IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Received 14 August 2024, accepted 22 October 2024, date of publication 28 October 2024, date of current version 8 November 2024.

Digital Object Identifier 10.1109/ACCESS.2024.3486772

RESEARCH ARTICLE

KubeRosy: A Dynamic System Call Filtering Framework for Containers

JIN HER[1], CHI HYEON JO[1], TAEJUNE PARK[2], AND SEUNGSOO LEE[1]
[1]Incheon National University, Incheon 22012, Republic of Korea
[2]Chonnam National University, Gwangju 61186, Republic of Korea

Corresponding authors: Taejune Park (taejune.park@jnu.ac.kr) and Seungsoo Lee (seungsoo@inu.ac.kr)

This work was supported by the National Research Foundation of Korea (NRF) Grant funded by the Korea Government (MSIT) under Grant 2022R1C1C1006093.

ABSTRACT With the rapid adoption of cloud environments, container technology has become crucial for the efficient operation of large-scale applications. Although container technology offers high efficiency and scalability through low-level isolation via shared host operating systems, it also introduces security vulnerabilities, such as container escape and privilege escalation attacks through system call exploitation. Seccomp-BPF, one of the most widely used system call filtering mechanisms, supports container environments but cannot update system call policies while containers are running. To address these limitations, we propose KubeRosy, a system call filtering framework that allows dynamic modification of system call policies without downtime, even during container runtime. KubeRosy leverages eBPF and LSM hooks to support fine-grained system call policies while ensuring compatibility with existing Seccomp-BPF environments. This approach enables the application of customized, granular system call policies tailored to container environments, thereby reducing the attack surface. Our evaluation shows that KubeRosy incurs an additional overhead of only 722 ns compared to traditional Seccomp-BPF, which is negligible. Furthermore, KubeRosy allows for dynamic policy modification without container downtime and provides precise argument-based filtering, demonstrating its practicality and efficiency.

INDEX TERMS System call filtering, container runtime security, ebpf security.

I. INTRODUCTION

The adoption of cloud environments for the efficient operation of large-scale applications has rapidly accelerated in recent years. According to a report published by CSA, 98% of the financial services industry is utilizing some form of cloud computing [1]. This surge has been significantly driven by the advantages offered by containerization technologies, particularly Docker [2]. Containers provide a lightweight and portable runtime environment, ensuring consistency across different computing environments. They facilitate easy scaling, deployment, and management of applications, enabling organizations to efficiently utilize resources and achieve fast startup times. Orchestration platforms like Kubernetes [3] are essential for managing and automating the deployment, scaling, and operation of containers in a cloud-native environment.

However, from a security perspective, the lower level of isolation provided by containerization, which shares the host operating system, poses risks such as container escape and privilege escalation attacks. These vulnerabilities can be exploited through the malicious use of system calls by compromised containers [4]. System calls serve as a critical interface between user space applications and the kernel, enabling essential operations such as file access, process management, and network communication, which require access to the host's hardware resources. As of Linux kernel version 5.15, approximately 400 system calls are supported; however, most applications utilize only a subset of these calls. If a program permits system calls beyond those it necessarily uses, along with their associated arguments, malicious users could exploit these additional system calls to compromise the system.

The associate editor coordinating the review of this manuscript and approving it for publication was Parul Garg.

A notable example of a vulnerability caused by the exploitation of system calls in containers is the leaky vessels vulnerability (CVE-2024-21626) [5], [6], [7]. It has a CVSS score of 8.6 and arises from file descriptor leakage and improper working directory settings in the runc container runtime. This vulnerability allows an attacker to gain access to the host file system through a malicious container image, thereby not only compromising the container's isolation mechanisms but also potentially escalating privileges on the host system. To defend against such attacks, it is essential to prevent file descriptor leakage and filter the chdir system call to block the setting of improper working directory paths. This highlights the critical importance of system call security in protecting the host kernel from untrusted containers in a containerized environment.

Several previous studies have aimed to enhance security in container environments through system call filtering. Notably, system call policy generation [8], [9], [10], [11], [12], [13], [14] involves analyzing container images, binaries, and other components to generate a whitelist of allowed system calls. Containers are then restricted from invoking system calls that are not on this list, thereby preventing malicious behavior. However, these studies commonly rely on secure computing (seccomp) profile for policy enforcement. Seccomp has limitations, such as the inability to modify the seccomp profile while the container is running and the restriction to only inspect basic data types of system call arguments. On the other hand, various container runtime security frameworks [15], [16], [17], [18], [19], [20] focus on enforcing container security policies. However, they also face limitations in achieving comprehensive security for system call execution itself.

In this paper, we propose KubeRosy, a framework designed to allow dynamic updates to system call security policies even while containers are running, and to provide more granular, argument-based system call security policies. KubeRosy leverages the extended berkeley packet filter (eBPF) [21] and linux security module (LSM) hooks [22] to apply policies. Additionally, when applying basic seccomp profiles to containers, our system automatically converts and applies them as KubeRosy policies that we devise, ensuring compatibility with environments that use seccomp-BPF. Moreover, it supports fine-grained system call security policies, ultimately enabling the deployment of a deny-all policy for system calls while allowing only the specific system calls and argument values required by the application, thus enforcing the principle of least privilege. Our evaluation demonstrates that KubeRosy enables dynamic system call policy updates without any service downtime of containers and the conversion of existing seccomp profiles in containerized environments, all while incurring negligible performance overhead, even in scenarios involving detailed argument-based filtering.

In summary, our main contributions are as follows:

- *Dynamic updates of system call policy.* By leveraging eBPF and LSM hooks, we enable dynamic updates to system call security policies during container runtime, eliminating the need for container redeployment and the associated service downtime typically required to update these policies.
- *Support for fine-grained system call policies.* We provide argument-based system call filtering functionality, allowing the creation of policies that block specific system calls while permitting them for certain required argument values. This approach reduces the attack surface of containers in a more refined and precise manner.
- *Compatibility with seccomp profiles.* We provide compatibility with existing seccomp profiles by automatically converting and deploying them as KubeRosy Policy that we devise for containers with seccomp profiles applied.

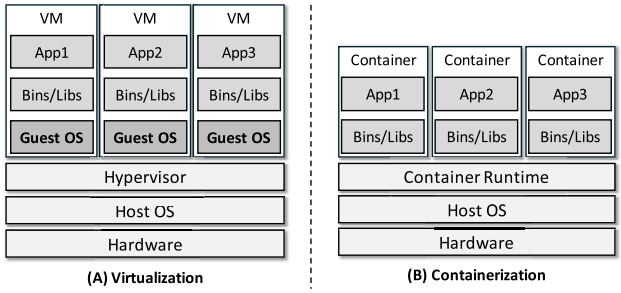


FIGURE 1. Comparison between virtualization and containerization.

The rest of this paper is organized as follows: in Section II, we introduce the relevant background knowledge and motivating example. Section III reviews related work and existing frameworks, highlighting their limitations. In Section IV, we explain the system design and overall workflow of KubeRosy. Next, seccomp profile conversion, dynamic policy enforcement, and fine-grained system call filtering, are discussed in detail in Section V. Section VII covers the performance evaluation and functionality correctness of KubeRosy. Finally, Section VIII addresses the limitations and future works of KubeRosy, and Section IX concludes the paper by summarizing the research findings.

II. BACKGROUND AND MOTIVATION

A. CONTAINERIZATION

As illustrated in Figure 1, virtualization [23] (A) operates by running a hypervisor on the physical hardware, which provides each virtual machine (VM) with an independent operating system and kernel. VMs operate in fully isolated environments, offering high security. However, each VM includes its own OS, consuming significant system resources and requiring the guest OS to boot every time a VM is started. In contrast, containerization [24], [25] (B) employs OS-level virtualization, allowing multiple containers to share the kernel from the host operating system. Containers are implemented using Linux kernel technologies such as namespaces and control groups (cgroups).

