Using Modern Linux Security Modules for Enhancing Container Security

Haitao Hu[*]
huht2022@shanghaitech.edu.cn
ShanghaiTech University
Shanghai, China

Guowei Yang[*]
yanggw2022@shanghaitech.edu.cn
ShanghaiTech University
Shanghai, China

ABSTRACT

Containers have become a cornerstone of modern cloud services and applications, prized for their lightweight and isolated runtime environments. However, a container is not inherently a secure sandbox in essence. Containers were designed to share the same underlying operating system kernel, which inherently introduces new attack surfaces for workloads residing within these containers. Moreover, there are specific security concerns unique to containerized environments, including Kubernetes security issues and the potential for container escape. This paper presents an investigation into the vArmor project, a cloud-native sandbox system that enhances Linux container with Linux security modules (LSM). We delve into the vArmor project and explore the solution for enhancing container security, examining its built-in security rules and extending its capabilities by proposing new rules for defensing other threats. Our study includes a rigorous evaluation of the effectiveness and performance impact of these security implementations within vArmor. The results demonstrate that AppArmor based rules have minimal impact on the overall system, while BPF based rules have a slightly larger impact. The builtin rules of vArmor are able to defend common threats, while special threats can be defended by new targeted rules that are easily drafted.

1 INTRODUCTION

Many of today's services and applications run in containers, which provide lightweight isolated runtime environments. Compared the isolation provided by virtual machine, instead of emulating a new guest kernel with the hardware virtualization, containers directly share the host kernel and hardware resources. While shared kernel is less secure. For instance, an untrusted process within a container may exploit a kernel vulnerability to directly affects the host system, which is known as *container escapes*. In other words, containers are not a sandbox. Using containers to run untrusted or potentially malicious code without additional isolation is not a good idea.

To mitigate security problems of containers, the additional security hardening is necessary. The sandboxing technology is a typical solution which puts user application in a totally isolated environment. According to requirements of isolation for user applications, the available schemes can be grouped into *strong* and *weak* isolation. The former often puts user applications in an complete virtual machine (VM) which maintains an own kernel. Similarly, the latter also have to maintain a VM, but it takes advantage of the host kernel's functionality in an appropriate way. For example, the QEMU [8] is a hypervisor that can run an arbitrary guest OS for building strong isolation, then containers run in the guest OS. The gVisor [4] is *application sandbox* has weak isolation. It intercepts application system calls and acts as the guest OS, without the need for translation through virtualized hardware. These two schemes have their own features, and which one to use depends on balancing security and performance needs.

These two above sandbox technologies are both complex and requires overhead for mocking functions of the kernel. As we know, a container is a user-process on the host operating system, and they can be confined by setting security contexts specified by the kernel. There are some sandboxes [3, 6] are built upon the capabilities provided operating system. For example, we can limiting the system resources for a container such that file system and networking. This way the container can continue to share the host kernel and reduce the attack surface.

Fortunately, the majority of *container runtime* tools have combined several native security interfaces provided by the operating system. Users can configure the appropriate security profile for their workloads. These security profiles often are applied in initialization of containers, and works for the whole lifecycle of the workloads. However, it is not a trivial work to manage profiles for hundreds and thousands of containers in a real production environment. For instance, the profile creation suffers from the problem that different applications require different security profiles. Furthermore, the profiles are supposed to updating without restarting or stopping the running container.

In this paper, we follows the *vArmor* [7] project and explores the solution for enhancing container security. vArmor is a cloud-native sandbox system that enhances Linux container with *Linux security modules* support. vArmor provides a number of builtin security-related rules implemented by AppArmor [11] or BPF [13] LSM. Moreover, vArmor supports to quickly construct a customized BPF profile to achieved more fine-grained security controls for special workloads. vAromor applies agent-manager architecture in the cluster, where the agent is responsible for deploying the profiles the manager is responsible for distributing them.

We will analyze the builtin rules of vArmor and contribute new rules for defensing other threats. The threats in the container are often from local applications or remote communications. For example, the malware running in the container may abuse system calls through some operations to gain control of the host, steal host information, or cause damage on the host. Furthermore, recent work points out that under cloud native systems, the entire distributed system may also become the target of attacks, such as further attacks on the kubernates cluster after gaining control of the host through container escape. As a result, we mainly focus following aspects of container security.

