BeeGuard: Explainability-based Policy Enforcement of eBPF Codes for Cloud-native Environments

Neha Chowdhary
IIT Kharagpur, India
nehachow.cse@kgpian.iitkgp.ac.in

Utkalika Satapathy
IIT Kharagpur, India
utkalika.satapathy01@gmail.com

Theophilus Benson
Carnegie Mellon University, USA
theophib@andrew.cmu.edu

Subhrendu Chattopadhyay
IDRBT, Hyderabad, India
subhrendu@idrbt.ac.in

Palani Kodeswaran
IBM-IRL, Bangalore, India
palani.kodeswaran@in.ibm.com

Sayandeep Sen
IBM-IRL, Bangalore, India
sayandes@in.ibm.com

Sandip Chakraborty
IIT Kharagpur, India
sandipc@cse.iitkgp.ac.in

***Abstract*—eBPF enables loading user space code into the kernel, thereby extending the kernel functionalities in an application-aware manner. This flexibility has led to the widespread adoption of the technology across hyperscalers and enterprises for several use cases, including observability, security, network policy enforcement, etc. In general, the safety of the loaded eBPF programs are ensured through a kernel verifier that performs different syntactic/structural checks to secure the kernel against unwanted crashes. In this paper, we motivate the case that this verifier-based security check, while necessary, is insufficient to ensure that the deployed eBPF code complies with organizational policies. Consequently, we propose *BeeGuard*, a framework to understand program behavior to extract capability lists from eBPF programs. *BeeGuard* introduces a policy compliance layer on top of the existing verifier, and the extracted capability lists of a program are then checked against organizational policies to allow or block loading the programs. Thorough experiments across the most popular open-source eBPF tools show that *BeeGuard* can enforce typical enterprise policies with minimal overhead in terms of loading latency and resource utilization.**

***Index Terms*—eBPF, observability, kernel verifier, policy compliance**

I. Introduction

Observability and efficiency improvement of large-scale distributed systems (such as Serverless platforms [1], [2], hyperscalers [3], content delivery networks (CDNs) [4], etc.) has gained a lot of attention in recent literature [5]–[7]. Among these, several solutions have utilized *Extended Berkeley Packet Filters* (eBPF) [8] to design fine-grained monitoring platforms [9] for cloud-native applications deployed over such large-scale systems. The advantage of eBPF is that it allows developers to introduce custom functionalities into the kernel transparently, which improves efficiency [10]–[12] and flexibility [13], ranging from security, observability [9], [14]–[17], network profiling [18] and policy enforcement [19]–[23], etc. eBPF source codes are compiled into a *machine-independent intermediate representation* (also known as bytecodes) that is further compiled with a Just-in-Time (JIT) compiler at runtime to achieve platform independence. In addition, each eBPF program passes through an in-kernel verifier, which ensures that the program does not initiate kernel panic. Furthermore, eBPF utilizes in-kernel sandboxing, which restricts the eBPF program from directly modifying kernel data structures, thus enhancing the platform's security. To access the kernel information, each eBPF program is associated with triggering events (hook points) that initiate the program's execution. Additionally, eBPF programs use specialized data structures (known as "maps") to enable data exchange across multiple eBPF and userspace programs.

Despite these advantages, using the eBPF eco-system leads to compliance concerns for large regulated organizations. For example, let us consider the scenario where a banking and financial organization utilizing eBPF programs also needs to ensure policy compliance strictly. The task becomes challenging for compliance officers of the banking organization who have minimal understanding of the eBPF eco-system. The primary hurdles are as follows.

① eBPF programs deployed inside kernel enjoy greater privileges, and therefore, runtime risk analysis for the deployed eBPF programs is non-trivial. Given a set of organizational policies, it requires technically skilled auditors with in-depth knowledge of the eBPF eco-system to check if the deployed eBPF programs are actually in compliance with the organizational policies. Notably, reverse engineering of eBPF bytecodes [24] has been proposed, which requires skilled reverse engineers to understand the program behavior.

② Policies for kernel-level programs should be implemented inside the kernel. However, in large-scale organizations, policies are dynamic and change quite often, which is difficult if the policies are directly implemented inside the kernel.

③ Implementing source control has been challenging for eBPF programs. Existing works have relied on the hypervisors [25] and customized device drivers [26] to check the sanity of the deployed eBPF programs. However, hypervisors and device drivers need an additional level of redirection, thus significantly increasing the program load time. On the other hand, implementing a signature scheme to ensure source control requires trusted user space applications [27].

To address the above challenges, in this paper, we propose a framework called *BeeGuard*, which develops an observability framework for the eBPF programs during runtime to allow/deny the programs based on compliance with the

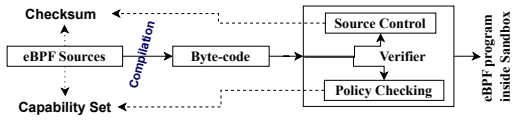


Fig. 1: Conceptual view of *BeeGuard*

organizational policies. We hypothesize that selecting a suitable behavioral model of an eBPF program which can even be understandable by the policy makers/compliance officers having little or no understanding of eBPF construct and accurately capture the key behaviors without instrumenting the source code can address the first challenge mentioned above. Notably, the same behavioral model can also be utilized to represent the policies. To address the second challenge, we propose separating policy management from policy enforcement so that policy compliance becomes easier for naive users. For example, a global policy store where policies can be loaded dynamically and a policy enforcement framework that can continuously police the system is required. Finally, to address the third challenge, we propose an in-kernel source control primitive to ensure the non-compromization of eBPF programs from a trustworthy source without relying on any third-party/user-space application.

The core of our proposed solution works as follows. To express the behavioral model, we use a novel approach of utilizing the "*capability set*" of an eBPF program, which is defined as a list of used hook points, along with maps and header functions used to develop the eBPF program. We use a natural language processing (NLP)-enabled code analysis module, which can extract the capabilities of an eBPF program from its source code. As shown in Figure 1, the extracted capabilities are used by the in-kernel verifier, which checks if the program complies with the organizational policies expressed in terms of capabilities and if the source is from a legitimate source (via source-control primitive). Our implementation requires minor kernel-level modification ($\approx$ 160 LOC) for the in-kernel verifier, which makes it easy to incorporate into mainline kernel releases. Moreover, the other components of *BeeGuard* are deployed as containerized microservices, making it scalable and easy to manage.

