

Inline Security Enforcement for Containers Using eBPF

PETER SERGO & AXEL HARTWIG

MASTER'S THESIS

DEPARTMENT OF ELECTRICAL AND INFORMATION TECHNOLOGY

FACULTY OF ENGINEERING | LTH | LUND UNIVERSITY



Inline Security Enforcement for Containers Using eBPF

Peter Sergo
pe3016se-s@student.lu.se

Axel Hartwig
ax0272ha-s@student.lu.se

Department of Electrical and Information Technology
Lund University

Supervisor: Christian Gehrman

Examiner: Thomas Johansson

June 5, 2026

Abstract

This thesis explores the possibility of using extended Berkeley Packet Filter (eBPF) to implement inline security enforcement for containerized workloads. The objective is to automatically create profiles for runtime enforcement within cloud environments such as Kubernetes. A key feature of eBPF programs is that they can attach to various different kernel events, similar to the Linux Security Module (LSM) AppArmor, which is already used in cloud security. AppArmor enforces Mandatory Access Control (MAC) policies by restricting the actions that programs can perform on a system.

By attaching eBPF hooks to some of the same LSM framework hook points a proof of concept is developed to demonstrate the flexibility of eBPF, where only the relevant hooks need to be utilized. The approach involves automatically constructing a whitelist-based profile through a training phase, during which expected benign behavior is observed and recorded. This profile is subsequently enforced to deny unauthorized actions, resulting in a solution that is straightforward to deploy. Furthermore, this is achieved without relying on the AppArmor module being enabled by default on all nodes across a Kubernetes cluster but instead relies on BPF LSM for enforcement.

The findings in this paper show that an eBPF solution is effective for locking down a container based on file paths and network access. However, additional development is required to achieve the same level of maturity and comprehensiveness as AppArmor.

Popular Science Summary

More and more companies are moving their applications to the cloud [1]. This means that data and computing power can be accessed on remote servers through the internet, offering benefits such as scalability, load balancing and redundancy. A smaller company can rent virtual machines on a big cloud provider such as Amazon Web Services (AWS) instead of hosting their applications in-house. Not only do they just pay for the computing power needed, but a deployment can also automatically scale up and down to handle short surges of traffic.

Cloud development has increasingly shifted to using container images. A container image packages an application together with all its dependencies to make it more portable and easy to deploy across multiple machines. The image can be uploaded to a container registry, from which it can be fetched when needed. Container runtimes such as Docker's containerd allow for running them as services in isolated containers on a system, instead of booting up a full new virtual machine.

An orchestration tool such as Kubernetes is used to deploy these containers on multiple nodes (servers), making up a Kubernetes cluster. The desired deployment state can be specified in a Kubernetes deployment file, and the control plane then monitors the state and health of the pods where the containers run. Failed pods are automatically restarted, replicas can be added or removed to allow for scaling and updates can be rolled out across multiple nodes without disruption.

A containerized app deployed in the cloud still requires security. Different vulnerabilities require distinct mitigation approaches. A container on its own uses namespaces to isolate processes from each other, while network policies decide which pods are allowed to communicate with each other. Taking it a step further, a profile can be built specifically for the application to restrict the allowed system calls and file accesses. These can be based on Linux Security Modules (LSM), like AppArmor and SELinux. The LSM framework provides a set of hooks that can be used by the modules to allow or deny actions according to their security policies. This replaces Linux Discretionary Access Control (DAC) with Mandatory Access Control (MAC) where the LSM policy decides the access instead of the user. Although this approach is beneficial, there are a lot of different LSMs, all of which are not enabled by default and they come with different benefits and drawbacks.

This is where extended Berkeley Packet Filtering (eBPF) comes in. It is a relatively new Linux technology that allows handwritten code to run safely within the kernel by attaching to kernel events, such as system calls and extending the

kernel behavior. This in turn can be used for monitoring and runtime threat detection, but it also has potential for inline security enforcement, where every action of the container is inspected and either allowed or blocked in real time.

The findings show that an eBPF solution can successfully audit containers to create behavioural profiles, which can then be used to lock down containers. These profiles achieve a 99% match in accessed file paths when compared to AppArmor for the same applications. By using the same hooks as AppArmor the permission types can be matched logging different kinds of accesses such as reading, writing and executing a file.

Table of Contents

1	Introduction	1
1.1	Problem Formulation	1
1.2	Research Questions	1
1.3	Work Outline	2
1.4	Contribution Statement	2
2	Background	3
2.1	Container Technology	3
2.2	Kubernetes	4
2.3	eBPF	5
2.4	LSM Framework	6
2.5	Kubernetes Security Tools	7
3	Related Work	11
3.1	Cloud Security	12
3.2	eBPF	13
3.3	Summary	15
4	eBPF Profiling and Enforcement Design	17
4.1	Compilation and Loading	17
4.2	Hooks	17
4.3	Mapping cgroup ID to Policies	18
4.4	Intercepting Events From the Start	19
4.5	Profiles and Policies	19
4.6	Matching Paths	20
4.7	Kubernetes Deployment	21
4.8	Summary of Proposed Architecture	21
5	Evaluation	23
5.1	Test Environment	23
5.2	Early Stage Validation Setup	24
5.3	Evaluation of the Design	24
5.4	Results	25

6	Discussion	33
6.1	Blocking Requests Benchmark	33
6.2	Attack Prevention Benchmark	33
6.3	Performance Benchmark	34
6.4	Comparison to AppArmor Generated Profiles	35
7	Conclusion	37
7.1	Main Findings	37
7.2	Possible Applications	38
7.3	Future Work	38
7.4	Test Scripts	44

List of Figures

2.1	Container deployment	4
4.1	Overview of the proof of concept	22
5.1	Automatically generated Alpine file access profile.	28
5.2	Demo of enforce file access, execution and sockets.	29
5.3	The number of times each hook was triggered.	29
5.4	The execution time of each hook.	30
5.5	A sorted selection of AppArmor (left) and eBPF (right) generated profiles for the application "api-test-projects".	30
5.6	Permission mismatches for the "api-test-projects" test.	30

List of Tables

5.1	HTTP Request Duration (Latency) across eBPF execution modes. All values are in milliseconds (ms).	27
5.2	Comparison of file path coverage between AppArmor and eBPF . . .	31

The rapid adoption of cloud-based architectures has fundamentally changed how applications are developed and deployed. As organizations increasingly migrate to the cloud, security approaches must evolve accordingly.

Currently, MAC-based solutions such as AppArmor and SELinux are the standard for securing containers. While these are effective, they rely on specific kernel-level configurations being enabled by default. On systems where AppArmor or SELinux are not enabled another solution, extended Berkeley Packet Filter (eBPF), often is. eBPF enables the execution of sandboxed programs directly within the kernel, providing enhanced visibility and control over kernel events.

This thesis was conducted in collaboration with Bifrost Security AB, a company focused on cloud security with a solution based on AppArmor, and explores the viability of using eBPF for container inline security enforcement.

1.1 Problem Formulation

While eBPF has seen a lot of use for its capabilities for network monitoring, its use for blocking malicious runtime behavior is an active area of research. The primary problem this thesis addresses is whether eBPF can reliably enforce fine-grained runtime security policies with an acceptable overhead, which is evaluated by measuring the latency added by monitoring a deployed app using our proof of concept.

This includes conducting a technical study on whether it is possible to use eBPF to block malicious runtime behavior for containers running in Kubernetes using profiling and whitelisting. Additionally, this thesis aims to create a proof-of-concept solution demonstrating this capability.

1.2 Research Questions

1. Is it feasible to use eBPF for inline security enforcement?
2. Is it possible to use eBPF to create profiling information about a container's runtime behavior?
3. Does eBPF enable creating more or different fine-grained blocking or profiling than what is possible with current MAC-based solutions?

1.3 Work Outline

The remainder of this thesis is structured as follows:

- **Chapter 2: Background** provides an overview of containers, Kubernetes and eBPF and describes current tools available for cloud security.
- **Chapter 3: Related Work** describes the current research being done in the fields of cloud security and eBPF.
- **Chapter 4: eBPF Profiling and Enforcement Design** explains the design and design choices made during the creation of the proof of concept.
- **Chapter 5: Evaluation** describes how the proof of concept was evaluated and the results of the evaluation.
- **Chapter 6: Discussion** interprets the experimental results and discusses the limits of the eBPF solution
- **Chapter 7: Conclusion** summarizes the findings of this paper and provides direction for future work.

1.4 Contribution Statement

The work has been a joint effort between A. Hartwig and P. Sergo. While all parts were worked on by both authors, there was a separation of responsibilities throughout the project period.

A.Hartwig took a leading role in the development of the proof of concept while P.Sergo wrote a major part of the paper in addition to the testing and deployment of the software. Generative AI tools were utilized to enhance wording and sentence structure, resulting in a smoother and more cohesive flow throughout the text.

The work would not have been possible without the help from Bifrost Security giving valuable insight into cloud security and providing virtual machines for testing. The authors also thank the supervisor Christian Gehrmann for all the support and feedback throughout the work.

This chapter provides the necessary background for understanding cloud deployments, eBPF technology and the security approaches considered in this thesis. It begins with an overview of container technology and its role in modern software development, followed by a discussion on Kubernetes. Next, a brief overview of eBPF and its security applications is presented, followed by an introduction to the LSM framework and its modules. Finally, the chapter reviews the current security tools available for Kubernetes.

2.1 Container Technology

A container is a running isolated user program built from a container image [2]. The image acts as a static blueprint for the application with all the dependencies, ensuring that the application functions the same way no matter where it is deployed. Apart from being isolated and portable, containers are also very lightweight because they share the same OS kernel, as can be seen in Figure 2.1.

The running container depends on a lot of Linux kernel technologies [3]. To achieve process isolation namespaces are used for the Process IDs (PID), file system, etc. This gives each container its own view of the system by labelling the processes. The PID starts from 1 within the container and each container has its own root file system.

Cgroups are another kernel technology used to limit and manage the resources a container can use [4]. CPU, memory and so on are limited and this acts as a way to ensure that no single container uses an excessive share of resources when running multiple containers on the same server.

A third kernel technology is OverlayFS, which is a union filesystem implemented in the Linux source code. This mounts and merges image layers when running. Images are made out of several read-only layers and an upper writable layer which means that different images can share the same lower-level dependencies. Two applications can depend on Ubuntu and Python:3.11 and their images can share the same layers, saving space and internet bandwidth by only downloading the new layers.

Combining all this is the container runtime. A layered approach is used to fetch the image and start the container process. A Docker daemon or a Kubernetes kubelet agent talks to a runtime manager (containerd / CRI-O) which unpacks

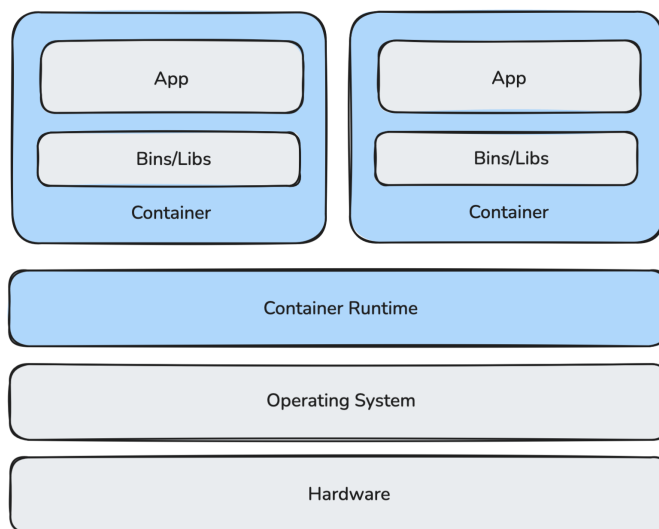


Figure 2.1: Container deployment

the image layers and delegates to the low-level OCI runtime (runc) to spawn the container process. Runc creates the Linux namespaces, applies the cgroups and so on before actually starting the process.