[*]Both authors contributed equally to this research.

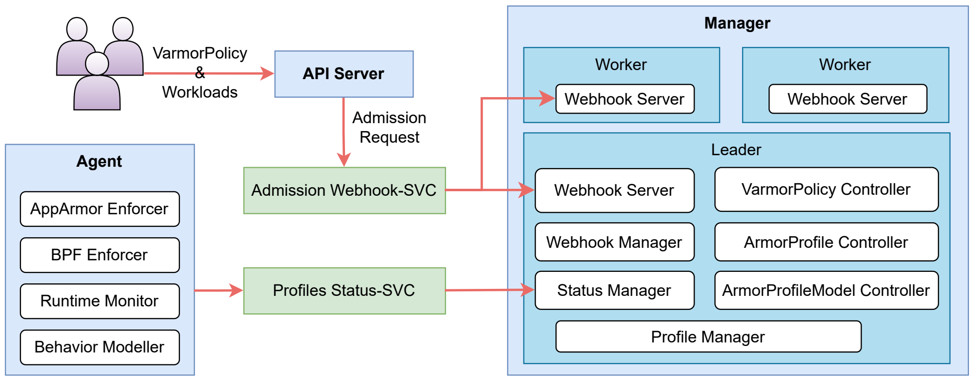

Figure 1: Overall architecture of the vARMOR. It mainly consists of two components: (1) manager to maintain and update the global status and security profiles, agent to monitor and protect specific containers on each node.

(1) **Container escape**: an attacker can start a process out of scope of the container.
(2) **Escalate privileges**: an attacker can start a process with elevated privileges (root) that is not allowed in the container.
(3) **Laterally move**: an attacker can run a process within another container among current system.

The validation and performance of security implementation are most concerned aspects of a security system. In this paper, we will evaluate the validation of builtin rules of vArmor by artificially triggering some security events. The security events includes synthetic malware sample and the exploitation of CVEs. For overhead evaluation, we utilize the benchmark suite *byte-unixbench* [1] for testing performance BPF LSM based rules. The performance testing of AppArmor can refer to results [5] from Linux community.

2 RELATED WORK

The growing adoption of cloud-native systems has put container security in the spotlight. Leveraging Linux Security Modules (LSM), several frameworks [17] [12] [16] have been developed to monitor and restrict container behavior, ensuring robust isolation from host system resources.

Some researchers [9] [15] [18] argue that simple policies are insufficient to model entire runtime behaviors. They advocate for static analysis to examine application code, deduce the system calls that legitimate programs might use, and then generate BPF policies to limit other system calls. The primary aim is to minimize the operational overhead from monitoring and restricting system calls while enforcing stricter constraints to prevent unexpected errors. Pailoor et al. [15] used abstract interpretation to derive system call invariants and then applied program synthesis techniques to create optimal policies that closely approximate these invariants. Zhang et al. [18] noted that previous policies were static and proposed dynamically identifying necessary system calls during runtime, incrementally blocking access to unnecessary ones. Furthermore, the BPF modules often used in container sandboxes are not entirely secure. Gershuni et al. [9] employed an abstract interpretation-based method to directly verify BPF programs.

Security concerns for container-based cloud-native systems focus on network attacks and lateral movements between containers. Within a Kubernetes cluster, which forms its own local network, node-to-node attacks are more feasible than external ones. Bastion [14] introduces a smart, container-aware communication sandbox into the network protocol stack. This sandbox offers network and traffic visibility services, allowing for controlled inter-container communication within the cluster and effectively reducing the risk of internal network attacks. Lateral movements are another concern. Recent research [10] has uncovered that certain malware can exploit BPF to break out of containers, seize control of the resident pods, and potentially compromise the entire Kubernetes cluster. They then proposed an BPF permission model to mitigate BPF-based attacks within containers.

3 DESIGN

In this section, we introduce agent-manager architecture of vArmor, including vArmor policy/profile creation, distribution and update in the cluster. Furthermore, we demonstrate several security threats and how to detect and mitigate them.