We have tested *BeeGuard* using 15 publicly available popular eBPF programs (totaling $\approx$ 500 lines of codes) used for cloud-native development. The experimental results suggest minimal overhead during the program load time ($\approx$ 3ms), very minimal overhead in CPU and memory utilization ($< 1\%$) during program loading, and zero overhead during the eBPF execution time. We have also observed that the enforcement framework can block eBPF programs that do not satisfy the policy specifications. Furthermore, qualitative experiments indicate that the modification of the policies requires minimal effort, while the implementation overhead of the modified policy is negligible ($<$ 10ms). Our qualitative analysis shows that the proposed framework can effectively enforce policy compliance for the deployed eBPF programs in a scalable, robust, and efficient way. In summary, the contributions of this paper are as follows.

1) We develop a novel framework for in-kernel processing of organizational policies and use it during the eBPF program loading to enable or disable a program from loading over the runtime environment.
2) We utilize the program capabilities extracted from the eBPF programs to develop a novel in-kernel verifier extension to check whether the program capabilities match the organization's policies, thus enforcing policy compatibility during eBPF runtime.

II. System Architecture

Trust Model: In designing *BeeGuard*, we make the following trust assumptions. We assume the kernel and eBPF verifier are trusted and un-compromised entities. We also assume that the attacker can compromise any client device and alter the eBPF program sources using a compromised user-space program. For this reason, we can not trust the eBPF loader programs as the loader programs execute at the user space.

System Overview: We design *BeeGuard* as a centralized controller-based architecture to provide flexibility in policy management using two main components: (a) Global eBPF Sentinel (G-sent), and (b) Kernel-space (KSpace) as depicted in Figure 2.

Here, the G-sent is a central component executing inside a secured environment (such as a demilitarized zone (DMZ) [28]), responsible for the overall policy management. The DMZ-based deployment makes G-sent immune to the compromised eBPF programs. On the other hand, the "KSpace" component is part of the eBPF runtime environment and is executed on each device where the eBPF programs are supposed to be deployed. The "KSpace" is responsible for policy enforcement. Following the standard practice, the eBPF source codes are compiled to the bytecode at the user space (USpace) of each device (except DMZ) where they are supposed to be deployed using the runtime environment as shown in Figure 2. Using an untrusted loader eBPF can be loaded into the kernel which internally invokes the eBPF verifier. At this time the "policy fetcher" module at the "KSpace" checks the compliance of the eBPF program with the help of G-sent before actually allowing the eBPF program to execute. The implementation details of *BeeGuard* are as follows.

A. Global eBPF Sentinel (G-sent)

The *Global eBPF Sentinel* (G-sent) provides a single point of management that offers a dashboard to allow easy interaction between the system administrator and *BeeGuard*. This dashboard can be used to introduce eBPF programs into the system. Since this component also stores the organizational policies, it provides ease of augmenting the policies as per the requirements. Although implementation of this module in the kernel space may improve security, it significantly reduces the ease of policy modification. The policies are represented as a set of allowed and blocked *"capabilities"*. In this context, we define a capability as a "sub-action"

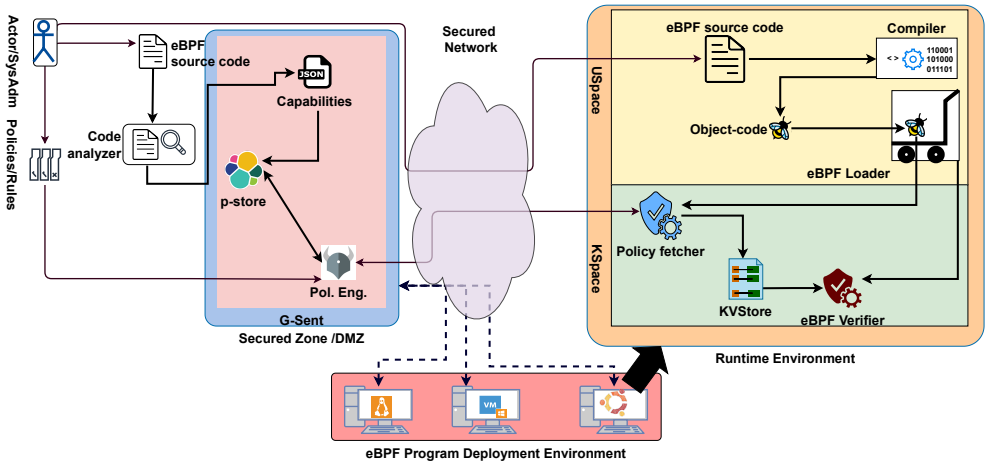


Fig. 2: Proposed architecture of *BeeGuard*

which can be performed by a program. For example, packet header manipulation, system tracing, packet cloning, etc. can be considered capability labels. We assume that an `eBPF` program is represented as a set of capability labels and stored inside the "*Persistent Capability Store*" (p-store) sub-module of `G-sent` (See Section II-C for proof of concept). The stored capabilities are referred to by the "*Policy Engine*" sub-module to ascertain the compliance of the program as per the organizational policies.

1) p-store: The rationale behind using the capabilities from each `eBPF` program is that the set of capabilities together represents the behavioral model of the `eBPF` programs. "p-store" works as a persistent store for the behavioral model so that the policy compliance checking can be expedited by the use of an index to query the capabilities for each function of the `eBPF` program. For our implementation purpose, we have used the *Elasticsearch database* [29], which is a scalable, fast, lightweight, fault-tolerant database. We have used containerized deployment of Elasticsearch to ease the deployment process. The capabilities stored in "p-store" are frequently consulted by the "*Policy Engine*", as discussed next.

2) Policy Engine: The objective of the "*Policy Engine*" is to ascertain if the invoked `eBPF` program violates the system-wide policy specifications. Suppose at least one capability is incompatible with the system, then the "*Policy engine*" can decide not to allow such an `eBPF` program. In this paper, we have primarily focused on the restrictions that should be enforced related to `eBPF` programs. To ease the policy management, we have used Open Policy Agent (OPA) [30], which can store and manage the policies in the form of codes and provides a platform to control the loading or rejection of `eBPF` programs as per the organizational policy. Although the list of policies can be extended significantly, we have implemented three different types of policies for experimental purposes to allow programs with specific capabilities from a trusted source and not having certain capabilities (more details in Section III-B). The decisions taken by the `G-sent` are enforced by the rest of the two components.

