# Leveraging eBPF to Produce Container Audit Data for the Purpose of AppArmor Profile Generation

Evelyn Beck
evelyn@snctfd.io

Department of Electrical and Information Technology
Lund University

Supervisor: Christian Gehrmann

Examiner: Thomas Johansson

December 11, 2023

Printed in Sweden
Tryckeriet i E-huset, Lund

# Abstract

This thesis investigates the possibility of leveraging eBPF to audit containerized workloads to produce data equivalent to AppArmor auditing, with the goal of later using that data to automatically generate AppArmor profiles to protect those workloads. The report presents several challenges posed by the restrictive nature of eBPF as well as how these challenges were addressed in the implementation of a prototype titled `bpfsnoop`. The prototype is then evaluated and we find that the performance overhead of this approach is slightly worse than the existing AppArmor-based auditing solution. Despite this, a potential use case is identified by considering availability; several top Linux distributions do not support the existing solution but do support `bpfsnoop`. In other words, `bpfsnoop` could provide value by enabling AppArmor-equivalent auditing in environments where it would not otherwise be possible.

# Popular Science Summary

Cloud services are a critical component of modern society. Everything from how we consume media and play games to how we pay our bills and do our work relies on cloud services, and the sheer scale of the customer base served by these services requires that cloud applications are constructed in a scalable fashion. The industry's answer to this problem is *microservice architectures*; instead of building a large monolithic web application that handles all business logic, you build many small services. For example, if you're building a web store, you might build one service for user registration and login, one that keeps track of which products are available, one that handles orders and payments, and a frontend service. These services all talk to each other, but keeping them separate makes it easy to change one component without affecting others, and perhaps more importantly, makes scaling up components that are under heavy load easier.

A key piece of technology enabling the scalability of microservices is *containerization*. Essentially, developers can package a service and all its dependencies into an *image*. Identical copies of the service can then be deployed as containers on servers across the world. From inside the container, the service can only see data included in the image and nothing more. This, however, is a trick - in fact, many containers may be running on the same server, but the operating system gives each container its own view of what files exist.

While containers have allowed tech to scale to levels previously not possible, those levels of scaling bring new security challenges. When companies like Netflix and Tinder may have millions of containers deployed at any given time, you may ask yourself, how do they keep them secure? The answer to this, clearly, is automation.

Enter Bifrost Security: They offer companies the ability to monitor everything their microservices do using AppArmor, a Linux security technology, and analyze their behavior. Based on this, they can generate an AppArmor *security profile* which only allows behavior which has been previously observed. Because hackers often try to trick services into doing something they're not supposed to, this can prevent breaches of security before they even happen and alert operators to attempts in progress.

However, the entire Bifrost value proposition relies on their customers' servers having AppArmor available, and unfortunately, this is not always the case. The work presented in this thesis attempts to begin to address this by developing

`bpfsnoop`, a new tool for auditing microservices. The goal is to be able to produce audit data equivalent to what AppArmor produces, but instead using eBPF. eBPF is an emerging technology that allows us to write small programs and tell the operating system to run them whenever a specific event occurs; for example, when a process opens a file or sends a message over the internet. By attaching eBPF programs to the same events observed by AppArmor, we can export the same data, achieving our goal.

In the evaluation phase of our work, we compare several aspects of `bpfsnoop` against the existing solution. We find that while our eBPF-based approach yields worse performance, the goal of achieving better availability is very much accomplished; the features required to support `bpfsnoop` are available on all popular Linux distributions where AppArmor is not. This indicates a possible niche for `bpfsnoop` as a fallback solution where AppArmor is not available.

# Table of Contents

# List of Figures

# List of Tables

# Chapter 1
# Introduction

## 1.1 Background

In the modern era of cloud computing, applications and services are increasingly deployed using microservice-based architectures. In such an environment, one company may have anywhere from tens of thousands [1] to millions [2] of containers deployed across dozens of locations worldwide. While this approach to constructing systems has been massively successful in allowing systems to scale effectively, it also brings unique security challenges. Systems of this scale are of course impossible to monitor manually.

Bifrost attempts to address this problem by offering a service that aims to reduce the attack surface of containerized applications. Based on work by Zhu and Gehrmann [3], their service audits all security-sensitive actions taken by processes running inside a container during a known-good period (for example, during testing) and allows for the generation of a custom AppArmor [4] MAC profile. Once the generated profile is applied to the container, it will only allow the container to take actions that were previously observed during the audit period. For example, files that are only ever read from during the audit period will have their access restricted so that they cannot be written to. In fact, AppArmor is also used during the auditing portion of the process; when auditing a container, it is run with a special AppArmor profile that will cause all accesses to be logged.

The Bifrost service architecture will be discussed further in section 2.3, but as can be understood from this, AppArmor is a critical component for all of this to work, and it is used for good reason - it is a very powerful and flexible MAC enforcement engine, but it is not without its drawbacks. Its performance impact has been found by Zhang [5] to be moderate, though the exact overhead of Bifrost's use case remains to be seen. Another perhaps greater concern is that of availability; while certain major Linux distributions such as Debian, Ubuntu, and OpenSUSE ship with AppArmor enabled [4], others such as Amazon Linux, RHEL, and CentOS do not.

With this in mind, one growing technology that has been used to address similar concerns of observability and runtime security is eBPF[1] [6]. eBPF will be

[1] eBPF formerly stood for "extended Berkeley packet filter", but its capabilities have grown to a point where this no longer accurately describes the technology. As such, eBPF is now considered a standalone term.

explained in far greater detail in section 2.4, but its primary function is to allow userspace processes to attach small programs to almost any point in the kernel. These programs can perform functions such as monitoring and manipulating network traffic, monitoring and filtering system calls, and other tasks that require deep access into the host system.

## 1.2 Goals and Limitations

The purpose of this thesis is to investigate how eBPF-based tooling could be used to collect the same audit data currently provided by AppArmor. While eBPF-based runtime enforcement is theoretically possible (and indeed, a subject of active research and development [7]–[9]), the work described in this thesis will not attempt such an endeavor. It may, however, serve as a starting point for future work in this area. Some of the questions we will attempt to answer are as follows:

- Can eBPF be used to produce similar or equivalent security audit data to the existing AppArmor auditing system? What existing tools can be used to aid this process? What solutions exist in the literature to the problem of producing audit data using eBPF?
- How does the performance of eBPF-based container auditing compare to that of the AppArmor-based system, and how does our solution compare to other similar eBPF-based solutions in the literature?
- How does the choice of eBPF over AppArmor impact security and availability?

### 1.2.1 Scope

The scope of our work is limited to applying eBPF to the auditing process. While generating AppArmor profiles is the primary motivating factor for producing AppArmor-equivalent audit data, the actual details of how this is done are out of scope. It is possible that eBPF could be used to produce richer audit data that may improve the quality of generated profiles, but the goal of this work will merely be to match the capabilities of existing work.

## 1.3 Methodology

In order to answer the questions stated above, we will develop a prototype and evaluate it. The evaluation includes both performance benchmark experiments and analysis of the resulting prototype from a security and availability perspective.

Chapter 2

# Technical Background and Related Work

## 2.1 The Linux Security Model

### 2.1.1 Discretionary Access Control

The primary method by which access to resources is controlled in Linux is its file permission system based on the DAC (Discretionary Access Control) model. In this model, every file is owned by a specific user and group. The owner user is allowed to specify how a file can be accessed by themselves, members of the file's group, and other users. The most common permissions are **r**ead, **w**rite, and e**x**ecute, but there are other properties that can be set as well, such as the `setuid` and `setgid` bits, which allow an executable file to run with the permissions of its owner user or group, respectively.

