USENIX

THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

KrakenGuard: Towards Fine-Grained eBPF Isolation

Jainil Patel, *IIT Roorkee;* Lucas Graeff Buhl-Nielsen, *Quantco;*
Adrien Ghosn, *Microsoft;* Marios Kogias, *Imperial College London*

https://www.usenix.org/conference/nsdi26/presentation/patel

This paper is included in the Proceedings of the 23rd USENIX Symposium on Networked Systems Design and Implementation.

May 4–6, 2026 • Renton, WA, USA

ISBN 978-1-939133-54-0

Open access to the Proceedings of the 23rd USENIX Symposium on Networked Systems Design and Implementation is sponsored by

جامعة الملك عبدالله للعلوم والتقنية
King Abdullah University of Science and Technology

KRAKENGUARD: Towards Fine-Grained eBPF Isolation

Jainil Patel* *IIT Roorkee*
Lucas Graeff Buhl-Nielsen* *Quantco*
Adrien Ghosn *Microsoft*
Marios Kogias *Imperial College London*

Abstract

eBPF is a powerful in-kernel virtual machine that enables dynamic, safe extensions to operating system kernels. Despite the guarantees provided by its in-kernel verifier, eBPF's access control model remains coarse-grained, relying on broad Linux capabilities, such as CAP_BPF. Once granted, these allow unrestricted loading of eBPF programs to different kernel hooks. This poses serious security risks in multi-tenant or untrusted environments, where a compromised or malicious process can misuse eBPF to trace sensitive activity, access kernel memory, or disrupt system behavior. While existing verification ensures safety properties, it cannot enforce fine-grained constraints on what programs can do.

We present KRAKENGUARD, a trusted user-space manager that enforces fine-grained, policy-driven constraints on eBPF bytecode at load time. Using symbolic execution, it checks all program paths for compliance with policies on helper usage, memory accesses, and return values. It enables safe delegation of program loading by unprivileged processes and detects cross-program interference to ensure safe co-location of eBPF programs on the same host.

We show that KRAKENGUARD can block the misuse of restricted helpers, unauthorized memory and map access, and unsafe packet modifications in real-world eBPF programs, while also being able to detect existing CVEs. As a use case, we implement an XDP-as-a-Service application that securely runs XDP programs belonging to different tenants directly on the host interface after guaranteeing they cannot do anything malicious and that they do not interfere with each other.

1 Introduction

The extended Berkeley Packet Filter (eBPF) [14] is a powerful in-kernel virtual machine that allows the safe execution of user-defined programs within the operating system kernel. Originally developed for low-overhead packet filtering, eBPF rapidly evolved into a general-purpose mechanism for observability [42, 43], networking [13, 44], and security enforcement [13, 19] in modern systems [48]. Today, it underpins core functionalities in leading infrastructure projects such as Cilium for Kubernetes networking [7], Meta's load balancer Katran [46], and Cloudflare's load balancer Unimog [36], while hyperscalers such as Meta and Google run tens of eBPF programs in their production servers [48]. Its integration into major cloud providers and operating systems, including Linux, Windows [37], and several BSD variants [35], has made eBPF one of the most pervasive kernel extensibility and instrumentation technologies in production. This widespread adoption is driven by eBPF's ability to dynamically and safely extend kernel behavior without modifying kernel code. Strong safety guarantees, such as memory safety, guaranteed termination, and restricted kernel interaction, are enforced through static verification at load time by the in-kernel verifier [16].

As eBPF's adoption scales across diverse infrastructure environments, robust orchestration and lifecycle management tools are increasingly needed. Traditional methods of deploying and managing eBPF programs, often manual and host-specific, struggle to meet the demands of complex, large-scale, dynamic, or even multi-tenant systems. In response, new management frameworks emerged to streamline eBPF operations at scale. For example, `bpfman` [3] is an open-source eBPF manager designed to simplify the deployment, modification, and monitoring of eBPF programs across both standalone Linux hosts and Kubernetes clusters. Similarly, Meta has developed NetEdit [51], a production-grade orchestration platform that manages the composition, deployment, and lifecycle of eBPF-based network functions across thousands of servers. These developments highlight a shift towards structured eBPF management, where orchestration platforms not only enhance operational efficiency but also serve as critical components for enforcing security, ensuring compliance, and maintaining system stability in complex, distributed environments.

Despite the existence of eBPF managers and eBPF's wide adoption, the eBPF security model remains fundamentally coarse-grained. The prevailing permission structure grants

*Work done while at Imperial

privileges at the process or container level, relying on capabilities [6,25], *e.g.,* `CAP_BPF`, which – once granted – allow unrestricted access to attach, modify, and run eBPF programs. This "all-or-nothing" model creates a substantial security blind spot, especially in multi-tenant or containerized environments where a process with sufficient capabilities can misuse eBPF to exfiltrate data, trace sensitive execution paths, or disrupt kernel behavior. A growing body of research and incident reports [10,38,59,65] demonstrates how malicious or buggy eBPF programs can exploit their privileges to undermine system security, *e.g.,* achieving container escapes. The lack of a fine-grained and robust isolation or security-enforcing mechanism allows only privileged processes/users to leverage eBPF benefits, while unprivileged ones are either completely unable to use eBPF or have to resort to expensive workarounds to guarantee isolation that harm performance, *e.g.,* run eBPF on virtual interfaces.

We believe there is an emerging need for a new type of trusted eBPF manager that can apply *security and isolation policies* at program load time. Such a trusted manager can run in userspace with elevated privileges to load eBPF programs and work complementarily to the kernel. It can use more elaborate and compute-heavy techniques to test and verify fine-grained policies, while the kernel verifier focuses purely on liveness and safety properties. Non-privileged applications that need to load an eBPF program communicate directly with the trusted manager and delegate loading, while the manager ensures the program is safe to load according to the policies and access rights of the communicating process before loading it. Such an approach follows similar trends towards retrofitting a micro-kernel design to Linux [60,71], without increasing the complexity of the in-kernel verifier [64].

The insight behind the proposed approach stems from a distinctive feature of eBPF that enables rigorous static analysis. All eBPF programs must pass a verification process before being loaded. The in-kernel verifier performs bounded loop analysis, checks memory safety, ensures type correctness, and enforces control flow constraints to guarantee that eBPF programs terminate safely and do not corrupt kernel state. To satisfy this mandatory step, eBPF developers, intentionally or through long trial-and-error efforts, provide a strong foundation upon which additional, more precise static analyses can be layered. Unlike general-purpose code, eBPF bytecode operates within a tightly controlled execution model, making it well-suited for advanced formal reasoning. As a result, static analysis tools can check for higher-level security, compliance, correctness, and interference properties without requiring any further help from eBPF programmers.

We design KRAKENGUARD[1], to our knowledge, the first eBPF security manager that refines eBPF's access control model by applying fine-grained policies for eBPF programs at load time. KRAKENGUARD leverages exhaustive symbolic execution to explore all execution paths of an eBPF program for any potential input and checks those against a given set of policies. These policies govern: (1) how a program can access and modify its input, *e.g.,* the parts of a packet an XDP [33] program can read or write; (2) the program's interactions with the kernel and userspace, *e.g.,* its allowed eBPF helpers and eBPF maps; (3) the program's interference with other collocated eBPF programs, *e.g.,* guaranteeing it is impossible for two programs to access the same memory, thus ensuring isolation by proving the lack of interference.

We build KRAKENGUARD on top of KLEE [53], a symbolic execution engine for LLVM IR. KRAKENGUARD receives eBPF bytecode and can apply policies that constrain and/or refine: (i) eBPF helpers, (ii) return values, (iii) eBPF maps, and (iv) memory accesses, either for the entire program or *conditionally* for certain execution paths based on previous program actions.

We use KRAKENGUARD with a variety of eBPF programs from academic papers and the Linux kernel, and implement a range of security policies to demonstrate its practicality. KRAKENGUARD's execution times commonly remain below 30 seconds, even for relatively complex eBPF programs. We show that KRAKENGUARD can prevent well-known CVEs that depend on offensive eBPF features, but also CVEs that target vulnerabilities in the kernel verifier [8–11], thus working complementarily to the kernel verifier toward a defense-in-depth solution. Going one step further, we suggest that KRAKENGUARD can be used not only to strengthen isolation, but also to improve the performance of eBPF execution in a containerized environment. We implement an `XDP`-as-a-Service application that allows containers to download and safely run `XDP` programs on the host interface, thus achieving native performance, while ensuring that these programs cannot interfere with each other nor can they harm the host despite the lack of other isolation mechanisms.

2 eBPF: Guarantees and Shortcomings

Despite eBPF gaining traction with a wide range of powerful use cases, its availability in multi-tenant environments remains challenging [40,41]. In this section, we first briefly describe how eBPF works and its main use cases; then, we analyze the existing state-of-the-art mechanisms for eBPF access control and confinement. We explain what can go wrong with the existing isolation mechanisms, both from a security and performance point of view, and motivate the need for a better and more fine-grained approach to eBPF isolation.