2.2 Kubernetes

Kubernetes (k8s) is an open-source container orchestrator that automates the deploying, scaling and managing of containers across multiple nodes (a cluster) [5]. These nodes are typically VMs hosted on a cloud provider for isolation and scalability but they could just as well be physical machines.

The containerized applications are run in pods on worker nodes. Each worker node has a kubelet agent (daemon) that manages the node, a kube proxy for routing and a container runtime that runs the containers.

The cluster is then controlled by a master node (the control plane) via the `kubectl` cli tool. Deployments and services can be written in declarative YAML files stating the desired state of a deployment, number of replicas and the container image.

Helm is a package manager for Kubernetes. Instead of managing and applying multiple different YAML manifests—such as Deployments, Services, ConfigMaps, Secrets and Ingress resources—Helm allows these resources to be packaged together into a single unit called a Chart. A Chart is a collection of templated Kubernetes manifests along with metadata and configuration values, enabling applications to be defined, installed and managed as a single package.

Finally to enable monitoring and enforcement of all running pods DaemonSets can be used. DaemonSets are used to deploy pods on every selected node [6]. This

can be used in combination with Helm to install security agents on all nodes across the cluster.

2.3 eBPF

Extended Berkeley Packet Filtering (eBPF) is a technology that allows handwritten code to run safely within the kernel by attaching to different hook points like system calls, function entry/exit and more. This enables extension of the kernel, without having to change the kernel source code or load kernel modules [7].

One of the main benefits of eBPF is that it allows for data processing directly in kernel space without first sending it to user space, reducing context switch overhead. Additionally the privilege and system overview afforded when running code in the kernel is ideal for implementing things like observability and security [7].

2.3.1 Security

The ability for users to run code in the kernel comes with many security risks. Not only does it enable malicious actors access to the core of the system, but it also enables a naive programmer to introduce unintended errors unknowingly. To prevent things from breaking, eBPF programs are required to pass strict checks before being loaded into the kernel.

Importantly any process that wants to load eBPF programs needs to be either running as root, or with the capability `CAP_BPF`. Making sure that untrusted programs cannot load eBPF programs.

Additionally, before being loaded, the eBPF program needs to pass through the eBPF verifier, which ensures the safety of the program. The verifier ensures that the program:

- Runs to completion.
- Does not use uninitialized variables or access memory out of bounds.
- Fit in the size requirements of the system.
- Have a finite complexity.

After passing through the verifier the program is run through a hardening process to prevent malicious attacks. These include security measures such as making the memory holding the eBPF program read only and blinding constants to prevent JIT spraying attacks [7].

Lastly, an eBPF program cannot access arbitrary kernel or user memory directly. Instead programs must use eBPF helper functions, ensuring that data accesses are consistent and subject to the privileges of the eBPF program [7].

2.3.2 Writing and Loading eBPF Programs

eBPF programs are typically written in a restricted version of C and compiled into eBPF-bytecode. Programs are then loaded into the kernel via the `bpf()` syscall during which they pass through the verifier [8].

The restrictions placed on eBPF programs can make it hard to write useful code. Luckily there exists a large number of kernel-defined helper functions which eBPF programs can call to interact with the kernel [9]. Documentation of available helper functions can be found in the kernel or in `bpf-helpers(7)`, a few relevant examples include:

- `bpf_map_lookup_elem`, for reading elements in a map.
- `bpf_tail_call`, for triggering a tail call, which is when you jump into another eBPF program.
- `bpf_probe_read_user`, for reading user memory.
- `bpf_probe_read_kernel`, for reading kernel memory.

eBPF programs make use of maps. These are data structures that can be accessed both from an eBPF program and from user space, which enables communication between eBPF programs and user space applications. Types include hashmaps, arraymaps, ring buffers and more.

There exist several toolchains to help in the development and loading of eBPF programs. A few examples include BCC, bpfftrace, the eBPF Go library and the libbpf C/C++ library.

2.3.3 A Sample eBPF kprobe Hook

```
1 SEC("kprobe/sys_execve")
2 int hello_exec(void *ctx) {
3     char comm[16];
4
5     bpf_get_current_comm(&comm, sizeof(comm));
6
7     bpf_printk("Execve called by: %s\n", comm);
8     return 0;
9 }
```

Listing 2.1: Sample eBPF program

2.4 LSM Framework

The LSM framework is defined in the Linux kernel source code and defines a set of hooks (a hook API) at security critical points in the kernel, like opening a file, executing binaries and so on [10]. This can then be used by security modules to implement Mandatory Access Control (MAC), as opposed to the conventional way of Discretionary Access Control (DAC) typically used in Linux.

A security policy is a set of rules that define which actions are allowed and which are not. Whenever a sensitive task is run, the corresponding hooks get executed and the security policy is checked. This means that an LSM can block unwanted behavior.

2.4.1 AppArmor

AppArmor is a LSM using profiles [11]. It is based on the idea of profiles that are unique to each program to determine what files and permissions it requires [12]. It is path-based meaning that the profiles apply to specific file system paths as opposed to SELinux where security labels are assigned to the files directly.

There are two modes of operation: complaining (learning) and enforcing (confined). When creating a profile a test plan should be designed with all the standard use cases to create a comprehensive profile. The profiles are then stored in `/etc/apparmor.d/` with rules for path entries and capability entries.

2.4.2 BPF LSM

BPF LSM is a relatively recent addition to the LSM framework, providing a way to implement LSM policies using eBPF. Developed by KP Singh at Google, it allows developers to attach custom eBPF programs to LSM hooks, enabling security decisions to be made dynamically at runtime [13]. This approach combines the flexibility and programmability of eBPF with the system-wide enforcement capabilities of an LSM, offering a level of customization and responsiveness that traditional LSMs cannot achieve. Whereas traditional LSMs operate through static hooks and fixed policy modules, such as AppArmor, BPF LSM allows policies to adapt to evolving runtime conditions, monitor process behavior in real time, and implement fine-grained controls based on process metadata, file paths, or network context.

2.4.3 Comparison to Seccomp

Seccomp or Seccomp-BPF is another way to limit what system calls a process can make [14]. Essentially, Seccomp allows a process to define a filter that permits, denies, or logs certain system calls. This approach allows processes to operate with limited privileges, containing untrusted code while leaving the rest of the system unaffected. In contrast, Linux Security Modules operate at a higher, system-wide level. It enforces security policies across all processes in the system and is more of a complete framework.

2.5 Kubernetes Security Tools

A wide range of security tools exist for Kubernetes, reflecting the growing need to protect cloud-native environments. The Cloud Native Computing Foundation (CNCF)¹ has cataloged many of these tools in the Cloud Native Landscape following major categories, like Networking and Security, to provide a comprehensive overview of the current ecosystem [15]. When combined with platforms like the Bifrost Agent and emerging projects such as SBOB, these tools illustrate the diversity of approaches available for enforcing security in Kubernetes clusters. The

¹The Cloud Native Computing Foundation (CNCF) is a nonprofit organization that supports the growth and adoption of cloud-native technologies such as Kubernetes, Prometheus, and Envoy. See <https://www.cncf.io/>

following subsections explore these tools in detail, highlighting their functionality, scope, and how they contribute to robust cloud-native security.

2.5.1 Bifrost Agent

The Bifrost agent is a runtime security tool based on AppArmor that automatically creates security profiles for clusters and is installed via Helm [16]. In order to audit the application the deployment file needs to be changed for the agent to actively monitor the program. Once the profile has been created the agent can be set to enforce mode to lock down the application, preventing it from doing anything it did not do during the audit mode. It is also possible to run the bifrost agent in complain mode, in which it, instead of blocking behaviors it has not seen before, complains about them but lets them proceed.

By incorporating this with a web portal Bifrost offers a solution that not only generates the profiles but also stores them. Since these profiles are specific to the version of the container image, it becomes possible to roll back to a previous state.

2.5.2 KubeArmor

KubeArmor is a runtime security system for pods, containers and nodes. It uses eBPF for monitoring and runtime enforcement but also supports the use of AppArmor or SELinux if BPF LSM is not available [17][18].

KubeArmor tries to simplify the LSMs and make policy enforcing more straightforward, while also addressing other challenges in Kubernetes native Pod Security Context—such as difficulties in predicting which LSMs are available and limited support for BPF LSM. Once installed in the cluster KubeArmor policies must be defined as YAML files to restrict unwanted behavior. For instance, using a container label to specify which app should be blocked from accessing a sensitive file.

2.5.3 Falco

Falco is an open-source cloud-native security tool for detecting and alerting on abnormal behavior in real time [19]. Falco was originally created by Sysdig and is now a graduate of the CNCF project.

Falco operates by monitoring system activity at a low-level, primarily using eBPF probes to capture and analyze system calls. By inspecting syscall streams, it can identify potentially malicious or unexpected behavior, such as unauthorized access to sensitive directories (e.g., `/etc` or `/usr/bin`), privilege escalation attempts, or abnormal process execution patterns. Its detection capabilities are driven by a rule-based engine, where users can define custom policies in YAML to describe suspicious behavior and trigger alerts.

While Falco itself functions primarily as a detection and auditing tool, it can forward Falco alerts to third parties with Falcosidekick [20]. This means that a response engine such as Falco Talon can react to events using rules [21].

2.5.4 Tracee

Tracee is another open-source tool that leverages eBPF to track Kubernetes activity. Developed by Aqua Security, it provides real-time information on events such as syscalls, network activity and file operations. One of Tracee's key strengths is its ability to detect attacks by analyzing behavioral patterns associated with advanced defense evasion techniques, including fileless execution.

Like many other tools, Tracee is configured using YAML files [22]. Similar to Falco, it primarily serves as a detection and observability tool, focusing more on detecting suspicious behavior and generating alerts than actively blocking events in real time.

2.5.5 Cilium Tetragon

The Tetragon project, developed by Isovalent, is also a well-established eBPF-based security tool [23]. Isovalent has been deeply involved in the development of eBPF and were originally known for their work on the Cilium project, an open-source networking solution for Kubernetes. Tetragon provides observability and runtime enforcement using eBPF [24]. In addition to this Cilium has their own Go solution to load, compile and debug eBPF programs called ebpf-go [25].

A key point of the Tetragon solution is that it is Kubernetes aware, meaning that it recognizes different namespaces and pods to enable per workload security. In its simplest form Tetragon can perform basic execution monitoring, but it is also capable of more fine-grained monitoring for file accesses and network as well enforcement using policies defined in YAML files. This is in contrast to KubeArmor which is primarily built around runtime security enforcement and the LSM framework.