3.1 Architecture Design

The architecture of vArmor is shown in Figure 1. The API server validates and configures data for the API objects which include pods, services, replication controllers, and others in Kubernetes. For example, a client can deploy pods or register resources by accessing the REST APIs specified by the API server. In the design, there are one agent and one manager on every nodes in the cluster. Additionally, there is a leader between these managers which is elected in initialization of managers. The managers of vArmor intercept these API requests by register an admission webhook service. For every incoming API requests, the API server will forward them to the registered webhook servers first. The manager mutates a matched request by adding an annotation to specify the corresponding *vArmor policy* object. The vArmor policy is a custom resource definition (CRD) that are supposed to be instantiated by a security administrator. The leader of managers is responsible for monitoring the changes of the instances of vArmor policy, and transforming them into *vArmor profile* which is an another CRD. There are two type of vArmor profile objects correspond the AppArmor and BPF LSM based profiles. The agents synchronize changes of the instances of vArmor profile by loading or reloading them in their own node. Then agents send back results of profile synchronization to profile status service pointing to the leader of manager.

3.2 AppArmor and BPF

Linux provides a very powerful framework *Linux Security Module* (LSM), which supports a variety of computer security models. vArmor enhances pre container security by setting profiles of AppArmor or BPF LSM. Currently all common Kubernetes-supported container runtimes should support AppArmor, but BPF LSM does not require. The former is achieved by modifying the security context of the process which is first process in the container. The latter works required loading a BPF program into kernel, which takes effects to all processes in the host.

AppArmor allows the system administrator to restrict programs' capabilities with per-program profiles. Profiles can allow capabilities like network access, raw socket access, and the permission to read, write, or execute files on matching paths. Each profile can be run in either enforcing mode, which blocks access to disallowed resources, or complain mode, which only reports violations. In the vArmor, a profile of AppArmor can be applied to a container by adding an annotation in the Pod metadata i.e., `container.apparmor.security.beta.kubernetes.io/<container_name>: <profile_ref>`. The content of the profile are modifiable in vArmor, and new profile works replace the old one. This is achieved by the agent in the node, we will discuss later.

BPF provides more fine-grained security controls by loading a customized program into kernel. There are many tracepoints (which are static code) located in strategic points throughout the kernel. A BPF program can setting up callback functions into these tracepoints, and they are invoked with specific parameters when the tracepoint is executed in runtime. However, the challenge is that BPF program process the events from all processes. In our scenario, a BPF program should be specified per container and take no effect for the process out of the container.

We know that each container container has its own mount namespace, which provides isolation of the file system mount points seen by processes in the container. In other words, the processes in a container should have same mount namespace. The mount namespace can be identified by a mount namespace id (mnt-ns-id) i.e., `/proc/pid/ns/mnt`. In vArmor, we can get the process id (pid) of first task (process) in the target container by monitoring the contain runtime. Then we can further get the mnt-ns-id by the pid. In the BFP program, we only confine the processes that have same mnt-ns-id. To identify a container as one that needs to be protected, we also can setting an related annotation in the Pod configuration. The annotation can be obtained by communicating with the container runtime.

3.3 Agent Design

The architecture of agent of vArmor is showed in the Figure 2.

Setting up. The agents manipulate profiles of LSM including profile loading, profile uninstall and profile reloading. A node is limited to only one agent, and the agent is also running in a container instead of on the host. The necessary files of AppArmor and BPF LSM

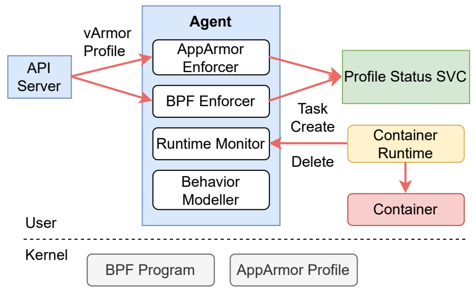

Figure 2: The detailed architecture of vArmor agent.

