBPF компонентами. Це дозволяє уникнути ручної роботи з ELF-файлами та робить використання eBPF у Go набагато простішим.

Як вже було зазначено, що bpf2go спрощує компіляцію eBPF коду, тому при розробці колектору було використано bpf2go та Go інструкцію `//go:generate` для зручного компіляції усіх eBPF програм для подальшого завантаження в ядро (рис. 3.5).



```
run go generate ./... | run go generate
//go:generate go run github.com/cilium/ebpf/cmd/bpf2go -cc clang -cflags "-O2 -g -Wall -Werror" bpf ../../bpf/logs.bpf.c -- -I../../bpf
//go:generate go run github.com/cilium/ebpf/cmd/bpf2go -cc clang -cflags "-O2 -g -Wall -Werror" metrics ../../bpf/metrics.bpf.c -- -I../../bpf
//go:generate go run github.com/cilium/ebpf/cmd/bpf2go -cc clang -cflags "-O2 -g -Wall -Werror" traces ../../bpf/traces.bpf.c -- -I../../bpf
```

Рисунок 3.5 – `go:generate` команди для компіляції eBPF коду через bpf2go

Як показано на рис. 3.5, bpf2go також приймає аргументи для компілятора clang, зокрема: `-O2` для оптимізації та зменшення розміру коду, `-g` для додавання інформації для відладки та перегляду інструкцій, `-Wall` для увімкнення всіх стандартних попереджень компілятора та `-Werror`, який змушує компілятор розглядати будь-які попередження як помилки.

Додатково, bpf2go генерує два варіанти ELF-файлів: `*_bpfel.o` для little-endian систем і `*_bpfeb.o` для big-endian. Наявність обох версій забезпечує сумісність із ядром Linux на різних архітектурах та дозволяє реалізувати концепцію CO-RE (Compile Once – Run Everywhere), завдяки якій одна й та сама програма може виконуватися на різних платформах без повторної компіляції. Такий підхід звільняє розробника від необхідності вручну керувати порядком

байтів (endianess) або деталями завантаження програми в ядро, при цьому ядро за допомогою Verifier гарантує коректність виконання eBPF програми.

3.3.3 Завантаження eBPF програм через libbpf та cilium/ebpf

Щоб завантажити eBPF програму чи карту є декілька варіантів. Один з основних це використовуючи libbpf.

Libbpf це C бібліотека, яка широко використовується у світі eBPF. Вона дозволяє робити наступне:

- парсинг ELF-файлів;
- створення та ініціалізацію карт eBPF;
- завантаження програм у ядро через системний виклик bpf();
- застосування CO-RE релокацій за допомогою BTF (BPF Type Format);
- прикріплення програм до точок виконання (kprobe, uprobe, tracepoint, XDP тощо).

Як було зазначено вище, libbpf використовує системний виклик bpf(). Системний виклик bpf() у Linux забезпечує низькорівневий доступ до ядра для створення та завантаження eBPF програм і карт, використовуючи структуру bpf_attr для передачі параметрів і команд (BPF_PROG_LOAD, BPF_MAP_CREATE тощо). Бібліотека libbpf інкапсулює ці виклики, спрощуючи роботу з ELF файлами, BTF та CO-RE relocations.

Але так як колектор написаний на мові Go, використання libbpf не є зручним, хоч і можливим. Так як, libbpf виокристовує системний виклик bpf(), то можна реалізувати подібну логіку і в Go. Саме це і було зроблено командою Cilium у модулі cilium/ebpf.

cilium/ebpf це Go-бібліотека, яка реалізує основну функціональність libbpf у середовищі Go:

- читання ELF файлів;
- створення карт і програм;

- контроль за життєвим циклом eBPF об'єктів;
- прикріплення до точок у просторі ядра та користувацьких процесів.

Перед тим як байткод буде виконуватись у ядрі, він проходить обов'язкову перевірку компонентом під назвою верифікатор. Це критичний компонент системи безпеки ядра.

3.3.4 Верифікація eBPF програм

Верифікатор перевіряє відсутність нескінченних циклів, коректність доступу до пам'яті, відповідність типів, правильність доступу до карт, коректний розмір стеку, валідність переходів, відсутність небезпечних інструкцій. Лише після успішної перевірки програма допускається до виконання.

На рис. 3.6 візуально зображений процес компіляції, завантаження в ядро, та прикріплення eBPF програм до окремих системних викликів. Процес прикріплення буде розглянутий в наступному розділі.

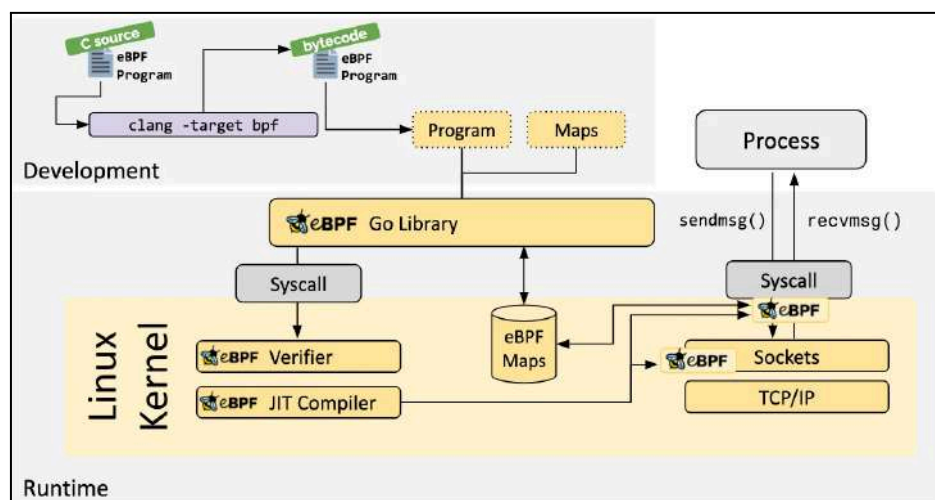


Рисунок 3.6 – Загальна діаграма процесу компіляції та завантаження eBPF програм [24]

Імплементация eBPF верифікатора поки що не є ідеальною: досить часто виникають false-positive спрацьовування, через що необхідно додатково перепроверити діапазони значень змінних та переконуватися, що покажчик не дорівнює NULL, навіть де це не потрібно. Водночас з кожним новим релізом ядра Linux точність і надійність верифікатора поступово покращуються [25].

Так як точність верифікатора покращується, збільшується і цикломатична складність. Цикломатична складність – це метрика, запропонована Томасом МакКейбом, яку застосовують для вимірювання логічної складності програмного коду. Вона визначає кількість лінійно незалежних шляхів у програмі, тобто показує, скільки унікальних маршрутів виконання може пройти алгоритм згідно зі своєю структурою та розгалуженнями [26]. За формулою МакКейба (3.1), наведеною нижче, її можна вирахувати для верифікатора:

$$M = E - N + 2P, \quad (3.1)$$

де M – цикломатична складність;

E – кількість переходів між базовими блоками;

N – кількість базових блоків;

P – кількість зв'язних компонент.

Використовуючи інструмент Lizard, який використовує формулу наведену вище, можемо порахувати цикломатичну складність верифікатора протягом декількох релізів ядра Linux [27].

Нижче наведений графік з значеннями середньої цикломатичної складності верифікатора протягом декількох релізів ядра Linux (рис. 3.7).

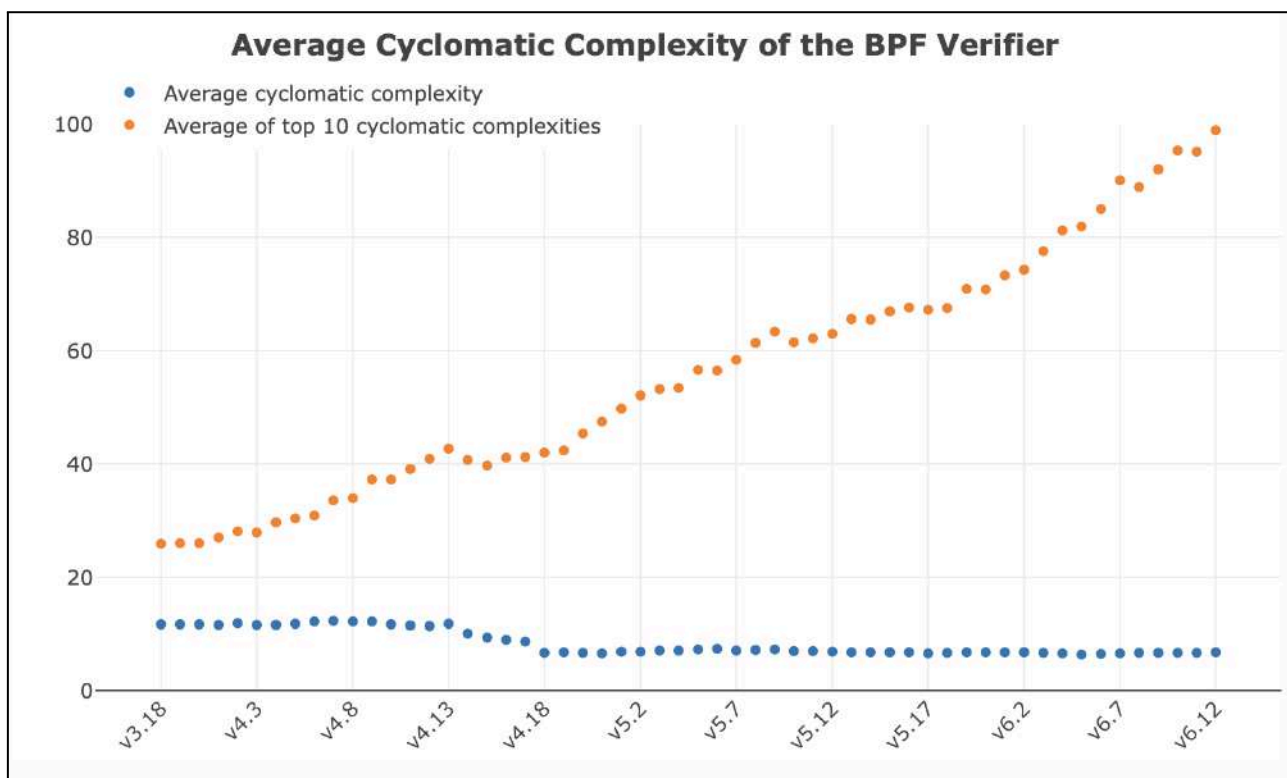


Рисунок 3.7 – Середня цикломатична складність eBPF верифікатора протягом декількох релізів Linux [28]

З графіку на рис. 3.7 можна помітити, що середня цикломатична складність верифікатору зменшується, але це не означає, що код верифікатору став простішим. Це пов'язано з тим, що код з часом рефакториться і логіка виноситься в інші функції. Якщо звернути увагу на помаранчеву лінію, що характеризує середню цикломатичну складність 10 найбільш використовуваних функцій в eBPF верифікаторі, то можна побачити, як їх складність зростає.

Чому це важливо? Як було вже зазначено, верифікатор перевіряє eBPF програми, які завантажують користувач системи Linux, тому є дуже важливим процес перевірки цього коду, аби запобігти зловмиснику спровокувати неправильну поведінку ядра, або взагалі викликати критичну поломку всередині ядра. Верифікатор є гарантією стабільності запущених програм в ядрі Linux і тому важливо, щоб якість і точність верифікатору покращувалась кожного релізу, але з цим не вносились баги в код верифікатору.

3.3.5 Як cilium/ebpf завантажує програми

Завдяки cilium/ebpf модулю та згенерованим функціям і класам bpf2go, які зображені на рис. 3.8, можна лише одним викликом функції loadMetricsObjects завантажити усі eBPF програми та карти необхідні для збору метрик з ядра. Як проілюстровано на рис. 3.9, далі можна прикріплювати eBPF програми до окремих функцій ядра або користувацьких програм.

```
1 ref | move...
func NewMetricsTracer() (*MetricsTracer, error) {
    → objs := metricsObjects{}
    → if err := loadMetricsObjects(obj: &objs, opts: nil); err != nil {
    →     return nil, fmt.Errorf(format: "loading objects: %w", a...: err)
    → }

    → forkLink, err := link.Tracepoint(group: "sched", name: "sched_process_fork", prog: objs.TraceSchedProcessFork, opts: nil)
    → if err != nil {
    →     objs.Close()
    →     return nil, fmt.Errorf(format: "opening fork tracepoint: %w", a...: err)
    → }

    → oomLink, err := link.Tracepoint(group: "oom", name: "mark_victim", prog: objs.TraceOomKill, opts: nil)
    → if err != nil {
    →     forkLink.Close()
    →     objs.Close()
    →     return nil, fmt.Errorf(format: "opening oom tracepoint: %w", a...: err)
    → }
}
```

Рисунок 3.8 – Приклад завантаження eBPF об'єктів в ядро за допомогою згенерованих функцій і класів на прикладі MetricsTracer

```
func loadMetricsObjects(obj interface{}, opts *ebpf.CollectionOptions) error {
    → spec, err := loadMetrics()
    → if err != nil {
    →     return err
    → }

    → return spec.LoadAndAssign(to: obj, opts)
}

// metricsSpecs contains maps and programs before they are loaded into the kernel.
//
// It can be passed ebpf.CollectionSpec.Assign.
no refs | impl... | move...
type metricsSpecs struct {
    → metricsProgramSpecs
    → metricsMapSpecs
}
```

Рисунок 3.9 – Приклад згенерованого коду для завантаження eBPF об'єктів для збору даних для метрик колектора

На рис. 3.10 можна побачити імплементацію функції LoadAndAssign з модулю cilium/ebpf, що займається безпосередньо завантаженням eBPF програм та карт за допомогою внутрішньої функції BPF, що в середині використовує системний виклик bpf().