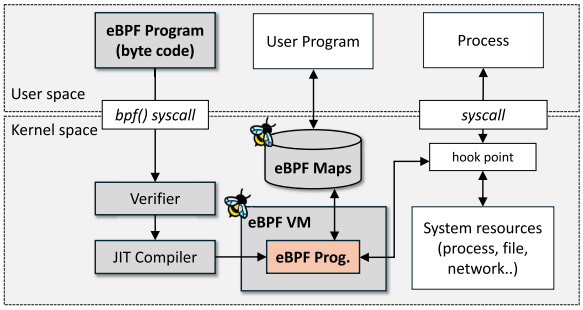


FIGURE 2. The workflow of an eBPF program with key components.

Namespaces virtualize system resources, such as process IDs, network interfaces, and filesystem mounts, providing each container with an independent execution environment. Cgroups manage and isolate resource usage, such as CPU, memory, and network bandwidth, ensuring that one container does not excessively consume resources to the detriment of others. Containers package applications and their dependencies into a single unit, enabling consistent execution across various environments. The lightweight and portable nature of containers has significantly simplified the development and deployment of applications.

However, due to the shared host OS kernel, containers offer a lower level of isolation than VMs. Consequently, a security vulnerability in one container can potentially threaten other containers or the host system. In addition, this has also highlighted the need for efficient management of numerous containers, maintaining their isolation, and ensuring overall system stability. To address these complexities, container orchestration has become essential. A prominent platform, *Kubernetes* [3], automates the deployment, scaling, and management of container-based applications. Kubernetes groups multiple containers into a single deployment unit called a *Pod* and efficiently distributes these across the entire cluster. This approach optimizes system resources, maintains high availability of services, and facilitates continuous updates and scaling of applications.

B. EXTENDED BERKELEY PACKET FILTER

The extended berkeley packet filter (eBPF) [21] is a technology designed to safely and efficiently extend kernel functionality without modifying the Linux kernel source code itself or loading kernel modules. Figure 2 illustrates the key components and working principles of eBPF. eBPF programs are loaded into the kernel as bytecode via the `bpf()` [26] system call. During this process, the verifier ensures the program's safety by enforcing rules such as guaranteeing program termination and restricting arbitrary memory access, allowing safe execution in kernel mode. The verified programs are then optimized for execution performance by being converted into native machine code by the just-in-time (JIT) compiler according to the CPU architecture. Once compiled, these programs are attached to various kernel hook points, such as sockets and tracepoints, and are executed when these hook points are invoked.

The eBPF map serves as an in-kernel data structure accessible by both eBPF programs and user space programs, acting as a communication layer between them. The helper functions form an application programming interface (API) and application binary interface (ABI) between eBPF programs and the kernel, enabling a range of operations from additional information retrieval (e.g., process information gathering) to packet redirection. A key advantage of eBPF is its ability to extend user intent into the kernel space. For example, performing packet filtering directly at the kernel level significantly reduces the overhead associated with frequent context switching between user space and kernel space. This leads to improved network performance and increased efficiency in system resource utilization.

Seccomp-BPF: Secure computing (secommp) [27] is a sandboxing mechanism provided by the Linux kernel that offers two modes. First, in SECCOMP_SET_MODE_STRICT, it restricts a process to executing only the read(), write(), sigreturn(), and exit() system calls,[1] blocking all others. In contrast, the SECCOMP_SET_MODE_FILTER mode uses seccomp-BPF [28], allowing more granular restrictions on a wider range of system calls specified in the profile [29]. Seccomp-BPF, an extension of seccomp is a Linux kernel feature designed to define rules for system calls that an application can perform, thereby blocking unnecessary or dangerous system calls. Initially introduced as a simple process isolation tool in kernel version 2.6.12, it has since evolved to include BPF filtering capabilities (classic BPF, not eBPF) classicbpf-vs-ebpf. This allows seccomp-BPF to use the BPF language to define system call filters that allow, log, or deny calls by examining specific system call numbers and non-pointer argument values

LSM-BPF: Linux security module (LSM) [22] is a lightweight framework designed to integrate various security mechanisms, providing a standardized interface for implementing security policies within the kernel. The key of the LSM is LSM hooks, which are strategically placed at critical points in the kernel operations, such as file system actions and network socket manipulations. As of Linux kernel version 5.15, there are about 190 LSM hooks [30]. The LSM itself does not directly perform security enhancement functions. Instead, it facilitates the implementation of security policies through specific modules such as Apprmor [31], SELinux [32], and LSM-BPF [33]. LSM-BPF incorporates eBPF technology into the LSM framework, enabling privileged users to dynamically manipulate LSM hooks at runtime and flexibly implement mandatory access control (MAC) [34] and auditing policies across the system. Furthermore, LSM-BPF maintains compatibility with existing LSM-based security solutions while enabling more granular and dynamic security policy application.

[1]In this paper, the terms 'system call' and 'syscalls' are used interchangeably.

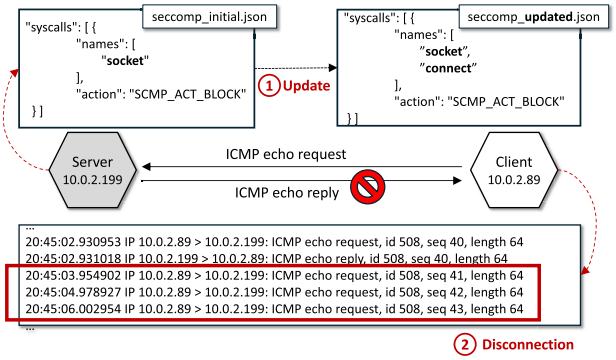


FIGURE 3. This figure illustrates how updating a seccomp profile leads to session disconnection due to the inherent limitation of the seccomp enforcement mechanism, which cannot be updated at runtime.

C. MOTIVATING EXAMPLE

Figure 3 illustrates a scenario where a container with an initial seccomp profile (seccomp_initial.json) attempts to modify it to an updated profile (seccomp_updated.json) at runtime. In this example, it is assumed that the server and client applications are deployed as pods within the Kubernetes environment, with the client performing simple ping communications with the server. During this normal ping communication, an administrator identifies a change in security requirements (adding the `connect()` system call to the seccomp profile) for the server pod and attempts to modify the seccomp profile. When the administrator modifies the server's seccomp profile, the server undergoes a rolling update, deleting the existing pod and deploying a new pod with the updated seccomp profile (1). However, during this update, the session between the client and the server is interrupted as the existing server is deleted (2).

The problem is that it is not possible to modify or delete the seccomp profile of a running container at runtime. The only way to apply an updated seccomp profile is to terminate the container and restart it with the new profile. Although Kubernetes provides a rolling update mechanism to automate this process, it still results in unavoidable downtime, potentially introducing additional attack surfaces.

III. RELATED WORK

A. SYSTEM CALL POLICY GENERATION

System call policy generation is a critical research area focused on reducing the attack surface between containers and the kernel, which has spurred various studies [8], [9], [10], [12], [13], [14]. Ghavamnia et al. [8] proposed Confine, a system that automatically generates restrictive system call policies for Docker containers through static code analysis. On the other hand, Iiguni et al. [9] introduced Sprofiler, which combines static analysis of application executables with dynamic analysis of system calls issued by containers to generate filtering rules suitable for container workloads. Yang et al. [10] proposed Optimus, a tool that continuously monitors and dynamically restricts system calls. This emphasizes identifying and applying the set of system calls required by the container at runtime, rather than relying on a fixed set of system calls. Sysverify [12] presented a methodology that combines static and dynamic analysis to generate seccomp profiles through binary and source code analysis. Li et al. [13] proposed AUTOARMOR, which integrates a static analysis-based request extraction mechanism with a graph-based policy management mechanism in cloud environments. Kim et al. [14] introduced Prof-gen, a tool that generates system call whitelists by combining dynamic and static analysis based solely on container images and execution commands.

However, those studies commonly focus on generating seccomp profiles for system call filtering. As demonstrated in Section II-C, seccomp profiles are applied when a container is executed and cannot be modified or deleted without redeploying that container. Additionally, they have limitations in fine-grained control over system calls using non-pointer argument-based filtering. In contrast, our research addresses these limitations by utilizing eBPF and LSM to dynamically modify eBPF maps and apply syscall policies while the container is running, with the added capability of performing fine-grained filtering using argument values obtained from LSM hooks.