### 2.1.2 Linux Security Modules

The Linux Security Module (LSM) framework [10] was developed to allow the Linux security model to be extended. It does this by providing a rich set of hooks that are invoked before security-sensitive operations are performed. By attaching to these hooks, LSMs can be used to implement a large range of different access control models and other security features. Some examples of security frameworks that use the LSM interface are AppArmor (more on this in section 2.2), SELinux [11], and Landlock [12]. When invoked, an LSM's hook handler has access to a wealth of information about the access request, and can choose to deny or allow the access. If multiple LSMs have attached to the same hook, they are invoked sequentially and access can be denied by any individual handler.

It is worth noting that unlike normal kernel modules, LSMs cannot be dynamically loaded and unloaded while the kernel is running. Instead, they are built into the kernel during compilation. It is also possible to control which LSMs are loaded during system startup by using the `security` kernel boot parameter. A list of currently active LSMs is available at `/sys/kernel/security/lsm`.

### 2.1.3 Capabilities

Traditionally, there have been two levels of privilege with which a process can run in Linux: Either as a normal user, or as the superuser. In order to provide

more fine-grained control over what a process can do, the concept of capabilities was introduced. Capabilities are assigned to each thread and control their ability to perform a variety of different system operations. There is a long list [13] of different capabilities, but some examples are:

- `CAP_NET_RAW`, which permits the use of raw and packet sockets as well as binding to any address,
- `CAP_CHOWN`, which permits arbitrary changes to file ownership,
- `CAP_SYS_RAWIO`, which permits a range of sensitive I/O operations,
- `CAP_SYS_ADMIN`, which has been used as a catch-all permission and thus grants wide-ranging powers.

### 2.1.4 Namespaces

Namespaces [14] are an isolation feature of the Linux kernel that allows a process or group of processes to be assigned their own private, restricted view of some global resource. For example, by placing a process within its own PID (process ID) namespace, it cannot see other processes present on the system, and it even has a virtualized view of its own PID. If that process spawns a second process, both processes will be within the same namespace and thus be able to see each other, but not other processes on the system. There are several other types of namespaces that all work in a similar way, but arguably the most important are:

- Cgroup namespaces virtualize the view of cgroups (see below for an explanation of what cgroups are) present on the system.
- Network namespaces virtualize networking, providing the namespace with its own networking stack, routing tables, etc.
- IPC (inter-process communication) namespaces isolate various methods of inter-process communication.
- Mount namespaces virtualize the view of available mounts, which isolates the file system.

### 2.1.5 Control Groups

Control groups, usually referred to as cgroups [15], are a kernel feature by which processes can be organized into a hierarchy. At each level of the hierarchy, limits can be set on the contained processes' combined resource usage. Cgroups are created and managed by interacting with the cgroup filesystem.

### 2.1.6 How Containers Contain

A very common application of these isolation primitives is to create containers. Similarly to virtual machines, the goal of containers is to put an application within a "box" inside which they cannot see the rest of the system. Virtual machines accomplish this by running a full-fledged operating system separate from the host

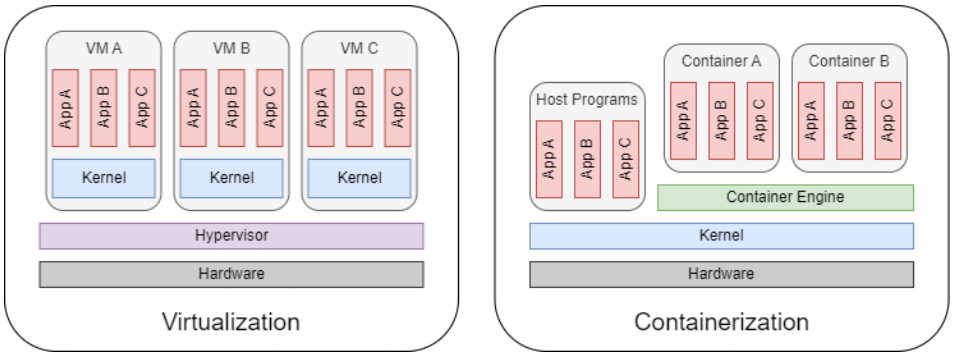


**Figure 2.1:** Virtualization and containerization visualization.

machine's OS, but this is wasteful as it introduces massive overhead. Containers, then, are a more lightweight alternative that instead uses namespaces to give processes their own view of the system. Crucially, this means they are still running within the same kernel as uncontained processes and processes within other containers, but these are simply hidden from view. A diagram demonstrating the architectural difference between virtual machines and containers can be found in figure 2.1.

## 2.2 AppArmor

AppArmor [4] is an LSM that introduces Mandatory Access Control (MAC) into the Linux security model. This differs from the DAC model in that access control restrictions are placed on programs, as opposed to users. In other words, it allows system administrators to improve the security of individual applications by placing restrictions on what resources the application can access and how it can access them. AppArmor can control access to a large variety of different resources, including [16]:

- Network access, with support for differentiating between operations (`bind`, `read`, `accept`, etc), protocols, and socket types
- Filesystem objects, with rich functionality for globbing and specifying rules for various operations and access modes
- The ability to perform IPC and to send and receive signals
- Capabilities, which can be granted to or revoked from confined processes

These restrictions are applied to programs by creating *profiles*. Profiles contain a set of rules that grant or restrict access and can be applied to processes. Typically, this is done automatically by specifying in the profile which executable path(s) it should apply to, but it is also possible to create named profiles that can be applied in other ways, for example to an entire container by the container runtime.

```
1 #include <tunables/global>
2 /bin/ping flags=(complain) {
```

```
3   #include <abstractions/base>
4   #include <abstractions/consoles>
5   #include <abstractions/nameservice>
6
7   capability net_raw,
8   capability setuid,
9   network inet raw,
10
11  /bin/ping mixr,
12  /etc/modules.conf r,
13 }
```

**Listing 2.1:** A simple AppArmor profile

Listing 2.1 contains a very simple profile provided by AppArmor [17] that restricts the `ping` program. While some of the complexity has been hidden in the included files, this profile demonstrates how easily small programs can be confined with AppArmor.

### 2.2.1 How Violations are Handled

There are several modes by which violations of the AppArmor policy can be handled. In complain mode, which the above `ping` profile has been configured to use via the `flags=(complain)` option in the profile header, violations of the AppArmor policy will merely be reported in the kernel audit log. Enforce mode, however, will additionally cause illegal operations to fail, and kill mode will cause the offending process to be killed. Listing 2.2 below shows an example [17] of audit log output produced by AppArmor policy violations.

```
1 [1521056.552037] audit: type=1400 audit(1571868402.378:24425):
      apparmor="DENIED" operation="open" profile="/usr/sbin/cups
    -browsed" name="/var/lib/libvirt/dnsmasq/" pid=1128 comm="
    cups-browsed" requested_mask="r" denied_mask="r" fsuid=0
    ouid=0
2 [1482106.651527] audit: type=1400 audit(1571829452.330:24323):
      apparmor="AUDIT" operation="sendmsg" profile="snap.lxd.lxc
    " pid=24115 comm="lxc" laddr=10.7.0.69 lport=48796 faddr
    =10.7.0.231 fport=445 family="inet" sock_type="stream"
    protocol=6 requested_mask="send" denied_mask="send"
```

**Listing 2.2:** Sample AppArmor audit log lines

As can be seen, these contain a wealth of information, such as:

- The PID, command, and active profile of the offending process.
- The attempted operation as well as operation-specific details such as what resource the process attempted to access and which access modes were requested and denied.
- The action taken by AppArmor; in the first case the operation was denied because the profile was in enforce mode, while the second operation was merely logged.