B. Kernel-space Component (`KSpace`)

The policy enforcement process for each `eBPF` program is carried out by the `KSpace` component, which controls the permission of the target `eBPF` program, either allowing or blocking its execution. This placement of the enforcement framework in the kernel space is strategic, providing enhanced security and resilience against modifications compared to user-space implementation. In the interaction between the "G-sent" and "KSpace", we have introduced a "Policy Fetcher" module. The overall sequence of execution is as follows.

The `eBPF` source code, once compiled to generate equivalent bytecode, is loaded via a user-space loader program. During the loading, the bytecode is submitted to the in-kernel verifier. The verifier interacts with the `G-sent` module via "Policy Fetcher", which is a customized Loadable Kernel Module (LKM) and is responsible for fetching decisions about the target `eBPF` program and storing it inside the kernel space enforcement component securely. Additionally, the loader program loads the `eBPF` program into the memory and hands it over to the kernel-space "`eBPF` verifier". Since "Policy Fetcher" is developed as an LKM, it can be secured via traditional signature-based mechanisms. The communication between `G-Sent` and the "Policy fetcher" is assumed to be secured using SSL-based encryption.

Additionally, $\approx$ 160 lines of code were added inside the kernel to modify the `BPF_PROG_LOAD` system call. This particular system call is used to load an `eBPF` program and leverages the `bpf_attr` structure to store attributes of the program such as program type, instruction count, etc. as input arguments. This system call is also responsible for the allocation of maps and verification of the `eBPF` program subsequently using the `BPF_CHECK` system call.

Our proposed enforcement logic validates the authenticity of the program, fetches the corresponding *policy decision*, and enforces the decision before sending the `eBPF` program for verification. This proposed code modification is independent of the platform or versions and written with the intent to be

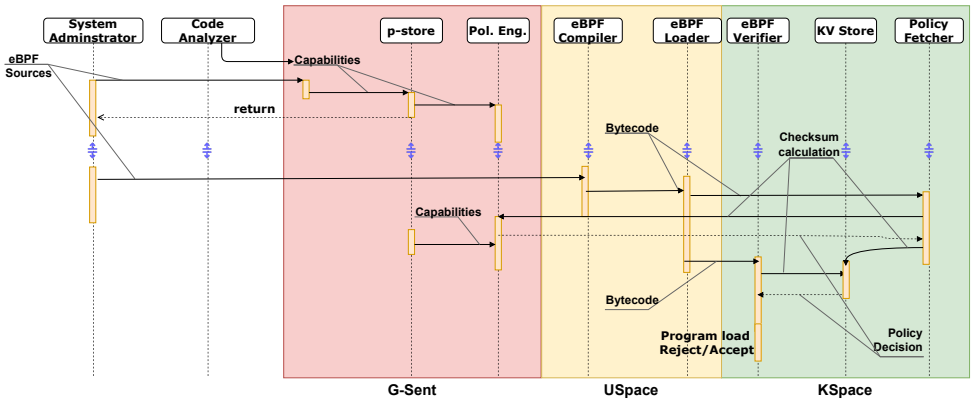


Fig. 3: Sequence diagram for loading of an eBPF program

officially incorporated in mainstream kernel versions in due course. The augmented code performs two specific tasks; (a) checks the program identity, and (b) compares it with the allowed capabilities provided by G-sent.

To calculate the unique identity (ID) we have implemented an in-kernel check-sum-based signature calculation function. The obtained ID is cross-referenced with an in-kernel local Key-Value cache ("KV Store"). The "KV Store" keeps the program ID as a key and the decision variable (allow/block) as a value. The entries in the "KV Store" are updated by the "Policy Fetcher". The "KV Store" avoids performing costly policy fetching operations from the G-sent server each time the programs are loaded. The ID can also be leveraged as signatures to implement the source control mechanism. Based on the decision variable inside the "KV Store" the system call determines whether to proceed by forwarding the eBPF program to the verifier block or to return an error code. The overall interaction between these components is depicted in Figure 3.

During the implementation, we faced the following two major challenges; (a) checksum computation for an eBPF program, (b) interaction with the "G-sent". While attempts have been made to extend the source control mechanism [31] of LKM to the eBPF subsystem, none have been accepted by the community so far due to the two-phase compilation process of eBPF. During the JIT compilation of eBPF programs, the bytecodes are optimized using code relocation techniques and then converted into machine code. This process changes the calculated checksum at the time of loading. Another difficulty in checksum calculation during the program verification is due to the size restriction of the eBPF programs. To avoid the restriction, each eBPF program is subdivided into multiple subroutines before verification. To overcome this challenge, we have utilized eBPF's Instruction Set Architecture (ISA) [32] and devised our custom checksum mechanism from the bpf_insn() structure which can be generated in both userspace and kernel space. Once the checksum is calculated, it is used to query the G-sent about the authenticity and policy compliance of the program. However, consulting G-sent each time before loading a program becomes time-consuming; therefore, we have used the local "KV Store".

C. Code Analyzer

So far, we have assumed that the behavioral model of an eBPF program is available for perusal, which can be achieved in multiple ways through static analysis [33], [34]. It has been observed that traditional approaches like binary analysis [35] provide significant information about the structure of the program but are not suitable for predicting output from the program; hence, modern static analysis approaches rely on neural machine translation (NMT) [36]. Therefore, we have relied on a similar approach to design our *Code Analyzer* module in this work. This module aims to find the list of capability labels of each eBPF program. These capability labels of the program specify the set of actions, for example, manipulating the IP header, adding a VLAN header, dropping packets, etc., that the program could potentially perform at run time. These capability labels are fed into the policy enforcement engine and used when authoring enterprise policies.