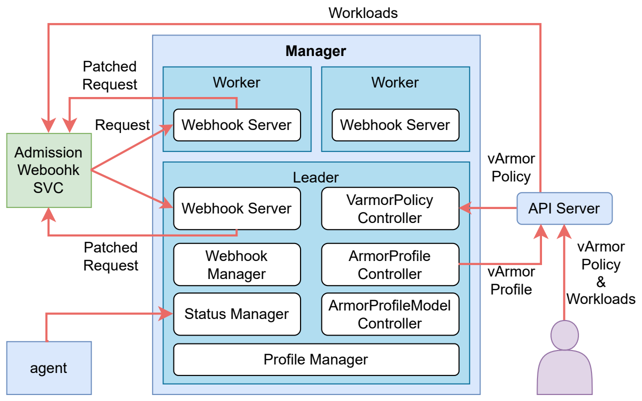

Figure 3: The detailed architecture of vArmor manager.

i.e., `/sys/kernel/security` and `/sys/module/apparmor` and container runtime `/run/containerd /containerd.sock` are exposed to agent for manipulating the profiles and monitoring container runtime.

Initialization. Firstly, the agent checks if the AppArmor and BPF LSM are enable in the host. The default AppArmor profile is removed for preventing interference when the agent asked to support AppArmor enforcer. The container runtime monitor is enable when the agent asked to support BPF enforcer. Secondly, the agent registers a informer to listen the operation events of vArmor profiles from API server.

Persistent Running. For any incoming events of vArmor profiles, the agent have to synchronize the corresponding profile of AppArmor or BPF in its own node. For a AppArmor profile, the agent saves it as a file and loads in into kernel if the profile is fresh. Otherwise AppArmor uninstall the profile for a delete event and reload the profile for a update event. Similarly, the agent do the same things for a BPF profile, one difference is that BPF profile works by the agent and AppaArmor profile works by container runtime. Notice that we can not change other profile for a container when it already started without restarting it. Although you can modify the contents of the current profile to achieved same effects, it may also affect other containers that use this profile. When a vArmor profile is synchronized or failed, the agent sends the status to leader of manager.

3.4 Manager Design

The architecture of vArmor's manager is showen in the Figure 3.
Setting up. The Managers collects vArmor policy, transforming vArmor policy into vArmor profile and patch the matched admission request to configuring the vArmor profile for pre container. By default a node has a management, and each manager has the capability for access and updating vArmor profile, while the agent only can access it.
Initialization. Firstly, each manager create a cache to store all available instances of vArmor policy for matching the incoming admission requests. Secondly, all managers elect a leader for registering the admission webhook service. Notice that each manager are able to receive the admission request after that. Therefore each manager runs a webhook server for patching the incoming admission requests. Then all managers do a leader election again for determining the manager that maintains the vArmor profile and maintains the status of profile application from agents. For simply, we call this leader as mLeader.
Persistent Running. For any incoming admission request, each manager first check the type and only patches the appreciated workloads identified by the selector in a instance of vAmror policy. For example, a selector can be specified for matching certain workload name. If a instance of vArmor policy is matched for current request, then a annotation will be inserted in the request for specifying the type of enforcer and certain profile. For any incoming events of vArmor policy, mLeader synchronizes the changes to corresponding instances of vArmor profile. Furthermore, the agents also have to synchronize the changes of vArmor profile. Then mLeader triggers a event for waiting the number of responses from the agent to validate if the synchronization is successful.

3.5 Security Model and Builtin Rules

Security model. A security model is a scheme for specifying and enforcing security policies. For example, we may use a security policy to specify which binaries can be run in the system. There are two possible templates for this policy: 1. allows some binaries if all binaries are denied by default; 2. deny some binaries if all binaries are allows by default. The security model of the former is called *deny by default* (DBD), and of the latter is called *allow by default* (ABD). In practice, the choice of which one to use is usually dictated by demand. DBD has the advantages of fine-grained access control and strong constraints. It is usually useful for specific application scenarios, as it is more difficult to characterize the behavior of legal user programs in complex environments. In contrast, ABD has coarse-grained access control and weak constraints. It works well for a variety of complex workloads. vArmor adopts ABD to accommodate complex business situations on the cloud.
Builtin rules. The security policy in vArmor is a combination of security rules. **Always Allow** and **Runtime Default** are two basic rules in vArmor, which correspond no restrctions and default AppArmor provided by container runtime. There are three general types of builtin rules in vArmor:

- **Hardening**: The rules for containers aims to reduce the attack surface and includes the following measures: 1. Blocking common escape vectors for privileged containers; 2. Disabling specified capabilities; 3. Blocking exploitation vectors for certain kernel vulnerabilities.
- **Attack Protection**: The rules aims to protect against hacker penetration techniques, thereby increasing the difficulty and cost of attacks and enabling a defense-in-depth approach. It includes: 1. Mitigating information leakage within containers; 2. Prohibiting the execution of sensitive actions; 3. Applying sandbox restrictions to specific executable files (AppArmor enforcer only).
- **Vulnerability Mitigation**: The rules is aimed at providing protection against vulnerabilities caused by insecure configurations, specific zero-day vulnerabilities, and security vulnerabilities caused by software features. It aims to block or increase the difficulty of vulnerability exploitation until the vulnerabilities are fixed.

These rules are implemented with either AppArmor or BPF, and some of them are implemented in both AppArmor and BPF.

3.6 Linux Kernel Hardening

We can say that there is no container security without kernel security. The earliest defenses for kernel are often placed at syscalls, we can disable these unsafe and unnecessary syscall for certain user applications. The second defenses for kernel are often placed at critical components of kernel such as memory allocation and memory access. We may set canary for memory block to prevent accessing memory out of bound, or restrict the kernel code cannot access content of user memory. Furthermore, the kernel exploitation vectors are often conducted by return-oriented programming (ROP) attacks. The third defense is placed for reducing available ROP gadgets or increasing difficulty of construction i.e., KASLR. The last defenses are often placed on the return point from kernel space to user space. We may use a control flow integrity (CFI) checker to guarantee the current PC is a valid return point.

vArmor primarily intended to enhance kernel security by reducing the attack surface. In Linux, capabilities are a way to assign specific privileges to a running process. Furthermore, accessing certain code of kernel often requires certain certain capability. For example, modifying network configurations requires capability *CAP_NET_ADMIN*. A natural idea is that we can restrict capabilities of a process to prevent certain code is being accessed by the process.

In general, a user process in a container does not have the privileged capabilities. However, the privilege escalation is possible. We have mentioned mount namespace in Section 3.2. Linux uses different kind of namespaces for isolating specific system resource. USER namespace is a special namespace, since it allows the owner to operate as "root" inside it, which has all privileged capabilities. In other words, you can use the syscall `unshare` and specify *CLONE_NEWUSER* flag to gain a full permission namespace and override original namespace. Recently, a number of exploitation vectors have used the method such as CVE-2021-22555, CVE-2022-0185, CVE-2022-25636, CVE-2022-2588 and CVE-2023-32233. We can directly deny the syscall `unshare` with a *CLONE_NEWUSER* flag, or simply deny privileged capabilities for all processes in the container.

For other certain Linux vulnerabilities, we can build appreciate rules to temporally mitigate it. For example, we can deny disk mounting for mitigating CVE-2023-0386 that occurs in components of OverlayFS. However, the precise rules for some special Linux vulnerabilities are not easy to build. For example, the exploitation of Dirty Pipe (CVE-2022-0847) is data-only and mutable. In other words, it does not have some distinctive features. Directly restricting the normal pipe operations may affects other user programs. The best mitigation for such vulnerability is patching.

3.7 Application Protection

In reality, the container-self security and application security in the container are typically get more attention. The former concerns if container specification is correctly implemented. While the latter concerns if application specification is correctly implemented. For example, CVE-2019-5736 allows an attacker inside a container to rewrite container runtime binary runC and achieve container escape. Obviously, it violates container specification. Moreover, a container escape also can be achieved unsafe container configuration. For example, an attacker can write arbitrary files located in the host by mounting disks in a privileged container. Some workloads require the docker API service `docker.sock` to be mounted inside the container. An attacker inside the container can achieve container escape by creating a new container and mounting the host filesystem service through the service. To mitigate such container escape or information leakage, we can deny processes inside a container access or mount the sensitive files.