B. CONTAINER RUNTIME SECURITY FRAMEWORK

In cloud-native environments, various tools and frameworks have been proposed to restrict container behavior in real-time [15], [16], [17], [18], [19], [20]. Falco [15], introduced by Sysdig, is an open-source project that monitors container activity and alerts on any abnormal behavior that violates predefined rules. Similarly, KubeArmor [16] and Tetragon [17] offer runtime security enforcement and observability based on user-defined policies. These tools specifically leverage eBPF to monitor or restrict container activities, including process execution, network, and file access (I/O operations). KRSIE [18] is a BPF-LSM-based security policy enforcement system that provides fine-grained control and dynamic application of security policies using eBPF. Whenever users deploy BPF-LSM policies in a Kubernetes environment, KRSIE analyzes them to generate corresponding BPF-LSM programs, which are then executed in the kernel. SPEAKER [19] proposes a mechanism that profiles system calls required during the boot and execution phases of a container, dynamically changing the allowed system calls as the container transitions for such life cycle. bpfbox [20] enhances system security by using eBPF programs and LSM probes to allow, audit, or block specific operations based on defined policies as well.

However, while KubeArmor and Falco do not support system call-level blocking policies, Tetragon does support such policies but implements blocking by sending a signal to terminate the process executing the system call, rather than blocking the execution flow of the system call itself. This reactive model can potentially allow attacks, such as ransomware, to proceed before mitigation actions can

be taken, resulting in encrypted assets or deleted data. In contrast, our research offers the advantage of blocking system calls during their execution flow, thereby preventing attacks like ransomware before they can cause harm. Additionally, KRSIE incurs overhead due to the parsing of policies, generating BPF programs, and loading them into the kernel. In comparison, KubeRosy updates policies by modifying the eBPF map after parsing, resulting in lower overhead during policy enforcement. Moreover, SPEAKER requires the insertion of a kernel module, which could introduce additional attack surfaces, and since it uses Seccomp-BPF for system call filtering, it has the limitation that it can only inspect arguments of basic data types.

IV. KubeRosy OVERVIEW

This section outlines the design considerations that underpin KubeRosy, along with a comprehensive description of its system architecture. Succinctly, our system is designed to mitigate malicious activities within containers by enabling real-time enforcement and updates of system call policies, while ensuring the continuity of container operations without disruption.

A. DESIGN CONSIDERATIONS

The motivating example (from Section II-C) demonstrated how runtime seccomp profile updates impact the workload of the container being performed. In addition, simply restricting system calls themselves without fine-grained arguments is not enough to prevent sophisticated malicious activity. Hence, we propose a framework for dynamically and elaborately identifying and blocking system calls in cloud-native environments based on the following three design considerations.

R1: Container-Tailored Syscall Filtering. It should accurately identify the containers to which the system call policy will be applied and ensure compatibility with existing seccomp profiles. To track the lifecycle of rapidly changing containers in a cloud-native environment in real-time, the system monitors creation and deletion events by communicating with the container runtime API server. It also leverages the namespace mechanism [35] provided by Linux to identify processes within the same container enforced by a syscall policy. Furthermore, it includes an automated methodology for converting existing seccomp profiles into the dynamic syscall policies we propose. These approaches enable us to provide a container-specialized syscall filtering system.

R2: Dynamic Updates Without Downtime. It should be capable of dynamically modifying the syscall policy without causing downtime for containers and services they provide. Additionally, the overhead imposed on the container during filtering should be minimized. To achieve this, eBPF (Extended Berkeley Packet Filter) and LSM (Linux Security Modules) hooks are employed to filter system calls within the kernel. This approach enables dynamic filtering while minimizing the context switching overhead on the container. Thus, our approach seeks to update the system call policy as needed with minimal overhead and without requiring the redeployment of containers, thereby enhancing its flexibility.

R3: Fine-Grained Filtering by Arguments. It should support not only basic system calls filtering but also more fine-grained filtering based on the argument values for each system call. For this, we devise a concrete policy scheme specifying the arguments and propose a methodology for efficiently managing eBPF maps for argument filtering in the kernel by specifying the argument values associated with each system call. This approach allows us to minimize the attack surface of malicious containers by by ensuring that only necessary system calls with specific arguments are allowed.

B. SYSTEM ARCHITECTURE

This section presents the overall architecture of KubeRosy and explains its components. As illustrated in Figure 4, our framework comprises three main components: *KubeRosy policy*, *operator*, and *agent*. Our system operates on a Kubernetes cluster, the de facto standard for container orchestration, with the operator running on the master node and the agent on the worker nodes. The framework operates in two distinct phases. The first phase, policy enforcement, focuses on monitoring the lifecycle of containers and their associated syscall policies, subsequently updating these policies into the eBPF maps. The second phase, system call filtering, applies the policy by preventing system calls from running in the container unless they have specific parameters that fall under an exception.

KubeRosy Policy: Figure 5 depicts the *KubeRosy Policy (KRP)*[2] structure, which is designed to implement security measures at the system call level within the container. The KRP comprises three main parts: a selector that identifies the target container to which the KRP is applied, a list of security rules, and a status field containing information about the container subjected to the policy. Notably, the status is not user-defined; instead, it is updated in real-time by the operator during the policy enforcement phase. Each rule consists of a system call name and a corresponding list of argument values, allowing the system to filter these system calls with fine-grained precision.

KubeRosy Operator: The operator consists of two submodules: the *policy handler* and the *log collector*. The policy handler receives policy information from the Kubernetes API server, which monitors the KRP. The policy information includes a selector used to identify the container to which the policy will be applied and to determine the node on which that container is running. The handler then transmits the policy information to the agent deployed on the corresponding node. It can also detect when the default seccomp profile is applied to the container, convert it to a KRP, and deploy it. The log collector gathers logs related to system call policies from the agents deployed on each node, enabling users to understand when, where, and why a system call was blocked.

[2]In this paper, the KubeRosy Policy is abbreviated as 'KRP'.

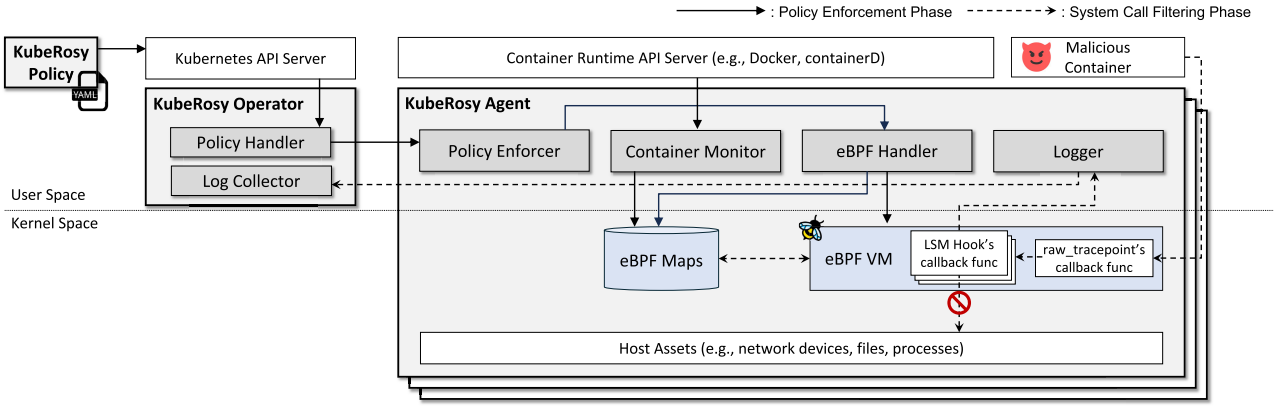


FIGURE 4. Overall architecture of KUBEROSY with three key components: (*i*) KUBEROSY policy, (*ii*) operator, and (*iii*) agent. The workflow is divided into two phases: policy enforcement and system call filtering.

