IMPERIAL

KrakenGuard: Towards Fine-Grained eBPF Isolation

Jainil Patel — IIT Roorkee

Lucas Graeff Buhl-Nielsen — Quantco

Adrien Ghosn — Microsoft

Marios Kogias — Imperial College London

What is eBPF ?

- **eBPF (extended Berkeley Packet Filter)** is a **safe, in-kernel virtual machine**
- Allows **user-defined programs** to run inside the **OS kernel**
- Programs are: Loaded at runtime and attached to **kernel hooks** (e.g., networking, tracing)
- Can observe and influence system-wide behavior:
  - Inspect packets, trace processes, interact with kernel state
- **Safety guarantees** enforced by in-kernel verifier:
  - Memory safety, termination

Key idea: Dynamically extend kernel functionality without modifying kernel code

eBPF in Production

- Katran (Meta), Unimog (Cloudflare): production eBPF at hyperscale.
- Hooks: XDP / TC / kprobes — line-rate packet processing and observability
- Widely deployed ⇒ **isolation now matters**

Usage in shared environment

- Increased use in shared environment (different untrusted users sharing same resources).
- Increased usage among users especially cloud developers.
- Growth of projects like **Cilium, KubeArmour, Pixie etc.**
- Most common ways to use eBPF program in shared environments currently are:
    1. Containers
    2. Kubernetes pods
    3. Host-level agents (DaemonSets / system services)

Niche Kernel Feature → Production Systems → Multi-tenant environments

New development for Multi-tenant environments

bpfman

- lifecycle management across hosts and Kubernetes
- Can restrict loading of program based on access rules.
- Eg: Restrict use of kprobe.

Meta's NetEdit

- large-scale orchestration of eBPF network functions

bpfman: An ebpf manager. https://bpfman.io
Netedit, Theophilus A et al. ACM SIGCOMM 2024

Current eBPF permission model

Current model: all-or-nothing permissions

- Loading eBPF program requires access to broad capabilities like CAP_BPF and CAP_PERFMON.
- Get access to all helpers, maps and eBPF program types.
- Same privileges are also needed in shared environment.
    1. Privileged Container
    2. Privileged / capability-enabled Kubernetes pods
    3. Host service with required privileges.

Problems

- Problem related to use of eBPF in multi-tenant environment is far from being solved.
- Broader privileges in containers and pod can be exploited by a user.
- Loaded programs can interfere with the functionality of other program adversely.
- Current solution for isolation leads to performance loss.
- Verifier does not always guarantee safety.

Broad eBPF Privileges Enable Container Escape

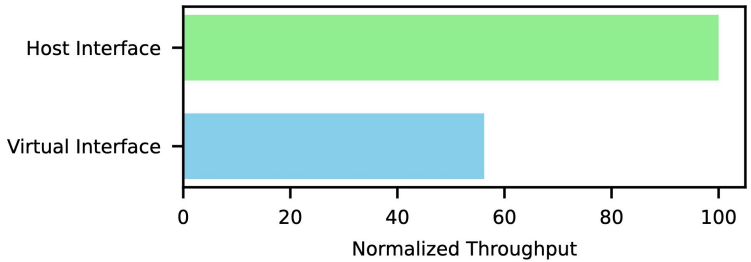
Real-world attacks demonstrate this behavior (CVE-2022-42150)

Problem: coarse-grained permissions expose system-wide capabilities in shared environments