## 2.3 The Bifrost AppArmor Profile Generator

As previously described in chapter 1, Bifrost offers a service that audits containers and allows for the generation of AppArmor profiles that only permit the operations observed during the audit period. With the concepts described in this chapter thus far in mind, we can now elaborate on some of the technical details of this service.

Figure 2.2 contains a diagram depicting a typical usage scenario of the service. In this scenario, the customer has two Kubernetes clusters (staging and production) with two nodes each. In addition to the customer's pods, each node runs an instance of the `qtool` service, which is responsible for communicating with the backend and performing AppArmor-related operations. The workflow proceeds roughly as follows:

1. The customer configures their environments in the Bifrost web portal and creates a service.
2. After selecting that the service should be audited in the staging environment, the backend will distribute an audit profile with a unique name to each node in the staging environment via `qtool`, which will load the profiles into the kernel.
3. The customer deploys their service in the staging environment with the audit profile enabled and runs their test suite against the service.
4. As the service runs, the audit profile causes AppArmor to log all accesses. `qtool` sends the log lines to the backend, which uses the unique audit profile name present in each line to determine which service produced it. This is necessary because several service deployments may be running on the same node.
5. When the service has been fully tested, the customer can request that a security profile be generated in either complain or enforce mode. At this time, the profile generator will pull all audit data produced by the service from the audit log storage, use it to generate a profile, and store the profile.
6. The customer can now distribute the generated profile to their production environment nodes and then deploy the service with it enabled. Violations of the generated security profile will be reported by AppArmor (and prevented if an enforce-mode profile was used) and sent to the backend by `qtool`.

## 2.4 eBPF: Safely Extending the Kernel

As was previously introduced, eBPF [6] is a Linux kernel technology that allows userspace applications to extend the kernel's capabilities by attaching small programs to specific events. Among other things, eBPF has been used to create observability tools [7], load balancers [18], and container security frameworks [8], [9].

What makes eBPF so powerful is that eBPF programs can be hooked into almost anywhere in the kernel, where they can monitor and control core parts of

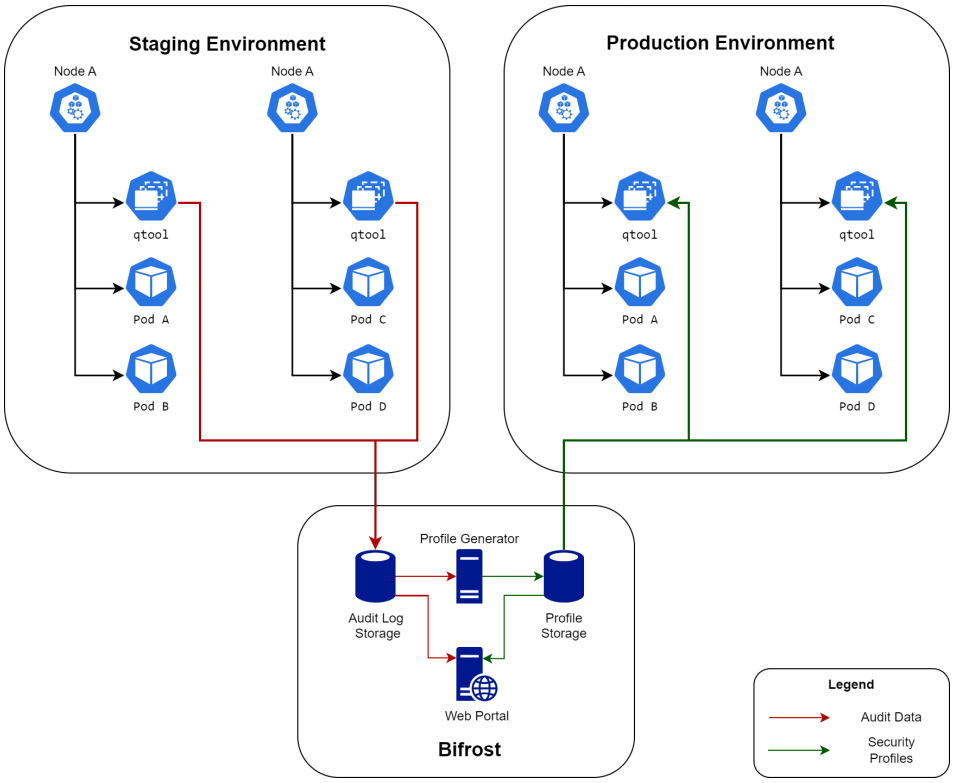


**Figure 2.2:** Bifrost Service Architecture

the system. Typically, eBPF programs are written in C and compiled to bytecode that can later be JIT-compiled. This enables eBPF programs to be portable between different kernel versions. There are many types of eBPF programs with different capabilities and purposes [19], but some of the important ones are:

- XDP (eXpress Data Path) and tc (traffic control) eBPF programs can read, modify, and redirect network traffic and are used to implement networking functionality such as load balancers, firewalls, and monitoring tools.
- Kprobes, kretprobes, uprobes, and uretprobes allow eBPF programs to attach to arbitrary functions in the kernel as well as in userspace code such as executables and libraries. They can read kernel and userspace memory, and retprobes can modify the return value of the function they attach to.
- LSM eBPF programs can attach to LSM hooks and read kernel and userspace memory, including details of the attempted access. Similarly to how LSMs behave, they have the ability to deny access attempts.

Typically, eBPF programs communicate with userspace through the use of ring buffers [20] and maps [21]. eBPF offers functionality by which these collections can be created and maintained within the kernel, where they can be accessed and modified by both userspace and eBPF programs. A very simple eBPF program can be found in appendix B.1. After attaching this program to the `execve` kprobe, it will be invoked every time a program uses the `execve` system call. Each time it is invoked, the program will write an event consisting of the calling process' PID and command to the ring buffer, where it can be consumed by a userspace program.

### 2.4.1 The Verifier and its Limitations

At this point, the security-minded reader has certainly realized that offering such wide-ranging powers to userspace programs comes with a great deal of risk. Indeed, eBPF programs run synchronously, i.e. blocking further execution of any hooked functionality until they complete, and to make matters worse, they run with kernel-level privileges. In other words, eBPF programs are one malicious instruction away from wreaking havoc on the system, either by corrupting internal structures or simply by refusing to return control back to the kernel.

For this reason, eBPF programs are verified before they are loaded. The verification process places some quite strict requirements on the program:

- The program must always halt. Determining whether a program halts is an undecidable problem [22], so this is overcome by placing a fixed limit on the number of instructions executed (typically 1 million [23]). During verification, all possible code paths are simulated and the program is rejected if any path exceeds this limit. Among other things, this means all loops must be bounded by a fixed number of iterations.
- All memory accesses must be verifiably correct. This means ensuring that segmentation faults, buffer overflows, and other illegal behavior are impossible. In practice, this means pointer arithmetic is highly restricted and we

cannot read arbitrary memory locations without knowing what's going to be there.

- There are also restrictions on things like stack size (max 512 bytes), total program size (4096 instructions), and which helper functions are available depending on the program type.

As one may imagine, some of these restrictions quite severely limit what is possible in eBPF. For example, looping until the end of an arbitrarily string is simply not possible, due to the termination requirement. Some of these limitations are overcome by the use of helper functions [24] that are available to call from eBPF programs. One such helper function, for example, is `bpf_probe_read_kernel_str()`, which safely copies a string from kernel memory to a new location.

