Intelligent Automation & Soft Computing

Tech Science Press

DOI: 10.32604/iasc.2023.039565
Article

Container Instrumentation and Enforcement System for Runtime Security of Kubernetes Platform with eBPF

Songi Gwak, Thien-Phuc Doan and Souhwan Jung*

Soongsil University, Seoul, 06978, Korea
*Corresponding Author: Souhwan Jung. Email: souhwanj@ssu.ac.kr
Received: 06 February 2023; Accepted: 14 April 2023; Published: 23 June 2023

Abstract: Containerization is a fundamental component of modern cloud-native infrastructure, and Kubernetes is a prominent platform of container orchestration systems. However, containerization raises significant security concerns due to the nature of sharing a kernel among multiple containers, which can lead to container breakout or privilege escalation. Kubernetes cannot avoid it as well. While various tools, such as container image scanning and configuration checking, can mitigate container workload vulnerabilities, these are not foolproof and cannot guarantee perfect isolation or prevent every active threat in runtime. As such, a policy enforcement solution is required to tackle the problem, and existing solutions based on LSM (Linux Security Module) frameworks may not be adequate for some situations. To address this, we propose an enforcement system based on BPF-LSM, which leverages eBPF (extended Berkeley Packet Filter) technology to provide fine-grained control and dynamic adoption of security policies. In this paper, we compare different LSM implementations to highlight the challenges of current enforcement solutions before detailing the design of our eBPF-based Kubernetes Runtime Instrumentation and Enforcement System (KRSIE). Finally, we evaluate the effectiveness of our system using a real-world scenario, as measuring the performance of a policy enforcement system is a complex task. Our results show that KRSIE can successfully control containers' behaviors using LSM hooks at container runtime, offering improved container security for cloud-native infrastructure.

Keywords: Container; kubernetes; runtime security; eBPF; enforcement

1 Introduction

Containerization has become a vital component of cloud-native infrastructure, providing a lightweight operational environment, supporting application isolation, and minimizing performance overhead compared to the existing VM (Virtual Machine) technology [1]. Kubernetes is widely used among container orchestration systems for effortless deployment and scalability [2]. However, the nature of containerization, which shares a kernel among multiple containers [3], compromises the container security generating privilege escalation and container breakout. As a result, Kubernetes, which is based on containerization technology, cannot avoid the vulnerability as well. To achieve better container security, some ways to protect container vulnerabilities exist, such as container image vulnerability and container misconfiguration scanning tools (Anchore [4], Clair [5], and Trivy [6]) before containers are actually running. Nonetheless, it is impossible to prohibit all active threats in real-world running applications since administrators cannot prevent every unexpected activity caused by malware concealed in source code or inconsiderable RBAC (Role-Based Access Control) configuration errors [7].

Thus, It is critical to protect workloads at runtime, which containers are running in reality, since it is the final layer of protecting containerized applications against dynamic threats. According to the research by Redhat, Among 93% of respondents having experienced at least one incident while operating Kubernetes, 30% of incidents occurred at runtime [8]. Runtime protection, according to the sysdig, can be categorized into two main aspects: enforcement and auditing [7]. Regarding enforcement, it allows administrators to create security policies that specify access rights or resource permissions and to apply it to workloads. These commonly utilize LSM implementations such as AppArmor [9], SELinux [10,11], etc. As a result, users can prevent processes breaking out of the container process in runtime and having over-privileged permissions from accessing to other processes' resource in kernel-level. Thus, we can protect the system even though Kubernetes workloads are compromised by vulnerabilities.

However, existing LSM solutions cannot dynamically apply security policies at container runtime. In addition, each solution only has its usage with specific objects such as files, capabilities, and access rights. To the best of our knowledge, no solution provides fine-grained control with the parameter of LSM functions dynamically. We compare various LSM solutions, emphasizing the advantages of eBPF-based LSM implementation (KRSI (Kernel Runtime Security Instrumentation), now BPF-LSM [12]) in Section 2.1.1. On the one hand, since eBPF is the most appropriate technology for cloud-native runtime security, CNCF (Cloud Native Computing Foundation) releases security solutions using eBPF, such as Cillium [13], Tetragon [14], and KubeArmor [15]. Though they provide useful functionalities for container security, there are still differences in hooking points for eBPF programs and security targets. We highlight the differences between existing open-source security tools in Section 2.2.1. To address container runtime security with enforcement using eBPF to solve the abovementioned problems, we adapt KRSI to implement KRSIE (Kubernetes Runtime Security Instrumentation and Enforcement). Our security policy enforcement system can provide dynamic adaption and fine-grained control and apply security policies to an individual Kubernetes workload. To evaluate the ability of KRSIE, we set a test scenario that cannot be solved by existing LSM implementations since there are no proper tools to measure the effectiveness of the security policy enforcement tool.

Our key contributions can be summarized as follows:

- We implement KRSIE, which can apply security policies to Kubernetes workloads at runtime without redeployment of the container. Besides our implementation, AppArmor and SELinux seem to apply a security policy to a running container if container solutions support it, though those solutions have to restart the container to apply the security policy. However, according to the report, container restart or redeployment to fix the minor issue is impractical in real cloud environments due to performance issues [16]. Our solution can deal with this problem by providing the policy's runtime application.