2.5.6 SBOB

SBOB (Software Bill of Behavior) is a new project based on the idea that software vendors can provide profiles with runtime rules describing how the software is supposed to behave during execution [26]. The benefit here is that the end user can do direct anomaly detection based on the profile without having in depth knowledge of the application's internal implementation.

This concept can be compared to SBOMs (Software Bill of Materials), which provides a static inventory of a software's dependencies [27]. While SBOMs improve visibility into the software supply chain, they do not capture how the software behaves at runtime. SBOB complements this by introducing a dynamic perspective, focusing on what the software is expected to do rather than what it contains.

A key aspect of SBOB is its integration with modern software distribution practices. When a new version of an application is released, an updated behavioral profile is packaged together with the corresponding OCI image. This shift from the user writing the rules to the vendor provided profiles enhances scalability and streamlines policy management.

Adding to this, SBOB uses eBPF to collect the runtime data behavior of an application when building the profile. This enables it to capture all sorts of events,

capabilities, network endpoints and file paths.

Overall, SBOB complements existing security mechanisms by adding a dynamic, runtime-focused layer, helping to strengthen software supply chain security and improve protection in cloud-native environments.

Previous thesis work has been done on this subject by Evelyn Beck in 2023 [28]. It serves as a starting point for this master thesis. The paper examines the inherent challenges of using eBPF eventually resulting in Beck developing a bpf tracing tool called Bpfsnoop. The goal was to replicate Bifrost's AppArmor solution to create audit data by attaching the program to the same LSM hooks. The resulting program suffered from worse performance but fulfilled the criteria of making a solution that could run on all major Linux distributions where AppArmor is unavailable.

The paper includes a section on the availability of eBPF. The provided solution based on BPF LSM needs the module to be enabled in the specific Linux distribution. Not only that but a relatively new kernel version back in 2023 was also important for the use of ring buffers and certain hooks.

One of the challenges was reading full paths from eBPF programs. The author mentions an eBPF helper function `bpf_d_path` that could have solved this but the patch was not yet merged into the kernel at the time of publication.

A second challenge was filtering the events by container. Many processes could run at the same time and it is crucial to identify which process belongs to which container profile. The chosen solution made use of the cgroup ID of the current process, as it usually is the same for all processes within the same container. As eBPF matures more tooling becomes available and some of these previous challenges are overcome [29]. An example of this was when the program size limit was raised from 4096 instructions to a million instructions [30].

Finally, Beck points out the security considerations when using the proposed bpfsnoop solution. The container needed the additional `CAP_SYS_ADMIN` capability, essentially making the DaemonSet run as root. While the Bifrost agent also needs extra capabilities none of them are as dangerous.

Additionally, work done by Christian Gehrman and Hui Zhu at the department regarding the use of AppArmor for profile generation forms the goal of what an ideal eBPF-implementation should achieve [31]. In their paper, a secure cloud service was proposed that creates profiles for containerized services through runtime profiling. The solution is to use AppArmor with Mandatory Access Control to deal with the risk of several containers sharing a single kernel. This was tested on a live web service using 11 known exploits showing that the security was improved significantly in comparison to the baseline.

The AppArmor profile generator as a cloud service was further expanded upon

by Gehrman and Zhu in their following paper on Kub-Sec [32]. Kub-Sec applies the original idea to Kubernetes and integrates it with the cluster using DaemonSets to deploy a logging agent. It also describes the use of an AppArmor loader to load the profiles and a central service to store the raw data and create the profiles.

These approaches also formed an important foundation for the Bifrost agent, which builds upon the same core principles of runtime observation and automated profile generation. By leveraging AppArmor in combination with Kubernetes entire deployments can be tracked and profiled.

3.1 Cloud Security

In 2023 an academic article was published dedicated to exploring security in cloud-native services and explaining security threats specific to the cloud-native environments [33]. The authors emphasize the challenges of keeping a cloud based infrastructure secure and a wide attack surface susceptible to different kinds of threats. They also found that previous attempts at surveying cloud security has been very general and lacked a focus on cloud-native services with containerization.

The article functions as a mapping from key aspects of security, to their corresponding features and finally to their corresponding solutions. Key technologies include the use of a service mesh to deal with the exponential complexity of each service talking to many other services. Rather than maintaining security features like TLS in each service, an infrastructure plane is introduced using sidecar proxies. Also Zero Trust Architecture (ZTA) is discussed as a way to prevent lateral movement where trust is never granted by default and the principle of least privilege is followed to only grant the minimum necessary privileges, thus giving a more granular segmentation.

The findings concludes that the suggested security solutions can work together in the making of a robust system. The concept of Service meshes are built around the principle of native security mechanisms that enable a more thorough, automated, and granular security posture by providing observability and analysis across multiple security dimensions similar to using DaemonSets.

A second paper analyzes the 15 most popular open-source security tools for cloud native environments in response to the different vulnerabilities for cloud deployments [34]. The analysis includes tools such as Falco, KubeArmor and Tracee.

The results showed that no tool had all the examined features, such as low-level system event collection and CVE detection. Static analysis tools such as Trivy and Kubescape can detect misconfigurations and known vulnerabilities but lack runtime detection capabilities. Runtime tools such as Falco and Tracee collect low-level system call events using eBPF but depend heavily on predefined rule sets, meaning that novel or undocumented attacks may go undetected.

These limitations point to the need for more comprehensive and flexible security mechanisms. In particular, eBPF-based tools with extra runtime enforcement that can detect previously unknown vulnerabilities.

3.2 eBPF

eBPF has proven to be a valuable technology for container security. A number of scientific papers have been written with different approaches to eBPF monitoring and runtime enforcement providing inspiration for a proof of concept.

3.2.1 Proposed eBPF-Based Enforcement Systems

Lu et al. present BPFGuard, a MAC container security system based on eBPF and integrated with Kubernetes [35]. The authors argue that eBPF provides better hooking capabilities and programmability in comparison to other solutions. In BPFGuard, these capabilities are specifically utilized by hooking onto LSM hooks, system calls, and kernel operations. Despite these advantages, the authors acknowledge that limitations still remain, particularly in handling emerging vulnerabilities or specific security requirements.

Enforcement is done in two ways, signal sending and return value overwriting. Using the eBPF function `bpf_send_signal`, you can send signals from the kernel to a process that has broken some policy. For example, a malicious policy could be terminated by sending a Kill signal to it. It is also possible to modify a return value using `bpf_override_return` to a failure-indicating value, which might be preferable to killing the process.

Other work has also been done on the subject, Gwak et al. [36] present KRSIE, an eBPF-based Kubernetes Runtime Instrumentation and Enforcement System. Their system is split into four main parts: A Policy generation part, used to have a user friendly way to create policies; a policy enforcement part which parses the generated policies and uses them to create and execute BPF LSM programs; a Monitoring part, implemented with tracee-ebpf which collects kernel events and lastly a policy augmentation part is used which uses the data from the monitor part to update policies if necessary.

The authors found that the primary advantage of using an eBPF-based solution is the ability to do dynamic enforcement. Traditional MAC solutions such as AppArmor or SELinux often require a container restart or redeployment to apply updated security profiles which is impractical in real cloud environments.

Relating to the work presented in our report, both BPFGuard and KRSIE indicate that eBPF-based runtime enforcement using LSM hooks provides greater hooking capabilities and flexibility than other solutions. In addition, support for dynamic enforcement enables eBPF-based solutions to be more effective at blocking threats through dynamic policy handling and loading via eBPF maps.

3.2.2 Syscall-Based Monitoring and Filtering

Another solution to securing containers using eBPF is DeSFAM (Dynamic eBPF-Driven Syscall Filtering and Anomaly Mitigation) [37], which uses eBPF to do real-time syscall enforcement. Static analysis is combined with dynamic eBPF tracing to generate a syscall allowlist enabling the system to enforce least-privilege access without requiring restarts.

DeSFAM utilizes eBPF programs attached to LSM hooks, more specifically it attaches to the hook `security_syscall`, allowing observing and acting on all

syscalls, which is done with the `bpf_override_return` helper function. When testing DeSFAM on real CVE attack scenarios the authors observed low enforcement latency and negligible performance overhead.

The report aligns well with our goal of achieving a practical eBPF solution with low overhead. It also demonstrates that eBPF enables more fine-grained blocking, as the DeSFAM solution can observe and act on all system calls.

3.2.3 Comparative Analyses of eBPF Security Tools

A government-funded study from Korea highlights that various eBPF-based security tools have already been developed but that they serve different purposes. The article tries to make a comprehensive comparative analysis of four of the main options, namely KubeArmor, Falco, Tetragon and Tracee, and calls attention to that most of the solutions are only focused on runtime detection [38].

Their research shows that administrators often deploy Kubernetes clusters with default settings, posing serious risks. The recommendation is to use eBPF-based tooling but there is a lack of knowledge of the inner workings of the tools, such as what hook points are used and complexity making effective utilization strenuous.

A qualitative analysis is done listing different auditing features and the architectures of each of the tools. All of them have in common that they rely on 3 primary forms of hook points: process, file and network, but the number of hooks differ. Tetragon has a dynamic approach and can add more hooks if needed while the others use a predetermined fixed set. The drawback of using more hooks is that the overhead becomes bigger.

All of the tools use eBPF hook points, but KubeArmor and Tetragon also adds LSM-BPF. This results in KubeArmor and Tetragon providing security enforcement as well in addition to visibility. Both of them also use Go with the Cilium/eBPF package.

Finally the report underlines the limitations of eBPF. The eBPF verifier imposes several constraints on the structure of the programs meaning that more complex programs become more difficult. There is also the possibility of performance bottlenecks whenever data is moved from kernel space to user space.

Key takeaways are that most existing eBPF tools prioritize detection over enforcement, and that hook selection impacts performance. This motivates our focus on inline enforcement using a minimal set of LSM hooks, while also differentiating our approach through automatic policy generation rather than relying on predefined rules.

3.2.4 Protection Against eBPF Attacks and Container Identification

A recent paper by Xu et al. [39] proposes eBPF-sec, a defense framework that automatically generates defense policies against eBPF attacks. eBPF-sec utilizes eBPF to collect information necessary to build up a profile of which eBPF operations should be allowed. The restriction model also works by using eBPF and BPF LSM hooks to intercept operations and stop or allow them based on the

defined policy data. The authors find that eBPF-sec protects containers from eBPF-attacks while having a minimal performance overhead.

Additionally, the authors utilize a relevant technique for identifying which process belongs to which containers by using cgroups and the BPF-MAP structure. Compared to other methods, this method improves accuracy and allows dynamic integration and monitoring during the containers entire lifecycle.

Furthermore a paper [40] uses eBPF as a way to trace system calls for preventing escalation attacks using a Bag of System Calls method. The authors use three different types of eBPF programs, kprobes, tracepoints and raw tracepoints. and they found that raw tracepoints had the smallest overhead.

The most relevant aspect for our design is how processes are associated with containers through cgroups and represented using BPF map structures. While using raw tracepoints is interesting from a monitoring perspective, it was excluded from our implementation because it provides limited abstraction and flexibility compared to LSM hooks. In addition, tracepoints do not align well with our goal of path- and behavior-based enforcement.