### 2.4.2 `bpftrace`: A Better Way?

Another potential drawback of eBPF is that it is quite difficult to use. Writing eBPF programs often requires intimate knowledge of kernel structures, problems can be difficult to troubleshoot, and documentation is often missing, outdated, or otherwise lacking. In an attempt to address this and make eBPF more accessible, `bpftrace` [25] was created. `bpftrace` offers a script-like language that makes it far easier to write relatively powerful and general-purpose eBPF programs. An example of a script filling roughly the same purpose as the example program in appendix B.1 follows:

```
bpftrace -e 'kprobe:sys_enter_execve { print("%s %s", comm, pid); }'
```

The possibility of using `bpftrace` in our prototype was strongly considered but ultimately rejected as there were several key problems that could not be solved, including reading a path in an LSM hook. Section 3.1.2 elaborates on why this is a difficult problem to solve in eBPF in general. While this problem was eventually solved in our final prototype, we could not produce a minimal proof of concept using `bpftrace`'s more limited scripting language that could read full paths and that satisfied the eBPF verifier.

## 2.5 Closely Related Works

eBPF is a rapidly evolving technology, and in certain areas formal research appears to be falling behind the development of new eBPF-based tools. While there is a wealth of research surrounding the use of eBPF as a networking tool for applications such as routing and packet filtering [26]–[30], less research exists surrounding the use of eBPF to achieve security-oriented goals such as audit data collection and confinement. We now present some of the work that does exist surrounding the use of eBPF for security purposes.

### 2.5.1 Enhancing Security in Docker Web Servers Using AppArmor and BPFtrace

During literature review, the most relevant work we found was a thesis produced by Mukherjee [31] that describes their attempt at solving a problem very similar to the one we seek to solve. Their proposed solution involves combining the output from several pre-existing `bpftrace` scripts to create security profiles. The quality of the resulting profiles is compared to that of Docker-sec [32] by testing whether each profile defends against various exploits.

However, there are several critical flaws within their work. One relatively minor problem relates to how they capture file access events; they use `opensnoop.bt` [33], which captures filenames from the `open` and `openat` system calls. These system calls take a path in the form of a string, and this string is what they capture and ultimately use in their AppArmor profile. However, in the event that the path is relative or that it points to a symbolic link, it will not reflect the path of the actual file accessed [34], which is what AppArmor cares about.

To make matters worse, the script they present in section 3.5.1 appears to only actually use the output from the file opens when generating a profile, and even then they simply assign all opened files read-only permissions (see the line `echo "  \" ${file}\" r, " >> "${output_file}"`). Despite all this, they find that their profile not only works, but that it prevents all 10 exploits they tested, whereas Docker-sec only prevents 4.

This may seem strange, but the explanation becomes apparent by studying the generated profile in section 4.3.2. Large parts of the profile appear to have been written either by hand or by a generator far more advanced than what is described in the report itself. Indeed, the "future work" section appears to confirm that manual effort was involved in creating the profile. The profile contains several rules that are framed as "general security best practices" that restrict access to things like shell executables and other system files. These are likely what stopped the majority of the exploits tested, and there is no reason these could not also be applied to the Docker-sec profile. In other words, the eBPF part of their work does not actually contribute to the improvements observed.

There are certainly lessons that can be learned from this, and we will see in chapter 3 how we avoid some of the pitfalls described above.

### 2.5.2 *e*Audit: A Fast, Scalable and Deployable Audit Data Collection System

At some point during the development of our work, Sekar, Kimm, and Aich published a report [35] detailing *e*Audit, an eBPF-based audit data collection system. Their report begins with a motivating experimental study of existing security audit systems in which they investigate two commonly used systems, `auditd` and sysdig, as well as two more recent prototypes, ProvBPF and CamFlow. The study shows that these systems all suffer from a significant number of dropped events during peak loads as well as overheads ranging from 100% to well over 600%.

Having demonstrated the drawbacks of existing systems, they go on to present the design of their own prototype, *e*Audit. The prototype consists of three primary

components: *e*Cap, which represents the eBPF programs responsible for capturing audit events and writing them to userspace, *e*Log, which reads the events from *e*Cap's eBPF buffers and exports them, either to persistent storage or to *e*Parse, which parses events in binary format and prints them in a human-readable format.

*e*Audit employs a number of interesting techniques to optimize eBPF data collection. Perhaps the most architecturally interesting among these is the use of a double-layered buffer system. Essentially, eBPF buffers can be either shared between all CPUs or instanced per-CPU, such that each CPU has its own buffer. While per-CPU buffers offer better performance as they do not require synchronization to be accessed safely, they are more prone to data loss compared to using a single buffer that is $N$ times as large ($N$ being the number of CPUs). To achieve the best of both worlds, *e*Audit initially writes audit events to per-CPU message caches. Once a cache fills up, its contents are written to a shared ring buffer from which the events can be read by *e*Log.

Additionally, *e*Audit attempts to improve performance and minimize data loss by encoding events in a very compact format. This is achieved through methods such as only including the least significant 3 bytes of the timestamp when possible, omitting thread ID information for single-threaded applications, but perhaps most important is the use of variable-length encoding for arguments and return values. This is useful in cases where a value (e.g. a file descriptor) is represented by 8 bytes in the kernel but the actual value is small enough to fit in fewer bytes.

The result of employing these optimization techniques is that *e*Audit incurs a much smaller overhead of 4-5% in a benchmark where competing systems have an overhead of 36-65%. It also manages to completely avoid data loss, which all other systems suffered from in some cases.

### 2.5.3 ViperProbe: Rethinking Microservice Observability with eBPF

Levin and Benson [36] have written an article detailing ViperProbe, a microservice observability tool they developed using eBPF, written in Python using IOVisor's BCC frontend. As opposed to our work which aims to export security audit data (necessitating that all events are handled), ViperProbe focuses on performance metrics. Because of this, they put significant emphasis on developing a framework for analyzing the performance profile of a specific application, identifying which metrics are of importance, and excluding the ones that are not in order to reduce performance overhead.

In their work, they perform several benchmarks aimed at measuring latency and CPU overhead. Their performance analysis shows that ViperProbe has an overhead of approximately 10-20% and additionally reveals one important takeaway: Sampling rate matters far less than which metrics are measured, indicating that optimizing metrics with a large performance impact may be the best way to reduce overhead.

# Chapter 3

# bpfsnoop: Technical Design

Before a full-scale prototype could be developed, there were a number of technical challenges to solve. This was done by producing several small-scale proof-of-concept programs, the details of which will not be discussed at length. Instead, this chapter aims to describe the challenges encountered and how they were solved, as well as the architecture of the final prototype implementation.

## 3.1 Technical Challenges

### 3.1.1 Choosing the Right Hookpoints

One of the first questions that needs to be answered when writing eBPF programs is where they are going to attach to the kernel. Perhaps the most obvious solution to this is to simply attach to various system calls, as that is how userspace programs interact with the kernel. However, as was explained in section 2.5.1, this may cause problems. Instead, our solution attempts to mimic AppArmor by primarily using LSM hooks, since it stands to reason that by attaching to the same hooks we should be able to extract the same information.

### 3.1.2 Reading Paths in eBPF

When developing proof-of-concept programs that solve small-scale versions of what we want to accomplish, it was quickly found that reading paths from eBPF programs is not as trivial as one might think. Specifically, there is a group of hooks that take a `path` structure as an argument (e.g. `path_chmod`, `path_mkdir`, etc.). The `path` structure consists of a mount reference and a pointer to a `dentry` (directory entry) structure. What makes things difficult is that the full path is not stored anywhere; instead, each `dentry` describes one level of the directory tree and holds a pointer to its parent directory. To read a full path, we must traverse the `parent` pointers until we get to the file system root, copying the directory name at each step.