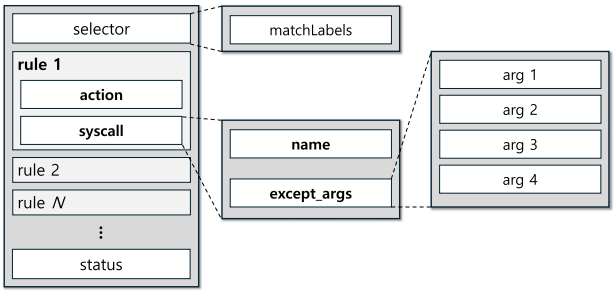


FIGURE 5. Overview of KUBEROSY policy (KRP) structure. The KRP primarily operates in a blacklist manner, with exceptions for specific arguments.

KubeRosy Agent: The KubeRosy agent has four sub modules: the *container monitor*, *policy enforcer*, *eBPF handler*, and *logger*. The container monitor detects both pre-existing containers in the cluster and those deployed in real time, storing identifiable information in the eBPF map. Since multiple processes can run within a single container, applying policies to each process individually is inefficient. To address this, our system uses the mount namespace ID (mntns) and PID namespace ID (pidns), obtained from the PID of the init process of the container, to apply policies collectively to processes within the same container. Upon the container deployment, the container monitor interacts with the container runtime API server [2], [36] and updates the eBPF map with the init process PID and container ID per each container, enabling simultaneous identification of multiple processes in a container for the filtering phase later.

The policy enforcer receives the KRP policy information from the policy handler in the operator and converts it into a format suitable for storage in the eBPF map. Then, it queries the Kubernetes API server with the KRP selector to obtain the container ID and then retrieves the init process PID of the container managed by the container monitor. Using this PID, it determines the mntns and pidns of the container, and they are mapped with the policy rules and passed to the eBPF handler for storage in the eBPF map.

The eBPF handler is basically responsible for loading the eBPF map, LSM hook's callback functions, and raw_tracepoint's callback function into the kernel space of the host when the agent is deployed. The raw_tracepoint's callback function is triggered when a system call is executed in the container, recording information about the system call in the eBPF map. Next, the triggered LSM hook's callback function uses this system call information, along with the policy information, to determine in real time whether to block the system call. If a system call is blocked, the LSM hook's callback function uses ring buffers to send log information to the logger and the logger running inside each node forwards it to the log collector in the operator.

V. KUBEROSY SYSTEM DETAILS

This section details the features of KUBEROSY that meet the design requirements (§ IV-A), organized as follows: seccomp profile conversion (§ V-A), dynamic policy enforcement (§ V-B), and fine-grained system call filtering (§ V-C).

A. SECCOMP PROFILE CONVERSION

KUBEROSY automatically converts existing static seccomp profiles into dynamically manageable the KRPs by utilizing an admission control mechanism provided by Kubernetes. This mechanism ensures that resources (e.g., pods, services, etc.) are created safely and correctly by authenticating, authorizing, and verifying or modifying requests through various admission controllers when a request of the resource creation is made to the Kubernetes API server. There are two types of admission controllers: mutating and validating. Our system employs a mutating admission controller [37] (i.e., webhook) to intercept pod creation requests and, if a seccomp profile is set, converts it into a KRP policy. This process provides seamless compatibility for environments using existing seccomp profiles, eliminating the need to manually create KRPs.

Figure 6 shows a flowchart of the process for converting a seccomp profile into a KRP. When a pod creation is requested, the mutating webhook defined within the policy

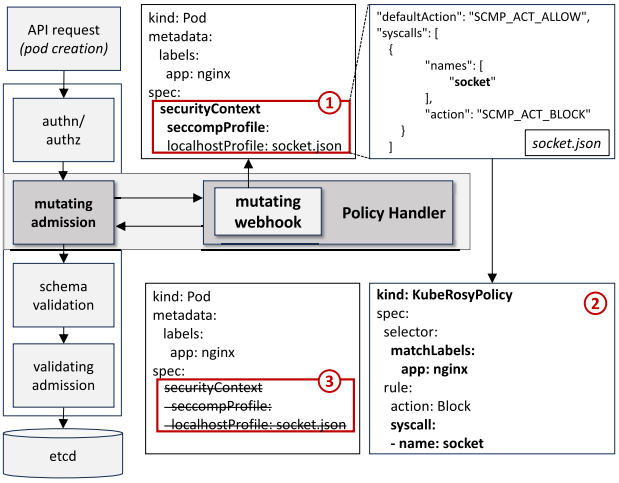


FIGURE 6. The example scenario of automatic seccomp profile conversion by the policy handler.

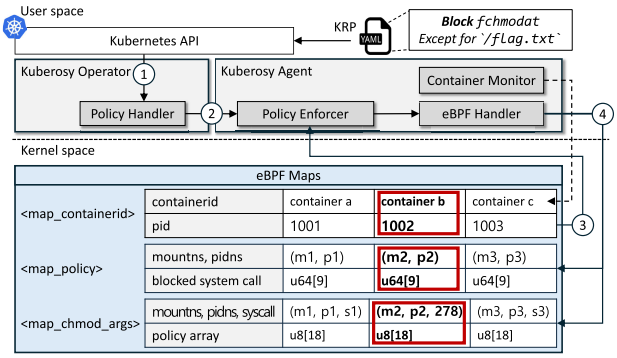


FIGURE 7. The steps of enforcing the KRP (block the `fchmodat` syscall, except for `/flag.txt`) through the eBPF map updates.

handler is triggered to check if the seccomp profile is specified in the 'securityContext' in the pod specification. If the profile exists, the handler extracts the profile information from the received pod specification, analyzes it, and maps the listed allowed or blocked system calls to the rules in the KRP. In this example, the pod labeled 'app: nginx' specifies a seccomp profile that blocks the `socket` system call (1). Upon detecting this, the handler converts it into a corresponding KRP (2). Once the conversion is complete, the seccomp-related settings are removed from the original pod specification, and the newly created KRP is deployed (3). If any issues arise during the conversion process, the mutating admission controller returns the original request unchanged. This process replaces the static seccomp settings with a dynamically manageable KRP, ensuring that the container is deployed with equivalent functionality.

B. DYNAMIC POLICY ENFORCEMENT

The dynamic policy enforcement process begins with the eBPF handler in the agent deployed on each node, which loads the necessary eBPF programs, and four eBPF maps into the kernel. The key, value, and purpose of each eBPF map

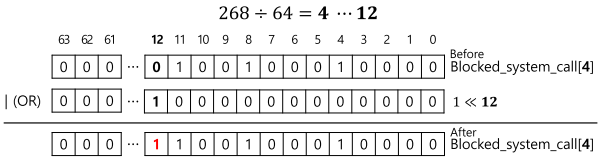


FIGURE 8. The value update of map_policy for KRP enforcement using bitmap operations.

are summarized in Table 1. After loading the programs and maps, the container monitor in the agent runs as a separate thread, monitoring container creation and deletion events on the node via the container runtime API, as illustrated by the dotted line arrow line in Fig 7. Upon receiving a container creation event from the API, the monitor stores the container ID and the PID of its init process as key-value pairs in the map_containerid eBPF map. These procedures fulfill the minimum requirements for enforcing the KRP.

The solid line arrows in Fig 7 indicate the sequence in which eBPF maps are updated during the KRP enforcement. The KRP blocks the fchmodat system call, but allows an exception if the path argument of the syscall is `/flag.txt`. When the KRP is deployed to the cluster via the Kubernetes API, the policy handler, which monitors for such events, is first triggered, and extracts the policy information (1). It then analyzes the selector in the policy to identify the node hosting the target pod and transmits the policy information to the policy enforcer in the agent on that node (2). The policy enforcer, upon receiving the policy information, obtains the container ID matching the selector from the Kubernetes API (3). It then uses this container ID (`container b`) to find the init process ID (`1002`) via map_containerid and searches the `/proc/1002/ns` path to acquire the mount namespace ID (mntns) and PID namespace ID (pidns). This is done to apply a single system call policy to all processes within the same container, as they share the same mntns and pidns values. When the policy information, including mntns (`m2`) and pidns (`p2`), is passed to the eBPF handler, the handler updates map_policy (4).