The proposed *Code analyzer* uses the insight that the eBPF programs can *only use* well-defined helper functions to read and manipulate the system states. Consequently, we have focused on extracting and leveraging information about eBPF helper functions for extracting the capability labels of the program. To generate the capability labels, the code analyzer leverages a combination of (a) *static code analysis* and (b) *semantic understanding* to understand program/functional intent from an enterprise perspective. When a new eBPF program is submitted to the system, the *Code Analyzer* runs a static analysis pass on the source code to extract the program features such as the set of eBPF maps that each function reads/writes, eBPF helper functions per function calls, etc. to identify program functionality and potential developer intent. To extract semantic capabilities, *BeeGuard* employs a domain-specific NLP pipeline. The tool is pre-trained with a custom-built dataset consisting of a human-annotated dataset[1]. The NLP pipeline leverages a combination of sources such as

[1]https://github.com/eBPFDevSecTools/annotations (Accessed: November 12, 2024)

helper function man pages, program comments, human annotations of eBPF codes, etc. Specifically, our NLP pipeline can identify action words from man pages of the helper functions and programmer comments. This set of actions is then filtered using a curated list of enterprise-sensitive action words that are of typical importance from a policy perspective, such as packet manipulation, packet dropping, system tracing, etc. Additionally, we leverage the comments around the eBPF map definitions to identify potential semantic data types such as IP header, MAC address, cgroup ID, etc., stored in the map.

The combination of map-related helper functions (obtained from static analysis) along with the map data types (inferred from comments) to associate additional capability labels. For instance, a program that *writes* to a map and stores the *source IP address* is assigned the capability label *store_header*. Furthermore, we have created a human annotation dataset describing the functionality of each eBPF function in natural language. Our pipeline extracts action words from these human annotations to extract higher-level capability labels, such as load balancer, port forwarder, firewall, etc., that describe how the eBPF programs are leveraged in enterprise deployments. Finally, the function call graphs are generated from syntactic analysis to assign each function's union of capability labels for the entire eBPF program. An example of capability identification is given in Figure 4.

```
"capabilities": [
    "bpf_get_current_pid_tgid",
    "bpf_get_current_comm"
]
```

Listing (1) Capabilities generated from do_perf_event()

```
"capabilities": [
    "bpf_trace_printk"
]
```

Listing (2) Capabilities generated from print_arg()

```
"capabilities": [
    "bpf_get_current_pid_tgid",
    "bpf_get_current_comm",
    "bpf_trace_printk"
]
```

Listing (3) Capabilities generated from eBPF program consisting of do_perf_event() and print_arg() both

Fig. 4: Representation of generated capabilities labels for an eBPF program.

As shown in Figure 4, let us consider an eBPF program that monitors the performance of various user programs. The program is composed of two functions, namely do_perf_event() and print_arg(). Listing 1 presents the capability labels generated for the function do_perf_event() which is primarily attributed to the use of two helper functions: (a) bpf_get_current_pid_tgid (To obtain the process and thread group id) and (b) bpf_get_current_comm (Obtains command name of current process). Similarly Listing 2 shows the helper function bpf_trace_printk from the function print_arg() which prints formatted output to the */sys/kernel/tracing/trace* file. The overall capability label of the entire eBPF program is depicted in Listing 3, which is a union of the functions mentioned above and represents an overall behavioral profile for the program.

TABLE I: Distribution of extracted capabilities across studied open-source eBPF repositories

Program Capability	Number of functions
pkt_stop_processing_drop	30
pkt_goto_next_module	120
read_sys_info	38
update_pkt	12
read_skb	30
map_read	100
map_update	20

The capabilities of the extracted program are stored in an eBPF registry that provides an online catalog of eBPF functions, extracted capabilities, and meta-data describing their functionality, requirements, and constraints. Table I shows a subset of capabilities extracted from eBPF programs studied in this paper, along with the frequency of their use in our aforementioned open-sourced database. Describing the internal details of the generation of code comments is beyond the scope of this paper and hence omitted for the sake of brevity.

III. Evaluation

The objective of the evaluation is to understand the efficacy, efficiency, and overhead of *BeeGuard* in correctly loading an eBPF program while maintaining policy compliance. We use a server with 2 Intel Xeon Platinum 8358 CPUs and $512GB$ memory for experimental evaluation of *BeeGuard*. We have configured 2 VMs, one for the G-Sent and the other as a client device. Both the VMs have 12 VCPUs and 32GB memory with Linux Kernel 6.1.38 with eBPF support using the bcc [37] framework. The communication latency between the VMs $\approx$ 1ms. During our evaluation, we have used 15 different eBPF programs from open-source repositories [38] as listed in Table II. The programs can be categorized into 3 broad categories based on their functionality.

Kernel Function Tracing: Table IIa contains a list of programs for tracing kernel functions and monitoring system behavior, which are particularly useful for debugging purposes. For example, stacksnoop can capture kernel function's stack traces and help diagnose bottlenecks in system call paths. Additionally, these programs can perform several other diagnostic operations such as measurement of the length of strings called by the *strlen()* function call to monitor string handling (e.g., strlen_count), analyze process creation by adding kprobes on the *clone* system call (e.g., trace_perf), etc. The detailed objectives of these programs are also listed in the table.

Performance Profiling: Table IIb emphasizes the profiling of performance metrics and latency analysis. For example, I/O latency distributions can be measured using biolatpcts, which can help to understand inefficient workloads. Similarly, for disk access pattern, vfslatency helps with providing

TABLE II: List of eBPF programs used for testing.

(a) Kernel Function Tracing

Name of Program	No. of Instructions	Description
stack_buildid	51	Profiles the call stacks of processes using system libraries
stacksnoop	31	Traces a kernel function and prints all kernel stack traces
strlen_count	43	Traces strlen() and print a frequency count of strings
strlen_hist	71	Histogram of system-wide strlen() return values
trace_fields	15	Traces an event and printing custom fields
trace_perf	43	Traces the 'clone' system call using a kprobe

(b) Performance Profiling

Name of Program	No. of Instructions	Description
biolatpcts	69	Calculates IO latency percentile
hello_perf	25	Hello World example that uses BPF_PERF_OUTPUT
hello_perf_ns	36	Hello World example that uses BPF_PERF_OUTPUT with bpf_get_ns_current_pid_tgid()
sync_timing	40	Trace time between syncs
vfslatency	85	VFS read latency distribution

(c) Network Monitoring

Name of Program	No. of Instructions	Description
dddos	62	Using ePBF to detect a potential DDOS attack against a system
nflatency	120	Attaches a kprobe and kretprobe to nf_hook_slow
tcpv4connect	62	Traces TCP IPv4 connect()s
undump	46	Dumps UNIX socket packets

data on file system performance, and sync_timing can be used to perform several other diagnostic operations such as measurement of time taken during synchronization operations to enhance I/O operations.