На рис. 3.11 показано як cilium/ebpf використовує системний виклик bpf().

```
func (cs *CollectionSpec) LoadAndAssign(to interface{...}, opts *CollectionOptions) error {
→ loader, err := newCollectionLoader(coll: cs, opts)
→ if err != nil {
→     return err
→ }
→ defer loader.close()

→ // Support assigning Programs and Maps, lazy-loading the required objects.
→ assignedMaps := make(map[string]bool)
→ assignedProgs := make(map[string]bool)

→ getValue := func(typ reflect.Type, name string) (interface{...}, error) {
→     switch typ {
→
→     case reflect.TypeOf(i: (*Program)(nil)):
→         assignedProgs[name] = true
→         return loader.loadProgram(progName: name)
→
→     case reflect.TypeOf(i: (*Map)(nil)):
→         assignedMaps[name] = true
→         return loader.loadMap(mapName: name)
→
→     default:
→         return nil, fmt.Errorf(format: "unsupported type %s", a...: typ)
→     }
→ }
→ }
```

Рисунок 3.10 – Приклад функції LoadAndAssign з модулю cilium/ebpf, що займається завантаженням eBPF програм і карт

```
// BPF wraps SYS_BPF.
//
// Any pointers contained in attr must use the Pointer type from this package.
func BPF(cmd Cmd, attr unsafe.Pointer, size uintptr) (uintptr, error) {
    // Prevent the Go profiler from repeatedly interrupting the verifier,
    // which could otherwise lead to a livelock due to receiving EAGAIN.
    if cmd == BPF_PROG_LOAD || cmd == BPF_PROG_RUN {
        maskProfilerSignal()
        defer unmaskProfilerSignal()
    }

    for {
        r1, _, errNo := unix.Syscall(trap: unix.SYS_BPF, a1: uintptr(cmd), a2: uintptr(attr), a3: size)
        runtime.KeepAlive(x: attr)

        // As of ~4.20 the verifier can be interrupted by a signal,
        // and returns EAGAIN in that case.
        if errNo == unix.EAGAIN && cmd == BPF_PROG_LOAD {
            continue
        }

        var err error
        if errNo != 0 {
            err = wrappedErrno{errNo}
        }

        return r1, err
    }
}
```

Рисунок 3.11 – Приклад імплементації з модулю cilium/ebpf, що використовує системний виклик bpf()

Якщо завантаження пройшло успішно, за допомогою інструмента bpftool ми можемо побачити які eBPF програми та карти зараз знаходяться в ядрі. Результат команди `sudo bpftool prog list` зображений на рис 3.12.

```
27882: tracepoint name trace_sched_process_fork tag 00109d5be093d210 gpl
        loaded_at 2025-11-16T16:03:01+0000 uid 0
        xlated 424B jited 240B memlock 4096B map_ids 15505,15506
        btf_id 16032
27897: perf_event name sample_function_calls tag 036d14e4a60f6a53 gpl
        loaded_at 2025-11-16T16:03:22+0000 uid 0
        xlated 408B jited 241B memlock 4096B map_ids 15512,15513
        btf_id 16052
27906: tracepoint name trace_write_enter tag 6bdd65ba0d7cd5fd gpl
        loaded_at 2025-11-16T16:03:36+0000 uid 0
        xlated 416B jited 239B memlock 4096B map_ids 15516
        btf_id 16062
```

Рисунок 3.12 – Завантажені eBPF програми колектора

3.4 Читання даних з eBPF програм на стороні Go (user-space)

Як вже було зазначено, для читання даних з eBPF програм в розробленому додатку використовується циклічний буфер (ringbuf). На стороні Go, з ringbuf можна отримати масив байтів, який потім можна декодувати в зручні структури даних. З початку, було використано найпростіший метод декодування даних із масиву байтів в структуру – за допомогою функції `binary.Read`, яка в якості аргументів приймає сам масив і посилання на структуру в яку треба покласти отримані дані. Функція динамічно визначить кількість полів в структурі, типи даних, і динамічно проставить дані в ці поля. Але після тестування продуктивності додатку виявилось, що обробка даних не є достатньо швидкою. Тому було прийнято рішення, порівняти існуючі методи декодування сирих даних з масиву в структуру за допомогою бенчмарків, які є частиною стандартної бібліотеки Go.

З додаткових варіантів декодування були:

- використання `binary.Read` з `sync.Pool` який зберігатиме попередньо виділений буфер з об'єктами;
- використання окремих функцій під різні типи даних з пакету `binary`, таких як `binary.Uint16`, `binary.Uint32`, і т.д.

Нижче наведено результати запуску бенчмарків (рис. 3.13, 3.14):

Benchmark	Name	Iterations	Time/Op	Bytes/Op	Allocs/Op
BinaryRead		302,144	11,083 ns	1200 B	2 allocs
FieldByField		89,416,429	38.75 ns	0 B	0 allocs
BinaryReadWithPreallocatedBuffer		317,779	11,072 ns	1152 B	1 alloc

Рисунок 3.13 – Результати описаного бенчмарку

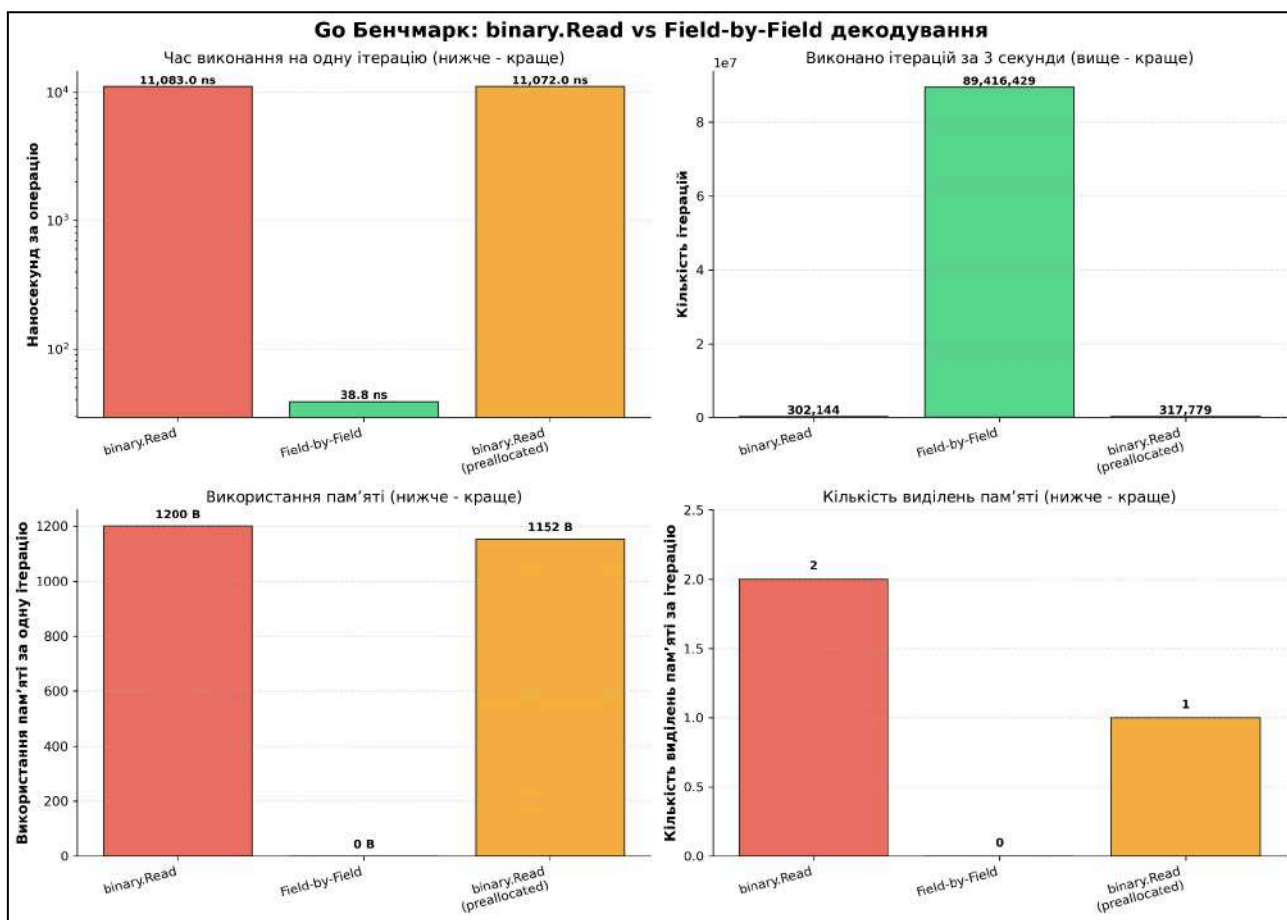


Рисунок 3.14 – Візуалізація результатів описаного бенчмарку

Результати показали, що найшвидшим методом декодування є “Field-By-Field”, тобто декодування окремих полів за допомогою функцій під окремі типи даних з модулю `binary`, надаючи фіксовану позицію даних для цього поля в масиві. Як можна бачити з рис. 3.13 і 3.14, за фіксовану кількість часу для запуску кожного бенчмарку, найбільшу кількість ітерацій було можливо зробити використовуючи підхід “Field-By-Field” – 89,416,429, що майже в 300 разів більше ніж використовуючи підхід “Binary-Read” або “Binary-Read-With-Preallocated-Buffer”. Швидкість однієї спроби декодування сирих даних в структуру використовуючи підхід “Field-By-Field” є в 290 разів швидший ніж інші, при цьому до цей метод не потребує додаткових виділень в heap пам’ять, як інші методи. Одним із мінусів цього підходу, є те що треба

знати порядок полів і їх розмір (кількість байт що займає окремий тип даних), але це не є критичним, якщо є можливість отримати такі суттєві покращення в обробці даних. Однією з причин, чого така суттєва різниця між результатами зумовлена використанням пакету reflect в функції binary.Read, що відома не найкращими показниками продуктивності.

3.5 eBPF та контейнерезовані середовища

Найпоширенішою практикою є розгортання додатків які працюють з eBPF і ядром Linux це через DaemonSet, що гарантує запуск одного екземпляра на кожному вузлі кластера з необхідними привілеями та доступом до ресурсів ядра. Такий підхід дозволяє збирати системні метрики, логи та трасування всіх контейнерів незалежно від того, в якому поді вони знаходяться. Тому для розгортання колектору рекомендується використовувати саме DaemonSet на відміну від Deployment, який може запустити один і той самий контейнер на одній і тій самій машині декілька разів, що може спричинити дублікацію даних (рис. 3.15).

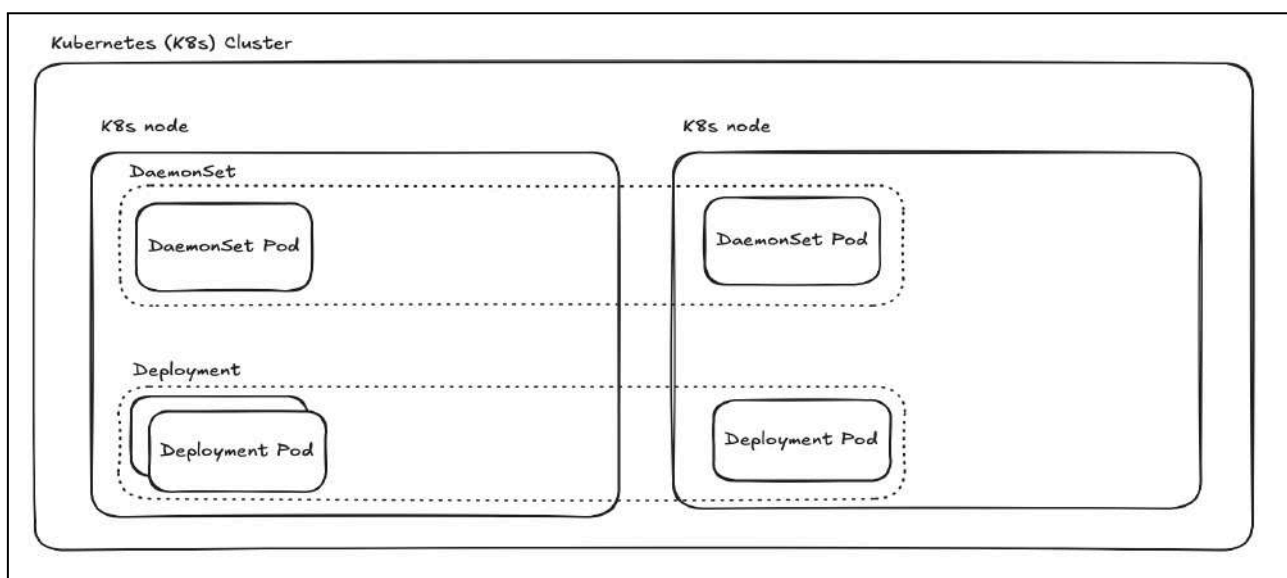


Рисунок 3.15 – Порівняння DaemonSet та Deployment методів розгортання додатків в кластері Kubernetes (K8s) [29]

Але не на кожному контейнерезованому середовищі можливо розгорнути додатки, що працюють з eBPF. В Amazon Web Services існує доволі популярний сервіс Elastic Container Service, що слугує спрощеною імплементацією оркестрації контейнерів, на відміну від Kubernetes. Цей сервіс дозволяє створити кластер в двох режимах: оснований на EC2 машинах (на віртуальних машинах) та Fargate, що позбавляє користувача контролю над віртуальними машинами на яких запускаються контейнери, цим самим спрощує підтримку такої інфраструктури (рис. 3.16), але позбавляє можливості взаємодіяти з ядром цих віртуальних машин.

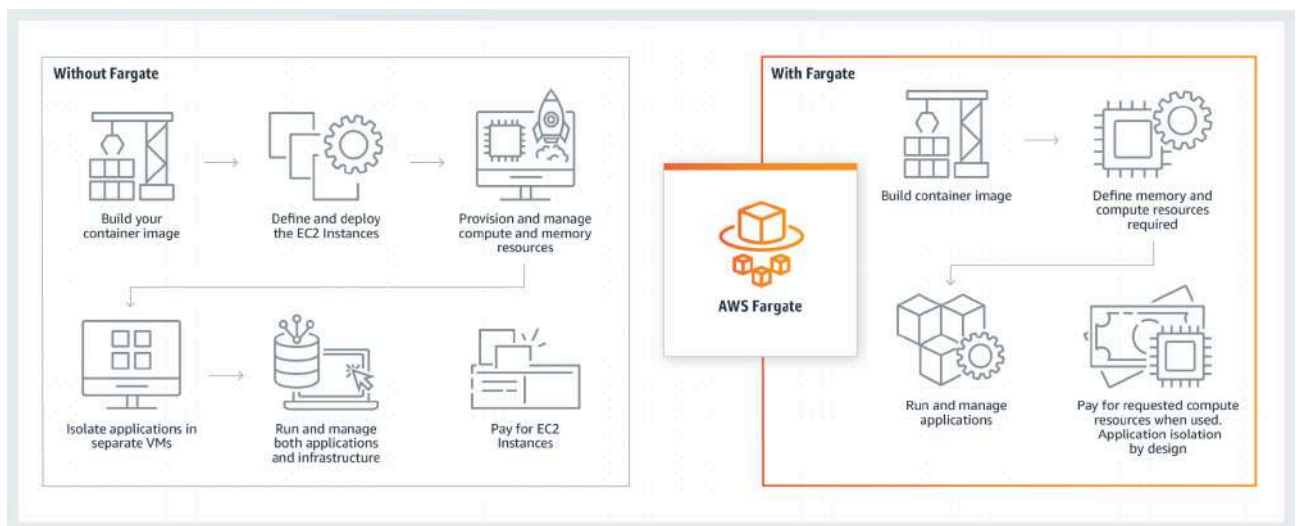


Рисунок 3.16 – Порівняння можливостей Fargate з іншими рішеннями розгортання додатків [30]

Як вже було зазначено, для розроблюваного колектору моніторингових даних для запуску в контейнерезованих середовищах потрібен доступ до PID простору імен всієї машини, доступ до мережевого простору машини, і спеціальний перелік capabilities, що дозволяють запускати колектор не від root користувача з необмеженими правами, а лише з окремими. При розгортанні будь-яких додатків, завжди рекомендується притримуватись принципу least privilege, аби знизити так званий attack surface (поверхню атаки).