Each record in the map_policy, which determines whether to block a system call, uses mntns and pidns as the key. The value associated with each key is a u64[9] array (i.e., blocked_system_call), totaling 576 bits. Each bit position represents a specific system call number, allowing the policy to control up to 576 system calls. As of Linux kernel version 6.10, there are 410 system calls, though the system call numbers are not sequential, with the highest being 547. The 576-bit array is provided to efficiently cover this range. This implementation uses a single bit to indicate whether to block or allow a system call, and bitwise operators are used to quickly check these policies, enhancing performance. In this scenario, the fchmodat system call (number 268) is blocked for the containers with the mntns and pidns values of m2 and p2, respectively. The value (2320) is updated at the 4th index in the blocked_system_call array, which is retrieved

TABLE 1. The summary of eBPF maps used for KRP enforcement, including keys, values, and descriptions.

Name	Key	Value	Description
map_containerid	containerid	pid	Identification of whether the process is within the same container
map_lsm_syscall	mountns, pidns	LSM hook array	Identification of the system call triggering the LSM hook
map_policy	mountns, pidns	policy array	KRP enforcement for blocking system calls on a per-container basis
map_LSM_hook_args	mountns, pidns, system call	arguments	Fine-grained syscall filtering on arguments per each LSM hook

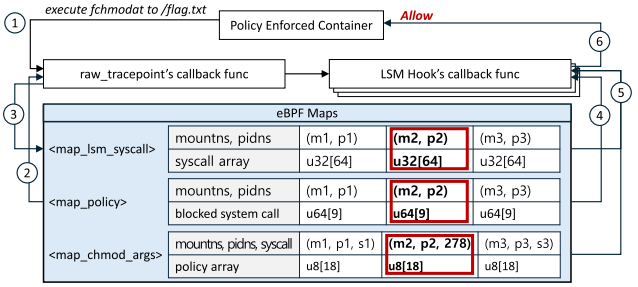


FIGURE 9. The steps for allowing the fchmodat syscall with the /flag.txt argument. The final decision conducted at the LSM hook's callback function.

by searching for the m2 and p2 in the map_policy as shown in Figure 8. Notably, since the policy blocks the fchmodat system call but permits it for the specified path (/flag.txt), the path value is updated in the map_chmod_args map to filter the argument value of the system call. This map stores the path (16 bytes) and mode (2 bytes), totaling 144 bits, as values for the arguments of the fchmodat system call.

C. FINE-GRAINED SYSTEM CALL FILTERING

Figure 9 illustrates the runtime blocking process when the system call (fchmodat) is invoked from the container after the policy enforcement described above. When the fchmodat is invoked, the raw_tracepoint's callback function installed by the eBPF handler in the agent is triggered, and the our system specifies whether this process in the container is monitored by the KRP, following the steps outlined in Algorithm 1. First, a helper function retrieves the mntns and pidns of the process that executed the system call (lines 1-4). These values are then used as the key (ns) to look up the map_policy to verify whether the process is subject to the policy (line 5). If so, the arg[1] of ctx is accessed to obtain the number of the system call, and the lsm_map, which we defined, is used to find the LSM hook number associated with that system call (lines 7-8). Next, map_lsm_syscall is searched using ns as the key to obtain the syscall_arr value; if not found, it is initialized (lines 9-11). The LSM hook number retrieved in arr is then updated. In this example, the fchmodat triggers the security_path_chmod LSM hook, corresponding to the hook number 4, so arr[4] is updated to 278 in map_lsm_syscall, and 0 is returned (lines 12-14). This procedure ensures that although a LSM hook intercepts syscalls, a specific system call is accurately identified, as multiple system calls may trigger the same LSM hook.

Algorithm 1 System Call Monitoring per Container

Input: raw_tracepoint_args ctx
Output: 0
1 $T \leftarrow$ get_current_task() // Get current task;
2 $m \leftarrow$ get_mount_namespace_id(T);
3 $p \leftarrow$ get_pid_namespace_id(T);
4 $ns \leftarrow$ struct(m, p);
5 $is\_container\_process \leftarrow$ map_lookup(map_policy, ns);
6 **if** $is\_container\_process \neq NULL$ **then**
7 $syscall\_id \leftarrow$ ctx.arg[1];
8 $lsm\_id \leftarrow$ lsm_map[$syscall\_id$];
9 $arr \leftarrow$ map_lookup_elem(map_lsm_syscall, ns);
10 **if** $arr = NULL$ **then**
11 $arr \leftarrow$ initialized array;
12 $arr[lsm\_id] \leftarrow syscall\_id$;
13 map_update_elem(map_lsm_syscall, ns, arr);
14 **return** 0
15 **return** -1

After identifying the system call in the triggered raw_tracepoint's callback function, the callback function of the corresponding LSM hook's callback function is triggered to determine whether the call should be blocked or allowed, as outlined in Algorithm 2. First, the mntns and pidns are obtained from the LSM hook's callback function, and map_policy is referred using these values to verify whether the process that invoked the system call is associated with the container to which the policy applies (lines 5-6). If so, map_lsm_syscall is queried with ns to identify the specific system call that activated the LSM hook, by obtaining the system call array and using the LSM hook number as an index (lines 7-8). The array and ns are then used to check whether the system call is blocked by the policy; if it is allowed, the function returns 0 here (lines 9-11). Otherwise, the policy with the arguments for the system call is checked again (line 12). If the policy exists, it is evaluated to determine if the system call should be permitted, and if so, the function returns 0 (lines 12-16). In this scenario, while the fchmodat system call is generally blocked, it is allowed if the path argument is '/flag.txt', resulting in a return value of 0. If not, a ring buffer is created, storing information such as the PID of the process that executed the system call and the specific system call details, then -1 is returned (lines 17-21).

Algorithm 2 Final Decision in LSM Hook

Input: LSM_Hook's_args ctx
Output: 0 or -1

```
T ← get_current_task()          // Get current task;
m ← get_mount_namespace_id(T);
p ← get_pid_namespace_id(T);
ns ← struct(m, p);
policy_array ← map_lookup(map_policy, ns);
if policy_array ≠ NULL then
    syscall_array ← map_lookup(map_lsm_syscall, ns);
    syscall_id ← syscall_array[lsm_id];
    is_blocked ← lookup_policy(ns, syscall_id, policy_array);
    if is_blocked = NULL then
        return 0;
    arr ← map_lookup_elem(policy_argu_map, ns, syscall_id);
    if arr ≠ NULL then
        is_allowed ← check_argu(ctx, arr);
    if is_allowed then
        return 0;
    pid ← get_current_pid();
    event ← ringbuf_reserve();
    event.pid ← pid;
    ringbuf_submit(event);
    return -1
return 0
```

VI. IMPLEMENTATION

To demonstrate the feasibility of KubeRosy, we have implemented a prototype using Go and C languages comprising three key components totaling approximately 3K lines of code. This system was basically designed as a framework for filtering system calls executed within containers in a Kubernetes cluster. Technically, the prototype runs on the Linux 5.15 kernel, alongside Kubernetes v1.28.2 and Docker v24.0.7, or containerd v1.6.31. The operator, implemented using Kubebuilder [38], deploys a mutating admission webhook for modifying seccomp profiles. When a pod containing a seccomp profile is created, the webhook server, implemented with the admission and HTTP libraries, redirects the creation event. The eBPF features responsible for system call filtering within the kernel were written in C and compiled into object files using Clang [39]. These eBPF features were then loaded into the kernel using the Go-eBPF [40] library. Additionally, communication between the operator's policy handler and the agent's policy enforcer, as well as between the agent's logger and the operator's log collector modules, was implemented using gRPC [41].

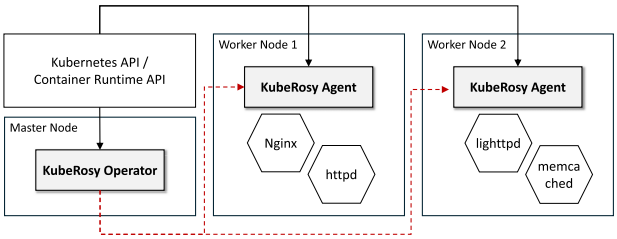


FIGURE 10. Test environment consisting of one master node and two worker nodes.