3.3 Summary

In summary, the reviewed literature provided both direct design inspiration and contextual understanding for the proof of concept. Several key mechanisms were adopted, most notably the use of eBPF programs attached to LSM hooks for runtime enforcement, as demonstrated in systems such as BPFGuard and KRSIE. In addition, the approach of using cgroups in combination with BPF maps to accurately associate processes with Kubernetes containers, as proposed by Xu et al., was incorporated to enable precise container-level policy enforcement.

While not used directly, the work on eBPF attack mitigation, such as eBPF-sec highlights that eBPF itself can introduce new attack surfaces if not properly constrained. Similarly, studies focusing on syscall tracing and other types of monitoring were not included due the focus of the thesis revolving around the viability of eBPF enforcement.

eBPF Profiling and Enforcement Design

This chapter describes the design behind a proof of concept for profiling and enforcement using eBPF in a Kubernetes environment. First, the eBPF programs need to be loaded into the kernel. Each program needs to execute when a specific hook is called. It needs integration with Kubernetes in the form of following the correct processes, intercepting their events immediately when the containers start and applying the correct policies in the correct mode. Finally the profiles need to be stored, looked up efficiently and be able to deal with changing path names.

4.1 Compilation and Loading

For the proof of concept the `ebpf-go` package by Cilium was used, as it provides seamless integration with Kubernetes while maintaining support for Compile Once - Run Everywhere (CO-RE). The loader component is implemented in Go, with `ebpf-go` added as a dependency using the Go package manager. Furthermore it adds convenient functions to access the eBPF maps straight from the Go program.

The eBPF programs are compiled using `clang`, and `bpf2go` is used to generate corresponding Go bindings. These bindings include embedded Go structures representing the eBPF programs and maps, enabling type-safe interaction between kernel space and user space.

4.2 Hooks

Many options are available to produce more fine-grained protection apart from the LSM hook framework. This includes

- `kprobes`, that hook directly into kernel functions.
- `tracepoints`, that are predefined kernel points and thus more stable than `kprobes`.
- `XDP`, for processing packets.

In this prototype, LSM hooks were chosen as the primary mechanism as they provide an easy way to deny access for a call and the hooks provided enable us to imitate existing solutions closely by hooking into the same points they use. A

complain mode is also supported by replacing the return value of the LSM hooks, allowing actions to proceed while still being logged.

Two key LSM hooks that were used for enforcing read, write and execute of programs and files were:

- `bprm_check_security`, triggered before a program is executed
- `file_open`, triggered when a file is about to be opened

The context from these hooks includes kernel structures such as `linux_binprm` which contains data that is used when loading binaries and the `file` struct that contains data like `f_mode` flags, describing how the file was opened, and its path [41].

For monitoring network activity, the following hook was used:

- `socket_create`, triggered when a process tries to create a new socket

This hook provides context like address family and type of the socket. The numbers correspond to values like `AF_INET` for IPv4 and `AF_INET6` for IPv6. There is also a parameter `kern` which is used when the kernel itself initiates a connection, which should typically not be blocked.

4.3 Mapping cgroup ID to Policies

The kernel-level eBPF code does not know what container is running when hooks are triggered, however it can retrieve what cgroup ID the process that triggers the hook has by using the bpf helper `bpf_get_current_cgroup_id()`. Furthermore the cgroup ID changes every time the container is started. This means that we need some way to connect a container (or deployment) to the policy we want to apply and then some way to connect the container to the internal cgroup ID it gets when it is created. Additionally, this connection needs to happen before the container starts running user code, as we do not want to miss enforcement or auditing of any events that happen in the beginning of the running program.

To achieve this we make use of NRI-hooks which enables us to hook into different lifecycle events of containers and run code before the container runtime system is allowed to continue. More specifically we hook into the `StartContainer` event. In the event we are able to infer the cgroup path by using the available container metadata. Conveniently, the inode number of this cgroup path corresponds to the cgroup ID of the container.

Now that the cgroup ID is available we need to connect a container instance with the policy that should be applied to the container. An initial approach might involve using the image name of the container, however, this is problematic as one can deploy the same image with different parameters or in different environments, which would result in the same image having different behaviors. Instead, we decide to use annotations in the deployment file, which are easily accessible in the container metadata.

By storing this information inside of eBPF-maps we can now correlate a cgroup ID to a container instance and enforce the correct policy.

4.4 Intercepting Events From the Start

As mentioned the connection between cgroup ID to policies needs to be made before the container gets the chance to run user code. When the monitored containers start, their corresponding profiles need to be loaded in addition to the LSM hooks for the chosen mode. This needs to be done synchronously to intercept all events and prevent race conditions. For instance, an event could be missed in the audit phase and then stopped in the enforce phase resulting in a false positive and a container failing to start.

The NRI API is also used because it blocks the container from starting before it returns. The runtime calls the NRI plugin's StartContainer hook and the code is loaded. But some concurrency is still needed. In the audit phase signaling is used to determine when the training is done and the policy file should be generated. This and a few other cases are handled with the help of go routines to run functions in the background.

4.5 Profiles and Policies

For the proof of concept, profiles are stored as files in a key-value format. Each profile defines access rules for system resources on a per container basis. For files and directories, entries consist of file paths mapped to a set of permitted operations, namely read, write and execute. In addition to file-based rules, separate files with profiles for sockets also exist with a combination of the socket context like address family and socket type.

4.5.1 Profile Creation

Unlike traditional MAC systems such as AppArmor, which rely on static and comprehensive profiles, the BPF LSM framework operates by attaching eBPF programs to individual LSM hooks. During the training phase, eBPF maps are used to store the benign behavior observed from a target container.

When the container that is being monitored is deployed with an audit label a new set of maps are created specific to that payload. Any action that should be allowed in the monitored container is performed, either manually or by the use of test scripts. Then when the agent receives the correct signal it loops through all the monitored events corresponding to that cgroup and creates a new audit file with the name of the deployment. Then by removing duplicate path accesses, space is saved and any ambiguity with multiple policies for a single path is removed.

As detailed in Section 4.4, the architecture guarantees that all runtime events are captured from the exact moment the container starts, ensuring no behavior is missed. However a fundamental challenge with automated profile generation remains regarding the coverage of the benign behavior. The system only records the actions that are actively triggered during the audit phase. Therefore the completeness of the profile relies heavily on the thoroughness of the test scripts or manual actions used to exercise the application. If a specific benign feature is not triggered during the training phase it will not be recorded. If this missed behavior later occurs when the container is in enforce mode, it will be blocked as an

unauthorized action, resulting in a false positive. To mitigate this in production, deployments can first be run in complain mode to log unrecorded behaviors before strict enforcement is applied or testing suites can be improved so that all expected behavior is covered.

4.5.2 Bitmasks

When checking the permissions of a file access a bitmask is used. Each file action, such as read, write and execute, corresponds to a bit. This representation allows multiple permissions to be compactly stored in a single integer. But more importantly, it enables fast permission checks during enforcement. Instead of doing string comparisons or iterating over lists of allowed actions, the system can determine whether an operation is permitted using a simple bitwise AND operation between the requested action and stored permission mask.

4.6 Matching Paths

Before dynamic path matching can be done, the full path must first be extracted from the kernel context. In previous work by Beck [28] resolving the full path was identified as a major technical obstacle. However this is no longer a limiting factor as the helper function `bpf_d_path` is now available inside of the kernel, giving us the absolute path without issue.

When enforcing policies, matching the path directly is not always useful. Consider for example an access inside of the `/proc/<PID>/` directory. When auditing one pid might be used and while enforcing another pid could be used. Or when dealing with logging files, we might see timestamps being used. If we only rely on the exact path for choosing what policy should apply we would miss cases like this where the filepath varies from run to run.

To solve this AppArmor uses a regex-like concept known as *Globbering* [42]. For example, a rule allowing access inside of the `proc` directories could use globbing like this: `/proc/*/`. Automatic profile generators such as LicSec [43] make use of globbing when generating rules, not only to account for randomly generated names, but also to minimize the number of rules needed to create an accurate ruleset.

Unfortunately, globbing is not natively available inside of the kernel, and available libraries would likely not work anyways as the verifier is very strict when dealing with things like unbounded loops. Instead we have to implement something similar, while dealing with the limitations of the verifier.

Inspired by a talk by Justin Ngai at Meta [44], presenting a partial regex-engine based on a DFA, we chose to implement something similar in our prototype. Instead of creating a state for each separate char, we chose to separate each state by directories. This enables us to do path matching based on the closest parent directories rules as well as adding the possibility to use wildcards `*` in place of whole directory names, enabling having rules for paths like `/proc/*/` and having a general rule for parent directories while having more specific rules for directories further down the file tree.

4.7 Kubernetes Deployment

The program is deployed as a DaemonSet using Kubernetes manifest files listing the permissions and capabilities of the program. Docker was used to build the container image which was then uploaded to a container registry and pulled by the manifest.

```
1     labels:
2       app: alpine-shell
3       hargo/enabled: "true"
4     annotations:
5       hargo/mode: enforce
6       hargo/name: test-alpine
```

Listing 4.1: Annotation for monitored deployment

Labels and annotations were used for the deployments that were monitored using the eBPF agent. A label with the enabled status was used to see if the new pod should be monitored or not. In addition to this two annotations were used. One with the name of the deployment to separate the different policies from each other and one with the mode of the deployment.

- **audit**, used when building the profile.
- **complain**, used for alerting of any deviating behavior.
- **enforce**, used for preventing any actions that are not part of the policy.

4.8 Summary of Proposed Architecture

To summarize, the prototype consists of two parts. Firstly a user space application in Go that is responsible for loading all eBPF programs and NRI-hooks, populating eBPF maps with the necessary information for enforcing rules and collecting information from the eBPF programs through ringbuffers.

Secondly a kernel space program written in C, consisting of the eBPF programs, eBPF-maps and a DFA-based path matcher is responsible for collecting information from the kernel when events are triggered and either sending them to the user space program if in audit mode, or deciding on whether the execution is allowed when in complain/enforce mode.

A simplified diagram describing the architecture is given in Figure 4.1.

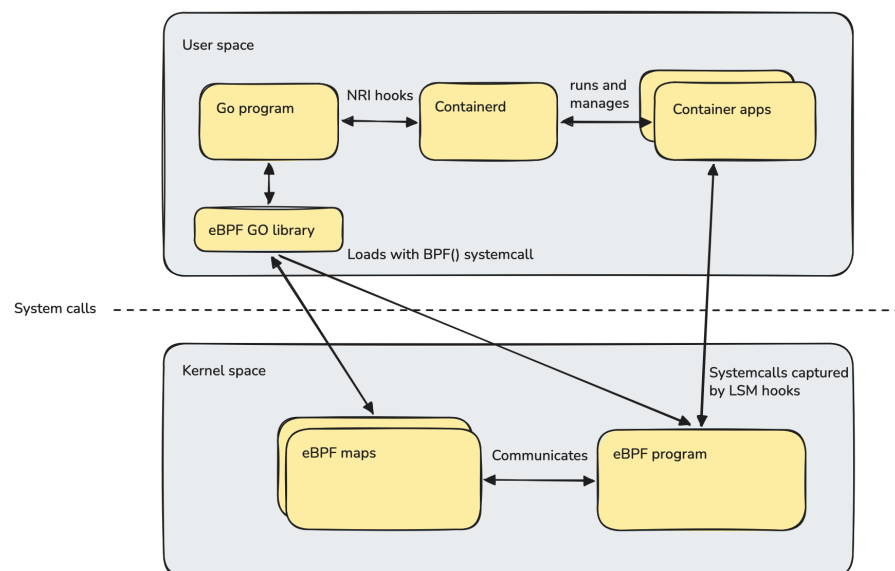


Figure 4.1: Overview of the proof of concept

The evaluation consists of setting up a test environment and verifying that the eBPF agent correctly attaches to the intended containers and hooks. The system is then evaluated, and the results are presented.