Контейнерезовані середовища надають широкий спектр можливостей конфігурування дозволів та рівнів ізоляції контейнерів.

3.6 Опис OpenTelemetry для експорту моніторингових даних

Важливою складовою роботи колектора також є механізм експорту зібраних даних через стандартизований інтерфейс та інструменти OpenTelemetry, який забезпечує їх подальшу обробку, зберігання та візуалізацію у зовнішніх системах (рис. 3.17).

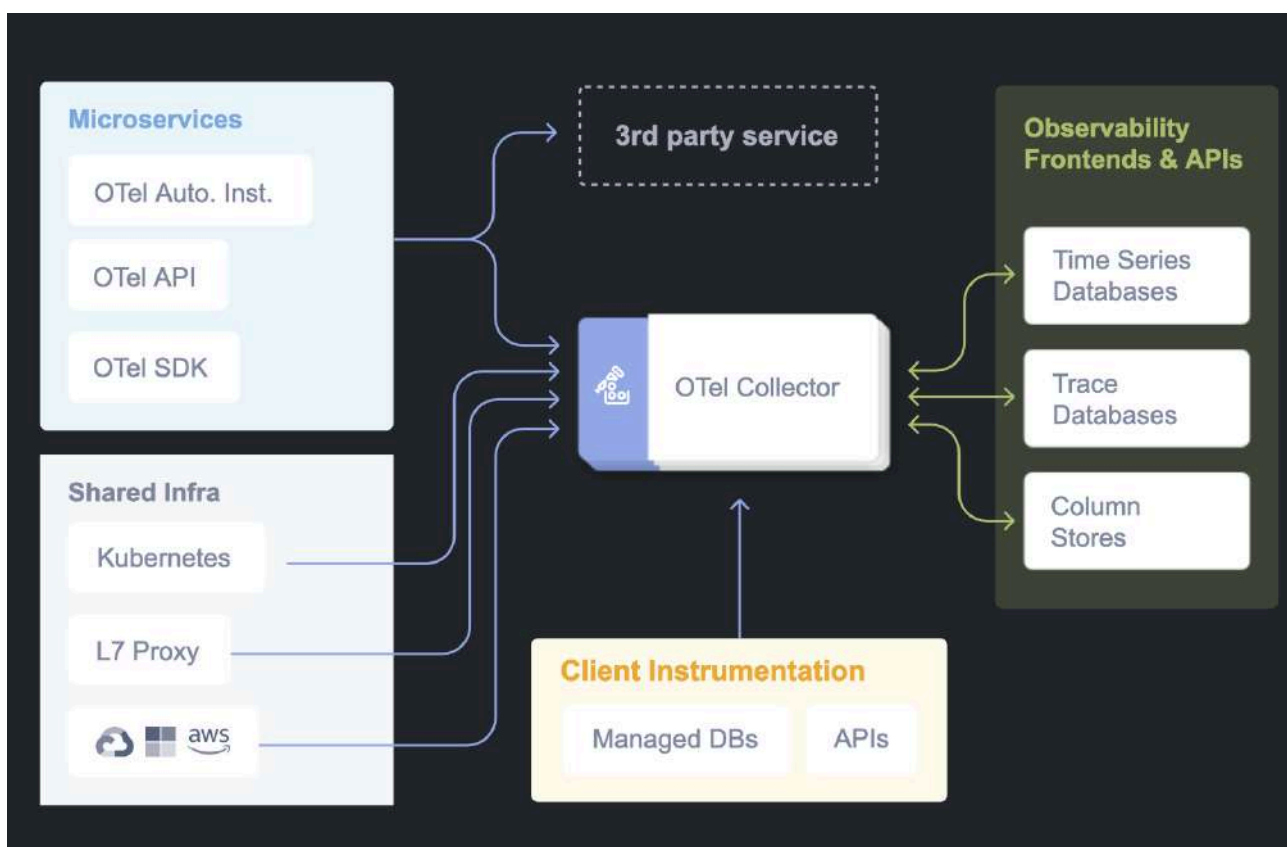


Рисунок 3.17 – Візуалізація екосистеми OpenTelemetry [31]

Екосистема OpenTelemetry надає SDK майже під кожну популярну мову програмування для швидкої інтеграції інструментів моніторингу. В випадку з колектором використовується пакет `go.opentelemetry.io/otel/sdk/trace` для

експорту трасувань, `go.opentelemetry.io/otel/sdk/metric` для експорту метрик і `go.opentelemetry.io/otel/sdk/log` для експорту журналів.

Для експорту зібраних моніторингових даних, колектор використовує OTLP, що розшифровується як OpenTelemetry Protocol, що дозволяє передавати дані по HTTP або gRPC використовуючи Protocol Buffers для серіалізації даних [32].

У коді колектора використовуються спеціальні провайдери (Provider), які відповідають за створення об'єктів для роботи з журналами, метриками та трасуванням. Зокрема, виклик `provider.Tracer()` у класі `sdktrace.TracerProvider` повертає об'єкт типу `trace.Tracer`, що надає API для створення нових проміжків. Метод `span.Start()` ініціює створення об'єкта `trace.Span` – основної одиниці трасування, яка містить дані про операцію: її назву, тривалість, часові мітки, а також набір додаткових атрибутів, що описують контекст виконання. Після завершення операції виконується `span.End()`, і сформований спан стає готовим для обробки та експорту.

На стороні систем візуалізації, таких як Grafana Tempo, Jaeger або Zipkin, ці проміжки об'єднуються у цілісні ланцюжки виконання запитів. Користувач може переглядати їх у вигляді дерева або графу, що дозволяє легко аналізувати продуктивність, виявляти “вузькі місця” та досліджувати взаємозв'язки між компонентами системи (рис. 3.18).

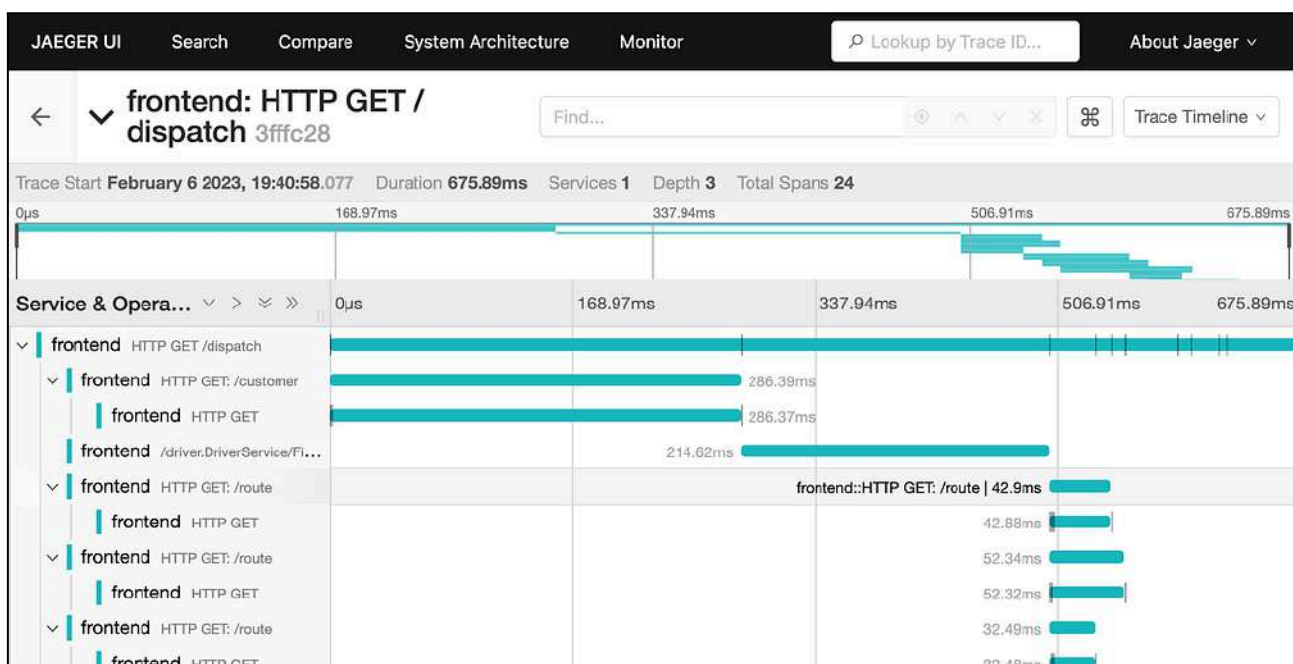


Рисунок 3.18 – Приклад візуалізації трасування з декількома проміжками [33]

Експорт журналів організовано подібним чином до експорту трасувань. Для кожного рядка, отриманого зі стандартних потоків Stdout або Stderr, колектор за допомогою OpenTelemetry SDK створює об'єкт типу `log.Record`. Під час формування запису заповнюються всі основні поля: мітка часу події (Timestamp), рівень логування (SeverityNumber та SeverityText), текст повідомлення (Body), а також набір атрибутів (Attributes), що містять пари ключ-значення для додаткової фільтрації. Якщо додаток використовує структуроване логування (наприклад, JSON-логування), ці ключі та значення парсяться та додаються до `Record` як окремі атрибути, що дозволяє користувачам потім створювати окремі колонки по цим атрибутам в системах агрегації і візуалізації журналів (рис. 3.19).

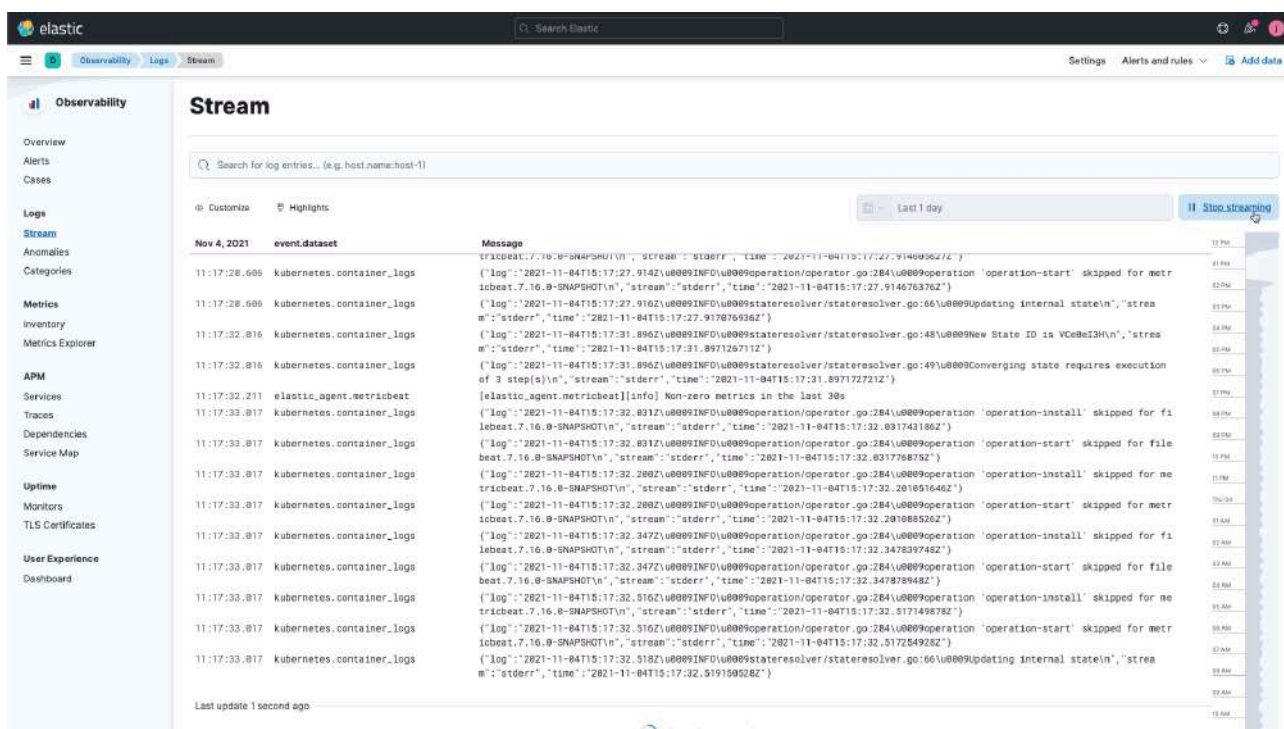


Рисунок 3.19 – Приклад візуалізації потоку журналів в системі Kibana [34]

Для роботи з метриками в колекторі так само використовується спеціальний компонент типу Provider – MeterProvider, який відповідає за створення метрик й керування їх життєвим циклом. За допомогою методу `provider.Meter()` клас `sdkmeter.MeterProvider` повертає об'єкт типу `metric.Meter`, що надає API для створення різних видів метрик, таких як лічильники (Counter), гістограми (Histogram) та метрики типу вимірювач. Наприклад, виклик `meter.Int64Counter()` створює інструмент для інкрементування числових значень, а `meter.Float64Histogram()` дозволяє фіксувати розподіли затримок або розмірів подій. Після створення інструмента виклики на кшталт `counter.Add(ctx, value, attributes...)` або `histogram.Record(ctx, duration, attributes...)` реєструють нові точки метрик, які зберігають виміряне значення та контекст виконання.

У розроблюваному колекторі реєструється кілька типів метрик, зокрема:

- `process.oom_kills.total` – лічильник, що показує скільки процесів було припинено із-за високого (за межами ліміту) споживання пам'яті;

- `process.starts.total` – лічильник, що показує скільки процесів було запущено або перезапущено;
- `process.active_pids` – метика типу вимірювач, що показує скільки процесів наразі запущено;
- `process.memory.usage` – метика типу вимірювач, що показує скільки пам'яті процес використовує;
- `process.cpu.usage` – метика типу вимірювач, що показує скільки процесорного часу процес використовує.

3.7 Висновки за розділом 3

У розділі наведено основні технічні рішення, використані під час реалізації системи моніторингу на основі eBPF. Описано роботу з процесами та PID-просторами, структуру файлової системи `/proc`, необхідні можливості ядра та особливості функціонування у контейнеризованих середовищах. Розглянуто процес компіляції та завантаження eBPF програм, роботу верифікатора, використання карт, `ringbuf` та різних типів програм (`kprobe`, `uprobe`, `tracpoint`, `fentry/fexit`), а також механізми CO-RE (Compile Once – Run Everywhere).