VII. EVALUATION

In this section, we conduct a real-world evaluation of KubeRosy to showcase its *(i) performance* and *(ii) functionality correctness* in practical environments.

A. EVALUATION ENVIRONMENTS

We evaluated KubeRosy using a high-performance server equipped with a Xeon Gold 6342 CPU, 768 GB of RAM, 8 TB SSD, and 28 TB HDD. As shown in Fig. 10, the Kubernetes cluster was configured with one master node and two worker nodes, all hosted on Ubuntu 22.04 virtual machines. Containerd was used as the container runtime for Kubernetes, and Cilium was employed as the Container Network Interface (CNI) for communication between the containers. In the cluster, the KubeRosy operator works on the master node, while the KubeRosy agent was deployed on all worker nodes using a DaemonSet. Subsequently, Nginx, httpd, lighttpd, and memcached pods were distributed across the two worker nodes to measure the application performance, microbenchmarking, and use case testing.

B. PERFORMANCE EVALUATION

1) MICROBENCHMARK

The microbenchmark evaluation measures the overhead incurred by enforcing system call policies within the Nginx pod. Specifically, the overhead of system call enforcement was assessed by executing the `socket()` system call 5 million times and calculating the average execution time, as shown in Figure 11. For the allow policy, the comparison was made between the overhead of having no policy applied (default), seccomp's SCMP_ACT_ALLOW, and Tetragon's [17] policy. The results indicate that KubeRosy introduced an overhead of 1068ns and 722ns compared to no policy and seccomp, respectively, which is minimal. In contrast, Tetragon exhibited a higher overhead.

On the other hand, for the block policy, we compared the average execution times of the socket() system call blocked by seccomp and our system. Although Tetragon also supports system call blocking policies, it was excluded from this experiment because its blocking mechanism terminates the entire process rather than just blocking the system call. The results show a significant difference compared to the allow policy, which is due to the timing differences in how seccomp and LSM hooks operate. Specifically, when a user-space process

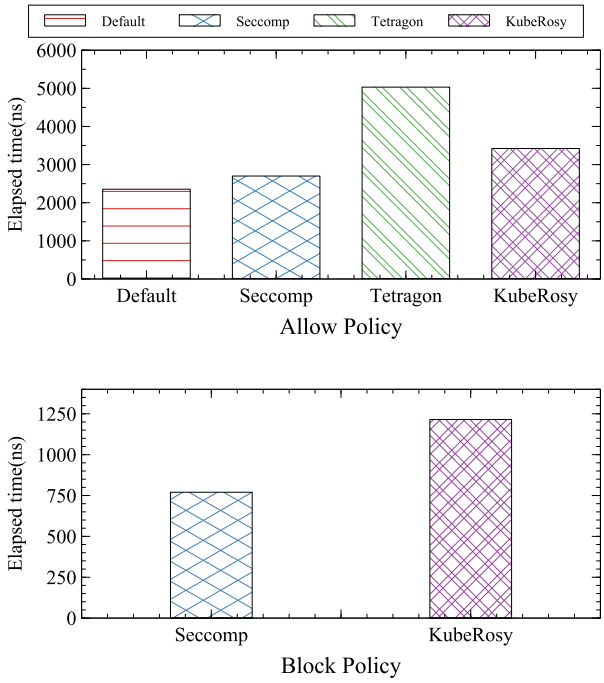


FIGURE 11. The results of the microbenchmark measuring socket() system call execution.

initiates a system call, tasks such as context switching from user space to kernel space and configuring the pt_regs structure are performed before the system call is executed through the do_syscall_64 or do_syscall_32 kernel functions. The do_syscall_64 function (Listing 1) applies seccomp filters via the syscall_enter_from_user_mode() function before executing the system call. If syscall_enter_from_user_mode() returns −1 due to a seccomp block policy, the condition in the do_syscall_x64() function is not met, and the system call is blocked before it can be executed.

In contrast, with the LSM hook used by our system, the system call is executed after passing through the syscall_enter_from_user_mode() function and the do_syscall_x64() function before reaching the LSM hook point, which results in a difference in timing. However, our system has the advantages of allowing dynamic policy changes and providing control over arguments.

2) APPLICATION PERFORMANCE

As shown in Table 2, performance was measured for three web servers (i.e, nginx, httpd, and lighttpd) and the memcached in-memory cache application. The results are presented in Figure 12. Specifically, the performance of the web server applications was measured using the Apache HTTP server benchmarking tool [42], with concurrency set to 100, while the performance of the in-memory cache application was measured using memtier_benchmark [43], with threads set to 10. Figure 12 shows the normalized throughput and latency values for each application, using

```
void do_syscall_64(struct pt_regs *regs, int nr)
{
 add_random_kstack_offset();
 nr = syscall_enter_from_user_mode(regs, nr);

 instrumentation_begin();

 if (!do_syscall_x64(regs, nr) && !do_syscall_x32(
    regs, nr) && nr != -1) {
  /* Invalid system call, but still a system call.
     */
  regs->ax = __x64_sys_ni_syscall(regs);
 }

 instrumentation_end();
 syscall_exit_to_user_mode(regs);
}
```

LISTING 1. /arch/x86/entry/common.c.

TABLE 2. The summary of applications used for evaluating application performance degradation.

Application	Description	Benchmark
nginx	Web server	ab(100 clients) [42]
httpd	Web server	ab(100 clients) [42]
lighttpd	Web server	ab(100 clients) [42]
memcached	In-memory cache	memtier(10 threads) [43]

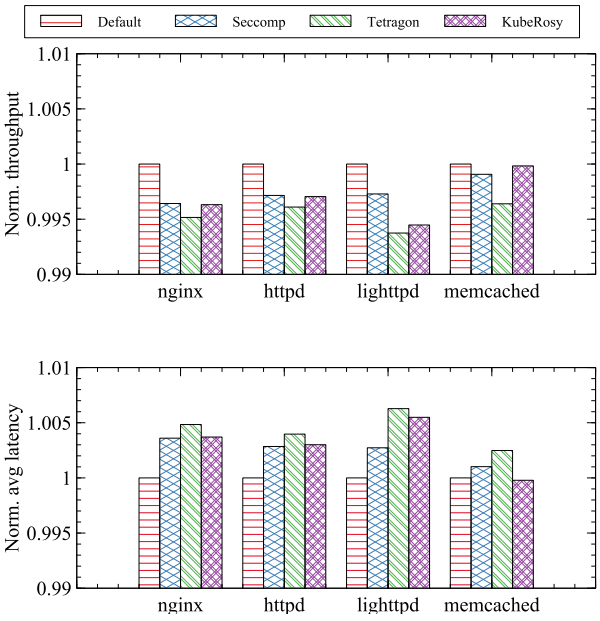


FIGURE 12. The results of throughput and latency normalization across the applications.

the default configuration (with no policies applied) as the baseline. These values were compared with the overhead introduced by the allow policies of seccomp, KubeRosy, and Tetragon. The results indicate that our system exhibits a minimal performance difference of approximately 0.3% in both throughput and latency compared to the default configuration. When normalized against seccomp, the performance difference between seccomp and KubeRosy is around 0.01%.

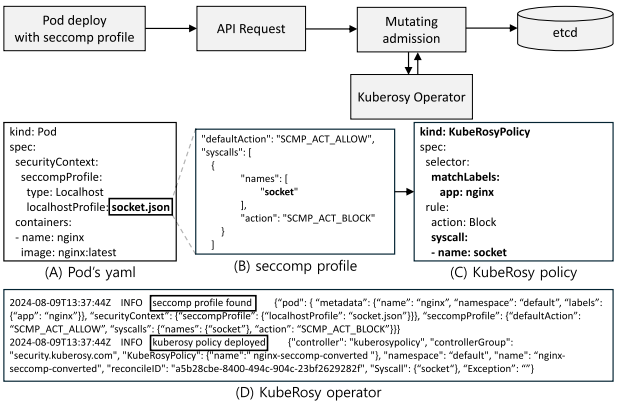


FIGURE 13. The results of the seccomp profile conversion to KRP.

```
apiVersion: security.kuberosy.com/v1
kind: KubeRosyPolicy
metadata:
    name: nginx-block
    namespace: default
spec:
    selector:
        matchLabels:
            app: nginx
    rule:
        action: Block
        syscall:
            - name: fchmodat
              execption:
                  - path: "/benign.txt"
                  - mode: "755"
```

LISTING 2. nginx-block-chmod.yaml.