- Our solution supports fine-grained control by directly handling the LSM function's parameters, which differs from the existing methods' ability. In the case of AppArmor and SELinux, since they have strict use cases when used in userland, it is hard to control specific kernel objects. Aside from that, KubeArmor, which is the most similar to our system, supports the abstraction of a policy, but its abstraction level is pretty high, so it cannot give users fine-grained control with a variety of LSM function parameters as well. KRSIE allows users to make policies with the actual parameters of LSM functions.

The rest of the paper is organized as follows. Section 2 gives background on LSM and eBPF technologies and comparisons between LSM implementations and eBPF-based cloud security solutions. In Section 3, we describe the architecture of the proposed system. Section 4 gives the result of experiments to test our system to check if our system works in real-world applications properly. Finally, in Section 5, we conclude by explaining the value of this paper and discussing future work.

2 Background

This section explains the essential background of the paper, such as LSM and eBPF, and discusses various LSM solutions implemented with the LSM framework, such as AppArmor, SELinux, and KRSI. Moreover, we also compare eBPF-based cloud-native security solutions such as Cilium, Tetragon, and KubeArmor to emphasize the effectiveness and difference among numerous tools.

2.1 LSM (Linux Security Module)

LSM is a critical security framework that allows for operating environments with security modules in the Linux kernel. By placing hooks in the kernel code, LSM supplements the inadequate security checking of Linux's DAC (Discretionary Access Control). These hooks enable the LSM module to mediate access to kernel objects just before they are accessed, where the security function defined by the administrator is checked [17]. The result is then returned as yes or no, depending on the outcome of the security function. The use of LSM offers several advantages, including enhanced access control and improved security in the kernel. This makes LSM a popular choice for many existing access control implementations, such as AppArmor [9], SELinux [10,11], and Tomoyo [18].

2.1.1 LSM Implementations

AppArmor allows an administrator to restrict a program's capabilities with a profile consisting of file paths and capabilities such as network and socket access with specific actions like read, write, and execute [9]. For example, to allow users to read all files in the "/some/random/example" directory, an administrator specifies the condition "/some/random/example/$*$ r" in the AppArmor profile. **SELinux** (Security-Enhanced Linux) is another LSM implementation that allows users to control access permissions for the system [10,11]. It defines the access level of applications, processes, and files as security policies. In this mechanism, every file or process has an SELinux label, which controls policies by using the label of every object. The label consists of the following format: user:role:type:level. Whenever a subject, such as a process, makes an access request for an object, such as a file, it checks a privilege of the subject and object, then, permission is granted or denied.

The two technologies mentioned above are kernel built-in security frameworks. Since they work directly in the kernel and can dereference the pointer of functions' paramters, these tools are efficient in execution. Moreover, because of their fixed usages (AppArmor's path-based access control, SELinux's label-based access control), they are relatively easy to apply to the system. However, these frameworks cannot load and change security policies dynamically due to the nature of LSMs, which leads to difficulty in adoption. Whenever minor changes happen in their policies, containers should be restarted to reflect the small changes, which might not proper to the cloud environment.

KRSI is an LSM implementation that allows users to attach modular BPF programs to various LSM hooks and inject errors to block unwanted operations [12]. It enables users to define their own MAC policies with arbitrary code. To use KRSI in the system, users must code a BPF program with LSM hooks and execute the program in the kernel. The workflow of KRSI is shown in Fig. 1. Since KRSI is also an LSM implementation, it has the advantages of LSM solutions, such as efficiency and stability in performing security functions. Moreover, it has additional advantages thanks to the eBPF nature. KRSI loads and changes policies dynamically, providing finer-grained control in comparison to other LSM implementations. However, this solution requires administrators to understand deep kernel knowledge to program eBPF code, making it more challenging to adopt in systems. Overall, each of these technologies has its advantages and disadvantages. AppArmor and SELinux are efficient and easy to use, but their fixed usage may limit control. KRSI provides finer-grained control and dynamically loads policies, but requires expertise in programming eBPF code.

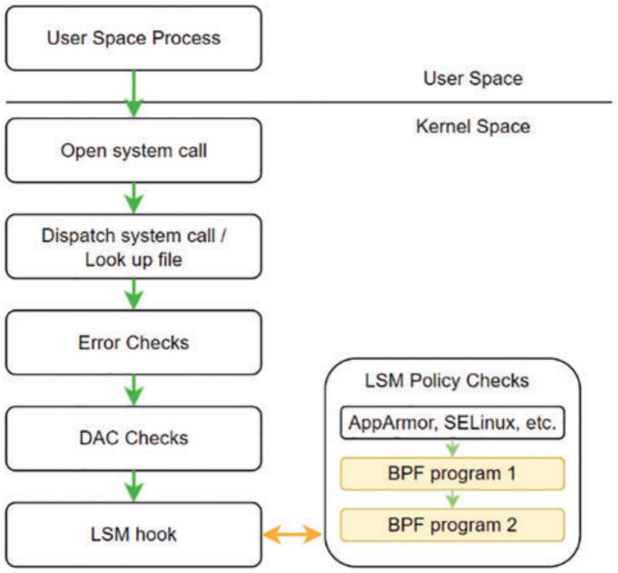


Figure 1: The workflow of KRSI

2.1.2 Seccomp-Bpf

Seccomp-bpf is a kernel feature in Linux that enables secure computing by restricting the available behaviors within a workload [19]. This is achieved by reducing available system calls through a filtering function for incoming system calls. The functionality of seccomp-bpf is provided by BPF, which ensures efficient system call filtering without adding any extra functionality. It is worth noting that seccomp-bpf is not an LSM implementation and is unable to enforce specific controls on the kernel.