2.1 eBPF in the Linux Kernel

eBPF was introduced in the Linux kernel in 2014 as a means to enable safe and efficient execution of user-defined bytecode within kernel space. eBPF depends on a powerful virtual

[1] https://github.com/krakenguard-ebpf

Tool/ Solution	Helper Function	Map	Data Access	Cross Program Analysis	Memory Safety Guarantee	Type of Approach
Current Linux Permission Model	×	×	×	×	×	Load-Time
BPF Token [5]	×	△ (type of Maps, not specific)	×	×	×	Load-Time
CapBits [59]	✓	△ (type of Maps, not specific)	×	×	×	Runtime
BeeGuard [54]	✓	△	△	×	×	Load-Time
KRAKENGUARD	✓	✓	✓	✓	✓	Load-Time

×: Completely unavailable, ✓: Provides solid guarantees, △: Partially supported (not exact/inaccurate)

Table 1: Comparison of different eBPF access management approaches and their features.

machine. Programmers usually write eBPF programs in high-level languages that are then compiled down to eBPF bytecode and loaded into the kernel. As part of the loading phase, an in-kernel verification process ensures safety (*e.g.,* no loops or invalid memory access) and just-in-time (JIT) compilation improves performance. For more complex functionality and interaction with the rest of the system, eBPF programs use eBPF helpers, *i.e.,* built-in kernel functions. Examples of helpers are `bpf_csum_diff()` to implement checksums or `bpf_probe_write_user()` to write to userspace memory. To manage state across different programs and program invocations, and to communicate with userspace, eBPF programs use eBPF maps [2]. Depending on the use case, there are different types of maps, ranging from simple key-value store interfaces, *e.g.,* `BPF_MAP_TYPE_ARRAY` [28], to more complicated ones, such as `BPF_MAP_TYPE_PROG_ARRAY` [29], used to manage eBPF tail-calls from one program to another.

eBPF programming follows an event-driven model with eBPF programs being attached to different hooks inside the kernel, thus supporting different use cases. For example, Linux provides hooks throughout the networking stack (`XDP` [61], `TC`, and `SOCK` hooks) with eBPF programs implementing functionalities such as network functions (NFs), *e.g.,* load balancing [46, 80], quality of service, traffic shaping, and stack configurations [51, 67]. Outside of networking, eBPF is used in tracing (*e.g.,* via `kprobes`, `uprobes`, and `tracepoints`), performance monitoring (*e.g.,* perf_events), and security (*e.g.,* via LSM hooks to enforce policies).

2.2 eBPF Access Control

eBPF access control is coarse-grained and remains so, despite the efforts to refine it. eBPF enforces access control through Linux capabilities [6, 25], which are generic permissions that govern what a process can do. The kernel uses these capabilities to restrict which processes can load and attach eBPF programs and which eBPF hooks each process can use. Processes with the `CAP_SYS_ADMIN` capability have access to all eBPF features, but this comes with many other strong privileges that can be dangerous in containerized environments. Thus, newer kernel versions starting from Linux 5.6 introduced dedicated eBPF and domain-specific capabilities. Loading eBPF programs requires `CAP_BPF`, using network-related hooks requires `CAP_NET_ADMIN`, while attaching monitoring and perf probes can be enabled through `CAP_PERFMON`.

Even after the addition of more capabilities, though, eBPF access control remains close to an "all-or-nothing" approach. For example, a process with `CAP_BPF` and `CAP_PERFMON` can load any tracing program, observe the behavior of any other process in the system, and abuse helpers such as `bpf_probe_read_kernel()` and `bpf_probe_read_user()` to read arbitrary memory. Section 2.3 describes such attack scenarios in more detail. Given that `CAP_BPF` cannot be namespaced or sandboxed, it is impossible to ensure that such programs only read memory belonging to processes in a given namespace.

Acknowledging the problem, the research and kernel communities have proposed solutions to provide more fine-grained access control to the eBPF subsystem, but only partially support features desired for multi-tenancy. Bpftoken [5] allows for delegating a subset of eBPF functionality from a privileged system-wide daemon to a trusted unprivileged application. Yet, bpftoken is limited to functionalities accessed through the `bpf()` syscall. Hence, it can only restrict an unprivileged application from loading certain kinds of programs or using eBPF maps. CapBits [59] uses the eBPF LSM subsystem [4] to implement fine-grained access control for eBPF features. However, it depends on expensive runtime checks, which can harm performance for syscall-heavy or eBPF-heavy use cases, and requires extensive kernel modifications. Another work, BeeGuard [54], enforces policies on eBPF programs before they are loaded and constrains helper functions and memory accesses based on static analysis and natural language processing. Yet, it is an approximate solution that cannot provide strong guarantees.

Table 1 lists the available proposals for managing eBPF permissions along with their provided features, and explains how these permissions are applied at runtime or program load time. All systems, apart from the current Linux implementation, can restrict access to helpers. BpfToken and CapBits can only restrict access to different map types, but not spe-

Helper Function	Offensive Capability
bpf_probe_write_user	Write to user space memory
bpf_probe_read_user	Read user space memory
bpf_override_return	Override kernel function return value
bpf_map_get_fd_by_id	Get map file descriptor (for tampering)
bpf_send_signal	Send signals to processes

Table 2: List of eBPF helper functions and their offensive capabilities.

cific maps identified by name. BeeGuard does so, but suffers from accuracy issues due to a lack of solid verification and isolation strategy. BeeGuard is the only tool that attempts to constrain the data accesses performed by the eBPF program, *e.g.,* how an XDP program accesses different parts of a packet, but with a similar lack of strong guarantees due to its NLP-based approach. KRAKENGUARD is the only solution that restricts access to maps identified by names and constrains data accesses in a fine-grained manner. It also goes beyond by providing cross-program analysis, identifying interference across different programs, and providing memory safety guarantees by applying the principle of least privilege, which limits what an eBPF program can do and which parts of memory it can access, achieved via static analysis at load time.

2.3 eBPF-based Attacks

To demonstrate that using eBPF in its current form requires full trust in the application loading the program, despite the in-kernel verifier, and it is therefore unsuitable for multi-tenant or untrusted environments, we present a few attack scenarios illustrating how eBPF features can be used offensively. These scenarios represent large classes of attacks indicating that the current job of the in-kernel verifier is to guarantee liveness and safety in the presence of non-malicious eBPF programs. We show how malicious programs that pass the verifier can mount attacks, *e.g.,* escape container boundaries or cause Denial of Service, with or even without exploiting verifier bugs.

Container escape: The ability to call any helper function provides a toolset of offensive abilities that can lead to privilege escalation [59]. Table 2 shows a sample list of helper functions and their offensive capability. The CVE-2022-42150 [11] includes a set of attacks that use eBPF helper functions to perform container escape. One such example is the attack on the kubelet process, which is Kubernetes' node agent. In this attack, the attacker attaches a kernel return probe to the `__x64_sys_read()` syscall, which executes whenever the `read()` syscall returns. The attack on the kubelet process uses `bpf_override_return()` and `bpf_probe_write_user` helper functions to override the syscall's return value and inject a malicious YAML payload. This YAML payload is a Kubernetes Pod specification which gives `SYS_ADMIN` privileges to the pod. The pod thus has the privilege to perform file system operations (*e.g.,* mounting/unmounting), trace any process and update its memory, abuse exposed host directories, disable the host SELinux from a container [34], and even escape the container [6, 17, 18].

Map Tampering: Maps are an integral part of the eBPF ecosystem. Programs such as firewalls and load balancers use maps to store important configuration information, e.g., blacklisted IP addresses. Tracing and profiling programs use maps to store and send critical tracing information, such as system call arguments and stack traces to userspace applications. Larger eBPF applications such as datadog [12], tetragon [32], and cilium [7] use `BPF_MAP_TYPE_PROG_ARRAY` maps storing pointers to eBPF programs to implement tail calls.

Any process with sufficient privileges [25] can access and manipulate any eBPF map in the system via `bpf_map_get_fd_by_id()`, leading to map tampering attacks. In a firewall, such attacks allow the attacker to block ranges of IPs leading to Denial of Service or to whitelist dangerous endpoints, a first step towards more advanced attacks [59]. Access to maps that store tracing information can lead to silent data corruption in observability pipelines, information theft, or even enable bypassing kernel address space layout randomization (KASLR) [21, 59]. Tampering with tail calls enables an attacker to replace existing programs with malicious ones to circumvent network restrictions or install backdoors into the cluster [38].

Verifier escape: The eBPF verifier is not completely free of bugs [39, 56, 70, 72] and can be bypassed to load malicious programs in the kernel. CVE-2022-23222 [10] describes such a case. An eBPF program can exploit a bug in the verifier that does not prohibit arithmetic operations on certain `OR_NULL` pointers in the eBPF code, such as the `PTR_TO_MEM_OR_NULL` [30]. Using this vulnerability, attackers can gain out-of-bound read or write access to the kernel memory, given that eBPF programs are not further sandboxed or isolated after the in-kernel verifier endorses them. Other examples of such attacks are CVE-2020-8835 [8] and CVE-2021-4204 [9].

2.4 Performance Incentives

We also study the potential performance impact of existing approaches to isolate specifically `XDP` programs in a multi-tenant environment. The canonical approach to do so in a containerized deployment, *e.g.,* on Kubernetes, is to attach the `XDP` program to the container's virtual network interface. This way this program cannot interfere with the rest of the traffic on the host via the equivalent routing configuration, while any malicious traffic the program might generate will be dropped through host firewall mechanisms, *e.g.,* using `iptables`.

We measure the performance drop when running an XDP program on a virtual interface versus running it directly on the host interface. For our experiment, we use a simple UDP echo server. In one case, we run a container and deploy the eBPF program on its virtual interface connected to a Linux

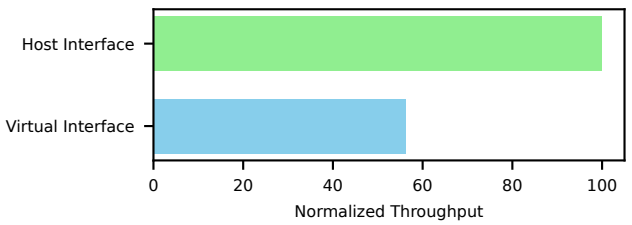

Figure 1: Throughput comparison for an XDP UDP echo server attached to a virtual interface versus attached directly to the host interface.