Quite annoyingly, there is an eBPF helper function, `bpf_d_path`, that solves this exact problem, but it is unavailable in almost all eBPF LSM hooks. A fairly trivial patch [37] which adds most LSM path-related hooks to the `bpf_path_d` allowlist has been developed, but it has not been merged into the kernel at this time. Instead, a custom function (contained in appendix B.2) was built that

traverses the path tree and builds the path as a string in a buffer. While this solution works, it does require a fixed limit to be placed on how deep of a path we can traverse, as the verifier forbids unbounded loops. It is also likely that performance could be improved if `bpf_d_path` is made available to LSM hooks.

### 3.1.3 Filtering Events by Container

In the existing AppArmor-based auditing system, audit log entries contain the name of the violated AppArmor profile, which the Bifrost backend uses to determine which service produced a given log entry. When using eBPF to audit system calls, however, we do not have this luxury. Additionally, if our eBPF programs simply export all events produced by the kernel, a lot of time will be wasted since we only care about auditing specific containers.

The solution to this was found by realizing that eBPF allows us to get the cgroup ID of the current process, and that in most circumstances, all processes in the same container will have the same cgroup ID. Thus, we can simply keep track of all running containers' cgroup IDs in an eBPF map, and when an eBPF program is invoked it can check if the current process' cgroup ID is contained within the map. If it is, it proceeds to export the security event, but if not, it can simply exit early. By then including the cgroup ID in the event data, our userspace component can later figure out which service produced a specific event.

## 3.2 Proposed Architecture

With the previously described technical challenges and their solutions in mind, we can now proceed to describe the architecture of the developed prototype. A diagram providing an overview of the architecture can be found in figure 3.1.

As is typically the case with eBPF applications, the solution consists of a userspace component and several eBPF programs - one for each attached hook. Events captured in eBPF are written to a single ring buffer shared between all eBPF programs, where they can be read and parsed by the `bpfsnoop` process. Once parsed, the event is transformed into a JSON representation and sent to a remote endpoint.

The choice of using a single ring buffer was made due to the large number of different events; having a dedicated buffer for each event type would use significantly more memory. However, when reading from the buffer, the userspace application only sees a sequence of bytes. Consequently, it needs some way to determine what type of event it has read. This problem was solved by placing a byte at the start of each event that identifies the event type.

### 3.2.1 Integrating With Existing Infrastructure

While our work is primarily focused on the intricacies of collecting relevant audit data using eBPF, it is also worth considering how `bpfsnoop` can be integrated with the existing Bifrost infrastructure. The primary obstacle to integration is the fact that `bpfsnoop` outputs events formatted as JSON, whereas the Bifrost auditing

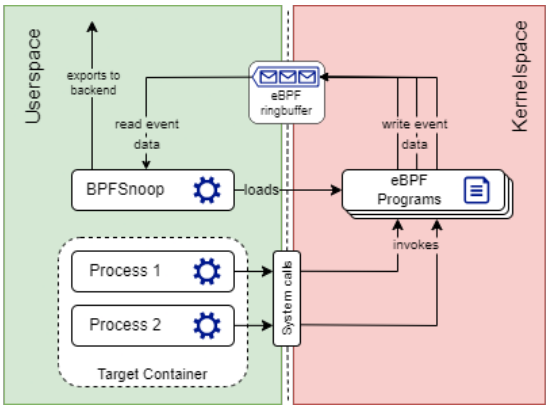


**Figure 3.1:** Overview of how events are collected

tool sends raw AppArmor audit lines to the backend. This difference in behavior can be handled in several different ways, for example:

- Adding functionality to `bpfsnoop` to allow it to output events in the AppArmor format. While this may appear to be the most straightforward solution, there is to our knowledge no formal specification of AppArmor's audit log format, which may result in incorrect handling of corner cases as well as future problems if AppArmor changes its output format. This also adds complexity to `bpfsnoop` which may make future development more difficult.
- Creating an endpoint within the backend designed to accept audit data from `bpfsnoop` and convert it to the representation used in the backend. While this avoids adding complexity to `bpfsnoop` and fully decouples `bpfsnoop` from AppArmor, it does increase architectural complexity and come with the added cost of maintaining an additional service within the backend.

Regardless of which approach is chosen, it may be beneficial to formalize some specification of what data an audit event is expected to contain. This will help ensure that both methods of auditing are handled correctly.

# Chapter 4
# Evaluation of the Design

## 4.1 Performance Evaluation

One critical question this thesis attempts to answer is how the choice of using eBPF to collect audit data impacts performance when compared to the existing AppArmor-based solution. To answer this, two benchmarks were devised and tested in a Kubernetes environment for three different cases; without any auditing, with the existing Bifrost auditing, and with `bpfsnoop` auditing.

### 4.1.1 API Benchmark

The first benchmark attempts to mimic a real-world use case in the form of a microservice. The setup (heavily based on [38]) in the baseline case consisted of the following components:

- A simple blog API written in Python [39] and a MySQL database, running in separate pods.
- A k6 load generator sending requests to the blog API.
- An InfluxDB database receiving load test results from the load generator and a Grafana instance for data visualization.

The Kubernetes cluster used in the experiment consists of two nodes, $N_1$ and $N_2$, running on Ubuntu 20.04 with kernel version `5.15.0-1048-aws x86_64`. To isolate the effects of each auditing mechanism, the baseline experiment was run without either the AppArmor or the eBPF LSMs enabled, while the Bifrost experiment was run with AppArmor enabled but eBPF disabled, and the `bpfsnoop` experiment was run with eBPF enabled and AppArmor disabled. Additionally, in order to prevent metrics collection from affecting the experiment, the REST API and the MySQL database run on $N_1$ while the load generator, InfluxDB, and Grafana pods run on $N_2$.

For reference, the k6 script used to perform load testing can be found in appendix B.3. The script uses three virtual users that rapidly send requests to the audited service. Worth noting is that k6 opens an HTTP connection for each virtual user at the start of the test and reuses it for the duration of the run.

### 4.1.2 Kernel Build Benchmark

In addition to the API benchmark which almost exclusively performs network operations, a second benchmark was constructed. This benchmark consists of simply measuring the time it takes to build the Linux 6.5 kernel in a container in the three cases. This was done by constructing a Docker image containing the Linux source code which runs `make` upon container start. This image was scheduled onto the target node as a Kubernetes job, and the build time was calculated as the time between when the job started and when it finished. Similarly to in the API benchmark, the AppArmor LSM was only enabled in the Bifrost case and the eBPF LSM was only enabled in the `bpfsnoop` case.

The purpose of this benchmark was to get a more varied set of operations, particularly file operations. Because building a kernel is a large task that involves traversing file trees, reading source code, writing temporary build files, etc., it is well suited for this purpose, which is why it is quite commonly used as a benchmark [40], [41].

### 4.1.3 Results and Discussion

The API benchmark was performed three times for each configuration. For each run, the mean, 90th percentile, and 95th percentile values of HTTP request duration was recorded. The kernel build benchmark was performed five times for each configuration, and the build time for each run was recorded. The averages of these values across all runs along with their 95% confidence interval can be found in table 4.1.

