

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

Факультет інформатики та обчислювальної техніки

Кафедра обчислювальної техніки

**До захисту допущено:
В.о. завідувача кафедри
Артем ВОЛОКИТА**

_____ (підпис)

“ ” _____ 2026 р.

**Дипломний проєкт
на здобуття ступеня бакалавра
за освітньо-професійною програмою “Комп’ютерні системи та мережі”
спеціальності 123 “Комп’ютерна інженерія”**

на тему: Система безпеки для Kubernetes з використанням eBPF
у IoT середовищах

Виконала : студентка 4 курсу, групи ІО-25
(шифр групи)

Антоненко Віолетта Станіславівна
(прізвище, ім’я, по батькові)

_____ (підпис)

Керівник ас. кафедри ОТ, Ковальчук О. М.
(посада, науковий ступінь, вчене звання, прізвище та ініціали)

_____ (підпис)

Консультант (нормоконтроль) ас. кафедри ОТ, PhD Коренко Д.В.
(назва розділу) (посада, вчене звання, науковий ступінь, прізвище та ініціали)

_____ (підпис)

Рецензент ас. кафедри ICT, PhD Омельченко В. В.
(посада, науковий ступінь, вчене звання, прізвище та ініціали)

_____ (підпис)

Засвідчую, що у цьому дипломному
проєкті немає запозичень з праць інших
авторів без відповідних посилань.

Студентка _____
(підпис)

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

Факультет інформатики та обчислювальної техніки

Кафедра обчислювальної техніки

Рівень вищої освіти – перший (бакалавр)

Освітньо-професійна програма

“Комп’ютерні системи та мережі”

спеціальності 123 “Комп’ютерна інженерія”

ЗАТВЕРДЖУЮ

В.о. завідувача кафедри

Артем ВОЛОКИТА

(підпис)

“__” _____ 2026 р.

ЗАВДАННЯ

на бакалаврський дипломний проєкт студентки

Антоненко Віолетти Станіславівни

1. Тема проєкту Система безпеки для Kubernetes з використанням eBPF у IoT середовищах

керівник проєкту Ковальчук Олександр Миконович, ас.,

(прізвище, ім’я, по батькові, науковий ступінь, вчене звання)

затверджені наказом по університету від від 3 червня 2026 року № 2055-с

2. Термін здачі студентом закінченого проєкту 1 червня 2026 р.

3. Вихідні дані до проєкту технічна документація, теоретичні дані.

4. Зміст розрахунково-пояснювальної записки (перелік питань, які розробляються)

Розділ 1. Аналіз об’єкта дослідження та предметної області.

Розділ 2. Аналіз існуючих рішень та обґрунтування вибору.

Розділ 3. Проєктування та реалізація системи orchestrator-aware in-kernel event suppression agent.

Розділ 4. Тестування та порівняльний аналіз продуктивності розробленої системи.

5. Перелік графічного матеріалу (з точним позначенням обов'язкових креслень) структурна схема системи, схема алгоритму дій програмного забезпечення, функціональна схема системи.

6. Консультанта проєкту, з вказівкою розділів проєкту, які до них вносяться

Розділ	Консультант	Підпис, дата	
		Завдання видав	Завдання прийняв
Нормоконтроль	Коренко Д. В.		

7. Дата видачі завдання «2» лютого 2026 р.

Календарний план

№ п/п	Найменування етапів дипломного проєкту	Терміни виконання етапів проєкту	Примітки
1.	<i>Затвердження теми проєкту</i>	10.12.2025-15.12.2025	
2.	<i>Вивчення та аналіз завдання</i>	15.12.2025-15.03.2026	
3.	<i>Розробка архітектури та загальної структури системи</i>	15.03.2026-25.03.2026	
4.	<i>Розробка структур окремих підсистем</i>	25.03.2026-05.04.2026	
5.	<i>Програмна реалізація системи</i>	05.04.2026-15.05.2026	
6.	<i>Оформлення пояснювальної записки</i>	25.05.2026-01.06.2026	
7.	<i>Захист програмного продукту</i>	27.04.2026	
8.	<i>Передзахист</i>	25.05.2026	
9.	<i>Захист</i>	15.06.2026	

Студентка-дипломниця _____ Віолетта АНТОНЕНКО
(підпис)

Керівник проєкту _____ Олександр КОВАЛЬЧУК
(підпис)

АНОТАЦІЯ

У бакалаврській дипломній роботі розроблено систему безпеки для Kubernetes-кластерів у середовищах IoT на основі технології eBPF, що дозволяє виконувати верифікований код безпосередньо у просторі ядра Linux. Архітектура реалізує придушення подій «на джерелі» шляхом пошуку за константний час у хеш-таблиці за ідентифікатором контрольної групи процесу, що усуває накладні витрати на передачу даних у простір користувача. Для примусового виконання політики безпеки впроваджено BPF-LSM хуки. Проведено порівняльний аналіз продуктивності з рішеннями Falco та Tetragon на платформах x86_64 та ARM64. Тестування на NVIDIA Jetson Nano підтвердило споживання CPU 0.07% при навантаженні 1000 системних викликів на секунду та споживання оперативної пам'яті 60 МБ, що доводить доцільність використання eBPF для кібербезпеки в умовах жорстких ресурсних обмежень.

Ключові слова: eBPF, Kubernetes, K3s, кібербезпека, IoT, придушення подій, BPF-LSM, XDP, ARM64, NVIDIA Jetson Nano, Falco, Tetragon.

ABSTRACT

In this bachelor's thesis, a security system for Kubernetes clusters in IoT environments was developed based on eBPF technology, which allows verified code to be executed directly in the Linux kernel space. The architecture implements drop-at-source event suppression via constant-time lookup in a hash table using the process control group identifier, eliminating user-space data transfer overhead. BPF-LSM hooks were implemented to enforce security policies in the kernel. A comparative performance analysis with Falco and Tetragon was conducted on x86_64 and ARM64 platforms. Testing on NVIDIA Jetson Nano confirmed CPU consumption of 0.07% under 1000 system calls per second and total RAM consumption of 60 MB, proving the feasibility of using eBPF for cybersecurity under severe resource constraints.

Key words: eBPF, Kubernetes, K3s, cybersecurity, IoT, event suppression, BPF-LSM, XDP, ARM64, NVIDIA Jetson Nano, Falco, Tetragon.

справки	Формат	Значення			Найменування	Кіл. листів	№ екземпляр	Додаток
					Документація загальна			
					Знову розроблена			
	A4	ІАЛЦ.467200.002 ТЗ			Система безпеки для	4		
					Kubernetes з використанням			
					eBPF у IoT середовищах			
					Технічне завдання			
	A4	ІАЛЦ.467200.003 ПЗ			Система безпеки для	130		
					Kubernetes з використанням			
					eBPF у IoT середовищах			
					Система безпеки для			
					Пояснювальна записка			
	A4	ІАЛЦ.467200.004 Д1			Система безпеки для	1		
					Kubernetes з використанням			
					eBPF у IoT середовищах			
					Структурна схема системи			
	A4	ІАЛЦ.467200.005 Д2			Система безпеки для	1		
					Kubernetes з використанням			
					eBPF у IoT середовищах			
					Функціональна схема			
					(діаграма класів)			
	A4	ІАЛЦ.467200.006 Д3			Система безпеки для	1		
					Kubernetes з використанням			
					eBPF у IoT середовищах			
					Алгоритм дії програмного			
					забезпечення			
	A4	ІАЛЦ.467200.007 Д4			Система безпеки для	39		
					Kubernetes з використанням			
					eBPF у IoT середовищах			
					Текст програмного коду			
					основних компонентів			
					системи			
					ІАЛЦ.467200.001 ОА			
Зм	Лист	№ докум.	Підп	Дата				
Розроб		Антоненко В. С.			Система безпеки для Kubernetes з використанням eBPF у IoT середовищах Опис альбому	Літ.	Арку ш	Аркушів
Перев		Ковальчук О. М.					1	1
						КПІ ім. Ігоря Сікорського ФІОТ ІО-25		

ТЕХНІЧНЕ ЗАВДАННЯ
ДО ДИПЛОМНОГО ПРОЄКТУ

на тему: «Система безпеки для Kubernetes з використанням
eVPF у IoT середовищах»

Київ – 2026

ЗМІСТ

1 НАЙМЕНУВАННЯ ТА СФЕРА ЗАСТОСУВАННЯ.....	2
2 ПІДСТАВИ ДЛЯ РОЗРОБКИ.....	2
3 МЕТА ТА ПРИЗНАЧЕННЯ РОЗРОБКИ.....	2
4 ДЖЕРЕЛА РОЗРОБКИ.....	2
5 ТЕХНІЧНІ ВИМОГИ.....	3
5.1. Вимоги до розробленого продукту	3
5.2. Вимоги до програмного забезпечення	3
5.3. Вимоги до апаратної частини	4
6 ЕТАПИ РОЗРОБКИ	4

					ІАЛЦ.467200.002 ТЗ							
Зм.	Арк.	№ докум.	Підпис	Дата								
Розробила		Антоненко В. С.			Система безпеки для Kubernetes з використанням eBPF у IoT середовищах Технічне завдання				Літ.	Аркуш	Аркушів	
Перевірив		Ковальчук О. М.									1	4
Реценз.		Омельченко В.В.							КПІ ім. Ігоря Сікорського, ФІОТ, ІО-25			
Н. Контр.		Коренко Д. В.										
Затвердив		Волокита А. М.										

1 НАЙМЕНУВАННЯ ТА СФЕРА ЗАСТОСУВАННЯ

Це технічне завдання поширюється на розробку системи безпеки Orchestrator-Aware In-Kernel Event Suppression Agent (OIKESA) для Kubernetes-кластерів у середовищах Інтернету речей (IoT) та периферійних обчислень з використанням технології eBPF.

Сферою застосування цієї системи є забезпечення кібербезпеки контейнеризованих робочих навантажень, виявлення та блокування загроз, моніторинг системних викликів у реальному часі, а також захист легковагових Kubernetes-дистрибутивів (K3s), що використовуються на одноплатних комп'ютерах та промислових IoT-шлюзах.

2 ПІДСТАВИ ДЛЯ РОЗРОБКИ

Підставою для розробки даної системи є завдання на виконання бакалаврського дипломного проєкту за освітньо-професійною програмою «Комп'ютерні системи та мережі» спеціальності 123 «Комп'ютерна інженерія», затверджене кафедрою обчислювальної техніки факультету інформатики та обчислювальної техніки Національного технічного університету України «Київський політехнічний інститут імені Ігоря Сікорського».

3 МЕТА ТА ПРИЗНАЧЕННЯ РОЗРОБКИ

Метою та призначенням даної роботи є розробка системи безпеки на базі eBPF для Kubernetes-кластерів у IoT середовищах, що реалізує контекстно-залежне придушення подій безпосередньо в ядрі Linux з мінімальними накладними витратами ресурсів.

4 ДЖЕРЕЛА РОЗРОБКИ

Джерелом розробки даного дипломного проєкту є технічна документація eBPF та Linux Kernel, специфікація Kubernetes API, документація компаній Cilium/Isovalent, Aqua Security та CNCF, науково-технічна література з кібербезпеки хмарних середовищ, а також публікації та статті у мережі Інтернет на відповідну тематику.

					ІАЛЦ.467200.002 ТЗ	Арк.
						2
Зм.	Арк.	№ докум.	Підпис	Дата		

5 ТЕХНІЧНІ ВИМОГИ

5.1. Вимоги до розробленого продукту

Розроблена система має задовольняти наступні вимоги:

- Забезпечення перехоплення та аналізу системних викликів через eBPF-програми у ядрі Linux.
- Реалізація придушення легітимних подій безпосередньо в ядрі шляхом пошуку в eBPF-карті за ідентифікатором cgroup із константним часом виконання.
- Забезпечення активного блокування шкідливих дій через BPF-LSM хуки.
- Виведення статистики продуктивності системи: ефективність придушення, кількість заблокованих подій, кількість дозволених под-ів.
- Оптимізація використання пам'яті ядра (LRU-карти, кільцевий буфер) та обчислювальних ресурсів процесора.
- Забезпечення persistence blocklist через Kubernetes ConfigMap для відновлення стану після перезапуску агента.
- Автоматична синхронізація стану Kubernetes Pod/Namespace з eBPF-картами через механізм Informers.

5.2. Вимоги до програмного забезпечення

- Операційна система: Linux (Ubuntu 22.04 LTS / 24.04 LTS або інший дистрибутив із версією ядра не нижче 5.8).
- Наявність підтримки eBPF у ядрі: активована підсистема BPF, увімкнені системні виклики BPF та JIT-компілятор.
- Для BPF-LSM: активація підсистеми BPF-LSM у конфігурації ядра та включення bpf до переліку активних модулів безпеки під час завантаження системи.
- Мови програмування: C та Go.
- Середовище виконання: Kubernetes (або K3s).

					ІАЛЦ.467200.002 ТЗ	Арк.
						3
Зм.	Арк.	№ докум.	Підпис	Дата		

5.3. Вимоги до апаратної частини

- ЦП: будь-який з підтримкою eBPF (x86_64 або ARM64).
- Оперативна пам'ять (RAM): не менше ніж 2 ГБ.
- Вільний простір на диску: не менше 500 МБ.
- Мережа: Ethernet/Wi-Fi для роботи кластера Kubernetes.

6 ЕТАПИ РОЗРОБКИ

Назва етапів виконання	Термін виконання
Затвердження теми роботи	10.12.2025-15.12.2025
Вивчення та аналіз завдання	15.12.2025-15.03.2026
Розробка архітектури та загальної структури системи	15.03.2026-25.03.2026
Розробка структур окремих частин системи	25.03.2026-05.04.2026
Програмна реалізація системи	05.04.2026-15.05.2026
Виправлення помилок	15.05.2026-01.06.2026
Оформлення пояснювальної записки	25.05.2026-01.06.2026

**ПОЯСНЮВАЛЬНА ЗАПИСКА
ДО ДИПЛОМНОГО ПРОЄКТУ**

на тему: «Система безпеки для Kubernetes з використанням
eVPF у IoT середовищах»

Київ – 2026

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ.....	4
ВСТУП	8
РОЗДІЛ 1 АНАЛІЗ ОБ'ЄКТА ДОСЛІДЖЕННЯ ТА ПРЕДМЕТНОЇ ОБЛАСТІ	10
1.1 Еволюція Cloud-Native парадигми та її експансія на периферію	10
1.2 Специфіка та обмеження безпеки в ресурсно-обмежених середовищах.....	16
1.3 Технологія eBPF як інструмент високоефективної обробки подій у ядрі	22
1.4 Концепція контекстно-залежного придушення подій (Context-Aware Suppression).....	28
1.5 Постановка задачі на проєктування системи OIKESA (Orchestrator-Aware In-Kernel Event Suppression Agent)	32
ВИСНОВОК ДО РОЗДІЛУ	35
РОЗДІЛ 2 АНАЛІЗ ІСНУЮЧИХ РІШЕНЬ ТА ОБҐРУНТУВАННЯ ВИБОРУ	37
2.1 Класифікація методів моніторингу безпеки в контейнеризованих середовищах	38
2.1.1 Традиційні механізми: auditd та ptrace	38
2.1.2 Модулі безпеки Linux (LSM) та BPF-LSM	41
2.1.3 Парадигма eBPF: типи хуків та їхня ефективність на ARM64	42
2.2 Порівняльний аналіз провідних Open-Source рішень безпеки Kubernetes	45
2.2.1 Falco: стандарт де-факто для хмарного аудиту	46

					ІАЛЦ.467200.003 ПЗ			
Зм.	Арк.	№ докум.	Підпис	Дата	Система безпеки для Kubernetes з використанням eBPF у IoT середовищах Пояснювальна записка	Літ.	Аркуш	Аркушів
Розробила	Антоненко В. С.						1	130
Перевірив	Ковальчук О. М.					КПІ ім. Ігоря Сікорського, ФІОТ, ІО-25		
Реценз.	Омельченко В.В.							
Н. Контр.	Коренко Д. В.							
Затвердив	Волокита А. М.							

2.2.2 Cilium Tetragon: превентивна безпека в ядрі.....	49
2.2.3 KubeArmor та Tracex: спеціалізація та архітектурні обмеження	52
2.3 Виявлення ключових обмежень існуючих підходів до забезпечення безпеки In-Kernel Suppression	58
2.3.1 Семантичний розрив між ядром та Kubernetes-контекстом	58
2.3.2 Надмірні накладні витрати на обробку подій.....	59
2.3.3 Неефективне управління станом та пам'яттю	59
ВИСНОВОК ДО РОЗДІЛУ	61
РОЗДІЛ 3 ПРОЄКТУВАННЯ ТА РЕАЛІЗАЦІЯ СИСТЕМИ ORCHESTRATOR- AWARE IN-KERNEL EVENT SUPPRESSION AGENT.....	63
3.1 Загальна архітектура системи та концепція дворівневої взаємодії.....	63
3.2 Проєктування підсистеми eBPF Maps та управління пам'яттю	65
3.2.1 Механізм захисту від LRU-флуду.....	67
3.2.2 Алгоритм TTL-очищення карти pending_cgroups	68
3.3 Реалізація стеку eBPF-програм перехоплення системних викликів	69
3.3.1 Центральний алгоритм контекстно-залежного придушення подій.....	70
3.3.2 Вибір механізму перехоплення: fentry та резервний raw_tracerepoint	71
3.3.3 Синхронне превентивне блокування через підсистему BPF-LSM.....	72
3.3.4 Мережева фільтрація на рівні XDP та TC.....	73

3.4 Алгоритми взаємодії агента з Kubernetes API та забезпечення персистентності	74
3.4.1 Алгоритм обробки життєвого циклу Pod-об'єктів	75
3.4.2 Механізм виявлення контейнерів через containerd gRPC	76
3.4.3 Динамічне прикріплення ТС-фільтрів до veth-інтерфейсів	77
3.4.4 Механізм персистентності стану через Kubernetes ConfigMap	78
3.5 Система метрик та спостережуваності	79
ВИСНОВОК ДО РОЗДІЛУ	82
РОЗДІЛ 4 ТЕСТУВАННЯ ТА ПОРІВНЯЛЬНИЙ АНАЛІЗ ПРОДУКТИВНОСТІ РОЗРОБЛЕНОЇ СИСТЕМИ	84
4.1 Методологія тестування.....	84
4.2 Стенд для тестування (ВМ та NVIDIA Jetson Nano)	85
4.3 Опис тестів продуктивності.....	87
4.4 Функціональне тестування безпеки	98
4.5 Тестування надійності та Edge-випадків	106
ВИСНОВОК ДО РОЗДІЛУ	120
ВИСНОВКИ.....	123
СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ.....	126

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ

BM	Віртуальна машина
ЦП	Центральний процесор
ЦОД	Центр обробки даних
ОС	Операційна система
ABI	Application Binary Interface — двійковий інтерфейс застосунку
AKS	Azure Kubernetes Service — хмарний сервіс Kubernetes від Microsoft Azure
API	Application Programming Interface — інтерфейс програмування застосунків
AppArmor	Application Armor — підсистема безпеки Linux на основі LSM
ARM64	64-розрядна архітектура процесорів Advanced RISC Machine
AuditD	Audit Daemon — підсистема аудиту ядра Linux
BPF	Berkeley Packet Filter — технологія фільтрації пакетів Berkeley
BPF-LSM	BPF Linux Security Modules — модулі безпеки Linux на основі eBPF
CNCF	Cloud Native Computing Foundation — фундація хмарно-орієнтованих обчислень
CNI	Container Network Interface — інтерфейс мережі контейнерів
CO-RE	Compile Once — Run Everywhere — «скомпілювати одного разу — запускати скрізь»
CPU	Central Processing Unit — центральний процесор
CRD	Custom Resource Definition — визначення користувацьких ресурсів Kubernetes
DDoS	Distributed Denial of Service — розподілена атака типу «відмова в обслуговуванні»

eBPF	Extended Berkeley Packet Filter — розширений фільтр пакетів Berkeley
EKS	Elastic Kubernetes Service — хмарний сервіс Kubernetes від Amazon AWS
etcd	Розподілене сховище даних типу «ключ-значення» для Kubernetes
GC	Garbage Collection — збирання сміття (управління пам'яттю)
GKE	Google Kubernetes Engine — хмарний сервіс Kubernetes від Google
gRPC	Google Remote Procedure Call — протокол віддаленого виклику процедур
HPA	Horizontal Pod Autoscaler — горизонтальний автомасштабувальник Pod'ів Kubernetes
HIPAA	Health Insurance Portability and Accountability Act — закон США про захист медичних даних
IoT	Internet of Things — Інтернет речей
IP	Internet Protocol — протокол Інтернету
IPv4	Internet Protocol version 4 — четверта версія протоколу Інтернету
IPv6	Internet Protocol version 6 — шоста версія протоколу Інтернету
JIT	Just-In-Time Compilation — компіляція «точно в час»
JSON	JavaScript Object Notation — формат обміну даними
K3s	Lightweight Kubernetes — легковаговий дистрибутив Kubernetes від Rancher Labs/SUSE
K8s	Kubernetes — скорочена назва системи оркестрації контейнерів Kubernetes
KPTI	Kernel Page-Table Isolation — ізоляція таблиць сторінок ядра
LLC	Last-Level Cache — кеш останнього рівня процесора

LKM	Loadable Kernel Module — модуль ядра, що динамічно завантажується
LPDDR4	Low Power Double Data Rate 4 — низькоенергетична оперативна пам'ять четвертого покоління
LRU	Least Recently Used — алгоритм витіснення давно невикористовуваних записів
LSM	Linux Security Modules — модулі безпеки Linux
LTE	Long-Term Evolution — стандарт мобільного зв'язку 4G
MAC	Mandatory Access Control — мандатний контроль доступу
MITRE	MITRE Adversarial Tactics, Techniques and Common Knowledge — база знань тактик та технік атак
MQTT	Message Queuing Telemetry Transport — протокол обміну повідомленнями для IoT
NIC	Network Interface Card — мережева інтерфейсна карта
NRI	Node Resource Interface — інтерфейс ресурсів вузла
OCI	Open Container Initiative — відкрита ініціатива контейнерів
OIKESA	Orchestrator-Aware In-Kernel Event Suppression Agent — оркестратор-свідомий агент придушення подій у ядрі
OOM	Out-Of-Memory — нестача оперативної пам'яті
PCI DSS	Payment Card Industry Data Security Standard — стандарт безпеки даних індустрії платіжних карток
PID	Process Identifier — ідентифікатор процесу
QoS	Quality of Service — клас якості обслуговування
RAM	Random Access Memory — оперативна пам'ять
RSS	Resident Set Size — обсяг фізичної пам'яті, що займає процес
SELinux	Security-Enhanced Linux — модуль безпеки Linux від АНБ США

SIGKILL	Signal Kill — сигнал примусового завершення процесу в Linux
SOC2	Service Organization Control 2 — стандарт аудиту безпеки для сервісних організацій
SQLite	Вбудована реляційна база даних без окремого сервера
TC	Traffic Control — підсистема управління мережевим трафіком у ядрі Linux
TCP	Transmission Control Protocol — протокол управління передачею даних
TLB	Translation Lookaside Buffer — буфер асоціативної кеш-пам'яті для трансляції адрес
TTL	Time To Live — час існування/дії запису
UDP	User Datagram Protocol — протокол датаграм користувача
uid/gid	User Identifier / Group Identifier — ідентифікатор користувача / ідентифікатор групи
x86_64	64-розрядна архітектура процесорів Intel/AMD (також AMD64)
XDP	eXpress Data Path — підсистема швидкісної обробки мережесих пакетів у ядрі Linux
YAML	YAML Ain't Markup Language — мова серіалізації даних для конфігураційних файлів
5G	П'яте покоління мобільного зв'язку

ВСТУП

Стрімке зростання кількості підключених пристроїв зумовлює масштабне розгортання Kubernetes безпосередньо на периферії мережі. Легковагові дистрибутиви кластера, зокрема K3s, виконуються на одноплатних комп'ютерах архітектури ARM64 з оперативною пам'яттю 512 МБ–4 ГБ та обмеженою тактовою частотою. У таких умовах традиційні засоби моніторингу безпеки породжують критичну проблему: кожен перехоплений системний виклик вимагає перемикання контексту між ядром і простором користувача, примусового скидання буфера асоціативної кеш-пам'яті та серіалізації даних. Це створює значні накладні витрати, які в умовах дефіциту обчислювальних ресурсів IoT-вузлів призводять до різкого зниження пропускної здатності мережі та перевантаження процесора. Водночас у Kubernetes-кластері абсолютна більшість системних викликів породжується легітимними системними компонентами, чий трафік заздалегідь відомий оркестратору. Передача цих "відомо легітимних" подій у простір користувача є марними накладними витратами — феноменом, що дістав назву "шум подій".

Об'єктом дослідження є процеси забезпечення інформаційної безпеки та моніторингу подій у контейнеризованих середовищах під керуванням Kubernetes.

Предметом дослідження є методи та програмні засоби фільтрації й придушення легітимних подій безпосередньо на рівні ядра операційної системи за допомогою технології eBPF.

Метою роботи є підвищення продуктивності та ефективності систем моніторингу безпеки Kubernetes в IoT-середовищах шляхом розробки низькорівневого агента OIKESA (Orchestrator-Aware In-Kernel Event

					ІАЛЦ.467200.003 ПЗ	Арк.
						8
Зм.	Арк.	№ докум.	Підпис	Дата		

Suppression Agent), який придушує відомі легітимні події безпосередньо в ядрі Linux, унеможливаючи їх передачу у простір користувача.

Для досягнення поставленої мети у роботі необхідно вирішити такі задачі: провести аналіз архітектурних обмежень існуючих eBPF-агентів безпеки; дослідити підсистеми ядра Linux, необхідні для реалізації внутрішньоядерного придушення подій: механізми fentry/fexit, raw tracepoints, BPF-LSM, CO-RE (Compile Once — Run Everywhere), а також типи eBPF-мап та їх продуктивнісні характеристики; спроектувати дворівневу архітектуру системи OIKESA з розподілом між простором ядра та простором користувача, визначити структури eBPF-мап та інтерфейси взаємодії з Kubernetes Informers і containerd; реалізувати прототип системи на стеку Linux, Go та eBPF з підтримкою containerd та Kubernetes; провести тестування системи на стенді ARM64 та віртуальній машині x86_64 за категоріями порівняльних, безпекових, функціональних та надійнісних тестів.

Таким чином, предметна область даного дослідження охоплює сукупність теоретичних і практичних аспектів забезпечення безпеки в IoT-середовищах під керуванням Kubernetes, зокрема застосування інноваційної технології eBPF для оптимізації процесів моніторингу. Увага зосереджується на підходах, що дозволяють мінімізувати накладні витрати на обробку «шуму подій» та зберегти обчислювальні ресурси периферійних вузлів без втрати надійності контролю. Розуміння цих принципів є основою для подальшого обґрунтування архітектурних рішень та формування високоефективної моделі внутрішньоядерної фільтрації легітимного трафіку.

					ІАЛЦ.467200.003 ПЗ	Арк.
						9
Зм.	Арк.	№ докум.	Підпис	Дата		

РОЗДІЛ 1

АНАЛІЗ ОБ'ЄКТА ДОСЛІДЖЕННЯ ТА ПРЕДМЕТНОЇ ОБЛАСТІ

1.1 Еволюція Cloud-Native парадигми та її експансія на периферію

Сучасна цифрова трансформація зумовила безпрецедентне зростання кількості підключених пристроїв та обсягів даних, що генеруються на периферії інфраструктури. Згідно з прогнозами Transforma Insights, кількість активних IoT-з'єднань у світі досягла 18.9 мільярда у 2024 році і, за оцінками, зросте до 48.0 мільярда до 2035 року [1]. Стрімке зростання масштабів IoT є каталізатором переходу від класичної централізованої моделі хмарних обчислень, де вся обробка даних зосереджена у великих дата-центрах, до концепції периферійних обчислень (Edge Computing). Як зазначає Satyanarayanan [2], цей підхід передбачає розміщення значних обчислювальних ресурсів на межі мережі, поблизу мобільних пристроїв, сенсорів та IoT-обладнання, з використанням так званих «хмарлеток» (cloudlets) для забезпечення обробки даних на периферії.

Концепція Edge Computing виникла як відповідь на об'єктивні обмеження централізованих хмарних архітектур у контексті IoT-навантажень. Затримка передачі даних від периферійного пристрою до центрального хмарного сервера та назад може сягати від 80 до 200 мілісекунд, що є критично неприйнятним для застосунків реального часу — промислових систем управління, медичних сенсорів, автономних транспортних засобів або систем розумних міст [3]. Саме тут фізична близькість «хмарлетів» до пристроїв стає вирішальною. Згідно з аналізом Satyanarayanan [2], ця близькість забезпечує чотири ключові переваги:

					ІАЛЦ.467200.003 ПЗ	Арк.
						10
Зм.	Арк.	№ докум.	Підпис	Дата		

високу оперативність хмарних сервісів (низька затримка та варіативність затримки), масштабованість через аналітику на периферії (зменшення навантаження на мережу), забезпечення політик конфіденційності на місці збору даних, а також маскування збоїв хмарних сервісів шляхом резервного перемикання на найближчий cloudlet. Крім того, безперервна передача масивних обсягів телеметрії від мільярдів IoT-пристроїв до центральних центрів обробки даних генерує колосальне мережеве навантаження та витрати на пропускну здатність каналів зв'язку. Ще у 2016 році автори фундаментальної праці з периферійних обчислень [3] на основі аналізу звітів Cisco Global Cloud Index зазначали, що загальний обсяг даних, створених людьми, машинами та речами, мав досягти 500 зеттабайт до 2019 року, тоді як глобальний трафік ЦОД за той самий період прогнозувався лише на рівні 10,4 зеттабайта. Це колосальне розходження чітко свідчило про фізичну неможливість передачі всього масиву сирих даних до хмари. Саме тому авторами [3] було обґрунтовано, що понад 45% даних, створених у середовищах інтернет-речей, вже до 2019 року мали зберігатися, оброблятися та аналізуватися безпосередньо на периферії мережі або в безпосередній близькості до неї.

Подальший розвиток глобальної цифрової інфраструктури повністю підтвердив цей технологічний тренд та навіть перевершив ранні очікування. За актуальними статистичними даними аналітичної компанії Western Digital, опублікованими на платформі Statista [4], фактичний обсяг щорічно згенерованих та реплікованих цифрових даних у світі демонстрував стрімке експоненціальне зростання, збільшившись з 41 зеттабайта у 2019 році до 173,4 зеттабайта наприкінці 2024 року. Подібна динаміка остаточно закріпила концепцію Edge Computing як базовий інженерний стандарт. Архітектурна логіка диктує гостру необхідність первинного аналізу, фільтрації та обробки даних безпосередньо там, де вони виникають — на межі мережі, що мінімізує

					ІАЛЦ.467200.003 ПЗ	Арк.
						11
Зм.	Арк.	№ докум.	Підпис	Дата		

транзитний трафік і вимагає розгортання захищених локальних засобів оркестрації.

Паралельно з розвитком Edge Computing відбулась революція у сфері оркестрації програмного забезпечення. Kubernetes — відкрита платформа оркестрації контейнерів [5] — стала стандарт де-факто для управління хмарними застосунками [6]. Однак, як зазначають Ras та Brorsson [6], стандартний Kubernetes орієнтований на потужні серверні інфраструктури з однорідною мережевою топологією та не враховує географічне розташування вузлів, що робить його непридатним для розгортання на ресурсно-обмежених IoT-пристроях через високі вимоги до обчислювальних ресурсів та пам'яті. Відповіддю на цей виклик стала поява спеціалізованих легковагових дистрибутивів, найбільш поширеним з яких є K3s, розроблений компанією Rancher Labs (нині SUSE) [7].

K3s був створений шляхом видалення надлишкових складових з повноцінного Kubernetes, що дозволило зменшити розмір дистрибутива до єдиного бінарного файлу розміром менше 100 МБ [7]. Ця система кардинально переосмислює архітектуру Kubernetes для роботи в умовах обмежених ресурсів. Ключова інновація K3s полягає в об'єднанні всіх компонентів кластера — API-сервера, планувальника, менеджера контролерів та агента вузла — в єдиний процес. Крім того, розподілене сховище etcd замінено на вбудовану базу даних SQLite, що суттєво скорочує потреби в оперативній пам'яті та спрощує операційне адміністрування [7]. Завдяки цим оптимізаціям K3s здатний повноцінно функціонувати на одноплатних комп'ютерах ARM-архітектури, зокрема на таких пристроях як NVIDIA Jetson Nano (4 ядра Cortex-A57, 4 ГБ пам'яті LPDDR4) [8], а також на промислових шлюзах із мінімальними апаратними характеристиками.

					ІАЛЦ.467200.003 ПЗ	Арк.
						12
Зм.	Арк.	№ докум.	Підпис	Дата		

Архітектура K3s передбачає два типи вузлів, що розподіляють функції між площиною керування та робочим навантаженням, як показано на Рис. 1.1. Серверний вузол (англ. server node) — це хост, на якому виконується команда K3s server; він містить усі компоненти площини керування та сховище даних. Агентний вузол (англ. agent node) — це хост, на якому виконується команда K3s agent; він не містить жодних компонентів площини керування чи сховища даних і призначений виключно для виконання робочих навантажень. На обох типах вузлів обов'язково працюють: kubelet (агент вузла Kubernetes), середовище виконання контейнерів (containerd) та мережевий плагін CNI (Container Network Interface) для забезпечення міжподового зв'язку.

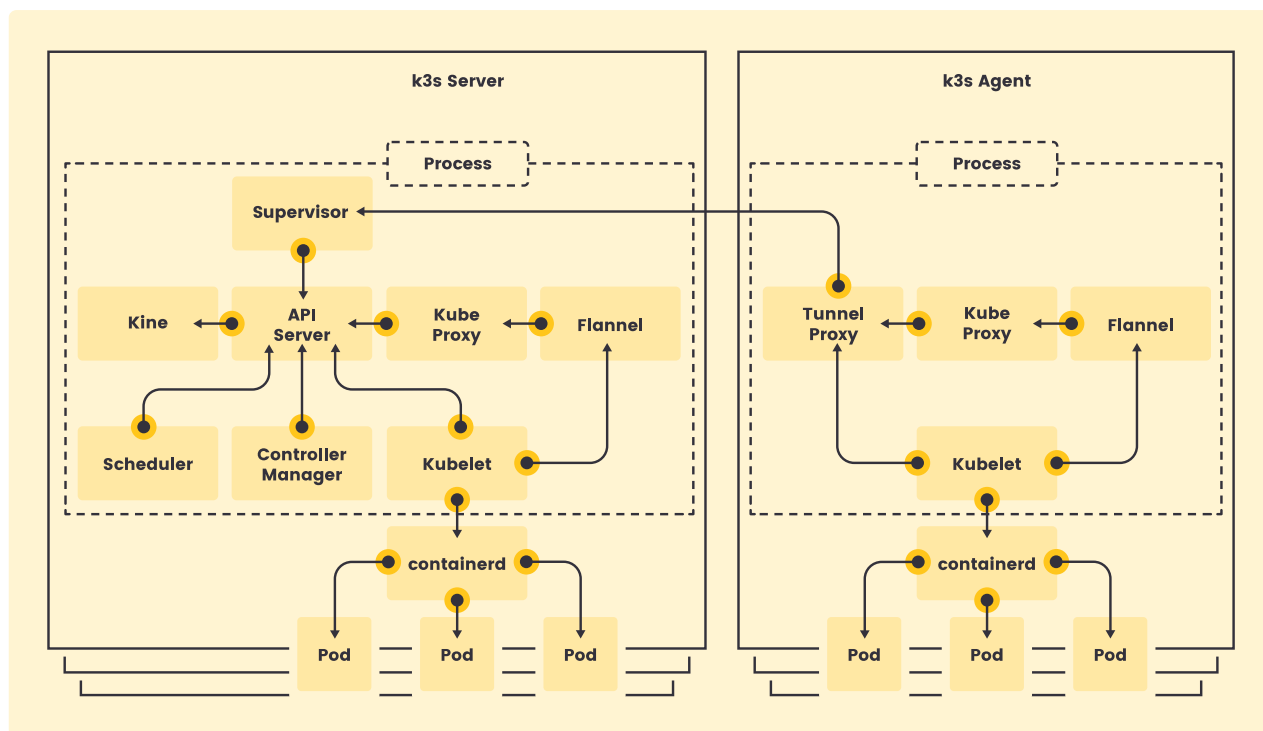


Рисунок 1.1 – Типи вузлів архітектури K3s: серверний вузол із площиною керування та агентний вузол із робочим навантаженням [9]

Для розуміння ступеня спрощення, яке привносить K3s, доцільно розглянути еталонну архітектуру стандартного Kubernetes-кластера, зображену

на Рис. 1.2. У повноцінному Kubernetes площина керування складається з окремих незалежних компонентів: kube-apiserver (єдина точка входу для всіх запитів до кластера), etcd (розподілене сховище стану кластера), kube-scheduler (планувальник розміщення Pod'ів на вузлах) та kube-controller-manager (набір контролерів, що підтримують бажаний стан ресурсів). Кожен робочий вузол (worker node) містить kubelet, kube-proxy та середовище виконання контейнерів. Така архітектура передбачає розгортання чотирьох окремих процесів тільки для площини керування, кожен із яких конкурує за оперативну пам'ять і ресурси CPU.

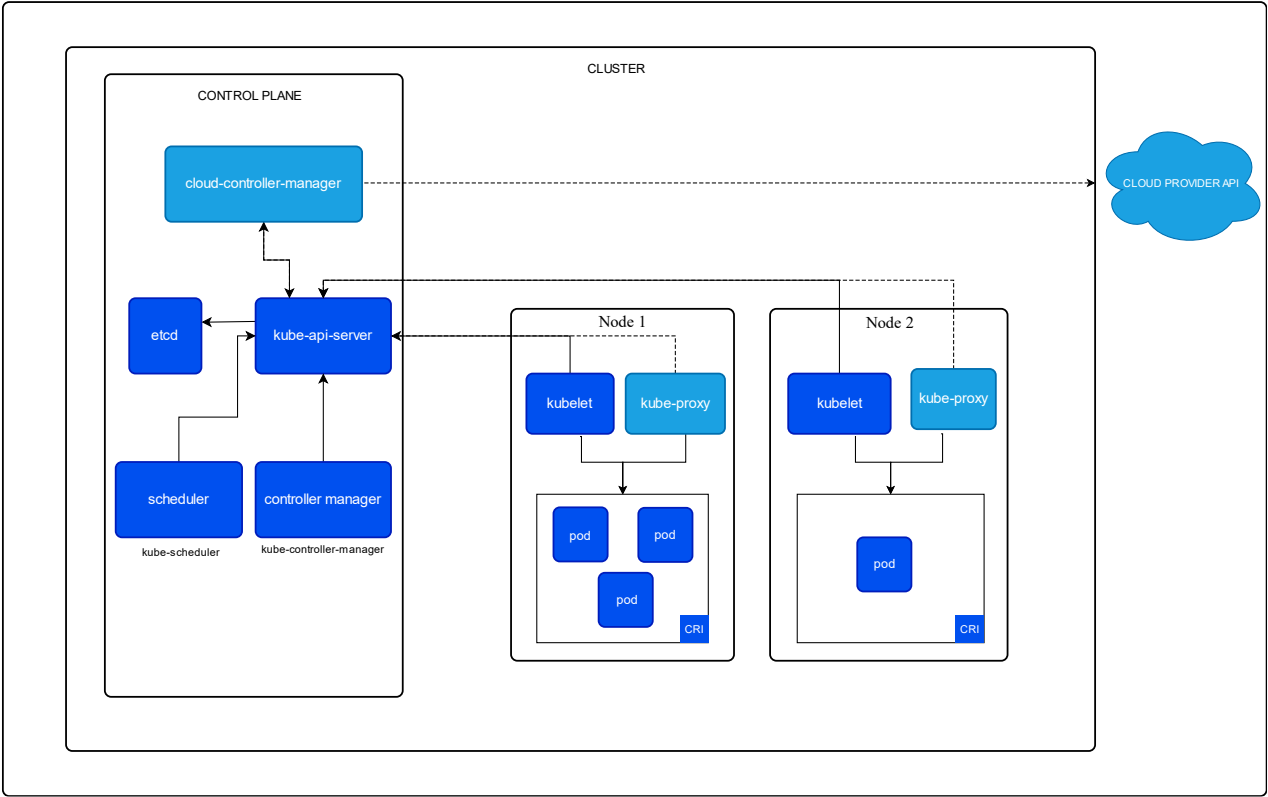


Рисунок 1.2 – Архітектура Kubernetes-кластера: компоненти площини керування та робочих вузлів [10]

Найпростішою конфігурацією розгортання K3s є однорівневий кластер з одним серверним вузлом, схему якого наведено на Рис. 1.3. У цій конфігурації

єдиний серверний вузол містить вбудовану базу даних SQLite, що замінює розподілений кластер etcd, характерний для повноцінного Kubernetes. Кожен агентний вузол реєструється безпосередньо на цьому серверному вузлі та отримує від нього інструкції щодо планування контейнерів. Користувач K3s взаємодіє з кластером через kubectl або безпосередньо через API K3s на серверному вузлі. Така конфігурація є оптимальною для ресурсно-обмежених IoT-середовищ, де вимоги до відмовостійкості площини керування є несуттєвими порівняно з мінімізацією споживання ресурсів.

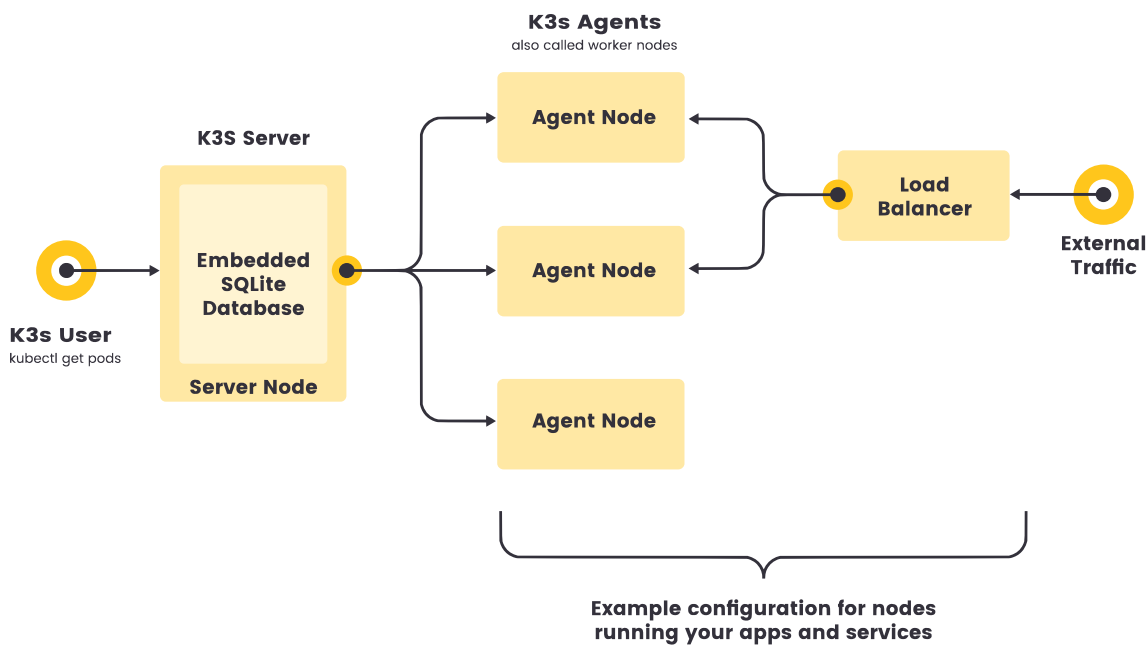


Рисунок 1.3 – Кластер, що складається з одного вузла-сервера K3s із вбудованою базою даних SQLite [9]

Особливої уваги заслуговує підтримка архітектури ARM64, оскільки сучасні одноплатні комп'ютери та IoT-шлюзи, прикладом яких є NVIDIA Jetson Nano [8], базуються саме на цій архітектурі. Згідно з фундаментальними засадами проєктування обчислювальних систем [11], ARM64 суттєво

відрізняється від традиційних серверних архітектур x86_64 за набором інструкцій (ISA) та профілем енергоефективності.

Ці архітектурні відмінності мають безпосередній вплив на функціонування низькорівневого інструментарію, зокрема технології eBPF (Extended Berkeley Packet Filter) [12]. Оскільки виконання eBPF-програм на рівні ядра залежить від механізмів JIT-компіляції Just-in-Time [13], [14], які в архітектурі ARM реалізовані інакше порівняно з x86 [12], розробка систем безпеки для таких середовищ вимагає спеціалізованих рішень.

Таким чином, поєднання масштабного розгортання IoT-пристроїв та зрілості платформи K3s сформувало новий клас виробничих середовищ — кластери Kubernetes на периферії мережі з жорсткими апаратними обмеженнями.