Подано загальний процес роботи системи, описано взаємодію з Linux через заголовки ядра, `vmlinux`, системні виклики й файлові дескриптори. Розглянуто інтеграцію з контейнерами та Kubernetes (DaemonSet) і використання інструмента `bpftool` для діагностики. Окремо описано архітектуру OpenTelemetry та способи експорту метрик, логів і трасувань, зібраних eBPF компонентами.

4 ЕКСПЛУАТАЦІЯ, ТЕСТУВАННЯ ТА ЕКСПЕРИМЕНТАЛЬНЕ ДОСЛІДЖЕННЯ ПРОГРАМИ

4.1 Опис застосування

Розроблений програмний продукт призначений для збирання, обробки та аналізу моніторингових даних у системах, що базуються на ядрі Linux. Його основною метою є забезпечення зручного та ефективного збору метрик, журналів подій і даних трасування як із самої операційної системи, так і з програмних додатків, встановлених у ній.

У програмному продукті реалізовано такі основні можливості:

- збір системних метрик (використання процесора, пам'яті, вводу/виводу, мережевої активності тощо);
- отримання журналів подій та даних трасування процесів для подальшого аналізу;
- гнучке налаштування конфігурації через feature flags і фільтри за назвами процесів;
- експорт зібраних даних у зовнішні системи моніторингу та аналітики (наприклад, Prometheus, Grafana, Loki тощо).

4.2 Умови виконання програми

Для коректної роботи розробленого програмного додатка необхідно дотримуватися певних вимог до апаратного та програмного забезпечення. Додаток підтримує роботу виключно в операційних системах сімейства Linux і може функціонувати як на фізичних машинах, так і у віртуалізованих або контейнеризованих середовищах.

Мінімальні вимоги до апаратної частини

- процесор одним або більше ядрами та тактовою частотою не нижче 2,5 ГГц;
- щонайменше 4 ГБ оперативної пам'яті;

- не менше 5 ГБ вільного простору на диску для завантаження самого додатку або його Docker контейнеру;
- підтримка Gigabit Ethernet для швидкої передачі даних.

Вимоги до програмного забезпечення:

- операційна система Linux (рекомендовані дистрибутиви: Ubuntu, Debian, CentOS або інші сумісні);
- встановлена актуальна версія Docker або Kubernetes у разі розгортання додатка в контейнеризованому середовищі;
- потребує повного (read-write) доступу до файлової системи;
- потребує доступу до мережевого простору імен системи;
- потребує доступу до PID простору імен системи;
- потребує наступних Linux capabilities – CAP_SYS_ADMIN, CAP_SYS_PTRACE, CAP_BPF, CAP_PERFMON.

4.3 Інструкція по експлуатації програми

4.3.1 Звернення до програми

Перед запуском колектора моніторингових даних необхідно розгорнути супроводжуючі сервіси, які допоможуть зі зберіганням метрик, трасувань і логів, а також з візуалізацією даних. На рис. 4.1 та 4.2 зображено запуск супроводжуючих сервісів через docker compose.

```

$ docker compose up -d
[+] Running 38/5
✓ loki 6 layers [#####] 0B/0B Pulled
✓ tempo 2 layers [##] 0B/0B Pulled
✓ otel-collector 3 layers [###] 0B/0B Pulled
✓ grafana 10 layers [#####] 0B/0B Pulled
✓ prometheus 12 layers [#####] 0B/0B Pulled
[+] Running 5/6
.: Network visibility_observability Created
✓ Container tempo Started
✓ Container prometheus Started
✓ Container loki Started
✓ Container grafana Started
✓ Container otel-collector Started

```

Рисунок 4.1 – Запуск сервісів через docker compose

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS
6f6b6bf4e908	grafana/grafana:10.3.3	"/run.sh"	7 seconds ago	Up 5 seconds	0.0.0.0:3000->3000/tcp, :::3000
bb6d98363b2a	otel/opentelemetry-collector-contrib:0.96.0	"/otelcol-contrib --"	7 seconds ago	Up 5 seconds	0.0.0.0:4317-4318->4317-4318/tcp, ::4317-4318->4317-4318/tcp, 0.0.0.0:8888-8889->8888-8889/tcp, ::8888-8889->8888-8889/tcp, 55678-55679/tcp
785ddad21c10	prom/prometheus:v2.50.1	"/bin/prometheus --c."	8 seconds ago	Up 6 seconds	0.0.0.0:9090->9090/tcp, :::9090
4c4b81ba8ac0	grafana/loki:2.9.4	"/usr/bin/loki -conf."	8 seconds ago	Up 6 seconds	0.0.0.0:3100->3100/tcp, :::3100
a853abe29a19	grafana/tempo:2.3.1	"/tempo -config.file."	8 seconds ago	Up 6 seconds	0.0.0.0:3200->3200/tcp, :::3200

Рисунок 4.2 – Запущені контейнери допоміжних сервісів

Після того як допоміжні сервіси запущені, можна запускати сам колектор, який за замовчуванням буде відправляти зібрані дані на localhost:4317, що є адресою OTEL collector, який потім відправляє дані на Tempo, Prometheus та Loki.

На рис. 4.3 показано журнали роботи колектора під час запуску з налаштованим фільтром на процес із назвою nginx.

```
OTel exporter created successfully
Started capturing process logs...
Started capturing process metrics...
Traces tracer created successfully

=== Attaching to 'nginx' processes ===
Function filter: 'ngx_'
Max functions: 500

Found 2 process(es) named 'nginx'
Attaching to PID 1915149 (/proc/1915149/root/usr/sbin/nginx)...
Discovered 500 functions matching 'ngx_'
Attaching to PID 1915232 (/proc/1915232/root/usr/sbin/nginx)...
Discovered 500 functions matching 'ngx_'

Successfully attached 1000 uprobes

Started capturing function traces...

Press Ctrl+C to stop...
```

Рисунок 4.3 – Журнали роботи колектора після запуску

Після того як колектор був запущений, користувач може відкрити сторінку в браузері за адресою localhost:3000, після чого користувачу відкриється головна сторінка Grafana. Аби переглянути зібрані логи, користувач має відкрити бокове меню і там обрати Explore (рис. 4.4).

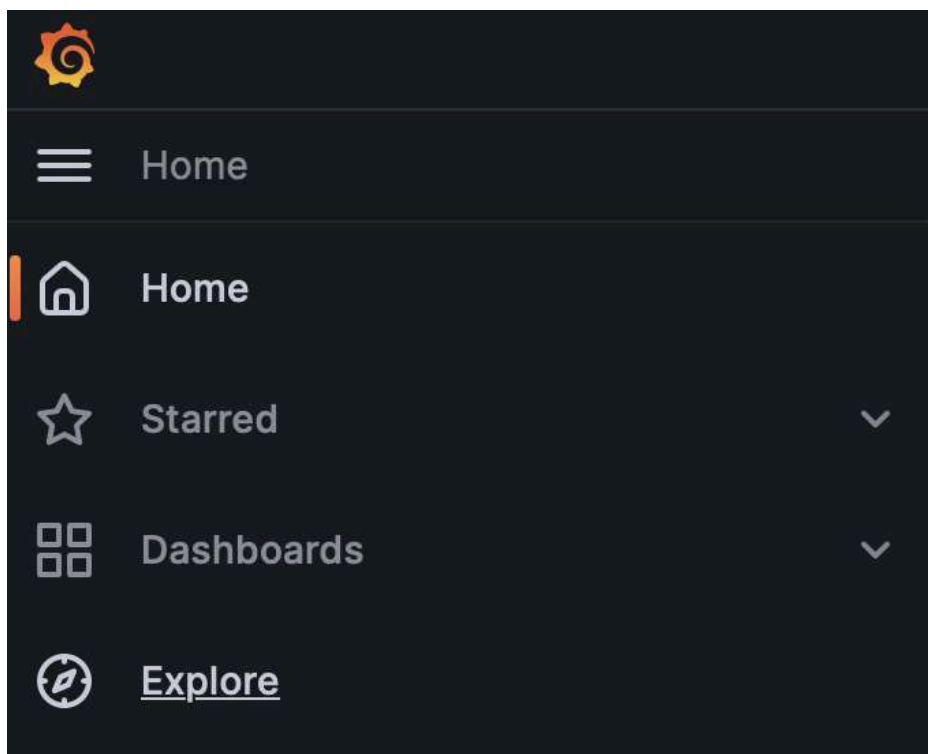


Рисунок 4.4 – Місцезнаходження посилання для переходу на сторінку Explore

Відкривши сторінку Explore, користувач може переглядати зібрані журнали, метрики та трасування. Для перегляду журналів, користувач має обрати Loki як датасорс і вказати запит, як це показано на рисунках 4.5 і 4.6. Після чого користувачу буде виведено список журналів знайдених по його запиту, як це показано на рис. 4.7 і 4.8.

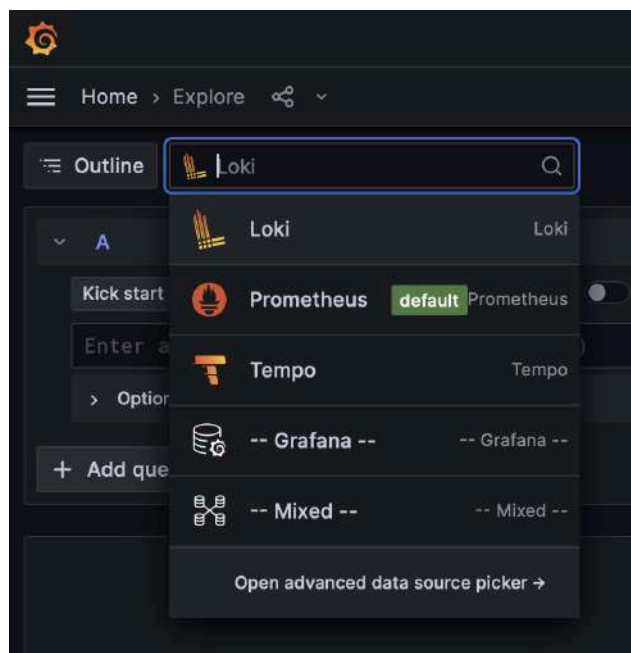


Рисунок 4.5 – Вибір Loki як датасорсу

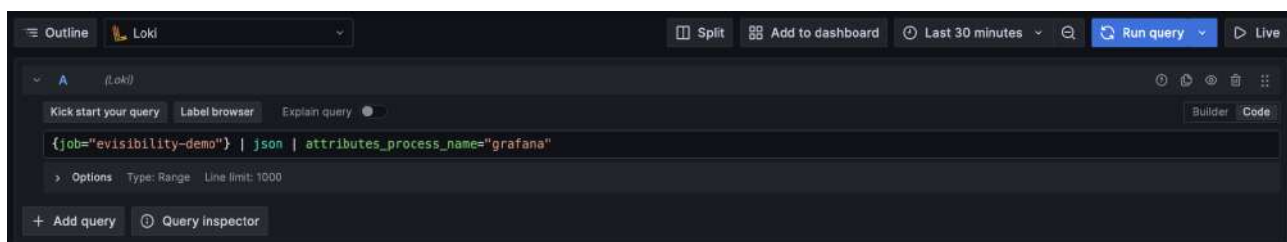


Рисунок 4.6 – Фільтрування логів

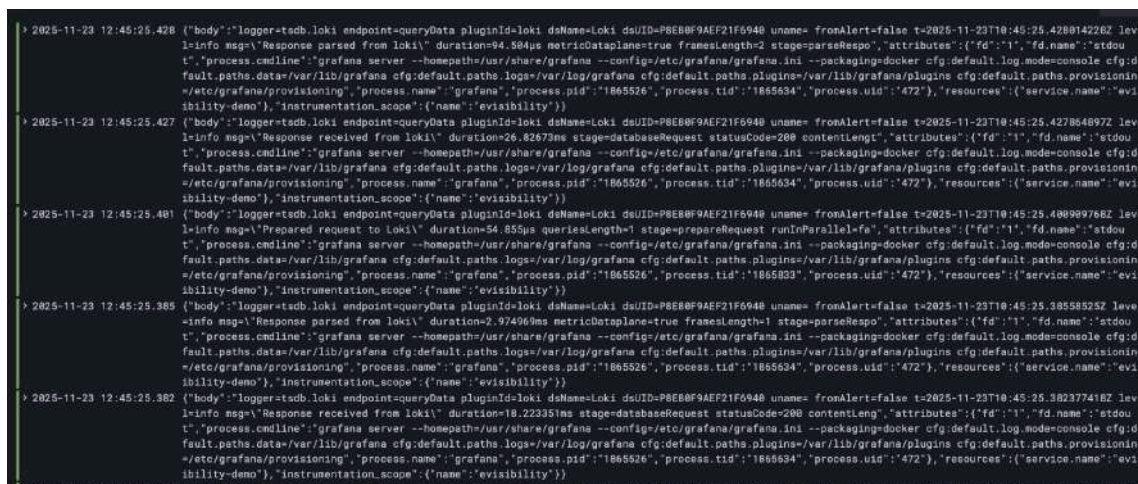


Рисунок 4.7 – Знайдені журнали по запиту користувача

Fields		
attributes_fd	1	
attributes_fd_name	stdout	
attributes_process_cmdline	grafana server --homepath=/usr/share/grafana --config=/etc/grafana/grafana.ini --packaging=docker cfg:default.log.mode=console cfg:default.paths.data=/var/lib/grafana cfg:default.paths.logs=/var/log/grafana cfg:default.paths.plugins=/var/lib/grafana/plugins cfg:default.paths.provisioning=/etc/grafana/provisioning	
attributes_process_name	grafana	
attributes_process_pid	1865526	
attributes_process_tid	1865662	
attributes_process_uid	472	
body	logger=tsdb.loki endpoint=callResource pluginId=loki dsName=Loki dsUID=P8E80F9AEF21F6940 uname= t=2025-11-23T10:45:25.3399558 75Z level=info msg="Response received from loki" status=ok statusCode=200 contentLength=55 duration=4.112174ms	
exporter	OTLP	
instrumentation_scope_name	evisibility	
job	evisibility-demo	
level	INFO	
resources_service_name	evisibility-demo	

Рисунок 4.8 – Додаткові поля прикріплені до одного логу

Аби переглянути зібрані метрики, користувач має змінити датасорс на Prometheus і також вказати запит (рис. 4.9). Для прикладу, зробимо запит по метриці, що показує кількість перезапусків процесів зі старту колектора (рис. 4.10, 4.11).