Network Monitoring: In the last group, as seen in Table IIc, we have eBPF programs used for network monitoring and security analysis. Where dddos can be used to flag potential Denial of Service attacks, nflatency helps improve network understanding by tracking packet processing latency and allows us to pinpoint performance bottlenecks in the network stack. tcpv4connect can be utilized to monitor TCP connection attempts to detect misuse patterns in connectivity.

A. Quantitative Evaluation

To understand the overhead imposed by the framework, we have compared the performance of *BeeGuard* with an equivalent unmodified kernel (UN). We have compared the performance of two versions of *BeeGuard*: (a) *BeeGuard* without the "KV Store" (BG) and (b) *BeeGuard* with the "KV Store" (BG'). All the experiments were repeated 10 times, and the mean and standard deviation (σ) for various time components are reported in Figure 5. We observe that the proposed framework introduces a small overhead ($\approx$ 3ms) during the loading of each eBPF program. This overhead observed is attributed to two sub-actions: (a) Signature calculation from the submitted bytecode and (b) Policy verification.

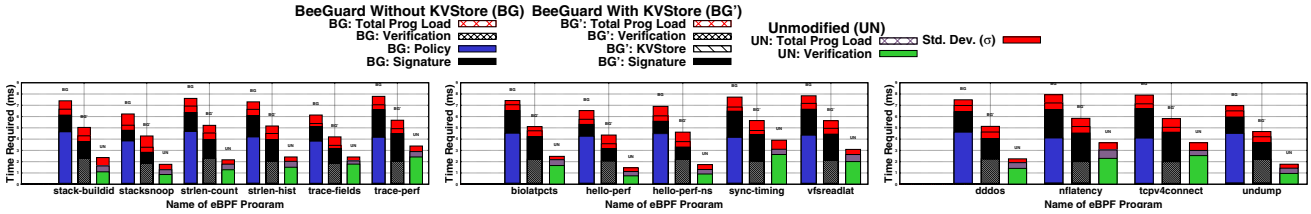
Figure 5 represents the loading time when the policy is calculated in the "KV Store" for the first time. We have observed that while the checksum calculation takes negligible time, the policy verification time at "G-sent" takes significantly longer. As policy compliance for each program can be carried out in parallel, this overhead is not cumulative and is independent of the number of programs executing inside the machine. From the experiments, we found that *BeeGuard* imposes an average overhead of $\approx 4.5 \pm 1$ms for kernel tracing (Figure 5a), and performance profiling (Figure 5b) programs due to added program loading complexity compared to an unmodified kernel. On the other hand, network monitoring programs Figure 5c show an average overhead of $\approx 5 \pm 1$ms. The reason behind the added overhead is that typical network monitoring programs are significantly larger ($\approx 1.5x$) than the other two categories; thus expected to require greater time. We have also observed that the average time to receive a response from the "policy engine" is $\approx 4.8 \pm 1.6$ms for all categories as it is independent of the program categories and size. From the experimental results depicted in Figure 5, we also observe that despite the notably low latency between the policy engine and client devices, the integration of KV Store as a cache can substantially enhance performance by reducing the decision making ($\approx 2ms$ faster). Intuitively, the performance benefit from the KV Store increases if the latency between G-Sent and the client device increases. We have also observed that, due to the signature calculation overhead, the average CPU utilization (see Figure 6) also increased marginally ($< 1\%$) for most of the programs. This increase is most noticeable in network monitoring programs (Figure 6c) and significantly lesser for kernel tracing (Figure 6a) and performance filtering (Figure 6b) programs. This increase in CPU utilization is due to the length of the program. Apart from the enforcement module, we have also measured the resource utilization overhead of the G-Sent micro-services using "docker stat", which reveals (see Figures 7a and 7b) a negligible overhead in terms of CPU and memory footprint.

B. Qualitative Analysis

Although, *BeeGuard* incurs marginal one-time overhead (during program load-time), it offers significant advantages in terms of usability, as discussed below.

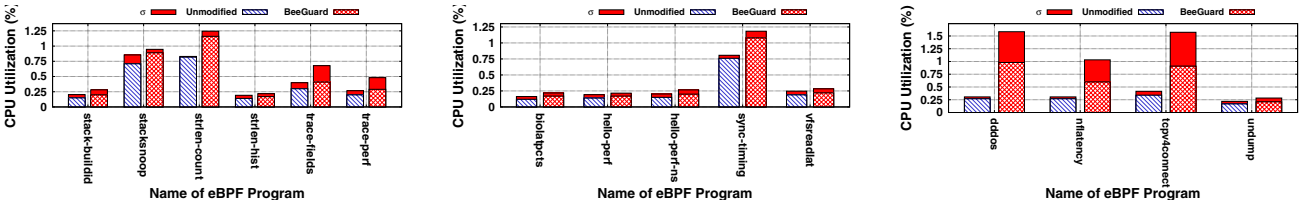
1) Ease of Deployment: All components of *BeeGuard* except the in-kernel enforcer are developed as containers and, thus, are highly scalable and easy to deploy. Moreover, *BeeGuard* can be deployed following the continuous integration/continuous deployment (CI/CD) pipeline.