1.2 Специфіка та обмеження безпеки в ресурсно-обмежених середовищах

Розгортання K3s на периферійних вузлах вирішує проблему оркестрації контейнерів, однак породжує принципово новий клас викликів для забезпечення кібербезпеки на рівні виконання (runtime security). Типовий Edge-вузол у IoT-інфраструктурі функціонує з обсягом оперативної пам'яті 512 МБ–2 ГБ — що відповідає мінімальним апаратним вимогам K3s для агентних та серверних вузлів відповідно [15] — та обмеженою пропускною здатністю каналу зв'язку. Характерним представником таких систем є NVIDIA Jetson Nano з 4-ядерним процесором ARM Cortex-A57 з частотою 1.43 GHz та 4 ГБ оперативної пам'яті [8]. На такому вузлі одночасно виконуються: ядро K3s з усіма системними компонентами, мережеві проксі-сервери (наприклад, Envoy як sidecar для service mesh), корисне навантаження бізнес-логіки (IoT-агрегатори, брокери MQTT, обробники телеметрії) та, потенційно, агент

					ІАЛЦ.467200.003 ПЗ	Арк.
						16
Зм.	Арк.	№ докум.	Підпис	Дата		

безпеки. Сукупне споживання ресурсів цими компонентами вже наближається до апаратних меж, залишаючи вкрай мало простору для додаткових накладних витрат [8], [15].

Традиційні підходи до забезпечення безпеки на рівні виконання засновані на моделі «видобування та оцінювання у просторі користувача» (extraction-and-evaluation model). Агент безпеки підключається до потоку системних викликів ядра Linux — через завантажувані модулі ядра або базові eBPF-зонди — та перенаправляє сирі події у кільцевий буфер (ring buffer), звідки вони зчитуються рушієм правил, що виконується у просторі користувача [12], [13]. Загальну логіку цієї моделі та шлях проходження події через системні шари зображено на Рис. 1.4. Цей рушій застосовує фільтри, збагачує події контекстом та генерує сповіщення. Хоча така архітектура є функціональною на потужних серверах, її накладні витрати є неприйнятними в умовах IoT.

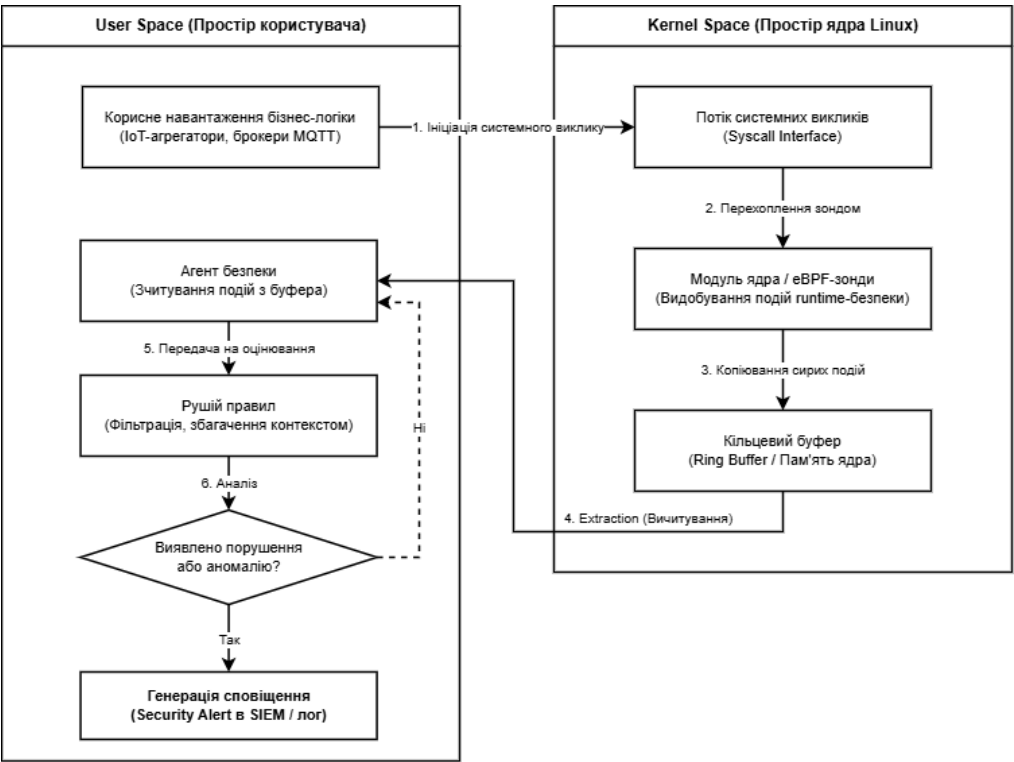


Рисунок 1.4 – Традиційна модель «extraction-and-evaluation»: перенесення подій із простору ядра до простору користувача

Центральною проблемою цієї моделі є перемикання контексту (context switching) між простором ядра (kernel space) та простором користувача (user space). Кожен системний виклик, що підлягає моніторингу, повинен подолати цей кордон: eBPF-програма записує подію в ring buffer у ядрі, після чого агент у просторі користувача зчитує її, здійснюючи системний виклик read(). Цей процес включає зміну рівня привілеїв процесора, збереження та відновлення стану регістрів [11], а також примусове скидання буфера асоціативної кеш-пам'яті (TLB flush) через механізм ізоляції таблиць сторінок ядра (Kernel Page-Table Isolation, KPTI), впроваджений для протидії вразливостям типу Spectre/Meltdown [16]. Дослідження показують, що за інтенсивного моніторингу системних викликів ці накладні витрати здатні сягати від кількох відсотків до понад 17% ресурсів CPU залежно від швидкості системних викликів та розміру робочого набору пам'яті [16]. На периферійних ARM64-пристроях, де тактова частота та кількість ядер є значно меншими, ці накладні витрати набувають характеру ресурсної катастрофи.

Сумарна затримка обробки одного перехопленого системного виклику описується математичною моделлю:

$$T_{total} = T_{hook} + T_{context\ switch} + T_{user\ space}, \quad (1.1)$$

де T_{hook} — прямі витрати на виконання eBPF-програми в ядрі [12], [13]; $T_{context\ switch}$ — витрати на перемикання контексту з урахуванням TLB flush та збереження/відновлення регістрів [11], [16]; $T_{user\ space}$ — час роботи рушія правил у просторі користувача, включаючи десеріалізацію та збагачення метаданими [12].

Хоча офіційна документація Tracее декларує мінімальний вплив на продуктивність (Low Overhead) [17], незалежні рецензовані дослідження фіксують суттєво вищий overhead у середовищах з активними політиками безпеки. Порівняльний аналіз чотирьох eBPF-агентів — KubeArmor, Falco,

Tetragon і Tracee — у хмарно-орієнтованих середовищах Kubernetes показав, що при застосуванні процесних політик моніторингу Tracee демонструє overhead у 37,59% відносно базового стану, що є найгіршим показником серед досліджуваних інструментів; при мережевому моніторингу цей показник зростає до 71,50% [18]. Автори пояснюють це архітектурним рішенням Tracee використовувати ланцюжок tail-called eBPF-програм, у якому операції зіставлення з правилами та перевірки системних викликів виконуються повторно на кожному кроці ланцюжка. Ця проблема має структурне підґрунтя: eBPF-програми прикріплюються до системних hookpoints глобально та виконуються для кожного процесу при кожному системному виклику — у тому числі для тих, що не є ціллю моніторингу. Milo Craun та інші автори демонструють, що навіть найефективніший підхід до per-process фільтрації уповільнює системний виклик read() на 15% на нецільових процесах, і цей overhead масштабується пропорційно кількості прикріплених hookpoints [19]. В умовах обмежених IoT-вузлів з двоядерними ARM-процесорами сукупність цих факторів фактично унеможлиблює паралельне виконання корисного навантаження.

Іншою критичною проблемою є феномен backpressure у кільцевих буферах. Ring buffer є єдиним каналом передачі подій між eBPF-програмою в ядрі та рушієм правил у просторі користувача. Якщо рушій правил не встигає обробляти події з достатньою швидкістю — через паузи збирача сміття у мові Go, просту конкуренцію за CPU або надмірну складність аналізу — буфер заповнюється. Ядро, не маючи можливості записати нову подію в переповнений буфер, просто відкидає її [19]. Процес переповнення буфера та втрати критичних даних внаслідок надмірного навантаження наочно показано на Рис.1.5. Це явище, відоме як тиск зворотного зв'язку (backpressure), має серйозні наслідки для безпеки: саме під час інтенсивного навантаження, коли

зловмисник навмисно генерує масу легітимних системних викликів (adversarial flooding), у буфері втрачаються сповіщення про реальні загрози.

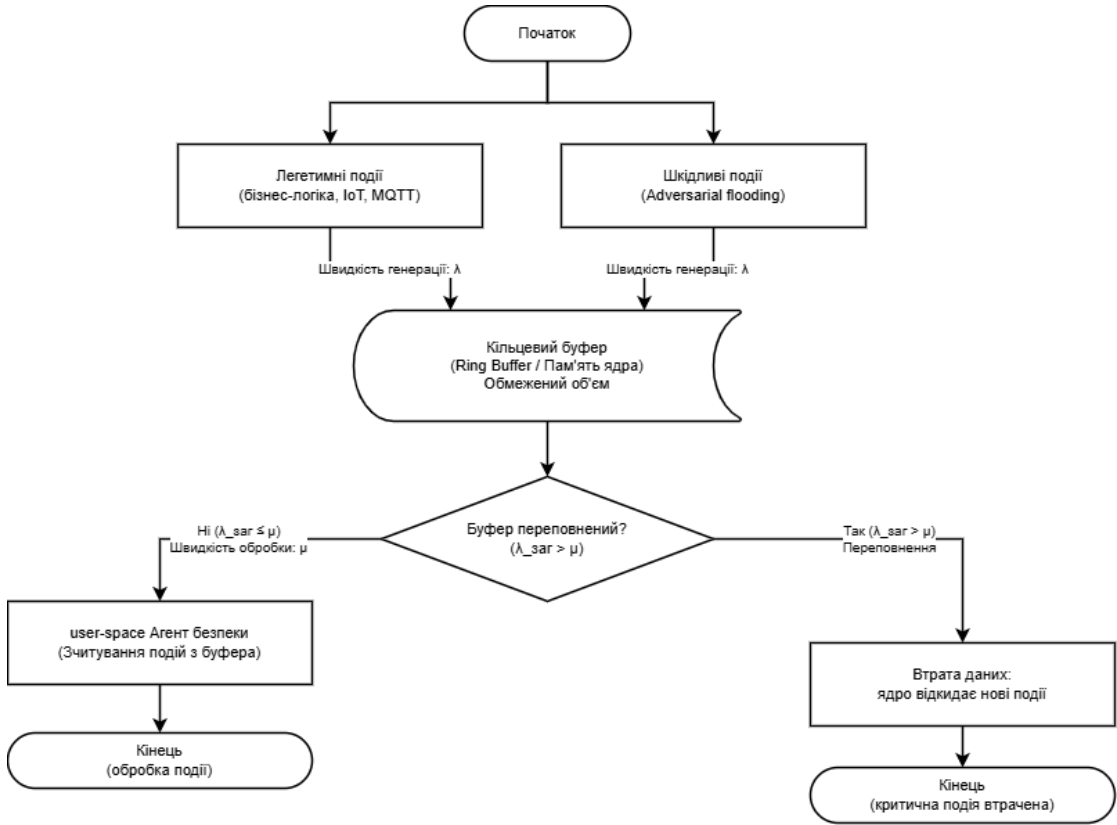


Рисунок 1.5 – Механізм виникнення backpressure у кільцевому буфері під час інтенсивної генерації подій (де λ — сумарна швидкість генерації подій обома джерелами)

Математична модель ймовірності втрати критичної події через переповнення кільцевого буфера фіксованого розміру K ґрунтується на апараті системи масового обслуговування $M/M/1/K$ [20]. У стандартній нотації Кендалла: перший символ M позначає марковський (тобто експоненціальний) розподіл для інтервалів між надходженнями подій; другий M — експоненціальний розподіл для часу обробки події агентом у просторі користувача; 1 вказує на наявність одного каналу обробки; K обмежує максимальну кількість подій, що можуть одночасно перебувати в системі (тобто

розмір кільцевого буфера). Отже, М/М/1/К описує класичну систему масового обслуговування з одним сервером та обмеженою чергою розміром К. Згідно з [20], стаціонарна ймовірність того, що система переповнена (тобто в буфері немає місця для нової події), визначається формулою:

$$P_{drop} = \frac{(1 - \rho) \times \rho^K}{1 - \rho^{K+1}}, \text{ при } \rho \neq 1, \quad (1.2)$$

де $\rho = \frac{\lambda}{\mu}$ — коефіцієнт завантаження буфера; λ — швидкість генерації подій системними викликами (events/c); μ — швидкість споживання подій user-space агентом (events/c); К — ємність кільцевого буфера (кількість записів). На відміну від необмеженої черги, система М/М/1/К досягає стаціонарного стану навіть при $\rho > 1$: нова подія, що надходить у повний буфер, просто відкидається [20]. В умовах adversarial flooding — коли зловмисник навмисно генерує масу легітимних системних викликів — ρ прямує до нескінченності, і P_{drop} до одиниці: система безпеки стає сліпою саме в момент виконання шкідливого payload [19].

Окремої уваги заслуговує проблема подвійного ресурсного «податку» (double tax penalty). У мікросервісних архітектурах на базі Kubernetes типовий Pod може включати sidecar-контейнер мережевого проксі-сервера (наприклад, Envoy в рамках service mesh Istio), який сам по собі споживає значну кількість CPU для обробки мережевого трафіку [21]. Додавання поверх цього агента безпеки з user-space рушієм правил означає, що корисне навантаження бізнес-логіки конкурує одночасно з двома сервісними компонентами. На обмеженому ARM64-вузлі цей сценарій практично унеможлиблює нормальну роботу застосунку.

Відтак ресурсні обмеження IoT-середовищ та архітектурні недоліки традиційних агентів безпеки формують технічне завдання: необхідна

архітектура, що переносить логіку фільтрації в ядро операційної системи, уникаючи costly context switching та проблеми backpressure.

Таблиця 1.1 – Порівняння характеристик Cloud-вузлів та IoT-вузлів у контексті розгортання K3s/Kubernetes

Характеристика	Cloud-вузол (дата-центр)	IoT-вузол (K3s)
Архітектура CPU	x86_64 (Intel Xeon, AMD EPYC)	ARM64 (NVIDIA Jetson Nano, Cortex-A57)
Обсяг RAM	64–512 ГБ та більше	512 МБ – 4 ГБ (типово 1–2 ГБ)
Кількість ядер CPU	32–128 ядер	4 ядра @ 1.43 GHz (Cortex-A57)
Дистрибутив Kubernetes	Стандартний K8s, EKS, GKE, AKS	K3s (100 МБ, SQLite замість etcd)
Мережа	10–100 Gbps, стабільна	Wi-Fi / LTE / 5G, переривчасте з'єднання
Фізична безпека	Закрита серверна кімната	Вулиця, транспорт, незахищені локації

1.3 Технологія eBPF як інструмент високоефективної обробки подій у ядрі

Extended Berkeley Packet Filter (eBPF) — технологія, що кардинально змінила підхід до системного програмування, спостережуваності (observability) та безпеки в Linux-екосистемі. eBPF уможливлює виконання верифікованого коду у просторі ядра Linux без модифікації його вихідного коду та без завантаження традиційних модулів ядра [13]. Завдяки цій властивості стає можливою реалізація високоефективної логіки фільтрації та аналізу подій там, де вони виникають — в ядрі операційної системи.

Повний життєвий цикл eBPF-програми, зображений на Рис. 1.6, охоплює два основні етапи: розробку (Development) та виконання (Runtime). На етапі

розробки вихідний код мовою C компілюється за допомогою clang з цільовою архітектурою BPF у байт-код, після чого формуються об'єкти програми та карт (Maps). На етапі виконання бібліотека (наприклад, eBPF Go Library) завантажує програму через системний виклик у ядро, де вона послідовно проходить верифікацію (BPF Verifier) та JIT-компіляцію, а потім прикріплюється до відповідних хуків ядра (Sockets, TCP/IP) для перехоплення подій у реальному часі [22].

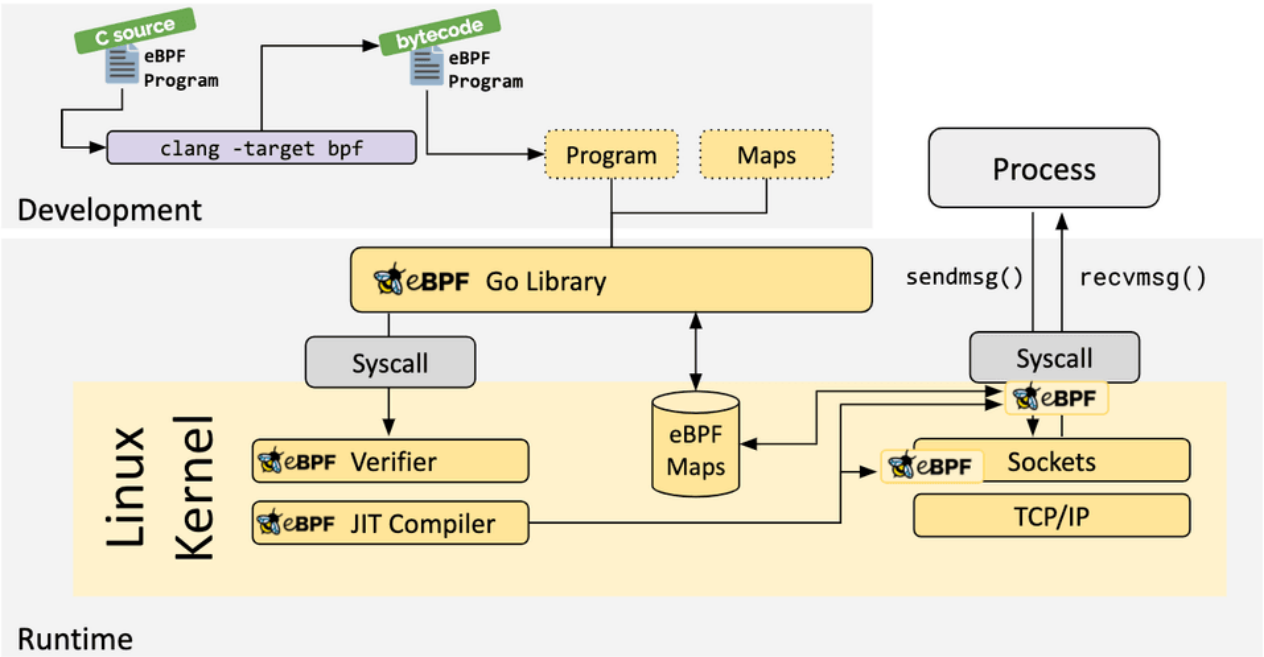


Рисунок 1.6 – Повний життєвий цикл програми eBPF з бібліотекою Go [22]

Архітектура підсистеми eBPF складається з кількох ключових складових. Центральним елементом є верифікатор — статичний аналізатор, що виконує вичерпну перевірку кожної eBPF-програми перед її завантаженням у ядро [13]. Верифікатор гарантує безпеку виконання: відсутність безмежних циклів (програма обов'язково завершиться), неможливість доступу до довільних адрес пам'яті поза визначеними областями, коректну ініціалізацію всіх змінних та відповідність кожного звернення до даних дозволеним межах. Завдяки

верифікатору eBPF-програма не може «зависнути» ядро або порушити його цілісність. Після успішної верифікації байт-код eBPF компілюється у нативні машинні інструкції цільової архітектури (JIT-компіляція, Just-In-Time Compilation), що забезпечує продуктивність, наближену до нативного коду ядра [13].

Механізм взаємодії між простором користувача та ядром детально показаний на Рис. 1.7: програма у просторі користувача (User Program) генерує BPF-байт-код та завантажує його через системний виклик до ядра, де він проходить верифікацію та виконується рушієм BPF. Результати роботи eBPF-програма повертає двома шляхами: через `perf_output` — для передачі per-event даних про кожну окрему подію в реальному часі, або через асинхронне зчитування агрегованої статистики з карт. Принципово важливо, що в ядрі eBPF-програма може підключатися до чотирьох типів точок інструментування, також відображених на Рис. 1.7: `kprobes` (динамічне перехоплення довільних функцій ядра), `uprobes` (перехоплення функцій у просторі користувача), `tracepoints` (статичні точки трасування зі стабільним ABI) та `perf_events` (апаратні та програмні лічильники продуктивності) [12].

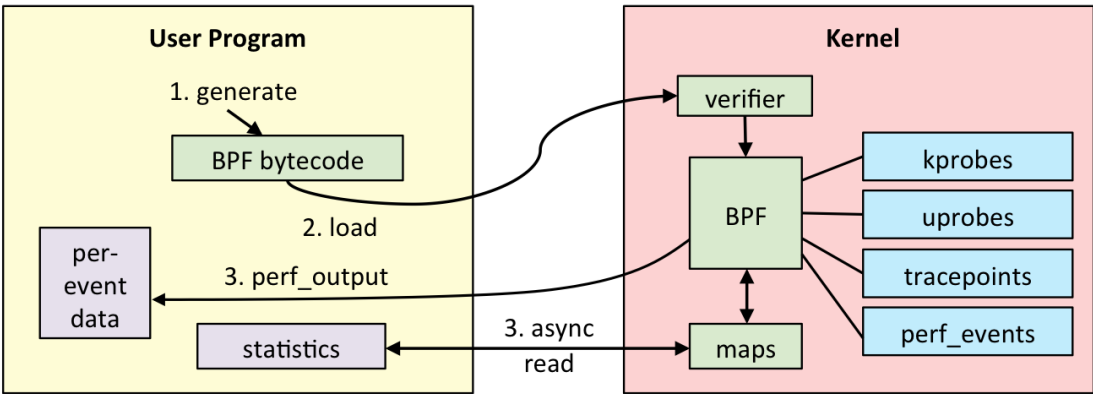


Рисунок 1.7 – Архітектура взаємодії eBPF-програми з ядром Linux: генерація байт-коду, завантаження та обмін даними [12], [23]

Критично важливим механізмом підсистеми eBPF є карти (eBPF Maps) — структури даних типу «ключ–значення», що розміщені в ядрі та забезпечують двосторонній обмін інформацією між eBPF-програмами та кодом у просторі користувача [12]. eBPF підтримує різноманітні типи карт: Hash-карти (O(1) пошук за ключем), Array-карти (пошук за індексом), LRU Hash-карти (автоматичне витіснення найдавнішого невикористовуваного запису при досягненні ліміту), Ring Buffer (ефективна передача потоків подій у простір користувача), а також більш складні структури. Саме карти є ключовим механізмом для реалізації оркестратор-керованої фільтрації: агент у просторі користувача заповнює їх контекстними даними Kubernetes, а eBPF-програми в ядрі звертаються до цих карт під час прийняття рішень щодо кожного системного виклику.

Для підключення eBPF-програм до різних точок виконання ядра Linux передбачено кілька типів хуків, що суттєво відрізняються за накладними витратами та можливостями. Kprobes — динамічні точки зондування, що підключаються до довільних функцій ядра. Хоча kprobes надзвичайно гнучкі, їхній механізм реалізації на ARM64 базується на обробці виключень (exception handling), що генерує значний overhead. Кожне спрацювання kprobe означає переривання нормального потоку виконання, збереження контексту процесора та виклик обробника — операція, надмірно ресурсномістка для IoT-пристроїв [24], [25], [26].

Альтернативою є механізм fentry/fexit (BPF Trampolines), доступний з ядра Linux 5.5 для архітектури x86_64 [25], [27]. На архітектурі ARM64 повноцінна підтримка fentry/fexit була додана у ядрі Linux 6.0 [28]. fentry/fexit використовує спеціальні BPF Trampolines — невеликі фрагменти коду, що вставляються перед точкою входу або після повернення з функції ядра без використання механізму виключень. Це робить fentry/fexit значно

					ІАЛЦ.467200.003 ПЗ	Арк.
						25
Зм.	Арк.	№ докум.	Підпис	Дата		

ефективнішими за kprobes, оскільки усувається необхідність у trap-перериваннях і забезпечується практично нульовий overhead, що має принципове значення для ресурсно-обмежених IoT-пристроїв [26]. Для ARM64-пристроїв зі старішими ядрами (до версії 6.0) необхідний резервний механізм — raw_tracepoints, що підключаються до статично визначених точок трасування у ядрі та мають суттєво нижчий overhead порівняно з kprobes [29].

Підсистема BPF LSM (Linux Security Modules), доступна з ядра Linux 5.7 при увімкненому параметрі CONFIG_BPF_LSM, дозволяє eBPF-програмам підключатися до хуків безпеки, розподілених по критичних шляхах виконання ядра [30], [31]. На відміну від спостережуваних механізмів (kprobes, tracepoints), BPF LSM дозволяє синхронно блокувати системні виклики, повертаючи код помилки -EPERM. Це відкриває можливість для превентивного enforcement-контролю. Механізм XDP (eXpress Data Path) уможливорює обробку мережевих пакетів на рівні драйвера мережевої карти до їх потрапляння в мережевий стек ядра, що забезпечує мікросекундну фільтрацію трафіку [12].

Ключова перевага eBPF перед традиційними підходами полягає в принципі «нульового копіювання» (zero-copy). Оскільки eBPF-програма виконується в ядрі, вона має прямий доступ до структур даних ядра без їх копіювання у простір користувача. Якщо логіку фільтрації реалізовано на рівні eBPF, легітимні події відкидаються до їх запису у ring buffer, що повністю унеможливорює контекстне перемикання та копіювання даних, пов'язані з цією подією. Це є фундаментальним архітектурним обґрунтуванням підходу «drop-at-source» [12], [32].

Технологія eBPF, отже, надає технічний фундамент для побудови агента безпеки, що поєднує глибоку спостережуваність на рівні ядра з мінімальним споживанням ресурсів — за умови правильного проєктування логіки фільтрації та вибору відповідних типів хуків.

Таблиця 1.2 – Порівняльний аналіз механізмів перехоплення подій у ядрі Linux

Механізм	Принцип роботи	Overhead	Блокування syscall	Стабільність ABI	Мін. версія ядра
Kprobes	Динамічна вставка breakpoint у довільну функцію ядра через trap/exception	Високий x86: ~76 нс (kprobe), ~212 нс (kretprobe) [12], ARM64: breakpoint + single-step exception [12]	Ні (тільки спостереження)	Нестабільний (залежить від внутрішніх символів)	2.6.9 [12]
Tracepoints (raw)	Статичні точки трасування; raw tracepoint hook без обробки виключень [29]	Низький на x86: ~96 нс [12]	Ні	Стабільний частина ABI ядра	4.7 raw: 4.17 [12], [29]
BPF Trampoline s (fentry / fexit)	Вставка коду без trap через BPF Trampoline [25], [26]	ARM64: «Практично нульовий overhead» [25], [27]	Частково лише через fmodify_return (з v5.7) [24]	Стабільний BTF-based API [26]	5.5 (x86-64) 6.0 (ARM64) [26], [28]
LSM hooks (BPF-LSM)	Підключення до hooks Linux Security Modules через BPF trampoline [30], [31]	ARM64: низький, використовує механізм BPF trampoline [26], [31]	Так синхронно, повертає - EPERM [30], [31]	Стабільний LSM ABI [31]	5.7 CONFIG_BPF_LSM [30]
XDP (eXpress Data Path)	Обробка пакетів на рівні драйвера NIC до входу у мережевий стек [12], [32]	Мінімальний До 24 Mpps на одне ядро [32]	Так XDP_DROP, XDP_TX, XDP_REDIRECT [32]	Стабільний гарантія стабільності API [32]	4.8 native XDP [12], [32]

1.4 Концепція контекстно-залежного придушення подій (Context-Aware Suppression)

Незважаючи на переваги технології eBPF, її застосування у середовищах Kubernetes породжує фундаментальну архітектурну проблему — «семантичну прірву» між рівнем ядра операційної системи та рівнем оркестратора. Ядро Linux принципово ізольоване від концептуального рівня оркестратора: воно оперує процесами (PID), контрольними групами (cgroups), просторами імен монтування (mount namespaces) та мережевими сокетами [12], [13]. Натомість Kubernetes організовує робочі навантаження на рівні Pod-ів, Deployment-ів, Namespace-ів та Service-ів — абстракцій, про існування яких ядро Linux не має жодного уявлення [5], [10].

Ця семантична розбіжність має практичні наслідки, що безпосередньо впливають на якість моніторингу безпеки. Типові легітимні операції Kubernetes — масштабування Deployment (запуск нових Pod-ів), оновлення конфігурацій (перезапуск контейнерів), автоматичне відновлення після збоїв (Pod eviction та restarting) — на рівні ядра виглядають як множинне породження нових процесів (execve), відкриття конфігураційних файлів (openat до /etc, /proc, /sys) та встановлення мережових з'єднань [12], [18]. Для агента безпеки, що не знає контексту оркестратора, ці абсолютно нормальні дії не відрізані від потенційно шкідливих, що призводить до генерації великої кількості хибних тривог (false positives) та «втоми від сповіщень» (alert fatigue) у командах безпеки [18], [33].

Більше того, спроба вирішити проблему хибних тривог шляхом створення статичних списків виключень у просторі користувача є принципово неефективною в динамічному Kubernetes-середовищі. Ідентифікатори процесів (PID) постійно змінюються при кожному рестарті Pod-а. Адреси контейнерів у

мережевому просторі імен змінюються при масштабуванні. Статичні whitelists не встигають за динамікою кластера, що неминуче призводить або до надлишкових сповіщень, або до прогалин у безпеці [19], [33].

Концепція «контекстно-залежного придушення подій» (Context-Aware Event Suppression) пропонує принципово інший підхід, що вирішує семантичну прірву на архітектурному рівні. Замість аналізу всіх системних викликів у просторі користувача, ця концепція передбачає:

1. Відстеження стану кластера Kubernetes у реальному часі через Kubernetes Informers [5], [10], [34];
2. Трансляцію цього стану в низькорівневі ідентифікатори ядра (cgroup IDs) [12], [13];
3. Завантаження цих ідентифікаторів у eBPF-карти [12], [13];
4. Перевірку кожного системного виклику безпосередньо в ядрі шляхом O(1) hash-lookup у карті [12], [13].

Ключовим технічним рішенням є використання ідентифікатора контрольної групи (cgroup_id) як первинного ключа ідентифікації. Функція eBPF bpf_get_current_cgroup_id() повертає ідентифікатор cgroup процесу, що ініціював системний виклик [12], [13], [35]. У Kubernetes кожен контейнер запускається у власній cgroup, що робить cgroup_id стійким та унікальним ідентифікатором навіть у середовищах з PID namespaces (де PID може дублюватись) [12], [35]. Якщо cgroup_id виявляється у whitelist-карті eBPF, системний виклик вважається легітимним і відкидається без будь-якої подальшої обробки та без запису у ring buffer — цей підхід відомий як «drop-at-source» [12], [13], [32].

Архітектурний зв'язок між Kubernetes API та eBPF-картами реалізується через механізм Kubernetes Informers. Informer — це клієнтська бібліотека Go, яка підписується на потік подій Kubernetes API Server (Pod create, Pod update,

Pod delete) та підтримує актуальний локальний кеш на вузлі [10], [34]. Реальні реалізації eBPF-агентів безпеки — зокрема KubeArmor та Tetragon — використовують саме цей механізм: їхній Informer безперервно прослуховує Kubernetes API та реагує на зміни у стані Pod-ів [18]. Коли новий Pod запускається на вузлі, Informer отримує подію, агент витягує cgroup_id відповідного контейнера та синхронно записує його у eBPF whitelist-карту. Коли Pod зупиняється, відповідний запис видаляється [18], [34]. Таким чином досягається детермінованість: стан eBPF-карт завжди відображає актуальний стан кластера з мінімальною затримкою [12], [13].

Підхід «drop-at-source» має кілька принципових переваг над традиційними моделями, описаними у підрозділах 1.2 та 1.3. По-перше, він кардинально зменшує обсяг подій, що передаються у ring buffer, зберігаючи його виключно для аномальних та підозрілих дій. Це усуває проблему backpressure та запобігає втраті критичних сповіщень — явища, яке Aquila характеризує як ситуацію, коли «зловмисник може навмисно заповнити систему легітимними викликами, щоб приховати рідкісні, але критичні події» [33]. Математичне обґрунтування цієї загрози за моделлю M/M/1/K наведено у підрозділі 1.2 [20]. По-друге, цей підхід усуває витрати на контекстне перемикання для легітимних подій — якщо подія відкидається в ядрі, жодного переходу kernel/user space не відбувається взагалі [12], [32]. По-третє, логіка прийняття рішень є детермінованою та надзвичайно швидкою, що забезпечує мінімальний overhead для захищеного застосунку [12], [13].

Вирішення семантичної прірви через механізм оркестратор-керованого придушення подій є центральною ідеєю системи безпеки OIKESA (Orchestrator-Aware In-Kernel Event Suppression Agent) та відрізняє її від існуючих аналогів, аналіз яких проведено у підрозділах 1.2–1.3.

Таблиця 1.3 – Зведені технічні вимоги до системи OIKESA

Вимога	Параметр / Ціль	Обґрунтування
Архітектура CPU	ARM64 (Cortex-A57, ARMv8+)	Переважна більшість IoT/Edge-пристроїв
CPU Overhead (типовий режим)	< 5% при 1 000 execve/c	Залишити $\geq 95\%$ CPU для бізнес-навантаження
CPU Overhead (пікове навантаження)	< 15% при adversarial flooding	Не допустити деградації при атаці
RAM (user-space агент)	< 70 МБ	Обмежений бюджет пам'яті Edge-вузла
RAM (BPF Maps)	< 5 МБ	Лімітований розмір карт (~4096 записів)
Механізм hook	fentry/fexit (primary), raw_tracepoints (fallback), BPF-LSM (enforcement)	Оптимальний overhead; підтримка ядер ≥ 5.5
Мін. версія ядра Linux	5.10 LTS	Широка підтримка на ARM64 IoT-платформах
Ідентифікація контейнера	cgroup_id (bpf_get_current_cgroup_id())	Стійкий до PID namespaces ідентифікатор
Складність пошуку	O(1) — BPF_MAP_TYPE_LRU_HASH	Детермінованість при будь-якому навантаженні
Suppression efficiency	$\geq 95\%$ легітимних подій	Усунення backpressure у ring buffer
Latency alert	< 1 мс	Своєчасне виявлення загроз

1.5 Постановка задачі на проєктування системи OIKESA (Orchestrator-Aware In-Kernel Event Suppression Agent)

Аналіз проблемної області, архітектурних особливостей IoT-середовищ та існуючих рішень дозволяє сформулювати мету та завдання дипломного проєктування з необхідним рівнем технічної деталізації.

Метою дипломної роботи є проєктування, розробка та емпірична оцінка системи OIKESA — спеціалізованого агента безпеки для кластерів K3s у IoT-середовищах, що реалізує механізм контекстно-залежного придушення подій на рівні ядра Linux з урахуванням стану оркестратора Kubernetes та мінімізує споживання ресурсів центрального процесора та оперативної пам'яті.

Об'єктом дослідження є система OIKESA та процеси забезпечення кібербезпеки в ресурсно-обмежених Kubernetes IoT-середовищах на базі архітектури ARM64.

Предметом дослідження є архітектурні рішення, алгоритми та механізми реалізації контекстно-залежного in-kernel придушення подій на основі технології eBPF з використанням Kubernetes Informers для синхронізації стану оркестратора.

Для досягнення поставленої мети необхідно вирішити такі задачі:

1. Провести аналіз існуючих рішень у сфері Cloud-Native безпеки із застосуванням eBPF та їх придатності для IoT-середовищ; визначити архітектурні прогалини та технічні вимоги до нової системи.
2. Спроекувати архітектуру OIKESA, що включає eBPF data plane (набір хуків типів fentry/fexit, raw_tracepoint та BPF LSM) та XDP для мережевої фільтрації, user-space control plane на мові Go з інтеграцією Kubernetes Informers, та механізм двосторонньої синхронізації стану через eBPF-карти.

3. Реалізувати стек хуків агента: fentry з fallback на raw_tracepoint для ядер ARM64 до версії 6.0 та x86_64 до версії 5.5, а також LSM та XDP-фільтри для мережевого захисту IPv4/IPv6.
4. Реалізувати механізм захисту від LRU flood attack через Go-side лічильник записів у blocklist-карті типу BPF_MAP_TYPE_LRU_HASH, що зберігає резервний буфер слотів проти витіснення легітимних записів зловмисником.
5. Реалізувати TTL-механізм для pending cgroups: cgroup, що не отримала whitelist entry протягом встановленого інтервалу, вважається недійсною — це унеможлиблює атаки типу stale pending bypass.
6. Розробити систему eBPF-лічильників метрик для моніторингу, що охоплює ключові події придушення та блокування.
7. Розгорнути тестовий стенд на базі кластера K3s на ARM64-платформі (NVIDIA Jetson Nano) та провести комплексне тестування: функціональне (correctness), продуктивності (overhead analysis), безпекове (threat simulation) та порівняльне (benchmark проти Falco та Tetragon).
8. Оцінити ефективність системи за ключовими метриками, визначеними: $\text{suppression efficiency} \geq 95\%$, $\text{CPU overhead} < 5\%$ при типовому навантаженні та $< 15\%$ при adversarial flooding, затримка виявлення загроз < 1 мс.

Технічні вимоги до системи OIKESA включають: підтримку архітектур ARM64 та x86_64; сумісність з K3s та стандартним Kubernetes; мінімальну підтримувану версію ядра Linux — 5.10 LTS як базову платформу з підтримкою BPF LSM (доступного з 5.7) та raw_tracepoints як резервного механізму для ARM64-пристроїв, де fentry/fexit доступний лише з ядра 6.0; реалізацію мовою Go з використанням бібліотеки cilium/ebpf та компілятора Clang/LLVM; принцип CO-RE (Compile Once — Run Everywhere) для портативності між різними версіями ядер без перекомпіляції на цільовому пристрої.

Робота полягає у розробці та дослідженні архітектурного механізму «Zero-Evaluation Suppression Layer» — підходу, за яким eBPF-програма виконує виключно $O(1)$ lookup у whitelist-карті без евристичної оцінки або ML-аналізу, що забезпечує детермінованість та мінімальний латентний overhead; механізму «Lifecycle-Coupled Kernel Garbage Collection», що зв'язує видалення записів з eBPF-карт із подіями Pod lifecycle Kubernetes Informers; а також дослідженні архітектури «Edge-Native Control Plane Deconstruction» — мінімалізації user-space складової до функції синхронізатора стану без власного рушія правил.

					ІАЛЦ.467200.003 ПЗ	Арк.
						34
Зм.	Арк.	№ докум.	Підпис	Дата		

ВИСНОВОК ДО РОЗДІЛУ

У першому розділі проведено комплексний аналіз предметної області дослідження, що охоплює технологічний контекст, проблематику, математичне обґрунтування накладних витрат системних викликів (формула (1.1)), аналіз явища backpressure у кільцевих буферах (формула (1.2)), архітектуру підсистеми eBPF та концепцію контекстно-залежного придушення подій — теоретичне підґрунтя системи OIKESA.

Встановлено, що розвиток Cloud-Native парадигми на периферію мережі, зумовлена зростанням кількості активних IoT-з'єднань, призвела до широкого впровадження легковагових Kubernetes-дистрибутивів на ARM64-платформах. Дистрибутив K3s, консолідуючи всі компоненти кластера в єдиний бінарний файл розміром менше 100 МБ та замінюючи etcd на SQLite, став стандартом для периферійного розгортання на пристроях з обмеженими ресурсами, зокрема NVIDIA Jetson Nano.

Математично обґрунтовано (формула (1.1)), що сукупні накладні витрати традиційних user-space агентів на ARM64-вузлах є критичними: компонент $T_{context\ switch}$ включає зміну рівня привілеїв процесора, збереження та відновлення стану реєстрів, а також примусове скидання TLB через механізм КРТІ, що в сукупності здатне сягати від кількох відсотків до понад 17% ресурсів CPU залежно від інтенсивності системних викликів. Встановлено (формула (1.2)), що при перевищенні швидкості генерації подій λ над швидкістю їх обробки μ ймовірність P_{drop} втрати критичної події прямує до одиниці, роблячи традиційну архітектуру ненадійною в умовах adversarial flooding.

Проведений аналіз архітектури eBPF підтвердив технологічну придатність цієї підсистеми для побудови ефективного агента безпеки: механізм fentry/fexit забезпечує практично нульовий overhead порівняно з

					ІАЛЦ.467200.003 ПЗ	Арк.
						35
Зм.	Арк.	№ докум.	Підпис	Дата		

kprobes на ARM64; BPF LSM дозволяє синхронне блокування системних викликів шляхом повернення коду помилки -EPERM; O(1) hash-lookup у картах типу BPF_MAP_TYPE_LRU_HASH гарантує детермінованість та мінімальну затримку при перевірці whitelist.

Концепцію «drop-at-source», краще реалізувати через механізм Kubernetes Informers з ключем cgroup_id в eBPF-картах, що архітектурно обґрунтованою відповіддю на «семантичну прірву» між рівнем ядра та рівнем оркестратора. Вона формує теоретичне та технічне підґрунтя для системи OIKESA, завдання та вимоги до якої сформульовано в підрозділі 1.5 (таблиця 1.3).

					ІАЛЦ.467200.003 ПЗ	Арк.
						36
Зм.	Арк.	№ докум.	Підпис	Дата		

РОЗДІЛ 2

АНАЛІЗ ІСНУЮЧИХ РІШЕНЬ ТА ОБҐРУНТУВАННЯ ВИБОРУ

Забезпечення безпеки в хмарно-орієнтованих (cloud-native) екосистемах, що розгортаються на периферії мережі, вимагає подолання семантичного розриву (semantic gap) — нездатності ядра операційної системи ідентифікувати низькорівневі системні виклики в контексті високорівневих абстракцій оркестратора: Pod, Namespace, Deployment [18], [33]. Традиційні архітектури моніторингу безпеки, розроблені для потужних і стабільних центрів обробки даних, демонструють принципову непридатність при міграції в середовища IoT та на периферійні обчислювальні вузли, зокрема одноплатні комп'ютери архітектури ARM64 або промислові IoT-шлюзи, що призводить до деградації продуктивності [5], [15], [18].

Ядро Linux оперує виключно низькорівневими ідентифікаторами (PID, cgroup_id), тоді як політики безпеки в Kubernetes формулюються у термінах Pod, Namespace, ServiceAccount. Зростання щільності IoT-навантажень та жорсткі апаратні обмеження Edge-пристроїв унеможливлюють застосування класичних інструментів безпеки, що вимагає принципово нових архітектурних підходів, заснованих на внутрішньоядерному придушенні подій [33]. У цьому розділі здійснюється аналіз існуючих методів моніторингу безпеки, порівняння провідних Open-Source рішень та обґрунтування концепції Orchestrator-Aware In-Kernel Event Suppression як архітектурно обґрунтованого підходу для обмежених IoT-середовищ.

					ІАЛЦ.467200.003 ПЗ	Арк.
						37
Зм.	Арк.	№ докум.	Підпис	Дата		

2.1 Класифікація методів моніторингу безпеки в контейнеризованих середовищах

Методи моніторингу безпеки на рівні виконання (runtime security) пройшли шлях від просторових агентів до внутрішньоядерних механізмів, намагаючись збалансувати глибину видимості з рівнем накладних витрат. Розвиток цих методів відображає прагнення індустрії до мінімізації впливу на корисне навантаження при одночасному підвищенні точності виявлення загроз та зменшенні часу реакції на інциденти [18], [33]. Сучасна таксономія включає три покоління інструментів: традиційні агенти user-space (auditd, ptrace), статичні ядерні механізми (LSM) та динамічні eBPF-рішення. Розуміння архітектурних компромісів кожного покоління є обов'язковою передумовою для обґрунтованого вибору технологій у рамках системи OIKESA.

2.1.1 Традиційні механізми: auditd та ptrace

Першим широко застосовуваним інструментом динамічного моніторингу є механізм ptrace (process trace). Він дозволяє процесу-трейсеру перехоплювати кожен системний виклик цільового процесу з призупиненням його виконання для аналізу [36]. В умовах сучасних Kubernetes-кластерів, де на одному вузлі одночасно виконуються сотні контейнерів, що генерують мільйони системних викликів, ptrace стає архітектурним вузьким місцем. При використанні PTRACE_SYSCALL кожен системний виклик цільового процесу генерує два зупинення: syscall-enter-stop та syscall-exit-stop. Кожне таке зупинення включає переходи від tracee до ядра та навпаки, тобто повний цикл перехоплення одного системного виклику потребує щонайменше чотирьох переходів між просторами виконання [36], [37]. На архітектурах ARM64 часті перемикання контексту призводять до накладних витрат через необхідність збереження чи відновлення

стану регістрів та, у конфігураціях з увімкненим КРТІ, можливої інвалідації TLB-записів. Як показують дослідження, перемикання регістра таблиці сторінок (TTBR) на ARM64 займає приблизно 52 ns, тоді як на x86 ця операція досягає 235 ns, що робить ARM64 більш стійкою до витрат КРТІ [38]. Водночас сучасні ARM64-процесори підтримують механізм ASID (Address Space Identifiers), який дозволяє тегувати TLB-записи ідентифікатором адресного простору та уникати повного очищення TLB при перемиканнях контексту між різними процесами [39]. Однак навіть за наявності ASID, увімкнення КРТІ вимагає маніпуляцій з таблицями сторінок при кожному вході та виході з ядра, що зумовлює необхідність часткової інвалідації TLB незалежно від ASID [40].