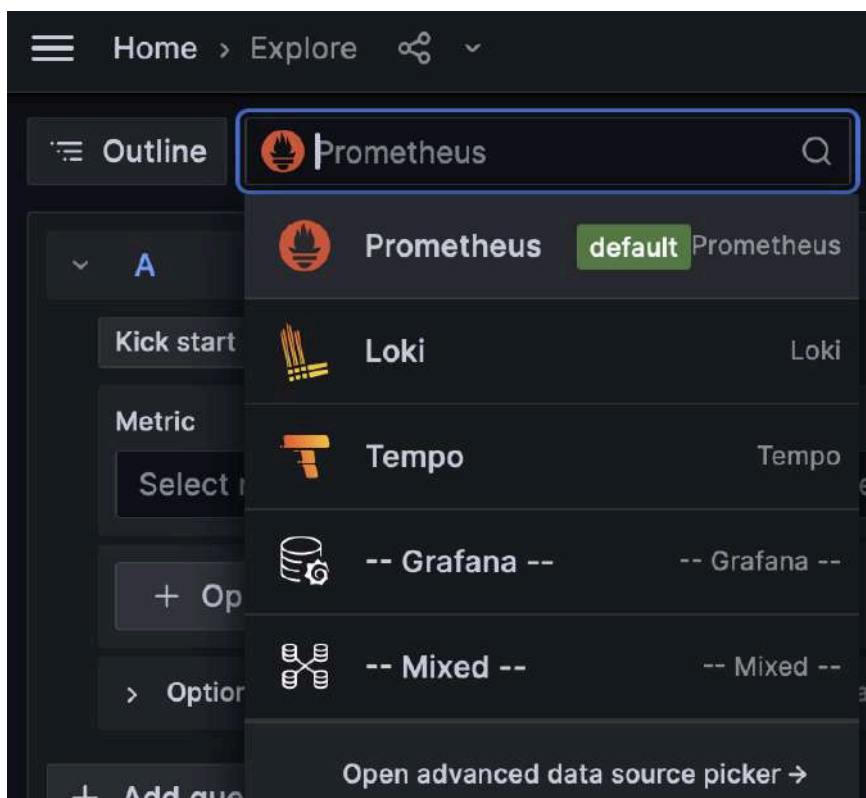


Рисунок 4.9 – Вибір Prometheus як датасорсу

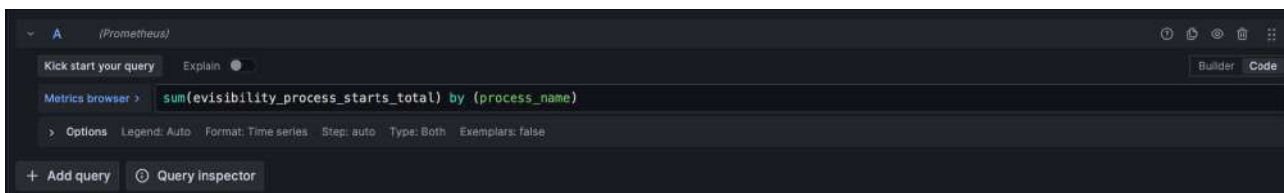


Рисунок 4.10 – Створення запиту на отримання значень метрики



Рисунок 4.11 – Значення метрики по кожному процесу в системі

Також користувач може знайти значення метрики по конкретному процесу, в якості прикладу оберемо nginx (рис 4.12). Після декількох перезапусків nginx, для прикладу, на рис. 4.13 можемо бачити зміну в значенні метрики.

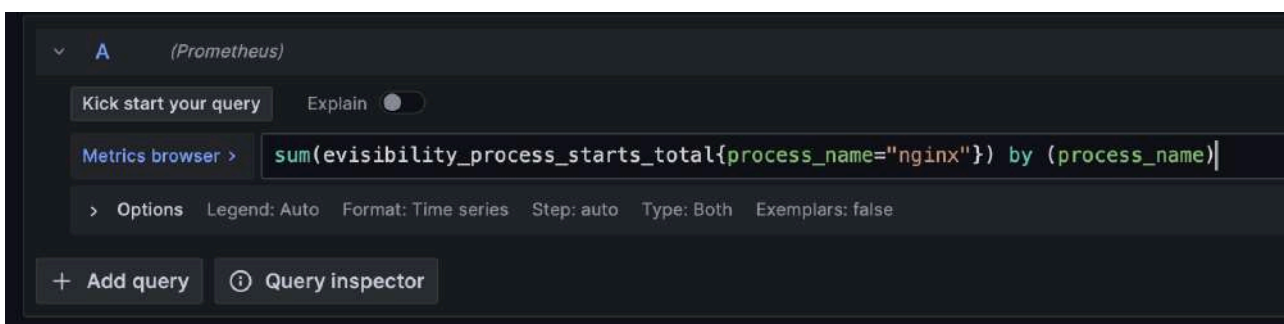


Рисунок 4.12 – Створення запиту на отримання значень метрики тільки по процесу nginx



Рисунок 4.13 – Значення метрики тільки по процесу nginx

Так як колектор трасувань налаштований на збір даних тільки з процесів з назвою nginx, користувач може зробити декілька HTTP викликів на вебсервер аби викликати активність на nginx процесі. Після цього, користувач має змінити датасорс на Tempo, вказати запит (рис. 4.14), і після цього Grafana відобразить записані проміжки по викликам функцій в nginx (рис. 4.15, 4.16, 4.17, 4.18).

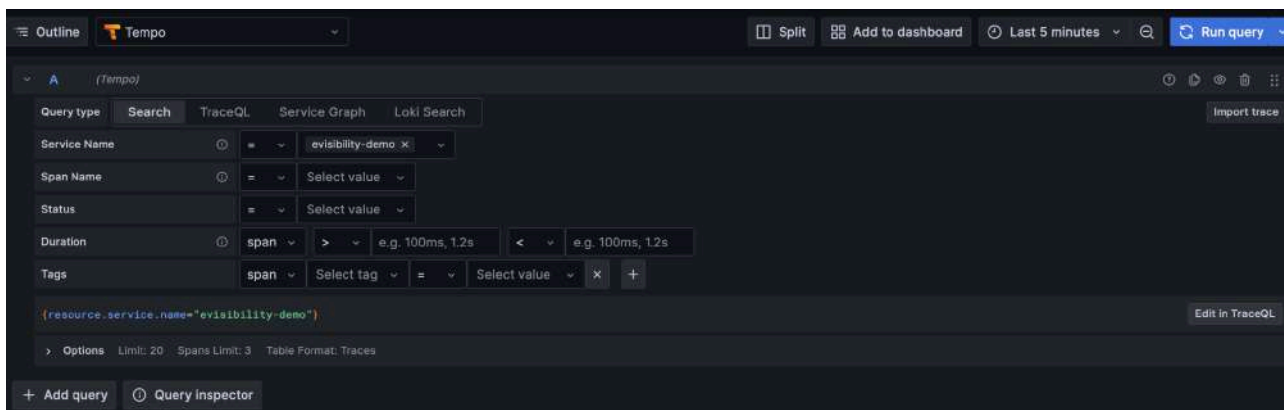


Рисунок 4.14 – Створення запиту на отримання даних трасувань

Table - Traces				
	Trace ID	Start time	Service	Duration
>	1d3960187aa8c8004ac6...	2025-11-23 16:23:00	<root span not yet received>	6 ms
>	1d3960187aa89aacf11c7...	2025-11-23 16:19:46	<root span not yet received>	10 ms

Рисунок 4.15 – Знайдені дані трасувань

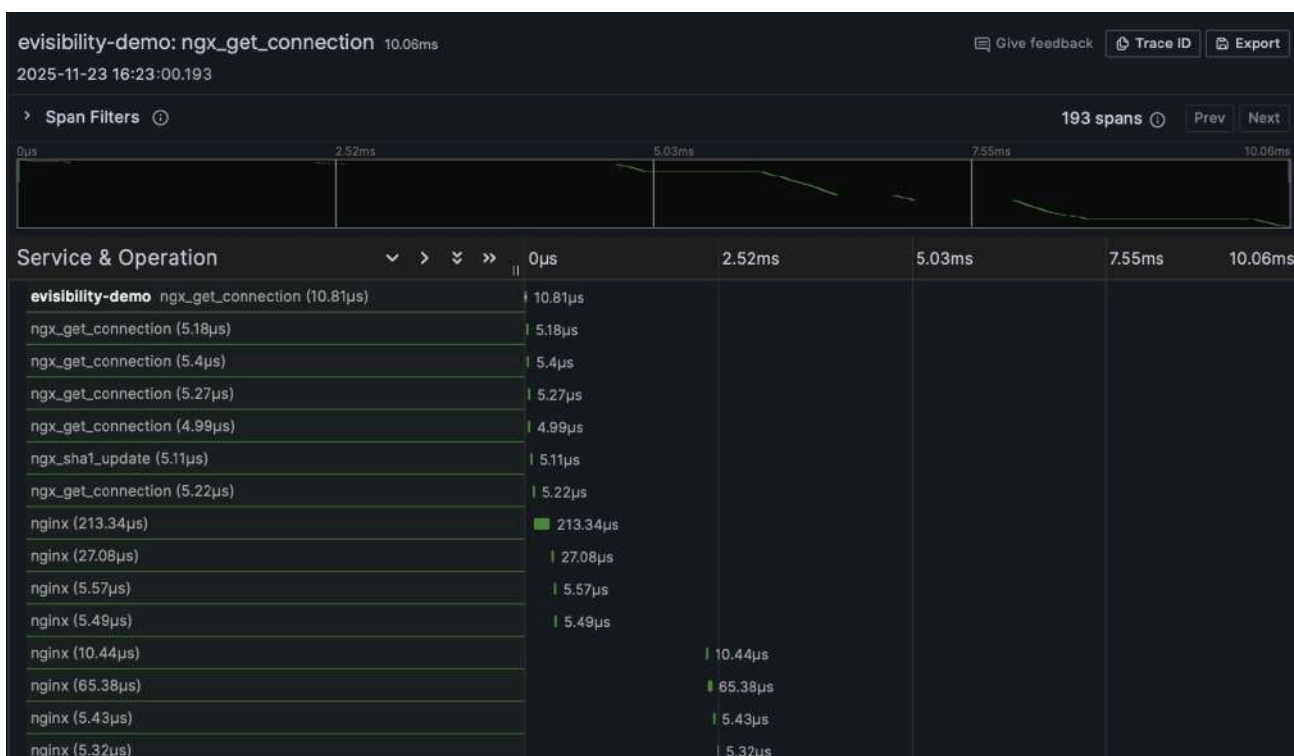


Рисунок 4.16 – Візуалізація викликів функцій для обробки запиту в nginx

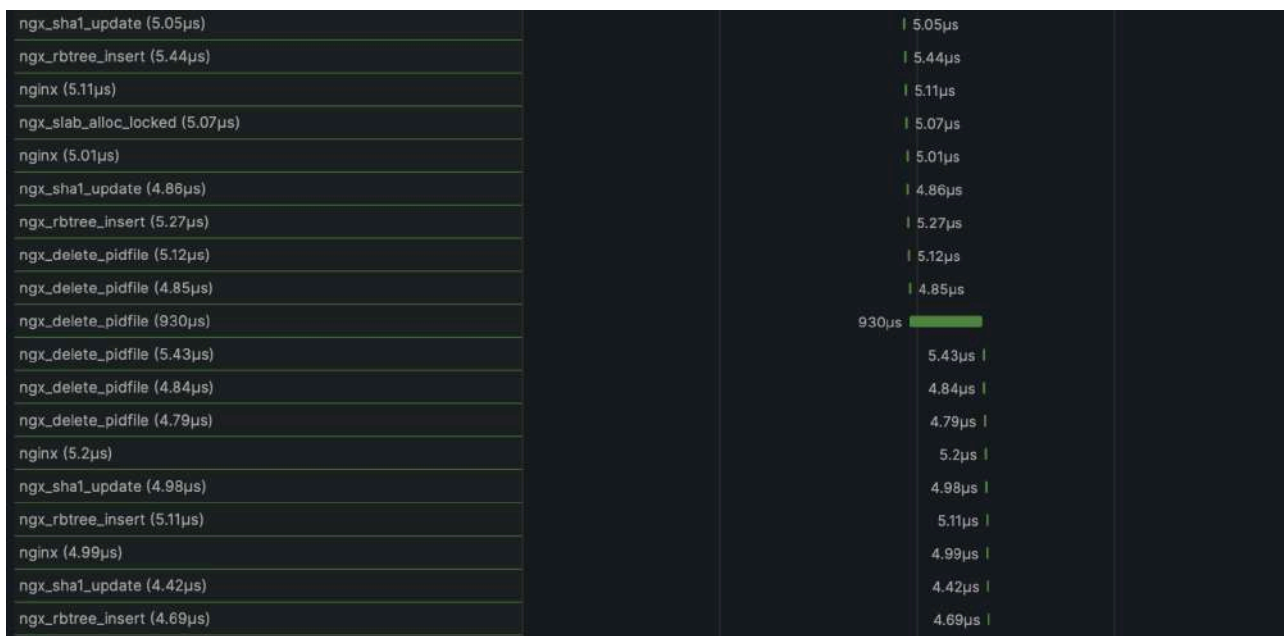


Рисунок 4.17 – Візуалізація викликів функцій для обробки запиту в nginx

4.3.3 Повідомлення

Серед основних типів повідомлень системи до користувача можна виділити наступні:

- повідомлення про запуск додатка: містять інформацію про початок роботи програми та надану конфігурацію;
- повідомлення про завантаження eBPF програм: інформують про успішне або неуспішне завантаження програм у ядро Linux та початок збору даних;
- відлагоджувальні повідомлення (debug): надають детальну інформацію про процес збору даних; можуть бути увімкнені окремо для діагностики та налагодження системи.

4.4 Тестування програмного забезпечення

На етапі тестування програмного продукту було проведено декілька видів перевірок для оцінки його стабільності, коректності роботи та відповідності заданим вимогам. Зокрема, застосовувалися такі типи тестування:

- модульне тестування – передбачає перевірку окремих функцій і класів з різними вхідними даними з метою підтвердження правильності їх роботи;
- інтеграційне тестування – охоплює тестування взаємодії кількох компонентів системи з використанням реальних даних;
- end-to-end тестування – дозволяє перевірити повний цикл роботи програмного продукту з точки зору користувача;
- тестування продуктивності – дозволяє перевірити наскільки ефективно додаток працює під високим навантаженням.

У процесі тестування перевірялися основні функціональні елементи системи, зокрема обробка даних із ядра Linux, валідація конфігураційних

параметрів (наприклад, фільтрів за назвами процесів) та інші критично важливі компоненти.

У випадку некоректної конфігурації додаток формує повідомлення про помилку в журналі подій із детальним описом проблеми. Якщо виявляється помилка, пов'язана з доступами, система також надає пояснення та рекомендації щодо її усунення. Всі повідомлення про помилки мають спеціальне форматування, що полегшує їх сприйняття під час аналізу журналів.