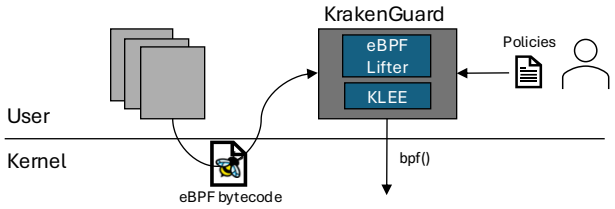

Figure 2: KRAKENGUARD architecture. Unprivileged processes (gray boxes) communicate with the privileged eBPF manager (KRAKENGUARD) over a Unix socket. KRAKENGUARD receives policies from policy writers, runs the analysis, and finally loads the program into the kernel.

bridge on one server and use another server as a client. In the other case, we run the same program attached to the host interface directly. Figure 1 shows the normalized throughput of the eBPF-based UDP echo server during a 60-second window with 1000-byte payloads for the two experiment configurations. We observe that the throughput of the echo server decreases by more than 40% when attached to the virtual interface. This example motivates the need for a solution that enables containers to achieve native performance when deploying `XDP` programs, yet without forgoing the benefits of traffic isolation.

3 Design

Acknowledging the security risks and the potential performance degradation of running eBPF in a multi-tenant environment, we design KRAKENGUARD, a tool that enables an eBPF-as-a-Service model by providing the necessary infrastructure to securely isolate different eBPF programs coming from different tenants, e.g., unprivileged processes or containers. Instead of trying to enforce isolation based on hardware mechanisms, KRAKENGUARD depends on a set of security policies that describe what an eBPF program is or is not allowed to do and, through static analysis, ensures that the program abides by that policy at load time. Using the same mechanisms, KRAKENGUARD also goes beyond a single program and implements a cross-program analysis that detects interference among them, *e.g.,* programs accessing the same memory, thus offering a holistic defense-in-depth solution.

Such an approach is only possible due to two unique characteristics of eBPF and how it is used inside the kernel. First, eBPF programs that satisfy the in-kernel verification process are amenable to further and more robust static analysis. As a result, we speculate that one can exhaustively enumerate all paths and all possible behavior for any input of a particular eBPF program. Second, given eBPF's execution model, one can fully enumerate and capture an eBPF program's interactions with the rest of the system. Programs are attached to specific hooks, manage well-defined state, and interact through maps. Enumerating all possible ways in which the program interacts with the system enables KRAKENGUARD to apply policies that confine or prevent these interactions with other programs, the kernel, or userspace processes.

There are four different ways an eBPF program interacts with its environment, and KRAKENGUARD can impose constraints on all of them. Those are: (i) eBPF helpers, (ii) maps, (iii) return values, and (iv) input arguments. The input arguments correspond to kernel memory that the BPF program can read from and write to. Any other way the program tries to access system resources is considered illegal and will be flagged.

3.1 High-level Design

Figure 2 gives a high-level overview of KRAKENGUARD's architecture. KRAKENGUARD runs as a high-privilege daemon and operates as a trusted eBPF manager. Unprivileged processes or containers that want to load eBPF programs communicate with KRAKENGUARD over a Unix or TCP socket and they send over the eBPF program they want to load as eBPF bytecode. KRAKENGUARD runs an analysis that checks the program against the set of policies provided by the system administrator or the users themselves. Given that KRAKENGUARD works as a centralized manager of all programs running on a host, it can also check the program against other programs loaded on the same machine to detect interference. If the analysis indicates that the program is safe to load, *i.e.,* it does not violate the supplied policies or conflict with other programs, KRAKENGUARD loads it in the kernel.

Internally, KRAKENGUARD depends on a symbolic execution engine. Depending on the type of the eBPF program, KRAKENGUARD creates a symbolic input representing all potential values and through exhaustive symbolic execution, it exercises all potential paths of the program. For example, network-related programs, such as `XDP` or `TC`, take a packet as input whose headers and payload will be symbolic, *i.e.,* they only take concrete values if they affect the program's control flow. This way, KRAKENGUARD can reason separately for each execution path and check if it violates the provided policies. KRAKENGUARD assumes that the programs it attempts to load already satisfy the in-kernel verifier, hence exhaus-

tive symbolic execution is likely possible. If that is not the case and the analysis does not terminate after a fixed amount of time, KRAKENGUARD refuses to load the program and returns a negative response.

3.2 Single-Program Policies

KRAKENGUARD allows for different types of policies that constrain the behavior of a program. We show how this is achieved through exhaustive symbolic execution and the benefits of this approach.

Restricting Helper Functions: Policies can define exactly which helpers an eBPF program is allowed to use. During symbolic execution, KRAKENGUARD keeps track of which helper functions are called given the specific symbolic input and reports an error whenever restricted helpers are used. Although it is possible to detect helper function calls through simpler static analysis of eBPF source code, approaches that do not use symbolic execution may be imprecise and prone to false negatives, potentially discarding programs that contain unreachable calls to these functions. For instance, a helper may appear in a branch that is never executed. This can be common as eBPF libraries grow in size and complexity [1,15].

Restricting Return Values: Depending on the program type, the return value of an eBPF program can be quite powerful. For example, `XDP` programs can drop or transmit packets based on that, i.e., `XDP_DROP` or `XDP_TX`. As a result, KRAKENGUARD also tracks and can apply policies on the return value by specifying all allowed or restricted return values. During its analysis, KRAKENGUARD keeps track of all the return values across different execution paths, and if any of them is not permitted by the target policy, rejects the program.

Restricting Map Accesses: KRAKENGUARD policies also define the set of maps, identified by name, and the access rights for each of them that the eBPF program is allowed to use. The program can have Read, Write, or ReadWrite permissions. During the analysis, the symbolic execution engine initializes the maps in memory and tracks the memory allocated to each map. When the program tries to perform any memory operation, i.e., load or store, on that memory range, KRAKENGUARD checks the constraints and if they are not satisfied, it reports a violation. By doing so, KRAKENGUARD does not depend on identifying helper functions accessing maps, and can also handle pointer-based map accesses.

Restricting Memory Access: KRAKENGUARD can accurately track and restrict an eBPF program's memory accesses, which is a key differentiating factor compared to prior work. This is useful both to catch malicious programs that try to exploit bugs in the in-kernel verifier and to apply the principle of least privilege on eBPF programs. KRAKENGUARD accepts eBPF programs that only access the memory necessary for their functionality and nothing more. For example, a monitoring program that is attached to the networking stack should not be allowed to modify network packets.

KRAKENGUARD assumes that the memory an eBPF program is allowed to access includes only maps and the kernel memory reachable as the program input, and it enables defining a precise subset of that memory to be accessible during program execution. We already discussed how to constrain map accesses. To constrain accesses to the program input, KRAKENGUARD uses relative memory ranges and associates access rights, *i.e.,* Read and/or Write, for each of these ranges as a part of its policy. Using exhaustive symbolic execution, KRAKENGUARD checks any memory access a program is able to perform across all viable execution paths and ensures that this access falls within the allowed ranges with the correct access rights. If that is not the case, KRAKENGUARD aborts its execution and rejects the eBPF program.

To simplify the creation of meaningful and easy-to-express security policies, KRAKENGUARD allows the use of context-aware rules. For example, for network-related programs, rather than writing complicated and error-prone rules based on byte ranges, policy writers can use specific packet header fields to define their policies, such as the source and destination IP or port. Under the hood, KRAKENGUARD translates those rules into byte ranges. Currently, KRAKENGUARD supports these constraints only for network programs that have access to the packet data and cover only certain fields of the packet. This can be easily extended to cover other header fields, but also other kinds of programs.

We note that eBPF programs access memory either directly based on their logic or indirectly via the use of helpers. Many exploits leverage various kernel functions and eBPF helper functions to access out-of-bounds kernel memory. CVE-2022-23222 [10] which exploits `bpf_ringbuf_output()` and CVE-2021-4204 [9] which exploits `bpf_ringbuf_submit()` and `bpf_ringbuf_discard()` are such examples. KRAKENGUARD's policies cover holistically any memory access, irrespective of where it comes from. To achieve this, KRAKENGUARD implements models for the eBPF helpers and kernel functions used by the program. Given that KRAKENGUARD assumes that the only reachable memory is maps and program input, it does not support helpers that directly access other kernel memory.

Conditional policies: KRAKENGUARD can also enforce all the previously described policies conditionally instead of globally across the entire program. The conditions depend either on previous actions or memory content. For example, an execution path of a program that accesses a particular map, *e.g.,* a map holding secrets, should not be allowed to modify packet contents and send a packet out as this could lead to se-

cret leaks. Another example with conditional policies on input content is the case of an XDP program belonging to a service listening to a specific TCP port. This program should only be able to access and modify certain parts of incoming packets if and only if these packets have the said destination port. Program actions that KRAKENGUARD uses as conditionals are map and memory accesses, and helper calls. Conditional policies enable KRAKENGUARD to implement a much more fine-grained analysis and express policies that are applied at an execution path granularity, which is a unique feature among all other available eBPF access control mechanisms.

3.3 Cross-Program Interference

Despite the per-program security policies that constrain what a program can do, there still might be cases where programs are not completely isolated. This might either be because the policies are incomplete, have bugs, or do not wish to restrict a certain functionality that leads to interference with other collocated programs. We consider that two programs interfere if they access the same key on shared eBPF maps and/or the same bytes of kernel memory, *i.e.,* through their input. This interference might be expected and wanted, *e.g.,* when deploying a chain of network functions (NFs) with each NF being a separate eBPF program, or unwanted, *e.g.,* XDP programs belonging to different containers should not have access to the same traffic.

KRAKENGUARD can detect such interference between programs loaded on the same host by using the same mechanism of tracking map and memory accesses. It tracks the read and write sets for each program and each path and checks for overlaps in those sets. KRAKENGUARD can optionally check for interference only after ensuring that the program complies with the provided security policies.