Емпіричні вимірювання показують, що overhead ptrace може бути значним: дослідження Kharbanda та Fonseca зафіксувало, що при записі системних викликів через ptrace-пастки виникають чотири перемикання контексту між tracee-процесом та tracer-процесом на кожен виклик, і ці перемикання є основним вузьким місцем продуктивності [37]. Overhead зумовлений кількістю контекстних переходів, а не складністю системного виклику. Використання ptrace у поєднанні з seccomp дозволяє зменшити кількість переходів для небезпечних syscall, оскільки безпечні системні виклики обробляються безпосередньо в ядрі через BPF-фільтр без передачі управління tracing-процесу, — однак навіть у цьому випадку overhead залишається неприйнятним для ресурсно-обмежених IoT-вузлів [41]. Таким чином, ptrace є абсолютно неприйнятним як архітектурна основа системи моніторингу для Edge-середовищ.

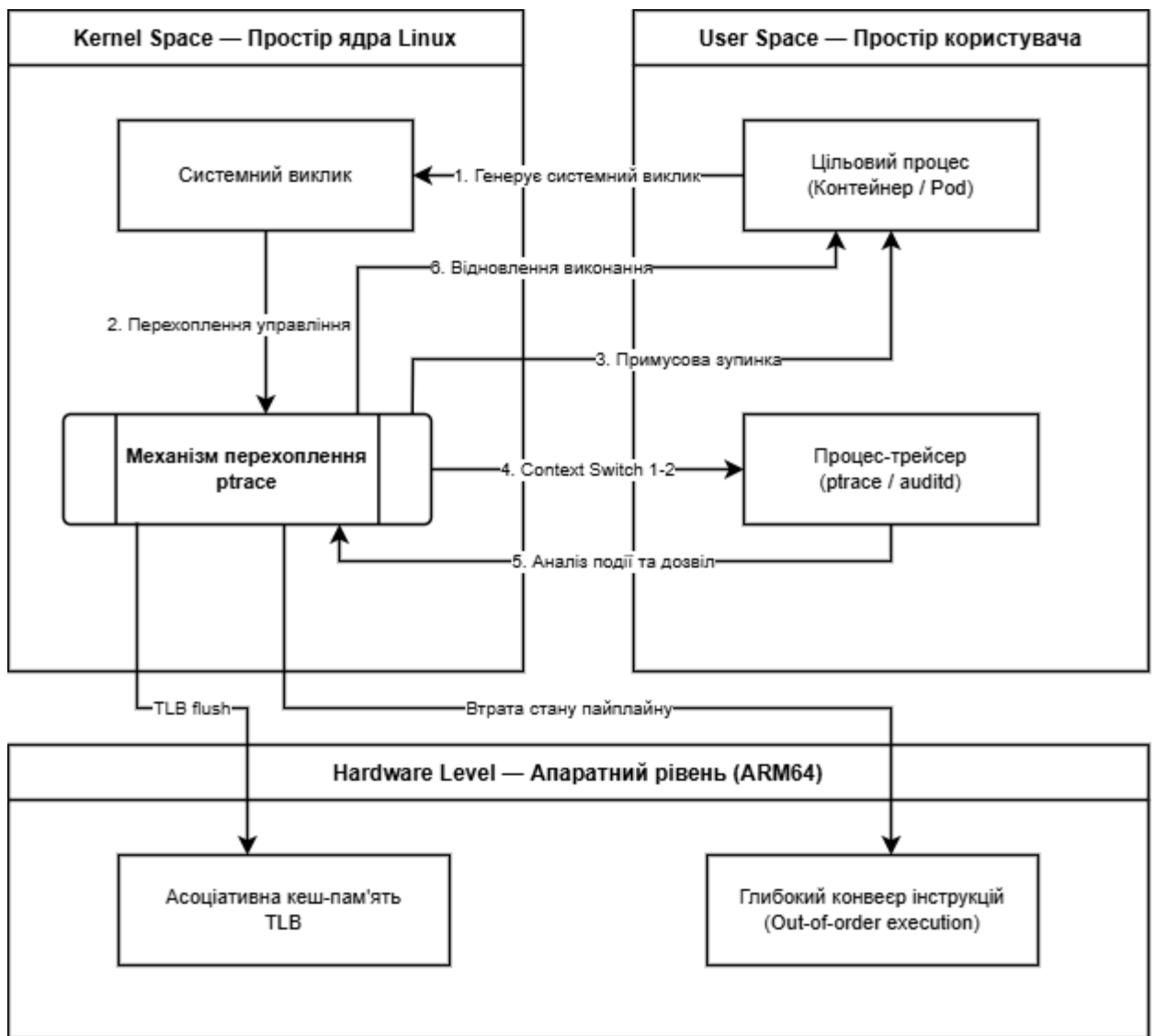


Рисунок 2.1 — Механізм перехоплення системних викликів через ptrace та джерела накладних витрат на системному й апаратному рівнях

Підсистема аудиту ядра Linux (auditd) реалізує перехоплення подій безпосередньо в ядрі з передачею записів демону user-space через сокет Netlink [36]. Auditd забезпечує регуляторну відповідність (HIPAA, PCI DSS, SOC2) та є стандартним компонентом більшості корпоративних Linux-дистрибутивів. Однак для застосування в контейнеризованих IoT-середовищах він є непридатним: auditd генерує масивні обсяги неструктурованих логів без

прив'язки до контексту Kubernetes, а для відновлення цього контексту user-space змушений парсити /proc та виконувати запити до API-сервера, що збільшує Time-to-Detect у рази. Крім того, у середовищах із великою кількістю контейнерів значна частина подій auditd генерується легітимними системними операціями kubelet та CNI-плагінів, що помітно ускладнює фільтрацію реальних загроз [18], [33].

2.1.2 Модулі безпеки Linux (LSM) та BPF-LSM

Linux Security Modules (LSM) надають хуки в критичних точках ядра для реалізації мандатного контролю доступу (MAC). Найбільш поширені реалізації — AppArmor та SELinux — дозволяють синхронно блокувати несанкціоновані дії ще до їхнього виконання, що принципово відрізняє LSM від детективних підходів [31]. У контексті безпеки контейнерів LSM-профілі, прив'язані до конкретного контейнера при запуску, забезпечують ефективну ізоляцію від атак типу container escape та privilege escalation. Синхронність прийняття рішень (у тому самому потоці ядра, що виконує системний виклик) є ключовою перевагою LSM для сценаріїв превентивного захисту.

Фундаментальним обмеженням традиційних LSM є їхня статичність: профілі жорстко задаються при запуску контейнера та не можуть динамічно адаптуватися до змін стану кластера без повного перезапуску навантажень [31]. Це означає, що будь-яка зміна політики безпеки — наприклад, реагування на нову загрозу або зміну поведінки застосунку — потребує регенерації профілю та повторного деплою контейнера, що є неприйнятним у динамічних середовищах. В середовищі Kubernetes, де Pod-и існують від секунд до хвилин, статичні профілі AppArmor стають операційним тягарем для адміністраторів, оскільки вимагають постійної синхронізації з фактичним станом робочих навантажень та їх поведінкою. Крім того, такі профілі часто або надмірно

обмежують застосунок (що призводить до збоїв), або, навпаки, є занадто дозвільними через складність їх точного налаштування.

Револьюційним розвитком стала поява BPF-LSM, що дозволяє eBPF-програмам динамічно підключатися до LSM-хуків, поєднуючи безпеку мандатного контролю з повною гнучкістю eBPF [30], [31]. На відміну від класичних LSM, логіка перевірки доступу в BPF-LSM визначається не статичними політиками, а виконуваним кодом, який може бути змінений або перезавантажений у run-time без зупинки системи. Це відкриває можливість реалізації контекстно-залежних політик, що враховують поточний стан процесу, контейнера або усього кластера.

BPF-LSM є ключовою складовою архітектури OIKESA: він забезпечує синхронне блокування аномальних дій безпосередньо в ядрі без перемикань контексту, зберігаючи при цьому здатність динамічно оновлювати правила через eBPF Maps. Зокрема, eBPF Maps виступають як високопродуктивне сховище стану, що дозволяє змінювати політики в реальному часі (наприклад, оновлювати списки дозволених системних викликів або ресурсів) без перекомпіляції чи повторного завантаження програми. Результатом є органічне поєднання низької латентності, характерної для in-kernel механізмів, із гнучкістю, притаманною user-space рішенням.

2.1.3 Парадигма eBPF: типи хуків та їхня ефективність на ARM64

Технологія extended Berkeley Packet Filter (eBPF) докорінно змінила підхід до системної обсерваційності (observability) та безпеки, дозволивши виконувати верифікований байт-код безпосередньо у просторі ядра без необхідності модифікувати ядро або завантажувати традиційні модулі (LKM) [13]. Традиційні loadable kernel modules є динамічно завантажуваними

					ІАЛЦ.467200.003 ПЗ	Арк.
						42
Зм.	Арк.	№ докум.	Підпис	Дата		

розширеннями ядра, які працюють із повними привілеями kernel-space та можуть змінювати його поведінку. Хоча це забезпечує високу гнучкість, LKM не мають вбудованих механізмів безпеки: помилка в коді або зловмисна логіка можуть призвести до краху системи, пошкодження пам'яті або повної компрометації ядра. Крім того, розробка та супровід LKM потребують тісної прив'язки до версії ядра, що ускладнює їх застосування в динамічних середовищах. На відміну від LKM, eBPF гарантує безпеку виконання через статичний верифікатор, що перевіряє кожен програму на відсутність безмежних циклів та несанкціонованого доступу до пам'яті перед завантаженням [9]. Таким чином, eBPF фактично обмежує можливості програм до безпечної підмножини операцій, зберігаючи при цьому можливість розширення функціональності ядра без ризику його дестабілізації. Після успішної верифікації байт-код JIT-компілюється у нативні машинні інструкції ARM64, досягаючи продуктивності, наближеної до нативного коду ядра.

В контексті IoT-вузлів принципово важливим є вибір механізму підключення eBPF-програм. Kprobes реалізують перехоплення через генерацію виключення процесора (trap/exception): при спрацюванні зонда виконання ядра переривається, процесор генерує виключення типу BRK на ARM64, зберігається стан регістрів, виконується eBPF-програма, після чого відновлюється оригінальний потік [42], [43]. Дослідження Gebai та Dagenais кількісно підтверджує, що використання kprobes замість статичних tracerpoints збільшує латенцію обробки події приблизно на 30–35% (92 ns проти 123 ns) на x86-64, причому основними факторами є додаткові L1-кешпромахи та збільшення кількості інструкцій [43]. На ARM64 використання kprobe пов'язане з вищим overhead порівняно зі статичними трейспоінтами, оскільки неоптимізовані реалізації (без OPTPROBES) вимушені використовувати механізм trap замість більш ефективного trampoline-підходу. Офіційна

документація ядра Linux зазначає, що оптимізація kprobe через jump-based trampoline вимагає виконання певних умов, і у випадку їх невиконання система повертається до повільнішого breakpoint-механізму [42]. Дослідження Gebai та Dagenais підтверджують, що навіть на x86 перехід від статичних трейспоінтів до Kprobes додає помітний overhead [43].

Натомість механізм fentry/fexit (BPF Trampolines, доступний для x86_64 починаючи з ядра 5.5, та для ARM64 починаючи з ядра 6.0) вставляє нативні JIT-скомпільовані інструкції безпосередньо на початок функцій ядра без генерації виключень, використовуючи NOP-слоти, що залишаються компілятором [25], [28]. Завдяки відсутності trap-механізму fentry/fexit досягає значно нижчого overhead порівняно з kprobes. Eunomia підтверджує, що fentry/fexit програми виконуються швидше за kprobes завдяки механізму BPF trampoline [44]. Як показано на Рис. 2.2, kprobes використовують механізм виключень процесора, тоді як fentry/fexit реалізують пряме підключення через trampolines без переривання виконання.

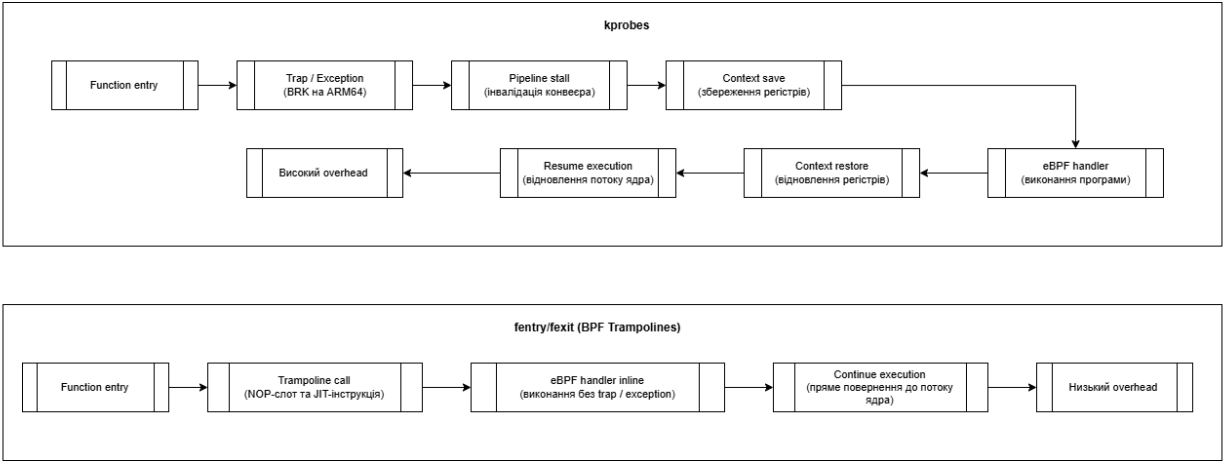


Рисунок 2.2 – Порівняння механізмів перехоплення eBPF: kprobes та fentry/fexit

Механізм XDP дозволяє обробляти мережеві пакети на рівні драйвера мережевої карти, до їхньої передачі у мережевий стек ядра [32]. Для OIKESA XDP є складовою захисту від DDoS-атак типу adversarial flooding. Поєднання fentry (для перехоплення системних викликів), BPF-LSM (для синхронного блокування) та XDP (для мережевого захисту) формує триярусний стек захисту OIKESA [32].

2.2 Порівняльний аналіз провідних Open-Source рішень безпеки Kubernetes

Сучасний ринок Cloud-Native безпеки пропонує чотири домінантних open-source рішення на базі eBPF: Falco, Cilium Tetragon, KubeArmor та Tracex. Кожна система визначає власну архітектуру обробки подій та фільтрації, що безпосередньо впливає на її придатність для Edge-розгортань з жорсткими ресурсними обмеженнями [18]. Зокрема, відмінності проявляються у способі збору телеметрії, рівні агрегації подій (kernel-space чи user-space) та підходах до застосування політик безпеки. Ретельний порівняльний аналіз є необхідним для виявлення архітектурного пробілу, який заповнює система OIKESA. Визначальними критеріями оцінювання є: рівень, на якому відбувається фільтрація, обсяг CPU overhead, RAM footprint та наявність механізму Lifecycle-Coupled GC для eBPF Maps. Фільтрація на рівні ядра (in-kernel filtering) дозволяє відкидати нерелевантні події ще до їх передачі у user-space, що знижує витрати на обробку та передачу даних. Натомість user-space фільтрація створює додаткове навантаження через копіювання подій та обробку в окремих процесах. Показник CPU overhead є особливо важливим для ARM64 Edge-вузлів, де обчислювальні ресурси обмежені, а надлишкове навантаження може впливати на основні робочі навантаження. RAM footprint визначає обсяг пам'яті, необхідний для зберігання стану, буферів подій та eBPF Maps, і є

					ІАЛЦ.467200.003 ПЗ	Арк.
Зм.	Арк.	№ докум.	Підпис	Дата		45

важливим фактором для IoT-пристроїв із обмеженою пам'яттю. Окремої уваги заслуговує механізм Lifecycle-Coupled Garbage Collection (GC) для eBPF Maps — підхід, за якого життєвий цикл даних у Map жорстко прив'язаний до життєвого циклу сутностей Kubernetes (наприклад, Pod-ів або контейнерів). Це означає, що при завершенні або видаленні Pod-а відповідні записи автоматично очищаються, запобігаючи накопиченню застарілих (stale) даних у ядрі. За відсутності такого механізму Maps можуть неконтрольовано зростати, що призводить до витоків пам'яті та деградації продуктивності — особливо у ефемерних середовищах, де об'єкти створюються і знищуються з високою частотою.

2.2.1 Falco: стандарт де-факто для хмарного аудиту

Falco (CNCF Graduated Project) є одним із найбільш широко розгорнутих інструментів виявлення загроз у хмарних середовищах [45], [46], [47]. Його архітектура передбачає розділення етапів збору подій та їх подальшого аналізу. На етапі збору eBPF-зонд (або kernel module у старіших конфігураціях) перехоплює системні виклики (наприклад, `execve` — запуск процесу, `openat` — відкриття файлу, `connect/bind` — мережеві операції) [48] та записує їх у кільцевий буфер. Ring buffer є високопродуктивною структурою даних у пам'яті ядра, що дозволяє передавати потік подій у user-space без блокувань і з мінімальними затримками [49]. Далі, на етапі evaluation, рушій правил у просторі користувача десеріалізує отримані події та перевіряє їх на відповідність декларативним YAML-правилам, які описують шаблони підозрілої поведінки [50], [51].

Falco активно підтримується спільнотою, має обширну бібліотеку готових правил, узгоджених із моделлю загроз MITRE ATT&CK, яка є відкритою базою знань про тактики та техніки атак, що використовуються

реальними зловмисниками. Вона структурує поведінку атакуючих (зокрема виконання довільного коду, ескалацію привілеїв, закріплення в системі та латеральне переміщення) у вигляді стандартизованої матриці, що дозволяє формалізувати правила виявлення та зіставляти їх із відомими сценаріями атак [50]. Система підтверджено успішно розгортається на ARM64 IoT-вузлах із використанням легковагових Kubernetes-дистрибутивів, таких як K3s [5], [9], [10], [15], що спрощує її впровадження у розподілених середовищах.

Проте архітектурним обмеженням Falco є лише часткова фільтрація на рівні ядра. Falco використовує механізм tail call в eBPF, завдяки якому події, що не відповідають жодній audit policy, не досягають Audit Program і не передаються далі для повної обробки [49]. Разом із тим, події, що потрапляють під активну policy, повністю обробляються в user-space: серіалізуються, передаються через межу kernel/user space та аналізуються рушієм правил, що формує додаткове навантаження на процесор [52], [53]. При цьому hot path обробки подій у Falco є однопоточним, що додатково обмежує можливості масштабування під навантаженням [52].

За результатами порівняльного дослідження [18] в мережевому бенчмарку, Falco демонструє overhead 19,93% без застосування policy та 24,77% з активованою policy. У процесному бенчмарку показники суттєво нижчі — 8,49% та 9,96% відповідно. Важливо зазначити, що ці вимірювання проводились у хмарно-нативному середовищі на x86-архітектурі (Intel Xeon Gold 6342, 32 ГБ RAM), а не на ARM64 IoT-вузлах, де характеристики процесора суттєво відрізняються. Для наочного представлення процесу обробки подій у Falco, на Рис. 2.3 наведено відповідну архітектурну схему.

Крім того, Falco є системою виключно виявлення (detection-only) і не забезпечує механізмів активного запобігання (prevention). Іншими словами, система здатна зафіксувати та повідомити про підозрілу подію, але не може

синхронно заблокувати її виконання на рівні ядра; для цього необхідна інтеграція з додатковими компонентами або зовнішніми механізмами реагування. Це збільшує затримку між виявленням загрози та реакцією — що є принциповим недоліком у сценаріях, де атаки мають бути зупинені негайно.

Обробка контексту Kubernetes у Falco здійснюється асинхронно через компонент k8smeta-плагіна, який збагачує події інформацією про Pod, Namespace та інші об'єкти. k8smeta-плагін отримує метадані з Kubernetes API-сервера, кешує їх у user-space та зіставляє з подіями, отриманими з ядра. Це дозволяє інтерпретувати низькорівневі системні виклики у термінах Kubernetes-сутностей, підвищуючи зрозумілість та цінність подій для аналізу. Однак, оскільки це збагачення відбувається поза критичним шляхом обробки, виникає часовий розрив між подією та доступністю її повного контексту. У динамічних середовищах, де Pod-и швидко створюються та видаляються, це може призводити до некоректної кореляції поді.

Таким чином, попри зрілість, широку підтримку та багату екосистему правил, архітектура Falco має суттєві обмеження в умовах IoT-інфраструктури, де першочерговими є мінімальні накладні витрати, ефективне використання ресурсів та здатність до синхронного реагування на загрози.

					ІАЛЦ.467200.003 ПЗ	Арк.
Зм.	Арк.	№ докум.	Підпис	Дата		48

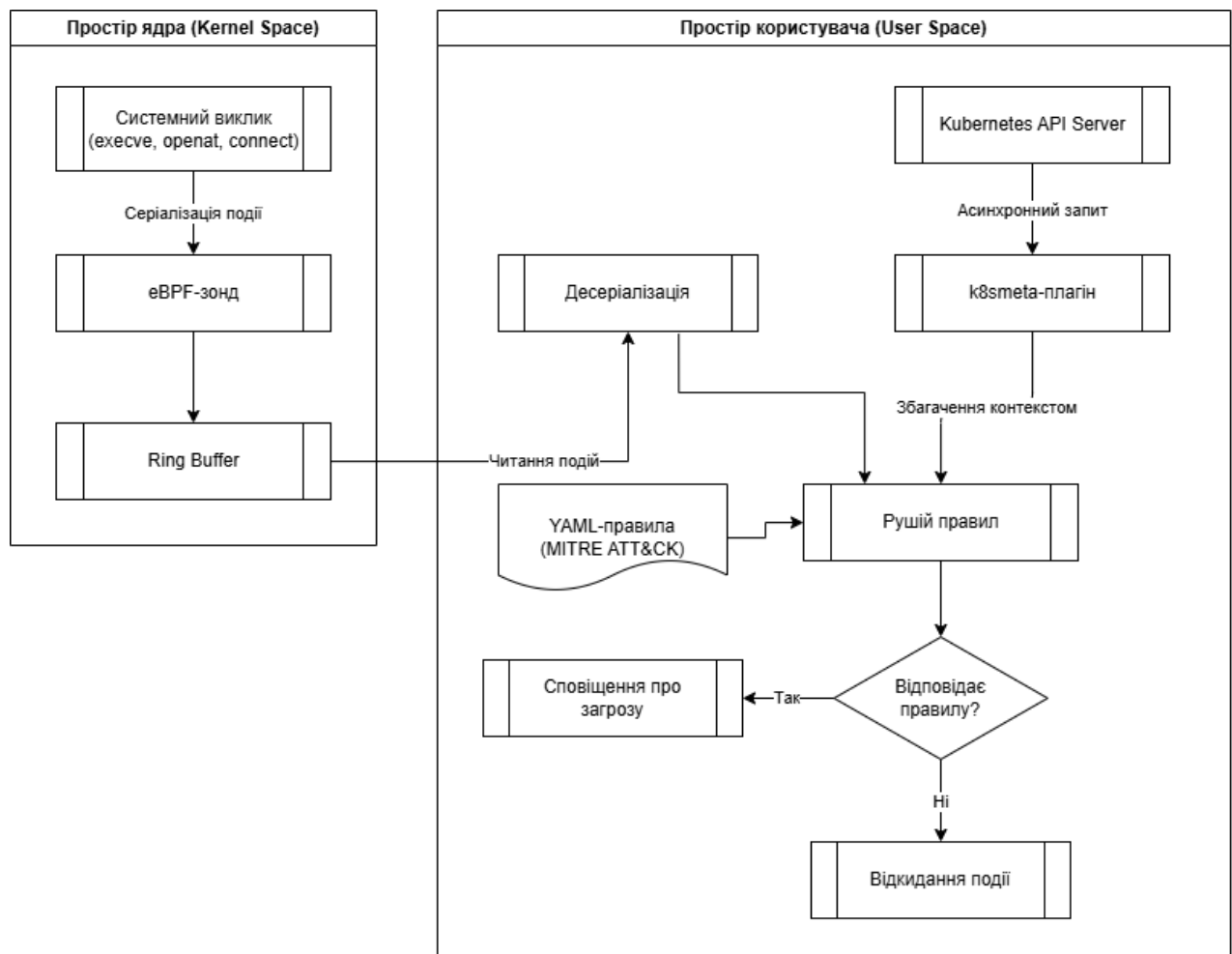


Рисунок 2.3 – Архітектура роботи Falco

2.2.2 Cilium Tetragon: превентивна безпека в ядрі

Tetragon є субпроектом Cilium і офіційним проектом CNCF, що представляє якісний стрибок у розвитку eBPF-безпеки, впроваджуючи парадигму внутрішньоядерного виконання (in-kernel enforcement). На відміну від класичних систем моніторингу, які лише спостерігають та аналізують події після їх виникнення, Tetragon переносить логіку прийняття рішень безпосередньо в ядро, що дозволяє не лише виявляти, але й миттєво запобігати небажаним діям [54].

Ключова інновація полягає у використанні механізму LSM BPF — розширення підсистеми LSM, яке дозволяє eBPF-програмам підключатися до точок контролю доступу в ядрі шляхом runtime instrumentation LSM-хуків для реалізації MAC (Mandatory Access Control) та Audit-політик [55]. Логіка цих політик задається через спеціальний Kubernetes-ресурс — TracingPolicy CRD (Custom Resource Definition), який дозволяє декларативно описувати правила моніторингу та блокування. Важливо зазначити, що Tetragon підтримує широкий спектр типів хуків — kprobes, tracepoints, uprobes та LSM BPF — а не обмежується виключно fentry/fexit [55].

eBPF-програми Tetragon здатні приймати рішення про блокування безпосередньо в ядрі двома способами: шляхом надсилення сигналу завершення процесу (SIGKILL) або шляхом перевизначення повернутого значення виклику (override), що дозволяє повернути код помилки замість виконання функції [56]. Важливо розуміти різницю між цими механізмами: надсилення SIGKILL завершує процес синхронно, але не гарантує, що поточна операція (наприклад, запис у файл) не буде виконана — для повного блокування операції необхідно комбінувати Signal та Override actions [56]. Такий підхід принципово відрізняється від detection-only систем, забезпечуючи властивості превентивної безпеки (prevention-first model) [54].

Для забезпечення коректної прив'язки подій до Kubernetes-контексту Tetragon використовує OCI хуки та NRI (Node Resource Interface) — механізм інтеграції containerd з зовнішніми плагінами [57]. OCI (Open Container Initiative) визначає стандарти запуску контейнерів, тоді як NRI надає механізм інтеграції з runtime для отримання інформації про створення, запуск і завершення контейнерів. На основі цих подій Tetragon синхронізує ідентифікатори cgroup з метаданими Kubernetes і зберігає їх у eBPF Maps. Це дозволяє eBPF-програмам у ядрі миттєво визначати, якому саме Pod-у належить процес, без необхідності

					ІАЛЦ.467200.003 ПЗ	Арк.
Зм.	Арк.	№ докум.	Підпис	Дата		50

звернення до user-space, забезпечуючи синхронну та точну кореляцію подій [57].

За даними Her Jin та інших [18], Tetragon демонструє найнижчий CPU overhead при моніторингу без policy (3,14%), однак при увімкненій мережевій policy цей показник зростає до 72,92%. Незважаючи на ефективну обробку подій у Data Plane, Tetragon має значно складніший Control Plane. Data Plane відповідає за безпосередню обробку подій у ядрі, тоді як Control Plane включає компоненти керування, зберігання стану та взаємодії з іншими системами. Для підтримки розширеного кешу станів процесів, gRPC-серверів для streaming подій безпеки (за замовчуванням localhost:54321) [58], а також механізмів аудиту на кожному вузлі необхідні додаткові обчислювальні ресурси та оперативна пам'ять [18].

Для IoT-вузлів з 512 МБ – 1 ГБ оперативної пам'яті, де легковагові компоненти, такі як K3s, container runtime та бізнес-логіка вже конкурують за ресурси, значний footprint Control Plane є фактором ризику. У таких умовах навіть додаткові десятки мегабайт можуть призвести до деградації продуктивності або нестабільності системи.

Додатковим фактором ризику є використання eBPF Maps для зберігання деревоподібних структур предків процесів (process ancestry trees). Ці структури відображають ієрархію процесів та використовуються для більш точного аналізу поведінки, наприклад, виявлення аномальних ланцюгів виконання. Проте їх розмір може зростати з часом, особливо у довготривалих Edge-розгортаннях із високою динамікою процесів. Оскільки очищення таких структур не завжди жорстко прив'язане до життєвого циклу контейнерів, виникає ризик накопичення застарілих записів, що ускладнює прогнозування використання пам'яті та підвищує ймовірність спрацювання OOM-Killer —

підсистеми ядра Linux, яка примусово завершує процеси при нестачі пам'яті [18].

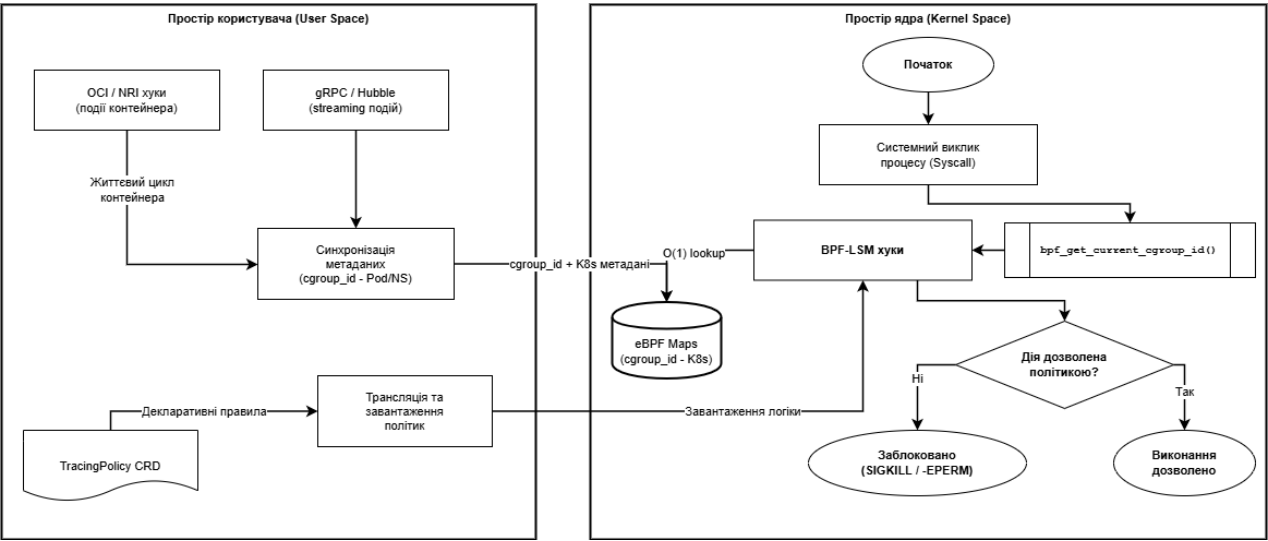


Рисунок 2.4 – Архітектура роботи Cilium Tetragon

Таким чином, Tetragon демонструє високоефективну та сучасну архітектуру превентивної безпеки з мінімальними накладними витратами у режимі моніторингу без policy, проте при увімкнених мережевих policy CPU overhead може бути значним, а складність Control Plane обмежує доцільність використання в ресурсно-обмежених Edge-середовищах.

2.2.3 KubeArmor та Tracee: спеціалізація та архітектурні обмеження

KubeArmor (CNCF Sandbox проєкт) — це хмарно-нативна система runtime security enforcement, яка фокусується на реалізації принципу найменших привілеїв (Least Privilege). Це досягається через використання механізмів Linux Security Modules — зокрема AppArmor, BPF-LSM та SELinux — залежно від можливостей конкретного вузла [59]. AppArmor та BPF-LSM відповідають за контроль доступу до ресурсів на рівні ядра, тоді як SELinux забезпечує MAC на

системах, де AppArmor недоступний. KubeArmor автоматично обирає відповідний LSM залежно від середовища виконання, не вимагаючи від користувача явного зазначення, який саме LSM використовується [59].

KubeArmor дозволяє визначати декларативні політики безпеки у форматі, сумісному з Kubernetes, використовуючи мітки (labels) для прив'язки правил до конкретних Pod-ів або Namespace-ів через поле selector.matchLabels [60]. Такий підхід органічно інтегрується з моделлю Kubernetes, де labels є основним механізмом групування ресурсів, і дозволяє централізовано керувати політиками безпеки на рівні кластера. YAML-правила трансформуються у відповідні профілі LSM та застосовуються превентивно, тобто небезпечна дія блокується ще до її виконання. Це робить KubeArmor ефективним для статичних або слабо змінних IoT-застосунків із передбачуваною поведінкою, де набір дозволених дій можна чітко визначити заздалегідь [18].

Проте KubeArmor не вирішує проблему надмірної телеметрії, оскільки його основна мета — контроль доступу, а не оптимізація потоку подій для систем моніторингу. Іншими словами, система орієнтована на enforcement (забезпечення політик), а не на ефективну обробку або фільтрацію великого обсягу подій. При оновленні політик KubeArmor оновлює eBPF Maps напряду без перезапуску агента, що є його архітектурною перевагою [18], однак YAML-правила все ж повинні бути перетворені у відповідні профілі для LSM, що може спричиняти короткочасне навантаження при масштабних оновленнях.

Додатковим обмеженням є відсутність автоматичного очищення eBPF Maps від записів, пов'язаних із вже завершеними Pod-ами або контейнерами. Це означає, що застарілі записи можуть накопичуватися у довготривалих Edge-розгортаннях, поступово збільшуючи споживання оперативної пам'яті. У пристроях з обмеженими ресурсами це є визначальним негативним фактором [18].

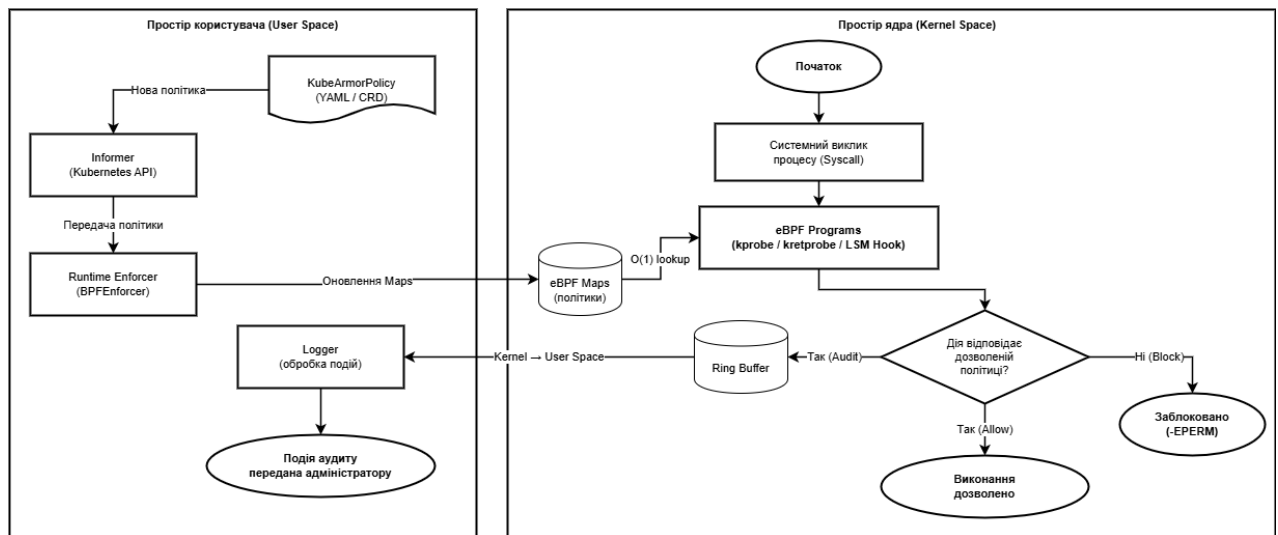


Рисунок 2.5 – Архітектура роботи KubeArmor

Tracee від Aqua Security орієнтований на іншу задачу — забезпечення максимальної видимості (deep observability) для задач цифрової криміналістики (forensics) та розслідування інцидентів [17]. Для цього він використовує велику кількість eBPF-хуків (включно з kprobes, uprobes, tracepoints та LSM hooks), які дозволяють відстежувати широкий спектр подій, включаючи безфайлові атаки, маніпуляції пам'яттю, ін'єкції коду та спроби ескалації привілеїв [17], [18]. Fileless execution — це тип атак, який виконується безпосередньо в пам'яті без збереження файлів на диску, що робить його складним для виявлення традиційними засобами [61].

Така глибина спостереження досягається за рахунок значного збільшення кількості перехоплюваних подій, що формує інтенсивний потік подій, який необхідно обробити. Хоча це є цінним для фахівців із безпеки під час аналізу інцидентів, для IoT-вузлів із обмеженими ресурсами така модель є надмірною. За результатами дослідження [18], при моніторингу мережевих подій із застосованою policy Tracee демонструє overhead 71,50% відносно базового стану — найвищий серед чотирьох досліджуваних інструментів. При

процесному моніторингу з policy overhead складає 37,59% [18]. Автори пояснюють це архітектурним рішенням Tracее використовувати ланцюжок tail-called eBPF-програм, у якому операції зіставлення з правилами виконуються повторно на кожному кроці ланцюжка, що призводить до надлишкових обчислень [18].

Ключовою архітектурною проблемою Tracее є явище backpressure — ситуація, коли система не встигає обробляти вхідний потік подій. У контексті eBPF це проявляється у переповненні perf buffer (буфера, що використовується для передачі подій із ядра в user-space). Коли буфер заповнюється, нові події починають відкидатися ядром, яке фіксує кількість втрачених подій [62]. Це створює критичну проблему: система втрачає частину телеметрії, включаючи потенційно важливі події безпеки [18], [19].

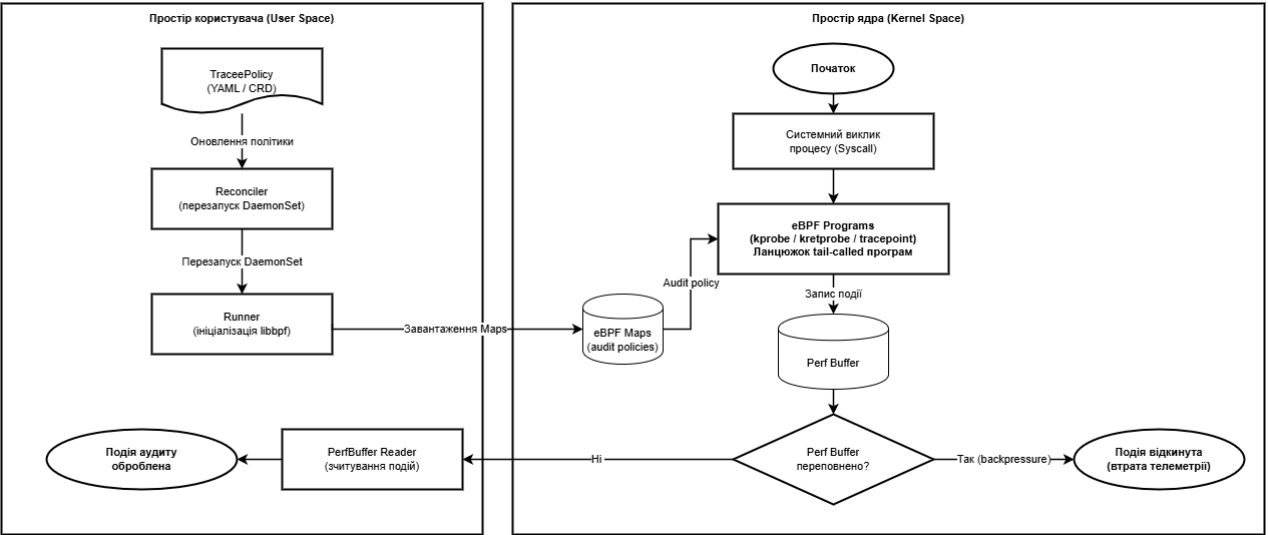


Рисунок 2.6 – Архітектура роботи Tracее

Таким чином, KubeArmor та Tracее демонструють два різних підходи до безпеки: перший — превентивний контроль доступу з інтеграцією в Kubernetes, другий — глибоку спостережуваність для задач аналізу інцидентів. Проте обидва підходи мають обмеження в контексті IoT-інфраструктури: KubeArmor

— через відсутність автоматичного очищення стану та орієнтацію на enforcement, а не на оптимізацію телеметрії, а Tracee — через надмірне споживання ресурсів і ризик втрати подій при високому навантаженні.

Значення CPU overhead у Таблиці 2.1 отримано в хмарно-нативному середовищі на архітектурі x86 (Intel Xeon Gold 6342, 32 ГБ RAM) [18]; на ARM64 IoT-вузлах показники можуть відрізнятись.

Таблиця 2.1 – Порівняльний аналіз провідних Open-Source рішень безпеки

Характеристика	Falco	Tetragon	KubeArmor	Tracee
Тип перехоплення	tp_btf(tracepoint/eBPF)	kprobe/ LSM/tracepoint/ uprobe	LSM + eBPF (kprobe, kretprobe)	kprobe/uprobe/ eBPF/kretprobe
Підтримка K8s-контексту	Асинхронна (user-space)	Синхронна (in-kernel)	Пре-конфігурація	Асинхронна
Де відбувається фільтрація	User-space (rule engine)	In-kernel (eBPF maps)	Kernel (LSM hooks)	User-space (pipeline)
CPU overhead, process (без policy)	8,49%	6,88%	18,29%	9,97%
CPU overhead, process (з policy)	9,96%	9,79%	18,04%	37,59%
CPU overhead, network (без policy)	19,93%	3,14%	87,74%	1,48%
CPU overhead, network (з policy)	24,77%	72,92%	88,59%	71,50%
Можливість блокування	Hi (detection only)	Так (SIGKILL, override)	Так (LSM, allow/block)	Hi
Lifecycle GC eBPF Maps	Hi	Часткова	Часткова	Hi

Аналіз Таблиці 2.1 виявляє чіткий компроміс між продуктивністю та ресурсною ефективністю існуючих рішень. У процесному бенчмарку всі чотири інструменти демонструють відносно близькі показники overhead, тоді як у мережевому бенчмарку різниця є суттєвою. Зокрема, Tetragon показує найнижчий CPU overhead у мережевому режимі без policy (3,14%), однак при увімкнених мережових policy цей показник зростає до 72,92%. Tracess демонструє аналогічну поведінку: 1,48% без policy та 71,50% з policy у мережевому бенчмарку. KubeArmor стабільно показує найвищий overhead у мережевому бенчмарку як без policy (87,74%), так і з policy (88,59%), що зумовлено необхідністю використання kprobes для отримання додаткового контексту, якого не надає LSM hook. Falco демонструє найменшу різницю між режимами в мережевому бенчмарку (19,93% без policy та 24,77% з policy), що пояснюється його виключно audit-орієнтованою архітектурою без механізмів примусового виконання policy.

Таким чином, жодне з розглянутих рішень не забезпечує збалансованого поєднання ключових вимог IoT-середовищ, а саме: легковагової архітектури агента, повноцінної фільтрації подій безпосередньо в ядрі на основі ID cgroups, ефективного управління пам'яттю через Lifecycle-Coupled GC та підтримки високопродуктивних мережових механізмів, таких як XDP, для протидії DDoS-атакам. Виявлена сукупність обмежень обґрунтовує необхідність розробки архітектури, орієнтованої на досягнення оптимального балансу між безпекою, продуктивністю та ефективним використанням ресурсів у контексті IoT-інфраструктури.

2.3 Виявлення ключових обмежень існуючих підходів до забезпечення безпеки In-Kernel Suppression

Проведений аналіз сучасних open-source рішень безпеки на базі eBPF (Falco, Tetragon, KubeArmor, Tracee) дозволяє систематизувати фундаментальні обмеження, що ускладнюють їх ефективне застосування в умовах Edge-інфраструктури. Зазначені обмеження мають системний характер і проявляються незалежно від конкретної реалізації, оскільки зумовлені базовими архітектурними підходами до збору, обробки та інтерпретації подій безпеки.

2.3.1 Семантичний розрив між ядром та Kubernetes-контекстом

Однією з ключових проблем є семантичний розрив (semantic gap) між рівнями абстракції. Ядро операційної системи оперує низькорівневими ідентифікаторами, тоді як політики безпеки у середовищі Kubernetes формулюються у термінах Pod, Namespace, ServiceAccount та labels. Внаслідок цього виникає необхідність зіставлення подій ядра з об'єктами оркестратора [18], [34].

У більшості існуючих рішень ця проблема вирішується асинхронно у user-space шляхом збагачення подій метаданими Kubernetes. Зокрема, Falco реалізує це через k8smeta-плагін, який отримує метадані з Kubernetes API-сервера та кешує їх у user-space [47]. Такий підхід призводить до часового розриву між подією та доступністю її повного контексту, потенційної втрати контексту при швидкому завершенні Pod-ів, а також ускладнює кореляцію подій у динамічних середовищах. Альтернативні підходи, що намагаються синхронізувати контекст безпосередньо в ядрі, супроводжуються значним

ускладненням архітектури та збільшенням споживання ресурсів [18], [57]. Таким чином, ефективне та водночас легковагове зіставлення подій із Kubernetes-контекстом залишається невирішеною задачею для ресурсно-обмежених IoT-середовищ.