2.2 eBPF Security Solutions

The Berkeley Packet Filter is a technology used in specific programs which need to analyze and filter network traffic. eBPF (extended BPF, which is now called BPF) is a technology that runs sandboxed programs in an OS kernel [20]. This technology extends the kernel's capabilities safely and efficiently while requiring no kernel source code changing or kernel modules loading. When developers write eBPF programs, these are compiled into eBPF bytecode programs. These programs are going through eBPF verifier to check if there are some security-critical problems to be executed. Then these are compiled into native kernel code using JIT compiler. Finally, these programs are attached to the defined hooks. Whenever hooks are called, the event is stored in the Maps and accessed from user space programs. This procedure is shown in Fig. 2. eBPF is event-driven and runs when a kernel passes a particular hook points such as syscall, function entry/exit, tracepoints, and etc.

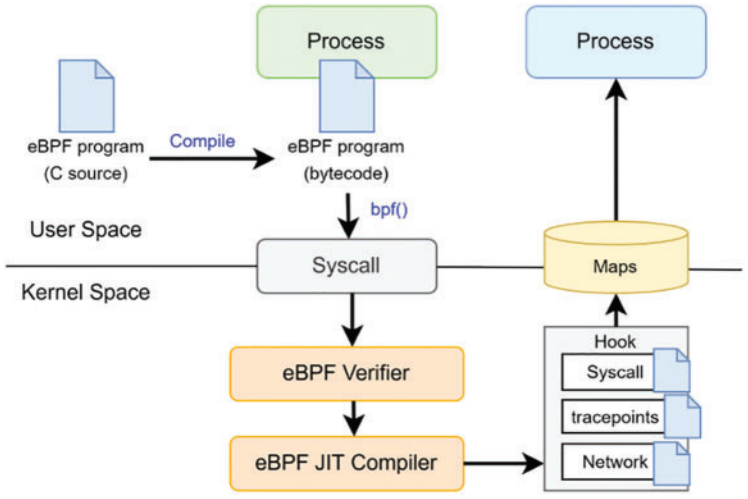


Figure 2: The workflow of eBPF program

CNCF has several projects that focus on security features with the use of eBPF. One of the most prominent open-source projects is **Cilium**, which provides networking and security solutions for containerized workloads [13]. Cilium uses eBPF technology to create network policies that ensure network security and supports traffic encryption to protect the packets of workloads. With the help of XDP (eXpress Data Path), it improves network throughput and reduces packet processing procedures [21]. Cilium is focused more on networking and network security than on workloads security in runtime, so it uses different hooking points for the eBPF program, and it does not use LSM hooking points.

Another open-source project under Cilium Enterprise is **Tetragon**, which provides eBPF-based observability and runtime enforcement for security [14]. Tetragon utilizes various hooking points, such as system calls, tracepoints, kprobes, and uprobes, to give users system security. However, it does not use LSM hooks for their implementation, which may make them vulnerable to TOCTOU (Time-of-Check to Time-of-Use) attacks when using system calls [14,22].

KubeArmor is another runtime security enforcement system that restricts the behavior of cloud workloads [15]. It uses LSM such as AppArmor, SELinux, or KRSI to help users create policies that restrict access to various resources like process, file, network, capabilities, and system calls. However, KubeArmor's security policy is more abstract compared to our system, and it does not allow users to access kernel contexts such as bprm structure. Our system provides users with the possibility to access LSM function's parameters directly, which helps them create fine-grained security policies.

We will take advantage of using LSM to provide the fast and secure execution in the kernel side and using eBPF to adopt security policies dynamically with in-depth and rich kernel contexts.

3 Proposed Design

The detailed architecture of the system is shown in Fig. 3. The system consists of four main parts, which are indicated in gray boxes: **Policy Generation**, **Policy Enforcement**, **Monitoring**, and **Policy Augmentation**. In the next section, we describe each part's role.