2) Ease of Policy Modification: Modifying policy in *BeeGuard* is easy due to its use of the policy engine. For example, when a new helper function is added within an eBPF program that is already in the list of predefined policy rules in "p-store", we simply need to update the capability label of the program. A new capability value addition does not require any change in the policy rules, and only a list update in Elastic will serve the purpose. On the other hand, to support the extension of the capability list (i.e., beyond the hook points, tracepoints, and



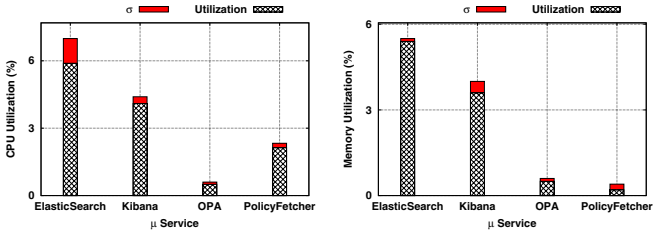
(a) Kernel Function Tracing (b) Performance Profiling (c) Network Monitoring

Fig. 5: Average load time (component-wise) analysis by the three groups of eBPF programs



(a) Kernel Function Tracing (b) Performance Profiling (c) Network Monitoring

Fig. 6: CPU utilization by the three groups of eBPF programs at runtime



(a) CPU utilization **(b) Memory utilization**

Fig. 7: Resource utilization for G-Sent

used helper functions), minor policy rule updation is required ($<$ 10LOC). We have tested these two features by adding a new capability (i.e., eBPF program daisy-chaining using the "pkt_go_to_next_module" helper function) which validates this particular claim.

3) Policy Robustness: Although often used by eBPF programs for monitoring and debugging the system, *tracepoints* can also be used to gain system insights and facilitate attacks. Therefore, the system should leverage the "policy engine" to control the accessibility of the various tracepoints. Additionally, the hookpoint and helper function control also needs to be ensured to avoid open vulnerabilities. In this paper, we have considered three types of policy rules in our "policy engine" to justify the robustness of the framework: (a) block tampered eBPF programs, (b) block side-channeling, and (c) block unrestricted tunneling. We only allow the eBPF bytecodes, which are *untampered* (i.e., the pre-computed signature matches the load time signature). Additionally, we observed that an eBPF program with header parsing capability can increase *side-channel* attack probability [39]. Therefore, any program that uses raw interfaces such as tracepoints is blocked. A rule to block *unrestricted tunneling* [40] has also been developed by restricting header modification, which in turn avoids tunneling-related helper functions. In a nutshell, the last two rules cover the hookpoint and helper function control. The corresponding policy rules are listed in Listing 4. As seen in Line 8 of Listing 4, all capabilities associated with the program are fetched from "p-store" are stored in the list ALL_CAPABILITIES. Two lists (BLKD_TRACEPOINT in Line 12 and BLKD_HELPER in Line 13) are used to store lists of potentially risky tracepoints and helper functions and should be excluded from the allowed list (Lines 15 and 16). We have used two rules to arrive at the policy decision for each eBPF program: (a) setting policy decision as allowed when the program signature is authenticated even though it may have blank capabilities labels; and (b) setting policy decision as allowed when all the capabilities of the program match the ones fetched from "p-store" along with the signature. Furthermore, we have identified all the combinations of these three rules and created customized 8 eBPF programs. For example, we have modified the sources by adding instructions inside the bytecode which should result in a signature mismatch. In a different case, we have added a packet parsing helper function to the strlen_hist to mimic side channeling. We have found that *BeeGuard* only allows legitimate programs to be loaded, and the programs violating policy were blocked.

4) Policy Expressiveness: By modifying suitable policies during the runtime, *BeeGuard* can control the hookpoints or tracepoints and the trusted helper functions. We performed 10 experiments by modifying the policies and observed no deviation from the expected behavior. We have also observed that the policy change takes negligible time ($<$ 10ms) to reflect its effect.

IV. Related Work

A. Existing works using eBPF for profiling

A significant focus of research on eBPF covers the usage of eBPF for profiling or analyzing various aspects of the system

```
### Subroutines
element_belongs_to_list(element, allowed_capabilities) {
    element == allowed_capabilities[_]
}
### Block all programs except explicitly allowed
default allow := false
### Fetch capabilities from p-store
ALL_CAPABILITIES:= sql.send({
    "driver": "sqlite",
    "query": "SELECT capabilities FROM ebpf_registry"
  })
BLKD_TRACEPOINT:=["netif_tx"]
BLKD_HELPER:=["pkt_go_to_next_module", "parse_pkt_headers"]
### List of Allowed capabilities
ALLOWED_CAPABILITIES:=ALL_CAPABILITIES - BLKD_TRACEPOINT
ALLOWED_CAPABILITIES:=ALLOWED_CAPABILITIES - BLKD_HELPER
### Allow programs with blank capability and Matching Signature
allow if{
    count(data._default[input.signature].
     cumulative_capabilities) == 0
}
### Allow programs having allowed capabilities and matching Signature
allow if{
 x =data._default[input.signature].cumulative_capabilities
 every element in x{
  element_belongs_to_list(element,ALLOWED_CAPABILITIES)
 }
}
```

Listing 4: Example of policies used

behavior like performance, networking, etc. Kmon [14] is an early work which proposes an in-kernel monitoring system for microservices which provide various runtime information of containers ranging from latency to performance metrics. eBPF has also been used recently to create application-specific network profiles [18] by using matchers to map processes to their respective applications, thus facilitating identification. From creating packet filtering rules to fit application profiles [10] to profiling the performance of container-based applications [11] or to create custom profiles for containers based on precise system metrics [12], eBPF has been used for system-level profile generation. In *BeeGuard*, we propose the generation of profiles for each eBPF program based on their behavior as inferred from their capability labels.

B. Verifier modifications in the eBPF subsystem

The BPF verifier has been a topic of much interest in academia as seen from recent literature on automated verification to check the soundness of the eBPF verifier for tracking values of its variables [41]. Range analysis research to study the correctness of the range analysis of the verifier for values in its registers [42] and even approaches to identify any illegal behavior in verified code [43] has been explored. However as cited in [44], kernel bugs can also be exploited to bypass the security provided by the verifier thus leading to suggestions to decouple the verification and JIT compilation process from the loader [45]. HyperBee [25] mentions a similar approach to modifying the verification process which requires a malicious program structure database to verify the eBPF programs. Building on this, in *BeeGuard* we exploit the idea of policy enforcement by extending the capacity of the verifier and loader to not just check code for safety but also policy compliance. Unlike, HyperBee, we rely on the helper functions, trace-point/hook-points, etc used in the program to generate the labels.