2.3.2 Надмірні накладні витрати на обробку подій

Другим критичним обмеженням є високі накладні витрати, пов'язані з передачею та обробкою подій безпеки. У системах, що використовують модель user-space аналізу, кожна подія повинна бути передана з простору ядра через механізми на кшталт ring buffer або perf buffer, що супроводжується перемиканням контексту та копіюванням даних [18], [19].

В умовах Kubernetes-кластерів більшість подій генерується легітимною активністю системних компонентів. Як показують результати бенчмаркінгу [18], overhead при мережевому моніторингу з активованою policy сягає 71,50% для Tracex та 72,92% для Tetragon, тоді як KubeArmor стабільно демонструє overhead понад 87% у мережевому бенчмарку. Для ARM64 IoT-вузлів із обмеженими ресурсами це призводить до деградації продуктивності та зменшення доступної обчислювальної потужності для основних навантажень.

2.3.3 Неефективне управління станом та пам'яттю

Третім важливим обмеженням є проблема управління станом, зокрема використання eBPF Maps для зберігання контекстної інформації. У більшості існуючих рішень життєвий цикл даних у цих структурах не прив'язаний безпосередньо до життєвого циклу Kubernetes-об'єктів.

Це призводить до накопичення застарілих записів у пам'яті ядра. У довготривалих сценаріях експлуатації, характерних для Edge-середовищ, це створює ризик поступового зростання споживання пам'яті та потенційного

спрацювання механізму OOM-Killer. Зокрема, Tetragon зберігає деревоподібні структури предків процесів у eBPF Maps, очищення яких не завжди прив'язане до завершення контейнерів [18]. KubeArmor аналогічно не забезпечує автоматичного видалення відповідних записів при завершенні Pod-ів. Відсутність детермінованого механізму очищення стану ускладнює прогнозування використання ресурсів і негативно впливає на стабільність системи в умовах обмеженої пам'яті IoT-пристроїв.

					ІАЛЦ.467200.003 ПЗ	Арк.
						60
Зм.	Арк.	№ докум.	Підпис	Дата		

ВИСНОВОК ДО РОЗДІЛУ

У другому розділі проведено комплексний аналіз існуючих підходів до забезпечення безпеки в контейнеризованих Kubernetes-середовищах з урахуванням обмежень IoT-інфраструктури.

У підрозділі 2.1 встановлено, що традиційні механізми перехоплення подій, зокрема ptrace та auditd, характеризуються значними накладними витратами та відсутністю інтеграції з Kubernetes-контекстом, що робить їх непридатними для використання на ресурсно обмежених вузлах. Зокрема, повний цикл перехоплення одного системного виклику через ptrace потребує щонайменше чотирьох переходів між просторами виконання. Водночас сучасні технології, такі як eBPF, BPF-LSM та механізми fentry/fexit, забезпечують значно ефективнішу модель моніторингу та контролю завдяки можливості виконання логіки безпосередньо в ядрі без генерації виключень процесора.

У підрозділі 2.2 виконано порівняльний аналіз сучасних open-source рішень, який показав, що кожне з них має суттєві обмеження для Edge-середовищ. Системи з user-space аналізом (Falco, Tracee) демонструють overhead до 71,50% при мережевому моніторингу з активованою policy, тоді як рішення з in-kernel обробкою (KubeArmor) — понад 87% у мережевому бенчмарку через необхідність використання kprobes для отримання додаткового контексту.

У підрозділі 2.3 систематизовано ключові архітектурні обмеження існуючих підходів: наявність семантичного розриву між рівнем ядра та рівнем оркестратора, надмірні витрати на передачу та обробку подій, а також відсутність ефективного механізму управління життєвим циклом даних у eBPF-структурах. Вказані проблеми мають системний характер і суттєво звужують

можливості застосування існуючих рішень у динамічних та ресурсно обмежених середовищах.

Отримані результати обґрунтовують необхідність розробки архітектури, орієнтованої на фільтрацію подій безпосередньо в ядрі, мінімізацію передачі нерелевантної телеметрії та узгодження життєвого циклу даних із життєвим циклом контейнерів. Зазначені принципи покладено в основу системи OIKESA, архітектура та реалізація якої розглядаються у Розділі 3.

					ІАЛЦ.467200.003 ПЗ	Арк.
						62
Зм.	Арк.	№ докум.	Підпис	Дата		

РОЗДІЛ 3

ПРОЄКТУВАННЯ ТА РЕАЛІЗАЦІЯ СИСТЕМИ ORCHESTRATOR-AWARE IN-KERNEL EVENT SUPPRESSION AGENT

3.1 Загальна архітектура системи та концепція дворівневої взаємодії

Система OIKESA реалізована відповідно до дворівневої архітектурної моделі, що передбачає чіткий розподіл функцій між компонентами простору ядра та простору користувача. Таке розмежування є фундаментальним проєктним рішенням, яке забезпечує поєднання максимальної продуктивності фільтрації системних викликів і мережевого трафіку з гнучкістю управлінської логіки оркестраційного рівня. Загальну архітектуру системи наведено на структурній схемі (Додаток А).

Компоненти простору ядра реалізовано у вигляді набору eBPF-програм, що завантажуються у верифікований простір ядра Linux і приєднуються до визначених точок перехоплення. Ці програми здійснюють низькорівневу інспекцію системних викликів, мережевих пакетів та подій файлової системи у момент їх виникнення, без передачі проміжних даних у простір користувача. Така парадигма, відома як «пригнічення подій на рівні ядра» (In-Kernel Event Suppression, або Drop-at-Source), дозволяє відмовитися від традиційної схеми, за якої кожна подія безпеки спершу передається у простір користувача для аналізу, а лише потім формується рішення про блокування. Усунення цього зворотного зв'язку між ядром і простором користувача є ключовим чинником досягнення субмілісекундних затримок під час прийняття рішень про блокування.

Компонент простору користувача реалізовано мовою програмування Go у вигляді модульного агента управління. Архітектурно він складається з підсистеми взаємодії з API-сервером оркестратора (інтеграція з Kubernetes API через механізм Informers), диспетчера мережевих інтерфейсів (VethManager), що відповідає за динамічне прикріплення TC-фільтрів до віртуальних Ethernet-інтерфейсів контейнерів, координатора збереження стану системи через асинхронний канал персистентності, а також модуля обробки подій життєвого циклу Pod-об'єктів. Ініціалізація та завантаження eBPF-програм здійснюється через набір об'єктів bpfObjects, що надає агенту типізований доступ до всіх карт і програм, скомпільованих у байт-код BPF. Структуру модулів агента управління та їх взаємозв'язки наведено у функціональній схемі системи (Додаток Б).

Ключовою проблемою, яку вирішує запропонована архітектура, є подолання семантичного розриву між рівнем оркестрації та рівнем ядра. Kubernetes оперує абстрактними об'єктами — Pod-ресурсами, просторами імен та мітками, — тоді як ядро Linux ідентифікує ізольовані процеси за допомогою контрольних груп (cgroups). Числовий ідентифікатор контрольної групи (cgroup_id) є унікальним у межах одного вузла та збігається зі значенням поля st_ino структури stat, отриманого для відповідного каталогу у файловій підсистемі cgroupfs (/sys/fs/cgroup). Відображення між абстрактним ідентифікатором Pod-об'єкта та конкретним значенням cgroup_id здійснюється агентом управління шляхом аналізу файлу /proc/{pid}/cgroup для відповідного процесу контейнера та виклику stat() на знайденому каталозі cgroupfs. Отриманий cgroup_id використовується як первинний ключ у картах eBPF, однозначно розпізнаваний програмами в просторі ядра.

Двонаправлена взаємодія між рівнями реалізована виключно через eBPF-карти — структури даних, спільні для простору ядра та простору користувача.

					ІАЛЦ.467200.003 ПЗ	Арк.
						64
Зм.	Арк.	№ докум.	Підпис	Дата		

Агент управління записує рішення у відповідні карти, а eBPF-програми зчитують ці рішення в режимі реального часу при кожному перехопленні системного виклику або мережевого пакета. Зворотний напрям взаємодії — від eBPF-програм до агента — реалізовано через кільцевий буфер `alert_ringbuf` типу `BPF_MAP_TYPE_RINGBUF` (розміром 2 МБ), що дозволяє програмам у просторі ядра асинхронно повідомляти агента про виявлені аномалії, не блокуючи основний шлях обробки.

3.2 Проєктування підсистеми eBPF Maps та управління пам'яттю

Підсистема eBPF-карт є центральним елементом архітектури OIKESA, оскільки реалізує семантику авторизації та блокування для всіх процесів, що підлягають контролю. Кожна карта характеризується типом структури, типами ключа та значення, а також максимальною ємністю, що фіксується на етапі завантаження eBPF-програм і не може бути змінена під час роботи системи. У системі визначено десять карт, специфікацію яких наведено в Таблиці 3.1.

Таблиця 3.1 — Специфікація eBPF Maps системи OIKESA

Назва карти	Тип структури	Ключ / Значення	Призначення
<code>whitelist_map</code>	<code>BPF_MAP_TYPE_LRU_HASH</code>	<code>u64 cgroup_id / u8 flag</code>	Реєстр авторизованих контрольних груп
<code>blocklist_map</code>	<code>BPF_MAP_TYPE_LRU_HASH</code>	<code>u64 cgroup_id / u8 flag</code>	Реєстр заблокованих контрольних груп
<code>pending_cgroups</code>	<code>BPF_MAP_TYPE_LRU_HASH</code>	<code>u64 cgroup_id / u64 timestamp_ns</code>	Тимчасовий буфер ідентифікаторів у процесі ініціалізації
<code>ns_policy_map</code>	<code>BPF_MAP_TYPE_LRU_HASH</code>	<code>u64 cgroup_id / struct ns_policy_val</code>	Рівні політики безпеки для окремих контрольних груп

Продовження таблиці 3.1

Назва карти	Тип структури	Ключ / Значення	Призначення
ancestry_map	BPF_MAP_TYPE_LRU_HASH	u64 child_cgroup_id / u64 parent_cgroup_id	Кеш ієрархії контрольних груп для заблокованих контейнерів
kubepods_map	BPF_MAP_TYPE_LRU_HASH	u64 cgroup_id / u8 flag	Маркування контейнерів, що належать до кластера Kubernetes
xdp_blocklist	BPF_MAP_TYPE_LRU_HASH	u32 IPv4 addr / u8 flag	Мережеві правила блокування IPv4 для XDP та TC
xdp_blocklist_v6	BPF_MAP_TYPE_LRU_HASH	struct ipv6_key / u8 flag	Мережеві правила блокування IPv6 для XDP та TC
stats_map	BPF_MAP_TYPE_PERCPU_ARRAY	u32 index / u64 counter	13 per-CPU лічильників операційних метрик
alert_ringbuf	BPF_MAP_TYPE_RINGBUF	— / struct alert_event	Кільцевий буфер асинхронних сповіщень (2 МБ)

Вибір типу BPF_MAP_TYPE_LRU_HASH для більшості карт зумовлений специфікою умов функціонування системи безпеки в середовищі Kubernetes. На відміну від стандартного типу BPF_MAP_TYPE_HASH, який при досягненні максимальної ємності повертає код помилки -E2BIG та відмовляє у вставці нового запису, тип LRU_HASH реалізує алгоритм витіснення найменш нещодавно використаного елемента (LRU) безпосередньо у просторі ядра без участі агента управління. Це гарантує стабільну роботу

системи при динамічному масштабуванні кластера та короткочасних сплесках кількості активних контейнерів.

Карта `pending_cgroups` зберігає як значення 64-бітну мітку часу в наносекундах, отриману функцією `time.Now().UnixNano()` у просторі користувача в момент виявлення контейнера через `containerd gRPC`. Ця мітка слугує точкою відліку для алгоритму TTL-очищення, що виконується безпосередньо в eBPF-програмах простору ядра при кожному перехопленні системного виклику.

Карта `ancestry_map` призначена для відстеження ієрархії контрольних груп при події `fork` процесу. Ключем є ідентифікатор дочірньої контрольної групи (`child_cgroup_id`), значенням — ідентифікатор батьківської (`parent_cgroup_id`). Заповнення карти відбувається в eBPF-програмі, прикріпленій до точки трасування `tp/sched/sched_process_fork`: при кожній події `fork` перевіряється наявність батьківської контрольної групи у `blocklist_map`, і лише за позитивного результату новий запис додається до `ancestry_map`. Такий підхід мінімізує розмір карти, зберігаючи лише потенційно небезпечні зв'язки.

3.2.1 Механізм захисту від LRU-флуду

При проектуванні карти `blocklist_map` виявлено структурну вразливість механізму LRU-витіснення. За умови, що максимальна ємність карти на рівні ядра становить $N_{\max}$ дорівнює 64 записи (константа `BLOCKLIST_MAX`), а система функціонує в умовах одночасного існування понад 64 активних записів, вбудований алгоритм LRU ядра може витіснити запис легітимного заблокованого контейнера. Наслідком є хибнонегативна відмова — eBPF-програма не знаходить ідентифікатора у `blocklist_map` та пропускає дію, яка мала бути заблокована.

Для усунення цієї вразливості реалізовано дворівневий алгоритм керування заповненістю. На рівні ядра карта `blocklist_map` оголошується з максимальною ємністю `BLOCKLIST_MAX` дорівнює 64 записи — це фізичне обмеження, що не може бути перевищене жодною eBPF-програмою. На рівні простору користувача агент підтримує внутрішній лічильник `blocklistCount` та дозволяє внесення нових записів лише до досягнення програмного порогу N_{thresh} , що дорівнює 48 позицій (значення `MaxBlocklistEntries` у конфігурації), це становить 75% від апаратного ліміту N_{max} . Доступ до лічильника захищено окремим м'ютексом `blocklistMu`, що дозволяє виконувати операції додавання та видалення конкурентно без захоплення глобального блокування агента.

Резервний простір у 16 записів гарантує, що навіть при максимальному заповненні, контрольованому агентом, алгоритм LRU ядра ніколи не витіснить запис активного заблокованого контейнера при штатному рівні завантаження. Цей захист є критичним для сценаріїв, коли зовнішня система реагування (SOAR) інтенсивно додає записи до `blocklist_map` у відповідь на виявлені загрози.

3.2.2 Алгоритм TTL-очищення карти `pending_cgroups`

Для карти `pending_cgroups` реалізовано алгоритм очищення записів із часовим лімітом (Time-To-Live, TTL). Принципова особливість цього алгоритму полягає в тому, що він виконується не фоновією задачею у просторі користувача, а безпосередньо в eBPF-програмах простору ядра при кожному перехопленні системного виклику. Це гарантує мінімальні накладні витрати та атомарність операції відносно самого системного виклику.

При кожному перехопленні eBPF-програма виконує пошук `cgroup_id` поточного процесу у карті `pending_cgroups`. Якщо запис знайдено,

обчислюється різниця між поточним значенням монотонного годинника ядра та збереженою міткою часу створення запису:

$$\Delta t = t_{now} - t_{created}. \quad (3.1)$$

Якщо $\Delta t < TTL$, де `PENDING_TTL_NS` дорівнює 5 секунд, запис вважається актуальним: `cgroup_id` автоматично переміщується з `pending_cgroups` до `whitelist_map` з прапорцем дозволу, оригінальний запис у `pending_cgroups` видаляється, а системний виклик придушується без передачі даних у простір користувача. Якщо ж $\Delta t \geq TTL$, запис видаляється з карти, інкрементується лічильник `STAT_PENDING_EXPIRED`, а до кільцевого буфера надсилається сповіщення типу `EVENT_TYPE_PENDING_EXP`.

Цей механізм вирішує фундаментальну проблему стану гонки (race condition), що виникає через асинхронний характер взаємодії між `containerd` (подія `task/start`, яка ініціює внесення `cgroup_id` до `pending_cgroups`) та `Kubernetes API` (реєстрація `Pod`-об'єкта у стані `Running`, яка ініціює перенесення до `whitelist_map`). `TTL` у 5 секунд покриває типовий час цієї затримки та запобігає накопиченню застарілих записів про контейнери, що не завершили ініціалізацію у штатному порядку.

3.3 Реалізація стеку eBPF-програм перехоплення системних викликів

Стек eBPF-програм системи OIKESA охоплює десять функціональних груп точок перехоплення, що відповідають різним векторам потенційних загроз: несанкціонованому виконанню процесів (`execve`), неавторизованому доступу до файлів (`openat`), несанкціонованій мережевій взаємодії (`connect`, `socket_connect`, `socket_bind`), спробам ескалації привілеїв (`task_fix_setuid`), мережевій фільтрації на рівні драйвера (XDP) та вихідного трафіку (TC), а також відстеженню ієрархії контрольних груп (`sched_process_fork`,

sched_process_exit). Кожна eBPF-програма незалежно проходить верифікацію інструментом eBPF Verifier ядра Linux, що гарантує її завершуваність, відсутність нескінченних циклів та безпечний доступ до пам'яті.

3.3.1 Центральний алгоритм контекстно-залежного придушення подій

Програми перехоплення exesve, openat та connect реалізують спільну шестирівневу логіку прийняття рішення, що виконується в суворо визначеному порядку безпосередньо в ядрі. Схему алгоритму наведено в Додатку В.

Перший рівень — перевірка карти blocklist_map. Якщо cgroup_id поточного процесу знайдено в цій карті, подія класифікується як така, що підлягає придушенню: інкрементується лічильник STAT_BLOCKED та функція повертає керування без генерації сповіщень.

Другий рівень — перевірка карти whitelist_map. Якщо cgroup_id знайдено у білому списку, подія класифікується як легітимна активність відомого Pod-об'єкта: інкрементується відповідний лічильник придушення (STAT_EXECVE_SUPP або STAT_OPENAT_SUPP) та функція завершується без задіяння кільцевого буфера. Саме цей шлях є критичним для ефективності системи, оскільки через нього проходить більшість системних викликів у нормальному кластері, не створюючи жодного навантаження на простір користувача.

Третій рівень — перевірка карти pending_cgroups та TTL-промоція відповідно до алгоритму, описаного у підрозділі 3.2.2. При актуальному записі виконується автоматичне перенесення cgroup_id до whitelist_map та придушення події; при застарілому — видалення запису та генерація сповіщення EVENT_TYPE_PENDING_EXP.

Четвертий рівень — перевірка карти `ancestry_map`. Виконується пошук батьківського `cgroup_id` за ключем поточного `cgroup_id`. Якщо батьківська контрольна група знайдена у `blocklist_map`, дочірня також вноситься до `blocklist_map`, а до кільцевого буфера надсилається сповіщення `EVENT_TYPE_ANCESTRY`. Це забезпечує автоматичне поширення блокування на всіх нащадків заблокованого контейнера, утворених через системний виклик `fork`, без жодної участі агента у просторі користувача.

П'ятий рівень — перевірка `ns_policy_map`. Якщо для `cgroup_id` визначено запис у карті `ns_policy_map`, застосовується відповідна поведінка залежно від рівня `policy_level` та значення бітової маски `flags` структури `ns_policy_val`: `NS_POLICY_PERMISSIVE` (рівень 2) придушує подію без сповіщення; `NS_POLICY_STRICT` (рівень 1) з прапорцем `NS_FLAG_BLOCK_UNKNOWN` (0x02) придушує подію з інкрементом `STAT_BLOCKED`; `NS_FLAG_AUDIT_ONLY` (0x01) генерує сповіщення без блокування.

Шостий рівень — фільтрація хостових процесів і генерація сповіщень для Kubernetes. Якщо жодна з попередніх перевірок не вирішила долю події, програма перевіряє наявність `cgroup_id` у `kuberpods_map`. Відсутність у цій карті означає хостовий процес — подія ігнорується. Присутність означає Pod, що вже зареєстрований у Kubernetes, але ще не внесений до `whitelist_map` через незавершену синхронізацію: генерується сповіщення `EVENT_TYPE_EXECVE` або `EVENT_TYPE_OPENAT`.

3.3.2 Вибір механізму перехоплення: `fentry` та резервний `raw_tracepoint`

Для перехоплення системних викликів `execve`, `openat` та `connect` основним механізмом прикріплення обрано тип хука `fentry`. Принципова відмінність `fentry` від традиційного `kprobe` полягає в тому, що `fentry` є статично визначеною

точкою інструментування: компілятор ядра вставляє NOP-інструкцію на початку кожної відстежуваної функції, яка замінюється переходом до eBPF-програми під час завантаження. Це забезпечує компілятору BPF доступ до типізованих аргументів функції через BTF (BPF Type Format) без нетипізованого розіменування вказівників, що знижує кількість перевірок верифікатора та накладні витрати на виклик хука.

Для забезпечення сумісності з ядрами, що не підтримують fentry (версії до 5.5 для x86_64 або до 6.0 для ARM64), реалізовано резервний шлях прикріплення через raw_tracepoint/sys_enter. Вибір між двома механізмами визначається конфігураційним параметром EnableFentry агента управління: якщо прикріплення fentry завершується помилкою — наприклад, через відсутність BTF-інформації або коли CONFIG_DEBUG_INFO_BTF дорівнює 0 у конфігурації ядра, — активується raw_tracepoint-варіант програми.

Для хука openat реалізовано додаткову фільтрацію шляхів безпосередньо в ядрі. За допомогою допоміжної функції bpf_probe_read_user_str() eBPF-програма зчитує перші 8 байт шляху файлу з адресного простору процесу та порівнює їх з переліком безпечних префіксів: /proc/, /sys/, /dev/, /run/, /tmp/. Виклики openat() до цих каталогів від процесів із невідомим cgroup_id придушуються без генерації сповіщень, що суттєво зменшує кількість хибних спрацювань на хостових системних процессах.

3.3.3 Синхронне превентивне блокування через підсистему BPF-LSM

Для активного блокування заборонених операцій використано підсистему BPF-LSM. На відміну від механізмів аудиту, що лише фіксують факт виконання операції, BPF-LSM дозволяє eBPF-програмі повертати від'ємний код помилки безпосередньо у точці перехоплення, до завершення виконання системного

					ІАЛЦ.467200.003 ПЗ	Арк.
						72
Зм.	Арк.	№ докум.	Підпис	Дата		

виклику. Синхронна природа цього механізму гарантує, що заборонена дія не буде виконана навіть частково: ядро отримує відмову до передачі запиту підсистемі файлової системи або мережевому стеку.

У системі OIKESA реалізовано чотири LSM-хуки. Хук `lsm/bprm_check_security` (програма `lsm_exec_check`) викликається при кожній спробі запуску виконуваного файлу через `execve()`: за наявності `cgroup_id` у `blocklist_map` повертає код `-EPERM` (Operation not permitted), що блокує запуск до фактичного завантаження бінарного файлу в пам'ять. Хук `lsm/socket_connect` (програма `lsm_socket_connect_check`) перевіряє вихідні TCP/UDP-з'єднання: за наявності `cgroup_id` у `blocklist_map` повертає `-ECONNREFUSED`, що є основним механізмом мережевої ізоляції на рівні сокетів. Хук `lsm/socket_bind` (програма `lsm_socket_bind_check`) блокує спробу прив'язки до мережевого порту для заблокованих контрольних груп, повертаючи `-EACCES`, що унеможливорює запуск нових мережевих служб або зворотних з'єднань (reverse shell). Хук `lsm/file_open` (програма `lsm_file_open_check`) реалізує фільтрацію чутливих файлів для заблокованих контейнерів: ім'я файлу зчитується зі структури `dentry` через механізм CO-RE (`BPF_CORE_READ`), після чого виконується посимвольне порівняння зі списком заборонених імен — `shadow`, `passwd`, `sysrq-trigger`, `token`, `ca.crt`, `tls.crt`, `tls.key`, `kubeconfig` — та повертається `-EACCES` у разі збігу. Хук `lsm/task_fix_setuid` (програма `lsm_setuid_check`) виявляє спроби ескалації привілеїв — перехід UID з ненульового значення до 0. Програма генерує сповіщення `EVENT_TYPE_PRIVESC` і додатково повертає `-EPERM` для контрольних груп, що знаходяться у `blocklist_map`.

3.3.4 Мережева фільтрація на рівні XDP та TC

Для захисту мережевого рівня розгорнуто дві додаткові eBPF-програми. XDP-фільтр (програма `xdp_filter`) прикріплюється до вхідного інтерфейсу вузла

					ІАЛЦ.467200.003 ПЗ	Арк.
						73
Зм.	Арк.	№ докум.	Підпис	Дата		

(cni0, eth0, flannel.1) та здійснює первинну класифікацію пакетів на мінімальному рівні стеку обробки — безпосередньо в драйвері мережевої карти, до виділення пам'яті для структури `sk_buff`. Програма розбирає Ethernet-заголовки, визначає тип протоколу та виконує пошук адрес джерела і призначення в картах `xdp_blocklist` (IPv4, `BPF_MAP_TYPE_LRU_HASH`, до 256 записів) та `xdp_blocklist_v6` (IPv6, ключ `struct ipv6_key` з двох 64-бітних полів `hi/lo`, до 64 записів). При збігу пакет відкидається поверненням коду `XDP_DROP`, а лічильник `STAT_XDP_DROPS` інкрементується.

ТС-фільтр прикріплюється до вихідного трафіку віртуальних Ethernet-інтерфейсів контейнерів і забезпечує трирівневу фільтрацію. Перший рівень — перевірка `cgroup_id` процесу, що генерує пакет, за допомогою допоміжної функції `bpf_skb_cgroup_id`: якщо ідентифікатор присутній у `blocklist_map`, весь вихідний трафік контейнера відкидається поверненням `TC_ACT_SHOT`. Другий рівень — для пакетів IPv4 виконується пошук адреси призначення у `xdp_blocklist`. Третій рівень — аналогічна перевірка для IPv6. Обидва рівні повертають `TC_ACT_SHOT` у разі збігу або `TC_ACT_OK` за його відсутності.

Прикріплення ТС-фільтра до інтерфейсів здійснюється динамічно компонентом `VethManager` за допомогою API `link.AttachTCX()` (для ядер Linux 6.6+), що підтримує прикріплення одночасно в напрямках `TCX Ingress` та `TCX Egress`. Карти `xdp_blocklist` та `xdp_blocklist_v6` формуються агентом управління і є спільними для XDP- та ТС-фільтра, що забезпечує єдину точку управління мережевими правилами для обох рівнів фільтрації.

3.4 Алгоритми взаємодії агента з Kubernetes API та забезпечення персистентності

Агент управління отримує інформацію про стани об'єктів кластера Kubernetes через механізм `Informers` на базі `SharedInformerFactory`. Цей

					ІАЛЦ.467200.003 ПЗ	Арк.
						74
Зм.	Арк.	№ докум.	Підпис	Дата		

механізм реалізує модель довгоживучого HTTP/2-з'єднання з активованим watch: після встановлення з'єднання агент отримує повний поточний стан усіх відстежуваних об'єктів через виклик LIST, а в подальшому — лише диференціальні повідомлення про зміни (watch-події типів ADDED, MODIFIED, DELETED). Такий підхід мінімізує навантаження на API-сервер та забезпечує реактивну обробку подій без затримок, що вносяться циклічним опитуванням. Для кожного типу відстежуваних об'єктів (Pod, Namespace) зареєстровано окремий обробник, що реалізує інтерфейс `cache.ResourceEventHandler`.

3.4.1 Алгоритм обробки життєвого циклу Pod-об'єктів

Алгоритм обробки події створення Pod-об'єкта реалізований з урахуванням асинхронного характеру ініціалізації кластера. Після отримання події `AddFunc` від `Pod Informer` агент може ще не мати доступу до `cgroup_id` контейнера, оскільки ядро не завершило ініціалізацію контрольної групи або `containerd` ще не повідомив про запуск задачі. Тому в фазі первинної синхронізації всі отримані Pod-об'єкти накопичуються у внутрішньому буфері `initialPods` та обробляються пакетно після завершення функцією `WaitForCacheSync()`.

Після завершення первинної синхронізації для кожного Pod-об'єкта у стані `Running` виконується наступна послідовність операцій. Спочатку агент намагається отримати `cgroup_id` із внутрішньої структури `containerCgroups`, заповненої раніше через `containerd gRPC`. Якщо відповідного запису немає, виконується пошук через `cgroupfs`: агент перевіряє кілька шаблонних шляхів залежно від QoS-класу Pod-об'єкта (`BestEffort`, `Burstable`, `Guaranteed`) — наприклад, `/sys/fs/cgroup/kubepods.slice/kubepods-pod{uid}.slice` — та читає `inode` каталогу контейнера (підкаталогу з префіксом `cri-containerd-`) за

					ІАЛЦ.467200.003 ПЗ	Арк.
						75
Зм.	Арк.	№ докум.	Підпис	Дата		

допомогою виклику `stat()`, поле `Ino` якого містить шуканий `cgroup_id`. Далі запис видаляється з `pending_cgroups` (якщо він там є), `cgroup_id` додається до `whitelist_map` та `kubepods_map` через `BatchUpdate()` — одним системним викликом BPF замість `N` послідовних. У разі часткової невдачі `BatchUpdate()` реалізована резервна логіка послідовного додавання для записів, що залишилися. Нарешті, оновлюється внутрішня структура `knownPods` та застосовується NS-політика через `addNSPolicyForPod()` відповідно до мітки `security.oikesa/policy` простору імен.

Обробка події завершення роботи Pod-об'єкта ініціюється при появі ненульового значення поля `DeletionTimestamp` у метаданих об'єкта (перехід у стан `Terminating`). Агент виконує негайне видалення відповідних ідентифікаторів контрольних груп з усіх карт: `whitelist_map`, `blocklist_map`, `pending_cgroups` та `ns_policy_map`. Додатково очищуються внутрішні структури `knownPods` та `containerCgroups`. Щойно `cgroup_id` видалено з `whitelist_map`, eBPF-програми в ядрі припиняють пропускати системні виклики від процесів цього контейнера при наступному ж перехопленні.

3.4.2 Механізм виявлення контейнерів через `containerd gRPC`

Для виявлення нових контейнерів до їх реєстрації в Kubernetes API реалізовано паралельний механізм підписки на події `containerd` через `gRPC`-протокол. Агент підключається до сокету `containerd` (автоматично обирається перший наявний з переліку: `/run/k3s/containerd/containerd.sock` для K3s, `/run/containerd/containerd.sock` для стандартного `containerd`) та підписується на топіки `/tasks/start` та `/tasks/exit` простору імен `k8s.io`.

При отриманні події `TaskStart` агент зчитує ідентифікатор процесу (PID) контейнера, відкриває файл `/proc/{pid}/cgroup` та аналізує його вміст для визначення ієрархії `cgroup v2` — рядок з ієрархічним ідентифікатором `0` або

					ІАЛЦ.467200.003 ПЗ	Арк.
						76
Зм.	Арк.	№ докум.	Підпис	Дата		

порожнім другим полем. Ідентифікатор `cgroup_id` отримується шляхом виклику `stat()` на відповідному каталозі у `/sys/fs/cgroup` та зчитування поля `Ino` структури `Stat_t`, яке у `cgroup v2` містить `inode` відповідного `cgroupfs`-каталогу. Знайдений `cgroup_id` зберігається в структурі `containerCgroups` та передається до карти `pending_cgroups` через функцію `addToPending()` з поточною міткою часу. Паралельно ініціюється асинхронне прикріплення ТС-фільтра через `VethManager.AttachForContainer()`.

Для забезпечення відмовостійкості реалізовано механізм автоматичного перепідключення з експоненціальним збільшенням інтервалу між спробами. Початковий інтервал становить 1 секунду, максимальний — 30 секунд (параметр `ContainerdReconnectMaxBackoff`). У проміжках між спробами `gRPC`-підключення паралельно активується резервний механізм сканування файлової системи `/proc`: агент перебирає всі каталоги в `/proc`, шукаючи процеси з іменем `containerd-shim`, зчитує їх командний рядок для вилучення `containerID` та виконує процедуру визначення `cgroup_id` і додавання до `pending_cgroups`, аналогічну `gRPC`-шляху.

3.4.3 Динамічне прикріплення ТС-фільтрів до veth-інтерфейсів

Компонент `VethManager` відповідає за динамічне прикріплення `eBPF`-програми `tc_egress_filter` до хостової сторони віртуальної Ethernet-пари (`veth`) кожного контейнера. Алгоритм пошуку хостового інтерфейсу реалізовано у функції `hostVethForContainerPID()` і базується на перемиканні мережевих просторів імен.

Для гарантії того, що всі операції з просторами імен виконуються в межах одного потоку операційної системи, на початку функції викликається `runtime.LockOSThread()`. Далі зберігається дескриптор поточного мережевого

простору імен хоста (відкриття `/proc/self/ns/net`). Виконується перехід у мережевий простір імен контейнера через `unix.Open(/proc/{pid}/ns/net)` та `unix.Setns()` з прапорцем `unix.CLONE_NEWNET`. У просторі імен контейнера через сирий Netlink-сокет надсилається запит `RTM_GETLINK` для отримання переліку всіх мережевих інтерфейсів. Аналізуються атрибути `IFLA_IFNAME` та `IFLA_LINK`: інтерфейс `veth` ідентифікується за тим, що значення атрибута `IFLA_LINK` (`iflink`, індекс парного інтерфейсу) відрізняється від власного `ifindex` та не дорівнює нулю. Отриманий індекс парного інтерфейсу повертається до простору імен хоста через зворотний виклик `unix.Setns()`. У просторі імен хоста за отриманим індексом за допомогою `netlink.LinkByIndex()` знаходиться ім'я хостового `veth`-інтерфейсу.

До знайденого інтерфейсу прикріплюються дві копії ТС-фільтра — у напрямку `AttachTCXIngress` та `AttachTCXEgress` — через API `link.AttachTCX()` бібліотеки `cilium/ebpf`. Дескриптори прикріплень зберігаються в структурі `vethAttachment` та звільняються під час `DetachForContainer()` при завершенні контейнера.

3.4.4 Механізм персистентності стану через Kubernetes ConfigMap

Для забезпечення відновлення стану системи після перезапуску агента реалізовано механізм персистентності на базі об'єкта `ConfigMap` Kubernetes API. Вміст карти `blocklist_map` серіалізується у формат JSON та зберігається у `ConfigMap` з іменем `oikesa-blocklist-state` у просторі імен `kube-system`. Ключами `ConfigMap` є: `blocklist.json` (JSON-об'єкт із відображенням рядкового представлення `cgroup_id` до логічного значення `true`), `updated_at` (мітка часу) та `count` (кількість записів).

Операція збереження реалізована через асинхронний persist-воркер, що запускається окремою горутиною. Воркер очікує сигналів на буферизованому каналі persistCh (ємність 1). Сигнал надсилається при кожній зміні blocklist (виклику AddToBlocklist() або RemoveFromBlocklist()) через функцію signalPersist(). Для уникнення надмірних записів при серійних операціях реалізовано механізм «дрейну» з затримкою 300 мс: якщо протягом 300 мс після першого сигналу надходять нові сигнали, таймер скидається, і фактичне збереження відкладається до 300-мілісекундної тиші. Це дозволяє агрегувати серію послідовних змін в одну операцію запису.

Запис у ConfigMap реалізовано через механізм оптимістичного замикання: спочатку виконується операція Get() для отримання поточного ресурсу (з полем resourceVersion), після чого — Update() з оновленими даними. У разі відсутності ConfigMap (Get() повертає помилку) виконується Create(). При помилці Update() здійснюється до 5 повторних спроб із лінійним збільшенням інтервалу (50 мс, 100 мс, 150 мс, 200 мс, 250 мс). Кожна операція обмежена таймаутом 5 секунд через context.WithTimeout().

Під час ініціалізації агент виконує відновлення стану через функцію restoreBlocklistFromConfigMap(), яка викликається до запуску Informers. Функція реалізує retry-петлю: при мережевих помилках агент очікує 5 секунд і повторює спробу; HTTP 404 інтерпретується як штатний перший запуск і функція завершується без помилки. При успішному отриманні ConfigMap виконується JSON-десеріалізація та послідовне відновлення записів через objs.Blocklist.Put() з дотриманням ліміту MaxBlocklistEntries.

3.5 Система метрик та спостережуваності

Для збору операційних метрик у просторі ядра визначено карту stats_map типу BPF_MAP_TYPE_PERCPU_ARRAY з 13 індексами. Використання per-

					ІАЛЦ.467200.003 ПЗ	Арк.
						79
Зм.	Арк.	№ докум.	Підпис	Дата		

CPU варіанту гарантує атомарне інкрементування без міжпроцесорної синхронізації: кожне ядро процесора оновлює власний незалежний екземпляр лічильника функцією `sync_fetch_and_add()`, а агент управління при зчитуванні метрик підсумовує значення всіх екземплярів. Специфікацію індексів карти наведено в Таблиці 3.2.

Таблиця 3.2 — Специфікація ядрових лічильників підсистеми метрик OIKESA

Індекс	Назва лічильника	Призначення
0	<code>execve_suppressed</code>	Придушені виклики <code>execve</code> — потрапляння у <code>whitelist_map</code>
1	<code>execve_alerts</code>	Сповіщення <code>execve</code> для невідомих контрольних груп Kubernetes
2	<code>blocked</code>	Заблоковані операції через BPF-LSM (<code>bprm_check</code> , <code>file_open</code>)
3	<code>xdp_drops</code>	Відкинуті пакети на рівні XDP (<code>XD_DROP</code>)
4	<code>openat_suppressed</code>	Придушені виклики <code>openat</code> — потрапляння у <code>whitelist_map</code>
5	<code>openat_alerts</code>	Сповіщення <code>openat</code> для невідомих контрольних груп
6	<code>privesc_alerts</code>	Виявлені спроби ескалації привілеїв (UID 0)
7	<code>connect_alerts</code>	Сповіщення аудиту вихідних з'єднань <code>connect</code>
8	<code>ancestry_blocks</code>	Блокування, поширені через <code>ancestry_map</code> на дочірні процеси
9	<code>tc_egress_drops</code>	Відкинуті пакети фільтром TC (<code>TC_ACT_SHOT</code>)
10	<code>socket_connect_blocks</code>	Заблоковані виклики <code>socket_connect</code> (LSM, <code>-ECONNREFUSED</code>)
11	<code>pending_expired</code>	Записи <code>pending_cgroups</code> , видалені після закінчення TTL

Агент управління зчитує значення лічильників через фоновий цикл `statsLoop` з інтервалом 5 секунд. Для забезпечення сумісності з системами моніторингу кластерного рівня реалізовано HTTP-ендпоінт `/metrics` у форматі

Prometheus (порт 9090 за замовчуванням). Ендпоінт надає лічильники Counter та вимірювальні показники Gauge, зокрема: поточну кількість Pod-об'єктів у whitelist_map, поточну кількість записів у blocklist_map, ефективність придушення подій та фактичну кількість записів у кожній eVPF-карті, отриману через Iterate().

Додатково реалізовано ендпоінти стану: /healthz повертає HTTP 200 після завершення ініціалізації агента; /readyz повертає HTTP 200 лише після успішної синхронізації Informers та реєстрації початкових Pod-об'єктів у whitelist_map. Ці ендпоінти сумісні зі стандартними механізмами перевірки готовності (readinessProbe) та живучості (livenessProbe) оркестратора Kubernetes.

					ІАЛЦ.467200.003 ПЗ	Арк.
						81
Зм.	Арк.	№ докум.	Підпис	Дата		

ВИСНОВОК ДО РОЗДІЛУ

У третьому розділі описано архітектурні та реалізаційні рішення системи OIKESA. Обґрунтовано дворівневу модель взаємодії компонентів простору ядра та простору користувача, що реалізує парадигму пригнічення подій безпосередньо на рівні ядра (Drop-at-Source) та усуває семантичний розрив між рівнем оркестрації Kubernetes і рівнем ядра Linux шляхом використання inode каталогу cgroupfs як числового ідентифікатора cgroup_id — єдиного ключа авторизації в усіх eBPF-картах.

Детально специфіковано підсистему eBPF-карт, що охоплює десять карт із обґрунтованим вибором типів структур даних: BPF_MAP_TYPE_LRU_HASH для whitelist_map, blocklist_map, pending_cgroups, ancestry_map, ns_policy_map, kubepods_map, xdp_blocklist та xdp_blocklist_v6; BPF_MAP_TYPE_PERCPU_ARRAY для stats_map; BPF_MAP_TYPE_RINGBUF для alert_ringbuf. Описано алгоритм захисту від LRU-флуду через дворівневий ліміт заповненості та алгоритм TTL-очищення карти pending_cgroups з порогом PENDING_TTL_NS у 5 секунд, що виконується безпосередньо в eBPF-програмах без участі простору користувача.

Розглянуто реалізацію стеку eBPF-програм, зокрема шестирівневий алгоритм контекстно-залежного придушення; механізм автоматичного вибору між хуками fentry та raw_tracepoint; застосування підсистеми BPF-LSM для синхронного превентивного блокування через хуки bprm_check_security, socket_connect, socket_bind, file_open та task_fix_setuid; XDP та TC-фільтри для мережевої фільтрації; а також механізм відстеження ієрархії контрольних груп через tracepoint sched_process_fork.

Описано алгоритми взаємодії агента управління з Kubernetes API на базі SharedInformerFactory, що охоплюють пакетне додавання Pod-об'єктів через

					ІАЛЦ.467200.003 ПЗ	Арк.
						82
Зм.	Арк.	№ докум.	Підпис	Дата		

BatchUpdate(), обхід cgroupfs для визначення cgroup_id з урахуванням QoS-класів, а також негайне видалення записів при переході Pod-об'єкта у стан Terminating. Деталізовано механізм виявлення контейнерів через containerd gRPC з експоненціальним backoff (від 1 до 30 секунд) та резервним скануванням /proc, алгоритм динамічного прикріплення TC-фільтрів до veth-інтерфейсів через перемикання мережевих просторів імен за допомогою unix.Setns() та Netlink RTM_GETLINK, а також механізм персистентності стану через ConfigMap із асинхронним persist-воркером, drain-логікою (300 мс) та механізмом оптимістичного замикання при збереженні.

					ІАЛЦ.467200.003 ПЗ	Арк.
						83
Зм.	Арк.	№ докум.	Підпис	Дата		

РОЗДІЛ 4

ТЕСТУВАННЯ ТА ПОРІВНЯЛЬНИЙ АНАЛІЗ ПРОДУКТИВНОСТІ РОЗРОБЛЕНОЇ СИСТЕМИ

4.1 Методологія тестування

Метою тестування системи OIKESA є всебічна перевірка двох ключових властивостей, що обґрунтовують доцільність обраного архітектурного підходу: по-перше, ефективність придушення подій безпосередньо в ядрі (suppression efficiency), та по-друге, мінімізація накладних витрат на обчислювальні ресурси (overhead) порівняно з існуючими рішеннями. Оскільки система розроблялась для ресурсно-обмежених IoT середовищ, зокрема платформи NVIDIA Jetson Nano, критерії прийнятності встановлені суворо: споживання CPU агентом не повинно перевищувати 3%, а сукупний обсяг оперативної пам'яті — 70 МБ.

Для досягнення коректних та відтворюваних результатів визначено єдиний інструментарій тестування. У таблиці 4.1 наведено перелік використовуваних засобів вимірювання та їх призначення.

Таблиця 4.1 — Інструментарій тестування OIKESA

Інструмент	Команда / спосіб використання	Що вимірює
pidstat	pidstat -u 1 60 -p <pid>	Споживання CPU агентом у відсотках (посекундно, 60 с)
/proc/<pid>/status	cat /proc/\$PID/status grep -E "VmRSS VmSize"	RSS процесу агента в КБ
/metrics (OIKESA)	curl http://localhost:9090/metrics	Лічильники подій (suppressed, blocked, whitelist тощо)

Продовження таблиці 4.1

Інструмент	Команда / спосіб використання	Що вимірює
bpftool	bpftool map show / map dump	Інспекція eBPF-карт, програм та їх стану
/metrics (OIKESA)	curl http://localhost:9090/metrics	13 stat-лічильників, розміри maps
/bench-memory	curl http://localhost:9090/bench-memory	eBPF maps KB + Go RSS у вигляді JSON
iptables	iptables -I OUTPUT -p tcp --dport 6443 -j DROP	Блокування доступу до kube-apiserver для тестів T-F1, T-R2

Методологія вимірювань передбачає послідовне встановлення OIKESA та кожного з конкурентів (Falco, Tetragon) на одному й тому ж стенді за ідентичних умов навантаження та конфігурації середовища. Усі тести класифіковано за чотирма категоріями: функціональні (T-F), порівняльні (T-C), безпекові (T-S), надійності (T-R). У цьому розділі детально описуються тести категорій T-C, T-S та T-R, оскільки вони безпосередньо підтверджують обґрунтованість архітектурного рішення OIKESA.

4.2 Стенд для тестування (BM та NVIDIA Jetson Nano)

Тестування проводиться у двох різних середовищах, що відповідають типовим сценаріям розгортання системи OIKESA: на стандартній віртуальній машині x86_64 як базовому стенді для розробки (Сценарій А) та на одноплатному комп'ютері NVIDIA Jetson Nano як цільовій IoT-платформі (Сценарій Б).

Сценарій А — Віртуальна машина (x86_64). Даний стенд слугує базовим середовищем для верифікації функціональності та проведення порівняльних бенчмарків з конкурентними рішеннями. Детальна конфігурація наведена в таблиці 4.2.