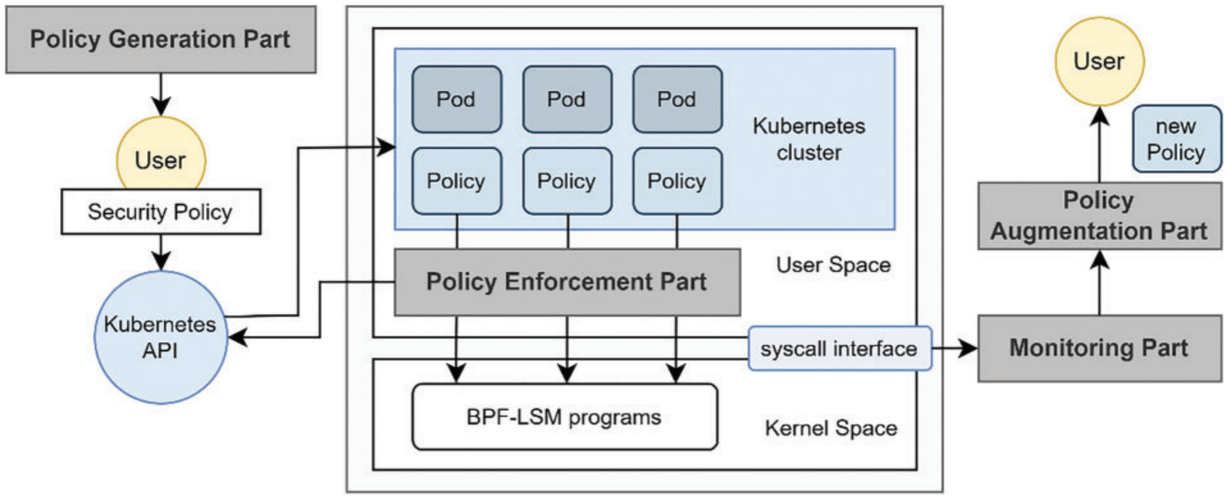


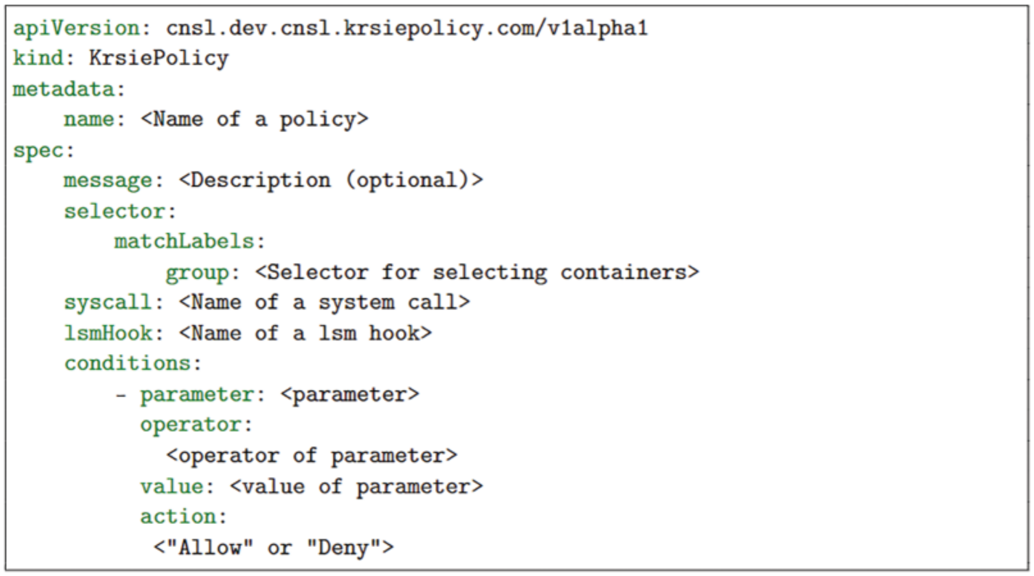
Figure 3: The architecture of the proposed system

3.1 *Policy Generation*

The policy Generation Part is responsible for creating a Kubernetes custom resource to provide accessible programming of BPF-LSM policies to users. By doing so, users do not need to code their policies in a complicated and restricted C language, which needs an understanding of low-level kernel knowledge. Through our system, users only are required to provide necessary information such as the container's label to distinguish the container that users want to control from others, the name of the syscall and lsm hook they want to control, and conditions. This part gives users a simple abstraction to code policies in a YAML file. The policy format is shown in Fig. 4.

3.2 *Policy Enforcement*

The Policy Enforcement Part parses security policies, one of the Kubernetes resources created by the policy generation part, and creates real BPF-LSM programs with parsed information that will be executed in the kernel. This part consists of two main parts: (1) **Manager**: this part parses the information of Kubernetes resources such as Cluster, Node, container, and KRSIE policies. After parsing all information from the Kubernetes API server, it sends information to the enforcer. (2) **Enforcer**: this part receives the necessary information to make actual policies from the manager and creates real BPF-LSM programs. Finally, this executes BPF-LSM programs in the kernel.



```
apiVersion: cnsl.dev.cnsl.krsiepolicy.com/v1alpha1
kind: KrsiePolicy
metadata:
    name: <Name of a policy>
spec:
    message: <Description (optional)>
    selector:
        matchLabels:
            group: <Selector for selecting containers>
    syscall: <Name of a system call>
    lsmHook: <Name of a lsm hook>
    conditions:
        - parameter: <parameter>
          operator:
            <operator of parameter>
          value: <value of parameter>
          action:
           <"Allow" or "Deny">
```