4 Implementation

We build KRAKENGUARD on top of KLEE [53], a symbolic execution engine for LLVM. To support eBPF, we use and extend `ebpf-se` [45, 63] to provide KLEE with symbolic implementations of eBPF helpers and maps. KRAKENGUARD includes 5529 lines of C++ and 2651 lines of Python: 3267 LOC for the eBPF-to-LLVM-IR lifter and related LLVM passes, 373 LOC for new `ebpf-se` models, 1889 LOC for KLEE modifications.

4.1 KRAKENGUARD Infrastructure

Controller: KRAKENGUARD depends on a Python controller that receives user requests to load eBPF programs and interfaces with the KRAKENGUARD engine that performs the analysis. The current controller is a stateless thin layer over the single-program and interference analysis, which receives

```
{
  "check_http_traffic": {
    "type": "MEM_CONDITION",
    "memory_condition": { "dPort": 80 }
  },
  "process_http_packet": {
    "type": "ACTION",
    "dependencies": {
      "memory": ["check_http_traffic"],
      "previous_actions": [],
      "dependency_logic": "OR"
    },
    "actions": {
      "memory_access": [
        {
          "read-access": ["0-54"],
          "write-access": ["42-54"]
        }
      ],
      "map_access": [],
      "helper_access": [],
      "return_value": []
    }
  }
}
```

Listing 1: Simple Policy Example: HTTP Traffic on Port 80

programs along with their policy files over a Unix socket, performs the analysis, and if successful, loads the program. This decoupling enables the controller to be developed and evolve independently, hence adding new features, such as caching analysis results or maintaining a database of all loaded programs, does not require changes to the analysis engine.

Policy language: KRAKENGUARD uses a JSON-based domain-specific language (DSL) to define policies. The DSL defines a set of rules. Each rule includes the *actions* the program is allowed to take, *i.e.,* access ranges of memory and maps with specific access rights, call helper functions, and return values. Each rule can also define a set of conditionals. The conditionals can be either on the memory content at this point in execution or on previous actions taken. In the presence of multiple conditionals, KRAKENGUARD can join them either with a logical `AND` or a logical `OR`. For simplicity and code reuse, the DSL allows defining and naming memory conditionals separately. Listing 1 describes the structure of a sample KRAKENGUARD policy file. This file gives specific read and write access to the program input if the destination port is 80, does not allow any helper or map access since the equivalent sets are empty, and does not constrain the program's return value. Concrete examples of rules can be seen in §5 and Appendix A. Also, Appendix B includes a full description of the policy language and the rule structure.

Once KRAKENGUARD receives the policy file, it parses

and translates the rule set into a graph to use during symbolic execution. Each node in the graph is a rule and edges describe action dependencies. Rules that have conditionals on previous actions point to the rules that define those actions. Thus, global rules, *i.e.,* ones without any dependencies, only have incoming edges. During that step, KRAKENGUARD performs a rule sanity check to identify potential rule conflicts or misconstructions.

Bytecode pre-processing: KRAKENGUARD accepts eBPF programs in ELF format, thus it does not require access to source code giving users the freedom to choose any programming language to implement their programs. Since KLEE operates on LLVM IR, though, KRAKENGUARD has to first lift the eBPF bytecode to LLVM IR. The lifter uses the LLVM API [26] and relies on `libbpf` to extract bytecode, program names, map definitions, and helper function names. It also parses relocation information from the ELF file to support correct handling of maps and helpers. To ensure its correctness, we validated the lifter by comparing the generated IR against the IR produced by the LLVM toolchain given the source code across a set of eBPF programs (Table 3).

KRAKENGUARD relies on an executor program to analyze the eBPF program with KLEE. The executor creates the symbolic eBPF program input, which varies based on the eBPF program type, initializes the required maps, and calls the main program function. Because each eBPF program differs in its inputs and used maps, the executor must be customized accordingly. To automate this, we use the `jinja2` [20] template engine, which uses metadata from the eBPF lifter to produce the final customized executor program. This program is finally compiled to LLVM IR.

Before invoking KLEE, the executor IR and the lifted IR of the eBPF program must be combined. This requires several transformations: making functions externally visible, adding external declarations for helper functions, and resolving input types and function names to ensure compatibility at link time. We implement two LLVM passes to perform these transformations at the function and module levels. Finally, KRAKENGUARD links the transformed IR files using `llvm-link` [27] and runs the resulting IR directly in KLEE. Figure 3 provides an overview of KRAKENGUARD processing the input ELF file so it can be finally analyzed with KLEE. It describes the two IR parts and how KRAKENGUARD generates them, the helper LLVM passes, and how they are linked together.

eBPF Helpers: KRAKENGUARD aims to track policy violations that also come from eBPF helpers. To address this, KRAKENGUARD employs symbolic models for eBPF helpers that imitate their memory access patterns. We implemented these models by carefully inspecting the kernel implementation. Depending on the helper complexity, the exact kernel code could be used; here, we opted for model implementations for the use cases we explored.

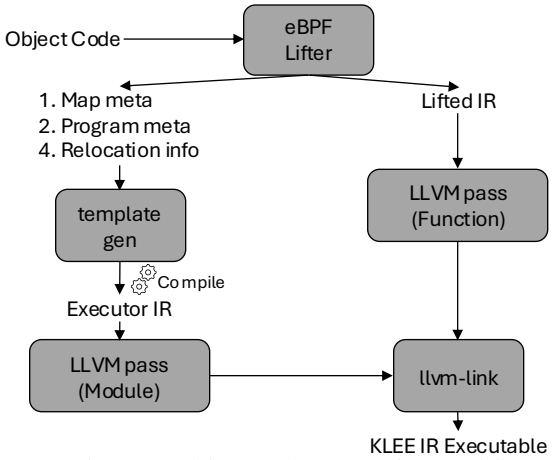

Figure 3: Object code preparation workflow.

4.2 KRAKENGUARD Analysis

Single Program Analysis: KRAKENGUARD leverages two KLEE data structures to perform its single-program analysis. KLEE manages each allocated memory as a set of unique `MemoryObjects` [23], one for each memory allocation. Whenever there is a load or store on a particular address, KLEE translates this access to an *offset* in a `MemoryObject`. KRAKENGUARD maps the ranges defined in the memory access policies into ranges of `MemoryObjects`. KLEE also uses an `ExecutionState` object [22] to track metadata associated with a specific execution path. KRAKENGUARD extends this object to keep the set of *actions* a program has taken up until this point to be used during conditional policy checking. This list includes the names of rules that allowed the actions.

KRAKENGUARD needs to evaluate its set of rules every time the program makes a memory access (*e.g.,* to a map or program input), calls an eBPF helper, or returns a value. We first describe the evaluation of rules without dependencies on previous program actions. The check of helper functions and return values is straightforward, as KRAKENGUARD searches for a rule that allows this particular helper or value. To check memory accesses, KRAKENGUARD translates the memory ranges and their access rights into SMT constraints and appends those to the existing set of constraints KLEE maintains and asks the solver to ensure that the memory access is allowed. KRAKENGUARD also translates the conditionals on memory content into SMT formulas and validates those in cases the rules have such a dependency before deciding if the rule applies. If no rule is found, the action is invalid and the analysis terminates. If there are matching rules, KRAKENGUARD adds the names of these rules in the list of taken actions maintained in the `ExecutionState` object.

For rules that have dependencies on previous actions, KRAKENGUARD first searches the list of taken actions. If the policy joins the dependencies with `AND`, KRAKENGUARD has to find all the names to proceed. Otherwise, finding a single

name is enough. If the dependencies are satisfied, *i.e.,* the described action has been taken, KRAKENGUARD proceeds with validating the rule as described in the previous paragraph. If the rule matches, it also adds that rule in the list of taken actions and proceeds with the analysis.

Cross-Program Analysis: KRAKENGUARD uses the same mechanism that intercepts all memory operations to detect cross-program interference. It tracks and compares the read and write sets of memory locations for each pair of programs. Naively doing so, *i.e.,* collecting a set across all execution paths in a program, can lead to false positives. Instead, KRAKENGUARD compares the read and write sets for each execution path independently by analyzing programs in sequence, one after another, separated by a separator function which marks the boundary between two programs. For each execution path, KRAKENGUARD keeps the memory accesses of each program independently and then compares them for overlapping accesses. Note that the cross-program analysis works for pairs of programs. Thus, the controller can call this analysis multiple times if it wants to check potential conflicts among a set of different programs.

```
{
  "ekubelet-constraints": {
    "type": "ACTION",
    "dependencies": {
      "memory": [],
      "previous_actions": []
    },
    "actions": {
      "map_access": [
        { "name": "raw_tracepoint_map
            ", "access": "ReadWrite" }
      ],
      "helper_access": ["
          bpf_get_current_comm", "
          bpf_probe_read_str", "
          bpf_probe_read", "
          bpf_map_lookup_elem", "
          bpf_map_update_elem", "
          bpf_trace_printk"]
    }
  }
}
```

Listing 2: ekubelet-leak constraint file.

5 Evaluation

This section evaluates KRAKENGUARD and demonstrates its effectiveness in addressing real-world threats and isolating existing eBPF programs. Specifically, we show:

- how KRAKENGUARD can prevent existing attacks and CVEs including bugs in the in-kernel verifier (§5.1,§5.2);
- how KRAKENGUARD constrains programs that perform network packet manipulations (§5.3);
- what the memory overhead and execution time of using KRAKENGUARD are for real-world eBPF programs (§5.4);
- how to use KRAKENGUARD to implement eBPF-as-a-Service for XDP programs (§5.5).

All experiments are conducted on a `B16als_v2` Azure virtual machine with 16 vCPUs, 32 GiB of RAM, and a 64 GiB NVMe SSD, running Ubuntu 24.04 LTS (kernel version 6.17.0-1008-azure).

5.1 Catching Offensive eBPF Features

To demonstrate how KRAKENGUARD can prevent offensive eBPF behavior and CVEs by restricting access to helper functions, we focus on CVE-2022-42150 [11].