This demonstrates that the performance degradation due to the KRP enforcement is negligible.

C. FUNCTIONALITY CORRECTNESS

1) SECCOMP PROFILE CONVERSION

Figure 13 demonstrates the automatic conversion of a seccomp profile to a KRP. For this use case, the Nginx pod with the socket.json seccomp profile was first deployed. Upon receiving the deployment request, the mutating webhook is triggered (A) and checks if a Seccomp profile is applied (B). If the seccomp profile is detected, as in this example, the profile information is extracted to obtain the defaultAction, system call names, and actions. The system then verifies whether the system calls are supported by our system. Since the socket() is supported by our system, the seccomp profile is converted to a KRP and deployed using the Kubernetes API (C). The logs of the KubeRosy operator show entries for detecting the seccomp profile, extracting information, and converting it to a KRP before deployment (D). The results of this experiment demonstrate that our system can seamlessly convert a default seccomp profile into a KRP.

2) FINE-GRAINED SYSCALL FILTERING

This experiment demonstrates that our system goes beyond simple system call filtering by enabling fine-grained filtering based on system call arguments. To showcase this, we deployed the nginx-block policy in the cluster alongside an nginx pod, as shown in Listing 2. This nginx-block policy blocks the fchmodat system call by default for pods labeled with `app: nginx`, but it makes an exception when the path argument is `/benign.txt` and the mode argument is 755 (rwxr-xr-x). After deploying the policy, we accessed the nginx pod and attempted to modify file permissions using the `chmod` command, which invokes the fchmodat system call. Before the system call accesses system resources, it is filtered by our system, which either allows or blocks the call based on the policy. As shown in Fig. 14, only the chmod 755 benign.txt command, which matches the policy exception, is allowed, while all other attempts are blocked.

```
maliciouscontainer$ chmod 777 benign.txt
chmod: changing permissions of 'benign.txt': Operation not permitted
maliciouscontainer$ chmod 777 critical.txt
chmod: changing permissions of 'critical.txt': Operation not permitted
maliciouscontainer$ chmod 755 critical.txt
chmod: changing permissions of 'critical.txt': Operation not permitted
maliciouscontainer$ chmod 755 benign.txt
maliciouscontainer$
```

FIGURE 14. The results of blocking the fchmodat system call except for /benign.txt and mode 755.

VIII. DISCUSSION

The coverage of system calls supported by KRP. Our system operates by detecting the execution of system calls through raw_tracepoint's callback function, storing execution information in the eBPF map, and subsequently enforcing policies at the triggered LSM hook's callback function. This process relies on the mapping between system calls and the corresponding LSM hooks, which requires significant time to implement. Currently, KubeRosy supports approximately 50% of system calls, including high-risk system calls identified in previous research [44] (based on kernel version 5.15). Due to this limited coverage, the system is not yet fully capable of implementing a complete whitelist-based system call filtering framework to enforce the principle of least privilege. To address this limitation, we plan to increase coverage to fully support a whitelist approach.

The integration of syscall policy generation. Currently, our system relies on user-defined KRP to enforce system call security policies. However, in real-world environments, manually creating a minimal list of system calls and their arguments required by an application is labor-intensive. To achieve our ultimate goal of enforcing the principle of least privilege through a system call filtering framework, it is essential to develop a generation component that automatically identifies the system calls and argument values used by containerized applications. To address this challenge, we plan to incorporate a generation component into KubeRosy by drawing on insights from previous research [8], [9], [10], [12], [13], [14]. This will allow us to provide automated and more granular system call security policies, better suited to the needs of KubeRosy.

IX. CONCLUSION

In this work, we have proposed KubeRosy, a dynamic system call filtering framework specifically designed for cloud-native environments. KubeRosy addresses the limitations of traditional static system call filtering mechanisms in containerized environments by leveraging eBPF and LSM hooks. This allows for dynamic modification of system call security policies even while containers are running and enables the enforcement of more granular, argument-based system call security policies. As a result, our system plays a crucial role in preventing malicious exploitation of system calls by containers, thereby enhancing security. In the experiments, we demonstrated that our system is feasible in real-world environments, delivering strong performance with minimal overhead from policy enforcement. This research contributes to the development of more secure cloud-native environments by providing elaborated system call security policy enforcement.

APPENDIX
SYSTEM CALL POLICY SPECIFICATION

KubeRosy is a framework designed to support fine-grained system call security policies for containers in a cluster, satisfying system-level security requirements. Listing. 3 shows the complete schema of KRP.

```
apiVersion: security.kuberosypolicy.com/v1
kind: KubeRosyPolicy

metadata:
  name: [policy name]
  namespace: [namespace name]
spec:
    selector:
        matchLabels:
            [key]: [value]
  - rule:
      action: [Allow|Block]
      syscall:
        name: [syscall name]
        except_args:
          - arg:
              key: [arg key]
              value: [arg value]
status:
  status: [policy status]
  lastUpdated: [last update time]
  - rule:
      action: [Allow|Block]
      syscall:
        name: [syscall name]
        except_args:
          - args:
              key: [arg key]
              value: [arg value]
```

LISTING 3. Schema of KRP.

The beginning of a KRP defines basic information such as `apiVersion`, `kind`, and `metadata`. The `apiVersion` and `kind` are always the same, and `metadata` contains the name and namespace of the KRP. Next, the `spec` consists of a selector, which specifies the Pods to apply the policy to, and a rule, which is the policy to apply to those Pods. The `selector` specifies Pods based on Labels via `matchLables`, which allows for more flexible and efficient policy enforcement in container environments. And a `rule` consists of an `action` (Allow, Block), which is the action the policy should take, and a `syscall`, which is the system call to take the action. The `syscall` consists of the name of the system call and `except_args` to specify exceptions to the policy. The `except_args` consists of a list of arguments containing the key and value to be excluded. This allows user to configure a policy to block a system call and then configure a policy to allow it only for certain argument values.

Lastly, `status` contains information about the status and recent updates to the policy. This allows you to determine if a system call should eventually be blocked or if it should be allowed based on argument values, even if multiple policies are deployed and modified. The status part is not user-defined, but updated by the KubeRosy operator. The KubeRosy operator is aware of the status information of the currently deployed KRPs and updates it during the policy enforcement phase.

REFERENCES

[1] (2023). *State of Financial Services in Cloud*. [Online]. Available: https://cloudsecurityalliance.org/artifacts/state-of-financial-services-in-cloud

[2] (2017). *Docker*. [Online]. Available: https://github.com/docker/go-docker

[3] (2024). *Kubernetes—Open Source System for Automating Deployment, Scaling, and Management of Containerized Applications*. [Online]. Available: https://kubernetes.io/

[4] C. Jelesnianski, M. Ismail, Y. Jang, D. Williams, and C. Min, "Protect the system call, protect (most of) the world with BASTION," in *Proc. 28th ACM Int. Conf. Architectural Support Program. Lang. Operating Syst.*, vol. 27, Mar. 2023, pp. 528–541.

[5] (2024). *Leaky Vessels*. [Online]. Available: https://nvd.nist.gov/vuln/detail/CVE-2024-21626

[6] (2024). *Docker Patches Multiple Vulnerabilities Impacting Runc, Buildkit, and Moby (Leaky Vessels)*. [Online]. Available: https://threatprotect.qualys.com/2024/02/02/docker-patches-multiple-vulnerabilities-impacting-runc-buildkit-and-moby-leaky-vessels/

[7] (2024). *Container Escape: New Vulnerabilities Affecting Docker and Runc*. [Online]. Available: https://www.paloaltonetworks.com/blog/prisma-cloud/leaky-vessels-vulnerabilities-container-escape/

[8] S. Ghavamnia, T. Palit, A. Benameur, and M. Polychronakis, "Confine: Automated system call policy generation for container attack surface reduction," in *Proc. 23rd Int. Symp. Res. Attacks, Intrusions Defenses (RAID)*, 2020, pp. 443–458.