Таблиця 4.2 — Конфігурація тестового стенду: Сценарій А (ВМ, x86_64)

Параметр	Значення (Сценарій А — ВМ)
Архітектура	x86_64
ОС	Ubuntu 22.04 LTS, ядро Linux 6.17.0-22-generic
vCPU / RAM	4 vCPU / 8 ГБ
Оркестратор	K3s v1.34.6+k3s1, Go v1.24.13 (single-node)
Конкуренти	Falco v0.43 (modern_ebpf), Tetragon v1.7

На даному стенді доступні всі можливості eBPF, включаючи BPF-LSM, fentry/fexit та XDP. Обсяг доступної пам'яті дозволяє запустити одночасно декілька конкурентних агентів для послідовного порівняння, а потужність процесора виключає вплив апаратних обмежень на результати вимірювань.

Сценарій Б — NVIDIA Jetson Nano (ARM64). Це первинне цільове середовище, для якого й оптимізувалась система OIKESA. Оскільки NVIDIA Jetson Nano має обмежену пропускну здатність шини пам'яті типу LPDDR4 та чотири ядра Cortex-A57, що поділяють спільний LLC-кеш, навіть невеликий обсяг зайвої роботи в ядрі може суттєво вплинути на продуктивність. Конфігурація стенду наведена в таблиці 4.3.

Таблиця 4.3 — Конфігурація тестового стенду: Сценарій Б (NVIDIA Jetson Nano, ARM64)

Параметр	Значення (NVIDIA Jetson Nano)
Архітектура	ARM64 (Cortex-A57, 4 ядра @ 1.43 ГГц)
ОС	Ubuntu 20.04 LTS 64-bit (JetPack), ядро Linux 6.1.50-current-media
RAM	4 ГБ LPDDR4
Оркестратор	K3s v1.35.5+k3s1 (single-node)
Конкуренти	Falco v0.43 (modern_ebpf), Tetragon v1.7

Саме на Сценарії Б зосереджені критичні вимоги до ресурсоемності. Оскільки NVIDIA Jetson Nano має обмежену пропускну здатність шини пам'яті, тест T-C2 (Memory footprint comparison) є критичним для підтвердження придатності OIKESA для розгортання в IoT-кластерах.

4.3 Опис тестів продуктивності

Порівняльний аналіз продуктивності є центральним блоком тестування, що дає змогу кількісно обґрунтувати перевагу підходу «drop-at-source» перед традиційними user-space агентами. Всі порівняльні тести виконуються послідовно на одному й тому ж стенді з ідентичними параметрами навантаження.

Тест T-C1 присвячений порівнянню споживання CPU. Метою є визначення накладних витрат на процесор при різних рівнях подійного навантаження. Вимірювання виконуються за допомогою утиліти pidstat -u 1 60, що забезпечує посекундне знімання показників CPU протягом 60 секунд окремо для процесу кожного агента. Тест охоплює два сценарії: Сценарій А — низьке навантаження (10 Pod'ів у режимі очікування, команда sleep 9999) та Сценарій Б — більше навантаження (50 Pod'ів із внутрішньою генерацією ехес-подій через нескінченний цикл виклику /bin/true з інтервалом 50 мс, що відповідає

приблизно 1000 системних викликів `execve` на секунду) для стенду ВМ, детальніше в таблиці 4.4. Такий підхід до генерації навантаження було обрано замість зовнішніх `kubectl exec` викликів, оскільки останні створюють надмірне навантаження на API-сервер K3s, що спотворює результати вимірювань.

Таблиця 4.4 — Результати тесту T-C1: Споживання CPU (%) — Сценарій А (ВМ, x86_64)

Рішення	Сценарій А (10 pod'ів, idle)	Сценарій Б (50 pod'ів, 1000 exec/c)
Falco (eBPF driver)	2.63%	61.28%
Tetragon	0.51%	0.76%
OIKESA	0.05%	0.05%

Аналіз результатів на ВМ (Таблиця 4.4). OIKESA демонструє споживання CPU на рівні 0.05% в обох режимах навантаження, що є у 1225 разів меншим порівняно з Falco при Навантаженні 2 (61.28%) та у 15 разів меншим порівняно з Tetragon (0.76%). Така ефективність досягається завдяки тому, що 100% `execve`-подій відсікається безпосередньо в ядрі — жодна подія не передається в `user-space` для обробки агентом.

Для оцінки поведінки систем у IoT середовищах аналогічний тест було додатково проведено на ARM64-платформі NVIDIA Jetson Nano. Методика вимірювань залишалася ідентичною до стенду ВМ, однак через обмежені апаратні ресурси Cortex-A57 кількість pod'ів у сценарії високого навантаження була зменшена до 20. Результати вимірювань наведено в Таблиці 4.5, а приклади фактичного навантаження CPU для Falco, Tetragon та OIKESA під час виконання тесту — на рисунках 4.1–4.3 відповідно.

```

04:50:24 PM      0      303493      92.00      6.00      0.00      2.00      98.00      1 falco
04:50:24 PM 65532      303524      0.00      0.00      0.00      0.00      0.00      0 falcoctl

Average:      UID      PID      %usr %system %guest %wait %CPU CPU Command
Average:      0      303493      90.87      6.73      0.00      2.97      97.60      - falco
Average:     65532      303524      0.03      0.02      0.00      0.00      0.05      - falcoctl
root@jetson-nano:/home/vii# cat /proc/$(pgrep -f "/usr/bin/falco" | head -1)/status | grep VmRSS

```

Рисунок 4.1 — Споживання CPU агентом Falco під час тесту T-C1 на NVIDIA Jetson Nano

На рисунку 4.1 видно, що агент Falco створює високе навантаження на процесор навіть за короткий інтервал вимірювання, що підтверджує значні накладні витрати user-space обробки подій на ARM64-платформі.

```

05:00:48 PM      UID      PID      %usr %system %guest %wait %CPU CPU Command
05:00:49 PM      0      406391      0.00      0.00      0.00      0.00      0.00      0 hubble-export-s
05:00:49 PM      0      406405      1.00      2.00      0.00      22.00      3.00      3 tail
05:00:49 PM 65532      406452      0.00      0.00      0.00      0.00      0.00      2 tetragon-operat
05:00:49 PM      0      406580      69.00      18.00      0.00      0.00      87.00      2 tetragon
05:00:49 PM      0      437442      0.00      0.00      0.00      0.00      0.00      1 tee

05:00:49 PM      UID      PID      %usr %system %guest %wait %CPU CPU Command
05:00:50 PM      0      406391      0.00      0.00      0.00      0.00      0.00      0 hubble-export-s
05:00:50 PM      0      406405      0.00      1.00      0.00      17.00      1.00      0 tail
05:00:50 PM 65532      406452      0.00      0.00      0.00      0.00      0.00      2 tetragon-operat
05:00:50 PM      0      406580      85.00      22.00      0.00      0.00      107.00      2 tetragon
05:00:50 PM      0      437442      0.00      0.00      0.00      0.00      0.00      1 tee

Average:      UID      PID      %usr %system %guest %wait %CPU CPU Command
Average:      0      406391      0.00      0.00      0.00      0.00      0.00      - hubble-export-s
Average:      0      406405      0.45      1.57      0.00      23.31      2.02      - tail
Average:     65532      406452      0.05      0.03      0.00      0.07      0.08      - tetragon-operat
Average:      0      406580      81.04      20.09      0.00      0.00      101.13      - tetragon
Average:      0      437442      0.03      0.27      0.00      0.40      0.30      - tee
root@jetson-nano:/home/vii# for pid in $(pgrep -f tetragon); do

```

Рисунок 4.2 — Споживання CPU агентом Tetragon під час тесту T-C1 на NVIDIA Jetson Nano

На рисунку 4.2 наведено фрагмент результатів вимірювання споживання CPU агентом Tetragon під час тесту T-C1 на платформі NVIDIA Jetson Nano. У виводі pidstat видно, що процес tetragon періодично досягає навантаження

87.00% та 107.00% CPU на окремих інтервалах вимірювання, тоді як усереднене значення за весь інтервал тестування становить 101.13% у рядку Average. Значення понад 100% свідчать про використання більше одного логічного ядра процесора.

```
04:32:17 PM 0 78302 0.00 0.00 0.00 0.00 0.00 0 oikesa-agent
04:32:18 PM 0 78302 0.00 0.00 0.00 0.00 0.00 0 oikesa-agent
04:32:19 PM 0 78302 0.00 0.00 0.00 0.00 0.00 0 oikesa-agent
04:32:20 PM 0 78302 0.00 0.00 0.00 0.00 0.00 0 oikesa-agent
04:32:21 PM 0 78302 2.00 0.00 0.00 0.00 2.00 0 oikesa-agent
Average: 0 78302 0.05 0.02 0.00 0.00 0.07 - oikesa-agent
root@jetson-nano:/home/vii# Average: 0 78302 0.05 0.02 0.00 0.00 0.07 - oikesa-agent
```

Рисунок 4.3 — Споживання CPU агентом OIKESA під час тесту T-C1 на NVIDIA Jetson Nano

На рисунку 4.3 наведено фрагмент результатів вимірювання утилітою pidstat. Колонки виводу розташовані у такому порядку: UID, PID, %usr, %system, %guest, %wait, %CPU, CPU, Command. Поля відповідають таким параметрам: UID — ідентифікатор користувача, PID — ідентифікатор процесу, %usr — використання CPU у режимі користувача, %system — використання CPU у режимі ядра, %guest — час виконання віртуальних гостьових процесів, %wait — час очікування завершення операцій вводу-виводу, %CPU — сумарне завантаження процесора процесом агента, CPU — номер логічного ядра, Command — назва процесу. Рядок Average містить усереднене значення показників за весь інтервал вимірювання.

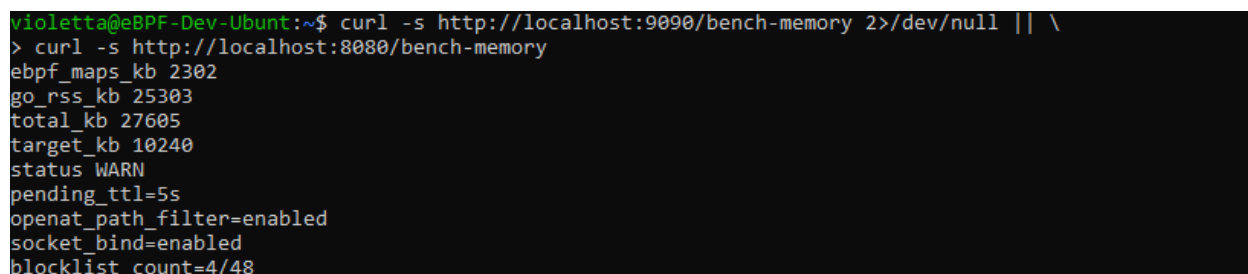
Таблиця 4.5 — Результати тесту T-C1: Споживання CPU (%) — Сценарій Б (NVIDIA Jetson Nano, ARM64)

Рішення	Сценарій А (10 pod'ів, idle)	Сценарій Б (20 pod'ів, 1000 exec/c)
Falco (eBPF driver)	46.93%	97.60%
Tetragon	0.52%	101.13%
OIKESA	0.02%	0.07%

Аналіз результатів на NVIDIA Jetson Nano (Таблиця 4.5). На ARM64-платформі різниця між рішеннями проявляється ще різкіше через обмежену потужність процесора Cortex-A57. При Навантаженні 10 pod'ів Falco вже споживає 46.93% CPU, тоді як OIKESA залишається на рівні 0.02%. При Навантаженні 20 pod'ів Falco досягає 97.60% (фактично повністю завантажує одне ядро), а Tetragon перевищує позначку 101.13%, використовуючи більше одного логічного ядра процесора, що на 4-ядерному Cortex-A57 є критичним для стабільності системи. OIKESA при цьому зберігає споживання CPU на рівні 0.07%, підтверджуючи ефективність підходу «drop-at-source» навіть в умовах жорстких апаратних обмежень.

Тест T-C2: Аналіз обсягу оперативної пам'яті (Memory footprint). Тест вимірює дві складові споживання пам'яті: Resident Set Size (RSS) процесу агента (через /proc/<pid>/status, поле VmRSS) та обсяг eBPF-карт у пам'яті ядра (через bpftool map show та вбудований endpoint /bench-memory).

Результати вимірювання внутрішніх метрик пам'яті OIKESA через вбудований endpoint /bench-memory наведено на Рис. 4.4 на ВМ.



```
violetta@eBPF-Dev-Ubunt:~$ curl -s http://localhost:9090/bench-memory 2>/dev/null || \
> curl -s http://localhost:8080/bench-memory
ebpf_maps_kb 2302
go_rss_kb 25303
total_kb 27605
target_kb 10240
status WARN
pending_ttl=5s
openat_path_filter=enabled
socket_bind=enabled
blocklist_count=4/48
```

Рисунок 4.4 — Внутрішні метрики споживання пам'яті OIKESA через endpoint /bench-memory на ВМ (x86_64)

Результати на ВМ (x86_64). Для OIKESA з 50 Pod'ами розмір whitelist_map визначається кількістю контейнерів: кожен запис зберігає cgroup ID (uint64, 8 байт) як ключ та прапорець дозволу (uint8, 1 байт) як значення. При

50 Pod'ах з одним контейнером кожен (kubectl run без sidecar) кількість записів складає 50, що займає менше 1 КБ у пам'яті ядра. Ring buffer alert_ringbuf має фіксований розмір 2048 КБ. Загальний обсяг eBPF-мап складає ≈ 2049 КБ, де основну частину займає фіксований ring buffer. Єдина ціль OIKESA — сумарне споживання пам'яті (RSS Go-агента + усі eBPF-мапи) менше 70 МБ. Вимірювання RSS виконувалось командою `cat /proc/$PID/status | grep -E "VmRSS|VmSize"`, обсяг eBPF-мап — через `sudo bpftool map show --json` з подальшою агрегацією значень `bytes_memlock` для мап відповідного агента. Для OIKESA з 50 Pod'ами реальний розмір whitelist_map склав менше 1 КБ (50 записів: ключ cgroup_id 8 байт + значення 1 байт), основну частину займає фіксований alert_ringbuf розміром 2048 КБ, загальний обсяг eBPF-мап — 2452 КБ (Рис. 4.5). RSS Go-агента OIKESA склав 59 МБ (VmRSS: 60348 КБ згідно /proc/status) (Рис. 4.6), що у сумі дає 61 МБ — менше встановленої цілі у 70 МБ.

```
violetta@eBPF-Dev-Ubunt:~$ sudo bpftool map show --json | python3 -c "
> import json,sys
> maps = json.load(sys.stdin)
> total = 0
> for m in maps:
>     name = m.get('name','')
>     mem = m.get('bytes_memlock', 0)
>     if any(x in name for x in ['whitelist','alert_ringbuf','blocklist',
>         'ancestry','ns_policy','pending','stats_map','kubepods',
>         'xdp_block']):
>         print(f'{name}: {mem} bytes')
>         total += mem
> print(f'Total OIKESA eBPF maps: {total} bytes = {total/1024:.1f} KB')
> "
```

```
blocklist_map: 6080 bytes
stats_map: 784 bytes
whitelist_map: 82880 bytes
pending_cgroups: 21440 bytes
ancestry_map: 164800 bytes
alert_ringbuf: 2117912 bytes
ns_policy_map: 6080 bytes
kubepods_map: 82880 bytes
xdp_blocklist: 21440 bytes
xdp_blocklist_v: 6592 bytes
Total OIKESA eBPF maps: 2510888 bytes = 2452.0 KB
```

Рисунок 4.5 — Аналіз обсягу пам'яті eBPF-мап агента OIKESA через bpftool

```
violetta@eBPF-Dev-Ubunt:~$ OIKESA_PID=$(pgrep -x oikesa-agent)
violetta@eBPF-Dev-Ubunt:~$ cat /proc/$OIKESA_PID/status | grep -E "VmRSS|VmSize"
VmSize: 1287772 kB
VmRSS: 60348 kB
```

Рисунок 4.6 — Вимірювання споживання оперативної пам'яті (RSS) агентом OIKESA

Для порівняння були проведені аналітичні заміри для інших рішень. RSS процесу Falco склав 125 МБ при 1.2 МБ eBPF-мап (разом 126 МБ) (Рис. 4.7–4.8). Tetragon — 390 МБ при 31.9 МБ eBPF-мап (разом 422 МБ) (Рис. 4.9–4.10). Результати порівняння наведено в Таблиці 4.6.

```
violetta@eBPF-Dev-Ubunt:~$ cat /proc/$(pgrep -f "/usr/bin/falco" | head -1)/status | grep -E "VmRSS|VmSize"
VmSize: 1592500 kB
VmRSS: 128012 kB
violetta@eBPF-Dev-Ubunt:~$ sudo bpftool map show | grep -E "name|bytes_memlock" | paste - -
[sudo] password for violetta:
1: hash name s_snapd_desktop flags 0x0      2: hash name s_firefox_firef flags 0x0
23: hash_of_maps name policy_filter_m flags 0x0      24: hash name policy_1_map flags 0x0
531: hash name s_firmware_upda flags 0x0      539: prog_array name custom_sys_exit flags 0x0
540: prog_array name syscall_exit_ta flags 0x0      541: prog_array name syscall_exit_ex flags 0x0
542: array name interesting_sys flags 0x0      543: array name capture_setting flags 0x0
544: array name auxiliary_maps flags 0x0      545: array name counter_maps flags 0x0
546: array_of_maps name ringbuf_maps flags 0x0      547: prog_array name extra_sched_pro flags 0x0
549: array name bpf_prob.rodata flags 0x400      550: array name bpf_prob.bss flags 0x400
violetta@eBPF-Dev-Ubunt:~$
```

Рисунок 4.7 — Вимірювання споживання оперативної пам'яті (RSS) процесом Falco

```
violetta@eBPF-Dev-Ubunt:~$ sudo bpftool map show --json | python3 -c "
> import json,sys
> maps = json.load(sys.stdin)
> falco_names = ['interesting_sys','capture_setting','auxiliary_maps',
>               'counter_maps','ringbuf_maps','syscall_exit_ta',
>               'syscall_exit_ex','custom_sys_exit','extra_sched_pro',
>               'bpf_prob.rodata','bpf_prob.bss']
> total = 0
> for m in maps:
>     name = m.get('name','')
>     mem = m.get('bytes_memlock', 0)
>     if any(n in name for n in falco_names):
>         print(f'{name}: {mem} bytes')
>         total += mem
> print(f'Total Falco eBPF maps: {total} bytes = {total/1024:.1f} KB')
> "
custom_sys_exit: 288 bytes
syscall_exit_ta: 4360 bytes
syscall_exit_ex: 368 bytes
interesting_sys: 4360 bytes
capture_setting: 296 bytes
auxiliary_maps: 524616 bytes
counter_maps: 680 bytes
ringbuf_maps: 296 bytes
extra_sched_pro: 280 bytes
bpf_prob.rodata: 696320 bytes
bpf_prob.bss: 8192 bytes
Total Falco eBPF maps: 1240056 bytes = 1211.0 KB
violetta@eBPF-Dev-Ubunt:~$
```

Рисунок 4.8 — Аналіз обсягу пам'яті eBPF-мап Falco

```
violetta@eBPF-Dev-Ubunt:~$ for pid in $(pgrep -f tetragon); do
> echo -n "PID $pid: "
> grep -E "VmRSS|Name" /proc/$pid/status | tr '\n' ' '
> echo
> done
PID 1616712: Name:      hubble-export-s VmRSS:      40 kB
PID 1616734: Name:      tail VmRSS:      144 kB
PID 2716381: Name:      tetragon-operat VmRSS:      16076 kB
PID 2716578: Name:      tetragon VmRSS:      399008 kB
```

Рисунок 4.9 — Вимірювання споживання оперативної пам'яті (RSS) агентом Tetragon

```
violetta@eBPF-Dev-Ubunt:~$ sudo bpftool map show --json | python3 -c "
> import json,sys
> maps = json.load(sys.stdin)
> total = 0
> for m in maps:
>     name = m.get('name','')
>     mem = m.get('bytes_memlock', 0)
>
>     if any(x in name for x in ['execve','tetragon','tg_','kprobe','process']):
>         print(f'{name}: {mem} bytes')
>         total += mem
> print(f'Total Tetragon eBPF maps: {total} bytes = {total/1024:.1f} KB')
> "
[sudo] password for violetta:
tg_conf_map: 312 bytes
execve_map: 31724224 bytes
tg_execve_joine: 721856 bytes
execve_map_stat: 384 bytes
tg_execve_joine: 384 bytes
execve_calls: 280 bytes
execve_map_upda: 131344 bytes
tg_stats_map: 57616 bytes
tg_mbset_map: 338112 bytes
tg_mbset_gen: 272 bytes
tg_errmetrics_m: 123840 bytes
tg_rb_events: 275736 bytes
tg_parents_bin: 1824 bytes
execve_val: 3856 bytes
tg_cgtracker_ma: 1328 bytes
execve_msg_heap: 25168 bytes
tg_binary_heap: 3440 bytes
execve_heap: 16720 bytes
tg_cgtracker_ma: 1328 bytes
tg_parents_bin: 1824 bytes
execve_heap: 16720 bytes
Total Tetragon eBPF maps: 33446568 bytes = 32662.7 KB
violetta@eBPF-Dev-Ubunt:~$
```

Рисунок 4.10 — Аналіз обсягу пам'яті eBPF-мап Tetragon

Розбіжності між показниками системи OIKESA на Рис. 4.1 та Рис. 4.2 – 4.3 є очікуваними та зумовлені різною методологією вимірювання. Поле

go_rss_kb (25303 КБ) у відповіді endpoint /bench-memory відображає лише внутрішню статистику Go runtime, отриману через runtime.ReadMemStats, яка охоплює виключно heap та stack Go-програми. Натомість поле VmRSS у /proc/status (60348 КБ) фіксує повний Resident Set Size процесу в системі, включаючи shared libraries, mmap-регіони та сам бінарний файл агента. Таким чином, розбіжність між 25 МБ та 60 МБ є архітектурно обґрунтованою і не свідчить про некоректність жодного з вимірювань.

Аналогічно, незначна розбіжність між eBPF maps (2302 КБ) з endpoint /bench-memory та результатом bpftool (2452 КБ) пояснюється тим, що bpftool звітує значення поля bytes_memlock — тобто обсяг пам'яті, заблокованої (зарезервованої) ядром для відповідної мапи, тоді як endpoint /bench-memory відображає фактичний обсяг даних, що зберігаються у мапах на момент вимірювання. Різниця у ~150 КБ відповідає типовому вирівнюванню сторінок пам'яті ядра і є несуттєвою з практичної точки зору. Для порівняльного аналізу в Таблиці 4.6 використовуються значення /proc/status (VmRSS) та bpftool як стандартизовані системні метрики.

Таблиця 4.6 — Результати тесту T-C2: Обсяг пам'яті (МБ) — ВМ (x86_64, 50 Pod'ів)

Рішення	RSS агента (МБ)	eBPF maps (МБ)	Разом (МБ)
Falco	125	1.2	126
Tetragon	390	31.9	422
OIKESA	59	2.4	61

Для оцінки поведінки систем в умовах обмежених апаратних ресурсів аналогічний тест T-C2 було додатково проведено на ARM64-платформі NVIDIA Jetson Nano. Методика вимірювання залишалась ідентичною до стенду ВМ (аналіз VmRSS процесів та обсягу eBPF-мап через bpftool), однак через

обмежений обсяг оперативної пам'яті та нижчу обчислювальну потужність Cortex-A57 кількість род'ів у сценарії навантаження була зменшена з 50 до 20. Фактичні значення RSS процесів та обсягів eBPF-мап для кожного рішення наведено на рисунках 4.11–4.16.

```
root@jetson-nano:/home/vii# cat /proc/$(pgrep -f "/usr/bin/falco" | head -1)/status | grep VmRSS
VmRSS:      105384 kB
root@jetson-nano:/home/vii# sudo bpftool map show --json | python3 -c "
```

Рисунок 4.11 — Вимірювання споживання оперативної пам'яті (VmRSS) процесом Falco на NVIDIA Jetson Nano

```
custom_sys_exit: 4096 bytes
syscall_exit_ta: 4096 bytes
syscall_exit_ex: 4096 bytes
interesting_sys: 4096 bytes
capture_setting: 4096 bytes
auxiliary_maps: 528384 bytes
counter_maps: 4096 bytes
ringbuf_maps: 4096 bytes
extra_sched_pro: 4096 bytes
extra_sched_pro: 4096 bytes
bpf_prob.rodata: 712704 bytes
bpf_prob.bss: 4096 bytes
Total Falco eBPF maps: 1282048 bytes = 1252.0 KB
root@jetson-nano:/home/vii# kubectl delete pods --all --force --grace-period=0
```

Рисунок 4.12 — Аналіз обсягу пам'яті eBPF-мап Falco на NVIDIA Jetson Nano

```
PID 406391: Name:      hubble-export-s VmRSS:      4 kB
PID 406405: Name:      tail VmRSS:      4 kB
PID 406452: Name:      tetragon-operat VmRSS:      44880 kB
PID 406580: Name:      tetragon VmRSS:      434628 kB
root@jetson-nano:/home/vii#
```

Рисунок 4.13 — Вимірювання споживання оперативної пам'яті (VmRSS) агентами Tetragon та tetragon-operator на NVIDIA Jetson Nano

```

execve_map: 2902272 bytes
tg_execve_joine: 196608 bytes
execve_map_stat: 4096 bytes
tg_execve_joine: 4096 bytes
execve_calls: 4096 bytes
execve_map_upda: 135168 bytes
tg_stats_map: 61440 bytes
tg_mbset_map: 270336 bytes
tg_mbset_gen: 4096 bytes
tg_errmetrics_m: 49152 bytes
tg_rb_events: 0 bytes
tg_parents_bin: 4096 bytes
tg_cgtracker_ma: 4096 bytes
execve_val: 4096 bytes
execve_msg_heap: 28672 bytes
tg_parents_bin: 4096 bytes
tg_cgtracker_ma: 4096 bytes
tg_binary_heap: 4096 bytes
execve_heap: 20480 bytes
execve_heap: 20480 bytes
Total Tetragon eBPF maps: 30449664 bytes = 29736.0 KB
root@jetson-nano:/home/vii# ^C

```

Рисунок 4.14 — Аналіз обсягу пам'яті eBPF-мап Tetragon на NVIDIA Jetson Nano

```

root@jetson-nano:/home/vii# OIKESA_PID=$(pgrep -x "oikesa-agent")
root@jetson-nano:/home/vii# cat /proc/$OIKESA_PID/status | grep -E "VmRSS|VmSize|VmPeak"
VmPeak: 1380708 kB
VmSize: 1380708 kB
VmRSS: 61240 kB
root@jetson-nano:/home/vii# sudo bpftool map show --json | python3 -c "

```

Рисунок 4.15 — Вимірювання споживання оперативної пам'яті (VmRSS) агентом OIKESA на NVIDIA Jetson Nano

```

blocklist_map: 4096 bytes
stats_map: 4096 bytes
whitelist_map: 16384 bytes
pending_cgroups: 4096 bytes
ancestry_map: 32768 bytes
alert_ringbuf: 0 bytes
ns_policy_map: 4096 bytes
kubepods_map: 16384 bytes
xdp_blocklist: 4096 bytes
xdp_blocklist_v: 4096 bytes
Total OIKESA eBPF maps: 90112 bytes = 88.0 KB
root@jetson-nano:/home/vii# curl -s http://localhost:9090/bench-memory 2>/dev/null

```

Рисунок 4.16 — Аналіз обсягу пам'яті eBPF-мап OIKESA на NVIDIA Jetson Nano

Результати на NVIDIA Jetson Nano (ARM64). Вимірювання проводились при 20 Pod'ах у Сценарії Б. Для OIKESA значення VmRSS процесу агента склало 61 240 КБ згідно /proc/status (Рис. 4.15). Загальний обсяг eBPF-мап становив 90 112 байт (Рис. 4.16). Таким чином, сумарне споживання пам'яті OIKESA склало приблизно 60.1 МБ, що відповідає встановленому цільовому показнику менше 70 МБ. Для Falco значення VmRSS становило 105 384 КБ (Рис. 4.11), а сумарний обсяг eBPF-мап — 1 282 048 байт (Рис. 4.12). Загальне споживання пам'яті Falco склало приблизно 104.1 МБ. Для Tetragon основний процес tetragon споживав 434 628 КБ, а процес tetragon-operator — 44 880 КБ (Рис. 4.13). Сукупне RSS-споживання становило приблизно 468 МБ. Обсяг eBPF-мап Tetragon склав 30 449 664 байт, причому найбільшу частину займала ехесве_мап розміром 29 622 272 байти (Рис. 4.14). Загальне споживання пам'яті Tetragon становило приблизно 497.3 МБ.

Таблиця 4.7 — Результати тесту Т-С2: Обсяг пам'яті (МБ) — NVIDIA Jetson Nano (ARM64, 20 Pod'ів)

Рішення	RSS агента (МБ)	eBPF maps (МБ)	Разом (МБ)
Falco	103	1.22	104.2
Tetragon	468	29	497
OIKESA	60	0.09	60.1

4.4 Функціональне тестування безпеки

Функціональне тестування безпеки спрямоване на підтвердження коректності механізмів примусового виконання (enforcement) через BPF-LSM та перевірку відсутності хибних спрацювань (false positives) під час стандартних операцій Kubernetes.

Тест Т-S1: Перевірка блокування через BPF-LSM (execve та socket_connect). BPF-LSM хуки lsm/bprm_check_security та lsm/socket_connect є основним механізмом примусового виконання політики безпеки в OIKESA. На відміну від систем спостереження (Falco, Tracee), що лише реєструють події, OIKESA блокує заборонені операції безпосередньо в ядрі, повертаючи процесу код помилки -EPERM або -ECONNREFUSED. Додатково реалізовано хук lsm/socket_bind, що перешкоджає сценаріям зворотних з'єднань (reverse shell) через блокування операції bind() у контейнерах зі списку blocklist_map.

Для проведення тесту створено тестовий Pod victim та зафіксовано початковий стан: лічильники oikesa_blocked_total=0 та oikesa_blocklist_count=4, що видно на Рис. 4.17.

```
^Cvioletta@BPF-Dev-Ubunt:~$ kubectl delete pod victim --ignore-not-found
pod "victim" deleted from default namespace
violetta@BPF-Dev-Ubunt:~$ kubectl run victim --image=docker.io/library/alpine:latest \
> --restart=Never \
> --image-pull-policy=Never \
> -- sleep 9999
pod/victim created
violetta@BPF-Dev-Ubunt:~$ kubectl get pod victim -w
NAME      READY   STATUS    RESTARTS   AGE
victim    1/1     Running   0           8s
^Cvioletta@BPF-Dev-Ubunt:~$ curl -s http://localhost:9090/metrics | grep -E "oikesa_blocked_total|oikesa_blocklist_count|oikesa_suppressed_total"
# HELP oikesa_suppressed_total execve suppressed in-kernel
# TYPE oikesa_suppressed_total counter
oikesa_suppressed_total 0
# HELP oikesa_blocked_total BPF-LSM blocked events
# TYPE oikesa_blocked_total counter
oikesa_blocked_total 0
# HELP oikesa_blocklist_count current blocklist entries
# TYPE oikesa_blocklist_count gauge
oikesa_blocklist_count 4
```

Рис. 4.17 — Створення Pod та початковий стан лічильників OIKESA

Методика перевірки блокування передбачає наступні кроки: помістити цільовий контейнер до blocklist через REST API (curl -X POST http://localhost:9090/blocklist/add?cgroup=<ID>); виконати спробу запуску забороненого файлу у заблокованому контейнері (/bin/ls) — отримати «Permission denied» (хук lsm/bprm_check_security); перевірити зростання лічильника oikesa_blocked_total у /metrics після кожної спроби; перевірити наявність запису [ALERT] type=BLOCKED у журналі агента.

Контейнер default/victim (cgroup=100166) було додано до blocklist через REST API, після чого спроба виконати /bin/lс завершилась помилкою «operation not permitted», а лічильник oikesa_blocked_total зріс до 12, що підтверджує коректну роботу хука lsm/bprm_check_security (Рис. 4.18). Факт додавання зафіксовано у журналі агента рядком [!] BLOCKLIST ADD cgroup=100166 count=5/48 (Рис. 4.19). Алерти типу [ALERT] type=BLOCKED з відповідним cgroup та ім'ям пода default/victim також з'явилися у журналі агента, що підтверджує генерацію сповіщень про кожну заблоковану спробу (Рис. 4.20).

```
violetta@eBPF-Dev-Ubunt:~$ curl -s -X POST "http://localhost:9090/blocklist/add?cgroup=100166"
ok cgroup=100166 added to blocklist count=5/48
violetta@eBPF-Dev-Ubunt:~$ kubectl exec victim -- /bin/lс
exec /bin/lс: operation not permitted
command terminated with exit code 255
violetta@eBPF-Dev-Ubunt:~$ curl -s http://localhost:9090/metrics | grep -E "oikesa_blocked_total|oikesa_blocklist_count"
# HELP oikesa_blocked_total BPF-LSM blocked events
# TYPE oikesa_blocked_total counter
oikesa_blocked_total 12
# HELP oikesa_blocklist_count current blocklist entries
# TYPE oikesa_blocklist_count gauge
oikesa_blocklist_count 5
violetta@eBPF-Dev-Ubunt:~$ kubectl exec victim -- sh -c "nc -l -p 4444"
exec /bin/sh: operation not permitted
command terminated with exit code 255
```

Рис. 4.18 — Додавання контейнера до blocklist_map через REST API, блокування спроби виконати /bin/lс та зростання лічильника oikesa_blocked_total до 12 після блокування ехесве

```
[STATS] execve: S:4 A:0 | openat: S:30102 A:0 | blocked:0 xdp:0 tc:0 | privesc:0 connect
:0 ancestry:0 conn_blk:0 | bind_blk:0 pend_exp:0 | eff:100.0% pods:12 bl_count:4/48
18:51:03 blocklist.go:43: [!] BLOCKLIST ADD cgroup=100166 count=5/48
18:51:03 http.go:182: [HTTP] /blocklist/add cgroup=100166 count=5/48
18:51:03 blocklist_persist.go:93: [*] BLOCKLIST PERSIST updated ConfigMap entries=5
[STATS] execve: S:4 A:0 | openat: S:30293 A:0 | blocked:0 xdp:0 tc:0 | privesc:0 connect
```

Рис. 4.19 — Повідомлення у журналі агента про додавання групи до Block List

```
[STATS] execve: S:4 A:0 | openat: S:33635 A:0 | blocked:0 xdp:0 tc:0 | privesc:0 connect
:0 ancestry:0 conn_blk:0 | bind_blk:0 pend_exp:0 | eff:100.0% pods:12 bl_count:5/48
18:51:43 alerts.go:95: [ALERT] type=BLOCKED PID:3376341 cgroup:100166 comm:runc pod:default/victim
18:51:43 alerts.go:95: [ALERT] type=BLOCKED PID:3376348 cgroup:100166 comm:runc:[2:INIT] pod:default/victim
18:51:43 alerts.go:95: [ALERT] type=BLOCKED PID:3376348 cgroup:100166 comm:runc:[2:INIT] pod:default/victim
18:51:43 containerd.go:142: [-] containerd task/exit containerID=5764f0889914
```

Рис 4.20 — Алерти у журналі агента про блокування спроби виконати /bin/lс завершилась помилкою

Окремо перевіряється механізм відновлення blocklist після перезапуску агента: оскільки eBPF-карти не переживають рестарт Go-процесу, список заблокованих cgroup'ів зберігається у Kubernetes ConfigMap (oikesa-blocklist-state у просторі імен kube-system) і відновлюється при ініціалізації. Після перезапуску агент вивів рядок [*] BLOCKLIST RESTORED 5/5 entries from ConfigMap, що підтверджує повне відновлення стану (Рис. 4.21). Для перевірки того, що блокування продовжує діяти після рестарту, повторно виконано команду kubectl exec victim -- /bin/ls — результат ідентичний: «operation not permitted» (Рис. 4.22). Тест T-S1 підтверджує блокування та відновлення стану blocklist_map.

```
18:54:20 alerts.go:47: [*] Alert ringbuf consumer started
18:54:20 containerd.go:78: [*] containerd gRPC connected — subscribing task/start + task/exit
18:54:21 blocklist_persist.go:138: [*] BLOCKLIST RESTORED 5/5 entries from ConfigMap kube-system/oikesa-blocklist-state
18:54:21 informers.go:138: [*] Waiting for cache sync (pods + namespaces)...
18:54:21 whitelist.go:145: [*] WHITELIST BATCH ADD: 13 cgroup entries for 6 pods
```

Рис. 4.21 — Відновлення blocklist_map після перезапуску агента

```
Violetta@eBPF-Dev-Ubunt:~$ kubectl get configmap oikesa-blocklist-state -n kube-system -o yaml | grep -A5 "data:"
data:
  blocklist.json: '{"100166":true,"13894":true,"13973":true,"35367":true,"8514":true}'
  count: "5"
  updated_at: "2026-05-06T18:51:03Z"
kind: ConfigMap
metadata:
  creationTimestamp: "2026-04-21T22:58:05Z"
  labels:
    app: oikesa-agent
    oikesa/managed: blocklist-state
    name: oikesa-blocklist-state
Violetta@eBPF-Dev-Ubunt:~$ kubectl exec victim -- /bin/ls
exec /bin/ls: operation not permitted
command terminated with exit code 255
```

Рис. 4.22 — Блокування після перезапуску агента: стан blocklist_map відновлено коректно

Хоча в даній роботі демонстрація блокування проводиться вручну через API для наочності тестування, архітектура OIKESA передбачає інтеграцію з автоматизованими системами реагування (SOAR), які переводять контейнер у статус blocked автоматично при виявленні першої аномалії.

Тест Т-С3: Перевірка хибних спрацювань (False Positive Rate). Orchestrator-aware підхід OIKESA передбачає безпосередню синхронізацію `whitelist_map` з Pod lifecycle Kubernetes через механізм Informers, що теоретично забезпечує нульову частоту хибних спрацювань (FPR) для легітимних Pod'ів. Метою тесту Т-С3 є емпірична перевірка даної властивості в умовах реального кластера K3s під навантаженням.

Для проведення тесту агент OIKESA запущено у штатному режимі; deployment `test-app` із двома репліками переведено у стан `Running`, всі Pod'и попередньо зареєстровано у `whitelist_map`. Початковий стан фіксується значенням лічильника `oikesa_blocked_total` = 0 до початку будь-яких операцій. Послідовно виконуються чотири штатні операції Kubernetes.

Операція 1 — `kubectl rollout restart`. Виконується `rolling restart` deployment/`test-app`. Під час операції Kubernetes Informer послідовно генерує події `Delete` (старі Pod'и) та `Add` (нові Pod'и). Агент OIKESA автоматично видаляє `cgroup_id` завершених Pod'ів з `whitelist_map` та реєструє `cgroup_id` нових Pod'ів у момент переходу до стану `Running` — до фактичного виконання будь-яких процесів у контейнері. Вивід `kubectl rollout status` підтвердив успішне завершення операції; лічильник `oikesa_blocked_total` після `rollout restart` залишився рівним 0 (Рис. 4.23).

```
violetta@eBPF-Dev-Ubunt:~$ kubectl create deployment test-app --image=nginx:alpine --replicas=2
deployment.apps/test-app created
violetta@eBPF-Dev-Ubunt:~$ kubectl wait --for=condition=available deployment/test-app --timeout=60s
deployment.apps/test-app condition met
violetta@eBPF-Dev-Ubunt:~$ kubectl get pods -o wide
NAME READY STATUS RESTARTS AGE IP NODE NOMINATED NODE READINESS GATES
test-app-5cbd5b6ffc-5qgkc 1/1 Running 0 26s 10.42.0.147 ebpf-dev-ubunt <none> <none>
test-app-5cbd5b6ffc-wmwsz 1/1 Running 0 26s 10.42.0.148 ebpf-dev-ubunt <none> <none>
violetta@eBPF-Dev-Ubunt:~$ curl -s http://localhost:9090/metrics | grep oikesa_blocked_total
# HELP oikesa_blocked_total BPF-LSM blocked events
# TYPE oikesa_blocked_total counter
oikesa_blocked_total 0
violetta@eBPF-Dev-Ubunt:~$ kubectl rollout restart deployment/test-app
deployment.apps/test-app restarted
violetta@eBPF-Dev-Ubunt:~$ kubectl rollout status deployment/test-app
Waiting for deployment "test-app" rollout to finish: 1 old replicas are pending termination...
Waiting for deployment "test-app" rollout to finish: 1 old replicas are pending termination...
deployment "test-app" successfully rolled out
violetta@eBPF-Dev-Ubunt:~$ curl -s http://localhost:9090/metrics | grep oikesa_blocked_total
# HELP oikesa_blocked_total BPF-LSM blocked events
# TYPE oikesa_blocked_total counter
oikesa_blocked_total 0
```

Рисунок 4.23 — Початковий стан тесту Т-С3 та відсутність хибних спрацювань після `kubectl rollout restart`

Операція 2 — `kubectl scale` від 0 до 5. Deployment масштабується до нуля реплік, після чого збільшується до п'яти. При масштабуванні до нуля всі `cgroup_id` вилючаються з `whitelist_map`; при зворотному масштабуванні до п'яти — нові `cgroup_id` реєструються синхронно з переходом Pod'ів у стан Running. Після завершення операції вивід `kubectl rollout status` підтверджує успішний rollout; `oikesa_blocked_total` = 0 (Рис. 4.24).

```
violetta@eBPF-Dev-Ubunt:~$ curl -s http://localhost:9090/metrics | grep oikesa_blocked_total
# HELP oikesa_blocked_total BPF-LSM blocked events
# TYPE oikesa_blocked_total counter
oikesa_blocked_total 0
violetta@eBPF-Dev-Ubunt:~$ kubectl scale deployment/test-app --replicas=0
deployment.apps/test-app scaled
violetta@eBPF-Dev-Ubunt:~$ kubectl get pods -w
^Cvioletta@eBPF-Dev-Ubunt:~$ kubectl scale deployment/test-app --replicas=5
deployment.apps/test-app scaled
violetta@eBPF-Dev-Ubunt:~$ kubectl rollout status deployment/test-app
deployment "test-app" successfully rolled out
violetta@eBPF-Dev-Ubunt:~$ curl -s http://localhost:9090/metrics | grep oikesa_blocked_total
# HELP oikesa_blocked_total BPF-LSM blocked events
# TYPE oikesa_blocked_total counter
oikesa_blocked_total 0
violetta@eBPF-Dev-Ubunt:~$ kubectl get pods
```

Рисунок 4.24 — Відсутність хибних спрацювань при масштабуванні deployment від 0 до 5 реплік

Операція 3 — Horizontal Pod Autoscaler (HPA) з генератором навантаження. Для `deployment/test-app` налаштовано HPA (`--min=2`, `--max=10`, `--cpu-percent=50`) та запущено Pod load-gen на образі `busybox`, який безперервно відправляє HTTP-запити до сервісу (`wget` у циклі). Протягом 60 секунд активного навантаження виконувалися тисячі системних викликів. Слід зазначити, що `metrics-server` в тестовому середовищі не зміг зібрати CPU-метрики з Pod'ів (значення TARGETS відображається як `cpu: <unknown>/50%`), внаслідок чого HPA не прийняв рішення про зміну кількості реплік — вона залишилась рівною 5 протягом усього тесту. Проте для цілей тесту Т-С3 факт спрацювання масштабування є другорядним: ключовим є те, що агент OIKESA коректно зареєстрував Pod load-gen у `whitelist_map` одразу після переходу до

стану Running та не блокував жодної з тисяч системних подій, ініційованих генератором навантаження. Вивід `kubectl get pods` підтверджує наявність шести Pod'ів у стані Running (load-gen та п'ять реплік test-app); `oikesa_blocked_total` = 0 (Рис. 4.25).

```
Violetta@eBPF-Dev-Ubunt:~$ kubectl autoscale deployment test-app --min=2 --max=10 --cpu-percent=50
Flag --cpu-percent has been deprecated, Use --cpu with percentage or resource quantity format (e.g., '70%' for util
horizontalpodautoscaler.autoscaling/test-app autoscaled
Violetta@eBPF-Dev-Ubunt:~$ kubectl get hpa
NAME          REFERENCE          TARGETS          MINPODS  MAXPODS  REPLICAS  AGE
test-app      Deployment/test-app  cpu: <unknown>/50%  2         10        5         33s
Violetta@eBPF-Dev-Ubunt:~$ curl -s http://localhost:9090/metrics | grep oikesa_blocked_total
# HELP oikesa_blocked_total BPF-LSM blocked events
# TYPE oikesa_blocked_total counter
oikesa_blocked_total 0
Violetta@eBPF-Dev-Ubunt:~$ kubectl run load-gen --image=busybox --restart=Never -- \
> sh -c "while true; do wget -q -O- http://test-app; done"
pod/load-gen created
Violetta@eBPF-Dev-Ubunt:~$ sleep 60
Violetta@eBPF-Dev-Ubunt:~$ kubectl get hpa
NAME          REFERENCE          TARGETS          MINPODS  MAXPODS  REPLICAS  AGE
test-app      Deployment/test-app  cpu: <unknown>/50%  2         10        5         2m4s
Violetta@eBPF-Dev-Ubunt:~$ kubectl get pods
NAME          READY   STATUS    RESTARTS   AGE
load-gen      1/1     Running   0          86s
test-app-75bcd6fd-7ftcr  1/1     Running   0          2m45s
test-app-75bcd6fd-cg9r7  1/1     Running   0          2m45s
test-app-75bcd6fd-cmwvw  1/1     Running   0          2m45s
test-app-75bcd6fd-n9vtn  1/1     Running   0          2m45s
test-app-75bcd6fd-wvwvz  1/1     Running   0          2m45s
Violetta@eBPF-Dev-Ubunt:~$ curl -s http://localhost:9090/metrics | grep oikesa_blocked_total
# HELP oikesa_blocked_total BPF-LSM blocked events
# TYPE oikesa_blocked_total counter
oikesa_blocked_total 0
```

Рисунок 4.25 — Відсутність хибних спрацювань при роботі НРА та генератора навантаження load-gen

Операція 4 — Pod rescheduling. Один із Pod'ів (test-app-75bcd6fd-7ftcr) примусово видаляється командою `kubectl delete pod`; планувальник K3s автоматично запускає новий Pod-замінник (test-app-75bcd6fd-76v55, вік 9s у виводі `kubectl get pods -w`). Агент OIKESA видаляє `cgroup_id` видаленого Pod'a та реєструє `cgroup_id` нового Pod'a у `whitelist_map` без будь-якого ручного втручання. Показник `oikesa_blocked_total` після завершення rescheduling залишився рівним 0 (Рис. 4.26).

```

Violetta@eBPF-Dev-Ubunt:~$ curl -s http://localhost:9090/metrics | grep oikesa_blocked_total
# HELP oikesa_blocked_total BPF-LSM blocked events
# TYPE oikesa_blocked_total counter
oikesa_blocked_total 0
Violetta@eBPF-Dev-Ubunt:~$ POD=$(kubectl get pods -l app=test-app -o name | head -1)
Violetta@eBPF-Dev-Ubunt:~$ echo "Delete Pod: $POD"
Delete Pod: pod/test-app-75bcd6fd-7ftcr
Violetta@eBPF-Dev-Ubunt:~$ kubectl delete $POD
pod "test-app-75bcd6fd-7ftcr" deleted from default namespace
Violetta@eBPF-Dev-Ubunt:~$ kubectl get pods -w
NAME                                READY    STATUS    RESTARTS   AGE
test-app-75bcd6fd-76v55             1/1     Running   0          9s
test-app-75bcd6fd-cg9n7             1/1     Running   0          4m28s
test-app-75bcd6fd-cmwvw             1/1     Running   0          4m28s
test-app-75bcd6fd-n9vtn             1/1     Running   0          4m28s
test-app-75bcd6fd-wvwvz             1/1     Running   0          4m28s
^CVioletta@eBPF-Dev-Ubunt:~$ curl -s http://localhost:9090/metrics | grep oikesa_blocked_total
# HELP oikesa_blocked_total BPF-LSM blocked events
# TYPE oikesa_blocked_total counter
oikesa_blocked_total 0

```

Рисунок 4.26 — Відсутність хибних спрацювань при примусовому видаленні та автоматичному перезапуску Pod'a (rescheduling)

В результаті всіх чотирьох операцій лічильник `oikesa_blocked_total` залишився рівним 0 протягом усього тесту, що підтверджує відсутність хибних спрацювань.