Figure 4: The format of a policy

3.2.1 Analysis on the LSM Hook Parameters

Before we analyze the parameters of LSM hooks, we choose a list of system calls used to extract parameters. Because there are many system calls in the kernel (390 system calls in the kernel version 5.12.11 [23], which is used for our implementation), we cannot provide BPF template programs for every system call. Thus, we list dangerous system calls, including equivalent system calls of each system call for command execution, permission, and network [24]. Then we analyze the parameters of LSM hooks that a system call has. We first look at the system call table to check the system call number and name. Then we match the system call's name and path to get the header information. With the system call header, we finally find the kernel function and LSM parameters. This analysis is performed until the end of all callee functions in a system call. The list of system calls and corresponding LSM hooks is shown in the Table 1. The label located in the left side (CMD Execution, Permission, and Network) indicates the type of a system call [24], and we align system calls which have the same LSM hooks and parameters.

Table 1: The list of dangerous system calls

		LSM hook	Parameters
CMD Execution	clone, fork	security_vm_enough _memory_mm	[mm][pages]
		security_task_alloc	[task] [clone_flags]
		security_task_free	[task]
	execve, execveat	security_bprm_check	[bprm]
		security_bprm_creds_for_exec	[bprm]
	ptrace	security_ptrace_traceme	NONE
		security_ptrace_access_check	[mode]

(Continued)

Table 1: Continued

		LSM hook	Parameters
Permission	chmod, fchmod, fchmodat	security_path_chmod	[path] [mode]
	mprotect, pkey_mprotect	security_file_mprotect	[vma] [reqprot] [prot]
	setgid	security_task_fix_setgid	[new_cred] [old_cred] [flags]
	setreuid, setuid	security_task_fix_setuid	[new_cred] [old_cred] [flags]
Network	accept4, accept	security_socket_accept	[sock]
	bind	security_socket_bind	[sock][address][addrlen]
	connect	security_socket_connect	[sock][address][addrlen]
	listen	security_socket_listen	[sock][backlog]
	recvfrom	security_socket_recvmsg	[sock][msg] [flags]
	socket	security_socket_create	[family][type][protocol][kern]
		security_socket _post_create	[sock][family][type] [protocol][kern]

3.2.2 The Workflow of BPF-LSM Program

BPF-LSM program is a program executed in the kernel with parsed information from the manager from the enforcement part. The BPF-LSM program follows the LSM framework rule: checking the condition and returning the result. 0 means pass (Allow), or error code means not pass (Deny).

We implement a BPF-LSM program per LSM functions, and we have 18 BPF-LSM programs working as templates. Each BPF-LSM program should follow the order: before it is attached to a particular LSM hook, it imports necessary libraries and creates data structures and events that will be collected from the kernel. Once the program is attached to the specific LSM hook, it follows **(1) Filtering part**: filters matched containers to control the behaviors. **(2) Kernel context pre-processing**: gets a kernel context observed only in the kernel. The BPF-LSM program is supposed to pre-process the necessary contexts for the corresponding LSM hook due to resource problems. **(3) Condition phase**: parses conditions the policy needs. The conditions will decide whether the system call is safe to process. **(4) Policy action**: checks security with an action. The policy will give back the results of the LSM function to the program. If the action is "Allow", the return value should be 0—otherwise, −1. Policies in this system should be stackable to the old policies if the policy affects the same container. The result is calculated by the OR operation of all relevant LSM hook results of the same function. The example of the BPF-LSM program is shown in Fig. 5. It is one of the BPF-LSM templates for the "socket_accept" LSM function. To utilize the BPF-LSM, the program has to declare the function with "LSM_PROBE" and the hook function's actual parameters. The following procedures are as described above. Note that the conditioning phase is executed just before checking the policy action. The parsed information replaces the branch's condition, and policy action will be executed.

```
/* Attach to the "socket_accept" LSM hook */
LSM_PROBE(socket_accept, struct socket *sock, struct socket *newsock) {
    /* Filtering Part */
    if(!container_should_be_filtered(MOUNT_NS_ID))
        return 0;

    /* Kernel Context Pre-processing*/
    bpf_get_current_comm(&data.comm, sizeof(data.comm));
    data.socket_state = sock->state;
    data.socket_type = sock->type;
    data.socket_flags = sock->flags;

    /* Condition Phase */
    /* Policy Action */
    if(CONDITIONS) {
        return ACTION;
    }
    return INVERSE;
}
```

Figure 5: The example of BPF-LSM program

3.3 Monitoring

The Monitoring Part collects all events from pods in the Kubernetes cluster. For this part, we set up the eBPF-based open-source monitoring tool of the CNCF, which is tracee-ebpf [25]. This tool monitors the behaviors of syscall, processes, and sockets by being attached to kprobes, raw tracepoints, and so on. This tool collects all kernel events from the pods. The monitoring part is running as a docker container, and it writes the container's logging messages to syslog, which is one of the standard interfaces of the UNIX system to process log messages. Since monitoring technology with eBPF is already mature, we choose to set up the monitoring system using an existing solution.