Lack of isolation among eBPF programs

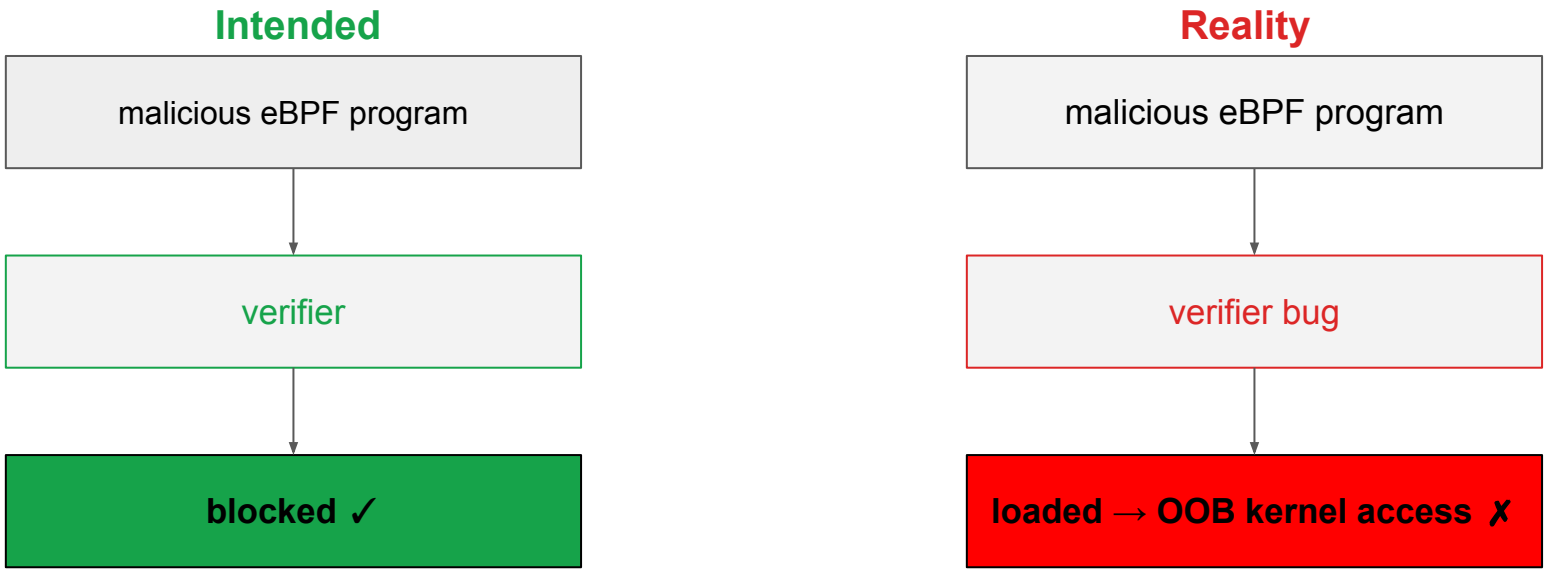
- Privileged host service → restrict a user from loading certain type of program.
- No check over program behaviour.
- Loaded program can interfere with other program.
- Could be solved for selected program types.
- Eg: XDP program loaded on container virtual interface cannot intercept other container’s traffic.

Isolation Should Not Sacrifice the Fast Path



- Isolation often pushes eBPF to a **virtual interface**
- But this gives up the **host NIC fast path**
- Host interface: **>40% higher throughput**

Verifier Bugs Break the Safety Barrier



CVE-2020-8835 CVE-2021-4204 CVE-2022-23222

Why Current Mechanisms Still Leave a Gap

Coarse privileges

- Broad authority once enabled.
- Poor fit for shared hosts.

Linux capabilities

Runtime enforcement

- Restricts behavior at execution time.
- Adds checks on the fast path.

Cross Container Attack[1]

Load-time checks

- Decides before deployment
- Still lacks strong guarantees

BeeGuard[2]

Still missing: strong, fine-grained, **load-time** isolation with **low / zero runtime cost**.

[1] Yi He et al., Cross container attacks: The bewildered eBPF on clouds. USENIX Security 23
[2] Neha Chowdhary, et al., Beeguard: Explainability-based policy enforcement of ebpf codes for cloud-native environments. COMSNETS 2025.

Improving eBPF Isolation in Multi-Tenant Systems

- **Constrain** what an eBPF program is allowed to do with exact guarantees.
- **Check** those constraints at load time → zero runtime overhead.
- **Allow** safe co-location without giving up host performance.