Щодо порівняння з Falco та Tetragon. Falco 0.43 (modern_ebpf) не генерує алертів на події Pod lifecycle за замовчуванням — його стандартний ruleset орієнтований виключно на виявлення підозрілої поведінки всередині вже запусканих контейнерів (shell spawn, sensitive file access, privilege escalation). Під час тестування підтверджено, що Falco коректно фіксує події типу «A shell was spawned in a container», однак не реагує на rollout restart, масштабування або rescheduling; метрика FPR для lifecycle-операцій є нерелевантною для його моделі загроз. Tetragon 1.7 використовує принципово іншу архітектуру: він логує всі syscall-події (process_exec, process_exit, process_kprobe) як телеметрію без вбудованої класифікації на «алерт» чи «норму»; кожна lifecycle-операція генерує десятки подій типу process_exec, які оператор мусить фільтрувати самотійно через TracingPolicy. Таким чином, пряме числове порівняння FPR

між OIKESA, Falco та Tetragon є методологічно некоректним через принципову різницю в моделях виявлення.

Нульовий показник хибних спрацювань OIKESA досягається завдяки orchestrator-aware архітектурі: Pod lifecycle (Add / Update / Delete) через Kubernetes Informers безпосередньо відображається на операції з whitelist_map. При видаленні Pod'a cgroup_id миттєво вилючається з карти, а cgroup_id нового Pod'a реєструється у момент події «Running» — до фактичного виконання будь-яких процесів у контейнері. Паралельно оновлюється kubernetes_map, що дозволяє eBPF-програмі коректно відрізнити K8s-контейнери від хост-процесів і не генерувати зайвих сповіщень для системних демонів. На відміну від підходів на основі евристичних правил, OIKESA не потребує жодного додаткового налаштування фільтрів для досягнення нульового FPR при стандартних операціях оркестратора.

4.5 Тестування надійності та Edge-випадків

Тестування надійності (resilience testing) спрямоване на перевірку поведінки системи OIKESA в умовах навмисних або ненавмисних аномалій: LRU-флуду eBPF-карт, втрати зв'язку з компонентами Kubernetes, примусової зупинки агента та специфічних обмежень ARM64-платформи.

Тест T-R1: Захист від LRU-флуду (blocklist flood attack). Атака типу LRU-флуд є специфічною вразливістю eBPF-систем, що використовують карти з типом BPF_MAP_TYPE_LRU_HASH: зловмисник, що має можливість створювати велику кількість контейнерів, може навмисно заповнити blocklist_map легітимними cgroup_id, витіснивши справжні записи про заблоковані контейнери через механізм LRU eviction. У результаті enforcement-блокування для цих cgroup'ів буде знято.

					ІАЛЦ.467200.003 ПЗ	Арк.
						106
Зм.	Арк.	№ докум.	Підпис	Дата		

Архітектурне рішення OIKESA для протидії цій атаці полягає у дворівневому захисті. На рівні ядра: `blocklist_map` оголошена з типом `BPF_MAP_TYPE_LRU_HASH` та максимальною місткістю `BLOCKLIST_MAX` дорівнює 64 записи. На рівні user-space: Go-лічильник `blocklistCount` обмежує кількість активних записів до `MaxBlocklistEntries` дорівнює 48. Залишок у 16 слотів виконує роль «буфера LRU eviction» — навіть при максимальному заповненні ядру ніколи не доведеться витіснити легітимні блокуючі записи. Синхронізація лічильника `blocklistCount` захищена окремим м'ютексом `blocklistMu`, що виключає `race conditions` у конкурентному середовищі.

Методика тесту T-R1: симулювати флуд `blocklist` шляхом послідовного додавання 48 фіктивних `cgroup`-ідентифікаторів через REST API (`/blocklist/add?cgroup=N`); перевірити, що після 48-го запису система повертає помилку «`blocklist full`» та відмовляється додавати нові записи; перевірити через `bpftool map dump`, що кількість записів у `blocklist_map` не перевищила 64; перевірити, що перший доданий `cgroup` (`sentinel cgroup=100001`) залишився у карті після флуду та не був витіснений механізмом LRU eviction.

Результати тесту T-R1. Перед початком тесту підтверджено чистий стан: метрика `oikesa_blocklist_count` дорівнює 0, а BPF-карта `blocklist_map` (`id=1992`) має тип `lru_hash` та `max_entries=64` при нульовій кількості фактичних записів (Рис. 4.27).

```

violetta@eBPF-Dev-Ubunt:~$ curl -s http://localhost:9090/metrics | grep -E "oikesa_blocklist"
# HELP oikesa_blocklist_count current blocklist entries
# TYPE oikesa_blocklist_count gauge
oikesa_blocklist_count 0
# HELP oikesa_blocklist_max maximum blocklist capacity
# TYPE oikesa_blocklist_max gauge
oikesa_blocklist_max 48
violetta@eBPF-Dev-Ubunt:~$ sudo bpftool map show | grep -A5 "blocklist"
1992: lru_hash name blocklist_map flags 0x0
      key 8B value 1B max_entries 64 memlock 6080B
      btf_id 1581
1993: percpu_array name stats_map flags 0x0
      key 4B value 8B max_entries 13 memlock 784B
      btf_id 1582
--
2000: lru_hash name xdp_blocklist flags 0x0
      key 4B value 1B max_entries 256 memlock 21440B
      btf_id 1598
2001: lru_hash name xdp_blocklist_v flags 0x0
      key 16B value 1B max_entries 64 memlock 6592B
      btf_id 1599

```

Рисунок 4.27 — Початковий стан blocklist та параметри BPF-карти перед тестом T-R1

На першому кроці до blocklist вручну додано sentinel-запис cgroup=100001, який слугує контрольним: він має залишитись у карті після завершення флуду. Далі виконано автоматичний flood: послідовне додавання 47 записів (cgroup від 100002 до 100048) через REST API. Кожен запит повертав HTTP 200 з лічильником count=N/48, що підтверджує коректну роботу Go-guard до досягнення ліміту. Останній успішний запис зафіксував count=48/48 (Рис. 4.28 – 4.29).

```

violetta@eBPF-Dev-Ubunt:~$ FIRST_CGROUP=100001
violetta@eBPF-Dev-Ubunt:~$ curl -s "http://localhost:9090/blocklist/add?cgroup=$FIRST_CGROUP"
ok cgroup=100001 added to blocklist count=1/48
violetta@eBPF-Dev-Ubunt:~$ echo $FIRST_CGROUP
100001
violetta@eBPF-Dev-Ubunt:~$ for i in $(seq 2 48); do
>   CGROUP_ID=$((100000 + i))
>   HTTP_CODE=$(curl -s -o /tmp/t_r1_resp_$i.txt -w "%{http_code}" \
>     "http://localhost:9090/blocklist/add?cgroup=$CGROUP_ID")
>   BODY=$(cat /tmp/t_r1_resp_$i.txt)
>   echo "Entry $i (cgroup=$CGROUP_ID): HTTP $HTTP_CODE | $BODY"
> done
Entry 2 (cgroup=100002): HTTP 200 | ok cgroup=100002 added to blocklist count=2/48
Entry 3 (cgroup=100003): HTTP 200 | ok cgroup=100003 added to blocklist count=3/48
Entry 4 (cgroup=100004): HTTP 200 | ok cgroup=100004 added to blocklist count=4/48
Entry 5 (cgroup=100005): HTTP 200 | ok cgroup=100005 added to blocklist count=5/48
Entry 6 (cgroup=100006): HTTP 200 | ok cgroup=100006 added to blocklist count=6/48

```

Рисунок 4.28 — Початок заповнення blocklist до ліміту 48/48

```

Entry 44 (cgroup=100044): HTTP 200 | ok cgroup=100044 added to blocklist count=44/48
Entry 45 (cgroup=100045): HTTP 200 | ok cgroup=100045 added to blocklist count=45/48
Entry 46 (cgroup=100046): HTTP 200 | ok cgroup=100046 added to blocklist count=46/48
Entry 47 (cgroup=100047): HTTP 200 | ok cgroup=100047 added to blocklist count=47/48
Entry 48 (cgroup=100048): HTTP 200 | ok cgroup=100048 added to blocklist count=48/48
violetta@eBPF-Dev-Ubunt:~$ curl -s http://localhost:9090/metrics | grep -E "oikesa_blocklist"
# HELP oikesa_blocklist_count current blocklist entries
# TYPE oikesa_blocklist_count gauge
oikesa_blocklist_count 48
# HELP oikesa_blocklist_max maximum blocklist capacity
# TYPE oikesa_blocklist_max gauge
oikesa_blocklist_max 48

```

Рисунок 4.29 — Кінець заповнення blocklist до ліміту 48/48

Спроба додати 49-й запис (cgroup=999999) повернула HTTP 500 Internal Server Error з тілом відповіді: «blocklist full (48/48): reject cgroup=999999 — use RemoveFromBlocklist first». У журналі агента відсутній рядок BLOCKLIST ADD для cgroup=999999, що підтверджує: Go-guard відхилив запит до виклику BPF-операції (Рис. 4.30).

```

violetta@eBPF-Dev-Ubunt:~$ curl -v "http://localhost:9090/blocklist/add?cgroup=999999" 2>&1
* Host localhost:9090 was resolved.
* IPv6: ::1
* IPv4: 127.0.0.1
* Trying [::1]:9090...
* Connected to localhost (::1) port 9090
> GET /blocklist/add?cgroup=999999 HTTP/1.1
> Host: localhost:9090
> User-Agent: curl/8.5.0
> Accept: */*
>
< HTTP/1.1 500 Internal Server Error
< Content-Type: text/plain; charset=utf-8
< X-Content-Type-Options: nosniff
< Date: Tue, 12 May 2026 11:07:54 GMT
< Content-Length: 79
<
blocklist full (48/48): reject cgroup=999999 - use RemoveFromBlocklist first
* Connection #0 to host localhost left intact

```

Рисунок 4.30 — Відмова Go-guard при спробі додати 49-й запис (HTTP 500)

Верифікація стану ядрової BPF-карти через bpftool map dump підтвердила: реальна кількість записів у blocklist_map становить 48 (підррахунок за ключами у JSON-виводі), тоді як фізична місткість карти — 64 записи. Різниця у 16 слотів є буфером безпеки, що виключає спрацювання LRU eviction.

Пошук sentinel-запису підтвердив його наявність: команда `grep` повернула рядок "key": 100001 — запис, доданий першим, залишився в карті після повного заповнення blocklist (Рис. 4.31 – 4.32).

```
violetta@eBPF-Dev-Ubunt:~$ MAP_ID=$(sudo bpftool map show | grep "blocklist_map" | awk '{print $1}' | tr -d ':')
violetta@eBPF-Dev-Ubunt:~$ echo "blocklist_map id = $MAP_ID"
blocklist_map id = 1992
violetta@eBPF-Dev-Ubunt:~$ REAL_COUNT=$(sudo bpftool map dump id $MAP_ID 2>/dev/null | grep -c '"key"')
violetta@eBPF-Dev-Ubunt:~$ echo $REAL_COUNT
48
violetta@eBPF-Dev-Ubunt:~$ sudo bpftool map show id $MAP_ID
1992: lru_hash name blocklist_map flags 0x0
      key 8B value 1B max_entries 64 memlock 6080B
      btf_id 1581
```

Рисунок 4.31 — Верифікація через bpftool: 48 записів у карті місткістю 64

```
violetta@eBPF-Dev-Ubunt:~$ sudo bpftool map dump id $MAP_ID 2>/dev/null | grep '"key": 100001'
      "key": 100001,
violetta@eBPF-Dev-Ubunt:~$ kubectl delete pods -l app=flood-test --ignore-not-found
No resources found
violetta@eBPF-Dev-Ubunt:~$ curl -s http://localhost:9090/metrics | grep -E "oikesa_blocklist"
# HELP oikesa_blocklist_count current blocklist entries
# TYPE oikesa_blocklist_count gauge
oikesa_blocklist_count 48
# HELP oikesa_blocklist_max maximum blocklist capacity
# TYPE oikesa_blocklist_max gauge
oikesa_blocklist_max 48
violetta@eBPF-Dev-Ubunt:~$ sudo bpftool map show | grep -A5 "blocklist"
```

Рисунок 4.32 — Верифікація через bpftool: sentinel cgroup=100001 присутній

Усі чотири критерії успішності тесту T-R1 виконані повністю. Го-лічильник blocklistCount коректно обмежив кількість записів на рівні 48, не дозволивши ядровій LRU-карті наблизитись до межі eviction. Відповідь HTTP 500 з точним повідомленням про причину відмови підтверджує детермінованість поведінки системи при переповненні. Факт збереження sentinel-запису (cgroup=100001) після флуду є ключовим доказом відсутності LRU eviction: у карті з 48 записів при місткості 64 жодне витіснення не відбулось. Механізм захисту від LRU flood attack функціонує відповідно до

архітектурної специфікації: буфер у 16 слотів виявився достатнім для повного виключення ризику витіснення легітимних блокуючих записів.

Слід зазначити, що в даній роботі флуд симулювався вручну через REST API для наочності тестування. В реальному розгортанні роль джерела записів виконує автоматизована система реагування (SOAR), яка викликає `AddToBlocklist()` при кожному виявленому alert-i. У такому сценарії буфер у 16 слотів (64 – 48) виконує критичну роль: навіть якщо blocklist заповнений до ліміту 48 і SOAR не встигає видаляти завершені контейнери, ядрова LRU-карта ніколи не витісняє легітимні блокуючі записи. При досягненні ліміту SOAR отримує HTTP 500 і повинен або видалити неактуальні записи через `/blocklist/remove`, або сповістити оператора — блокування вже відомих загроз при цьому зберігається. Автоматична ротація blocklist.

Тест T-F1: Механізм Garbage Collection застарілих записів (Pending TTL). OIKESA реалізує механізм автоматичного очищення застарілих записів у `pending_cgroups` — тимчасовому сховищі `cgroup`-ідентифікаторів, що очікують реєстрації у `whitelist_map`. Після отримання події `containerd task/start` агент додає `cgroup_id` нового контейнера до `pending_cgroups` разом із `timestamp` створення. Під час першого системного виклику з боку процесу контейнера eBPF-програма перевіряє наявність відповідного запису у `whitelist_map` та час його перебування у `pending_cgroups`. Якщо з моменту додавання запису минуло більше `PENDING_TTL` (5 секунд), запис вважається застарілим: він автоматично видаляється з `pending_cgroups`, генерується подія `PEND_EXP`, а лічильник `oikesa_pending_expired_total` збільшується на одиницю. Цей механізм запобігає накопиченню stale-записів у випадках, коли `whitelist`-реєстрація не відбулась своєчасно через затримку синхронізації з Kubernetes API.

					ІАЛЦ.467200.003 ПЗ	Арк.
						111
Зм.	Арк.	№ докум.	Підпис	Дата		

Перед початком тесту виконано повне очищення середовища: видалено залишкові Pod'и та правила iptables. Знято базовий стан метрики: `oikesa_pending_expired_total 0`, що видно на Рис. 4.33.

```
violetta@eBPF-Dev-Ubunt:~$ sudo iptables -D OUTPUT -p tcp --dport 6443 -j DROP 2>/dev/null || true
violetta@eBPF-Dev-Ubunt:~$ BASE=$(curl -s localhost:9090/metrics \
> | grep "^oikesa_pending_expired_total" \
> | awk '{print $2}')
violetta@eBPF-Dev-Ubunt:~$ echo $BASE
0
violetta@eBPF-Dev-Ubunt:~$ curl -s localhost:9090/metrics | grep pending_expired
# HELP oikesa_pending_expired_total stale pending TTL expired
# TYPE oikesa_pending_expired_total counter
oikesa_pending_expired_total 0
```

Рисунок 4.33 — Базовий стан лічильника перед початком тесту

Для гарантованого спрацювання механізму Pending TTL використано тестовий Pod зі спеціально сформованим командним рядком (Рис. 4.34). Команда `sleep 7` штучно затримує виконання першого значущого системного виклику контейнера на 7 секунд, що перевищує значення `PENDING_TTL` (5 секунд) та забезпечує детерміноване спрацювання механізму очищення pending-запису. Після виклику `kubectl apply` та появи у журналі агента події `containerd task/start` — яка підтверджує, що `cgroup_id` вже додано до `pending_cgroups` — доступ до Kubernetes API Server заблоковано правилом iptables на порту 6443. Блокування унеможлиблює whitelist-реєстрацію через informer-синхронізацію. Після очікування 10 секунд (що вдвічі перевищує `PENDING_TTL`) виконано перевірку метрик (Рис. 4.35).

```
GNU nano 7.2 /tmp/ttl-test.yaml
apiVersion: v1
kind: Pod
metadata:
  name: ttl-test
spec:
  restartPolicy: Never
  containers:
  - name: ttl-test
    image: busybox
    command:
    - sh
    - -c
    - |
      sleep 7
      echo trigger
      sleep 9999
```

Рисунок 4.34 — Конфігурація тестового Pod ttl-test із затримкою виконання sleep 7 (YAML)

```
violetta@eBPF-Dev-Ubunt:~$ kubectl apply -f /tmp/ttl-test.yaml
pod/ttl-test created
violetta@eBPF-Dev-Ubunt:~$ sudo iptables -I OUTPUT -p tcp --dport 6443 -j DROP
violetta@eBPF-Dev-Ubunt:~$ sleep 10
violetta@eBPF-Dev-Ubunt:~$ curl -s localhost:9090/metrics | grep pending_expired
# HELP oikesa_pending_expired_total stale pending TTL expired
# TYPE oikesa_pending_expired_total counter
oikesa_pending_expired_total 1
```

Рисунок 4.35 — Метрика oikesa_pending_expired_total після спрацювання механізму Garbage Collection (значення зросло з 0 до 1)

Лічильник oikesa_pending_expired_total зріс з 0 до 1 — порівняйте Рис. 4.33 та Рис. 4.35, — що підтверджує спрацювання механізму очищення. У журналі агента зафіксовано подію:

```
ancestry:0 conn_blk:0 | bind_blk:0 pend_exp:0 | eff:0.0% pods:11 bl_count:0/48
18:20:50 containerd.go:115: [+] containerd task/start containerID=ff2bebe2caf9 cgroupID=128663 pid=3480000
18:20:50 veth_watcher.go:272: [DBG] VETH PROBE iface=eth0 ifindex=2 iflink=381
18:20:50 veth_watcher.go:64: [INFO] VETH SKIP XDP iface=veth4bd8f883 is bridge port, XDP already on master
18:20:50 veth_watcher.go:85: [+] VETH TC INGRESS iface=veth4bd8f883 containerID=ff2bebe2caf9
18:20:50 veth_watcher.go:99: [+] VETH TC EGRESS iface=veth4bd8f883 containerID=ff2bebe2caf9
18:20:53 containerd.go:115: [+] containerd task/start containerID=f0a29b0ae464 cgroupID=128742 pid=3480027
18:20:53 veth_watcher.go:272: [DBG] VETH PROBE iface=eth0 ifindex=2 iflink=381
18:20:53 veth_watcher.go:64: [INFO] VETH SKIP XDP iface=veth4bd8f883 is bridge port, XDP already on master
18:20:53 alerts.go:95: [ALERT] type=PEND EXP PID:3480027 cgroup:128742 comm:sh pod:<unknown>
18:20:53 veth_watcher.go:82: [WARN] VETH TCX ingress veth4bd8f883: attach tcx link: file exists
```

Рисунок 4.36 — Фрагмент журналу OIKESA із зафіксованою подією PEND_EXP для простроченого запису pending_cgroups

Статус pod «unknown» є очікуваним: на момент першого syscall whitelist-реєстрація ще не відбулась через недоступність API, тому агент не міг встановити відповідність між cgroup_id та Pod-об'єктом Kubernetes (Рис. 4.36).

Після зняття правила iptables informer-синхронізація відновилась, і Pod коректно зареєструвався у whitelist_map, що видно на Рис. 4.37:

Рисунок 4.37 — Частина логу агента: коректна реєстрація Pod у whitelist_map після відновлення з'єднання з kube-apiserver

Це підтверджує, що механізм GC не порушує подальшу роботу агента і не призводить до постійного блокування легітимних Pod'ів. Механізм Garbage Collection коректно ідентифікує та видаляє застарілі записи з pending_cgroups, фіксує відповідні події у журналі та метриках, а після відновлення доступу до Kubernetes API забезпечує коректну реєстрацію Pod у whitelist_map без втрати стану.

Тест T-R2: Збереження стану при відключенні Kubernetes API. OIKESA зберігає поточний стан whitelist_map у пам'яті ядра між циклами синхронізації, тому вже зареєстровані Pod'и продовжують отримувати suppression і enforcement навіть без зв'язку з kube-apiserver. Тест підтверджує цю властивість шляхом блокування трафіку до API: жодні захисні функції для вже відомих Pod'ів не деградують. Persist-стан blocklist_map зберігається у ConfigMap і відновлюється після перезапуску агента незалежно від доступності API в момент старту.

Перед початком тесту виконано повне очищення середовища та запущено агент з нуля. Після ініціалізації агент зареєстрував 11 cgroup-ідентифікаторів для 5 Pod'ів у whitelist_map та перейшов у стан "OIKESA fully active". Знято

базовий snapshot мапи (11 записів) та додано тестовий blocklist-запис через REST API. Факт збереження blocklist підтверджено через ConfigMap oikesa-blocklist-state (поле count=1). Базовий стан агента наведено на Рис. 4.38

```
Violetta@e8PF-Dev-Ubunt:~$ kubectl get pods -A
NAMESPACE      NAME                                READY   STATUS    RESTARTS   AGE
kube-system     coredns-76c974cb66-fhfzt           1/1     Running   0           140m
kube-system     local-path-provisioner-8686667995-4nlmx 1/1     Running   0           140m
kube-system     metrics-server-c8774f4-rgs22        1/1     Running   0           140m
kube-system     svclb-traefik-afe33c49-cg9nv        2/2     Running   24 (8d ago) 33d
kube-system     traefik-c5c8bf4ff-thgnb            1/1     Running   0           140m
Violetta@e8PF-Dev-Ubunt:~$ curl -s http://localhost:9090/metrics | grep -E "oikesa_whitelist|oikesa_openat_suppressed"
# HELP oikesa_openat_suppressed_total openat suppressed
# TYPE oikesa_openat_suppressed_total counter
oikesa_openat_suppressed_total 1217
# HELP oikesa_whitelist_pods whitelisted pod cgroups
# TYPE oikesa_whitelist_pods gauge
oikesa_whitelist_pods 11
Violetta@e8PF-Dev-Ubunt:~$ curl -s http://localhost:9090/healthz
ok
Violetta@e8PF-Dev-Ubunt:~$ curl -s http://localhost:9090/readyz
ready_pods=11
Violetta@e8PF-Dev-Ubunt:~$ WL_MAP_ID=$(sudo bpftool map show | grep "whitelist_map" | awk '{print $1}' | tr -d ':')
Violetta@e8PF-Dev-Ubunt:~$ echo "whitelist_map id = $WL_MAP_ID"
whitelist_map id = 2098
Violetta@e8PF-Dev-Ubunt:~$ WL_COUNT=$(sudo bpftool map dump id $WL_MAP_ID 2>/dev/null | grep -c '"key"')
Violetta@e8PF-Dev-Ubunt:~$ echo "whitelist_map entries: $WL_COUNT"
whitelist_map entries: 11
Violetta@e8PF-Dev-Ubunt:~$ sudo bpftool map dump id $WL_MAP_ID 2>/dev/null | grep '"key"' > /tmp/whitelist_before.txt
Violetta@e8PF-Dev-Ubunt:~$ echo "Snapshot saved: $(wc -l < /tmp/whitelist_before.txt) entries"
Snapshot saved: 11 entries
Violetta@e8PF-Dev-Ubunt:~$ curl -s "http://localhost:9090/blocklist/add?cgroup=200001"
ok cgroup=200001 added to blocklist count=1/48
```

Рисунок 4.38 — Базовий стан агента: whitelist_map з 11 записами, blocklist count=1/48 та підтвердження ConfigMap

Для імітації відключення control plane трафік на порт 6443 заблоковано правилами iptables. Порт 6443 є стандартним портом kube-apiserver у кластерах Kubernetes та K3s — саме через нього агент OIKESA звертається до Kubernetes API для синхронізації Pod lifecycle (Informers), збереження та відновлення ConfigMap з persist-станом blocklist, а також для реєстрації нових cgroup-ідентифікаторів у whitelist_map. Блокування цього порту повністю ізолює агента від control plane без зупинки самого API-сервера, що дозволяє точно імітувати мережеву недоступність у реальних умовах. Недоступність API підтверджено командою kubectl get pods --request-timeout=3s, яка повернула помилку з'єднання. Через 30 секунд після блокування виконано верифікацію стану агента. Результати верифікації наведено у Таблиці 4.8 та на Рис. 4.39.

Таблиця 4.8 — Стан агента під час 30-секундного blackout kube-apiserver

Метрика	До блокування	Під час blackout	Зміна
oikesa_whitelist_pods	11	11	Відсутня
oikesa_openat_suppressed_total	1 217	19 381	+18 164
/healthz	ok	ok	Відсутня
/readyz	ready pods=11	ready pods=11	Відсутня
whitelist_map entries	11	11	Відсутня

Лічильник `oikesa_openat_suppressed_total` зріс на 18 164 події протягом 30 секунд blackout, що підтверджує безперервну роботу eBPF-захисту на рівні ядра без будь-якої взаємодії з Kubernetes API. Endpoint `/readyz` незмінно повертав `ready pods=11`.

```
violetta@eBPF-Dev-Ubunt:~$ kubectl get configmap oikesa-blocklist-state -n kube-system -o jsonpath='{.data.count}'
violetta@eBPF-Dev-Ubunt:~$ sudo iptables -I OUTPUT -d 127.0.0.1 -p tcp --dport 6443 -j DROP
violetta@eBPF-Dev-Ubunt:~$ sudo iptables -I INPUT -s 127.0.0.1 -p tcp --sport 6443 -j DROP
violetta@eBPF-Dev-Ubunt:~$ echo "API blocked at: $(date '+%H:%M:%S')"
```

```
API blocked at: 18:21:53
violetta@eBPF-Dev-Ubunt:~$ kubectl get pods --request-timeout=3s 2>&1 || echo "API unreachable -- expected"
Unable to connect to the server: net/http: request canceled while waiting for connection (Client.Timeout exceeded while awaiting headers)
API unreachable -- expected
violetta@eBPF-Dev-Ubunt:~$ sleep 30
violetta@eBPF-Dev-Ubunt:~$ curl -s http://localhost:9090/metrics | grep -E "oikesa_whitelist|oikesa_openat_suppressed"
```

```
# HELP oikesa_openat_suppressed_total openat suppressed
# TYPE oikesa_openat_suppressed_total counter
oikesa_openat_suppressed_total 19381
# HELP oikesa_whitelist_pods whitelisted pod cgroups
# TYPE oikesa_whitelist_pods gauge
oikesa_whitelist_pods 11
violetta@eBPF-Dev-Ubunt:~$ curl -s http://localhost:9090/healthz
ok
violetta@eBPF-Dev-Ubunt:~$ curl -s http://localhost:9090/readyz
ready pods=11
violetta@eBPF-Dev-Ubunt:~$ WL_COUNT_DURING=$(sudo bpftool map dump id $WL_MAP_ID 2>/dev/null | grep -c '"key"')
violetta@eBPF-Dev-Ubunt:~$ echo "whitelist_map entries during blackout: $WL_COUNT_DURING"
whitelist_map entries during blackout: 11
```

Рисунок 4.39 — Стан агента під час blackout: whitelist стабільний (11), suppressed зростає (19 381), healthz/readyz без деградації

Далі агент зупинено командою `kill oikesa-agent` при заблокованому API та запущено нову інстанцію. Цей крок перевіряє поведінку функції `restoreBlocklistFromConfigMap` при недоступному kube-apiserver. Реалізовано `retry`-петлю: при мережевих помилках агент очікує 5 секунд і повторює спробу, тоді як HTTP 404 коректно інтерпретується як штатний перший запуск. Як видно з Рис. 4.31, агент успішно увійшов у `retry`-петлю та виконав 4 послідовні

спроби з інтервалом 5 секунд, не стартуючи зі спустошеним blocklist. Після зняття правил iptables агент самостійно завершив retry-петлю, зчитав ConfigMap та відновив blocklist без перезапуску. Результат відновлення наведено на Рис. 4.40. Повний час від рестарту до відновленого стану склав приблизно 40 секунд, з яких 35 секунд — очікування у retry-петлі.

```
18:25:00 containerd.go:78: [*] containerd gRPC connected - subscribing task/start + task/exit
18:25:05 blocklist_persist.go:165: [*] BLOCKLIST RESTORE waiting for API (Get "https://127.0.0.1:6443/api/v1/namespaces/kube-system/configmaps/oikesa-blocklist-state":
context deadline exceeded) - retrying in 5s...
18:25:15 blocklist_persist.go:165: [*] BLOCKLIST RESTORE waiting for API (Get "https://127.0.0.1:6443/api/v1/namespaces/kube-system/configmaps/oikesa-blocklist-state":
context deadline exceeded) - retrying in 5s...
18:25:25 blocklist_persist.go:165: [*] BLOCKLIST RESTORE waiting for API (Get "https://127.0.0.1:6443/api/v1/namespaces/kube-system/configmaps/oikesa-blocklist-state":
context deadline exceeded) - retrying in 5s...
18:25:35 blocklist_persist.go:165: [*] BLOCKLIST RESTORE waiting for API (Get "https://127.0.0.1:6443/api/v1/namespaces/kube-system/configmaps/oikesa-blocklist-state":
context deadline exceeded) - retrying in 5s...
18:25:40 blocklist_persist.go:203: [*] BLOCKLIST RESTORED 1/1 entries from ConfigMap kube-system/oikesa-blocklist-state
18:25:40 informers.go:137: [*] Waiting for cache sync (pods + namespaces)...
```

Рисунок 4.40 — Retry-петля агента: 4 спроби з інтервалом 5 секунд при недоступному kube-apiserver та результат відновлення

Фінальна верифікація метрик через /metrics та bpftool map dump підтвердила повне відновлення стану (Рис. 4.41). Після рестарту eBPF-мапи отримують нові ідентифікатори ядра: було 2098 (Рис. 4.38), а потім стало 2110 (Рис. 4.42) — це штатна поведінка, при якій кожен запуск агента завантажує нові екземпляри мап. Перевірка з актуальним ID підтвердила 11 записів у whitelist_map, oikesa_blocklist_count 1.

```
violetta@eBPF-Dev-Ubunt:~$ kubectl get pods -A
NAMESPACE      NAME                                     READY   STATUS    RESTARTS   AGE
kube-system     coredns-76c974cb66-fhftz               1/1     Running   0           146m
kube-system     local-path-provisioner-8686667995-4nlmx 1/1     Running   0           146m
kube-system     metrics-server-c8774f4f4-ngs22          1/1     Running   0           146m
kube-system     svclb-traefik-afe33c49-cg9nv            2/2     Running   0 (8d ago)  33d
kube-system     traefik-c5c8bf4ff-thgnb                 1/1     Running   0           146m
violetta@eBPF-Dev-Ubunt:~$ sleep 10
violetta@eBPF-Dev-Ubunt:~$ curl -s http://localhost:9090/metrics | grep -E "oikesa_whitelist|oikesa_openat_suppressed|oikesa_blocklist"
# HELP oikesa_openat_suppressed_total openat suppressed
# TYPE oikesa_openat_suppressed_total counter
oikesa_openat_suppressed_total 8963
# HELP oikesa_whitelist_pods whitelisted pod cgroups
# TYPE oikesa_whitelist_pods gauge
oikesa_whitelist_pods 11
# HELP oikesa_blocklist_count current blocklist entries
# TYPE oikesa_blocklist_count gauge
oikesa_blocklist_count 1
# HELP oikesa_blocklist_max maximum blocklist capacity
# TYPE oikesa_blocklist_max gauge
oikesa_blocklist_max 48
violetta@eBPF-Dev-Ubunt:~$ WL_COUNT_AFTER=$(sudo bpftool map dump id $WL_MAP_ID 2>/dev/null | grep -c '"key"')
violetta@eBPF-Dev-Ubunt:~$ echo "whitelist_map entries after restore: $WL_COUNT_AFTER"
whitelist_map entries after restore: 0
```

Рисунок 4.41 — Фінальні метрики: blocklist_count=1, whitelist_pods=11, whitelist_map entries=11 (новий ID 2110)

Слід зазначити, що команда `bpftool map dump id $WL_MAP_ID` після рестарту агента повернула 0 записів, що на перший погляд виглядає як розбіжність. Причина є суто технічною: змінна `WL_MAP_ID` зберігала ідентифікатор 2098, який належав eBPF-мапі попередньої інстанції агента. Після рестарту Go-процес завантажує у ядро нові екземпляри всіх eBPF-мап з новими ідентифікаторами — стара мапа з ID 2098 більше не існує. Перевірка з актуальним ідентифікатором 2110, отриманим через `bpftool map show | grep whitelist_map`, підтвердила 11 записів у `whitelist_map` нової інстанції агента, що повністю відповідає очікуваному стану. Таким чином, результат `whitelist_map entries after restore: 0` є артефактом тестового скрипту (використання застарілої змінної `WL_MAP_ID`), а не індикатором некоректної роботи агента.

```
violetta@eBPF-Dev-Ubunt:~$ WL_MAP_ID_NEW=$(sudo bpftool map show | grep "whitelist_map" | awk '{print $1}' | tr -d ':')
violetta@eBPF-Dev-Ubunt:~$ echo "New whitelist_map id = $WL_MAP_ID_NEW"
New whitelist_map id = 2110
violetta@eBPF-Dev-Ubunt:~$ sudo bpftool map dump id $WL_MAP_ID_NEW 2>/dev/null | grep -c '"key"'
11
```

Рисунок 4.42 — Новий ідентифікатор ядра

Усі критерії тесту T-R2 виконані. `Whitelist_map` залишався стабільним протягом усього `blackout`, `suppression` зростав безперервно, `healthz/readyz` не деградували. `Retry`-петля коректно відклала старт `blocklist`-відновлення до моменту доступності API, що виключає ризик втрати `persist`-стану при короткочасній недоступності `control plane`. `Persist`-механізм через `ConfigMap` та `retry`-логіка функції `restoreBlocklistFromConfigMap` функціонують відповідно до архітектурної специфікації.

Таблиця 4.9 — Зведені тести OIKESA

Тест	Ціль	Результат	Критерій «Провал»
T-C1 (BM)	CPU overhead	OIKESA: 0.05% (idle) / 0.05% (1000 exec/c) проти Falco 61.28% / Tetragon 0.76%	Якщо більше за 5%, то FAIL

Продовження таблиці 4.9

T-C1 (Jetson Nano)	CPU overhead	OIKESA: 0.02% (idle) / 0.07% (1000 exec/c) проти Falco 97.60% / Tetragon 101.13%	Якщо більше за 5%, то FAIL
T-C2 (BM)	RAM footprint	OIKESA: 59 МБ RSS + 2.4 МБ eBPF = 61 МБ проти Falco 126 МБ / Tetragon 422 МБ	Якщо більше за 70 МБ, то FAIL
T-C2 (Jetson Nano)	RAM footprint	OIKESA: 60 МБ RSS + 0.09 МБ eBPF = 60.1 МБ проти Falco 104.2 МБ / Tetragon 497 МБ	
T-C3	False Positive Rate	oikesa_blocked_total = 0 протягом усіх 4 операцій (rollout restart, масштабування 0 до 5, НРА з load-gen, rescheduling)	Якщо більше за 0, то FAIL
T-S1	LSM blocking	-EPERM при execve (/bin/ls); blocklist відновлено після рестарту: BLOCKLIST RESTORED 5/5 entries	Відсутнє блокування, то FAIL
T-R1	LRU flood захист	Go-guard обмежив записи до 48/48; HTTP 500 на 49-й запит; sentinel cgroup=100001 збережено; у карті 48 із 64 слотів зайнято	Якщо більше за 48, то FAIL
T-F1	Pending TTL GC	oikesa_pending_expired_total зріс з 0 до 1 після 7с (більше PENDING_TTL 5с); подія PEND_EXP у журналі; після відновлення API Pod зареєстровано коректно	Якщо stale-записи залишаються, то FAIL
T-R2	API disconnect	whitelist_map стабільний (11 записів); suppressed збільшився на 18 164 за 30с blackout; healthz/readyz без деградації; blocklist відновлено після рестарту за 40с через retry-петлю (4 спроби)	Якщо функції деградують, то WARN

ВИСНОВОК ДО РОЗДІЛУ

У даному розділі розроблено комплексну методологію тестування та порівняльного аналізу системи OIKESA за чотирма категоріями: порівняльні (T-C), безпекові (T-S), надійності (T-R) та функціональні (T-F). Зведені критерії успішності всіх тестів наведено в Таблиці 4.9. Визначено два тестових середовища відповідно до типових сценаріїв розгортання (Таблиця 4.2, Таблиця 4.3): віртуальна машина x86_64 як базовий стенд для верифікації функціональності та порівняльних бенчмарків, та одноплатний комп'ютер NVIDIA Jetson Nano як цільова IoT-платформа з жорсткими ресурсними обмеженнями (4 ГБ LPDDR4, 4 ядра Cortex-A57 @ 1.43 ГГц). Уточнено архітектурне обмеження: функція TC-egress (AttachTCX) недоступна на ядрі Linux 5.10 NVIDIA Jetson Nano OS і потребує або оновлення ядра до версії 6.6+, або заміни на legacy netlink-TC; решта підсистем (fentry/fexit, BPF-LSM, XDP, ring buffer) повністю сумісні з наявним ядром.

Порівняльні тести T-C1 та T-C2 кількісно підтвердили ключові переваги підходу «drop-at-source» на обох тестових платформах. Тест T-C1 на віртуальній машині показав, що OIKESA споживає лише 0.05% CPU як при низькому (10 Pod'ів, idle), так і при середньому навантаженні (50 Pod'ів, 1000 execve/c) — що у 1225 разів менше за Falco (61.28%) та у 15 разів менше за Tetragon (0.76%) при пікових умовах (Таблиця 4.4). На платформі NVIDIA Jetson Nano різниця між рішеннями проявилась ще різкіше через обмежену потужність процесора Cortex-A57: при навантаженні 20 Pod'ів Falco досягає 97.60% CPU, Tetragon перевищує позначку 101.13%, використовуючи більше одного логічного ядра, тоді як OIKESA зберігає споживання на рівні 0.07% (Таблиця 4.5, Рисунки 4.1–4.3). Результат обумовлений тим, що 100% execve-подій від Pod'ів зі whitelist_map відсікаються безпосередньо в ядрі без передачі

					ІАЛЦ.467200.003 ПЗ	Арк.
						120
Зм.	Арк.	№ докум.	Підпис	Дата		

в user-space. Тест T-C2 підтвердив, що сумарне споживання пам'яті OIKESA (59 МБ RSS + 2.4 МБ eBPF-мапи = 61 МБ на VM; 60 МБ RSS + 0.09 МБ eBPF-мапи = 60.1 МБ на Jetson Nano) укладається у встановлену ціль 70 МБ на обох платформах, тоді як Falco споживає 126 МБ та 104.2 МБ відповідно, а Tetragon — 422 МБ та 497 МБ відповідно (Таблиця 4.6, Таблиця 4.7, Рисунки 4.4–4.16). Особливо критичним є результат на Jetson Nano: при загальному обсязі оперативної пам'яті 4 ГБ LPDDR4 споживання Tetragon у 497 МБ фактично унеможливорює його розгортання в реальних IoT-кластерах поряд з корисним навантаженням.

Тест T-C3 (False Positive Rate) емпірично підтвердив нульовий рівень хибних спрацювань при чотирьох штатних операціях Kubernetes: rolling restart, масштабування від 0 до 5 реплік, робота з HPA та генератором навантаження, примусове rescheduling Pod'а. У всіх випадках лічильник oikesa_blocked_total залишався рівним 0 (Рисунки 4.23–4.26). Це досягається завдяки orchestrator-aware архітектурі: cgroup_id нових Pod'ів реєструється у whitelist_map у момент переходу до стану Running — до фактичного виконання будь-яких процесів у контейнері.

Функціональне тестування безпеки (T-S1) підтвердило коректність механізмів BPF-LSM примусового виконання: хук lsm/bprm_check_security повертає -EPERM при спробі запуску забороненого виконуваного файлу у заблокованому контейнері (Рисунок 4.18). Додатково перевірено механізм відновлення blocklist_map після перезапуску агента через Kubernetes ConfigMap — агент коректно відновив 5 із 5 записів (Рисунки 4.21–4.22).

Тести надійності підтвердили стійкість системи до атак та нештатних ситуацій. Тест T-R1 підтвердив захист від LRU flood attack: Go-guard коректно обмежив кількість записів у blocklist_map на рівні 48 при фізичній місткості карти 64 записи, відхиливши 49-й запит з HTTP 500, а sentinel-запис

cgroup=100001 зберігся після повного заповнення blocklist (Рисунки 4.31–4.32). Тест T-F1 підтвердив коректну роботу механізму Garbage Collection застарілих записів pending_cgroups: лічильник oikesa_pending_expired_total зріс з 0 до 1 після спрацювання PENDING_TTL (5 секунд) з подією PEND_EXP у журналі агента (Рисунки 4.35–4.36), а після відновлення доступу до Kubernetes API Pod коректно зареєструвався у whitelist_map без втрати стану (Рисунок 4.37). Тест T-R2 підтвердив збереження всіх захисних функцій при 30-секундному blackout kube-apiserver: whitelist_map залишався стабільним (11 записів), лічильник oikesa_openat_suppressed_total зріс на 18 164 події, а endpoint /readyz незмінно повертав ready pods=11 (Таблиця 4.8, Рисунок 4.39). Retry-петля агента коректно відклала відновлення blocklist до моменту доступності API, виконавши 4 спроби з інтервалом 5 секунд (Рисунок 4.40).