3.4 Policy Augmentation

This part helps users expand their policies if related events occur. It uses the logged events from the monitoring part. Once the monitoring part creates the logs about all events related to LSM hooks, the augmentation part receives collected events from the syslog and filters conditions. There are two conditions for creating and recommending a new policy to the users. (1) In case the same event type exists in the current policies and collected events. For example, if the system wants to monitor socket accept action, then any containers occurring the action will be targets. (2) If there are the same containers monitored by policies, the augmenter also makes a policy and recommends it. For instance, if a monitored container occurs a new event, it will be a target. The workflow is shown in Fig. 6.

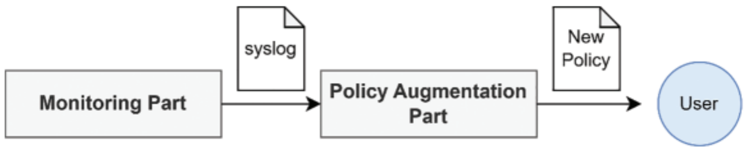


Figure 6: The workflow of policy augmentation part

4 Experiments

4.1 Experiments and Analysis on the LSM Hook Parameters

In this section, we experiment with simple scenarios to test if KRSIE system's functionalities work properly. First, we implement applications that are dedicated to executing a specific system call (chmod, setuid, bind, and etc) and deploy them as a pod. Then we make policies for these applications and deploy them as a Kubernetes resource, KRSIE policy. Now once the system starts, the enforcement part parses these pods and policies information from the Kubernetes API server and creates BPF-LSM programs. Finally, it executes BPF-LSM programs in the kernel. Each BPF-LSM program is applied to the corresponding pod.

The result is shown in Table 2. For most applications, our system works correctly with a pair of LSM hooks and conditions. In the case of "chmod" raw in Table 2. We test "path_chmod" LSM hook of the "chmod" system call, and the action is "deny" when the condition is "data.newmode==511" (data is an event structure, and "newmode" is umode_t structure's value that we pre-define to collect from the kernel). Thus, if a specified container tries to change the user mode with the "chmod" system call with the value "511", it will be blocked at runtime. As a result, it succeeds in controlling the container's behavior. However, the only failure is for the LSM hook "vm_enough_memory_mm." This might be because it has an original security function in the kernel. Integration or overriding will be further work.

Table 2: Experimental result

System call	LSM hook	Condition	Action	Result
clone	vm_enough_memory_mm	"data.task_size>=100"	deny	fail
	task_alloc	"data.clone_flags== 0x00000100"	deny	success
	task_free	"data.parent_tid==1000"	allow	success
execve	bprm_check	"data.task_secid==1"	allow	success
	bprm_creds_for_exec	"data.new_cred_uid==1"	deny	success
ptrace	ptrace_traceme	"data.ppid<1000"	allow	success
	ptrace_access_check	"data.child_pid<1000"	allow	success
chmod	path_chmod	"data.newmode==511"	deny	success
mprotect	file_mprotect	"reqprot==PROT_READ"	allow	success
setuid	task_fix_setuid	"data.new_uid==1000"	deny	success
setgid	task_fix_setgid	"data.new_gid==100"	deny	success
accept	socket_accept	"type==AF_LOCAL"	deny	success
bind	socket_bind	"data.port==8080"	deny	success
connect	socket_connect	"sock->type==SOCK_RAW"	deny	success
listen	socket_listen	"data.local_port=8888"	deny	success
recvfrom	socket_recfrom	"sock->type==SOCK_RAW"	deny	success
socket	socket_create	"type==AF_LOCAL"	deny	success
	socket_post_create	"type==AF_LOCAL"	deny	success

4.2 A Solution to Real-World Problem

In this section, we describe a specific attack and defense scenario. Because there are no similar solutions to adjust the container's behavior at runtime with LSM hook parameters, it is hard to test the performance of the proposed system. Instead, we point out the current runtime threat due to industrial habits and utilize our tool to handle the threat at runtime. According to the report from sysdig, real-productions do not stop containers even though they find risky configurations or vulnerabilities at runtime since they do not want to slow their deployment [16]. In this case, it seems industries will not use AppArmor and SELinux for policy enforcement because those require the container to stop to apply their policy. On the other hand, if the system utilizes our solution, they do not need to stop their workload but only make a new policy and apply it to the container.

For the sake of the reality, we adopt a prominent attack against cloud environment. Cryptomining attacks have been serious in Kubernetes environment since the development of blockchain and cryptocurrencies technology [26]. We assume the attacker has the ability to intrude on the host system to deploy Kubernetes pods for mining, especially Monero cryptocurrency. Once the attacker deploys a mining pod with his Monero account and pool, the pool allocates a mining task to the attacker operating the mining pod to solve the task. The workflow of the attack scenario is as shown in Fig. 7.