Exploring Pre-runtime Guarantees

- Constraining a program → limiting program's interaction with system
- Providing guarantees over program behavior requires
    1. Knowing all possible execution path a program can take.
    2. Keeping track of program's action on each path
- Knowing this pre-load time helps us ensure constrained execution.
- Helps providing fine-grained isolation and block program loading.

Insight-1 : Finite System interaction

- For eBPF programs interaction with the system reduces to 4 interfaces.
  1. Helpers
  2. Maps
  3. Memory
  4. Return Values

- These determine what the program can invoke, access, modify and affect.

Constraining these actions on every path → constrain the eBPF program

Insight-2 : Symbolic Execution is feasible

General Programs (why sym-exec is hard)	eBPF Programs (why it works here)
• No termination guarantees • Path explosion • Vast surface for analysis	• Verifier guarantees termination • Bounded loops, no unbounded branching. • Limited interaction interface

Exhaustive symbolic execution — exact guarantees, zero approximation.

KRAKENGUARD at a Glance

What it is

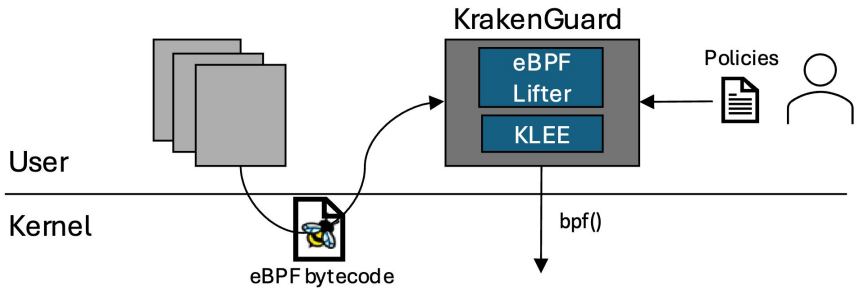
- Trusted **user-space eBPF manager**
- Policy-driven checks at **load time**
- Works **complementary** to the kernel verifier
- No kernel modification required

What it enables

- ✓ Constrain helpers, maps, returns, memory
- ✓ Path-sensitive via symbolic execution
- ✓ **Safe delegation** to unprivileged processes
- ✓ **Safe co-location** of tenant programs

From safe execution → safe delegation + safe co-location

KRAKENGUARD: A Trusted Load-Time eBPF Manager



- Privileged user space daemon
- Unprivileged process sends eBPF Object code over socket
- Analysis object code by lifting it to llvm IR using a eBPF lifter
- Checks **behavior** against policy via symbolic execution

✓ **safe** → load into kernel
✗ **unsafe** → reject

KRAKENGUARD Constraints Program Behavior

eBPF program

Helpers
which calls are permitted

Maps
by name: Read / Write / ReadWrite

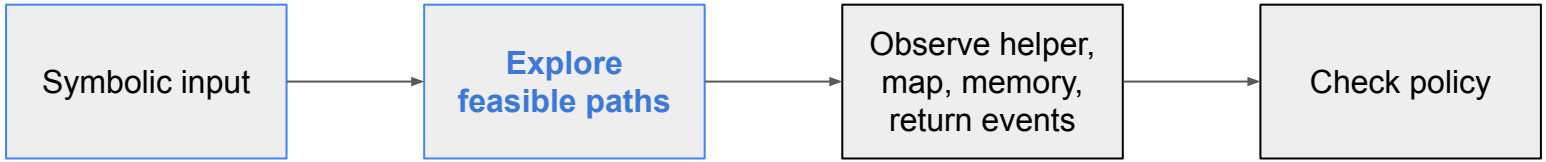
Return values
allowed set per program type

Memory
byte ranges: R / W / RW

+ can also apply conditional constraints

Not who loads — but what the program may do, on every path.