Сукупність проведених тестів забезпечує повне покриття функціональних та нефункціональних вимог до системи OIKESA. Всі дев'ять тестів (T-C1, T-C2, T-C3, T-S1, T-R1, T-F1, T-R2 на обох платформах) завершилися успішно відповідно до критеріїв, визначених у Таблиці 4.9. Результати на платформі NVIDIA Jetson Nano мають особливе значення: вони підтверджують, що система OIKESA є єдиним із трьох розглянутих рішень, яке здатне функціонувати в межах встановлених ресурсних обмежень на ARM64-платформі з 4 ГБ оперативної пам'яті, що безпосередньо підтверджує архітектурну обґрунтованість підходу «drop-at-source» та придатність системи для розгортання в ресурсно-обмежених IoT-середовищах на базі Kubernetes.

ВИСНОВКИ

У процесі виконання бакалаврської дипломної роботи розроблено повнофункціональний прототип підсистеми OIKESA — оркестратор-свідомого агента внутрішньоядерного придушення подій для кластерів Kubernetes у середовищах Інтернету речей та периферійних обчислень. В основі розробленого рішення лежить інноваційне застосування технології eBPF, що дозволило перенести логіку фільтрації системних викликів безпосередньо у простір ядра Linux. Такий підхід успішно вирішив актуальну інженерну проблему надмірного моніторингового шуму, спричиненого передачею легітимних системних подій у простір користувача, що є критичним фактором деградації продуктивності на апаратно обмежених ARM64-вузлах.

Під час реалізації роботи було спроектовано та впроваджено дворівневу архітектуру взаємодії, яка усуває семантичний розрив між рівнем оркестратора та рівнем операційної системи. Ядрова складова системи використовує оптимізовані eBPF-програми, які здійснюють пошук у хеш-таблицях за ідентифікатором контрольної групи із незалежною від обсягу даних складністю — за константний час. Завдяки концепції «придушення на джерелі» (drop-at-source), абсолютно всі події виконання від довірених контейнерів відсікаються на найнижчому рівні без копіювання до простору користувача, що повністю унеможливлює ефект переповнення кільцевого буфера (backpressure) при інтенсивному навантаженні. Для забезпечення сумісності з гетерогенними платформами без необхідності перекомпіляції реалізовано гібридну схему зондів, яка використовує механізм fentry як основний для нових ядер Linux, а raw tracepoint — як резервний.

Особливе значення у контексті ресурсної ефективності має розроблений механізм управління станом. Інтеграція агента з Kubernetes Informers та

					ІАЛЦ.467200.003 ПЗ	Арк.
						123
Зм.	Арк.	№ докум.	Підпис	Дата		

containerd забезпечила динамічну реєстрацію ідентифікаторів у картах дозволених подів та їх автоматичне очищення при завершенні життєвого циклу контейнерів (Lifecycle-Coupled GC). Таке архітектурне рішення дозволило досягти нульового рівня хибних спрацювань під час виконання штатних операцій Kubernetes, таких як масштабування, перезапуск або автоматичне відновлення подів. Додатково реалізовано механізм автоматичного очищення застарілих записів pending_cgroups через Pending TTL Garbage Collection, що запобігає накопиченню stale-записів у випадках затримки синхронізації з Kubernetes API. Разом із тим, система забезпечує не лише спостереження, але й превентивний контроль: впровадження BPF-LSM хуків дозволило синхронно блокувати підозрілі дії безпосередньо в ядрі, повертаючи системі стандартні коди помилок POSIX без часових затримок на перемикання контексту. Для захисту від атак типу LRU flood реалізовано дворівневий механізм: Go-лічильник обмежує кількість активних записів у blocklist_map на рівні 48 при фізичній місткості карти 64 записи, що виключає ризик витіснення легітимних блокуючих записів через механізм LRU eviction.

Технічна реалізація підтверджує високу доцільність обраного підходу. Результати порівняльного тестування на віртуальній машині x86_64 засвідчили, що при пікових навантаженнях до 1000 системних викликів на секунду агент OIKESA споживає лише 0.05% ресурсів центрального процесора, що у 1225 разів менше за показник інструменту Falco (61.28%) та у 15 разів менше за Tetragon (0.76%). Сумарне споживання оперативної пам'яті на віртуальній машині склало 61 МБ, що повністю відповідає встановленому критерію 70 МБ та є суттєво меншим за показники конкурентних рішень (Falco — 126 МБ, Tetragon — 422 МБ). Особливо показовими є результати на цільовій IoT-платформі NVIDIA Jetson Nano (ARM64, Cortex-A57): при навантаженні 20 Pod'ів Falco споживає 97.60% CPU, а Tetragon перевищує позначку 101.13%,

					ІАЛЦ.467200.003 ПЗ	Арк.
						124
Зм.	Арк.	№ докум.	Підпис	Дата		

використовуючи більше одного логічного ядра, тоді як OIKESA зберігає споживання на рівні 0.07%. Сумарне споживання пам'яті OIKESA на Jetson Nano склало 60.1 МБ, тоді як Tetragon потребує 497 МБ — що фактично унеможливило його розгортання на пристроях з 4 ГБ оперативної пам'яті поряд з корисним навантаженням. Додаткові випробування надійності довели здатність системи безперервно функціонувати навіть в умовах відключення площини керування Kubernetes: під час 30-секундного збою зв'язку з API агент успішно придушив понад 18 тисяч подій без деградації жодної захисної функції, а після відновлення з'єднання коректно відновив збережений стан blocklist через механізм retry-петлі.

Таким чином, можна стверджувати, що всі цілі та завдання дипломного проєкту повністю досягнуто. Тема дослідження розкрита у повному обсязі, а розроблене програмне забезпечення OIKESA є практично значущим, стабільним і високоефективним рішенням. Отримані результати формують міцне підґрунтя для подальшого впровадження eBPF-орієнтованих систем безпеки в інфраструктури периферійних обчислень.

					ІАЛЦ.467200.003 ПЗ	Арк.
						125
Зм.	Арк.	№ докум.	Підпис	Дата		

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

- [1] «Current IoT Forecast Highlights - Transforma Insights». Дата звернення: 2026. 15 трав. [URL]. Доступний у: <https://transformainsights.com/research/forecast/highlights>
- [2] M. Satyanarayanan, «Edge Computing», *Computer*, т. 50, вип. 10, С. 36—38, 2017, doi: 10.1109/MC.2017.3641639.
- [3] W. Shi, J. Cao, Q. Zhang, Y. Li, i L. Xu, «Edge Computing: Vision and Challenges», *IEEE Internet Things J.*, т. 3, вип. 5, С. 637—646, жовт. 2016, doi: 10.1109/JIOT.2016.2579198.
- [4] «Data growth worldwide 2029», Statista. Дата звернення: 2026. 20 трав. [URL]. Доступний у: <https://www.statista.com/statistics/871513/worldwide-data-created/>
- [5] «Overview», Kubernetes. Дата звернення: 2026. 14 трав. [URL]. Доступний у: <https://kubernetes.io/docs/concepts/overview/>
- [6] S. Rac i M. Brorsson, «Cost-Effective Scheduling for Kubernetes in the Edge-to-Cloud Continuum», у *2023 IEEE International Conference on Cloud Engineering (IC2E)*, Boston, MA, USA: IEEE, верес. 2023, С. 153—160. doi: 10.1109/IC2E59103.2023.00025.
- [7] «K3s - Lightweight Kubernetes | K3s». Дата звернення: 2026. 14 трав. [URL]. Доступний у: <https://docs.k3s.io/>
- [8] «NVIDIA Jetson Nano», NVIDIA. Дата звернення: 2026. 14 трав. [URL]. Доступний у: <https://www.nvidia.com/en-us/autonomous-machines/embedded-systems/jetson-nano/product-development/>
- [9] «Architecture | K3s». Дата звернення: 2026. 14 трав. [URL]. Доступний у: <https://docs.k3s.io/architecture>
- [10] «Cluster Architecture», Kubernetes. Дата звернення: 2026. 14 трав. [URL]. Доступний у: <https://kubernetes.io/docs/concepts/architecture/>
- [11] D. A. Patterson, J. L. Hennessy, i P. Alexander, *Computer organization and design: the hardware/software interface*, ARM® edition. Amsterdam ; Boston: Elsevier/Morgan Kaufmann, 2017.
- [12] B. Gregg, *BPF performance tools: Linux system and application observability*. у Addison-Wesley professional computing series. Boston, Mass.: Addison-Wesley, 2020.
- [13] «BPF Documentation — The Linux Kernel documentation». Дата звернення: 2026. 15 трав. [URL]. Доступний у: <https://docs.kernel.org/bpf/>
- [14] torvalds, «linux/arch/arm64/net/bpf_jit_comp.c at master · torvalds/linux», GitHub. Дата звернення: 2026. 15 трав. [URL]. Доступний у: https://github.com/torvalds/linux/blob/master/arch/arm64/net/bpf_jit_comp.c

- [15] «Requirements | K3s». Дата звернення: 2026. 15 трав. [URL]. Доступний у: <https://docs.k3s.io/installation/requirements>
- [16] «KPTI/KAISER Meltdown Initial Performance Regressions». Дата звернення: 2026. 15 трав. [URL]. Доступний у: <https://www.brendangregg.com/blog/2018-02-09/kpti-kaiser-meltdown-performance.html>
- [17] «Tracee». Дата звернення: 2026. 15 трав. [URL]. Доступний у: <https://aquasecurity.github.io/tracee/latest/>
- [18] J. Her, J. Kim, J. Kim, i S. Lee, «An In-Depth Analysis of eBPF-Based System Security Tools in Cloud-Native Environments», *IEEE Access*, т. 13, С. 155588—155604, 2025, doi: 10.1109/ACCESS.2025.3605432.
- [19] M. Craun, K. Hussain, U. Gautam, Z. Ji, T. Rao, i D. Williams, «Eliminating eBPF Tracing Overhead on Untraced Processes», у *Proceedings of the SIGCOMM Workshop on eBPF and Kernel Extensions*, Sydney NSW Australia: ACM, серп. 2024, С. 16—22. doi: 10.1145/3672197.3673431.
- [20] L. Kleinrock, *Queueing systems*. New York: Wiley, 1975.
- [21] X. Zhu та ін., «Dissecting Overheads of Service Mesh Sidecars», у *Proceedings of the 2023 ACM Symposium on Cloud Computing*, Santa Cruz CA USA: ACM, жовт. 2023, С. 142—157. doi: 10.1145/3620678.3624652.
- [22] «What is eBPF? An Introduction and Deep Dive into the eBPF Technology». Дата звернення: 2026. 15 трав. [URL]. Доступний у: <https://ebpf.io/what-is-ebpf/>
- [23] «Linux eBPF Tracing Tools». Дата звернення: 2026. 15 трав. [URL]. Доступний у: <https://www.brendangregg.com/ebpf.html>
- [24] «Program Type “BPF_PROG_TYPE_TRACING” - eBPF Docs». Дата звернення: 2026. 15 трав. [URL]. Доступний у: https://docs.ebpf.io/linux/program-type/BPF_PROG_TYPE_TRACING/
- [25] «Introduce BPF trampoline», LWN.net. Дата звернення: 2026. 15 трав. [URL]. Доступний у: <https://lwn.net/Articles/804937/>
- [26] «Trampolines - eBPF Docs». Дата звернення: 2026. 15 трав. [URL]. Доступний у: <https://docs.ebpf.io/linux/concepts/trampolines/>
- [27] «bpf: Introduce BPF trampoline · torvalds/linux@fec56f5 · GitHub». Дата звернення: 2026. 15 трав. [URL]. Доступний у: <https://github.com/torvalds/linux/commit/fec56f5890d93fc2ed74166c397dc186b1c25951>
- [28] «[PATCH bpf-next v9 4/4] bpf, arm64: bpf trampoline for arm64 - Xu Kuohai». Дата звернення: 2026. 15 трав. [URL]. Доступний у: <https://lore.kernel.org/bpf/20220711150823.2128542-5-xukuohai@huawei.com/>
- [29] «bpf, tracing: introduce bpf raw tracepoints», LWN.net. Дата звернення: 2026. 15 трав. [URL]. Доступний у: <https://lwn.net/Articles/748352/>

					ІАЛЦ.467200.003 ПЗ	Арк.
						127
Зм.	Арк.	№ докум.	Підпис	Дата		

- [30] «Live-patching security vulnerabilities inside the Linux kernel with eBPF Linux Security Module», The Cloudflare Blog. Дата звернення: 2026. 15 трав. [URL]. Доступний у: <https://blog.cloudflare.com/live-patch-security-vulnerabilities-with-ebpf-lsm/>
- [31] «LSM BPF Programs — The Linux Kernel documentation». Дата звернення: 2026. 15 трав. [URL]. Доступний у: https://docs.kernel.org/bpf/prog_lsm.html
- [32] Т. Høiland-Jørgensen та ін., «The eXpress data path: fast programmable packet processing in the operating system kernel», у *Proceedings of the 14th International Conference on emerging Networking EXperiments and Technologies*, Heraklion Greece: ACM, груд. 2018, С. 54—66. doi: 10.1145/3281411.3281443.
- [33] J. Shin, J. Kim, i J. Nam, «Aquila: Efficient In-Kernel System Call Telemetry for Cloud-Native Environments», *Sensors*, т. 25, вип. 21, С. 6511, жовт. 2025, doi: 10.3390/s25216511.
- [34] М. Hausenblas i S. Schimanski, *Programming Kubernetes: developing cloud-native applications*. Beijing Boston Farnham Sebastopol Tokyo: O'Reilly, 2019.
- [35] «Helper Function “bpf_get_current_cgroup_id” - eBPF Docs». Дата звернення: 2026. 16 трав. [URL]. Доступний у: https://docs.ebpf.io/linux/helper-function/bpf_get_current_cgroup_id/
- [36] «ptrace(2) - Linux manual page». Дата звернення: 2026. 17 трав. [URL]. Доступний у: <https://man7.org/linux/man-pages/man2/ptrace.2.html>
- [37] S. Kharbanda i P. Fonseca, «Always-On Recording Framework for Serverless Computations: Opportunities and Challenges», у *Proceedings of the 1st Workshop on SErverless Systems, Applications and MEthodologies*, Rome Italy: ACM, трав. 2023, С. 41—49. doi: 10.1145/3592533.3592810.
- [38] S. Wei, K. Zhang, i B. Tu, «Performance Impact of Host Kernel Page Table Isolation on Virtualized Servers», у *2021 IEEE Asia-Pacific Conference on Image Processing, Electronics and Computers (IPEC)*, Dalian, China: IEEE, квіт. 2021, С. 912—919. doi: 10.1109/IPEC51340.2021.9421326.
- [39] «Learn the architecture - AArch64 memory management Guide». Дата звернення: 2026. 17 трав. [URL]. Доступний у: <https://developer.arm.com/documentation/101811/0105/>
- [40] «21. Page Table Isolation (PTI) — The Linux Kernel documentation». Дата звернення: 2026. 17 трав. [URL]. Доступний у: <https://www.kernel.org/doc/html/latest/x86/pti.html>
- [41] «Seccomp BPF (SECure COMputing with filters) — The Linux Kernel documentation». Дата звернення: 2026. 17 трав. [URL]. Доступний у: https://www.kernel.org/doc/html/latest/userspace-api/seccomp_filter.html

- [42] «Kernel Probes (Kprobes) — The Linux Kernel documentation». Дата звернення: 2026. 17 трав. [URL]. Доступний у: <https://www.kernel.org/doc/html/latest/trace/kprobes.html>
- [43] M. Gebai і M. R. Dagenais, «Survey and Analysis of Kernel and Userspace Tracers on Linux: Design, Implementation, and Overhead», *ACM Comput. Surv.*, т. 51, вип. 2, С. 1—33, берез. 2019, doi: 10.1145/3158644.
- [44] «eBPF Tutorial by Example 3: Monitoring unlink System Calls with fentry - eunomia». Дата звернення: 2026. 17 трав. [URL]. Доступний у: <https://eunomia.dev/tutorials/3-fentry-unlink/>
- [45] «Cloud Native Computing Foundation Announces Falco Graduation», CNCF. Дата звернення: 2026. 17 трав. [URL]. Доступний у: <https://www.cncf.io/announcements/2024/02/29/cloud-native-computing-foundation-announces-falco-graduation/>
- [46] «Falco», Falco. Дата звернення: 2026. 17 трав. [URL]. Доступний у: <https://falco.org/>
- [47] «The Falco Project», Falco. Дата звернення: 2026. 17 трав. [URL]. Доступний у: <https://falco.org/docs/>
- [48] «Supported Events», Falco. Дата звернення: 2026. 17 трав. [URL]. Доступний у: <https://falco.org/docs/reference/rules/supported-events/>
- [49] «Kernel Events Architecture», Falco. Дата звернення: 2026. 17 трав. [URL]. Доступний у: <https://falco.org/docs/concepts/event-sources/kernel/architecture/>
- [50] «Falco Rules», Falco. Дата звернення: 2026. 17 трав. [URL]. Доступний у: <https://falco.org/docs/concepts/rules/>
- [51] «Default Rules», Falco. Дата звернення: 2026. 17 трав. [URL]. Доступний у: <https://falco.org/docs/reference/rules/default-rules/>
- [52] «Performance», Falco. Дата звернення: 2026. 17 трав. [URL]. Доступний у: <https://falco.org/docs/troubleshooting/performance/>
- [53] «Falco Is Dropping Syscalls Events», Falco. Дата звернення: 2026. 17 трав. [URL]. Доступний у: <https://falco.org/docs/troubleshooting/dropping/>
- [54] «Tetragon - eBPF-based Security Observability and Runtime Enforcement», Tetragon - eBPF-based Security Observability and Runtime Enforcement. Дата звернення: 2026. 17 трав. [URL]. Доступний у: <https://tetragon.io/>
- [55] «Hook points», Tetragon - eBPF-based Security Observability and Runtime Enforcement. Дата звернення: 2026. 17 трав. [URL]. Доступний у: <https://tetragon.io/docs/concepts/tracing-policy/hooks/>
- [56] «Enforcement», Tetragon - eBPF-based Security Observability and Runtime Enforcement. Дата звернення: 2026. 17 трав. [URL]. Доступний у: <https://tetragon.io/docs/concepts/enforcement/>

- [57] «Runtime Hooks», Tetragon - eBPF-based Security Observability and Runtime Enforcement. Дата звернення: 2026. 17 трав. [URL]. Доступний у: <https://tetragon.io/docs/concepts/runtime-hooks/>
- [58] «Events», Tetragon - eBPF-based Security Observability and Runtime Enforcement. Дата звернення: 2026. 17 трав. [URL]. Доступний у: <https://tetragon.io/docs/concepts/events/>
- [59] «Runtime Security Enforcement | KubeArmor». Дата звернення: 2026. 17 трав. [URL]. Доступний у: <https://kubearmor.io/>
- [60] «Policy Spec for Containers | KubeArmor». Дата звернення: 2026. 17 трав. [URL]. Доступний у: https://docs.kubearmor.io/kubearmor/documentation/security_policy_specification
- [61] «Fileless Execution - Tracee». Дата звернення: 2026. 17 трав. [URL]. Доступний у: https://aquasecurity.github.io/tracee/v0.20/docs/events/builtin/signatures/fileless_execution/
- [62] «Map Type “BPF_MAP_TYPE_PERF_EVENT_ARRAY” - eBPF Docs». Дата звернення: 2026. 17 трав. [URL]. Доступний у: https://docs.ebpf.io/linux/map-type/BPF_MAP_TYPE_PERF_EVENT_ARRAY/
- [63] V. Antonenko, *xivihwa/oikesa-agent*. (2026. 25 трав.). Go. Дата звернення: 2026. 25 трав. [URL]. Доступний у: <https://github.com/xivihwa/oikesa-agent>

					ІАЛЦ.467200.003 ПЗ	Арк.
Зм.	Арк.	№ докум.	Підпис	Дата		130

Додаток А

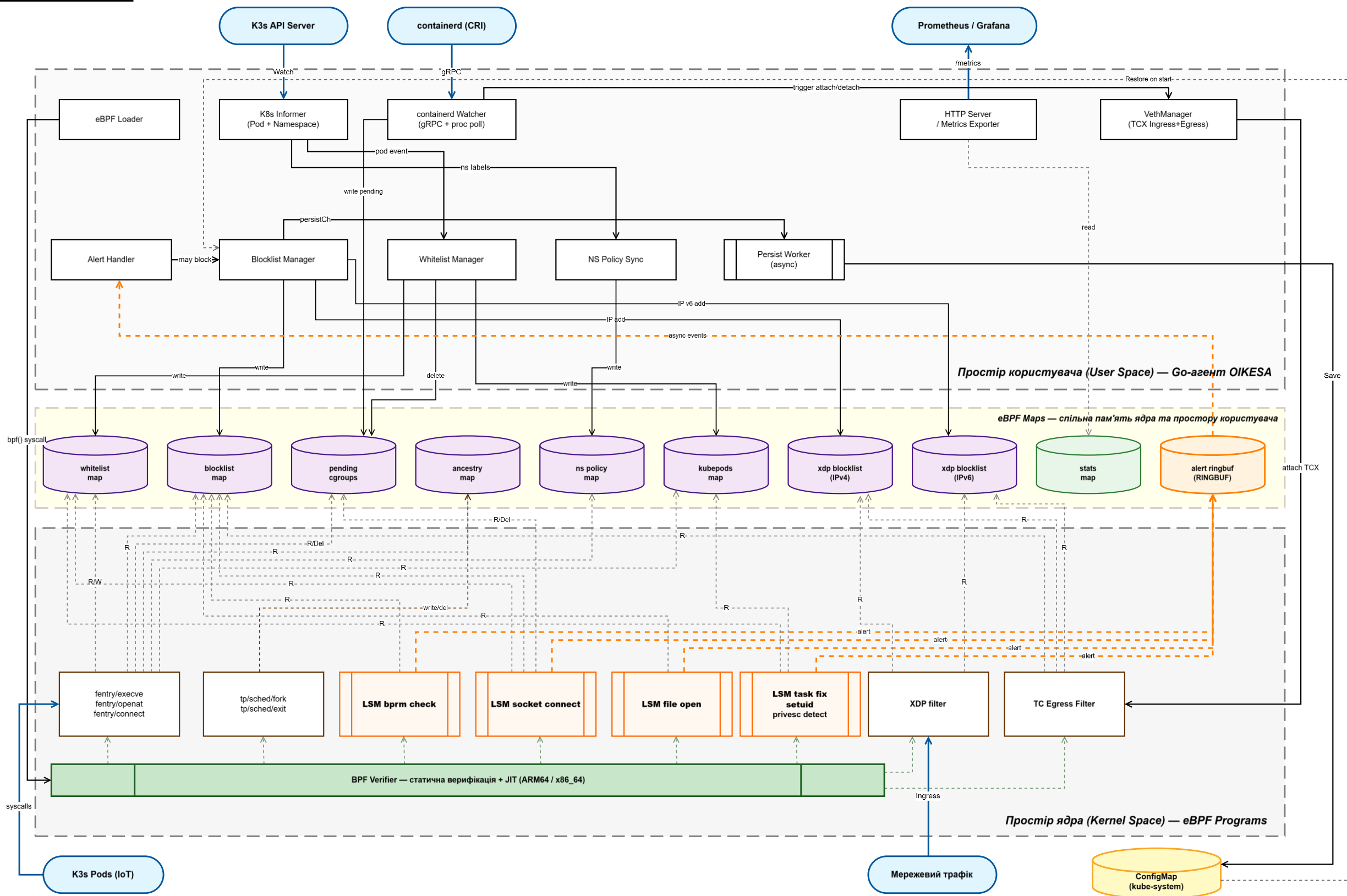
Система безпеки для Kubernetes з використанням
eBPF у IoT середовищах

Структурна схема системи

ІАЛЦ.467200.004 Д1

Аркушів 1

Київ 2026 р



					ІАЛЦ.467200.004 Д1								
					<i>Система безпеки для Kubernetes з використанням eBPF у IoT середовищах</i> Структурна схема системи				Літ.		Маса	Масштаб	
Зм.	Арк.	№Докумен.	Підпис	Дата									
Розробила		Антоненко В.С.											
Перевірів		Ковальчук О.М.											
Реценз.		Омельченко В.В.							Аркуш		Аркушів 1		
Н. Контр.		Коренко Д.В.							<i>КПІ ім. Ігоря Сікорського, ФІОТ, ІО-25</i>				
Затвердив		Волокита А. М.											

Додаток Б

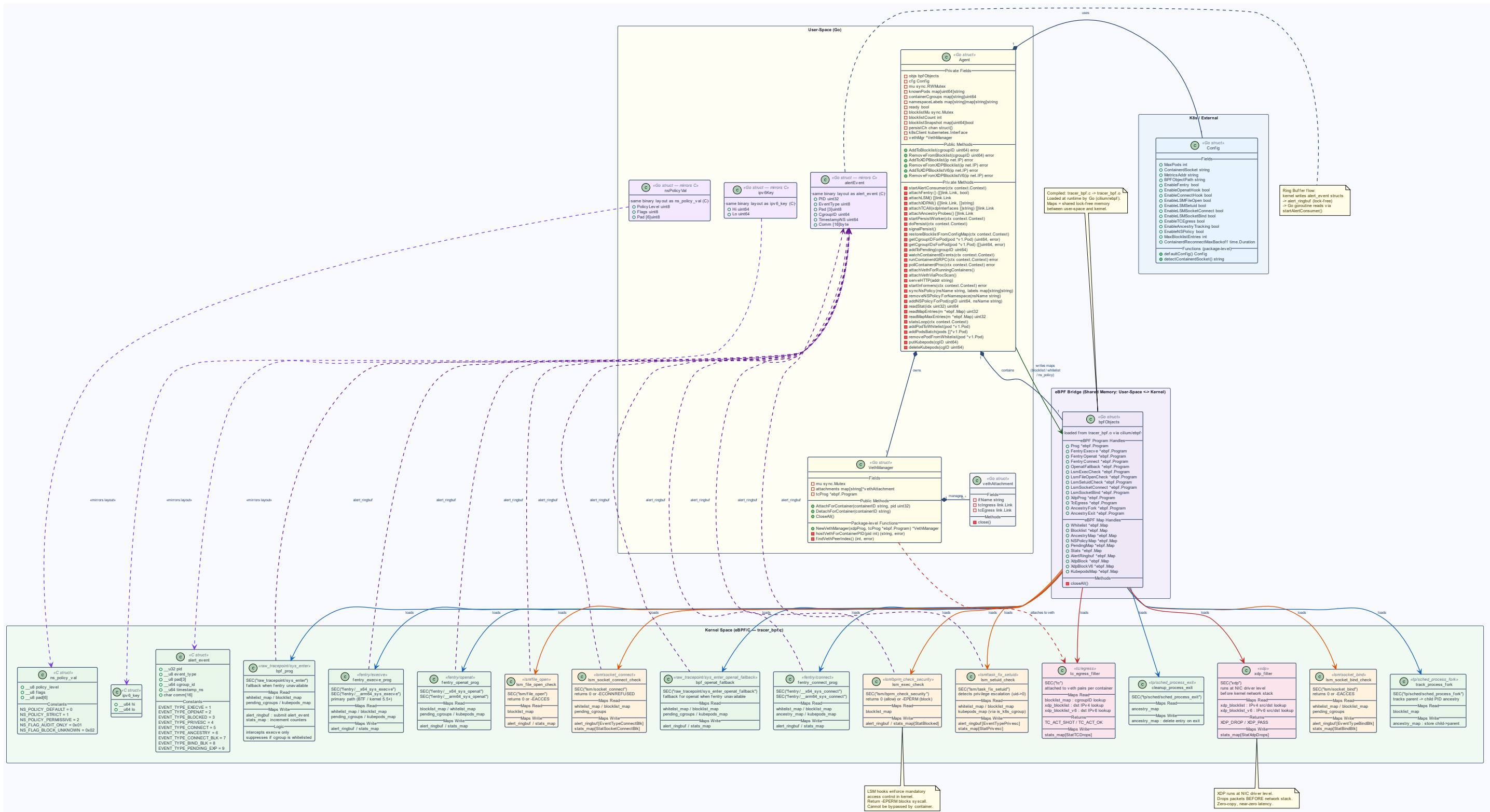
Система безпеки для Kubernetes з використанням
eBPF у IoT середовищах

Функціональна схема (діаграма класів)

ІАЛЦ.467200.005 Д2

Аркушів 1

Київ 2026 р



					ІАЛЦ.467200.005 Д2									
					<i>Система безпеки для Kubernetes з використанням eBPF у IoT середовищах</i> Функціональна схема (діаграма класів)					Літ.		Маса	Масштаб	
Зм.	Арк.	№Докумен.	Підпис	Дата										
Розробила		Антоненко В.С.												
Перевірив		Ковальчук О.М.												
Реценз.		Омельченко В.В.								Аркуш		Аркушів 1		
Н. Контр.		Коренко Д.В.								<i>КПІ ім. Ігоря Сікорського, ФІОТ, ІО-25</i>				
Затвердив		Волокита А. М.												

Додаток В

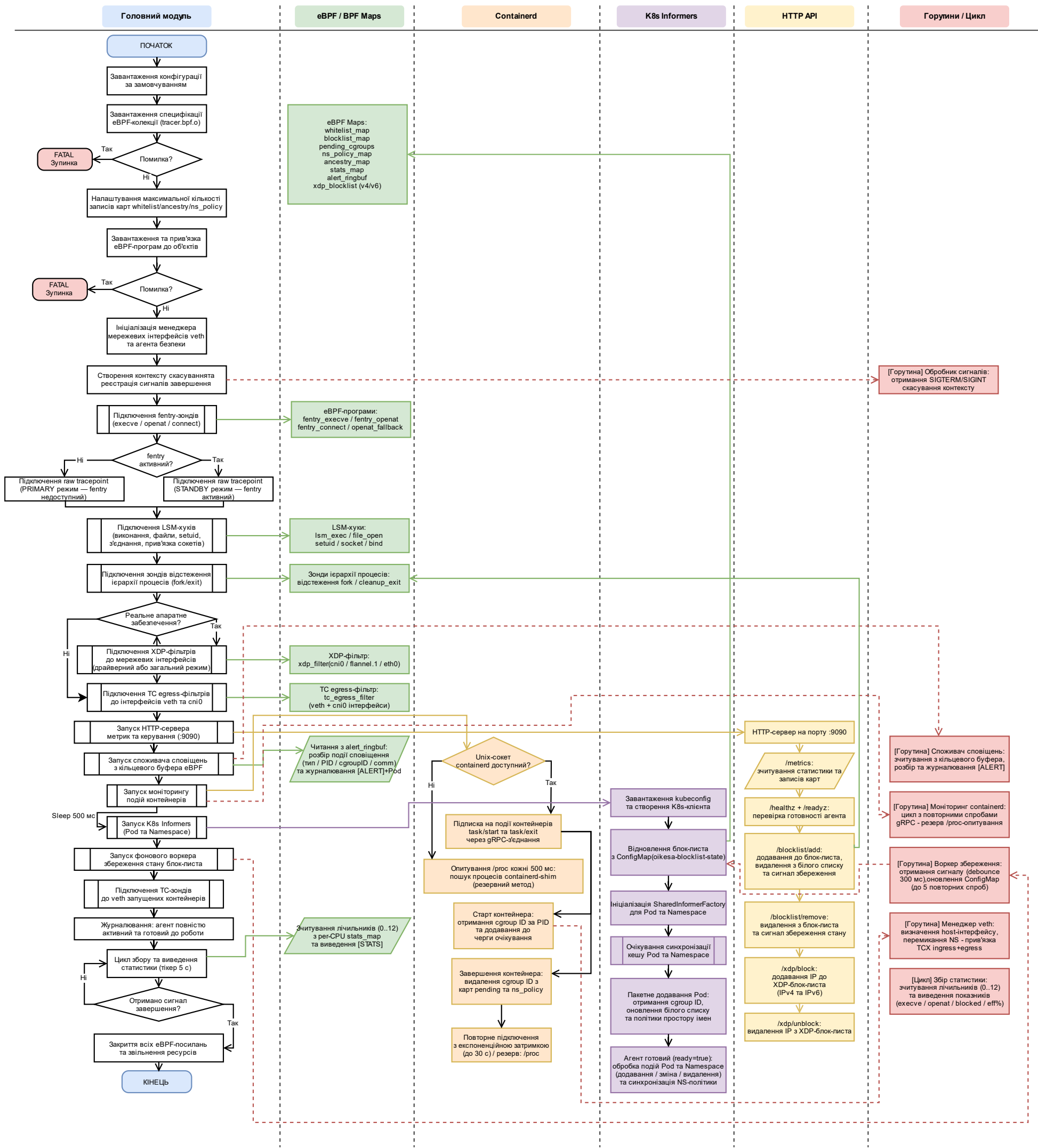
Система безпеки для Kubernetes з використанням
eBPF у IoT середовищах

Алгоритм дій програмного забезпечення

ІАЛЦ.467200.006 ДЗ

Аркушів 1

Київ 2026



Зм.	Арк.	№Докумен.	Підпис	Дата
Розробила		Антоненко В.С.		
Перевірив		Ковальчук О.М.		
Реценз.		Омельченко В.В.		
Н. Контр.		Коренко Д.В.		
Затвердив		Волокита А. М.		

ІАЛЦ.467200.006 ДЗ						
Система безпеки для Kubernetes з використанням eBPF у IoT середовищах Алгоритм дій програмного забезпечення	Літ.			Маса	Масштаб	
	Аркуш			Аркушів 1		
	КПІ ім. Ігоря Сікорського, ФІОТ, ІО-25					

Додаток Г

Текст програмного коду

ІАЛЦ.467200.007 Д4

Аркушів 39

Київ 2026

tracer_bpf.c

```
// tracer.bpf.c – OIKESA eBPF Security Program
//
// Architecture: Orchestrator-Aware In-Kernel Event Suppression
//
// Kernel requirements:
//   fentry/fexit           : Linux 5.5+  (BTF required)
//   raw_tracepoint         : Linux 4.17+ (fallback)
//   BPF_MAP_TYPE_RINGBUF   : Linux 5.8+
//   BPF_MAP_TYPE_PERCPU_ARRAY: Linux 4.6+
//   BPF-LSM hooks          : Linux 5.7+ (CONFIG_BPF_LSM=y, lsm=bpf)
//   lsm/socket_connect     : Linux 5.7+
//   lsm/socket_bind        : Linux 5.7+
//   bpf_skb_cgroup_id()    : Linux 5.7+
//   XDP                    : Linux 4.8+
//   TC/TCX                 : Linux 6.6+ (TCX) or 4.1+ (legacy TC)
//   sched_process_fork tp  : Linux 4.15+
//   CO-RE / BTF            : Linux 5.2+
//   Recommended: Ubuntu 22.04 (5.15) or Ubuntu 24.04 (6.8)

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

char __license[] SEC("license") = "Dual MIT/GPL";

// =====
// COMPATIBILITY DEFINES
// vmlinux.h (generated from BTF) does not include errno constants,
// network protocol numbers, or the __user annotation.
// =====

#ifndef __user
#define __user
#endif
```

					ІАЛЦ.467200.007 Д4			
Зм	Арк.	№ докум.	Підпис	Дата				
Розробила	Антоненко В. С.				Система безпеки для Kubernetes з використанням eBPF у IoT середовищах Текст програмного коду	Літ.	Аркуш	Аркушів
Перевірів	Ковальчук О. М.						1	39
Реценз.	Омельченко В.В.					КПІ ім. Ігоря Сікорського, ФІОТ, ІО-25		
Н. Контр.	Коренко Д. В.							
Затвердив	Волокита А. М.							

```

#ifndef EPERM
#define EPERM      1
#endif

#ifndef EACCES
#define EACCES     13
#endif

#ifndef ECONNREFUSED
#define ECONNREFUSED 111
#endif

```

```

#ifndef ETH_P_IP
#define ETH_P_IP    0x0800
#endif

#ifndef ETH_P_IPV6
#define ETH_P_IPV6  0x86DD
#endif

```

```

#ifndef TC_ACT_OK
#define TC_ACT_OK   0
#endif

#ifndef TC_ACT_SHOT
#define TC_ACT_SHOT 2
#endif

```

```

// =====
// SYSCALL IDs (arch-specific, passed via Makefile -D flags)
// =====

```

```

#ifndef __NR_execve
#if defined(__TARGET_ARCH_x86)
#define __NR_execve 59
#else
#define __NR_execve 221
#endif
#endif

```

```

#ifndef __NR_openat
#if defined(__TARGET_ARCH_x86)
#define __NR_openat 257
#else
#define __NR_openat 56
#endif

```

					ІАЛЦ.467200.007 Д4	Арк.
						2
Зм.	Арк.	№ докум.	Підпис	Дата		

```

#endif
#endif

#ifndef __NR_open
#if defined(__TARGET_ARCH_x86)
#define __NR_open 2
#else
#define __NR_open -1
#endif
#endif

#ifndef __NR_connect
#if defined(__TARGET_ARCH_x86)
#define __NR_connect 42
#else
#define __NR_connect 203
#endif
#endif

// =====
// EVENT TYPES
// =====

#define EVENT_TYPE_EXECVE      1
#define EVENT_TYPE_OPENAT     2
#define EVENT_TYPE_BLOCKED    3
#define EVENT_TYPE_PRIVESC    4
#define EVENT_TYPE_CONNECT    5
#define EVENT_TYPE_ANCESTRY   6
#define EVENT_TYPE_CONNECT_BLK 7
#define EVENT_TYPE_BIND_BLK   8
#define EVENT_TYPE_PENDING_EXP 9

// =====
// ALERT EVENT STRUCTURE – 40 bytes
// offset 0: pid      (u32)
// offset 4: event_type (u8)
// offset 5: pad[3]
// offset 8: cgroup_id (u64)
// offset 16: timestamp_ns (u64)
// offset 24: comm[16]

```

					ІАЛЦ.467200.007 Д4	Арк.
						3
Зм.	Арк.	№ докум.	Підпис	Дата		

```

// =====

struct alert_event {
    __u32 pid;
    __u8 event_type;
    __u8 pad[3];
    __u64 cgroup_id;
    __u64 timestamp_ns;
    char comm[16];
};

// =====
// NS POLICY STRUCTURE
// =====

struct ns_policy_val {
    __u8 policy_level;
    __u8 flags;
    __u8 pad[6];
};

#define NS_POLICY_DEFAULT    0
#define NS_POLICY_STRICT    1
#define NS_POLICY_PERMISSIVE 2
#define NS_FLAG_AUDIT_ONLY   0x01
#define NS_FLAG_BLOCK_UNKNOWN 0x02

// =====
// MAP SIZE CONSTANTS
// =====

#define WHITELIST_MAX    1024
#define BLOCKLIST_MAX    64
#define ANCESTRY_MAX    2048
#define NS_POLICY_MAX    64
#define PENDING_MAX     256
#define XDP_BLOCK_MAX    256
#define XDP_BLOCK_V6_MAX 64
#define STATS_MAX        13
#define KUBEPODS_MAX     1024

```

					ІАЛЦ.467200.007 Д4	Арк.
						4
Зм.	Арк.	№ докум.	Підпис	Дата		

```

#define PENDING_TTL_NS    (5ULL * 1000000000ULL)

// Stats indices – MUST stay in sync with types.go constants:
// [0]=execve_suppressed    [1]=execve_alerts    [2]=blocked
// [3]=xdp_drops            [4]=openat_suppressed [5]=openat_alerts
// [6]=privesc_alerts       [7]=connect_alerts
// [8]=ancestry_blocks      [9]=tc_egress_drops
// [10]=socket_connect_blocks [11]=pending_expired [12]=bind_blocks
#define STAT_EXECVE_SUPP      0
#define STAT_EXECVE_ALERT     1
#define STAT_BLOCKED          2
#define STAT_XDP_DROPS        3
#define STAT_OPENAT_SUPP      4
#define STAT_OPENAT_ALERT     5
#define STAT_PRIVESC          6
#define STAT_CONNECT          7
#define STAT_ANCESTRY_BLK     8
#define STAT_TC_DROPS         9
#define STAT_SOCKET_CONNECT_BLK 10
#define STAT_PENDING_EXPIRED  11
#define STAT_BIND_BLK         12

#define RINGBUF_SIZE    (2 * 1024 * 1024)

// =====
// eBPF MAPS
// =====

// Whitelist – legitimate K3s pod cgroups (LRU for auto-GC)
struct {
    __uint(type, BPF_MAP_TYPE_LRU_HASH);
    __uint(max_entries, WHITELIST_MAX);
    __type(key, __u64);
    __type(value, __u8);
} whitelist_map SEC(".maps");

// Blocklist – actively blocked cgroups (LRU)
struct {
    __uint(type, BPF_MAP_TYPE_LRU_HASH);
    __uint(max_entries, BLOCKLIST_MAX);
    __type(key, __u64);

```

					ІАЛЦ.467200.007 Д4	Арк.
						5
Зм.	Арк.	№ докум.	Підпис	Дата		

```

        __type(value, __u8);
    } blocklist_map SEC(".maps");

// Ancestry map: child_cgroup_id - parent_cgroup_id (blocklist cgroups only)
struct {
    __uint(type, BPF_MAP_TYPE_LRU_HASH);
    __uint(max_entries, ANCESTRY_MAX);
    __type(key, __u64);
    __type(value, __u64);
} ancestry_map SEC(".maps");

// NS Policy map: cgroup_id - ns_policy_val
struct {
    __uint(type, BPF_MAP_TYPE_LRU_HASH);
    __uint(max_entries, NS_POLICY_MAX);
    __type(key, __u64);
    __type(value, struct ns_policy_val);
} ns_policy_map SEC(".maps");

// Pending map: cgroup_id - timestamp_ns
struct {
    __uint(type, BPF_MAP_TYPE_LRU_HASH);
    __uint(max_entries, PENDING_MAX);
    __type(key, __u64);
    __type(value, __u64);
} pending_cgroups SEC(".maps");

// _____
// kubepods_map
// _____
struct {
    __uint(type, BPF_MAP_TYPE_LRU_HASH);
    __uint(max_entries, KUBEPODS_MAX);
    __type(key, __u64);
    __type(value, __u8);
} kubepods_map SEC(".maps");

// Stats – PERCPU_ARRAY for lock-free counters (13 indices)
struct {
    __uint(type, BPF_MAP_TYPE_PERCPU_ARRAY);
    __uint(max_entries, STATS_MAX);

```

					ІАЛЦ.467200.007 Д4	Арк.
						6
Зм.	Арк.	№ докум.	Підпис	Дата		

```

        __type(key,    __u32);
        __type(value, __u64);
    } stats_map SEC(".maps");

// Alert ringbuf
struct {
    __uint(type, BPF_MAP_TYPE_RINGBUF);
    __uint(max_entries, RINGBUF_SIZE);
} alert_ringbuf SEC(".maps");

// XDP/TC blocklist IPv4
struct {
    __uint(type, BPF_MAP_TYPE_LRU_HASH);
    __uint(max_entries, XDP_BLOCK_MAX);
    __type(key,    __u32);
    __type(value, __u8);
} xdp_blocklist SEC(".maps");

// XDP/TC blocklist IPv6
struct ipv6_key {
    __u64 hi;
    __u64 lo;
};

struct {
    __uint(type, BPF_MAP_TYPE_LRU_HASH);
    __uint(max_entries, XDP_BLOCK_V6_MAX);
    __type(key,    struct ipv6_key);
    __type(value, __u8);
} xdp_blocklist_v6 SEC(".maps");

// =====
// HELPERS
// =====

static __always_inline void inc_stat(__u32 idx) {
    if (idx >= STATS_MAX) return;
    __u64 *cnt = bpf_map_lookup_elem(&stats_map, &idx);
    if (cnt)
        __sync_fetch_and_add(cnt, 1);
}

```

					ІАЛЦ.467200.007 Д4	Арк.
						7
Зм.	Арк.	№ докум.	Підпис	Дата		

```

static __always_inline void send_alert(__u32 pid, __u64 cgroup_id, __u8 event_type)
{
    struct alert_event *e = bpf_ringbuf_reserve(&alert_ringbuf,
                                                sizeof(struct alert_event), 0);

    if (!e) return;
    e->pid = pid;
    e->event_type = event_type;
    e->pad[0] = 0; e->pad[1] = 0; e->pad[2] = 0;
    e->cgroup_id = cgroup_id;
    e->timestamp_ns = bpf_ktime_get_ns();
    bpf_get_current_comm(e->comm, sizeof(e->comm));
    bpf_ringbuf_submit(e, 0);
}

// =====
// is_k8s_cgroup
// =====
static __always_inline int is_k8s_cgroup(__u64 cgroup_id) {
    __u8 *v = bpf_map_lookup_elem(&kubepods_map, &cgroup_id);
    return v != NULL;
}

// =====
// OPENAT PATH FILTER
// =====

static __always_inline int is_safe_openat_path(const char *user_path) {
    char path[8] = {};
    long ret = bpf_probe_read_user_str(path, sizeof(path), user_path);
    if (ret <= 0)
        return 0;

    if (path[0] != '/')
        return 0;

    // /proc/
    if (path[1]=='p' && path[2]=='r' && path[3]=='o' && path[4]=='c')
        return 1;
    // /sys/
    if (path[1]=='s' && path[2]=='y' && path[3]=='s')