There is no consensus on how to protect the application inside container. Here, we only discuss some typical threats in practice. Sensitive files inside container are supposed to be strictly restrict for accessing and modifying. For example, the service account and taken for accessing Kubernetes API server should be inaccessible for a unprivileged process. The hardware information are unnecessary for user applications in most cases. Malware remote command execution can be mitigated by deny command language interpreter such sh, bash and dash. Similary, the sensitive binaries such as curl and wget are supposed to be denied when they are not necessary for the workload.

The security between multiple containers are also concerned. An attacker can discovery services that located in the other containers through networking in the cluster. Furthermore, the attacker can move to a vulnerable container to cause more damage. However, it is often hard to detect such exceptions in heavy networking traffic, and the overhead for filtering traffic is non-negligible. It has been very useful in some special situations. For example, when we find a container is infected, then we can immediately update the security profile to isolate it in the cluster and prevent it casuing more damage.

4 EVALUATION

In this section, we quantify the performance of vArmor, and demonstrate vArmor's ability to protect the cloud native system's confidentiality and integrity. Experiments were run on three virtual machines with two processor cores and 30GB HDD, including one manager with 8GB RAM and two agents with 4GB RAM. In order to ensure the simulation of a real distributed environment, we run a virtual machine with manager on a machine with Intel i9-11900, and others with agent on a machine with Intel i9-10900.

4.1 Performance Measurements

	no-varmor	no-policy	bpf	apparmor
real	25.8	35.9	130.2	40.6
user	6.3	6.9	8.5	7.1
sys	1.1	1.3	2.3	1.3

Table 1: Task creation time (s) in Kubernates

System. To comprehensively assess vArmor's performance impact, we tested its effect on various tasks, including system calls (*execl*, *fork*, *getpid*), inter-process communication (IPC), and script execution. We ran vArmor under three conditions: without rules, with k AppArmor rules, and with k BPF rules. Figure 4 illustrates the performance impact as a percentage compared to execution without vArmor. The results show that security rules slow down script execution, particularly noticeable in both single-threaded and 8-threaded scenarios. BPF rules, due to their need for interpreting in the kernel, exert a more substantial slowdown than AppArmor. System call performance, specifically getpid, reinforced this observation. AppArmor had negligible impact, whereas BPF's kernel-level interpretation and execution resulted in noticeable overhead. The process creation task, which just executes *fork*, did not activate any rules during execution, hence no significant additional overhead was observed under any of the settings. For IPC, we examined pipe throughput and context tasks, involving intra- and inter-process communication, respectively. Only inter-process communication triggered AppArmor and BPF checks, leading to additional performance overhead.

Disk read/write. We evaluated file read/write speeds using buffer sizes of 256, 1024, and 4096 bytes under various conditions: without rules, with AppArmor rules, and with BPF rules. The relative speeds compared to the baseline are presented in Figure 4. The results indicate a slight reduction in file read/write speeds when vArmor is active. However, the speed variations across different settings are minor and not statistically significant. This could be attributed to server load conditions, where file operations might be expedited due to factors such as cache warming.

Bug (CVEs)	Description
2019-5736	Allows attackers to overwrite the host runc binary
2021-22555	A heap out-of-bounds write in netfilter x_tables
2022-0185	A heap-based buffer overflow in the Filesystem Context
2022-25636	A heap out-of-bounds write in netfilter nf_tables_offload
2022-2588	A use-after-free in cls_route filter implementation
2023-32233	A use-after-free in netfilter nf_tables.

Table 2: Tested CVEs fro extra rules.