Exhaustive Symbolic Execution at Load Time

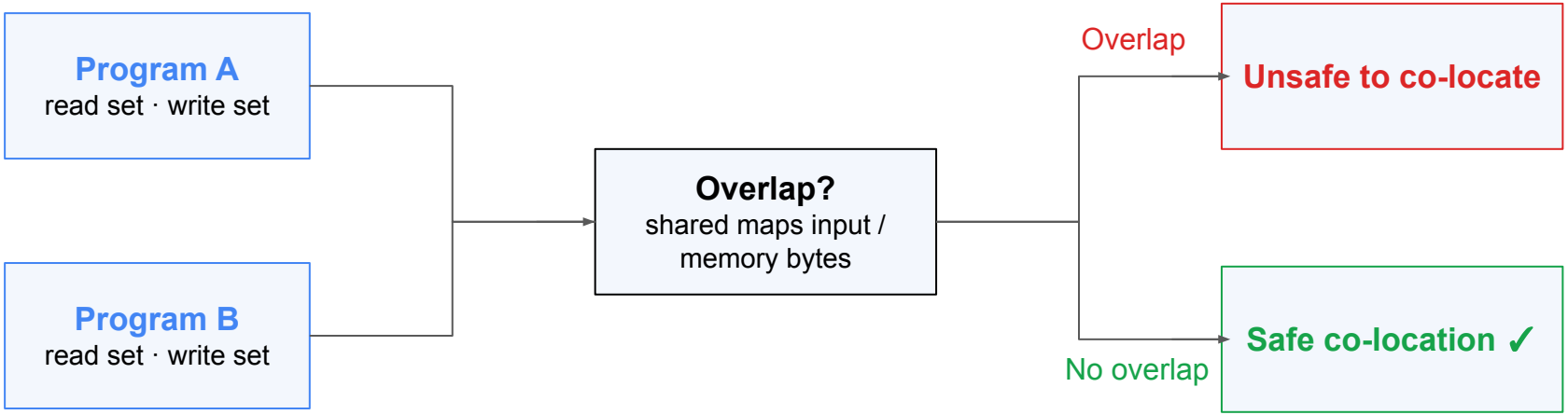


- Symbolic input per program type
  e.g. symbolic packet for XDP
- Path-sensitive - checks every feasible path

- Violation on any path → **reject**
- No violation on all paths → accept

Precise load-time analysis — **not** sampling, not approximation.