First, we examine the `ekubelet-leak.c` program. As described in Section 2.3, this program attaches to the `__x64_sys_read` system call and executes each time `read()` returns. It can inject a malicious Kubernetes Pod specification, potentially granting root-level access to an attacker. The attack relies on two key helpers: `bpf_override_return` to modify the syscall's return value, and `bpf_probe_write_user` to inject the malicious payload. KRAKENGUARD blocks access to both helpers by precisely describing the list of allowed helpers and not including those. Listing 2 shows the equivalent policy file allowing only specific helpers and maps. It takes 0.26 seconds and 85.316 MiB of memory to check the 9 different execution paths for the target helpers.

Second, we consider the `rop.bpf.c` offensive program. The program is an eBPF raw tracepoint handler that hijacks the `read()` system call in a target process. It extracts the system call arguments from the register state, writes the string `"/bin/sh"` into user memory, and scans the process's `libc` for ROP gadgets, such as `pop rdi`, `ret`, and `syscall`. Using the discovered gadget addresses, it crafts a ROP chain directly onto the user stack by calling `bpf_probe_write_user`, setting up arguments and system call numbers so that, upon system call return, the process pivots to the ROP chain and executes `execve("/bin/sh", NULL, NULL)`, effectively spawning a shell in the context of the victim process. With a similar policy file, we restrict the program from using the `bpf_probe_write_user` helper function. The program takes 0.32 seconds and 95.540 MiB of memory to explore the 14 unique execution paths.

5.2 Catching Illegal Memory Accesses

We now show that KRAKENGUARD is effective in catching malicious eBPF programs that attempt to make out-of-bounds kernel memory accesses by exploiting bugs in the kernel verifier and helper functions. We focus on three proof-of-concept (POC) implementations for CVE-2022-23222 [10],

CVE-2021-4204 [9] and CVE-2020-8835 [8]. The POC implementations consist of a combination of user-space code and malicious eBPF bytecode. The user-space program loads the bytecode and interacts with the eBPF program to trigger the attack. Before using KRAKENGUARD, we verified that each CVE was effective by loading the corresponding eBPF program into the kernel versions affected by that CVE. CVE-2022-23222 affects Linux kernel versions up to 5.15, CVE-2021-4204 affects Linux kernel versions up to 5.8.0, and CVE-2020-8835 affects versions from 5.5.0 to 5.6.1.

We then tried to load the same eBPF program through KRAKENGUARD. KRAKENGUARD successfully detected the out-of-bounds memory access and reported the exact location of the error. Note that for CVE-2022-23222 and CVE-2021-4204 the illegal memory access happened via a helper, and KRAKENGUARD was still able to catch it. For CVE-2022-23222 [10], the analysis took 0.22 seconds and used 75.465 megabytes of memory. Those numbers were 0.21 seconds and 73.398 megabytes, and 0.21 seconds and 68.773 megabytes for CVE-2021-4204 [9] and CVE-2020-8835 [8] respectively.

For comparison, we also used Prevail [57], a state-of-the-art eBPF verifier which operates in user space, to attempt to detect these CVEs. Prevail successfully detected that the program for CVE-2020-8835 was unsafe. However, it was unable to verify the program for CVE-2022-23222 and CVE-2021-4204 due to its use of a `BPF_MAP_TYPE_RINGBUF` map, which is not supported in Prevail.

5.3 Constraining Packet Processing Programs

To evaluate KRAKENGUARD's ability to enforce more complex policies, we focus on packet-processing eBPF programs and specifically, Electrode [84] and Katran [46]. Katran is Meta's high-performance Layer 4 load-balancing data plane built on eBPF, while Electrode uses eBPF to accelerate Multi-Paxos. We configure KRAKENGUARD to allow Electrode to read the first 118 bytes of a packet, covering the header and part of the payload containing operation metadata, and conditional write access to the same only when the UDP destination port is 12345, which is the port that Electrode uses. The application-level payload remains protected from modification. We also specify the list of maps along with their access rights and helpers the program is allowed to access. In our evaluation, KRAKENGUARD analyzed 21 execution paths for the FAST_REPLY mode [84] in 0.46 seconds using 93.113 MiB of memory, and 77 execution paths for the FAST_QUORUM_PRUNE mode [84] in 28.30 seconds using 218.184 MiB of memory.

Similarly, we configure KRAKENGUARD to restrict Katran to reading and writing only the first 512 bytes of packet data, regardless of destination port or IP address. Since Katran is designed to support multiple services, we do not impose constraints on destination ports, but we limit access to the payload, as the full message content is not relevant to its operation. Similarly, we defined the exact maps and helpers the program is allowed to access after inspecting the code. KRAKENGUARD analyzes 109 execution paths for Katran in 0.98 seconds, using 91.547 MiB of memory. You can find the complete policy files used for both applications in Appendix A.

5.4 Performance Evaluation

We evaluate KRAKENGUARD's performance using a variety of eBPF programs from academic papers and Linux kernel samples [24]. Table 3 presents the total runtime, maximum memory usage, number of unique paths explored, lines of code in the original eBPF program, and the number of instructions for each program, along with the KRAKENGUARD features used during the single-program analysis. We observe that all of the programs take less than 30 seconds to analyze, with the majority of them taking less than 1 second, while the memory requirements are less than a hundred megabytes.

We further explored the aspects that affect KRAKENGUARD's execution time. We varied the complexity of policies by enabling and disabling constraints (H,M,P features in that table) but we did not notice any changes in the execution time, explored paths, or memory usage as these are affected by the eBPF only. We further investigated the increased execution time for electrode-katran (FAST_QUORUM_PRUNE) and identified that the SMT queries generated by this program are much more complex. The reason for that is that this program uses a symbolic index to look up in a map, hence the entire map is included in every query.

Table 4 includes the execution time for cross-program analysis. We chose Katran and Electrode (`FAST_QUORUM_PRUNE` and `FAST_REPLY`) to demonstrate a real-world collocation scenario because these are the two most complex programs in our evaluation. Despite their complexity, we see that detecting cross-program interference requires approximately one minute.

5.5 Use Case: XDP as a Service

To show how KRAKENGUARD enables eBPF-as-a-service by enforcing per-program isolation policies and ensuring no interference between collocated programs at the load time, we revisit the use case described in Section 2.4, in which a container needs to run an `XDP`-based echo server. We consider a scenario in which two containers want to do that on the same host. The goal is to run the two `XDP` programs natively on the host for better performance, while KRAKENGUARD guarantees that (i) the program cannot do anything malicious that would otherwise be prevented in a containerized deployment, *e.g.,* send traffic to destinations outside the container's reach, and (ii) there is no interference between the two programs. KRAKENGUARD deals with the first requirement through per-program policies and with the second through cross-program analysis.

Program	Time (s)	Memory (MiB)	Paths	LOC	Instructions	Features
Katran [46]	0.98	91.547	109	4244	53,630	H,M,P
Electrode (FAST_REPLY) [84]	0.46	93.113	21	589	8,696	H,M,P
Electrode (FAST_QUORUM_PRUNE) [84]	28.30	218.184	77	480	12,666	H,M,P
hXDP (firewall) [52]	0.34	81.266	10	686	11,165	H,M,P
Fluvia [47]	0.95	84.281	23	156	148,975	H,M,P
xdp_fwd_kernel [24]	0.28	79.688	5	157	5,545	H,M,P
xdp_tx_iptunnel [24]	0.32	79.488	10	176	8,375	H,M,P
CVE-2022-23222 [10]	0.22	75.465	×	85	×	D
CVE-2020-8835 [8]	0.21	68.773	×	181	×	D
CVE-2021-4204 [9]	0.21	73.398	×	123	×	D
ekubelet-leak.c [11]	0.25	85.316	9	404	15,377	H,D
rop.bpf.c [11]	0.32	95.540	14	380	17,202	H,D

H: Helper Function, M: Map Access, P: Packet Access, D: Malicious program detection
×: Not reported because the analysis stopped when it detected the CVE

Table 3: Performance evaluation of various programs using KRAKENGUARD.

Program Pair	Time (s)	Memory (MiB)	Paths	LOC	Instructions
electrode-katran (FAST_REPLY)	3.67	105.652	1,289	4,813	139,554
electrode-katran (FAST_QUORUM_PRUNE)	62.69	256.363	2,881	4,813	234,416

Table 4: Performance evaluation of cross-program interference analysis using KRAKENGUARD

We use the dispatcher provided by `libxdp` [31] to load multiple XDP programs on the host interface. The dispatcher tries to apply all loaded eBPF programs one after the other and stops when a program returns `XDP_TX`. Given the non-interference analysis, KRAKENGUARD will guarantee that only the appropriate program will modify the packet. We assume that container A runs a service at IP `10.0.0.1` and port 8000, while container B runs a service at IP `10.1.0.2` and port 9000.

We define two policy files that instruct KRAKENGUARD to accept only `XDP` programs that: (i) perform actions only on packets with the desired IP and port values. (ii) conditionally modify only the packet header, given (i). (iii) never drop or redirect any packet and are allowed to return `XDP_TX` only if (i) is satisfied. (iv) have no access to any maps on the system. Appendix A includes the complete policy file. After evaluating the single-program policies, KRAKENGUARD also runs the cross-program analysis to ensure that there is no unwanted interference between the two programs. KRAKENGUARD took 0.897 seconds to process the two programs, of which 0.306 seconds were dedicated to checking each program against its policies, and the rest was used for cross-program analysis.