5.1 Test Environment

Initial testing was done locally using VMs with the CNCF cloud incubating project Lima. A Fedora x86_64 distribution was chosen due to its built in support for BPF LSM and the x86_64 architecture as it remains the most popular choice for cloud servers worldwide [45].

When conducting tests on other Linux distributions, such as Ubuntu, additional configuration steps were required to enable BPF LSM. First, support for BPF LSM had to be ensured at the kernel-level, involving verifying that the appropriate kernel version and configuration options were in place. Subsequently, BPF LSM needed to be explicitly enabled by modifying the bootloader configuration (e.g., GRUB) to include the necessary kernel parameters. These changes required a full system reboot before taking effect.

The Kubernetes cluster was configured using K3s, which is distributed as a single binary. This simplifies setup and maintenance while still providing a fully functional, production-grade Kubernetes cluster. A single node was used in the proof of concept to store the policy files. To enable data collection from multiple nodes and the distribution of profiles, integration with a central service would be required.

Clang, together with the LLVM toolchain, was used to compile the eBPF programs and they were loaded into the kernel using Libbpf with the bpftool cli tool before switching to ebpf-go.

With the Linux kernel version 5.8 the capability `CAP_BPF` was introduced [46]. The main purpose was to separate out BPF functionality instead of resorting to `CAP_SYS_ADMIN`. This meant that the only required capabilities when testing were `CAP_BPF`, `CAP_PERFMON` as well as `CAP_SYS_RESOURCE`.

5.2 Early Stage Validation Setup

To verify that the program worked as intended a simple deployment using Alpine Linux was used. Alpine Linux is minimalistic Linux distribution which fits our needs perfectly because of its small footprint.

The container was started in audit mode and shelled into to produce log data for the eBPF agent. Upon deletion of the container the profile was generated and was subsequently used for enforcing the deployment. Any action that had not been previously recorded was blocked and log messages created.

5.3 Evaluation of the Design

For the evaluation of the design 3 key metrics were chosen. Functionality, performance tradeoff and preventing actual attacks. The measured impact on the system should be low but it is pointless if it does not stop any threats. Tests were done in comparison to a baseline without any other agent deployed in the cluster. Additionally we compared generated profiles with profiles generated by the Bifrost Agent.

When benchmarking the monitored payloads were set to enforce mode. This was done both to verify that the solution worked but also to show how a real deployment in production could look like.

5.3.1 Blocking Requests

To show the functionality of the program a Alpine image was once again used but with a script accessing files in rapid succession. This includes reading, writing and executing files as well as creating socket connections.

5.3.2 Attack Prevention Benchmark

To show the design preventing real attacks a vulnerable Node.js Express application is used. The vulnerability in question is Server-Side Template Injection (SSTI) and enables an attacker to inject malicious JavaScript code, leading to remote code execution. An HTML textbox for the user to write in is commonly found in login forms and search boxes across many different websites. By entering a harmful payload surrounded by `#{...}` it gets evaluated and code gets executed.

5.3.3 Performance Benchmark

For the overhead measurement an open-source project by OWASP was used called OWASP Juice shop. It is a web application that is deliberately built to be insecure and contains vulnerabilities from OWASP top 10, which is a security document detailing all of the most common security risks for a web application. The test was automated using k6¹ to send requests to the API acting as a load generator.

¹k6 is an open-source load testing tool used to evaluate the performance of applications by simulating virtual users and generating traffic.

This includes many requests for loading the homepage and logging in repeatedly. We ran the test with 30 virtual users for 5 minutes in three different ways. A baseline(without our agent), while running our agent in auditing mode and lastly while running our agent in enforce mode, using the generated profile from the previous auditing. Five minutes and thirty virtual users were chosen to ensure that our application reached a steady state, as well as making sure that individual variance of each run would not negatively affect the result.

We also chose to micro-benchmark the eBPF C-code by measuring the time that was spent inside of our eBPF programs during invocation of the hooks. As eBPF programs are deployed globally across the kernel, they pick up events that are not a part of the container we are monitoring. In our implementation we filter out these events early and therefore we chose to ignore these events as they would artificially deflate the average latency of our eBPF programs. To trigger our three tracked hook points we ran the following command inside of our Alpine image:

```
1 i=1
2 while [ $i -le 20000 ]; do
3     nc 127.0.0.1 65535 < /dev/null 2>/dev/null
4     i=$((i + 1))
5 done
```

5.3.4 Comparison to AppArmor Generated Profiles

Finally, the profiles were compared to those produced by AppArmor. A set of eight distinct Node.js applications were used to generate audit data. Initially, the Bifrost Agent was used to generate AppArmor profiles and subsequently, the proposed proof of concept design.

The comparison aimed to evaluate the accuracy of the eBPF-based approach by identifying any discrepancies between the two sets of profiles. In particular, differences were analyzed with respect to both file path coverage and the associated permission sets using a script to compare the profiles.

The comparison was limited to direct file paths, excluding directory paths. This decision was made because the AppArmor solution iteratively included parent directories in the generated profile whereas our profiles only included directories if they were accessed explicitly (e.g., via commands such as `ls`). This only means that our profile is somewhat more restrictive, allowing direct directory access only when the directory has been directly accessed by a process before. To more closely replicate an AppArmor profile, all parent directories associated with each observed file access could be included automatically during profile generation.

5.4 Results

5.4.1 Blocking Requests Benchmark

For the initial test the script 7.1 was run on the Alpine container in audit mode generating the simple profile seen in Figure 5.1. Key entries include the `test.sh` file which has read, write and execute permissions and the `test2.sh` file which only has write permissions.

When run in enforce mode the same test is repeated but this time with a few additional commands as seen in 7.2. As these actions were not recorded during the audit phase they are blocked, which can be seen in 5.2.

In addition to new read, writes and executes being blocked there was also two tests using the socket hook. Wget was first used to retrieve a webpage in the auditmode, but this does not allow the use of ping to create a socket because it uses another protocol.

Extra data was also collected beyond the expected path patterns, appearing to be specific to the Alpine Linux distribution used in this setup. Paths beginning with a forward slash followed by a number do not correspond to actual directories in the filesystem root, but instead appear to map to subdirectories within the `/proc` filesystem. Furthermore, some entries do not resemble traditional filesystem paths at all, such as `ipc:[4026533940]`. In these cases, the LSM hook is triggered within the context of process namespaces, and such patterns must therefore be handled through explicitly defined exceptions.

We do not consider this to pose a security concern, as these pseudo-file paths can either be handled explicitly within the eBPF programs or included in the generated ruleset and processed that way.

5.4.2 Attack Prevention Benchmark

In the baseline scenario, without enforcement enabled, the SSTI payload was able to execute and access files within the container environment, demonstrating the presence of a runtime vulnerability.

When the BPF LSM enforcement solution was enabled, the same attack was executed under identical conditions. It successfully blocked the server-side template injection attack from accessing the sensitive files on the vulnerable server triggering the `file_open` hook. Attempts to read sensitive files such as `/etc/passwd` resulted in permission errors, indicating that the operation was blocked.

It was also observed that even though the attack was prevented there was still a stacktrace shown due to a design flaw in how the application was built. This is bad from a security perspective as this represents additional information disclosure risk, providing an attacker with insight into how the application is built.

5.4.3 Performance Benchmark

The results from the API benchmark test using the k6 script 7.3 can be seen in Table 5.1. To ensure reproducibility of the performance results, a brand new virtual machine was created using 4 x86_64 vCPUs and 16 GB of RAM from the cloud provider Evroc using a standard Ubuntu image with the 6.8.0-110-generic kernel. Nothing else was deployed on the K3s cluster apart from the monitored application that could lead to higher latency.

The eBPF C-code benchmark gave the following results. The number of times each hook was activated can be seen in Figure 5.3 and the average time in Figure 5.4.

Table 5.1: HTTP Request Duration (Latency) across eBPF execution modes. All values are in milliseconds (ms).

Mode	Median	Average	p(90)	p(95)
Baseline (Without)	27.51	61.06	151.52	207.11
Audit	29.17	64.51	167.68	229.04
Enforce	29.62	70.66	177.77	255.81

5.4.4 Comparison to AppArmor Generated Profiles

In total eight AppArmor profiles and eight proof of concept profiles were created. A subset of the data can be found in Figure 5.5.

A script was used to compare the file paths and their corresponding permission sets, producing the results shown in Table 5.2. The analysis indicates that 96% of the exact file paths identified in the AppArmor dataset were also detected by the eBPF program. When accounting for functionally equivalent file paths containing timestamps and inferred rules covered in eBPF by exact matches, an accuracy of 99% was achieved.

The results also include the number of functionally different permissions and paths present in AppArmor but not in eBPF, and vice versa and is further explained in the discussion. An example of the permission mismatches can be found in Figure 5.6 with the only difference being an extra i for each execute and the eBPF agent logging an extra execute on container startup.

```

> cat /test/test-alpine_policy.json
{/etc/resolv.conf:r}
{/19/task/19/fd:r}
{/25/task/25/attr/apparmor/exec:w}
{/usr/bin/ssl_client:rx}
{/dev/pts/*:rw}
{/38/task/38/fd:r}
{/ :r}
{ipc:[4026533939]:r}
{/etc/hosts:r}
{/proc/*/oom_score_adj:rw}
{/dev/pts/*:rw}
{/44/task/44/attr/apparmor/exec:w}
{cgroup:[4026534077]:r}
{/etc/passwd:r}
{/13/task/13/attr/apparmor/exec:w}
{/25/setgroups:r}
{/1/task/1/fd:r}
{/proc/*/oom_score_adj:rw}
{/13/task/13/fd:r}
{/32/task/32/attr/apparmor/exec:w}
{/etc/ssl/certs:r}
{/sys/kernel/mm/transparent_hugepage/hpage_pmd_size:r}
{/test.sh:rw}
{/44/setgroups:r}
{/19/task/19/attr/apparmor/exec:w}
{/38/setgroups:r}
{/38/task/38/attr/apparmor/exec:w}
{uts:[4026533938]:r}
{/dev/pts/*:rw}
{/test2.sh:w}
{/32/task/32/fd:r}
{/proc/*/oom_score_adj:rw}
{/proc/*/oom_score_adj:rw}
{/lib/libcrypto.so.3:r}
{/lib/ld-musl-x86_64.so.1:r}
{/runc:rx}
{/7/setgroups:r}
{/lib/libssl.so.3:r}
{/etc/ssl/openssl.cnf:r}
{/44/task/44/fd:r}
{/proc/*/oom_score_adj:rw}
{mnt:[4026533941]:r}
{/13/setgroups:r}
{/proc/*/oom_score_adj:rw}
{/32/setgroups:r}

```

Figure 5.1: Automatically generated Alpine file access profile.

```

> ./demos/testAlpine/enforcescript.sh
+ echo "echo \"hello\"" > test.sh
+ chmod 777 test.sh
+ ./test.sh
hello
+ echo "echo \"hello\"" > test2.sh
+ chmod 777 test2.sh
+ ./test2.sh
sh: ./test2.sh: Permission denied
command terminated with exit code 126
+ ls /
bin      index.html  opt      sbin      test2.sh
dev      lib         proc     srv        tmp
etc      media       root     sys        usr
home     mnt        run      test.sh    var
+ cat /dev
cat: can't open '/dev': Operation not permitted
command terminated with exit code 1
+ wget https://www.google.com
Connecting to www.google.com (142.251.151.119:443)
wget: can't open 'index.html': File exists
command terminated with exit code 1
+ ping -c 4 8.8.8.8
PING 8.8.8.8 (8.8.8.8): 56 data bytes
ping: permission denied (are you root?)
command terminated with exit code 1
+ echo "NewUser:!:20531:0:99994:::" >> /etc/shadow
sh: can't create /etc/shadow: Operation not permitted
command terminated with exit code 1
+ cat /etc/shadow
cat: can't open '/etc/shadow': Operation not permitted
command terminated with exit code 1

```

Figure 5.2: Demo of enforce file access, execution and sockets.