Using the data in the table, we can calculate that for the API benchmark, Bifrost auditing added approximately 5.3% overhead compared to baseline while `bpfsnoop` added approximately 6.7% overhead. In the kernel build benchmark, Bifrost auditing added 11% overhead while `bpfsnoop` added 19.9% overhead, which is not insignificant. That said, several possible improvements have been identified which could reduce overhead:

- General optimization of the eBPF programs. In particular, the significantly worse performance in the kernel build benchmark seems to indicate that file operations are a problem area where efforts can be focused.
- Batching events: Currently, `bpfsnoop` sends events one by one to the remote endpoint as soon as they are read from the eBPF event buffers. By instead keeping a buffer of read events and regularly dispatching them to the backend in batches (which is what the Bifrost daemon does), the number of necessary network operations could be significantly reduced. While this was not done in the current prototype in an effort to minimize complexity, it represents a fairly salient target for optimization.
- Double-layered event buffers: The approach introduced by *e*Audit [35] (described in section 2.5.2) in which events are initially written to per-CPU ring buffers and then periodically dumped to a shared buffer may be able to improve performance when many processes are producing events by reducing ring buffer contention.

- In an attempt to match AppArmor's behavior, the current `bpfsnoop` prototype targets many different kinds of operations. Through careful consideration of how the events are processed in the backend, the eBPF programs could likely be specialized to reduce the amount of unnecessary data exported.

## 4.2 Security Considerations

Given the security-oriented nature of the auditing being performed, it is important to be mindful of possible vulnerabilities introduced by our tools. While both the Bifrost auditing agent and `bpfsnoop` are designed to run within container environments that traditionally limit access to the system as a whole, the fact that they by design attempt to monitor the activities of other containers necessarily means some of the isolation mechanisms must be disabled in order for them to function.

This breakdown of isolation comes in two forms: By granting the container additional capabilities (see section 2.1.3), and by mounting filesystems that control system functions into the container. `bpfsnoop` requires the following capabilities and mounts:

- `CAP_SYS_ADMIN`, which is necessary to allow the use of the `fanotify` API [42] which lets `bpfsnoop` take action whenever a new container is created and add its cgroup ID to an eBPF map. It also supercedes `CAP_BPF`, allowing eBPF programs to be loaded.
- `/sys/kernel/debug`, `/sys/fs/cgroup`, `/usr/sbin/runc`, which are all used in the container discovery process.
- `/sys/kernel/btf/vmlinux`, which exposes BTF (BPF Type Format) information which enables BPF programs to be portable across different kernel versions.

Of these, by far the primary cause for concern is the use of the `CAP_SYS_ADMIN` capability. As described in [13], `CAP_SYS_ADMIN` is overloaded with responsibilities and has been described as "the new root". Indeed, there are fairly trivial scripts available online [43] that allow a container with only the `CAP_SYS_ADMIN` capability to fully break isolation and spawn a shell as root within the host environment. In other words, granting `bpfsnoop` the `CAP_SYS_ADMIN` capability may allow an attacker that compromises `bpfsnoop` to escalate their privilege.

| | API benchmark | | | Kernel build |
|---|---|---|---|---|
| Auditing | $\overline{t_{mean}}$ (ms) | $\overline{t_{90}}$ (ms) | $\overline{t_{95}}$ (ms) | $\bar{t}$ (s) |
| None | $278.46 \pm 0.36$ | $568.25 \pm 1.39$ | $596.92 \pm 2.08$ | $2712.2 \pm 4.0$ |
| Bifrost | $293.22 \pm 0.99$ | $634.63 \pm 0.92$ | $671.16 \pm 1.05$ | $3036.8 \pm 3.5$ |
| `bpfsnoop` | $297.15 \pm 0.29$ | $623.47 \pm 0.60$ | $656.20 \pm 0.78$ | $3252.8 \pm 22.2$ |

**Table 4.1:** Results from the two benchmarks.

However, it is important to underscore that this requires the attacker to be able to execute arbitrary code within the context of the `bpfsnoop` container. Exploits that grant this sort of access (RCE - Remote Code Execution) most commonly result from improper handling of attacker-controlled data, and the prototype in its current state does not actually process any data from external sources. While a more feature-complete implementation may require some degree of two-way communication with the Bifrost backend, the risks of this can likely be mitigated, for example by authenticating the backend and disallowing communication with other parties. As a result, escalation of privilege is mostly a theoretical concern that should not be feasible in practice. Nevertheless, reducing the capabilities necessary to run `bpfsnoop` in order to eliminate this risk entirely is desirable, but finding some way to detect newly started containers is left for future work.

The Bifrost auditing agent also requires several capabilities. While it avoids the use of `CAP_SYS_ADMIN`, it does require `CAP_MAC_ADMIN` which allows configuration of AppArmor, `CAP_NET_ADMIN` which allows configuration of network settings, and `CAP_NET_BROADCAST` which allows the use of certain Netlink features. Of these, `CAP_NET_ADMIN` is the most dangerous as it may allow an attacker to sniff the traffic of other containers. However, these permissions are as a whole less dangerous than `CAP_SYS_ADMIN`.

## 4.3 Availability

One of the primary motivating factors behind this work lies in making the Bifrost profile generation as widely available as possible. While AppArmor is active by default in several popular Linux distributions, it is far from ubiquitous. As a consequence, providing a solution to enable auditing of containerized applications in environments where AppArmor is not available will expand the possible reach of Bifrost services.

The primary factor determining whether `bpfsnoop` can run on a particular Linux distribution is whether the BPF LSM is loaded. Because `bpfsnoop` almost exclusively uses LSM hooks, the BPF LSM is absolutely necessary. It is technically possible to enable the BPF LSM on most distributions that have it disabled by default by simply changing a boot parameter, but this is arguably unreasonable to expect from clients to do.

Another factor is the kernel version. Based on available documentation [44], most features necessary to run `bpfsnoop` in its current state have been present since Linux 5.8, which introduced the ring buffers used to export events. There is one exception however: For highly technical reasons, a specific patch introduced in 6.1 [45] is necessary to enable the use of the `path_chown` hook [46].

To provide some clarity on the current state of the Linux distribution ecosystem, table 4.2 shows the kernel version and whether the BPF and AppArmor LSMs are loaded by default for several of the top server distributions. Other than Azure Linux (for which the most recent version was used), the distribution versions were chosen by simply using the default option when deploying virtual machines in AWS (Amazon Web Services), as this is likely what most companies would use.

As the table shows, the BPF LSM is available by default on all distributions

| Distribution | Version | Kernel Ver. | BPF LSM? | AppArmor? |
|---|---|---|---|---|
| Amazon Linux | 2023 | 6.1 | Yes | No |
| Azure Linux | 2.0 | 5.15 | No | Yes |
| CentOS Stream | 9 | 5.14 | Yes | No |
| Debian | 12 | 6.1 | Yes | Yes |
| RHEL | 9.2.0 | 5.14 | Yes | No |
| SUSE Linux | 15 SP5 | 5.14 | No | Yes |
| Ubuntu Server | 22.04 | 6.2 | No | Yes |

**Table 4.2:** Common distributions, their kernel versions, and whether the BPF LSM and AppArmor are enabled by default

where AppArmor is not. While several distributions are still on pre-6.1 kernels, this is not likely to remain the case for long. Similarly, as the adoption of eBPF-based tools is still growing, distributions that currently do not enable the BPF LSM by default are likely to do so in the future.

# Chapter 5

# Conclusion

As was described in chapter 1, the goal of this work was to produce and evaluate an eBPF-based framework for auditing containerized workloads in a manner that mimics Bifrost's existing AppArmor auditing. Achieving this goal required working around the limitations imposed by the eBPF verifier described in section 2.4 in order to solve several technical challenges. Chapter 3 describes some of these technical challenges and how they were solved, and more importantly, the approach that was ultimately chosen. In short, by attaching to the kernel in the same places that AppArmor would (i.e. LSM hooks), we ensure that `bpfsnoop` has access to the same parameters as AppArmor and avoid problems that come with attaching directly to system calls.