6 Discussion

Exhaustive Symbolic Execution Limitations: KRAKENGUARD depends on the assumption that passing the in-kernel verifier will allow exhaustive symbolic execution to terminate without suffering from path explosion. Although this was the case for most of the programs we analyzed, we stumbled upon a problematic scenario when analyzing the eBPF programs from bmc_cache [58], which led to KRAKENGUARD not terminating within a few minutes, despite the in-kernel verifier having accepted the program. After a closer study, we were able to track down the problem. One of the `XDP` programs used in bmc_cache hashes parts of a packet payload in eBPF and uses this hash value in conditional branches. This leads to large expressions being processed by the SMT solver, hence the long execution times.

The above case stems from the differences in the static analysis approach the kernel and KRAKENGUARD use. The kernel uses abstract interpretation, a coarse-grained (thus more conservative) approach that does not need to reason about every path of the loop. Yet, KRAKENGUARD requires the precision of symbolic execution to evaluate its policies. Prior work [50, 77] has explored the combination of symbolic execution and abstract interpretation by partially using abstract interpretation to improve the scalability of symbolic execution. We believe such an approach is feasible for KRAKENGUARD and we leave this for future work. In the meantime, given that we only identified one such case, KRAKENGUARD takes a more conservative approach and rejects programs for which the analysis does not terminate within a time limit. In such cases, users can always fall back to the suboptimal-performance deployment where the `XDP` program is attached to the container's virtual interface instead of the host one.

Analysis and load coupling: Currently, KRAKENGUARD runs its analysis when applications ask to load an eBPF program. Yet, the two can be easily decoupled to accelerate loading especially in the case of complex programs. KRAK-

ENGUARD can analyze a priori and cache combinations of popular eBPF programs and policies. Applications could then ask KRAKENGUARD to load registered programs with known policies instead of providing the eBPF byte code as input. Based on the cached outcome, KRAKENGUARD would decide to load the program or reject the request. Such an approach works for single-program policies and KRAKENGUARD would have to perform cross-program analysis if necessary at the time of request.

Kernel compatibility: Despite eBPF's efforts towards compatibility, kernel versions affect the behavior of eBPF programs via changes in the verifier, kernel data structures, and the implementations of helpers. Thus, the model implementations of those helpers KRAKENGUARD uses are kernel-version-specific and might need to change across versions. A KRAKENGUARD instance is specific to a kernel version and checks the compatibility between the target kernel and the one it currently executes on before proceeding with the program analysis and aborts execution if there is a mismatch.

7 Related Work

eBPF Isolation: Acknowledging the potential risk due to bugs in the in-kernel verifier, recent work focuses on further isolating eBPF [62]. MOAT [70] is a system that isolates potentially malicious BPF programs using Intel Memory Protection Keys (MPK). Hive [83] proposes a similar approach for the ARM architecture. Jin et al. [65] propose BeeBox, a security architecture that sandboxes the BPF runtime using Software Fault Isolation (SFI). SandBPF [66] is a software-based kernel isolation technique to enable unprivileged users to safely run eBPF programs in the Linux kernel. SandBPF dynamically sandboxes eBPF programs using binary rewriting to confine all memory accesses within the eBPF sandbox and enforce control flow integrity by limiting eBPF control transfers to only valid call targets. KRAKENGUARD is an architecture-agnostic solution that can apply more elaborate security policies, while avoiding any runtime enforcement.

eBPF Verifier and beyond: Several recent works aim to improve and provide guarantees about eBPF verifier infrastructure. Most of the effort focuses on formally verifying or finding bugs in the in-kernel verifier [75, 76, 78, 79]. Going one step further, Jitterbug [72] is a framework designed to formally verify BPF JIT compilers within the Linux kernel. Prevail [57] is a standalone eBPF verifier running in userspace also using abstract interpretation, similar to the in-kernel verifier. Unlike KRAKENGUARD, Prevail aims at providing the same guarantees as the kernel verifier, *i.e.,* termination and the absence of memory errors, instead of proving application- and use-case-specific properties. KRAKENGUARD is orthogonal to these approaches and assumes an in-kernel verifier will force programmers to write code that is amenable to further analysis. It can serve as a defense-in-depth solution in the presence of bugs in the verifier, but its main focus is the enforcement of fine-grained isolation policies in multi-tenant environments. Going beyond eBPF, Jia et al. [64] propose a new framework using Rust for kernel extensions which leverages Rust's safety guarantees to eliminate the need for the in-kernel verifier.

Network Function Verification: P4V [68], Vera [74], and assert-p4 [55] focus on verifying network functions written in P4. VigNAT [82] and Vigor [81] verify C-based network functions by separating stateful and stateless logic. DRACO [69] focuses on functional correctness verification of network functions written in eBPF. It also uses KLEE and exhaustive symbolic execution for verification, but requires an LLVM-compatible specification or annotations in the eBPF code to work. KRAKENGUARD works directly with eBPF bytecode and focuses on constraining instead of fully verifying the functional correctness of network functions, which is a much more complicated and rigorous process.

Compile-time Isolation: There have been multiple attempts to reduce the dependence on hardware-enforced or dynamically checked isolation through the use of safe languages, *e.g.,* Rust [73]. SingularityOS [49], unlike other OSes, does not enforce process isolation through hardware protection and implements Software Isolated Process (SIP) whose boundaries are established by language safety rules and they are enforced using type checking. Singularity's manifest-based programs resemble KRAKENGUARD's security policies. KRAKENGUARD depends on static analysis to provide confinement and non-interference guarantees of eBPF programs.

8 Conclusion

This paper presents KRAKENGUARD for fine-grained isolation policy enforcement over eBPF programs. KRAKENGUARD works complementary to the kernel verifier and implements a trusted eBPF manager that can load eBPF programs for non-privileged applications. KRAKENGUARD uses exhaustive symbolic execution to restrict the helper functions, maps, and memory an eBPF program accesses, as well as to prove the lack of interference between eBPF programs. We use KRAKENGUARD to analyze different real-world eBPF programs and show how it can detect existing CVEs, restrict offensive eBPF features, and implement an `XDP`-as-a-Service paradigm with the same isolation guarantees as a namespaced deployment, but with much better performance.

Acknowledgments

We would like to thank our shepherd Srinivas Narayana and the anonymous reviewers for their insightful comments. We also thank Cristian Cadar for providing useful feedback on the use of KLEE, and the authors of DRACO and `eBPF-se` for their help with the codebase. This work has been partially supported by an ERC Starting Grant (ID: 101220079).

References

[1] Aya: ebpf library for rust. https://aya-rs.dev/.

[2] Bpf maps. kernel documentation. https://docs.kernel.org/bpf/maps.html.

[3] bpfman: An ebpf manager. https://bpfman.io.

[4] Bpf_prog_type_lsm: ebpf program attached to lsm (linux security module). https://docs.ebpf.io/linux/program-type/BPF_PROG_TYPE_LSM/.

[5] bpftoken. https://lwn.net/Articles/936823/.

[6] capabilities - overview of linux capabilities. https://man7.org/linux/man-pages/man7/capabilities.7.html.

[7] Cilium: ebpf-based networking, observability, security. https://cilium.io/.

[8] Cve-2020-8835: Oob read/write in ebpf due to improper register bound for 32-bit opetations. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2020-8835.

[9] Cve-2021-4204: Oob read/write in ebpf due to improper input validation. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2021-4204.

[10] Cve-2022-23222: Previlege escalation in ebpf by exploiting _or_null. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2022-23222.

[11] Cve-2022-42150: Container escape using ebpf. https://cve.mitre.org/cgi-bin/cvename.cgi?name=2022-42150.

[12] Datadog: Cloud monitoring as a service. https://www.datadoghq.com/.

[13] ebpf case studies. https://ebpf.io/case-studies/.

[14] ebpf: extended berkeley packet filter. https://ebpf.io/.

[15] ebpf-go: A pure-go library to read, modify and load ebpf programs and attach them to various hooks in the linux kernel. https://github.com/cilium/ebpf.

[16] ebpf in-kernel verifier. https://docs.kernel.org/bpf/verifier.html.

[17] Excessive container capabilities. https://0xn3va.gitbook.io/cheat-sheets/container/escaping/excessive-capabilities.

[18] Exploiting excessive container capabilities. https://redfoxsec.com/blog/exploiting-excessive-container-capabilities/.

[19] Falco: Detect security threats in real time. https://falco.org/.

[20] Jinja: a fast, expressive, extensible templating engine. https://jinja.palletsprojects.com/en/stable/.

[21] Kaslr and fg-kaslr bypassing. https://lkmidas.github.io/posts/20210205-linux-kernel-pwn-part-3/#about-kaslr-and-fg-kaslr.

[22] Klee executionstate. https://github.com/klee/klee/blob/492800921e47141d509736f97731f2985c09704d/lib/Core/ExecutionState.h#L149.

[23] Klee memoryobject. https://github.com/klee/klee/blob/492800921e47141d509736f97731f2985c09704d/lib/Core/Memory.h#L36.

[24] Linux kernel bpf samples. https://github.com/torvalds/linux/tree/master/samples/bpf.

[25] linux-kernel kconfig definition for cap_bpf. https://elixir.bootlin.com/linux/v6.14.3/source/include/uapi/linux/capability.h#L412.

[26] Llvm and api reference documentation. https://llvm.org/docs/Reference.html.

[27] llvm-link - llvm bitcode linker. https://llvm.org/docs/CommandGuide/llvm-link.html.

[28] Map type bpf_map_type_array. kernel documentation. https://docs.kernel.org/bpf/map_array.html.

[29] Map type bpf_map_type_prog_array. kernel documentation. https://docs.ebpf.io/linux/map-type/BPF_MAP_TYPE_PROG_ARRAY/.

[30] Pointer arithmetic allowed for certain *_or_null type bug. https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/tree/kernel/bpf/verifier.c?h=v5.10.83#n6022.

[31] Protocol for atomic loading of multi-prog dispatchers. https://github.com/xdp-project/xdp-tools/blob/main/lib/libxdp/protocol.org.

[32] Tetragon – ebpf-based security observability runtime enforcement. https://github.com/cilium/tetragon.