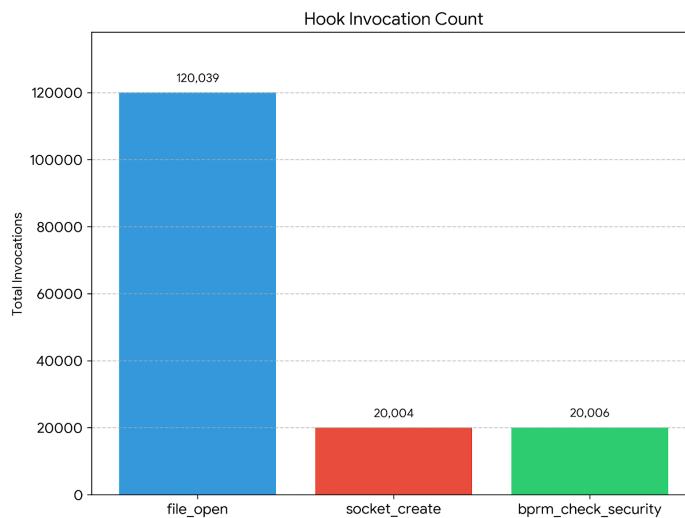


Figure 5.3: The number of times each hook was triggered.

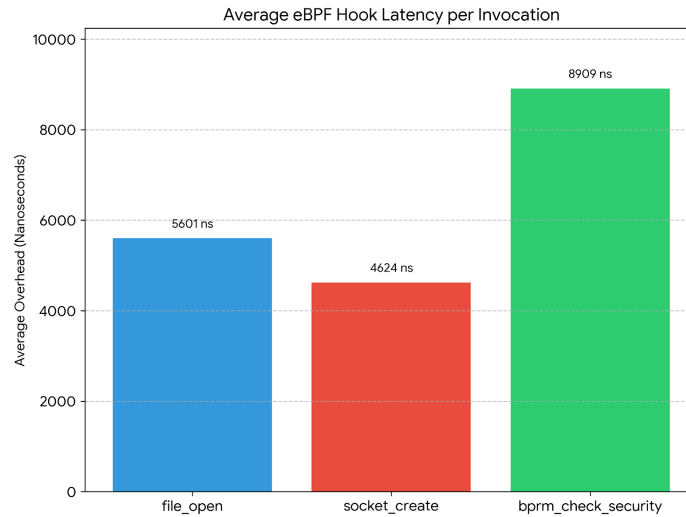


Figure 5.4: The execution time of each hook.

```

"/app/" r,
"/app/index.js" r,
"/app/models/" r,
"/app/models/bank.js" r,
"/app/models/library.js" r,
"/app/node_modules/" r,
"/app/node_modules/accepts/" r,
"/app/node_modules/accepts/index.js" r,
"/app/node_modules/accepts/package.json" r,
"/app/node_modules/array-flatten/" r,
"/app/node_modules/array-flatten/array-flatten.js" r,
"/app/node_modules/array-flatten/package.json" r,
"/app/node_modules/body-parser/" r,
"/app/node_modules/body-parser/index.js" r,
"/app/node_modules/body-parser/lib/" r,
"/app/node_modules/body-parser/lib/read.js" r,
"/app/node_modules/body-parser/lib/types/" r,
"/app/node_modules/body-parser/lib/types/json.js" r,
"/app/node_modules/body-parser/lib/types/raw.js" r,
"/app/node_modules/body-parser/lib/types/text.js" r,
"/app/node_modules/body-parser/lib/types/urlelncoded.js" r,
"/app/node_modules/body-parser/package.json" r,
"/app/node_modules/bytes/" r,
"/app/node_modules/bytes/index.js" r,
"/app/node_modules/bytes/package.json" r,
"/app/node_modules/call-bind-apply-helpers/" r,
"/app/node_modules/call-bind-apply-helpers/actualApply.js" r,
"/app/node_modules/call-bind-apply-helpers/functionApply.js" r,
"/app/node_modules/call-bind-apply-helpers/functionCall.js" r,
"/app/node_modules/call-bind-apply-helpers/index.js" r,
"/app/node_modules/call-bind-apply-helpers/package.json" r,
"/app/node_modules/call-bind-apply-helpers/reflectApply.js" r,
"/app/node_modules/call-bound/" r,
"/app/node_modules/call-bound/index.js" r,
"/app/node_modules/call-bound/package.json" r,
"/app/node_modules/content-disposition/" r,
"/app/node_modules/content-disposition/index.js" r,
"/app/node_modules/content-type/" r,
"/app/node_modules/content-type/package.json" r,
"/app/node_modules/cookie-signature/" r,
"/app/node_modules/cookie-signature/index.js" r,
"/app/node_modules/cookie/" r,
"/app/node_modules/cookie/package.json" r,
"/app/node_modules/debug/" r,
"/app/node_modules/debug/package.json" r,
"/app/node_modules/debug/src/" r,
"/app/node_modules/debug/src/index.js" r,
"/app/index.js:r,
/app/models/bank.js:r,
/app/models/library.js:r,
/app/node_modules/accepts/index.js:r,
/app/node_modules/accepts/package.json:r,
/app/node_modules/array-flatten/array-flatten.js:r,
/app/node_modules/array-flatten/package.json:r,
/app/node_modules/body-parser/index.js:r,
/app/node_modules/body-parser/lib/read.js:r,
/app/node_modules/body-parser/lib/types/json.js:r,
/app/node_modules/body-parser/lib/types/raw.js:r,
/app/node_modules/body-parser/lib/types/text.js:r,
/app/node_modules/body-parser/lib/types/urlelncoded.js:r,
/app/node_modules/body-parser/package.json:r,
/app/node_modules/bytes/index.js:r,
/app/node_modules/bytes/package.json:r,
/app/node_modules/call-bind-apply-helpers/actualApply.js:r,
/app/node_modules/call-bind-apply-helpers/functionApply.js:r,
/app/node_modules/call-bind-apply-helpers/functionCall.js:r,
/app/node_modules/call-bind-apply-helpers/index.js:r,
/app/node_modules/call-bind-apply-helpers/package.json:r,
/app/node_modules/call-bind-apply-helpers/reflectApply.js:r,
/app/node_modules/call-bound/index.js:r,
/app/node_modules/call-bound/package.json:r,
/app/node_modules/content-disposition/index.js:r,
/app/node_modules/content-type/index.js:r,
/app/node_modules/content-type/package.json:r,
/app/node_modules/cookie-signature/index.js:r,
/app/node_modules/cookie/index.js:r,
/app/node_modules/cookie/package.json:r,
/app/node_modules/debug/package.json:r,
/app/node_modules/debug/src/debug.js:r,
/app/node_modules/debug/src/index.js:r

```

Figure 5.5: A sorted selection of AppArmor (left) and eBPF (right) generated profiles for the application "api-test-projects".

```

=== Permission mismatches ===
/usr/local/lib/node_modules/npm/bin/npm-cli.js: AppArmor=irx eBPF=rx
/usr/local/bin/node: AppArmor=irx eBPF=rx
/usr/local/bin/docker-entrypoint.sh: AppArmor=r eBPF=rx
/bin/busybox: AppArmor=irx eBPF=rx

```

Figure 5.6: Permission mismatches for the "api-test-projects" test.

Tested Program	# File paths in AppArmor	# File paths in eBPF	# Exact file matches	# Functionally same file matches	# Functionally different paths in AppArmor but not in eBPF	# Functionally different paths in eBPF but not in AppArmor	# Functionally different permissions
api-test-projects	343	360	334	338	5	21	1
author-royalty-api	346	365	339	343	3	22	1
basic-express-apis	488	499	471	479	9	24	1
express-digi	338	356	330	335	3	22	1
node-ram-db	493	510	479	483	10	27	1
product-node-app	376	362	336	373	3	22	1
tally-counter	457	459	443	454	3	12	1
the-data-hub	330	349	323	327	3	22	1
Total	3171	3260	3055	3132	39	181	8

Table 5.2: Comparison of file path coverage between AppArmor and eBPF

The discussion is divided by the experimental tests used in the evaluation. Starting with proving that the eBPF-based agent worked, doing a performance benchmark and testing an actual attack. Finally the comparison with the AppArmor generated profiles and eBPF generated profiles discusses the differences between the two solutions.

6.1 Blocking Requests Benchmark

The eBPF agent was able to prevent all new unrecorded file accesses including reads, writes and executes that were not present in the stored profile. It was however not able to stop inode creation which happens before the LSM hooks activate. AppArmor uses additional LSM hooks such as `inode_create` that trigger on inode creation and this would be necessary to expand the current functionality of the agent.

Even though eBPF offers far more hook points than AppArmor, there was insufficient time to test all of them. The chosen LSM hooks represent the core idea of AppArmor which is built around working with file paths and proves that an eBPF solution is a viable alternative.

Part of the time constraint is due to each hook providing different context and access to different internal Linux kernel structs. Crucially the `inode_create` hook does not provide the path struct required to get the path name using the `bpf_d_path` helper function. It could be possible to use the available dentry and inode structs, similarly to the approach suggested by Beck [28], but further work would be needed to research this.

6.2 Attack Prevention Benchmark

The attack prevention benchmark shows that the agent helps in preventing server-side template injection attacks. A poorly built website that does not parse user input to escape dangerous characters gets an extra layer of protection but it does not fully prevent all sorts of attacks. A login form connected straight to a SQL database would still be vulnerable to a SQL injection attack. For such cases, mitigation requires for instance prepared statement logic, separating the SQL logic from the supplied data.

No false positives were observed during the conducted experiments, but this could pose a big problem with bigger applications that depend on other container images and external services. For the enforcement policy to be viable in production it must be sufficiently reliable to avoid disrupting entire deployments and causing service outages.

Consequently, extended testing under long-running production-like workloads is required. In particular, deploying the system in the permissive complain mode to identify and filter out potential false positives before enabling strict enforcement.