C. Literature on policy enforcement

Various organizations have always prioritized the need for policy management and with the rapid adaptation of cloud native architecture, security and data privacy have been emerging concerns. KRSI [19] is an architecture which addresses this problem of container runtime security by using eBPF to manage policies across Kubernetes clusters on the fly. Enforcement of policies at runtime for eBPF programs can be improved by modifying the verification process to monitor the runtime cost of programs [20] and laying tighter upper bounds on the runtime. Sauron proposed in [21] explores the idea of rules across multiple pods wherein eBPF is used to monitor cluster control. Similarly, the eBPF-IoT-MUD [22] uses eBPF to enforce manufacturer usage description rules directly in the lower level of the kernel stack. However, unlike [23] which proposes invoking an eBPF program to decide the legitimacy of policies that may not be scalable, in *BeeGuard* we exploit the modified kernel to match and enforce policies. In *BeeGuard* we have experiments to validate that our proposed architecture is independent of the number of programs running.

D. Source control mechanisms for eBPF programs

In contrast to Loadable Kernel Modules (LKMs) which have a signing mechanism that can be verified by the kernel, eBPF subsystem allows any privileged program that passes the verifier checks to load [27], [31]. HyperBee [25] brings forth a signature verifier in it's verifier chain to validate whether the program carries a trusted signature or not. Alternately, in Windows [26] eBPF programs can be compiled as driver images and then consecutively signed using the standard driver signing mechanisms. Since recent literature did not propose any valid source control mechanism, we used the checksum signature verification as proposed in *BeeGuard*.

V. Conclusion and Future Works

This paper proposes a user-friendly and extendable framework for ensuring policy compliance for kernel-level eBPF programs. We observe that the proposed framework is easily extendable (CI/CD compliant) and provides flexibility in terms of usability and future-proofing. This framework has an ingrained capability of extracting the behavioral models from the eBPF sources in terms of capabilities. The current version of the work can be enhanced with an optimized version of local cache access and end-to-end testing to address scalability concerns, which we plan to incorporate in future updates. We also aim to explore the integration of Linux Security Modules (LSM) with the eBPF program to compare the performance benefits with *BeeGuard*. The LSM-based policy enforcement will enhance the current framework's flexibility by conducting security checks at the time of eBPF program loading before entering the verifier.

References

[1] Z. Li, L. Guo, J. Cheng, Q. Chen, B. He, and M. Guo, "The Serverless Computing Survey: A Technical Primer for Design Architecture," *ACM Computing Survey*, vol. 54, no. 10s, pp. 1–34, September 2022.

[2] S. Qi, L. Monis, Z. Zeng, I.-C. Wang, and K. K. Ramakrishnan, "SPRIGHT: High-Performance eBPF-Based Event-Driven, Shared-Memory Processing for Serverless Computing," *IEEE/ACM Transactions on Networking*, vol. 32, no. 3, pp. 2539–2554, 2024.

[3] IBM, "Hyperscaler Cloud Service Providers Top 10," 2021, [accessed 30 September, 2024]. [Online]. Available: https://www.ibm.com/downloads/cas/LYZX6JB5

[4] P. Khan and B. Rajkumar, "A Taxonomy and Survey of Content Delivery Networks," *Grid Computing and Distributed Systems Laboratory, University of Melbourne, Technical Report*, vol. 4, no. 2007, p. 70, 2007.

[5] U. Naseer and T. A. Benson, "Configanator: A Data-driven Approach to Improving CDN Performance," in *USENIX Symposium on Networked Systems Design and Implementation*, 2022, pp. 1135–1158.

[6] A. N. Montanari and L. A. Aguirre, "Observability of Network Systems: A Critical Review of Recent Results," *Journal of Control, Automation and Electrical Systems*, vol. 31, no. 6, pp. 1348–1374, 2020.

[7] M. Usman, S. Ferlin, A. Brunstrom, and J. Taheri, "A Survey on Observability of Distributed Edge & Container-based Microservices," *IEEE Access*, vol. 10, pp. 86 904–86 919, 2022.

[8] eBPF Community, "Extended Berkeley Packet Filters," 2014, [accessed 30 September, 2024]. [Online]. Available: https://ebpf.io/

[9] S. Sundberg, A. Brunstrom, S. Ferlin-Reiter, T. Høiland-Jørgensen, and J. D. Brouer, "Efficient Continuous Latency Monitoring with eBPF," in *Passive and Active Measurement*. Springer Nature, 2023, pp. 191–208.

[10] D. Scholz, D. Raumer, P. Emmerich, A. Kurtz, K. Lesiak, and G. Carle, "Performance Implications of Packet Filtering with Linux eBPF," in *International Teletraffic Congress*, 2018.

[11] C. Cassagnes, L. Trestioreanu, C. Joly, and R. State, "The Rise of eBPF for Non-intrusive Performance Monitoring," in *IEEE/IFIP Network Operations and Management Symposium*, 2020.

[12] J. Levin and T. A. Benson, "ViperProbe: Rethinking Microservice Observability with eBPF," in *IEEE International Conference on Cloud Networking*, 2020.

[13] J. T. Humphries, N. Natu, A. Chaugule, O. Weisse, B. Rhoden, J. Don, L. Rizzo, O. Rombakh, P. Turner, and C. Kozyrakis, "ghOSt: Fast & Flexible User-Space Delegation of Linux Scheduling," in *ACM Symposium on Operating Systems Principles*, 2021, p. 588–604.

[14] T. Weng, W. Yang, G. Yu, P. Chen, J. Cui, and C. Zhang, "Kmon: An In-kernel Transparent Monitoring System for Microservice Systems with eBPF," in *International Workshop on Cloud Intelligence*, 2021.

[15] B. Sharma and D. Nadig, "eBPF-Enhanced Complete Observability Solution for Cloud-native Microservices," in *IEEE International Conference on Communications*, 2024, pp. 1980–1985.

[16] F. J. Bertinatto, D. Arioza, J. C. Nobre, and L. Z. Granville, "Container-Level Auditing in Container Orchestrators with eBPF," in *Advanced Information Networking and Applications*. Springer Nature, 2024, pp. 412–423.

[17] M. Craun, K. Hussain, U. Gautam, Z. Ji, T. Rao, and D. Williams, "Eliminating eBPF Tracing Overhead on Untraced Processes," in *SIGCOMM Workshop on eBPF and Kernel Extensions*. ACM, 2024, pp. 16–22.