[33] Xdp: Express data path. https://docs.ebpf.io/linux/program-type/BPF_PROG_TYPE_XDP/.

[34] Container tidbits: Adding capabilities to a container. https://www.redhat.com/en/blog/container-tidbits-adding-capabilities-container, 2016.

[35] ebpf implementation for freebsd. https://papers.freebsd.org/2018/bsdcan/hayakawa-ebpf_implementation_for_freebsd/, 2018.

[36] Unimog - cloudflare's edge load balancer. https://blog.cloudflare.com/unimog-cloudflares-edge-load-balancer/, 2020.

[37] Making ebpf work on windows. https://opensource.microsoft.com/blog/2021/05/10/making-ebpf-work-on-windows/, 2021.

[38] Mapping it out: Analyzing the security of ebpf maps. https://www.crowdstrike.com/en-us/blog/analyzing-the-security-of-ebpf-maps/, 2021.

[39] A proof-carrying approach to building correct and flexible in-kernel verifiers. https://homes.cs.washington.edu/~lukenels/slides/2021-09-23-lpc21.pdf, 2021.

[40] Alex murray - unprivileged ebpf disabled by default for ubuntu 20.04 lts, 18.04 lts, 16.04 esm. https://discourse.ubuntu.com/t/unprivileged-ebpf-disabled-by-default-for-ubuntu-20-04-lts-18-04-lts-16-04-esm/27047, 2022.

[41] Suse support - security hardening: Use of ebpf by unprivileged users has been disabled by default. https://www.suse.com/support/kb/doc/?id=000020545, 2022.

[42] ebpf explained: Why it's important for observability. https://www.kentik.com/blog/ebpf-explained-why-its-important-for-observability/, 2023.

[43] Next-generation observability with ebpf. https://isovalent.com/blog/post/next-generation-observability-with-ebpf/, 2023.

[44] Understanding the ebpf networking features. https://docs.redhat.com/en/documentation/red_hat_enterprise_linux/8/html/configuring_and_managing_networking/assembly_understanding-the-ebpf-features-in-rhel-8_configuring-and-managing-networking, 2023.

[45] Epfl dependable systems laboratory. 2024. ebpf-se source code respository. https://github.com/dslab-epfl/ebpf-se, 2024.

[46] Meta 2024. katran source code respository. https://github.com/facebookincubator/katran, 2024.

[47] Ntt communications. 2024. fluvia source code respository. https://github.com/nttcom/fluvia/, 2024.

[48] The state of ebpf. https://www.linuxfoundation.org/hubfs/LF%20Research/The_State_of_eBPF_010824.pdf, 2024.

[49] Mark Aiken, Manuel Fähndrich, Chris Hawblitzel, Galen Hunt, and James Larus. Deconstructing process isolation. In *Proceedings of the 2006 Workshop on Memory System Performance and Correctness*, MSPC '06, page 1–10, New York, NY, USA, 2006. Association for Computing Machinery.

[50] Eman Alatawi, Harald Søndergaard, and Tim Miller. Leveraging abstract interpretation for efficient dynamic symbolic execution. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*, ASE '17, page 619–624. IEEE Press, 2017.

[51] Theophilus A. Benson, Prashanth Kannan, Prankur Gupta, Balasubramanian Madhavan, Kumar Saurabh Arora, Jie Meng, Martin Lau, Abhishek Dhamija, Rajiv Krishnamurthy, Srikanth Sundaresan, Neil Spring, and Ying Zhang. Netedit: An orchestration platform for ebpf network functions at scale. In *Proceedings of the ACM SIGCOMM 2024 Conference*, ACM SIGCOMM '24, page 721–734, New York, NY, USA, 2024. Association for Computing Machinery.

[52] Marco Spaziani Brunella, Giacomo Belocchi, Marco Bonola, Salvatore Pontarelli, Giuseppe Siracusano, Giuseppe Bianchi, Aniello Cammarano, Alessandro Palumbo, Luca Petrucci, and Roberto Bifulco. hXDP: Efficient software packet processing on FPGA NICs. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 973–990. USENIX Association, November 2020.

[53] Cristian Cadar, Daniel Dunbar, and Dawson Engler. Klee: unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, page 209–224, USA, 2008. USENIX Association.

[54] Neha Chowdhary, Utkalika Satapathy, Theophilus Benson, Subhrendu Chattopadhyay, Palani Kodeswaran, Sayandeep Sen, and Sandip Chakraborty. Beeguard: Explainability-based policy enforcement of ebpf codes for cloud-native environments. In *2025 17th International Conference on COMmunication Systems and NETworks (COMSNETS)*, pages 757–765, 2025.

[55] Lucas Freire, Miguel Neves, Lucas Leal, Kirill Levchenko, Alberto Schaeffer-Filho, and Marinho Barcellos. Uncovering bugs in p4 programs with assertion-based verification. In *Proceedings of the Symposium on SDN Research*, pages 1–7, 2018.

[56] Elazar Gershuni, Nadav Amit, Arie Gurfinkel, Nina Narodytska, Jorge A. Navas, Noam Rinetzky, Leonid Ryzhyk, and Mooly Sagiv. Simple and precise static analysis of untrusted linux kernel extensions. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2019, page 1069–1084, New York, NY, USA, 2019. Association for Computing Machinery.

[57] Elazar Gershuni, Nadav Amit, Arie Gurfinkel, Nina Narodytska, Jorge A. Navas, Noam Rinetzky, Leonid Ryzhyk, and Mooly Sagiv. Simple and precise static analysis of untrusted linux kernel extensions. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2019, page 1069–1084, New York, NY, USA, 2019. Association for Computing Machinery.

[58] Yoann Ghigoff, Julien Sopena, Kahina Lazri, Antoine Blin, and Gilles Muller. BMC: Accelerating memcached using safe in-kernel caching and pre-stack processing. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, pages 487–501. USENIX Association, April 2021.

[59] Yi He, Roland Guo, Yunlong Xing, Xijia Che, Kun Sun, Zhuotao Liu, Ke Xu, and Qi Li. Cross container attacks: The bewildered eBPF on clouds. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 5971–5988, Anaheim, CA, August 2023. USENIX Association.

[60] Gernot Heiser. The sel4 microkernel. *The seL4 Foundation*, 1, 2020.

[61] Toke Høiland-Jørgensen, Jesper Dangaard Brouer, Daniel Borkmann, John Fastabend, Tom Herbert, David Ahern, and David Miller. The express data path: fast programmable packet processing in the operating system kernel. In *Proceedings of the 14th International Conference on Emerging Networking EXperiments and Technologies*, CoNEXT '18, page 54–66, New York, NY, USA, 2018. Association for Computing Machinery.

[62] Kaiming Huang, Mathias Payer, Zhiyun Qian, Jack Sampson, Gang Tan, and Trent Jaeger. Sok: Challenges and paths toward memory safety for ebpf. In *SP*, pages 848–866. IEEE, 2025.

[63] Rishabh Iyer, Katerina Argyraki, and George Candea. Performance interfaces for network functions. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 567–584, Renton, WA, April 2022. USENIX Association.

[64] Jinghao Jia, Raj Sahu, Adam Oswald, Dan Williams, Michael V. Le, and Tianyin Xu. Kernel extension verification is untenable. In *Proceedings of the 19th Workshop on Hot Topics in Operating Systems*, HOTOS '23, page 150–157, New York, NY, USA, 2023. Association for Computing Machinery.

[65] Di Jin, Alexander J. Gaidis, and Vasileios P. Kemerlis. BeeBox: Hardening BPF against transient execution attacks. In *33rd USENIX Security Symposium (USENIX Security 24)*, pages 613–630, Philadelphia, PA, August 2024. USENIX Association.

[66] Soo Yee Lim, Xueyuan Han, and Thomas Pasquier. Unleashing unprivileged ebpf potential with dynamic sandboxing. In *Proceedings of the 1st Workshop on EBPF and Kernel Extensions*, eBPF '23, page 42–48, New York, NY, USA, 2023. Association for Computing Machinery.

[67] Linux Foundation. net/ipv4/bpf_tcp_ca.c in the linux kernel source tree. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/net/ipv4/bpf_tcp_ca.c?h=v6.9, 2024.

[68] Jed Liu, William Hallahan, Cole Schlesinger, Milad Sharif, Jeongkeun Lee, Robert Soulé, Han Wang, Călin Caşcaval, Nick McKeown, and Nate Foster. P4v: Practical verification for programmable data planes. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on data communication*, pages 490–503, 2018.

[69] Dana Lu, Boxuan Tang, Michael Paper, and Marios Kogias. Towards functional verification of ebpf programs. In *Proceedings of the ACM SIGCOMM 2024 Workshop on EBPF and Kernel Extensions*, eBPF '24, page 37–43, New York, NY, USA, 2024. Association for Computing Machinery.

[70] Hongyi Lu, Shuai Wang, Yechang Wu, Wanning He, and Fengwei Zhang. Moat: towards safe bpf kernel extension. In *Proceedings of the 33rd USENIX Conference on Security Symposium*, SEC '24, USA, 2024. USENIX Association.

[71] Michael Marty, Marc de Kruijf, Jacob Adriaens, Christopher Alfeld, Sean Bauer, Carlo Contavalli, Michael Dalton, Nandita Dukkipati, William C. Evans, Steve Gribble, Nicholas Kidd, Roman Kononov, Gautam Kumar, Carl Mauer, Emily Musick, Lena Olson, Erik Rubow, Michael Ryan, Kevin Springborn, Paul Turner, Valas Valancius, Xi Wang, and Amin Vahdat. Snap: a microkernel approach to host networking. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, page 399–413, New York, NY, USA, 2019. Association for Computing Machinery.