The evaluation part of our work described in chapter 4 yielded mixed results. Two different kinds of performance benchmarks were performed, primarily targeting network and filesystem operations respectively, and it was found that `bpfsnoop` auditing yielded greater overhead than the existing Bifrost auditing in both cases. Furthermore, both methods of auditing yielded greater overhead in the file-based kernel build benchmark than in the network-based microservice benchmark, highlighting an area where improvements could be made.

The security analysis done in the same chapter showed that an attacker that gains access to the `bpfsnoop` container could potentially escalate their privilege and gain access to the host system and other containers running therein. However, as was discussed, this is only possible if the attacker can perform arbitrary code execution within the `bpfsnoop` container, and because `bpfsnoop` does not handle any data from external sources, we find this to be an unlikely avenue of attack.

While the evaluation of performance and security showed little reason to use `bpfsnoop` over the existing AppArmor-based auditing, the availability study performed in section 4.3 paints a different picture. In this section, we investigated many common Linux distributions used in cloud servers and found that several did not support AppArmor but did provide the BPF LSM by default. In fact, all distributions that were investigated supported either AppArmor or eBPF. This points to a possible niche for eBPF-based auditing as a fallback solution in cloud environments where AppArmor is not available.

## 5.1 Future Work

While our work presents a simple prototype solving the most important problems necessary to produce audit data using eBPF, the work is far from done. The `bpfsnoop` prototype has not been intentionally optimized, which could go a long way to addressing some of the performance concerns described in chapter 4. To inform such optimization work, more narrow benchmarks focusing on specific eBPF hooks could be performed with the goal of finding specific problem areas. In addition to manual optimization, it is likely that future work within the kernel could enable more efficient ways of accomplishing certain tasks, such as by allowing the use of the `bpf_d_path` helper function within LSM hooks.

In addition to this, some amount of work is likely necessary to integrate `bpfsnoop` with the existing Bifrost infrastructure. For example, the prototype currently does not match the output format produced by AppArmor auditing. This could be solved either by simply adjusting `bpfsnoop`'s output format or by extending the backend to allow for alternate audit line formats. Once this is done, it may be prudent to perform more formal verification to show that the two are completely equivalent.

# Bibliography

[1] C. O'Brien, C. Thomas, and J. Lee, "Tinder's move to Kubernetes," *Tinder Tech Blog*, Apr. 2019. [Online]. Available: https://medium.com/tinder/tinders-move-to-kubernetes-cda2a6372f44 (accessed Sep. 14, 2023).

[2] A. Spyker, A. Leung, and T. Bozarth, "The Evolution of Container Usage at Netflix," *Netflix Tech Blog*, Apr. 2017. [Online]. Available: https://netflixtechblog.com/the-evolution-of-container-usage-at-netflix-3abfc096781b (accessed Sep. 14, 2023).

[3] H. Zhu and C. Gehrmann, "AppArmor Profile Generator as a Cloud Service," English, in *Proceedings of the 11th International Conference on Cloud Computing and Services Science*, 11th International Conference on Cloud Computing and Services Science, CLOSER, SciTech Publishing, Apr. 2021, pp. 45–55. DOI: 10.5220/0010434100450055.

[4] *AppArmor*. [Online]. Available: https://apparmor.net/ (accessed Sep. 13, 2023).

[5] W. Zhang, P. Liu, and T. Jaeger, "Analyzing the Overhead of File Protection by Linux Security Modules," in *Proceedings of the 2021 ACM Asia Conference on Computer and Communications Security*, 2021, pp. 393–406.

[6] eBPF Foundation, *What is eBPF?* eBPF.io. [Online]. Available: https://ebpf.io/what-is-ebpf/ (accessed Sep. 14, 2023).

[7] Cilium, *Tetragon: Runtime Observability and Enforcement*, GitHub. [Online]. Available: https://github.com/cilium/tetragon (accessed Sep. 14, 2023).

[8] W. P. Findlay, *BPFContain*, GitHub. [Online]. Available: https://github.com/willfindlay/bpfcontain-rs (accessed Sep. 14, 2023).

[9] W. P. Findlay, "A Practical, Lightweight, and Flexible Confinement Framework in eBPF," M.S. thesis, Carleton University, Ottawa, Ontario, 2021.

[10] S. Smalley, T. Fraser, and C. Vance, *Linux Security Modules: General Security Hooks for Linux*, Linux Kernel Documentation. [Online]. Available: `https://docs.kernel.org/security/lsm.html` (accessed Sep. 15, 2023).

[11] Red Hat, *What is SELinux?* [Online]. Available: `https://www.redhat.com/en/topics/linux/what-is-selinux` (accessed Oct. 19, 2023).

[12] The Linux Foundation, *Landlock LSM: kernel documentation*, Linux Kernel Documentation. [Online]. Available: `https://docs.kernel.org/security/landlock.html` (accessed Oct. 19, 2023).

[13] The Linux Foundation, *capabilities(7)*, Linux Manual Page. [Online]. Available: `https://man7.org/linux/man-pages/man7/capabilities.7.html` (accessed Sep. 15, 2023).

[14] The Linux Foundation, *namespaces(7)*, Linux Manual Page. [Online]. Available: `https://man7.org/linux/man-pages/man7/namespaces.7.html` (accessed Sep. 18, 2023).

[15] The Linux Foundation, *cgroups(7)*, Linux Manual Page. [Online]. Available: `https://man7.org/linux/man-pages/man7/cgroups.7.html` (accessed Sep. 18, 2023).

[16] AppArmor, *Core Policy Reference Manual*, GitLab, 2018. [Online]. Available: `https://gitlab.com/apparmor/apparmor/-/wikis/AppArmor_Core_Policy_Reference` (accessed Sep. 14, 2023).

[17] Canonical, *Security - AppArmor*, Ubuntu Server Documentation, 2023. [Online]. Available: `https://ubuntu.com/server/docs/security-apparmor` (accessed Sep. 15, 2023).

[18] Facebook, *Katran*, GitHub. [Online]. Available: `https://github.com/facebookincubator/katran` (accessed Sep. 19, 2023).

[19] Cilium, *Program Types*, Cilium eBPF Documentation. [Online]. Available: `https://docs.cilium.io/en/stable/bpf/progtypes/` (accessed Sep. 19, 2023).

[20] The Linux Foundation, *BPF ring buffer*, Linux Kernel Documentation. [Online]. Available: `https://www.kernel.org/doc/html/next/bpf/ringbuf.html` (accessed Sep. 19, 2023).

[21] The Linux Foundation, *BPF Maps*, Linux Kernel Documentation. [Online]. Available: `https://www.kernel.org/doc/html/next/bpf/maps.html` (accessed Sep. 19, 2023).

[22] A. M. Turing *et al.*, "On computable numbers, with an application to the Entscheidungsproblem," *J. of Math*, vol. 58, no. 345-363, p. 5, 1936.

[23] The Linux Foundation, *BPF Design Q&A*, Linux Kernel Documentation. [Online]. Available: `https://www.kernel.org/doc/html/next/bpf/bpf_design_QA.html` (accessed Sep. 19, 2023).

[24] The Linux Foundation, *bpf-helpers(7)*, Linux Manual Page. [Online]. Available: `https://man7.org/linux/man-pages/man7/bpf-helpers.7.html` (accessed Sep. 19, 2023).

[25] IO Visor, *bpftrace*, GitHub. [Online]. Available: `https://github.com/iovisor/bpftrace` (accessed Sep. 19, 2023).