[18] L. Wüstrich, M. Schacherbauer, M. Budeus, D. F. von Künßberg, S. Gallenmüller, M.-O. Pahl, and G. Carle, "Network Profiles for Detecting Application-Characteristic Behavior Using Linux eBPF," in *Workshop on eBPF and Kernel Extensions*. ACM, 2023.

[19] S. Gwak, T. P. Doan, and S. Jun, "Container Instrumentation and Enforcement System for Runtime Security of Kubernetes Platform with eBPF," *Intelligent Automation and Soft Computing*, vol. 37, no. 2, pp. 1773–1786, 2023.

[20] R. Sahu and D. Williams, "Enabling BPF Runtime Policies for Better BPF Management," in *Workshop on eBPF and Kernel Extensions*. ACM, 2023, pp. 49–55.

[21] D. Soldani, P. Nahi, H. Bour, S. Jafarizadeh, M. F. Soliman, L. Di Giovanna, F. Monaco, G. Ognibene, and F. Risso, "eBPF: A New Approach to Cloud-Native Observability, Networking and Security for Current (5G) and Future Mobile Networks (6G and Beyond)," *IEEE Access*, vol. 11, pp. 57 174–57 202, 2023.

[22] A. Feraudo, D. A. Popescu, P. Yadav, R. Mortier, and P. Bellavista, "Mitigating IoT Botnet DDoS Attacks through MUD and eBPF based Traffic Filtering," in *International Conference on Distributed Computing and Networking*. ACM, 2024.

[23] J. Jia, M. V. Le, S. Ahmed, D. Williams, and H. Jamjoom, "Practical and Flexible Kernel CFI Enforcement using eBPF," in *Workshop on eBPF and Kernel Extensions*. ACM, 2023, pp. 84–85.

[24] A. Schendel, "Reverse Engineering eBPF Programs: A Deep Dive," https://www.armosec.io/blog/ebpf-reverse-engineering-programs/, 2024, [accessed 30 September, 2024].

[25] Y. Wang, D. Li, and L. Chen, "Seeing the Invisible: Auditing eBPF Programs in Hypervisor with HyperBee," in *Workshop on eBPF and Kernel Extensions*. ACM, 2023.

[26] A. Jowett, "Towards Debuggability and Secure Deployments of eBPF Programs on Windows," https://opensource.microsoft.com/blog/2022/10/25/towards-debuggability-and-secure-deployments-of-ebpf-programs-on-windows/, 2022, [accessed 30 September, 2024].

[27] D. Alden, "Securing BPF Programs Before and After Verification," https://lwn.net/Articles/977394/, 2024, [accessed 30 September, 2024].

[28] Fortinets, "What is DMZ?" https://www.fortinet.com/resources/cyberglossary/what-is-dmz, 2024, [accessed 30 September, 2024].

[29] Elastic, "Elasticsearch: The Official Distributed Search," 2023, [accessed 30 September, 2024]. [Online]. Available: https://www.elastic.co/elasticsearch/

[30] OPA, "Open Policy Agent," 2023, [accessed 30 September, 2024]. [Online]. Available: https://www.openpolicyagent.org/

[31] J. Corbet, "Toward Signed BPF Programs," 2021, [accessed 30 September, 2024]. [Online]. Available: https://lwn.net/Articles/853489/

[32] "BPF Instruction Set Architecture," 2023, [accessed 30 September, 2024]. [Online]. Available: https://www.kernel.org/doc/html/latest/bpf/standardization/instruction-set.html#bpf-instruction-set-architecture-isa

[33] T. H. Le, H. Chen, and M. A. Babar, "Deep Learning for Source Code Modeling and Generation: Models, Applications, and Challenges," *ACM Computing Surveys*, vol. 53, no. 3, pp. 1–38, 2020.

[34] Z. Zhou, H. Yu, and G. Fan, "Adversarial Training and Ensemble Learning for Automatic Code Summarization," *Neural Computing and Applications*, vol. 33, no. 19, pp. 12 571–12 589, 2021.

[35] F. Wang and Y. Shoshitaishvili, "Angr - The Next Generation of Binary Analysis," in *IEEE Cybersecurity Development*, 2017, pp. 8–9.

[36] T. Sharma, M. Kechagia, S. Georgiou, R. Tiwari, I. Vats, H. Moazen, and F. Sarro, "A Survey on Machine Learning Techniques Applied to Source Code," *Journal of Systems and Software*, vol. 209, p. 111934, 2024.

[37] A. Nakryiko, "HOWTO: BCC to libbpf Conversion," 2020, [accessed 30 September, 2024]. [Online]. Available: https://facebookmicrosites.github.io/bpf/blog/2020/02/20/bcc-to-libbpf-howto-guide.html

[38] BCC, "BCC Examples," 2023, [accessed 30 September, 2024]. [Online]. Available: https://github.com/iovisor/bcc/tree/master/examples

[39] MITRE, "Multi-Stage Channels," 2024, [accessed 30 September, 2024]. [Online]. Available: https://attack.mitre.org/techniques/T1104/

[40] Mitre, "Protocol Tunneling," 2024, [accessed 30 September, 2024]. [Online]. Available: https://attack.mitre.org/techniques/T1572/

[41] H. Vishwanathan, M. Shachnai, S. Narayana, and S. Nagarakatte, "Verifying the Verifier: eBPF Range Analysis Verification," in *Computer Aided Verification*. Springer Nature Switzerland, 2023.

[42] S. Bhat and H. Shacham, "Formal Verification of the Linux Kernel eBPF Verifier Range Analysis," 2022. [Online]. Available: https://sanjit-bhat.github.io/assets/pdf/ebpf-verifier-range-analysis22.pdf

[43] H. Sun, Y. Xu, J. Liu, Y. Shen, N. Guan, and Y. Jiang, "Finding Correctness Bugs in eBPF Verifier with Structured and Sanitized Program," in *European Conference on Computer Systems*. ACM, 2024, p. 689–703.

[44] J. Jia, R. Sahu, A. Oswald, D. Williams, M. V. Le, and T. Xu, "Kernel Extension Verification is Untenable," *Workshop on Hot Topics in Operating Systems*, pp. 150–157, 2023.

[45] M. Craun, A. Oswald, and D. Williams, "Enabling eBPF on Embedded Systems Through Decoupled Verification," in *Workshop on eBPF and Kernel Extensions*. ACM, 2023.