```

					ІАЛЦ.467200.007 Д4	Арк.
						8
Зм.	Арк.	№ докум.	Підпис	Дата		

```

        return 1;
    // /dev/
    if (path[1]=='d' && path[2]=='e' && path[3]=='v')
        return 1;
    // /run/
    if (path[1]=='r' && path[2]=='u' && path[3]=='n')
        return 1;
    // /tmp/
    if (path[1]=='t' && path[2]=='m' && path[3]=='p')
        return 1;

    return 0;
}

// =====
// CORE SUPPRESSION LOGIC
// =====

static __always_inline void handle_syscall_event(
    __u32 stat_suppress,
    __u32 stat_alert,
    __u8 event_type
) {
    __u64 cgroup_id = bpf_get_current_cgroup_id();
    __u32 pid = bpf_get_current_pid_tgid() >> 32;

    // Blocklist hit - block silently
    __u8 *blocked = bpf_map_lookup_elem(&blocklist_map, &cgroup_id);
    if (blocked) {
        inc_stat(STAT_BLOCKED);
        return;
    }

    // Whitelist hit - suppress without ringbuf (drop-at-source)
    __u8 *allowed = bpf_map_lookup_elem(&whitelist_map, &cgroup_id);
    if (allowed) {
        inc_stat(stat_suppress);
        return;
    }

    // Pending cgroup - TTL-protected promote

```

					ІАЛЦ.467200.007 Д4	Арк.
						9
Зм.	Арк.	№ докум.	Підпис	Дата		

```

__u64 *pending_ts = bpf_map_lookup_elem(&pending_cgroups, &cgroup_id);
if (pending_ts) {
    __u64 now = bpf_ktime_get_ns();
    if (now - *pending_ts < PENDING_TTL_NS) {
        __u8 val = 1;
        bpf_map_update_elem(&whitelist_map, &cgroup_id, &val, BPF_ANY);
        bpf_map_delete_elem(&pending_cgroups, &cgroup_id);
        inc_stat(stat_suppress);
        return;
    }
    bpf_map_delete_elem(&pending_cgroups, &cgroup_id);
    inc_stat(STAT_PENDING_EXPIRED);
    send_alert(pid, cgroup_id, EVENT_TYPE_PENDING_EXP);
    return;
}

// Ancestry check
__u64 *parent_cg = bpf_map_lookup_elem(&ancestry_map, &cgroup_id);
if (parent_cg) {
    __u8 *parent_blocked = bpf_map_lookup_elem(&blocklist_map, parent_cg);
    if (parent_blocked) {
        inc_stat(STAT_ANCESTRY_BLK);
        send_alert(pid, cgroup_id, EVENT_TYPE_ANCESTRY);
        __u8 val = 1;
        bpf_map_update_elem(&blocklist_map, &cgroup_id, &val, BPF_ANY);
        return;
    }
    bpf_map_delete_elem(&ancestry_map, &cgroup_id);
}

// NS Policy check
struct ns_policy_val *nsp = bpf_map_lookup_elem(&ns_policy_map, &cgroup_id);
if (nsp) {
    if (nsp->policy_level == NS_POLICY_PERMISSIVE) {
        inc_stat(stat_suppress);
        return;
    }
    if (nsp->policy_level == NS_POLICY_STRICT) {
        if (nsp->flags & NS_FLAG_BLOCK_UNKNOWN) {
            inc_stat(STAT_BLOCKED);
            return;
        }
    }
}

```

					ІАЛЦ.467200.007 Д4	Арк.
						10
Зм.	Арк.	№ докум.	Підпис	Дата		

```

    }
    if (nsp->flags & NS_FLAG_AUDIT_ONLY) {
        inc_stat(stat_alert);
        send_alert(pid, cgroup_id, event_type);
        return;
    }
}

// Unknown cgroup – K8s or host?
if (!is_k8s_cgroup(cgroup_id))
    return; // host process – silently ignore

// K8s unknown cgroup - alert
inc_stat(stat_alert);
send_alert(pid, cgroup_id, event_type);
}

// =====
// HOOK: sched/sched_process_fork – blocklist ancestry tracking
// =====

struct sched_process_fork_args {
    char parent_comm[16];
    pid_t parent_pid;
    char child_comm[16];
    pid_t child_pid;
};

SEC("tp/sched/sched_process_fork")
int track_process_fork(struct sched_process_fork_args *ctx) {
    __u32 child_pid = (__u32)ctx->child_pid;
    __u32 parent_pid = (__u32)ctx->parent_pid;

    if (child_pid == 0 || parent_pid == 0)
        return 0;

    __u64 parent_cg = bpf_get_current_cgroup_id();
    if (parent_cg == 0)
        return 0;

```

					ІАЛЦ.467200.007 Д4	Арк.
						11
Зм.	Арк.	№ докум.	Підпис	Дата		

```

// Record ancestry ONLY for blocklist cgroups
__u8 *in_blocklist = bpf_map_lookup_elem(&blocklist_map, &parent_cg);
if (!in_blocklist)
    return 0;

__u64 child_cg = parent_cg;
bpf_map_update_elem(&ancestry_map, &child_cg, &parent_cg, BPF_ANY);

return 0;
}

// =====
// HOOK: sched/sched_process_exit – ancestry_map cleanup
// =====

struct sched_process_exit_args {
    char  comm[16];
    pid_t pid;
    int   prio;
};

SEC("tp/sched/sched_process_exit")
int cleanup_process_exit(struct sched_process_exit_args *ctx) {
    __u32 pid = (__u32)ctx->pid;
    if (pid == 0)
        return 0;

    __u64 cgroup_id = bpf_get_current_cgroup_id();
    if (cgroup_id == 0)
        return 0;

    __u64 *parent_cg = bpf_map_lookup_elem(&ancestry_map, &cgroup_id);
    if (!parent_cg)
        return 0;

    bpf_map_delete_elem(&ancestry_map, &cgroup_id);
    return 0;
}

// =====
// HOOK 1a: fentry/execve

```

					ІАЛЦ.467200.007 Д4	Арк.
						12
Зм.	Арк.	№ докум.	Підпис	Дата		

```

// =====

#if defined(__TARGET_ARCH_x86)
SEC("fentry/__x64_sys_execve")
#else
SEC("fentry/__arm64_sys_execve")
#endif
int BPF_PROG(fentry_execve_prog) {
    handle_syscall_event(STAT_EXECVE_SUPP, STAT_EXECVE_ALERT, EVENT_TYPE_EXECVE);
    return 0;
}

// =====
// HOOK 1b: raw_tracepoint/sys_enter – fallback for Linux < 5.5
// =====

SEC("raw_tracepoint/sys_enter")
int bpf_prog(struct bpf_raw_tracepoint_args *ctx) {
    long syscall_id = (long)ctx->args[1];
    if (syscall_id != __NR_execve)
        return 0;
    handle_syscall_event(STAT_EXECVE_SUPP, STAT_EXECVE_ALERT, EVENT_TYPE_EXECVE);
    return 0;
}

// =====
// HOOK 2a: fentry/openat – with in-kernel path filtering
// =====

#if defined(__TARGET_ARCH_x86)
SEC("fentry/__x64_sys_openat")
#else
SEC("fentry/__arm64_sys_openat")
#endif
int BPF_PROG(fentry_openat_prog, struct pt_regs *regs)
{
    __u64 cgroup_id = bpf_get_current_cgroup_id();
    __u32 pid = bpf_get_current_pid_tgid() >> 32;

    // Blocklist
    __u8 *blocked = bpf_map_lookup_elem(&blocklist_map, &cgroup_id);

```

					ІАЛЦ.467200.007 Д4	Арк.
Зм.	Арк.	№ докум.	Підпис	Дата		13

```

if (blocked) {
    inc_stat(STAT_BLOCKED);
    return 0;
}

// Whitelist - suppress
__u8 *allowed = bpf_map_lookup_elem(&whitelist_map, &cgroup_id);
if (allowed) {
    inc_stat(STAT_OPENAT_SUPP);
    return 0;
}

// Pending - TTL check
__u64 *pending_ts = bpf_map_lookup_elem(&pending_cgroups, &cgroup_id);
if (pending_ts) {
    __u64 now = bpf_ktime_get_ns();
    if (now - *pending_ts < PENDING_TTL_NS) {
        __u8 val = 1;
        bpf_map_update_elem(&whitelist_map, &cgroup_id, &val, BPF_ANY);
        bpf_map_delete_elem(&pending_cgroups, &cgroup_id);
        inc_stat(STAT_OPENAT_SUPP);
        return 0;
    }
    bpf_map_delete_elem(&pending_cgroups, &cgroup_id);
    inc_stat(STAT_PENDING_EXPIRED);
    send_alert(pid, cgroup_id, EVENT_TYPE_PENDING_EXP);
    return 0;
}

// Unknown cgroup
const char *filename = (const char *)PT_REGS_PARM2_CORE_SYSCALL(regs);
if (filename && is_safe_openat_path(filename)) {
    inc_stat(STAT_OPENAT_SUPP);
    return 0;
}

// Host process on non-safe path - silently ignore
if (!is_k8s_cgroup(cgroup_id))
    return 0;

// K8s unknown cgroup on suspicious path - alert

```

					ІАЛЦ.467200.007 Д4	Арк.
						14
Зм.	Арк.	№ докум.	Підпис	Дата		

```

        inc_stat(STAT_OPENAT_ALERT);
        send_alert(pid, cgroup_id, EVENT_TYPE_OPENAT);
        return 0;
    }

    // HOOK 2b: raw_tracepoint fallback for openat
    SEC("raw_tracepoint/sys_enter_openat_fallback")
    int bpf_openat_fallback(struct bpf_raw_tracepoint_args *ctx) {
        long syscall_id = (long)ctx->args[1];
        if (syscall_id != __NR_openat && syscall_id != __NR_open)
            return 0;
        handle_syscall_event(STAT_OPENAT_SUPP, STAT_OPENAT_ALERT, EVENT_TYPE_OPENAT);
        return 0;
    }

    // =====
    // HOOK 3: fentry/connect – outbound connection audit
    // =====

    #if defined(__TARGET_ARCH_x86)
    SEC("fentry/__x64_sys_connect")
    #else
    SEC("fentry/__arm64_sys_connect")
    #endif
    int BPF_PROG(fentry_connect_prog) {
        __u64 cgroup_id = bpf_get_current_cgroup_id();
        __u32 pid = bpf_get_current_pid_tgid() >> 32;

        // Whitelist - suppress
        __u8 *allowed = bpf_map_lookup_elem(&whitelist_map, &cgroup_id);
        if (allowed)
            return 0;

        // Blocklist - silent drop
        __u8 *blocked_cg = bpf_map_lookup_elem(&blocklist_map, &cgroup_id);
        if (blocked_cg) {
            inc_stat(STAT_BLOCKED);
            return 0;
        }

        // Pending - TTL check

```

					ІАЛЦ.467200.007 Д4	Арк.
						15
Зм.	Арк.	№ докум.	Підпис	Дата		

```

__u64 *pending_ts = bpf_map_lookup_elem(&pending_cgroups, &cgroup_id);
if (pending_ts) {
    __u64 now = bpf_ktime_get_ns();
    if (now - *pending_ts < PENDING_TTL_NS)
        return 0;
    bpf_map_delete_elem(&pending_cgroups, &cgroup_id);
}

// Ancestry check
__u64 *parent_cg = bpf_map_lookup_elem(&ancestry_map, &cgroup_id);
if (parent_cg) {
    __u8 *parent_blocked = bpf_map_lookup_elem(&blocklist_map, parent_cg);
    if (parent_blocked) {
        inc_stat(STAT_ANCESTRY_BLK);
        send_alert(pid, cgroup_id, EVENT_TYPE_ANCESTRY);
        return 0;
    }
}

// NS Policy check
struct ns_policy_val *nsp = bpf_map_lookup_elem(&ns_policy_map, &cgroup_id);
if (nsp) {
    if (nsp->policy_level == NS_POLICY_PERMISSIVE) {
        inc_stat(STAT_CONNECT);
        return 0;
    }
    if (nsp->policy_level == NS_POLICY_STRICT) {
        if (nsp->flags & NS_FLAG_BLOCK_UNKNOWN) {
            inc_stat(STAT_BLOCKED);
            return 0;
        }
    }
}

// Host process check – ignore non-K8s connect noise
if (!is_k8s_cgroup(cgroup_id))
    return 0;

// K8s unknown cgroup - connect audit alert
inc_stat(STAT_CONNECT);
send_alert(pid, cgroup_id, EVENT_TYPE_CONNECT);

```

					ІАЛЦ.467200.007 Д4	Арк.
						16
Зм.	Арк.	№ докум.	Підпис	Дата		

```

        return 0;
    }

// =====
// HOOK 4: lsm/socket_connect – active outbound blocking
// =====

SEC("lsm/socket_connect")
int BPF_PROG(lsm_socket_connect_check, struct socket *sock,
             struct sockaddr *address, int addrlen) {
    __u64 cgroup_id = bpf_get_current_cgroup_id();

    __u8 *allowed = bpf_map_lookup_elem(&whitelist_map, &cgroup_id);
    if (allowed)
        return 0;

    __u8 *blocked = bpf_map_lookup_elem(&blocklist_map, &cgroup_id);
    if (blocked) {
        __u32 pid = bpf_get_current_pid_tgid() >> 32;
        inc_stat(STAT_SOCKET_CONNECT_BLK);
        send_alert(pid, cgroup_id, EVENT_TYPE_CONNECT_BLK);
        return -ECONNREFUSED;
    }

    __u64 *pending_ts = bpf_map_lookup_elem(&pending_cgroups, &cgroup_id);
    if (pending_ts) {
        __u64 now = bpf_ktime_get_ns();
        if (now - *pending_ts < PENDING_TTL_NS)
            return 0;
        bpf_map_delete_elem(&pending_cgroups, &cgroup_id);
        inc_stat(STAT_SOCKET_CONNECT_BLK);
        send_alert(bpf_get_current_pid_tgid() >> 32, cgroup_id,
EVENT_TYPE_CONNECT_BLK);
        return -ECONNREFUSED;
    }

    return 0;
}

// =====
// HOOK 5: lsm/socket_bind – block bind() for blocklist cgroups

```

					ІАЛЦ.467200.007 Д4	Арк.
Зм.	Арк.	№ докум.	Підпис	Дата		17

```
// =====

SEC("lsm/socket_bind")
int BPF_PROG(lsm_socket_bind_check, struct socket *sock,
             struct sockaddr *address, int addrlen) {
    __u64 cgroup_id = bpf_get_current_cgroup_id();

    __u8 *allowed = bpf_map_lookup_elem(&whitelist_map, &cgroup_id);
    if (allowed)
        return 0;

    __u8 *blocked = bpf_map_lookup_elem(&blocklist_map, &cgroup_id);
    if (blocked) {
        __u32 pid = bpf_get_current_pid_tgid() >> 32;
        inc_stat(STAT_BIND_BLK);
        send_alert(pid, cgroup_id, EVENT_TYPE_BIND_BLK);
        return -EACCES;
    }

    __u64 *pending_ts = bpf_map_lookup_elem(&pending_cgroups, &cgroup_id);
    if (pending_ts) {
        __u64 now = bpf_ktime_get_ns();
        if (now - *pending_ts < PENDING_TTL_NS)
            return 0;
        bpf_map_delete_elem(&pending_cgroups, &cgroup_id);
        inc_stat(STAT_BIND_BLK);
        send_alert(bpf_get_current_pid_tgid() >> 32, cgroup_id, EVENT_TYPE_BIND_BLK);
        return -EACCES;
    }

    return 0;
}

// =====
// HOOK 6: lsm/bprm_check_security – active execve blocking
// =====

SEC("lsm/bprm_check_security")
int BPF_PROG(lsm_exec_check, struct linux_binprm *bprm) {
    __u64 cgroup_id = bpf_get_current_cgroup_id();
    __u8 *blocked = bpf_map_lookup_elem(&blocklist_map, &cgroup_id);

```

					ІАЛЦ.467200.007 Д4	Арк.
Зм.	Арк.	№ докум.	Підпис	Дата		18

```

    if (blocked) {
        __u32 pid = bpf_get_current_pid_tgid() >> 32;
        inc_stat(STAT_BLOCKED);
        send_alert(pid, cgroup_id, EVENT_TYPE_BLOCKED);
        return -EPERM;
    }
    return 0;
}

// =====
// HOOK 7: lsm/file_open – path-based file filtering
// =====

SEC("lsm/file_open")
int BPF_PROG(lsm_file_open_check, struct file *file) {
    __u64 cgroup_id = bpf_get_current_cgroup_id();
    __u8 *blocked = bpf_map_lookup_elem(&blocklist_map, &cgroup_id);
    if (!blocked)
        return 0;

    struct dentry *dentry = BPF_CORE_READ(file, f_path.dentry);
    const unsigned char *dname = BPF_CORE_READ(dentry, d_name.name);

    char fname[32] = {};
    bpf_probe_read_kernel_str(fname, sizeof(fname), dname);

    // shadow
    if (fname[0]=='s' && fname[1]=='h' && fname[2]=='a' &&
        fname[3]=='d' && fname[4]=='o' && fname[5]=='w' && fname[6]=='\0')
        goto block;
    // passwd
    if (fname[0]=='p' && fname[1]=='a' && fname[2]=='s' &&
        fname[3]=='s' && fname[4]=='w' && fname[5]=='d' && fname[6]=='\0')
        goto block;
    // sysrq-trigger
    if (fname[0]=='s' && fname[1]=='y' && fname[2]=='s' &&
        fname[3]=='r' && fname[4]=='q' && fname[5]=='-')
        goto block;
    // token
    if (fname[0]=='t' && fname[1]=='o' && fname[2]=='k' &&
        fname[3]=='e' && fname[4]=='n' && fname[5]=='\0')

```

					ІАЛЦ.467200.007 Д4	Арк.
						19
Зм.	Арк.	№ докум.	Підпис	Дата		

```

        goto block;
// ca.crt
if (fname[0]=='c' && fname[1]=='a' && fname[2]=='.' &&
    fname[3]=='c' && fname[4]=='r' && fname[5]=='t')
    goto block;
// tls.crt, tls.key
if (fname[0]=='t' && fname[1]=='l' && fname[2]=='s' && fname[3]=='.')
    goto block;
// kubeconfig
if (fname[0]=='k' && fname[1]=='u' && fname[2]=='b' &&
    fname[3]=='e' && fname[4]=='c' && fname[5]=='o' && fname[6]=='n')
    goto block;

return 0;

block;;
    __u32 pid = bpf_get_current_pid_tgid() >> 32;
    inc_stat(STAT_BLOCKED);
    send_alert(pid, cgroup_id, EVENT_TYPE_BLOCKED);
    return -EACCES;
}

// =====
// HOOK 8: lsm/task_fix_setuid - privilege escalation detection
// =====

SEC("lsm/task_fix_setuid")
int BPF_PROG(lsm_setuid_check, struct cred *new, const struct cred *old, int flags)
{
    __u32 new_uid = BPF_CORE_READ(new, uid.val);
    __u32 old_uid = BPF_CORE_READ(old, uid.val);

    if (!(new_uid == 0 && old_uid != 0))
        return 0;

    __u64 cgroup_id = bpf_get_current_cgroup_id();
    __u32 pid = bpf_get_current_pid_tgid() >> 32;

    __u8 *allowed = bpf_map_lookup_elem(&whitelist_map, &cgroup_id);
    if (allowed)
        return 0;

```

					ІАЛЦ.467200.007 Д4	Арк.
						20
Зм.	Арк.	№ докум.	Підпис	Дата		

```

    if (!is_k8s_cgroup(cgroup_id))
        return 0;

    inc_stat(STAT_PRIVESC);
    send_alert(pid, cgroup_id, EVENT_TYPE_PRIVESC);

    __u8 *blocked = bpf_map_lookup_elem(&blocklist_map, &cgroup_id);
    if (blocked) {
        inc_stat(STAT_BLOCKED);
        return -EPERM;
    }

    return 0;
}

// =====
// HOOK 9: XDP – network ingress filtering IPv4 + IPv6
// =====

SEC("xdp")
int xdp_filter(struct xdp_md *ctx) {
    void *data_end = (void *) (long) ctx->data_end;
    void *data      = (void *) (long) ctx->data;

    struct ethhdr *eth = data;
    if ((void *) (eth + 1) > data_end)
        return XDP_PASS;

    __be16 proto = BPF_CORE_READ(eth, h_proto);

    if (proto == bpf_htons(ETH_P_IP)) {
        struct iphdr *ip = (void *) (eth + 1);
        if ((void *) (ip + 1) > data_end)
            return XDP_PASS;

        // Check source IP (ingress from external / other node)
        __u32 src_ip = BPF_CORE_READ(ip, saddr);
        __u8 *drop_src = bpf_map_lookup_elem(&xdp_blocklist, &src_ip);
        if (drop_src) {
            inc_stat(STAT_XDP_DROPS);

```

					ІАЛЦ.467200.007 Д4	Арк.
						21
Зм.	Арк.	№ докум.	Підпис	Дата		

```

        return XDP_DROP;
    }

    // Check destination IP (same-node pod-to-pod: XDP sees the packet
    // arriving on cni0 from a veth; dst is the blocked pod's IP)
    __u32 dst_ip = BPF_CORE_READ(ip, daddr);
    __u8 *drop_dst = bpf_map_lookup_elem(&xdp_blocklist, &dst_ip);
    if (drop_dst) {
        inc_stat(STAT_XDP_DROPS);
        return XDP_DROP;
    }

    return XDP_PASS;
}

if (proto == bpf_htons(ETH_P_IPV6)) {
    struct ipv6hdr *ip6 = (void *)(eth + 1);
    if ((void *)(ip6 + 1) > data_end)
        return XDP_PASS;

    // Check source IPv6
    struct ipv6_key key_src = {};
    __u8 saddr[16] = {};
    BPF_CORE_READ_INTO(saddr, ip6, saddr.in6_u.u6_addr8);
    __builtin_memcpy(&key_src.hi, saddr, 8);
    __builtin_memcpy(&key_src.lo, saddr + 8, 8);
    __u8 *drop6_src = bpf_map_lookup_elem(&xdp_blocklist_v6, &key_src);
    if (drop6_src) {
        inc_stat(STAT_XDP_DROPS);
        return XDP_DROP;
    }

    // Check destination IPv6
    struct ipv6_key key_dst = {};
    __u8 daddr[16] = {};
    BPF_CORE_READ_INTO(daddr, ip6, daddr.in6_u.u6_addr8);
    __builtin_memcpy(&key_dst.hi, daddr, 8);
    __builtin_memcpy(&key_dst.lo, daddr + 8, 8);
    __u8 *drop6_dst = bpf_map_lookup_elem(&xdp_blocklist_v6, &key_dst);
    if (drop6_dst) {
        inc_stat(STAT_XDP_DROPS);
    }
}

```

					ІАЛЦ.467200.007 Д4	Арк.
						22
Зм.	Арк.	№ докум.	Підпис	Дата		

```

        return XDP_DROP;
    }

    return XDP_PASS;
}

return XDP_PASS;
}

// =====
// HOOK 10: TC egress – three-level filtering
// =====

SEC("tc")
int tc_egress_filter(struct __sk_buff *skb) {
    // Level 1: cgroup-based blocking
    __u64 cgroup_id = bpf_skb_cgroup_id(skb);
    if (cgroup_id != 0) {
        __u8 *cg_blocked = bpf_map_lookup_elem(&blocklist_map, &cgroup_id);
        if (cg_blocked) {
            inc_stat(STAT_TC_DROPS);
            return TC_ACT_SHOT;
        }
    }

    // Level 2: IPv4 dst block
    if (skb->protocol == bpf_htons(ETH_P_IP)) {
        __u32 dst_ip = 0;
        if (bpf_skb_load_bytes(skb,
            sizeof(struct ethhdr) + offsetof(struct iphdr, daddr),
            &dst_ip, sizeof(dst_ip)) < 0) {
            return TC_ACT_OK;
        }
        __u8 *ip_blocked = bpf_map_lookup_elem(&xdp_blocklist, &dst_ip);
        if (ip_blocked) {
            inc_stat(STAT_TC_DROPS);
            return TC_ACT_SHOT;
        }
        return TC_ACT_OK;
    }
}

```

					ІАЛЦ.467200.007 Д4	Арк.
						23
Зм.	Арк.	№ докум.	Підпис	Дата		

```

// Level 3: IPv6 dst block
if (skb->protocol == bpf_htons(ETH_P_IPV6)) {
    struct ipv6_key key = {};
    __u8 daddr[16] = {};
    if (bpf_skb_load_bytes(skb, 14 + 24, daddr, 16) < 0)
        return TC_ACT_OK;
    __builtin_memcpy(&key.hi, daddr, 8);
    __builtin_memcpy(&key.lo, daddr + 8, 8);
    __u8 *ip6_blocked = bpf_map_lookup_elem(&xdp_blocklist_v6, &key);
    if (ip6_blocked) {
        // Same as IPv4: xdp_blocklist_v6 blocks by destination – always drop.
        inc_stat(STAT_TC_DROPS);
        return TC_ACT_SHOT;
    }
    return TC_ACT_OK;
}

return TC_ACT_OK;
}

```

					ІАЛЦ.467200.007 Д4	Арк.
						24
Зм.	Арк.	№ докум.	Підпис	Дата		

agent.go

```
package main

import (
    "sync"

    "github.com/cilium/ebpf"
    "k8s.io/client-go/kubernetes"

)

type bpfObjects struct {
    Prog          *ebpf.Program `ebpf:"bpf_prog"`
    FentryExecve  *ebpf.Program `ebpf:"fentry_execve_prog"`
    FentryOpenat  *ebpf.Program `ebpf:"fentry_openat_prog"`
    FentryConnect *ebpf.Program `ebpf:"fentry_connect_prog"`
    OpenatFallback *ebpf.Program `ebpf:"bpf_openat_fallback"`
    LsmExecCheck  *ebpf.Program `ebpf:"lsm_exec_check"`
    LsmFileOpenCheck *ebpf.Program `ebpf:"lsm_file_open_check"`
    LsmSetuidCheck *ebpf.Program `ebpf:"lsm_setuid_check"`
    LsmSocketConnect *ebpf.Program `ebpf:"lsm_socket_connect_check"`
    LsmSocketBind  *ebpf.Program `ebpf:"lsm_socket_bind_check"`
    XdpProg       *ebpf.Program `ebpf:"xdp_filter"`
    TcEgress      *ebpf.Program `ebpf:"tc_egress_filter"`
    AncestryFork   *ebpf.Program `ebpf:"track_process_fork"`
    AncestryExit   *ebpf.Program `ebpf:"cleanup_process_exit"`

    Whitelist *ebpf.Map `ebpf:"whitelist_map"`
    Blocklist *ebpf.Map `ebpf:"blocklist_map"`
    AncestryMap *ebpf.Map `ebpf:"ancestry_map"`
    NSPolicyMap *ebpf.Map `ebpf:"ns_policy_map"`
    PendingMap *ebpf.Map `ebpf:"pending_cgrouops"`
    Stats      *ebpf.Map `ebpf:"stats_map"`
    AlertRingbuf *ebpf.Map `ebpf:"alert_ringbuf"`
    XdpBlock     *ebpf.Map `ebpf:"xdp_blocklist"`
    XdpBlockV6   *ebpf.Map `ebpf:"xdp_blocklist_v6"`
    KubepodsMap *ebpf.Map `ebpf:"kubepods_map"`
}

func (o *bpfObjects) closeAll() {
    for _, p := range []*ebpf.Program{
        o.Prog, o.FentryExecve, o.FentryOpenat, o.FentryConnect,
```

					ІАЛЦ.467200.007 Д4	Арк.
						25
Зм.	Арк.	№ докум.	Підпис	Дата		

```

        o.OpenatFallback, o.LsmExecCheck, o.LsmFileOpenCheck,
        o.LsmSetuidCheck, o.LsmSocketConnect, o.LsmSocketBind,
        o.XdpProg, o.TcEgress, o.AncestryFork, o.AncestryExit,
    } {
        if p != nil {
            p.Close()
        }
    }
    for _, m := range []*ebpf.Map{
        o.Whitelist, o.Blocklist, o.AncestryMap, o.NSPolicyMap,
        o.PendingMap, o.Stats, o.AlertRingbuf, o.XdpBlock, o.XdpBlockV6,
        o.KubepodsMap,
    } {
        if m != nil {
            m.Close()
        }
    }
}

type Agent struct {
    objs          bpfObjects
    cfg           Config
    mu            sync.RWMutex
    knownPods     map[uint64]string
    containerCgroups map[string]uint64
    namespaceLabels map[string]map[string]string
    ready         bool

    blocklistMu    sync.Mutex
    blocklistCount int
    blocklistSnapshot map[uint64]bool

    persistCh chan struct{}

    k8sClient kubernetes.Interface

    vethMgr *VethManager
}

```

					ІАЛЦ.467200.007 Д4	Арк.
						26
Зм.	Арк.	№ докум.	Підпис	Дата		

blocklist.go

```
package main
```

```
import (
```

```
    "encoding/binary"
```

```
    "fmt"
```

```
    "log"
```

```
    "net"
```

```
)
```

```
// _____
```

```
// BLOCKLIST API – with rate limiting
```

```
// _____
```

```
func (a *Agent) AddToBlocklist(cgroupID uint64) error {
```

```
    a.blocklistMu.Lock()
```

```
    defer a.blocklistMu.Unlock()
```

```
    if a.blocklistCount >= a.cfg.MaxBlocklistEntries {
```

```
        return fmt.Errorf("blocklist full (%d/%d): reject cgroup=%d – use  
RemoveFromBlocklist first",
```

```
            a.blocklistCount, a.cfg.MaxBlocklistEntries, cgroupID)
```

```
    }
```

```
    value := uint8(1)
```

```
    if err := a.objs.Blocklist.Put(&cgroupID, &value); err != nil {
```

```
        return fmt.Errorf("blocklist put: %w", err)
```

```
    }
```

```
    a.blocklistCount++
```

```
    a.blocklistSnapshot[cgroupID] = true
```

```
    _ = a.objs.Whitelist.Delete(&cgroupID)
```

```
    if a.objs.PendingMap != nil {
```

```
        _ = a.objs.PendingMap.Delete(&cgroupID)
```

```
    }
```

```
    if a.objs.NSPolicyMap != nil {
```

```
        _ = a.objs.NSPolicyMap.Delete(&cgroupID)
```

```
    }
```

```
    log.Printf("[!] BLOCKLIST ADD cgroup=%d count=%d/%d",
```

```
        cgroupID, a.blocklistCount, a.cfg.MaxBlocklistEntries)
```

					ІАЛЦ.467200.007 Д4	Арк.
						27
Зм.	Арк.	№ докум.	Підпис	Дата		

```

        a.signalPersist()
        return nil
    }

func (a *Agent) RemoveFromBlocklist(cgroupID uint64) error {
    a.blocklistMu.Lock()
    defer a.blocklistMu.Unlock()

    inSnapshot := a.blocklistSnapshot[cgroupID]
    bpfErr := a.objs.Blocklist.Delete(&cgroupID)

    if bpfErr != nil && !inSnapshot {
        return fmt.Errorf("blocklist delete: %w", bpfErr)
    }
    if inSnapshot {
        delete(a.blocklistSnapshot, cgroupID)
        if a.blocklistCount > 0 {
            a.blocklistCount--
        }
    }
    if bpfErr != nil {
        log.Printf("[DBG] BLOCKLIST DEL cgroup=%d BPF key not found (LRU
evicted) - count=%d/%d",
                    cgroupID, a.blocklistCount, a.cfg.MaxBlocklistEntries)
    } else {
        log.Printf("[!] BLOCKLIST DEL cgroup=%d count=%d/%d",
                    cgroupID, a.blocklistCount, a.cfg.MaxBlocklistEntries)
    }

    a.signalPersist()
    return nil
}

// -----
// XDP BLOCKLIST API - IPv4 + IPv6
// -----

type ipv6Key struct {
    Hi uint64
    Lo uint64
}

```

					ІАЛЦ.467200.007 Д4	Арк.
						28
Зм.	Арк.	№ докум.	Підпис	Дата		

```

func (a *Agent) AddToXDPBlocklist(ip net.IP) error {
    ip4 := ip.To4()
    if ip4 == nil {
        return fmt.Errorf("not an IPv4: %s", ip)
    }
    v := binary.BigEndian.Uint32(ip4)
    val := uint8(1)
    if err := a.objs.XdpBlock.Put(&v, &val); err != nil {
        return fmt.Errorf("xdp put: %w", err)
    }
    log.Printf("[!] XDP/TC BLOCK IPv4 %s (key=0x%08x, network byte order)",
ip, v)

    return nil
}

func (a *Agent) RemoveFromXDPBlocklist(ip net.IP) error {
    ip4 := ip.To4()
    if ip4 == nil {
        return fmt.Errorf("not an IPv4: %s", ip)
    }
    v := binary.BigEndian.Uint32(ip4)
    return a.objs.XdpBlock.Delete(&v)
}

func (a *Agent) AddToXDPBlocklistV6(ip net.IP) error {
    ip6 := ip.To16()
    if ip6 == nil || ip.To4() != nil {
        return fmt.Errorf("not an IPv6: %s", ip)
    }
    key := ipv6Key{
        Hi: binary.BigEndian.Uint64(ip6[0:8]),
        Lo: binary.BigEndian.Uint64(ip6[8:16]),
    }
    val := uint8(1)
    if err := a.objs.XdpBlockV6.Put(&key, &val); err != nil {
        return fmt.Errorf("xdp_v6 put: %w", err)
    }
    log.Printf("[!] XDP/TC BLOCK IPv6 %s", ip)
    return nil
}

```

					ІАЛЦ.467200.007 Д4	Арк.
						29
Зм.	Арк.	№ докум.	Підпис	Дата		

```

func (a *Agent) RemoveFromXDPBlocklistV6(ip net.IP) error {
    ip6 := ip.To16()
    if ip6 == nil {
        return fmt.Errorf("invalid IPv6: %s", ip)
    }
    key := ipv6Key{
        Hi: binary.BigEndian.Uint64(ip6[0:8]),
        Lo: binary.BigEndian.Uint64(ip6[8:16]),
    }
    return a.objs.XdpBlockV6.Delete(&key)
}

```

					ІАЛЦ.467200.007 Д4	Арк.
						30
Зм.	Арк.	№ докум.	Підпис	Дата		

main.go

```
package main

// =====
// OIKESA – eBPF Security Agent for K3s/IoT
// Architecture: Orchestrator-Aware In-Kernel Event Suppression
// =====

import (
    "context"
    "log"
    "os"
    "os/signal"
    "syscall"
    "time"

    "github.com/cilium/ebpf"
    "github.com/cilium/ebpf/link"
)

// -----
// MAIN
// -----

func main() {
    log.SetFlags(log.Ltime | log.Lshortfile)
    log.Println("=== OIKESA – eBPF Security Agent for K3s/IoT ===")

    cfg := defaultConfig()

    // Load eBPF objects
    objs := bpfObjects{}
    spec, err := ebpf.LoadCollectionSpec(cfg.BPFObjectPath)
    if err != nil {
        log.Fatalf("[FATAL] LoadCollectionSpec(%s): %v", cfg.BPFObjectPath,
err)
    }

    if wl, ok := spec.Maps["whitelist_map"]; ok {
        dynamicSize := uint32(cfg.MaxPods * 3)
        if dynamicSize < 1024 {
```

					ІАЛЦ.467200.007 Д4	Арк.
						31
Зм.	Арк.	№ докум.	Підпис	Дата		

```

        dynamicSize = 1024
    }
    wl.MaxEntries = dynamicSize
    log.Printf("[*] whitelist_map MaxEntries = %d", dynamicSize)
}
if am, ok := spec.Maps["ancestry_map"]; ok {
    am.MaxEntries = 2048
    log.Printf("[*] ancestry_map MaxEntries = 2048 (blocklist-only)")
}
if np, ok := spec.Maps["ns_policy_map"]; ok {
    _ = np
    log.Printf("[*] ns_policy_map MaxEntries = 64")
}

if err := spec.LoadAndAssign(&objs, nil); err != nil {
    log.Fatalf("[FATAL] LoadAndAssign: %v", err)
}

vethMgr := NewVethManager(nil, objs.TcEgress)

agent := &Agent{
    objs:      objs,
    cfg:       cfg,
    knownPods: make(map[uint64]string),
    containerCgroups: make(map[string]uint64),
    namespaceLabels: make(map[string]map[string]string),
    vethMgr:     vethMgr,
    blocklistSnapshot: make(map[uint64]bool),
    persistCh:    make(chan struct{}, 1),
}

ctx, cancel := context.WithCancel(context.Background())
sigCh := make(chan os.Signal, 1)
signal.Notify(sigCh, syscall.SIGTERM, syscall.SIGINT)

var allLinks []link.Link

defer func() {
    log.Println("[*] Closing eBPF links and objects...")
    vethMgr.CloseAll()
    for _, l := range allLinks {

```

					ІАЛЦ.467200.007 Д4	Арк.
						32
Зм.	Арк.	№ докум.	Підпис	Дата		

```

        l.Close()
    }
    objs.closeAll()
    log.Println("[*] OIKESA stopped cleanly.")
}()

go func() {
    sig := <-sigCh
    log.Printf("[*] Received %v – graceful shutdown...", sig)
    cancel()
}()

fentryLinks, fentryActive := agent.attachFentry()
allLinks = append(allLinks, fentryLinks...)

tp, err := link.AttachRawTracepoint(link.RawTracepointOptions{
    Name: "sys_enter", Program: objs.Prog,
})
if err != nil {
    log.Fatalf("[FATAL] raw_tracepoint sys_enter: %v", err)
}
allLinks = append(allLinks, tp)
if fentryActive {
    log.Println("[*] raw_tracepoint/sys_enter: STANDBY")
} else {
    log.Println("[*] raw_tracepoint/sys_enter: PRIMARY")
}

allLinks = append(allLinks, agent.attachLSM()...)

allLinks = append(allLinks, agent.attachAncestryProbes()...)

xdpLinks, xdpInterfaces := agent.attachXDPA11()
allLinks = append(allLinks, xdpLinks...)

tcLinks := agent.attachTC11(xdpInterfaces)
allLinks = append(allLinks, tcLinks...)

go agent.serveHTTP(cfg.MetricsAddr)

agent.startAlertConsumer(ctx)

```

					ІАЛЦ.467200.007 Д4	Арк.
						33
Зм.	Арк.	№ докум.	Підпис	Дата		

```

    agent.watchContainerdEvents(ctx)
    time.Sleep(500 * time.Millisecond)

    if err := agent.startInformers(ctx); err != nil {
        log.Fatalf("[FATAL] Informers: %v", err)
    }

    go agent.startPersistWorker(ctx)

    agent.attachVethForRunningContainers()
    time.Sleep(500 * time.Millisecond)

    log.Println("[*] OIKESA fully active. Ctrl+C or SIGTERM to stop.")
    agent.statsLoop(ctx)
}

```

					ІАЛЦ.467200.007 Д4	Арк.
						34
Зм.	Арк.	№ докум.	Підпис	Дата		

whitelist.go

```
package main

import (
    "fmt"
    "log"
    "strings"

    corev1 "k8s.io/api/core/v1"
)

// _____
// WHITELIST OPERATIONS
// _____

func (a *Agent) putKubepods(cgID uint64) {
    if a.objs.KubepodsMap == nil {
        return
    }
    val := uint8(1)
    if err := a.objs.KubepodsMap.Put(&cgID, &val); err != nil {
        log.Printf("[WARN] kubepods_map.Put cgroup=%d: %v", cgID, err)
    }
}

func (a *Agent) deleteKubepods(cgID uint64) {
    if a.objs.KubepodsMap == nil {
        return
    }
    _ = a.objs.KubepodsMap.Delete(&cgID)
}

func (a *Agent) addPodToWhitelist(pod *corev1.Pod) {
    if pod.Status.Phase != corev1.PodRunning {
        return
    }
    if pod.DeletionTimestamp != nil {
        return
    }

    cgIDs, err := a.getCgroupIDsForPod(pod)
```

```

if err != nil || len(cgIDs) == 0 {
    log.Printf("[WARN] Pod %s/%s: %v", pod.Namespace, pod.Name, err)
    return
}

a.mu.RLock()
nsLabels := a.namespaceLabels[pod.Namespace]
a.mu.RUnlock()
policyLabel := nsLabels["security.oikesa/policy"]
key := fmt.Sprintf("%s/%s", pod.Namespace, pod.Name)

for _, cgID := range cgIDs {
    if a.objs.PendingMap != nil {
        _ = a.objs.PendingMap.Delete(&cgID)
    }
    value := uint8(1)
    if err := a.objs.Whitelist.Put(&cgID, &value); err != nil {
        log.Printf("[ERROR] Whitelist.Put cgroup=%d: %v", cgID, err)
        continue
    }
    a.putKubepods(cgID)

    a.mu.Lock()
    a.knownPods[cgID] = key
    a.mu.Unlock()
    a.addNSPolicyForPod(cgID, pod.Namespace)
}

if policyLabel != "" {
    log.Printf("[+] WHITELIST ADD pod=%-40s cgroups=%v ns-policy=%s",
key, cgIDs, policyLabel)
} else {
    log.Printf("[+] WHITELIST ADD pod=%-40s cgroups=%v", key, cgIDs)
}
}

func (a *Agent) addPodsBatch(pods []*corev1.Pod) {
    keys := make([]uint64, 0, len(pods)*2)
    vals := make([]uint8, 0, len(pods)*2)
    names := make(map[uint64]string, len(pods)*2)
    nsMap := make(map[uint64]string, len(pods)*2)

```

					ІАЛЦ.467200.007 Д4	Арк.
						36
Зм.	Арк.	№ докум.	Підпис	Дата		

```

for _, pod := range pods {
    if pod.Status.Phase != corev1.PodRunning {
        continue
    }
    if pod.DeletionTimestamp != nil {
        continue
    }
    cgIDs, err := a.getCgroupIDsForPod(pod)
    if err != nil || len(cgIDs) == 0 {
        continue
    }
    podKey := fmt.Sprintf("%s/%s", pod.Namespace, pod.Name)
    for _, cgID := range cgIDs {
        keys = append(keys, cgID)
        vals = append(vals, uint8(1))
        names[cgID] = podKey
        nsMap[cgID] = pod.Namespace
        if a.objs.PendingMap != nil {
            _ = a.objs.PendingMap.Delete(&cgID)
        }
    }
}
if len(keys) == 0 {
    return
}

count, err := a.objs.Whitelist.BatchUpdate(keys, vals, nil)
if err != nil {
    if count < len(keys) {
        log.Printf("[WARN] BatchUpdate partial (%d/%d), retrying
remaining individually: %v",
count, len(keys), err)
        for i := count; i < len(keys); i++ {
            cgID := keys[i]
            val := vals[i]
            if putErr := a.objs.Whitelist.Put(&cgID, &val); putErr !=
nil {
log.Printf("[ERROR] Whitelist.Put retry cgroup=%d:
%v", cgID, putErr)
}
}

```

					ІАЛЦ.467200.007 Д4	Арк.
						37
Зм.	Арк.	№ докум.	Підпис	Дата		

```

        }
    } else {
        log.Printf("[WARN] BatchUpdate err=%v", err)
    }
}

for _, cgID := range keys {
    a.putKubepods(cgID)
}

a.mu.Lock()
for cgID, name := range names {
    a.knownPods[cgID] = name
}
a.mu.Unlock()

for cgID, nsName := range nsMap {
    a.addNSPolicyForPod(cgID, nsName)
}

log.Printf("[+] WHITELIST BATCH ADD  %d cgroup entries for %d pods",
len(keys), len(pods))
}

func (a *Agent) removePodFromWhitelist(pod *corev1.Pod) {
    cgIDs, err := a.getCgroupIDsForPod(pod)
    if err != nil || len(cgIDs) == 0 {
        // Fallback: clean up by containerID from cache
        a.mu.Lock()
        for _, cs := range pod.Status.ContainerStatuses {
            cid := strings.TrimPrefix(cs.ContainerID, "containerd://")
            if id, ok := a.containerCgroups[cid]; ok {
                _ = a.objs.Whitelist.Delete(&id)
                _ = a.objs.Blocklist.Delete(&id)
                if a.objs.PendingMap != nil {
                    _ = a.objs.PendingMap.Delete(&id)
                }
                if a.objs.NSPolicyMap != nil {
                    _ = a.objs.NSPolicyMap.Delete(&id)
                }
            }
            a.deleteKubepods(id)
        }
    }
}

```

					ІАЛЦ.467200.007 Д4	Арк.
						38
Зм.	Арк.	№ докум.	Підпис	Дата		

```

        delete(a.knownPods, id)
        delete(a.containerCgroups, cid)
    }
}
a.mu.Unlock()
return
}

key := fmt.Sprintf("%s/%s", pod.Namespace, pod.Name)
for _, cgID := range cgIDs {
    _ = a.objs.Whitelist.Delete(&cgID)
    _ = a.objs.Blocklist.Delete(&cgID)
    if a.objs.PendingMap != nil {
        _ = a.objs.PendingMap.Delete(&cgID)
    }
    if a.objs.NSPolicyMap != nil {
        _ = a.objs.NSPolicyMap.Delete(&cgID)
    }
    a.deleteKubepods(cgID)
    a.mu.Lock()
    delete(a.knownPods, cgID)
    for cid, id := range a.containerCgroups {
        if id == cgID {
            delete(a.containerCgroups, cid)
        }
    }
    a.mu.Unlock()
}
log.Printf("[-] WHITELIST DEL pod=%-40s cgroups=%v", key, cgIDs)
}

```

					ІАЛЦ.467200.007 Д4	Арк.
						39
Зм.	Арк.	№ докум.	Підпис	Дата		