Під час тестування не було виявлено системних збоїв, помилок доступу або проблем із продуктивністю. Отже, можна зробити висновок, що розроблений програмний продукт працює стабільно, відповідає встановленим вимогам і є готовим до практичного використання.

4.5 Висновки за розділом 4

У цьому розділі розглянуто призначення та функціональні можливості розробленого програмного додатка для збору моніторингових даних у системах на базі ядра Linux. Визначено апаратні та програмні вимоги, необхідні для його встановлення та стабільної роботи у фізичних, віртуалізованих або контейнеризованих середовищах, зокрема у Kubernetes. Наведено опис принципів взаємодії користувача з додатком, типів вхідних і вихідних даних, а також повідомлень, які формує система під час роботи. Проведено тестування основних компонентів і процесів збору даних, за результатами якого підтверджено, що додаток функціонує коректно, відповідає визначеним вимогам і готовий до практичного використання.

ВИСНОВКИ

В ході виконання дипломної кваліфікаційної роботи магістра було проаналізовано та досліджено предметну область, програмні засоби, інструменти та технології для моніторингу систем на основі Linux.

Досліджено та проаналізовано найпоширеніші програмні рішення, технології та підходи для моніторингу систем на основі Linux. На основі проведеного аналізу визначено ключові функціональні вимоги до розроблюваного програмного забезпечення для моніторингу Linux-систем.

Проведено дослідження існуючих мов програмування, технологій для розробки моніторингового програмного забезпечення та підходів для збору моніторингових даних. Обрано наступні технології: eBPF – як інструмент для безпечного та ефективного перехоплення даних на рівні ядра; Go – для реалізації користувацької частини; C – для написання коду, що виконується в ядрі Linux; OpenTelemetry (OTel) – для експорту метрик, логів та даних трасування в єдиному форматі незалежно від програмного продукту для відображення та візуалізації моніторингових даних.

Було розроблено програмний додаток, який надає користувачам зручні та ефективні інструменти для збору моніторингових даних, зокрема метрик, журналів подій та даних трасування. Система забезпечує можливість отримання цих даних із будь-яких середовищ, що функціонують на основі ядра Linux, що робить її універсальним рішенням для різних типів інфраструктур. Завдяки оптимізованій архітектурі та раціональному використанню системних ресурсів, додаток не створює суттєвих накладних витрат на продуктивність, не впливає на стабільність роботи операційної системи чи інших програмних компонентів, встановлених у середовищі виконання.

У процесі перевірки роботи програмного продукту не було виявлено жодних системних помилок, збоїв чи порушень у функціонуванні окремих компонентів. Також не зафіксовано потенційних загроз безпеці, зокрема щодо несанкціонованого доступу до користувацьких даних, а також проблем з

швидкістю обробки даних та загальною продуктивністю додатку. Завдяки використанню технології eBPF та верифікатору, який є невід’ємною частиною процесу завантаження eBPF програм і має суворі правила щодо завантажуваного коду, забезпечується високий рівень контролю за виконанням програм у ядрі операційної системи, досягається підвищена надійність роботи, унеможлиблюється аварійне завершення системи або програм, що виконуються в програмному середовищі. Таким чином, можна зробити висновок, що розроблений програмний продукт функціонує стабільно, повністю відповідає визначеним технічним та функціональним вимогам і є готовим до подальшого практичного використання.

Інтеграція розробленого програмного продукту в середовище користувачів дозволить DevOps інженерам та SRE спеціалістам своєчасно виявляти потенційні проблемні місця, такі як затримки в обробці запитів, надмірне навантаження на CPU чи інші компоненти системи в застосунках, що працюють у цьому середовищі. У разі виникнення несправності інженер зможе швидко визначити її джерело, яке спричинило простій системи або некоректну роботу. Крім того, якщо користувач виявить повторювану проблему (наприклад, підвищене навантаження на диск), він може переглянути поточну архітектуру своєї системи, щоб оптимізувати її роботу або зменшити витрати на інфраструктуру.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Linux Statistics 2025: Desktop, Server, Cloud & Community Trends [Electronic resource]. – Access mode: <https://sqmagazine.co.uk/linux-statistics>.
2. Gregg B. Systems Performance (Addison-Wesley Professional Computing Series). 2nd ed. Boston, MA : Longman (Pearson Education), 2020. 928 p.
3. Linux Out-of-Memory Killer [Electronic resource]. – Access mode: <https://docs.rackspace.com/docs/linux-out-of-memory-killer>.
4. Linux Admin Guide. Concepts. [Electronic resource]. – Access mode: <https://docs.kernel.org/admin-guide/mm/concepts.html>.
5. Gregg B. BPF Performance Tools: Linux System and Application Observability. Boston, MA : Addison-Wesley Professional, 2019. 880 p.
6. Kirch O. Linux Network Administrator's Guide. Sebastopol : O'Reilly & Associates, Incorporated, 1995. 370 p.
7. Deep packet inspection [Electronic resource]. – Access mode: https://en.wikipedia.org/wiki/Deep_packet_inspection.
8. What Are Some Indicators Of Privilege Escalation In System Logs? [Electronic resource]. – Access mode: <https://www.cyberly.org/en/what-are-some-indicators-of-privilege-escalation-in-system-logs/index.html>.
9. Linux Logs Explained [Electronic resource]. – Access mode: <https://www.plesk.com/blog/product-technology/linux-logs-explained>.
10. Common logs locations in Linux [Electronic resource]. – Access mode: <https://www.iblue.team/linux-forensics/linux>.
11. AppArmor [Electronic resource]. – Access mode: <https://en.wikipedia.org/wiki/AppArmor>.
12. Linux audit: Log files in /var/log/audit [Electronic resource]. – Access mode: <https://linux-audit.com/linux-audit-log-files-in-var-log-audit>.
13. Cilium (computing) [Electronic resource]. – Access mode:

[https://en.wikipedia.org/wiki/Cilium_\(computing\)](https://en.wikipedia.org/wiki/Cilium_(computing)).

14. Monitoring using Pixie [Electronic resource]. – Access mode: https://con204.github.io/intermediate/241_pixie.

15. Falcosidekick-ui [Electronic resource]. – Access mode: <https://github.com/falcosecurity/falcosidekick-ui>.

16. Parca Demo [Electronic resource]. – Access mode: <https://demo.parca.dev>.

17. The Go Programming Language [Electronic resource]. – Access mode: <https://go.dev>.

18. BCC Repository [Electronic resource]. – Access mode: <https://github.com/iovisor/bcc>.

19. eBPF Docs [Electronic resource]. – Access mode: <https://docs.ebpf.io>.

20. What is OpenTelemetry [Electronic resource]. – Access mode: <https://opentelemetry.io/docs/what-is-opentelemetry>.

21. Visual Studio Code Documentation [Electronic resource]. – Access mode: <https://code.visualstudio.com/docs>.

22. Cyclomatic complexity [Electronic resource]. – Access mode: https://en.wikipedia.org/wiki/Cyclomatic_complexity.

23. Lizard [Electronic resource]. – Access mode: <https://github.com/terryyin/lizard>.

24. eBPF Verifier Complexity [Electronic resource]. – Access mode: <https://pchaigno.github.io/ebpf/2019/07/02/bpf-verifier-complexity.html>.

25. Helper functions in eBPF [Electronic resource]. – Access mode: <https://docs.ebpf.io/linux/helper-function>.

26. What is eBPF [Electronic resource]. – Access mode: <https://ebpf.io/what-is-ebpf/#what-is-ebpf>.

27. Kubernetes Workload Management [Electronic resource]. – Access mode: <https://kubernetes.io/docs/concepts/workloads/controllers>.

28. AWS Fargate. Serverless compute for containers [Electronic resource]. – Access mode: <https://aws.amazon.com/fargate>.

29. OpenTelemetry Docs [Electronic resource]. – Access mode: <https://opentelemetry.io/docs>.
30. OTLP Specification [Electronic resource]. – Access mode: <https://opentelemetry.io/docs/specs/otlp/#protocol-details>.
31. Experiment: Migrating OpenTracing-based application in Go to use the OpenTelemetry SDK [Electronic resource]. – Access mode: <https://medium.com/jaegertracing/experiment-migrating-opentracing-based-application-in-go-to-use-the-opentelemetry-sdk-29b09fe2fbc4>.
32. What is a log file? [Electronic resource]. – Access mode: <https://www.elastic.co/what-is/log-files>.
33. Rice L. Learning eBPF: Programming the Linux Kernel for Enhanced Observability, Networking, and Security. Sebastopol : O'Reilly Media, 2023. 234 p.
34. Lim, S. Y., Prasad, T., Han, X., Pasquier, T. SafeBPF: Hardware-assisted Defense-in-depth for eBPF Kernel Extensions. 2024. arXiv: 2409.07508.
35. Fournier, G., Afchain, S., Baubeau, S. Runtime Security Monitoring with eBPF. SSTIC 2021. URL: https://www.sstic.org/media/SSTIC2021/SSTIC-actes/runtime_security_with_ebpf/SSTIC2021-Article-runtime_security_with_ebpf-fournier_afchain_baubeau.pdf.
36. Cassagnes, C. The rise of eBPF for non-intrusive performance monitoring. In: Proceedings of the 17th USENIX Symposium on Operating Systems Design and Implementation (OSDI'20), 2020.
37. Gbadamosi, B., Leonardi, L., Pulls, T., Høiland-Jørgensen, T., Ferlin-Reiter, S., Sorce, S., Brunström, A. The eBPF Runtime in the Linux Kernel. 2024. arXiv: 2410.00026.
38. Psomadakis S., Mores K., Goumas G. Userspace-guided memory management in Linux with eBPF. 2024. URL: <https://arxiv.org/abs/2409.11220>.
39. Runtime Verification for Linux Kernel Based on eBPF. In: 8th Int. Conf. on Computer Information Science and Application Technology / L. Yang et al. 8th International Conference on Computer Information Science and Application Technology (CISAT), 11 July 2025. URL:

https://www.researchgate.net/publication/396088360_Runtime_Verification_for_Linux_Kernel_Based_on_eBPF.

40. Bachl, M., Fabini, J., Zseby, T. A flow-based IDS using Machine Learning in eBPF. Vienna, 2022. URL: <https://arxiv.org/abs/2102.09980>.

41. eBPF-Based Extensions of Linux Hosts for Using Virtual Networks / M. Haeberle et al. Computer Networks. 2025. Vol. 273.

ДОДАТОК А
Текст програми

A.1 pkg/otel/metrics.go

```

package otel

import (
    "context"
    "fmt"
    "time"

    "github.com/user/evisibility/pkg/exporter"
    "go.opentelemetry.io/otel/attribute"
    "go.opentelemetry.io/otel/exporters/otlp/otlpmetric/otlpmetricgrpc"
    "go.opentelemetry.io/otel/metric"
    sdkmetric "go.opentelemetry.io/otel/sdk/metric"
    "go.opentelemetry.io/otel/sdk/metric/metricdata"
    "go.opentelemetry.io/otel/sdk/resource"
    semconv "go.opentelemetry.io/otel/semconv/v1.24.0"
)

type MetricsExporter struct {
    provider *sdkmetric.MeterProvider
    meter    metric.Meter
    counters map[string]metric.Float64Counter
    gauges   map[string]metric.Float64Gauge
    histograms map[string]metric.Float64Histogram
}

func NewMetricsExporter(ctx context.Context, endpoint string, serviceName string) (*MetricsExporter, error) {
    otlpExporter, err := otlpmetricgrpc.New(ctx,
        otlpmetricgrpc.WithEndpoint(endpoint),
        otlpmetricgrpc.WithInsecure(),
        otlpmetricgrpc.WithTemporalitySelector(func(ik sdkmetric.InstrumentKind) metricdata.Temporality {
            return metricdata.DeltaTemporality
        })),
    )
    if err != nil {
        return nil, fmt.Errorf("failed to create OTLP exporter: %w", err)
    }

    res, err := resource.New(ctx,
        resource.WithAttributes(
            semconv.ServiceName(serviceName),
        ),
    )
    if err != nil {
        return nil, fmt.Errorf("failed to create resource: %w", err)
    }

    provider := sdkmetric.NewMeterProvider(
        sdkmetric.WithReader(sdkmetric.NewPeriodicReader(
            otlpExporter,
            sdkmetric.WithInterval(10*time.Second),
        )),
        sdkmetric.WithResource(res),
    )

    return &MetricsExporter{
        provider: provider,
        meter:    provider.Meter("evisibility"),
        counters: make(map[string]metric.Float64Counter),
        gauges:   make(map[string]metric.Float64Gauge),
    }
}

```

```

        histograms: make(map[string]metric.Float64Histogram),
    }, nil
}

func (e *MetricsExporter) ExportMetrics(ctx context.Context, metrics []exporter.Metric) error {
    for _, m := range metrics {
        attrs := make([]attribute.KeyValue, 0, len(m.Labels))
        for k, v := range m.Labels {
            attrs = append(attrs, attribute.String(k, v))
        }
        opt := metric.WithAttributeSet(attribute.NewSet(attrs...))

        switch m.Type {
        case exporter.MetricTypeCounter:
            counter, err := e.getOrCreateCounter(m.Name)
            if err != nil {
                return fmt.Errorf("failed to get counter: %w", err)
            }
            counter.Add(ctx, m.Value, opt)

        case exporter.MetricTypeGauge:
            gauge, err := e.getOrCreateGauge(m.Name)
            if err != nil {
                return fmt.Errorf("failed to get gauge: %w", err)
            }
            gauge.Record(ctx, m.Value, opt)

        case exporter.MetricTypeHistogram:
            histogram, err := e.getOrCreateHistogram(m.Name)
            if err != nil {
                return fmt.Errorf("failed to get histogram: %w", err)
            }
            histogram.Record(ctx, m.Value, opt)
        }
    }

    return nil
}

func (e *MetricsExporter) Shutdown(ctx context.Context) error {
    if err := e.provider.Shutdown(ctx); err != nil {
        return fmt.Errorf("failed to shutdown meter provider: %w", err)
    }
    return nil
}

func (e *MetricsExporter) getOrCreateCounter(name string) (metric.Float64Counter, error) {
    if counter, exists := e.counters[name]; exists {
        return counter, nil
    }

    counter, err := e.meter.Float64Counter(name)
    if err != nil {
        return nil, err
    }
    e.counters[name] = counter
    return counter, nil
}

func (e *MetricsExporter) getOrCreateHistogram(name string) (metric.Float64Histogram, error) {
    if histogram, exists := e.histograms[name]; exists {
        return histogram, nil
    }
}