There is also the need to assess how well the monitoring scales for larger deployments spread across more nodes. With multiple microservices communicating with each other, the volume of data increases, which may impact the performance.

6.3 Performance Benchmark

The API benchmark shows that there is a slowdown in the request duration, particularly during enforce mode, when the requested paths need to be resolved and possibly checked using globbing in the eBPF programs. This becomes especially apparent from the latency per hook invocation tests. Despite each hook invocation taking in the order of a few microseconds multiple calls accumulate and result in noticeable overhead.

Our C-benchmark shows that there is, on average, an overhead of a few microseconds ($<10\mu\text{s}$) for each hook. While a direct micro-benchmark comparison against AppArmor was outside the scope for this work, having a baseline comparison remains a critical area for future evaluation. Given that standard network latencies in cloud environments are typically measured in milliseconds, an induced kernel level overhead of under $10\mu\text{s}$ is negligible for our applications. However, this overhead could compound for applications with many file accesses, highlighting the need to evaluate acceptability on a per-application basis. Additionally it is important to point out that these measured values are only an average, and that there could be execution paths that experience higher latency. Enforcement of regular rules, without globbing, does not scale with the directory depth or filepath size, as they do not pass through the DFA-based path matcher. However, files which do have globbing elements would scale slower if the directory depth or filepath size increased, further work exploring the effect of large and complex globbing rules on the performance would be interesting to explore.

Additionally it is important to mention that the test that was run is not very accurate for a real-world application, only opening, executing and creating the same files and sockets over and over again. This could result in the agent performing better for these tests than it would on a real-world example, both because the system can utilize the cache better, but also because the profiles are smaller and simpler.

Another interesting point is the amount of times each hook was triggered. The `bprm_check_security` and `socket_create` were triggered once every iteration of the loop, while the `file_open` hook was triggered 6 times. Taking into account how often each hook is triggered can be important in deciding how optimized it needs to be, or how much time should be spent trying to optimize it.

6.4 Comparison to AppArmor Generated Profiles

The comparison between the AppArmor profiles and the profiles generated by the proposed proof of concept design indicates that good hook accuracy can be achieved using eBPF, similar to that of AppArmor. Thanks to the custom-built path matching even the globbing behavior of paths used in AppArmor can be replicated as seen in the `/proc/*` directories.

Path discrepancies arise from resolving globbed patterns (e.g., `/sys/fs/cgroup/memory.*`) and timestamped logs (e.g., `/root/.npm/_logs/2025-04-02T14_34_57_955Z-debug-0.log`) to exact matches. The quality of these rules depends on the automated rule engine, not the enforcement tool (AppArmor or eBPF). Our current engine is limited to basic inferences (e.g., `/proc/<PID>/` to `/proc/*`) and whole-directory globbing, lacking partial match support. However, because it captures the same system events as AppArmor, achieving equivalent rules is theoretically possible.

Almost all of the permissions were identical. The majority of mismatches can be explained by how AppArmor handles the execute flag. When execute permissions are granted to an executable, AppArmor requires an additional modifier flag dictating how the profile should be applied to the new program [47]. In our case, the only relevant modifier is the `ix` (inherit execute) flag, which forces the new process to inherit the AppArmor profile of the calling process. The automatically generated AppArmor profiles use the inherit flag by default. This is logical for enforcing a profile on a container, as any newly spawned process should be considered part of the container and accordingly follow the same rules. The eBPF tool does not enforce rules on a strictly per-process basis, but, as mentioned earlier, enforces them based on the cgroup ID of the running process. Therefore, our `x` flag is functionally equivalent to the `ix` flag in AppArmor.

Apart from this there exists one actual permission mismatch, `/usr/local/bin/docker-entrypoint.sh`, recurring in all policies. The AppArmor profile contains only read, while the eBPF profile also contains an execute permission. The file is the startup script inside the container image and is used when the container is first started, doing setup and before handing execution over to the actual application. This difference stems from how we and the kernel handle interpreted scripts. When running `./docker-entrypoint.sh`, the kernel tries to execute the file. Upon reading the shebang (`#!/bin/sh`) the kernel realizes that it needs to invoke the shell interpreter, with the file as the argument to be read. AppArmor seems to either only be capturing the execution of the interpreter, or capturing both events and correctly converting it into an execution of the interpreter and a read of the file. While the AppArmor profile is technically correct, as the file is eventually only being read by the interpreter, the eBPF profile's inclusion of the execute permissions more accurately represents the initial intent of the execution.

A notable limitation of the profile comparison is the exclusive use of Node.js applications for the evaluation dataset. Applications of a different type may exhibit different behavior, exposing missing features of our tool or resulting in a different result in the amount of matching file paths. This highlights the need for further comparative testing across a wider variety of programming languages and architectures, which would be needed to confirm the solution's generalizability.

An eBPF solution can be used to successfully lock down containers similar to AppArmor. Thanks to its many hooks all forms of telemetry can be collected and runtime enforcement can be done using the LSM framework. Mapping of processes to their pod names is possible via the use of their cgroup IDs and BPF maps means that lookups of stored rules can be done fast. The NRI API allows for hooking into container creation events and synchronous loading of profiles before the monitored containers get a chance to run any harmful code.

Path matching was proven to be a bigger challenge than anticipated due to the inherent limitations of using eBPF and its lack of Regex as well as the different file paths generated. This meant that a few manual rules may be required for file paths depending on the program type. While this is a severe limitation for scalability, it is a trade-off that is also seen in current AppArmor tools. One area of improvement lies in the open sourcing of Meta's eBPF regex engine [44] or parallel development of a similar tool which would enable more granular automated rules. Another way would be the configuration or improvement of automatic rule generators such as Lic-Sec [43] to process eBPF events.

Other limitations also exist. For a solution using BPF LSM to be a viable alternative it needs to be enabled in the distribution of all Kubernetes nodes. Depending on where the cluster is run this may or may not be an easy task, possibly warranting full reboots of the nodes and the disruption involved in doing so. Any eBPF logic also needs to pass through the eBPF verifier, meaning a limited instruction count and complexity.

It remains to be seen what the future holds for eBPF, but it is clear that it has sparked an interest in bigger companies such as Meta and Google, and will likely be important for custom defined rules as security becomes more important in cloud deployments.

7.1 Main Findings

Answering the research questions introduced in the introduction, it is clear that eBPF can be effectively used for inline security enforcement. The overhead of the monitoring program is manageable, and it successfully prevents a server-side template injection attack from extracting confidential information.

Furthermore, eBPF enables the generation of profiling data for a container's

runtime behavior through the use of cgroups. By storing policies between runs, it is possible to compensate for ephemeral identifiers.

Finally, although not included in the proposed design, eBPF allows for more fine-grained profiling than was previously achievable with MAC-based solutions. By not being limited to LSM hooks, eBPF provides a more versatile mechanism for attaching to a wide range of functions and data related to containers. In addition to hooking into kprobes and tracepoints, the extra networking capabilities enables inspection and filtering of network traffic. As seen with the socket hook, the amount and type of information that can be extracted depend on the desired level of granularity.

7.2 Possible Applications

The DaemonSet can be applied across a wide variety of payloads on Kubernetes clusters, making it a flexible solution for runtime enforcement. Any service that can benefit from a restricted runtime environment would be a promising candidate, albeit with some possible tweaking for different kinds of monitored payloads. In particular, small to medium sized organizations seeking to enhance their security posture may benefit from having such a simple to setup solution, integrating into existing cloud infrastructure.

Overall, there are clearly use cases where this BPF LSM solution is better and more flexible than current solutions. One example is that eBPF maps can be dynamically updated without requiring a redeployment of the monitored application to apply a new security profile. There is also the case where an organisation might want certain critical parts of an application to only be monitored while others are enforced, to not bring everything down in the case of false alerts.

It is still important to bear in mind that modules like AppArmor and SELinux have evolved over many years and the development behind them is far more scrutinized than what any small project can hope to match. A possible compromise could be achieved by combining a custom BPF LSM solution with AppArmor or SELinux.

7.3 Future Work

A few areas for improvement still remain. Firstly, using a more sophisticated regex solution for path matching similar to Meta's BpfJailer. This could offer a smaller attack surface by further limiting access to restricted files and directories based on automatically generated names like timestamps. However, doing this could lead to a higher amount of overhead so one must weigh whether or not the improved granularity is worth the extra cost.

Secondly, to reduce the manual effort involved in adding rules for previously encountered libraries and programming languages, additional generalized rules can be inferred, similar to replacing `/proc/<PID>/` with `/proc/*`. This does not solve the problem for new libraries that have not been seen before but AppArmor also struggles from the very same problem.

Thirdly, using more LSM hooks to retrieve more data and enforce more actions. While the proposed solution captures the core idea of automatic profile generation and path matching it is not impervious to all kinds of access attempts. For example one could prevent the creation of inodes as mentioned in Chapter 6.1 or add support for other types of file accesses such as memory mapping or file locking.

Finally, a solution for uploading the generated profiles to a server for storage and further processing. This could be integrated into a portal similar to the Bifrost Agent or other security platforms with an alerting system to notify the organization.

Bibliography

- [1] Gartner, Inc. “Gartner says more than half of enterprise it spending in key market segments will shift to the cloud by 2025,” Accessed: Jan. 22, 2026. [Online]. Available: <https://www.gartner.com/en/newsroom/press-releases/2022-02-09-gartner-says-more-than-half-of-enterprise-it-spending>.
- [2] P. A. Networks. “What is a container?” Accessed: Jan. 21, 2026. [Online]. Available: <https://www.paloaltonetworks.com/cyberpedia/what-is-a-container>.
- [3] D. Bernstein, “Containers and cloud: From lxc to docker to kubernetes,” *IEEE cloud computing*, vol. 1, no. 3, pp. 81–84, 2014.
- [4] C. Pahl, “Containerization and the paas cloud,” *IEEE Cloud Computing*, vol. 2, no. 3, pp. 24–31, 2015.
- [5] “Kubernetes.” [Online]. Available: <https://kubernetes.io/>.
- [6] Kubernetes Documentation. “Daemonset,” Accessed: Jan. 20, 2026. [Online]. Available: <https://kubernetes.io/docs/concepts/workloads/controllers/daemonset/>.
- [7] eBPF Project. “eBPF,” Accessed: Jan. 20, 2026. [Online]. Available: <https://ebpf.io/>.
- [8] eBPF Project. “eBPF loading,” Accessed: Apr. 6, 2026. [Online]. Available: <https://docs.ebpf.io/linux/>.
- [9] Linux man-pages project. “Bpf-helpers(7) - linux manual page,” Accessed: Apr. 6, 2026. [Online]. Available: <https://man7.org/linux/man-pages/man7/bpf-helpers.7.html>.
- [10] S. Smalley, T. Fraser, and C. Vance. “Linux security modules: General security hooks for linux,” Accessed: Jan. 22, 2026. [Online]. Available: <https://docs.kernel.org/security/lsm.html>.