Network. We utilized qperf to assess vArmor's effect on network latency, testing data sizes doubling incrementally from 1KB to 1MB. Our test scenarios included connections between two nodes in a virtual machine network (vmnet), between a virtual machine and a host machine via vmnet, and between two machines over LAN

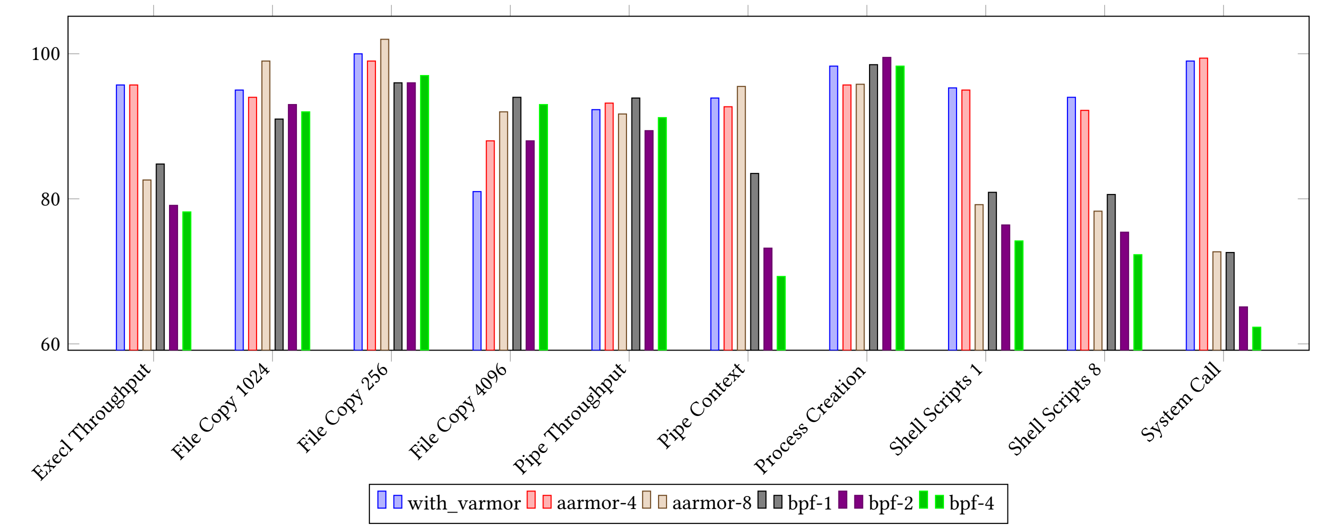

Figure 4: Quantified performance of vArmor's impact in runtime, the value under each task is the percentage (%) of the running speed of each setting compared to when vArmor is not started. Among them, *with_varmor* means that Kubernates has loaded vArmor without any rules, *aarmor* -k ($k \in \{4, 8\}$) indicates that the AppArmor protection with k rules is enabled, *bpf* -k ($k \in \{1, 2, 4, 8\}$) indicates that BPF protection with k rules is enabled.

and WAN. In these scenerios, we observed no significant impact on network latency from any configuration. This suggests that the time taken for security policy checks by vArmor is negligible compared to the inherent network communication times.

Threat Type	Description
Information Gathering	Mounts of sensitive files and directories from host.
Information Gathering	Available sensitive binaries.
Information Gathering	Sensitive local files.
Discovery	Retrieve kubernetes cluster information with system:anonymous.
Reverse Shell	Spawn an interactive reverse shell to remote IP.
Container Escape	Mount host devices to achieve reading and writing arbitrary host file.
Container Escape	Overwrite host cgroup's release_agent file to execute arbitrary code.
Container Escape	Mount host procfs to achieve arbitrary code.
Container Escape	Priviliged container i.e., SYS_PTRACE

Table 3: Tested threats for builtin rules.

Kubernates. We assessed vArmor's impact on Kubernetes task execution by measuring the time overhead during task creation for two reasons: (1) task creation is minimally influenced by system randomness like scheduling, and (2) vArmor is heavily involved during this phase. We generated 12,000 tasks and recorded the total, user, and kernel layer execution times, as detailed in Table 1. The difference between total time and the sum of user and kernel times represents the blocking time. When running with vArmor, we observed an overall increase in runtime, with blocking time showing the most significant rise due to vArmor's need for extensive network communication with Kubernetes to synchronize data during task creation. With AppArmor, the extra time is primarily due to profile synchronization communication overhead for new containers. When employing BPF, both blocking and kernel times notably increase. The rise in blocking time stems from network interactions with Kubernetes and containerd for monitoring data. The kernel time grows due to the need to interpret and execute BPF monitoring code within the kernel.