[9] T. Iguni, H. Kamei, and K. Saisho, "Sprofiler: Automatic generating system of container-native system call filtering rules for attack surface reduction," in *Proc. Int. Conf. Comput. Sci. Comput. Intell. (CSCI)*, Dec. 2021, pp. 704–709.

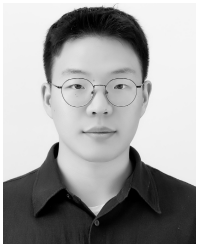
[10] S. Yang, B. B. Kang, and J. Nam, "Optimus: Association-based dynamic system call filtering for container attack surface reduction," *J. Cloud Comput.*, vol. 13, no. 1, p. 71, Mar. 2024.

[11] S. Song, A. Kundu, and B. Tak, "POSTER: Seccomp profiling with dynamic analysis via ChatGPT-assisted test code generation," in *Proc. 19th ACM Asia Conf. Comput. Commun. Secur.* New York, NY, USA: Association for Computing Machinery, Jul. 2024, pp. 1928–1930, doi: 10.1145/3634737.3659426.

[12] D. Zhan, Z. Yu, X. Yu, H. Zhang, and L. Ye, "Shrinking the kernel attack surface through static and dynamic syscall limitation," *IEEE Trans. Services Comput.*, vol. 16, no. 2, pp. 1431–1443, Mar. 2023.

[13] X. Li, Y. Chen, Z. Lin, X. Wang, and J. H. Chen, "Automatic policy generation for inter-service access control of microservices," in *Proc. 30th USENIX Secur. Symp. (USENIX Secur.)*, Aug. 2021, pp. 3971–3988. [Online]. Available: https://www.usenix.org/conference/usenixsecurity21/presentation/li-xing

[14] S. Kim, B. J. Kim, and D. H. Lee, "Prof-GEN: Practical study on system call whitelist generation for container attack surface reduction," in *Proc. IEEE 14th Int. Conf. Cloud Comput. (CLOUD)*, Sep. 2021, pp. 278–287.
[15] (2024). *Falco—Cloud-Native Security Tool Designed for Linux Systems*. [Online]. Available: https://falco.org/
[16] (2024). *Kubearomr—Runtime Kubernetes Security Engine*. [Online]. Available: https://kubearmor.io/
[17] (2024). *Tetragon—EBPF Based Security Observability and Runtime Enforcement*. [Online]. Available: https://tetragon.io/
[18] S. Gwak, T.-P. Doan, and S. Jung, "Container instrumentation and enforcement system for runtime security of kubernetes platform with eBPF," *Intell. Autom. Soft Comput.*, vol. 37, no. 2, pp. 1773–1786, 2023.
[19] L. Lei, J. Sun, K. Sun, C. Shenefiel, R. Ma, Y. Wang, and Q. Li, "Speaker: Split-phase execution of application containers," in *14th Int. Conf. Detection Intrusions Malware, Vulnerability Assessment (DIMVA)*, Bonn, Germany. Cham, Switzerland: Springer, Jul. 2017, pp. 230–251.
[20] W. Findlay, A. Somayaji, and D. Barrera, "BPFBox: Simple precise process confinement with eBPF," in *Proc. ACM SIGSAC Conf. Cloud Comput. Secur. Workshop*, Nov. 2020, pp. 91–103.
[21] (2024). *EBPF—Dynamically Program the Kernel for Efficient Networking, Observability, Tracing, and Security*. [Online]. Available: https://ebpf.io/
[22] (2024). *Linux Security Module*. [Online]. Available: https://www.kernel.org/doc/html/v5.15/admin-guide/LSM/index.html#
[23] (2024). *Virtualization—What is Virtualization?* [Online]. Available: https://www.ibm.com/topics/virtualization
[24] (2023). *Containerization—What is Containerization?* [Online]. Available: https://www.ibm.com/topics/containerization
[25] (2024). *container vs VM—Containers Versus Virtual Machines (VMS): What's the Difference?* [Online]. Available: https://www.ibm.com/think/topics/containers-vs-vms
[26] *EBPF Syscall*. Accessed: Oct. 29, 2024. [Online]. Available: https://docs.kernel.org/userspace-api/ebpf/syscall.html
[27] (2015). *A Seccomp Overview*. [Online]. Available: https://lwn.net/Articles/656307/
[28] *Seccomp BPF (Secure Computing With Filters)*. Accessed: Oct. 29, 2024. [Online]. Available: https://docs.kernel.org/userspace-api/seccomp_filter.html
[29] (2023). *Seccomp Profile—Restrict a Container's Syscalls With Seccomp*. [Online]. Available: https://kubernetes.io/docs/tutorials/security/seccomp/
[30] *LSM Hook Define*. Accessed: Oct. 29, 2024. [Online]. Available: https://elixir.bootlin.com/linux/v5.15/source/security/security.c
[31] (2024). *Apparmor*. [Online]. Available: https://apparmor.net/
[32] (2024). *Selinux*. [Online]. Available: https://selinuxproject.org/page/Main_Page
[33] (2024). *lSM BPF—Kernel Runtime Security Instrumentation*. [Online]. Available: https://lwn.net/Articles/798157/
[34] (2024). *MAC—Mandatory Access Control*. [Online]. Available: https://csrc.nist.gov/glossary/term/mandatory_access_control
[35] Y. Sun, D. Safford, M. Zohar, D. Pendarakis, Z. Gu, and T. Jaeger, "Security namespace: Making Linux security frameworks available to containers," in *Proc. 27th USENIX Secur. Symp.*, 2018, pp. 1423–1439.
[36] (2024). *Containerd*. [Online]. Available: https://github.com/containerd/containerd/tree/main
[37] (2024). *Mutating-Webhook*. [Online]. Available: https://kubernetes.io/docs/reference/access-authn-authz/extensible-admission-controllers/
[38] (2024). *Kubebuilder*. [Online]. Available: https://book.kubebuilder.io/
[39] *Clang: A C Language Family Frontend for LLVM*. Accessed: Oct. 29, 2024. [Online]. Available: https://clang.llvm.org/
[40] (2024). *GO-EBPF*. [Online]. Available: https://github.com/cilium/ebpf
[41] (2024). *GRPC—A High Performance, Open Source Universal RPC Framework*. [Online]. Available: https://grpc.io/
[42] (2024). *AB—Apache HTTP Server Benchmarking Tool*. [Online]. Available: https://httpd.apache.org/docs/2.4/programs/ab.html
[43] (2024). *Memtier_Benchmark: A High Throughput Benchmarking Tool for Redis and Memcached*. [Online]. Available: https://github.com/RedisLabs/memtier_benchmark
[44] S. Ghavamnia, T. Palit, S. Mishra, and M. Polychronakis, "Temporal system call specialization for attack surface reduction," in *Proc. 29th USENIX Secur. Symp.*, 2020, pp. 1749–1766.



JIN HER is currently pursuing the B.S. degree with the Department of Computer Science and Engineering, Incheon National University. His research focuses on automated system call security for advanced cloud-native environments.

CHI HYEON JO is currently pursuing the B.S. degree with the Department of Computer Science and Engineering, Incheon National University. His research focuses on confidential computing and container security.

TAEJUNE PARK received the B.S. degree in computer engineering from Korea Maritime and Ocean University, South Korea, and the M.S. and Ph.D. degrees in information security from KAIST, South Korea. He is an Assistant Professor with the Department of Artificial Intelligence Convergence, Chonnam National University, South Korea. His research interests include network and IoT security and reliable/low-latency communications.

SEUNGSOO LEE received the B.S. degree in computer science from Soongsil University, the M.S. degree in information security from KAIST, and the Ph.D. degree in information security from KAIST, in 2020. He is an Assistant Professor with the Department of Computer Science and Engineering, Incheon National University. His research interest includes developing secure and robust cloud/network systems against potential threats.