```

```

        histogram, err := e.meter.Float64Histogram(name)
        if err != nil {
            return nil, err
        }
        e.histograms[name] = histogram
        return histogram, nil
    }

    func (e *MetricsExporter) getOrCreateGauge(name string) (metric.Float64Gauge, error) {
        if gauge, exists := e.gauges[name]; exists {
            return gauge, nil
        }

        gauge, err := e.meter.Float64Gauge(name)
        if err != nil {
            return nil, err
        }
        e.gauges[name] = gauge
        return gauge, nil
    }

```

A.2 pkg/otel/logs.go

```

package otel

import (
    "context"
    "fmt"

    "github.com/user/evisibility/pkg/exporter"
    "go.opentelemetry.io/otel/exporters/otlp/otlplog/otlploggrpc"
    "go.opentelemetry.io/otel/log"
    sdklog "go.opentelemetry.io/otel/sdk/log"
    "go.opentelemetry.io/otel/sdk/resource"
    semconv "go.opentelemetry.io/otel/semconv/v1.24.0"
)

type LogsExporter struct {
    provider *sdklog.LoggerProvider
    logger   log.Logger
}

func NewLogsExporter(ctx context.Context, endpoint string, serviceName string) (*LogsExporter, error) {
    otlpExporter, err := otlploggrpc.New(ctx,
        otlploggrpc.WithEndpoint(endpoint),
        otlploggrpc.WithInsecure(),
    )
    if err != nil {
        return nil, fmt.Errorf("failed to create OTLP logs exporter: %w", err)
    }

    res, err := resource.New(ctx,
        resource.WithAttributes(
            semconv.ServiceName(serviceName),
        ),
    )
    if err != nil {
        return nil, fmt.Errorf("failed to create resource: %w", err)
    }

```

```

    provider := sdklog.NewLoggerProvider(
        sdklog.WithProcessor(sdklog.NewBatchProcessor(otlpExporter)),
        sdklog.WithResource(res),
    )

    logger := provider.Logger("evisibility")

    return &LogsExporter{
        provider: provider,
        logger: logger,
    }, nil
}

func (e *LogsExporter) ExportLogs(ctx context.Context, logs []exporter.LogEntry) error {
    for _, logEntry := range logs {
        attrs := make([]log.KeyValue, 0, len(logEntry.Attributes))
        for k, v := range logEntry.Attributes {
            attrs = append(attrs, log.KeyValue{
                Key: k,
                Value: convertToLogValue(v),
            })
        }

        var record log.Record
        record.SetTimestamp(logEntry.Timestamp)
        record.SetBody(log.StringValue(logEntry.Message))
        record.SetSeverity(convertLogLevel(logEntry.Level))
        record.AddAttributes(attrs...)

        e.logger.Emit(ctx, record)
    }

    return nil
}

func (e *LogsExporter) Shutdown(ctx context.Context) error {
    if err := e.provider.Shutdown(ctx); err != nil {
        return fmt.Errorf("failed to shutdown logger provider: %w", err)
    }
    return nil
}

func convertLogLevel(level exporter.LogLevel) log.Severity {
    switch level {
    case exporter.LogLevelDebug:
        return log.SeverityDebug
    case exporter.LogLevelInfo:
        return log.SeverityInfo
    case exporter.LogLevelWarn:
        return log.SeverityWarn
    case exporter.LogLevelError:
        return log.SeverityError
    case exporter.LogLevelFatal:
        return log.SeverityFatal
    default:
        return log.SeverityInfo
    }
}

func convertToLogValue(v any) log.Value {
    switch val := v.(type) {
    case string:

```

```

        return log.StringValue(val)
    case int:
        return log.Int64Value(int64(val))
    case int64:
        return log.Int64Value(val)
    case float64:
        return log.Float64Value(val)
    case bool:
        return log.BoolValue(val)
    case []any:
        values := make([]log.Value, len(val))
        for i, item := range val {
            values[i] = convertToLogValue(item)
        }
        return log.SliceValue(values...)
    case map[string]any:
        kvs := make([]log.KeyValue, 0, len(val))
        for k, v := range val {
            kvs = append(kvs, log.KeyValue{
                Key: k,
                Value: convertToLogValue(v),
            })
        }
        return log.MapValue(kvs...)
    default:
        return log.StringValue(fmt.Sprintf("%v", v))
}

```

A.3 pkg/otel/traces.go

```

package otel

import (
    "context"
    "fmt"

    "github.com/user/evisibility/pkg/exporter"
    "go.opentelemetry.io/otel/attribute"
    "go.opentelemetry.io/otel/codes"
    "go.opentelemetry.io/otel/exporters/otlp/otlptrace/otlptracegrpc"
    sdktrace "go.opentelemetry.io/otel/sdk/trace"
    "go.opentelemetry.io/otel/sdk/resource"
    semconv "go.opentelemetry.io/otel/semconv/v1.24.0"
    "go.opentelemetry.io/otel/trace"
)

type TracesExporter struct {
    provider *sdktrace.TracerProvider
    tracer   trace.Tracer
}

func NewTracesExporter(ctx context.Context, endpoint string, serviceName string) (*TracesExporter, error) {
    otlpExporter, err := otlptracegrpc.New(ctx,
        otlptracegrpc.WithEndpoint(endpoint),
        otlptracegrpc.WithInsecure(),
    )
    if err != nil {
        return nil, fmt.Errorf("failed to create OTLP trace exporter: %w", err)
    }
}

```

```

    res, err := resource.New(ctx,
        resource.WithAttributes(
            semconv.ServiceName(serviceName),
        ),
    )
    if err != nil {
        return nil, fmt.Errorf("failed to create resource: %w", err)
    }

    provider := sdktrace.NewTracerProvider(
        sdktrace.WithBatcher(otlpExporter),
        sdktrace.WithResource(res),
    )

    tracer := provider.Tracer("evisibility")

    return &TracesExporter{
        provider: provider,
        tracer:   tracer,
    }, nil
}

func (e *TracesExporter) ExportSpans(ctx context.Context, spans []exporter.Span) error {
    for _, span := range spans {
        traceID, err := trace.TraceIDFromHex(span.TraceID)
        if err != nil {
            return fmt.Errorf("invalid trace ID %s: %w", span.TraceID, err)
        }

        spanID, err := trace.SpanIDFromHex(span.SpanID)
        if err != nil {
            return fmt.Errorf("invalid span ID %s: %w", span.SpanID, err)
        }

        spanContext := trace.NewSpanContext(trace.SpanContextConfig{
            TraceID:   traceID,
            SpanID:    spanID,
            TraceFlags: trace.FlagsSampled,
        })

        attrs := make([]attribute.KeyValue, 0, len(span.Attributes))
        for k, v := range span.Attributes {
            attrs = append(attrs, convertToAttribute(k, v))
        }

        _, otelSpan := e.tracer.Start(
            trace.ContextWithSpanContext(ctx, spanContext),
            span.Name,
            trace.WithTimestamp(span.StartTime),
            trace.WithSpanKind(convertSpanKind(span.Kind)),
            trace.WithAttributes(attrs...),
        )

        for _, event := range span.Events {
            eventAttrs := make([]attribute.KeyValue, 0, len(event.Attributes))
            for k, v := range event.Attributes {
                eventAttrs = append(eventAttrs, convertToAttribute(k, v))
            }
            otelSpan.AddEvent(event.Name,
                trace.WithTimestamp(event.Timestamp),
                trace.WithAttributes(eventAttrs...),
            )
        }
    }
}

```

```

    }

    switch span.Status.Code {
    case exporter.SpanStatusOK:
        otelSpan.SetStatus(codes.Ok, span.Status.Message)
    case exporter.SpanStatusError:
        otelSpan.SetStatus(codes.Error, span.Status.Message)
    default:
        otelSpan.SetStatus(codes.Unset, span.Status.Message)
    }

    otelSpan.End(trace.WithTimestamp(span.EndTime))
}

return nil
}

func (e *TracesExporter) Shutdown(ctx context.Context) error {
    if err := e.provider.Shutdown(ctx); err != nil {
        return fmt.Errorf("failed to shutdown tracer provider: %w", err)
    }
    return nil
}

func convertSpanKind(kind exporter.SpanKind) trace.SpanKind {
    switch kind {
    case exporter.SpanKindInternal:
        return trace.SpanKindInternal
    case exporter.SpanKindServer:
        return trace.SpanKindServer
    case exporter.SpanKindClient:
        return trace.SpanKindClient
    case exporter.SpanKindProducer:
        return trace.SpanKindProducer
    case exporter.SpanKindConsumer:
        return trace.SpanKindConsumer
    default:
        return trace.SpanKindInternal
    }
}

func convertToAttribute(key string, value any) attribute.KeyValue {
    switch v := value.(type) {
    case string:
        return attribute.String(key, v)
    case int:
        return attribute.Int(key, v)
    case int64:
        return attribute.Int64(key, v)
    case float64:
        return attribute.Float64(key, v)
    case bool:
        return attribute.Bool(key, v)
    case []string:
        return attribute.StringSlice(key, v)
    case []int:
        return attribute.IntSlice(key, v)
    case []int64:
        return attribute.Int64Slice(key, v)
    case []float64:
        return attribute.Float64Slice(key, v)
    case []bool:
        return attribute.BoolSlice(key, v)
    }
}

```

```

        default:
            return attribute.String(key, fmt.Sprintf("%v", v))
    }
}

```

A.4 pkg/ebpf/logs.go

```

package ebpf

import (
    "bytes"
    "context"
    "encoding/binary"
    "errors"
    "fmt"
    "os"
    "path/filepath"
    "strings"
    "time"

    "github.com/cilium/ebpf/link"
    "github.com/cilium/ebpf/ringbuf"
    "github.com/user/evisibility/pkg/exporter"
)

//go:generate go run github.com/cilium/ebpf/cmd/bpf2go -cc clang -cflags "-O2 -g -Wall -Werror" bpf ../bpf/logs.bpf.c
//go:generate go run github.com/cilium/ebpf/cmd/bpf2go -cc clang -cflags "-O2 -g -Wall -Werror" metrics
//go:generate go run github.com/cilium/ebpf/cmd/bpf2go -cc clang -cflags "-O2 -g -Wall -Werror" traces
//go:generate go run github.com/cilium/ebpf/cmd/bpf2go -cc clang -cflags "-O2 -g -Wall -Werror" traces

const (
    logBufSize = 1024
    taskCommLen = 16
)

type logEvent struct {
    Pid uint32
    Tid uint32
    Uid uint32
    Fd  uint32
    Comm [taskCommLen]byte
    Len  uint32
    Buf  [logBufSize]byte
}

type LogsTracer struct {
    objs bpfObjects
    link link.Link
    reader *ringbuf.Reader
    ctx context.Context
    cancel context.CancelFunc
}

func NewLogsTracer() (*LogsTracer, error) {
    objs := bpfObjects{}
    if err := loadBpfObjects(&objs, nil); err != nil {
        return nil, fmt.Errorf("loading objects: %w", err)
    }
}

```

```

tp, err := link.Tracepoint("syscalls", "sys_enter_write", objs.TraceWriteEnter, nil)
if err != nil {
    objs.Close()
    return nil, fmt.Errorf("opening tracepoint: %w", err)
}

rd, err := ringbuf.NewReader(objs.LogEvents)
if err != nil {
    tp.Close()
    objs.Close()
    return nil, fmt.Errorf("opening ringbuf reader: %w", err)
}

ctx, cancel := context.WithCancel(context.Background())

return &LogsTracer{
    objs:  objs,
    link:  tp,
    reader: rd,
    ctx:   ctx,
    cancel: cancel,
}, nil
}

func (t *LogsTracer) Start(logsChan chan<- exporter.LogEntry) error {
    go func() {
        for {
            select {
            case <-t.ctx.Done():
                return
            default:
                record, err := t.reader.Read()
                if err != nil {
                    if errors.Is(err, ringbuf.ErrClosed) {
                        return
                    }
                }
                continue

                var event logEvent
                if err := binary.Read(bytes.NewBuffer(record.RawSample), binary.LittleEndian,
&event); err != nil {
                    continue
                }

                logEntry := t.eventToLogEntry(&event)

                select {
                case logsChan <- logEntry:
                default:
                }
            }
        }
    }()

    return nil
}

func (t *LogsTracer) eventToLogEntry(event *logEvent) exporter.LogEntry {
    comm := string(bytes.TrimRight(event.Comm[:], "\x00"))

    msgLen := event.Len

```

```

    if msgLen > logBufSize {
        msgLen = logBufSize
    }
    message := string(bytes.TrimRight(event.Buf[:msgLen], "\x00"))
    message = strings.TrimSpace(message)

    level := exporter.LogLevelInfo
    if event.Fd == 2 {
        level = exporter.LogLevelError
    }

    return exporter.LogEntry{
        Timestamp: time.Now().UTC(),
        Level:     level,
        Message:   message,
        Attributes: map[string]any{
            "process.name": comm,
            "process.pid":  event.Pid,
            "process.tid":  event.Tid,
            "process.uid":  event.Uid,
            "process.comm":  getProcessComm(event.Pid),
            "fd":            event.Fd,
            "fd.name":       fdName(event.Fd),
        },
    }
}

func fdName(fd uint32) string {
    switch fd {
    case 1:
        return "stdout"
    case 2:
        return "stderr"
    default:
        return fmt.Sprintf("fd:%d", fd)
    }
}

func getProcessComm(pid uint32) string {
    path := filepath.Join("/proc", fmt.Sprintf("%d", pid), "comm")
    data, err := os.ReadFile(path)
    if err != nil {
        return ""
    }

    comm := string(bytes.ReplaceAll(data, []byte{0}, []byte(" ")))
    return strings.TrimSpace(comm)
}

func (t *LogsTracer) Close() error {
    t.cancel()

    var errs []error

    if t.reader != nil {
        if err := t.reader.Close(); err != nil {
            errs = append(errs, fmt.Errorf("closing ringbuf reader: %w", err))
        }
    }

    if t.link != nil {
        if err := t.link.Close(); err != nil {
            errs = append(errs, fmt.Errorf("closing link: %w", err))
        }
    }
}

```

```

    }
}

if err := t.objs.Close(); err != nil {
    errs = append(errs, fmt.Errorf("closing objects: %w", err))
}

if len(errs) > 0 {
    return fmt.Errorf("multiple errors during close: %v", errs)
}

return nil
}