- [11] The Linux Kernel Documentation. “Apparmor,” Accessed: Jan. 21, 2026. [Online]. Available: <https://www.kernel.org/doc/html/v5.8/admin-guide/LSM/apparmor.html>.
- [12] Ubuntu Documentation Project. “Apparmor – ubuntu server documentation,” Accessed: Jan. 22, 2026. [Online]. Available: <https://documentation.ubuntu.com/server/how-to/security/apparmor/>.
- [13] “Lsm bpf programs — the linux kernel documentation,” Accessed: Jan. 22, 2026. [Online]. Available: https://docs.kernel.org/bpf/prog_lsm.html.
- [14] The Linux Kernel Documentation. “Seccomp bpf (secure computing with filters),” Accessed: Jan. 21, 2026. [Online]. Available: https://www.kernel.org/doc/html/v4.19/userspace-api/seccomp_filter.html.
- [15] Cloud Native Computing Foundation. “Cncf landscape guide,” Accessed: Jan. 21, 2026. [Online]. Available: <https://landscape.cncf.io/guide#introduction>.
- [16] Bifrost Security. “Bifrost documentation,” Accessed: Jan. 26, 2026. [Online]. Available: <https://docs.bifrostsec.com/guides/get-started/>.
- [17] KubeArmor. “Kubearmor documentation,” Accessed: Jan. 27, 2026. [Online]. Available: <https://docs.kubearmor.io/kubearmor>.
- [18] KubeArmor. “Kubearmor github,” Accessed: Jan. 27, 2026. [Online]. Available: <https://github.com/kubearmor/KubeArmor/>.
- [19] Falco. “Falco docs,” Accessed: Jan. 28, 2026. [Online]. Available: <https://falco.org/docs/>.
- [20] Falco. “Falcosidekick,” Accessed: Apr. 6, 2026. [Online]. Available: <https://falco.org/docs/concepts/outputs/forwarding/>.
- [21] Falco. “Falco Talon,” Accessed: Apr. 6, 2026. [Online]. Available: <https://falco.org/blog/falco-talon-v0-1-0/>.
- [22] Aqua Security. “Tracee,” Accessed: Jan. 28, 2026. [Online]. Available: <https://aquasecurity.github.io/tracee/latest/docs/overview/>.
- [23] Isovalent. “About us,” Accessed: Jan. 21, 2026. [Online]. Available: <https://isovalent.com/about-us/>.
- [24] Tetragon. “Tetragon documentation.” [Online]. Available: <https://tetragon.io/docs/overview/>.
- [25] Cilium. “Ebpf-go,” Accessed: Jan. 28, 2026. [Online]. Available: <https://github.com/cilium/ebpf>.

- [26] SBOB. “Software bill of behavior sbob,” Accessed: Jan. 27, 2026. [Online]. Available: <https://github.com/k8sstormcenter/bob?tab=readme-ov-file>.
- [27] Constanze Roedig. “Meet BOB: the supply chain provided “bill of behaviour” for anomaly-based runtime security,” Accessed: Apr. 10, 2026. [Online]. Available: <https://cloudnativesummit.de/schedule?event=904105>.
- [28] E. Beck, “Leveraging ebpf to produce container audit data for the purpose of apparmor profile generation,” 2023.
- [29] B. Gbadamosi et al., “The ebpf runtime in the linux kernel,” *arXiv preprint arXiv:2410.00026*, 2024.
- [30] ebpf. “Ebpf-updates-2021-02,” Accessed: Jan. 29, 2026. [Online]. Available: <https://ebpf.io/blog/ebpf-updates-2021-02/#did-you-know-program-size-limit>.
- [31] H. Zhu and C. Gehrmann, “Apparmor profile generator as a cloud service,” in *CLOSER*, 2021, pp. 45–55.
- [32] H. Zhu and C. Gehrmann, “Kub-sec, an automatic kubernetes cluster apparmor profile generation engine,” in *2022 14th International Conference on COMMunication Systems & NETworkS (COMSNETS)*, IEEE, 2022, pp. 129–137.
- [33] T. Theodoropoulos et al., “Security in cloud-native services: A survey,” *Journal of Cybersecurity and Privacy*, vol. 3, no. 4, pp. 758–793, 2023.
- [34] Q. Zeng, M. Kavousi, Y. Luo, L. Jin, and Y. Chen, “Full-stack vulnerability analysis of the cloud-native platform,” *Computers & Security*, vol. 129, p. 103 173, 2023.
- [35] H. Lu, X. Du, D. Hu, S. Su, and Z. Tian, “Bpfguard: Multi-granularity container runtime mandatory access control,” *IEEE Transactions on Cloud Computing*, vol. 13, no. 2, pp. 629–640, 2025. DOI: 10.1109/TCC.2025.3551838.
- [36] S. J. Songi Gwak Thien-Phuc Doan, “Container instrumentation and enforcement system for runtime security of kubernetes platform with ebpf,” *Intelligent Automation & Soft Computing*, vol. 37, no. 2, pp. 1773–1786, 2023, ISSN: 2326-005X. DOI: 10.32604/iasc.2023.039565. [Online]. Available: <http://www.techscience.com/iasc/v37n2/53245>.
- [37] S. Zehra, H. J. Syed, F. Samad, and U. Faseeha, “Desfam: An adaptive ebpf and ai-driven framework for securing cloud containers in real time,” *IEEE Access*, vol. 13, pp. 139 203–139 224, 2025. DOI: 10.1109/ACCESS.2025.3592192.

- [38] J. Her, J. Kim, J. Kim, and S. Lee, “An in-depth analysis of ebpf-based system security tools in cloud-native environments,” *IEEE Access*, 2025.
- [39] K. Xu, X. Wang, L. Li, and J. Gao, “Ebpf-sec: A defensive framework against ebpf attacks on containers,” in *2024 IEEE Symposium on Computers and Communications (ISCC)*, IEEE, 2024, pp. 1–7.
- [40] F. J. Bertinatto, D. Arioza Almeida, J. Nobre, and L. Z. Granville, “Ebpf-based approach to tracing system calls and predicting privilege escalation attacks,” in *GLOBECOM 2024 - 2024 IEEE Global Communications Conference*, 2024, pp. 3081–3086. DOI: 10.1109/GLOBECOM52923.2024.10901578.
- [41] Linux source code. “Lsm hooks definition,” Accessed: Mar. 12, 2026. [Online]. Available: https://elixir.bootlin.com/linux/v6.18.2/source/include/linux/lsm_hook_defs.h.
- [42] Linux man-pages project. “Glob(7) - linux manual page,” Accessed: Mar. 16, 2026. [Online]. Available: <https://www.man7.org/linux/man-pages/man7/glob.7.html>.
- [43] H. Zhu and C. Gehrmann, *Lic-sec: An enhanced apparmor docker security profile generator*, 2020. arXiv: 2009.11572 [cs.CR]. [Online]. Available: <https://arxiv.org/abs/2009.11572>.
- [44] Justin Ngai. “Kernel-resident regex and jails: Dfa-powered ebpf filtering and certificate-safe agent isolation at fleet scale,” Accessed: Mar. 16, 2026. [Online]. Available: <https://lpc.events/event/19/contributions/2176/>.
- [45] International Data Corporation, *Worldwide server market revenue increased 61% during the third quarter of 2025*, Accessed: 2026-01-26, Dec. 2025. [Online]. Available: <https://www.idc.com/getdoc.jsp?containerId=prUS54034325>.
- [46] Linux man pages. “Capabilities(7) — linux manual page,” Accessed: Jan. 29, 2026. [Online]. Available: <https://man7.org/linux/man-pages/man7/capabilities.7.html>.
- [47] Novell, Inc., *Novell apparmor administration guide: File permission access modes*, 2010. Accessed: May 5, 2026. [Online]. Available: https://www.novell.com/documentation/apparmor/apparmor201_sp10_admin/data/bx5bmls.html.

7.4 Test Scripts

```
1 #!/bin/bash
2
3 # List of commands to run
4 commands=(
5   'echo "echo \"hello\"" > test.sh'
6   'chmod 777 test.sh'
7   './test.sh'
8   'echo "echo \"hello\"" > test2.sh'
9   'chmod 777 test.sh'
10  'ls /'
11  'wget https://www.google.com'
12 )
13
14 # Loop over each command
15 for cmd in "${commands[@]}; do
16   echo "+ $cmd" # print the command
17   sudo kubectl exec -itf demos/testAlpine/deployment.yaml --
18     sh -c "$cmd"
19 done
```

Listing 7.1: Script for testing audit

```

1 #!/bin/bash
2
3 # List of commands to run
4 commands=(
5   'echo "echo \"hello\"" > test.sh'
6   'chmod 777 test.sh'
7   './test.sh'
8   'echo "echo \"hello\"" > test2.sh'
9   'chmod 777 test.sh'
10  './test2.sh'
11  'ls /'
12  'cat /dev'
13  'wget https://www.google.com'
14  'ping -c 4 8.8.8.8'
15  'echo "NewUser!!:20531:0:99994:::" >> /etc/shadow'
16  'cat /etc/shadow'
17 )
18
19
20 # Loop over each command
21 for cmd in "${commands[@]"; do
22   echo "+ $cmd" # print the command
23   sudo kubectl exec -itf demos/testAlpine/deployment.yaml --
24     sh -c "$cmd"
25 done

```

Listing 7.2: Script for testing enforce

```

1 import http from 'k6/http';
2 import { sleep, check } from 'k6';
3
4 // CONFIGURATION
5 const BASE_URL = __ENV.BASE_URL || 'http://localhost:31000';
6 const USER_EMAIL = 'admin@juice-sh.op';
7 const USER_PASSWORD = 'admin123';
8
9 export const options = {
10   vus: 30,
11   duration: '5m',
12   thresholds: {
13     http_req_duration: ['p(95)<500'],
14   },
15 };
16
17 export default function () {
18   // 1. Load homepage
19   let res = http.get(`${BASE_URL}/`);
20   // console.log("res: " + JSON.stringify(res, null, 2))
21   check(res, { 'homepage loaded': (r) => r.status === 200 });
22   sleep(1);
23
24   // 2. Get product list

```

```
25 res = http.get(`${BASE_URL}/rest/products/search?q=');
26 check(res, { 'products fetched': (r) => r.status === 200 });
27 sleep(1);
28
29 // 3. Login
30 res = http.post(`${BASE_URL}/rest/user/login`, JSON.
31   stringify({
32     email: USER_EMAIL,
33     password: USER_PASSWORD,
34   }), {
35     headers: { 'Content-Type': 'application/json' },
36   });
37
38 check(res, { 'login request sent': (r) => r.status === 200
39   });
40
41 let authToken = '';
42 if (res.status === 200) {
43   try {
44     authToken = JSON.parse(res.body).authentication.token;
45     console.log('Login successful, token obtained.');
```

```
    res.body);  
  }  
71 } else {  
72   console.error('Adding item to basket failed:', res.status,  
73     res.body);  
74 }  
75  
76 sleep(1);  
77  
78 // 5. Checkout (optional)  
79 res = http.post(`${BASE_URL}/rest/basket/1/checkout`, {}, {  
80   headers: { Authorization: `Bearer ${authToken}` },  
81 });  
82 check(res, { 'checkout attempted': (r) => [200, 401, 500].  
    includes(r.status) });  
83  
84 sleep(2);  
85 }
```

Listing 7.3: k6 script for the API-test



LUND
UNIVERSITY

Series of Master's theses
Department of Electrical and Information Technology
LU/LTH-EIT 2026-1127
<http://www.eit.lth.se>