Cross-Program Analysis for Safe Co-Location

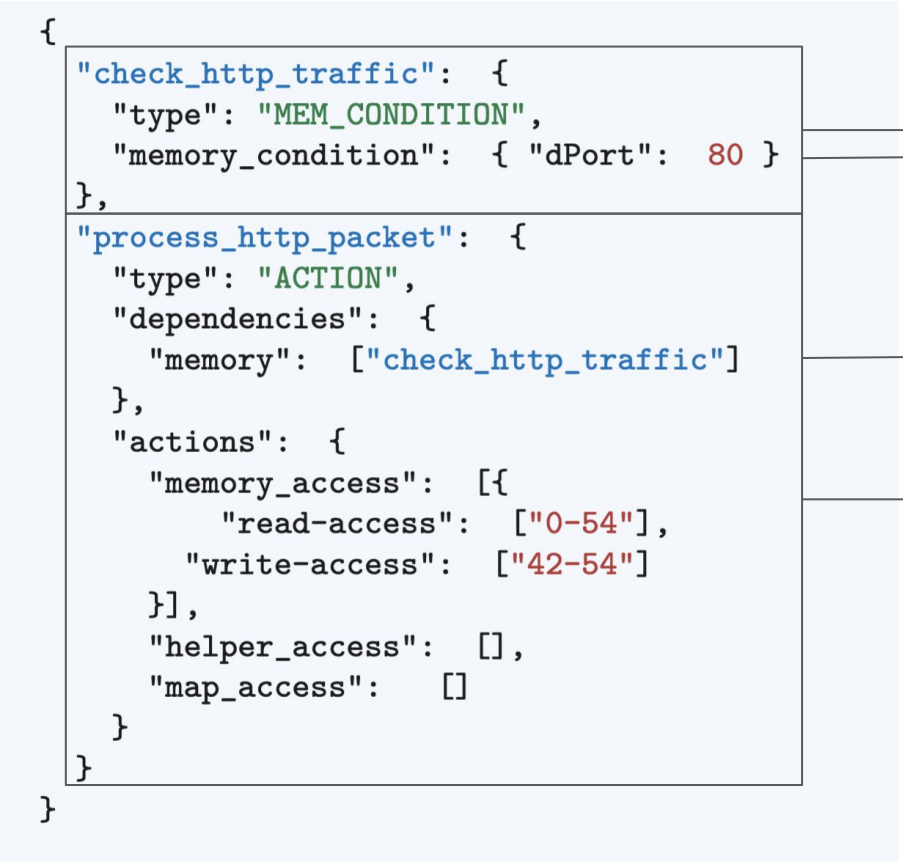


- Per-program policies alone are not enough on a shared host.
- KRAKENGUARD tracks read/write sets per program, per path.
- Enables multiple programs to safely co-locate on the host interface.

Policy: Rules, Actions, and Conditions

- KRAKENGUARD uses a **JSON-based DSL** to define policies.
- A policy can have multiple rules.
- A rule can have one of the following type
    1. Action : What action program can take and what it can access.
    2. Memory Condition : Defines a condition on memory content
- In case of conditional policies each rule can conditionally depend on other rules based on:
    1. Memory content
    2. Previous taken action
- Multiple conditionals can be combined with **AND, OR** and **NOT.**

Example Policy File



Evaluation

Our aim is to show

1. Feasibility of our solution
2. Evaluation of real world programs
3. Identify and block malicious programs
4. XDP as a service usecase

Load-Time Analysis Is Practical

<1 s
most programs

mem < 100 MiB

28.3 s
complex single program

Electrode
FAST_QUORUM_PRUNE

62.7 s
complex program pair

Electrode + Katran
(cross-prog.).

- Evaluated on Katran, Electrode, hXDP, fluvia, linux kernel networking examples and CVEs.
- Cost driven by **program complexity**, not policy complexity.

Practical enough for load-time enforcement — even for real production programs.

KRAKENGUARD can verify real world programs

Electrode

A performance-acceleration framework to optimize distributed protocols, such as Paxos using eBPF.

- **UDP dport = 12345:** read/write header bytes, restricted **maps/helpers**, return **XDP_DROP / XDP_PASS / XDP_TX**
- **Otherwise:** read-only header, **no writes**, return **XDP_PASS**
- FAST_REPLY**:** 21 paths, **0.46 s**
- FAST_QUORUM_PRUNE**:** 77 paths, **28.30 s**

Katran

Meta's open-source L4 load balancer

- Restricts packet, map, and helper access Limited to reading/writing only the header byte. Explored **109 paths** unique paths **in 0.98 s.**

KRAKENGUARD can block malicious programs

Offensive Helpers

✗ Blocks CVE-2022-42150 helper abuse
bpf_override_return, bpf_probe_write_user

<0.32 s

Verifier Escapes

✗ Catches OOB access via verifier-bug CVEs
CVE-2022-23222, CVE-2021-4204, CVE-2020-8835

<0.22 s each

KRAKENGUARD blocks real CVEs and enforces complex real-world program policies.

XDP as a service

- Allows containers to download and safely run XDP program on host interface.
- Achieve host performance with isolation guarantees.
- Evaluated two UDP server belonging to different containers.
- Cross program verification took **0.897 s,** single program for each program took **0.306 s** each.

Conclusion

Problem

- Coarse-grained eBPF permissions.
- Unsafe or costly existing isolation.

Idea

- Trusted load-time manager.
- Policy-driven symbolic analysis.

Real Program Policies

- Constrain helpers, maps, returns, memory.
- Path-sensitive symbolic checking.
- Cross-program interference analysis.

Code: https://github.com/krakenguard-ebpf

Email: jainilpatel2003@gmail.com