[72] Luke Nelson, Jacob Van Geffen, Emina Torlak, and Xi Wang. Specification and verification in the field: Applying formal methods to BPF just-in-time compilers in the linux kernel. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 41–61. USENIX Association, November 2020.

[73] Aurojit Panda, Sangjin Han, Keon Jang, Melvin Walls, Sylvia Ratnasamy, and Scott Shenker. {NetBricks}: Taking the v out of {NFV}. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 203–216, 2016.

[74] Radu Stoenescu, Dragos Dumitrescu, Matei Popovici, Lorina Negreanu, and Costin Raiciu. Debugging p4 programs with vera. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, pages 518–532, 2018.

[75] Hao Sun and Zhendong Su. Validating the {eBPF} verifier via state embedding. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pages 615–628, 2024.

[76] Hao Sun, Yiru Xu, Jianzhong Liu, Yuheng Shen, Nan Guan, and Yu Jiang. Finding correctness bugs in ebpf verifier with structured and sanitized program. In *Proceedings of the Nineteenth European Conference on Computer Systems*, EuroSys '24, page 689–703, New York, NY, USA, 2024. Association for Computing Machinery.

[77] Ignacio Tiraboschi, Tamara Rezk, and Xavier Rival. Sound symbolic execution via abstract interpretation and its application to security. In *VMCAI*, volume 13881 of *Lecture Notes in Computer Science*, pages 267–295. Springer, 2023.

[78] Harishankar Vishwanathan, Matan Shachnai, Srinivas Narayana, and Santosh Nagarakatte. Verifying the verifier: ebpf range analysis verification. In *International Conference on Computer Aided Verification*, pages 226–251. Springer, 2023.

[79] Xiwei Wu, Yueyang Feng, Tianyi Huang, Xiaoyang Lu, Shengkai Lin, Lihan Xie, Shizhen Zhao, and Qinxiang Cao. {VEP}: A two-stage verification toolchain for full {eBPF} programmability. In *22nd USENIX Symposium on Networked Systems Design and Implementation (NSDI 25)*, pages 277–299, 2025.

[80] Rui Yang and Marios Kogias. Heels: A host-enabled ebpf-based load balancing scheme. In *Proceedings of the 1st Workshop on EBPF and Kernel Extensions*, eBPF '23, page 77–83, New York, NY, USA, 2023. Association for Computing Machinery.

[81] Arseniy Zaostrovnykh, Solal Pirelli, Rishabh Iyer, Matteo Rizzo, Luis Pedrosa, Katerina Argyraki, and George Candea. Verifying software network functions with no verification expertise. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 275–290, 2019.

[82] Arseniy Zaostrovnykh, Solal Pirelli, Luis Pedrosa, Katerina Argyraki, and George Candea. A formally verified nat. In *Proceedings of the conference of the acm special interest group on data communication*, pages 141–154, 2017.

[83] Peihua Zhang, Chenggang Wu, Xiangyu Meng, Yinqian Zhang, Mingfan Peng, Shiyang Zhang, Bing Hu, Mengyao Xie, Yuanming Lai, Yan Kang, and Zhe Wang. HIVE: A hardware-assisted isolated execution environment for eBPF on AArch64. In *33rd USENIX Security Symposium (USENIX Security 24)*, pages 163–180, Philadelphia, PA, August 2024. USENIX Association.

[84] Yang Zhou, Zezhou Wang, Sowmya Dharanipragada, and Minlan Yu. Electrode: Accelerating distributed protocols with eBPF. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 1391–1407, Boston, MA, April 2023. USENIX Association.

A Complete Policy File Example

In this appendix, we include the complete policy files used in KRAKENGUARD's evaluation.

```
{
  "check_udp_port_12345": {
    "type": "MEM_CONDITION",
    "memory_condition": {
      "dPort": 12345
    }
  },
  "allow_packet_read_write": {
    "type": "ACTION",
    "dependencies": {
      "memory": ["check_udp_port_12345"],
      "previous_actions": []
    },
    "actions": {
      "memory_access": [
        {
          "read-access": ["0-117"],
          "write-access": ["0-117"]
        }
      ],
      "map_access": [
        { "name": "map_configure", "access": "Read" },
        { "name": "map_ctr_state", "access": "ReadWrite" },
        { "name": "map_msg_lastOp", "access": "ReadWrite" },
        { "name": "map_quorum", "access": "ReadWrite" },
        { "name": "batch_context", "access": "ReadWrite" },
        { "name": "map_progs_xdp", "access": "Read" },
        { "name": "map_prepare_buffer", "access": "Write" },
        { "name": "map_request_buffer", "access": "Write" }
      ],
      "helper_access": ["bpf_map_lookup_elem", "bpf_map_update_elem", "bpf_tail_call", "
          bpf_ringbuf_reserve", "bpf_ringbuf_submit", "bpf_xdp_adjust_head", "
          bpf_xdp_adjust_tail", "bpf_spin_lock", "bpf_spin_unlock"],
      "return_value": [
        1, // XDP_DROP
        2, // XDP_PASS
        3 // XDP_TX
      ]
    }
  },
  "only_read_if_not_12345" : {
    "type": "ACTION",
    "dependencies": {
      "memory": ["!check_udp_port_12345"],
      "previous_actions": []
    },
    "actions": {
      "memory_access": [
        {
          "read-access": ["0-117"],
          "write-access": ["x"]
        }
      ],
      "return_value": [
        2 // XDP_PASS
```

```
      ]
    }
  }
}
```

Listing 3: Electrode Constraint File used in §5.3

```
{
  "global_policy": {
    "type": "ACTION",
    "dependencies": {
      "memory": [],
      "previous_actions": []
    },
    "actions": {
      "memory_access": [
        {
          "read-access": ["0-511"],
          "write-access": ["0-511"]
        }
      ],
      "map_access": [
        { "name": "ch_rings", "access": "Read" },
        { "name": "reals", "access": "Read" },
        { "name": "quic_mapping", "access": "Read" },
        { "name": "ctl_array", "access": "Read" },
        { "name": "vip_map", "access": "ReadWrite" },
        { "name": "stats", "access": "ReadWrite" },
        { "name": "reals_stats", "access": "ReadWrite" },
        { "name": "fallback_cache", "access": "ReadWrite" },
        { "name": "lru_mapping", "access": "ReadWrite" },
        { "name": "subprograms", "access": "Read" },
        { "name": "pckt_srcs", "access": "Read" }
      ],
      "helper_access": [
        "bpf_map_lookup_elem",
        "bpf_map_update_elem",
        "bpf_csum_diff",
        "bpf_ktime_get_ns",
        "bpf_xdp_adjust_head",
        "bpf_xdp_adjust_tail",
        "bpf_tail_call",
        "bpf_fib_lookup",
        "bpf_trace_printk"
      ],
      "return_value": [
        1, // XDP_DROP
        2, // XDP_PASS
        3 // XDP_TX
      ]
    }
  }
}
```

Listing 4: Katran Constraint File used in §5.3

```
{
  "check_ip_10_0_0_1_port_8000": {
    "type": "MEM_CONDITION",
```

```
    "memory_condition": {
      "dIP": "10.0.0.1",
      "dPort": 16415
    }
  },
  "allow_echo_server_access": {
    "type": "ACTION",
    "dependencies": {
      "memory": ["check_ip_10_0_0_1_port_8000"],
      "previous_actions": []
    },
    "actions": {
      "memory_access": [
        {
          "read-access": ["0-42"],
          "write-access": ["0-42"]
        }
      ],
      "helper_access": [
        "bpf_ntohs"
      ],
      "return_value": [
        3
      ]
    }
  },
  "deny_other_traffic": {
    "type": "ACTION",
    "dependencies": {
      "memory": ["!check_ip_10_0_0_1_port_8000"],
      "previous_actions": []
    },
    "actions": {
      "memory_access": [
        {
          "read-access": ["0-42"],
          "write-access": ["x"]
        }
      ],
      "return_value": [
        2
      ]
    }
  }
}
```

Listing 5: eBPF-as-a-Service Policy File for XDP Echo Server (IP: 10.0.0.1, Port: 8000) used in §5.5

B Policy Structure Reference

Table 5 describes the fields available in the KRAKENGUARD policy JSON format.

Field	Data Type	Required	Applies To	Default Behavior	Description
`type`	String	Yes	All	—	Specifies the policy node type: `MEM_CONDITION` or `ACTION`.
`memory_condition`	Object	Yes	MEM_CONDITION	—	Defines packet field match criteria either using field identifiers (e.g., `dPort`, `sIP`) or memory ranges.
`dependencies`	Object	Yes	ACTION	—	Contains `memory`, `previous_actions`, and `dependency_logic` fields.
`memory`	String[]	Yes	ACTION	—	List of memory condition names that must be satisfied. The names come from `dependencies`. Negations using ! are supported.
`previous_actions`	String[]	Yes	ACTION	—	List of action names that must have been executed previously (inside `dependencies`).
`dependency_logic`	String	No	ACTION	OR	Logical operator for combining previous action dependencies: `AND` or `OR` (inside `dependencies`).
`actions`	Object	Yes	ACTION	—	Container object that describes actions via `memory_access`, `map_access`, `helper_access`, `return_value`.
`memory_access`	Object[]	No	ACTION	All allowed	Whitelist of packet memory ranges with `read-access` and `write-access` (inside `actions`). If not present, the entire input is accessible.
`map_access`	Object[]	No	ACTION	None allowed	Whitelist of eBPF map names with access types: `Read`, `Write` or `ReadWrite` (inside `actions`).
`helper_access`	String[]	No	ACTION	None allowed	Whitelist of allowed eBPF helper function names (inside `actions`).
`return_value`	Int[]	No	ACTION	All allowed	Whitelist of allowed program return values (e.g., `[1, 2, 3]` for XDP actions) (inside `actions`).

Table 5: Structure of the KRAKENGUARD policy JSON fields.