```

A.5 bpf/logs.bpf.c

```

// SPDX-License-Identifier: GPL-2.0 OR BSD-3-Clause
#include "vmlinux.h"
#include <bpf/bpf_helpers.h>
#include <bpf/bpf_tracing.h>
#include <bpf/bpf_core_read.h>

char LICENSE[] SEC("license") = "Dual BSD/GPL";

#define LOG_BUF_SIZE 1024
#define TASK_COMM_LEN 16

struct log_event {
    u32 pid;
    u32 tid;
    u32 uid;
    u32 fd;
    char comm[TASK_COMM_LEN];
    u32 len;
    char buf[LOG_BUF_SIZE];
};

struct {
    __uint(type, BPF_MAP_TYPE_RINGBUF);
    __uint(max_entries, 256 * 1024);
} log_events SEC(".maps");

SEC("tracepoint/syscalls/sys_enter_write")
int trace_write_enter(struct trace_event_raw_sys_enter *ctx) {
    u64 pid_tgid = bpf_get_current_pid_tgid();
    u32 pid = pid_tgid >> 32;
    u32 tid = (u32)pid_tgid;
    u64 uid_gid = bpf_get_current_uid_gid();
    u32 uid = (u32)uid_gid;

    unsigned int fd = (unsigned int)ctx->args[0];

    if (fd != 1 && fd != 2) {
        return 0;
    }

    struct log_event *event;
    event = bpf_ringbuf_reserve(&log_events, sizeof(*event), 0);
    if (!event) {
        return 0;
    }

```

```

    }

    event->pid = pid;
    event->tid = tid;
    event->uid = uid;
    event->fd = fd;

    bpf_get_current_comm(&event->comm, sizeof(event->comm));

    const char *buf = (const char *)ctx->args[1];
    size_t count = (size_t)ctx->args[2];

    if (count > LOG_BUF_SIZE) {
        count = LOG_BUF_SIZE;
    }
    event->len = count;

    if (count > 0) {
        bpf_probe_read_user(&event->buf, count, buf);
    }

    bpf_ringbuf_submit(event, 0);

    return 0;
}

```

A.6 bpf/traces.bpf.c

```

#include "vmlinux.h"
#include <bpf/bpf_helpers.h>
#include <bpf/bpf_tracing.h>
#include <bpf/bpf_core_read.h>

char LICENSE[] SEC("license") = "Dual BSD/GPL";

#define TASK_COMM_LEN 16
#define MAX_STACK_DEPTH 20

struct function_event {
    u64 timestamp;
    u32 pid;
    u32 tid;
    u32 uid;
    u32 cpu_id;
    char comm[TASK_COMM_LEN];
    u64 instruction_pointer;
    u64 kernel_stack_id;
    u64 user_stack_id;
    u64 duration_ns;
    u8 is_entry;
};

struct {
    __uint(type, BPF_MAP_TYPE_STACK_TRACE);
    __uint(max_entries, 10000);
    __uint(key_size, sizeof(u32));
    __uint(value_size, MAX_STACK_DEPTH * sizeof(u64));
} stack_traces SEC(".maps");

struct {

```

```

    __uint(type, BPF_MAP_TYPE_HASH);
    __uint(max_entries, 10240);
    __type(key, u64);
    __type(value, u64);
} function_entries SEC(".maps");

struct {
    __uint(type, BPF_MAP_TYPE_RINGBUF);
    __uint(max_entries, 1024 * 1024);
} function_events SEC(".maps");

static __always_inline int collect_trace_entry(void *ctx) {
    u64 pid_tgid = bpf_get_current_pid_tgid();
    u32 pid = pid_tgid >> 32;
    u32 tid = (u32)pid_tgid;

    if (pid == 0 || tid == 0) {
        return 0;
    }

    u64 uid_gid = bpf_get_current_uid_gid();
    u32 uid = (u32)uid_gid;

    u64 ts = bpf_ktime_get_ns();
    bpf_map_update_elem(&function_entries, &pid_tgid, &ts, BPF_ANY);

    struct pt_regs *regs = (struct pt_regs *)ctx;
    u64 ip = PT_REGS_IP(regs);

    u64 user_stack_id = bpf_get_stackid(ctx, &stack_traces, BPF_F_USER_STACK);

    struct function_event *event;
    event = bpf_ringbuf_reserve(&function_events, sizeof(*event), 0);
    if (!event) {
        return 0;
    }

    event->timestamp = ts;
    event->pid = pid;
    event->tid = tid;
    event->uid = uid;
    event->cpu_id = bpf_get_smp_processor_id();
    event->instruction_pointer = ip;
    event->kernel_stack_id = 0;
    event->user_stack_id = user_stack_id;
    event->duration_ns = 0;
    event->is_entry = 1;

    bpf_get_current_comm(&event->comm, sizeof(event->comm));

    bpf_ringbuf_submit(event, 0);

    return 0;
}

static __always_inline int collect_trace_exit(void *ctx) {
    u64 pid_tgid = bpf_get_current_pid_tgid();
    u32 pid = pid_tgid >> 32;
    u32 tid = (u32)pid_tgid;

    if (pid == 0 || tid == 0) {
        return 0;
    }
}

```

```

u64 *entry_ts = bpf_map_lookup_elem(&function_entries, &pid_tgid);
if (!entry_ts) {
    return 0;
}

u64 exit_ts = bpf_ktime_get_ns();
u64 duration = exit_ts - *entry_ts;

u64 uid_gid = bpf_get_current_uid_gid();
u32 uid = (u32)uid_gid;

struct pt_regs *regs = (struct pt_regs *)ctx;
u64 ip = PT_REGS_IP(regs);

u64 user_stack_id = bpf_get_stackid(ctx, &stack_traces, BPF_F_USER_STACK);

struct function_event *event;
event = bpf_ringbuf_reserve(&function_events, sizeof(*event), 0);
if (!event) {
    goto cleanup;
}

event->timestamp = exit_ts;
event->pid = pid;
event->tid = tid;
event->uid = uid;
event->cpu_id = bpf_get_smp_processor_id();
event->instruction_pointer = ip;
event->kernel_stack_id = 0;
event->user_stack_id = user_stack_id;
event->duration_ns = duration;
event->is_entry = 0;

bpf_get_current_comm(&event->comm, sizeof(event->comm));

bpf_ringbuf_submit(event, 0);

cleanup:
bpf_map_delete_elem(&function_entries, &pid_tgid);
return 0;
}

SEC("uprobe/generic")
int uprobe__generic(struct pt_regs *ctx) {
    return collect_trace_entry(ctx, TRACE_UPROBE);
}

SEC("uretprobe/generic")
int uretprobe__generic(struct pt_regs *ctx) {
    return collect_trace_exit(ctx, TRACE_UPROBE);
}

```

ДОДАТОК Б
Слайди презентації

Національний університет “Запорізька політехніка”
Кафедра програмних засобів

Дослідження на програмна реалізація моніторингу в Linux на основі eBPF у режимі реального часу

Виконав
Студент гр. КНТ-124м

Данило ГАНСЬКИЙ

Керівник
Доцент, к.т.н.

Олександр СТЕПАНЕНКО

Рисунок Б.1 – Слайд 1

Мета, об’єкт та предмет дослідження

Об’єкт дослідження	Процес моніторингу роботи, розроблення та тестування програмного забезпечення операційних системах, побудованих на ядрі Linux.
Предмет дослідження	Методи і засоби збору моніторингових даних про роботу систем на ядрі Linux та додатків у режимі реального часу.
Мета роботи	Створити програмне рішення, яке забезпечує збір моніторингових даних в ОС Linux з використанням технології eBPF, для збільшення їх обсягу та спрощення процесу моніторингу в контейнеризованих середовищах порівняно зі стандартними підходами.

2

Рисунок Б.2 – Слайд 2

Постановка завдання

1. Визначити ключові задачі і області моніторингу в операційних системах, які потенційно можуть стати об'єктом атак зловмисників або призвести до порушення коректної роботи програмних систем;
2. Здійснити порівняльний аналіз технології eBPF та інших наявних засобів і підходів до моніторингу й захисту визначених областей операційних систем;
3. Розробити програмне рішення для подолання зазначених проблем моніторингу та безпеки за допомогою технології eBPF.

3

Рисунок Б.3 – Слайд 3

Порівняння технологій для моніторингу систем на основі ядра Linux

Технологія	Переваги	Недоліки
eBPF	Висока продуктивність, безпека, інтеграція з ядром, віддалене трасування, один з найпопулярніших інструментів в сфері моніторингу	Вимагає сучасного ядра Linux (≥ 4.1)
DPDK	Максимальна продуктивність збору пакетів з мінімальними затримкам	Обмежений винятково мережевими задачами, вимагає спеціальної конфігурації та складніший у розгортанні
perf	Можливості профілювання CPU та подій ядра	Непридатний для моніторингу безпеки чи мережевого трафіку, сфокусований лише на продуктивності
ftrace	Можливість детального трасування функцій в ядрі	Вимагає доступу до ядра, складний синтаксис, важко інтегрувати в користувацький код
strace	Простий та зручний інструмент для перехоплення системних викликів	Високе навантаження, не підходить для реального часу та великого обсягу подій
DTrace	Потужні можливості трасування	Погана підтримка Linux, складне розгортання
iptables/nftables	Фільтрація пакетів та логування мережі	Тільки мережеві події, не підходить для моніторингу системних викликів
Auditd	Аудит подій та безпека	Високе навантаження при великому обсязі подій, обмежена аналітика
kprobes/uprobes	Гнучке трасування функцій ядра та user-space	Складність написання, низький рівень безпеки у порівнянні з eBPF

4

Рисунок Б.4 – Слайд 4

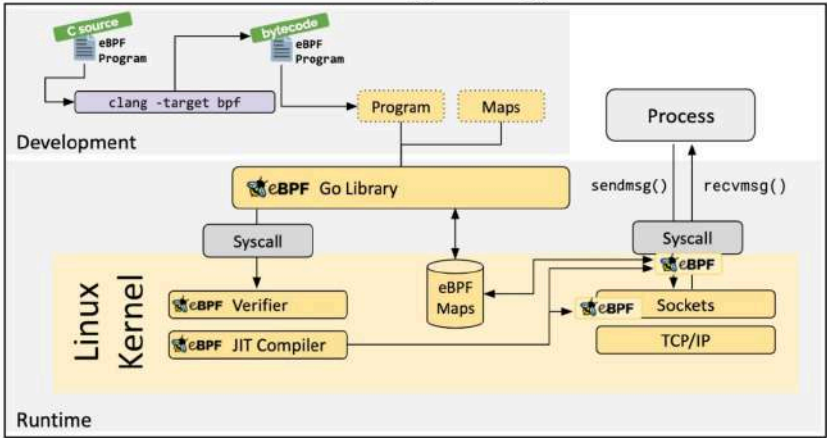
Порівняльна таблиця мов програмування для роботи з eBPF

Характеристика	Go (Golang)	Rust	Python
Продуктивність	Висока	Дуже висока	Середня
Безпека пам'яті	Середня	Висока	Середня
Простота мови	Висока	Середня	Дуже висока
Модулі для інтеграції з ядром	cilium/ebpf	aya-rs/aya	BCC (зв'язка з C)
Використання для eBPF проєктів	Є основною мовою для eBPF проєктів (Cilium, Falco, Pixie)	Поки немає високої популярності, але поступово зростає	Використовується в основному для прототипів
Можливості роботи з моніторинговими даними	Доступні модулі для роботи з OpenTelemetry	Доступні модулі для роботи з OpenTelemetry	Доступні модулі для роботи з OpenTelemetry

5

Рисунок Б.5 – Слайд 5

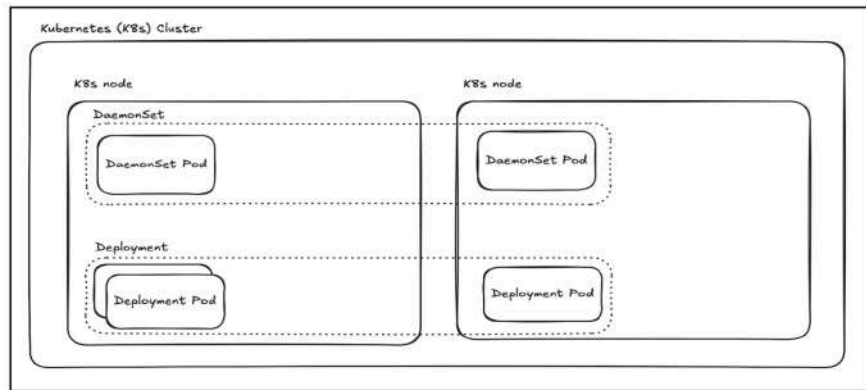
Процес компіляції й завантаження eBPF програм



6

Рисунок Б.6 – Слайд 6

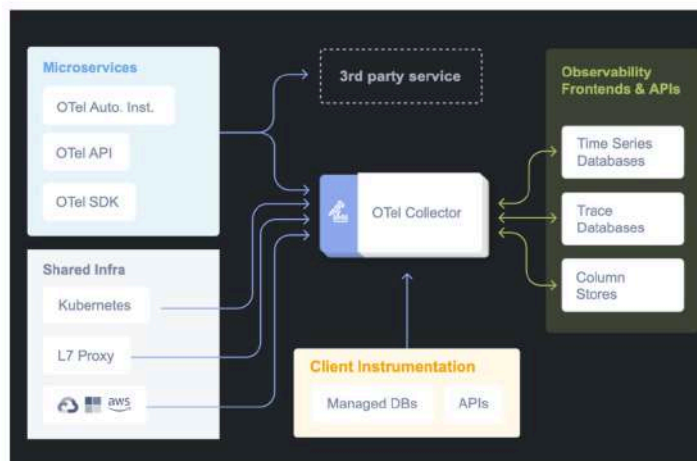
Розгортання додатку в контейнеризованих середовищах на прикладі Kubernetes



7

Рисунок Б.7 – Слайд 7

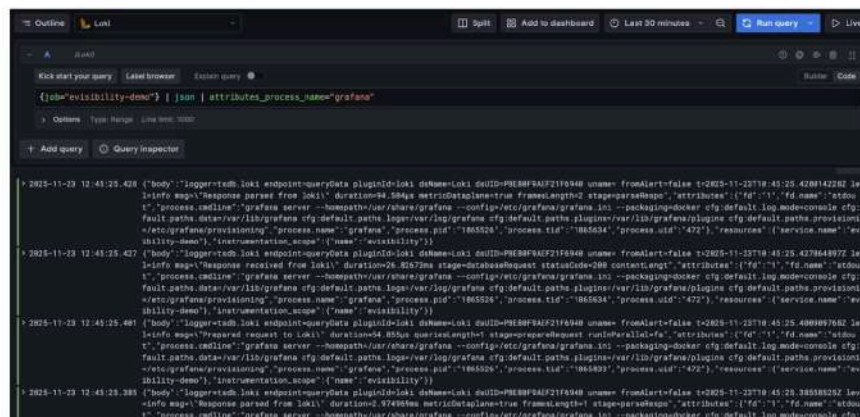
Екосистема OpenTelemetry



8

Рисунок Б.8 – Слайд 8

Результат збору логів



9

Рисунок Б.9 – Слайд 9

Результат збору метрик



10

Рисунок Б.10 – Слайд 10

Дякую за увагу!

13

Рисунок Б.13 – Слайд 13