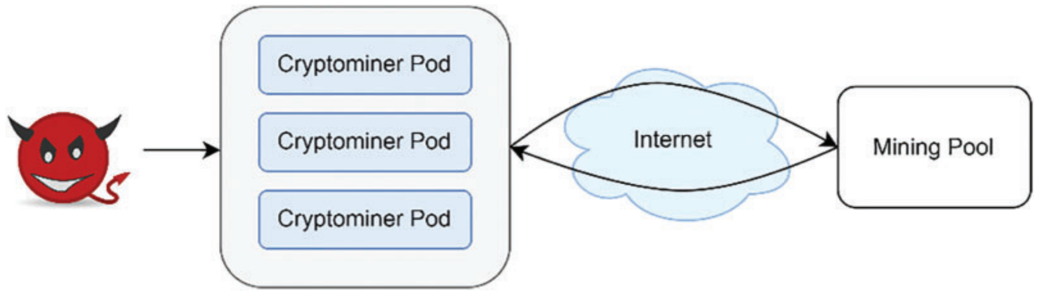
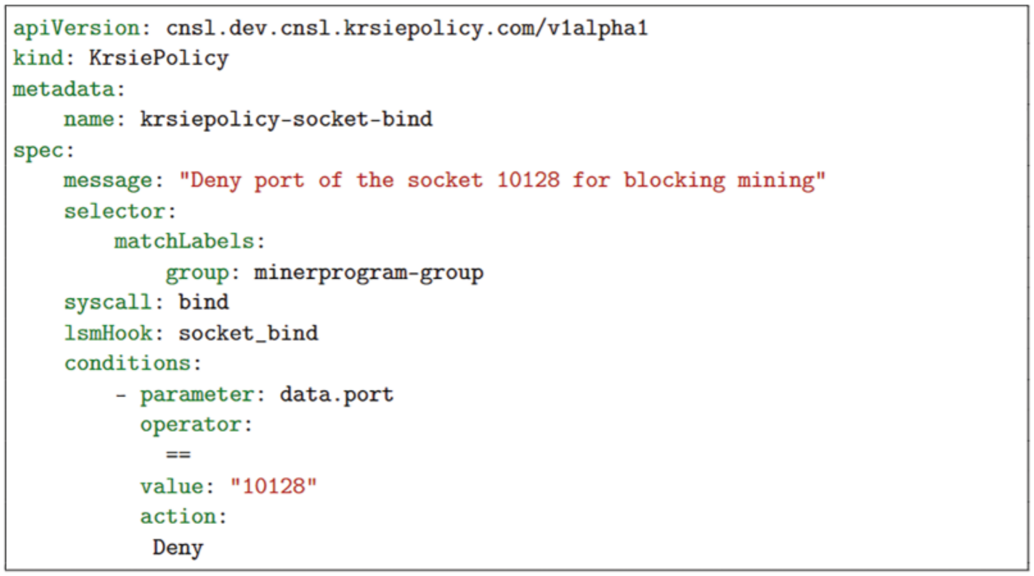


Figure 7: The workflow of the real world problem

There are several ways to detect the attack. The system administrator gets a report that the system is slow, or CPU resources lack, for example. The admin, now, investigates the process with the monitoring part of our system and reveals the process is a container and it has an outbound connection whose server name is well known for a mining pool. Before the admin finds clear evidence of intrusion and removes the container entirely from the host system (or he might not want to remove it entirely for system availability), he wants to stop the behavior first. To do that, the admin configures a policy with proper conditions: LSM hook, socket_bind, and conditions, port, "10128", and deny. The policy is shown in the Fig. 8. As a result, our system can prevent attacker's mining with the policy successfully without container.



```
apiVersion: cnsl.dev.cnsl.krsiepolicy.com/v1alpha1
kind: KrsiePolicy
metadata:
    name: krsiepolicy-socket-bind
spec:
    message: "Deny port of the socket 10128 for blocking mining"
    selector:
        matchLabels:
            group: minerprogram-group
    syscall: bind
    lsmHook: socket_bind
    conditions:
        - parameter: data.port
          operator:
            ==
          value: "10128"
          action:
           Deny
```

Figure 8: A policy example of preventing socket bind action

5 Conclusion

In this paper, we design and implement an eBPF-based Kubernetes Runtime Security Instrumentation and Enforcement System (KRSIE) to handle current container runtime security problems. To describe the efficiency and security of using LSM frameworks and eBPF, we compare different LSM solutions and various eBPF-based cloud-native security tools. Then we depict the design of KRSIE and how it works. With our system, users can apply container security policies dynamically at runtime to an individual Kubernetes workload. It provides fine-grained control, allowing users to utilize the parameters of LSM functions to control container behavior. Finally, to evaluate the system's ability, we create a scenario and test a real-world vulnerability that other security solutions cannot solve.

Even though our system works properly, as we proposed at the beginning, it has limitations. Firstly, it cannot solve a case if LSM security functions are duplicated. Integration of new security policies with existing security functions will be future work. Aside from that, since we implement BPF-LSM programs about the list of security-critical system calls and those LSM functions, the system cannot cover all real vulnerable cases. Further implementation will be future work as well.

Acknowledgement: The authors appreciate reviewers and editors for taking the time and effort necessary to help improve the quality of the manuscript.

Funding Statement: This work was partly supported by the Institute of Information & Communications Technology Planning & Evaluation (IITP) grant funded by the Korea Government (MSIT), (No. 2020-0-00952, Development of 5G edge security technology for ensuring 5G+ service stability and availability, 50%) and the Institute of Information and Communications Technology Planning and Evaluation (IITP) grant funded by the MSIT (Ministry of Science and ICT), Korea (No. IITP-2023-2020-0-01602, ITRC (Information Technology Research Center) support program, 50%).