4.2 Security Policy Validation

We employed the Container Development Kit (CDK) [2] to validate the built-in rules within vArmor. CDK is a comprehensive container penetration toolkit designed to facilitate a variety of security testing methodologies, including information gathering, vulnerability discovery, credential access, and container escape techniques. The threats tested using CDK are enumerated in Table 3. Additionally, Table 2 presents a compilation of real-world vulnerabilities that were assessed during our evaluation, utilizing custom-written rules for the purpose.

5 CONCLUSION

Container security is a critical and widely debated issue, leading to the development of multiple protective strategies. vArmor demonstrates the effectiveness of leveraging Linux security features to bolster container security. It seamlessly facilitates rapid security enhancements across Kubernetes clusters, making it highly suitable for cloud-native applications. Our experiments indicate that with a well-configured set of rules, vArmor's impact on performance is negligible. vArmor's built-in rules provide defense against common threats, and custom rules can be easily crafted to protect against specific vulnerabilities.

REFERENCES

[1] [n. d.]. byte-unixbench. https://github.com/kdlucas/byte-unixbench

[2] [n. d.]. CDK - Zero Dependency Container Penetration Toolkit. https://github.com/cdk-team/CDK

[3] [n. d.]. Firejail. https://firejail.wordpress.com/

[4] [n. d.]. gVisor. https://gvisor.dev/

[5] [n. d.]. Linux 5.5 Git Threadripper + No Apparmor. https://openbenchmarking.org/result/1912315-PTS-LINUX55G46

[6] [n. d.]. nsjail. https://github.com/google/nsjail

[7] [n. d.]. vArmor. https://github.com/bytedance/vArmor

[8] Fabrice Bellard. 2005. QEMU, a fast and portable dynamic translator.. In *USENIX annual technical conference, FREENIX Track*, Vol. 41. Califor-nia, USA, 46.

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

[10] Yi He, Roland Guo, Yunlong Xing, Xijia Che, Kun Sun, Zhuotao Liu, Ke Xu, and Qi Li. 2023. Cross Container Attacks: The Bewildered {eBPF} on Clouds. In *32nd USENIX Security Symposium (USENIX Security 23)*. 5971–5988.

[11] Tony Jones. 2006. security: AppArmor - Overview. https://lwn.net/Articles/180623/

[12] Fotis Loukidis-Andreou, Ioannis Giannakopoulos, Katerina Doka, and Nectarios Koziris. 2018. Docker-sec: A fully automated container security enhancement mechanism. In *2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 1561–1564.

[13] Steven McCanne and Van Jacobson. 1993. The BSD Packet Filter: A New Architecture for User-level Packet Capture.. In *USENIX winter*, Vol. 46. 259–270.

[14] Jaehyun Nam, Seungsoo Lee, Hyunmin Seo, Phil Porras, Vinod Yegneswaran, and Seungwon Shin. 2020. {BASTION}: A security enforcement network stack for container networks. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. 81–95.

[15] Shankara Pailoor, Xinyu Wang, Hovav Shacham, and Isil Dillig. 2020. Automated policy synthesis for system call sandboxing. *Proceedings of the ACM on Programming Languages* 4, OOPSLA (2020), 1–26.

[16] Alexander Van't Hof and Jason Nieh. 2022. {BlackBox}: a container security monitor for protecting containers on untrusted operating systems. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. 683–700.

[17] Thu Yein Win, Fung Po Tso, Quentin Mair, and Huaglory Tianfield. 2017. PROTECT: Container process isolation using system call interception. In *2017 14th International Symposium on Pervasive Systems, Algorithms and Networks & 2017 11th International Conference on Frontier of Computer Science and Technology & 2017 Third International Symposium of Creative Computing (ISPAN-FCST-ISCC)*. IEEE, 191–196.

[18] Quan Zhang, Chijin Zhou, Yiwen Xu, Zijing Yin, Mingzhe Wang, Zhuo Su, Chengnian Sun, Yu Jiang, and Jiaguang Sun. 2023. Building Dynamic System Call Sandbox with Partial Order Analysis. *Proceedings of the ACM on Programming Languages* 7, OOPSLA2 (2023), 1253–1280.