Conflicts of Interest: The authors declare that they have no conflicts of interest to report regarding the present study.

References

[1] W. Felter, A. Ferreira, R. Rajamony and J. Rubio, “An updated performance comparison of virtual machines and linux containers,” in *Proc. of IEEE Int. Symp. on Performance Analysis of Systems and Software*, Philadelphia, PA, USA, pp. 171–172, 2015.
[2] Kubernetes, “Kubernetes (K8s),” 2016. [Online]. Available: https://github.com/kubernetes/kubernetes
[3] R. Dua, A. R. Raja and D. Kakadia, “Virtualization vs containerization to support PaaS,” in *Proc. of IEEE Int. Conf. on Cloud Engineering*, Boston, MA, USA, pp. 610–614, 2014.
[4] Anchore, “Github action for vulnerability scanning,” 2019. [Online]. Available: https://github.com/anchore/scan-action
[5] Quay, “Clair,” 2015. [Online]. Available: https://github.com/quay/clair
[6] Aquasecurity, “Trivy,” 2019. [Online]. Available: https://github.com/aquasecurity/trivy
[7] Sysdig, “Kubernetes security: Managing runtime security threats in kubernetes,” 2022. [Onlune]. Available: https://sysdig.com/learn-cloud-native/kubernetes-security/runtime-security/
[8] RedHat, “The state of kubernetes security in 2022,” 2022. [Online]. Available: https://www.redhat.com/rhdc/managed-files/cl-state-of-kubernetes-security-report-2022-ebook-f31209-202205-en.pdf
[9] M. Bauer, “Paranoid penguin: An introduction to novell AppArmor,” 2006. [Online]. Avaiable: https://dl.acm.org/doi/fullHtml/10.5555/1149826.1149839
[10] RedHat, “What is SELinux?,” 2019. [Online]. Available: https://www.redhat.com/en/topics/linux/what-is-selinux
[11] S. Smalley, C. Vance and W. Salamon, “Implementing SELinux as a linux security module,” 2001. [Online]. Available: http://www.cs.unibo.it/~sacerdot/doc/so/slm/selinux-module.pdf
[12] B. Wilson, “Mitigating attacks on a supercomputer with KRSI, SANS,” 2020. [Online]. Available: https://sansorg.egnyte.com/dl/GX95bRNP5t
[13] Cilium, “Cilium,” 2017. [Online]. Available: https://github.com/cilium/cilium
[14] Isovalent, “Tetragon–eBPF-based security observability & runtime enforcement,” 2022. [Online]. Available: https://isovalent.com/blog/post/2022-05-16-tetragon/
[15] Kubearmor, “KubeArmor,” 2021. [Online]. Available: https://github.com/kubearmor/KubeArmor
[16] Sysdig, “2023 Cloud-native security & usage report,” 2023. [Online]. Available: https://dig.sysdig.com/c/pf-2023-cloud-native-security-and-usage-report?x=u_wfri
[17] C. Wright, C. Cowan, S. Smalley, J. Morris and G. Kroah-Hartman, “Linux security modules: General security support for the linux kernel,” in *Proc. of 11th USENIX Security Symp.*, San Francisco, CA, USA, pp. 17–31, 2002.
[18] T. Handa, “TOMOYO linux,” 2009. [Online]. Available: https://lwn.net/Articles/313346/
[19] freedesktop.org, “Seccomp BPF (SECure COMPuting with filters),” 2017. [Online]. Available: https://dri.freedesktop.org/docs/drm/userspace-api/seccomp_filter.html
[20] eBPF, “What is eBPF?,” 2022. [Online]. Available: https://ebpf.io/what-is-ebpf/
[21] M. Bélair, S. Laniepce and J. M. Menaud, “Leveraging kernel security mechanisms to improve container security: A survey,” in *Proc. of ARES ‘19: 14th Int. Conf. on Availability, Reliability and Security*, Canterbury, CA, United Kingdom, pp. 1–6, 2019.
[22] Rexguowork. “Phantom attack: Syscall monitoring/tracing bypass,” 2021. [Online]. Available: https://github.com/rexguowork/phantom-attack
[23] Bootlin, “Syscall_64.tbl-linux source code (v5.12.11),” 2021. [Online]. Available: https://elixir.bootlin.com/linux/v5.12.11/source/arch/x86/entry/syscalls/syscall_64.tbl
[24] S. Ghavamnia, T. Palit, S. Mishra and M. Polychronakis, “Temporal system call specialization for attack surface reduction,” in *Proc. of 29th USENIX Security Symp.*, Boston, MA, USA, pp. 1749–1766, 2020.

[25] Aqua, “Tracee: Runtime security and forensics using eBPF,” 2022. [Online]. Available: https://aquasecurity.github.io/tracee/v0.6.4/
[26] Sysdig, “Detecting cryptomining attacks “in the wild”,” 2022. [Online]. Available: https://sysdig.com/blog/detecting-cryptomining-attacks-in-the-wild/