[26] M. A. Vieira, M. S. Castanho, R. D. Pacífico, E. R. Santos, E. P. C. Júnior, and L. F. Vieira, "Fast packet processing with ebpf and xdp: Concepts, code, challenges, and applications," *ACM Computing Surveys (CSUR)*, vol. 53, no. 1, pp. 1–36, 2020.

[27] S. Miano, M. Bertrone, F. Risso, M. Tumolo, and M. V. Bernal, "Creating complex network services with ebpf: Experience and lessons learned," in *2018 IEEE 19th International Conference on High Performance Switching and Routing (HPSR)*, IEEE, 2018, pp. 1–8.

[28] M. Xhonneux, F. Duchene, and O. Bonaventure, "Leveraging ebpf for programmable network functions with ipv6 segment routing," in *Proceedings of the 14th International Conference on emerging Networking EXperiments and Technologies*, 2018, pp. 67–72.

[29] W. Tu, J. Stringer, Y. Sun, and Y.-H. Wei, "Bringing the power of ebpf to open vswitch," in *Linux Plumbers Conference*, 2018, p. 11.

[30] A. Deepak, R. Huang, and P. Mehra, "Ebpf/xdp based firewall and packet filtering," in *Linux Plumbers Conference*, 2018.

[31] A. Mukherjee, "Enhancing Security in Docker Web Servers Using AppArmor and BPFtrace," Ph.D. dissertation, Purdue University, 2023.

[32] F. Loukidis-Andreou, I. Giannakopoulos, K. Doka, and N. Koziris, "Docker-sec: A fully automated container security enhancement mechanism," in *2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS)*, 2018, pp. 1561–1564. DOI: `10.1109/ICDCS.2018.00169`.

[33] Gregg, Brendan, *opensnoop.bt*, GitHub. [Online]. Available: https://github.com/iovisor/bpftrace/blob/master/tools/opensnoop.bt (accessed Sep. 19, 2023).

[34] Y. Agman, "Using LSM Hooks with Tracee to Overcome Gaps with Syscall Tracing," *Aqua Blog*, May 2021. [Online]. Available: https://blog.aquasec.com/linux-vulnerabilitie-tracee (accessed Sep. 20, 2023).

[35] R. Sekar, H. Kimm, and R. Aich, "Eaudit: A fast, scalable and deployable audit data collection system," in *2024 IEEE Symposium on Security and Privacy (SP)*, IEEE Computer Society, 2023, pp. 87–87.

[36] J. Levin and T. A. Benson, "Viperprobe: Rethinking microservice observability with ebpf," in *2020 IEEE 9th International Conference on Cloud Networking (CloudNet)*, 2020, pp. 1–8. DOI: 10.1109/CloudNet51028.2020.9335808.

[37] Chen, Hengqi, *[bpf-next,v2] bpf: expose bpf_d_path helper to vfs_* and security_* functions*, Linux Patchwork mailing list. [Online]. Available: https://patchwork.kernel.org/project/netdevbpf/patch/20210719151753.399227-1-hengqi.chen@gmail.com/#24329481 (accessed Nov. 9, 2023).

[38] D. Le, "Load Testing Backend Services Using K6," *Earthly Blog*, Jul. 2023. [Online]. Available: https://earthly.dev/blog/load-testing-using-k6/ (accessed Oct. 30, 2023).

[39] Cuong, Le Dinh, *earthly-k6-load-test-service*, GitHub. [Online]. Available: https://github.com/cuongld2/earthly-k6-load-test-service (accessed Oct. 30, 2023).

[40] Phoronix Media, *Timed Linux Kernel Compilation*. [Online]. Available: https://openbenchmarking.org/test/pts/build-linux-kernel (accessed Dec. 4, 2023).

[41] A. Traeger, E. Zadok, N. Joukov, and C. P. Wright, "A nine year study of file system and storage benchmarking," *ACM Trans. Storage*, vol. 4, no. 2, May 2008, ISSN: 1553-3077. DOI: 10.1145/1367829.1367831. [Online]. Available: https://doi.org/10.1145/1367829.1367831.

[42] The Linux Foundation, *fanotify(7)*, Linux Manual Page. [Online]. Available: https://man7.org/linux/man-pages/man7/fanotify.7.html (accessed Nov. 22, 2023).

[43] 0xn3va, *Excessive Capabilities - Application Security Cheat Sheet*. [Online]. Available: https://0xn3va.gitbook.io/cheat-sheets/container/escaping/excessive-capabilities#cap_sys_admin (accessed Nov. 22, 2023).

[44] IO Visor, *BPF Features by Linux Kernel Version*, GitHub. [Online]. Available: `https://github.com/iovisor/bcc/blob/master/docs/kernel-versions.md` (accessed Nov. 21, 2023).

[45] Song, Yonghong, *bpf: Allow struct argument in trampoline based programs*, GitHub commit. [Online]. Available: `https://github.com/torvalds/linux/commit/720e6a435194fb5237833a4a7ec6aa60a78964a8` (accessed Nov. 21, 2023).

[46] Chen, Hengqi, *The function security_path_chown arg1 type STRUCT is unsupported.* GitHub issue. [Online]. Available: `https://github.com/iovisor/bcc/issues/3657` (accessed Nov. 21, 2023).

# Appendix A
# Glossary

## A.1 Acronyms

**AWS** Amazon Web Services

**BTF** BPF Type Format

**DAC** Discretionary Access Control

**IPC** Inter-Process Communication

**LSM** Linux Security Module

**MAC** Mandatory Access Control

**PID** Process ID

**REST** REpresentational State Transfer

**RHEL** Red Hat Enterprise Linux

# Appendix B
# Program Listings

## B.1 A Simple eBPF Program

```c
#include "vmlinux.h"
#include "bpf_helpers.h"

char __license[] SEC("license") = "Dual MIT/GPL";

struct event {
  u32 pid;
  u8 comm[80];
};

struct {
  __uint(type, BPF_MAP_TYPE_RINGBUF);
  __uint(max_entries, 1 << 24);
} events SEC(".maps");

// Force emitting struct event into the ELF.
const struct event *unused __attribute__((unused));

SEC("kprobe/sys_execve")
int kprobe_execve(struct pt_regs *ctx) {
  u64 id   = bpf_get_current_pid_tgid();
  u32 tgid = id >> 32;
  struct event *task_info;

  task_info = bpf_ringbuf_reserve(&events, sizeof(struct event
    ), 0);
  if (!task_info) {
    return 0;
  }

  task_info->pid = tgid;
  bpf_get_current_comm(&task_info->comm, 80);

  bpf_ringbuf_submit(task_info, 0);

```

```
  return 0;
}
```

## B.2 Path Export Function

```
static __attribute__((always_inline)) size_t read_path_data(
    struct path *path, struct path_data *pd) {
    u8 slash = '/';
    u8 zero = 0;
    u32 buf_off = (MAX_PATH_LEN >> 1);
    unsigned int len;
    unsigned int off;
    int sz;

    struct path f_path;
    bpf_probe_read(&f_path, sizeof(f_path), path);
    struct dentry *dentry = f_path.dentry;
    struct dentry *d_parent;
    struct qstr d_name;

    #pragma unroll
    for (int i = 0; i < MAX_PATH_DEPTH; i++) {
        // break if we've reached root
        d_parent = BPF_CORE_READ(dentry, d_parent);
        if (dentry == d_parent) break;

        d_name = BPF_CORE_READ(dentry, d_name);

        len = (d_name.len + 1) & (MAX_DNAME_LEN - 1);
        off = buf_off - len; // read current path starting at
    buf_off - len
        sz = 0;

        if (off <= buf_off) { // catch underflow
            len = len & ((MAX_PATH_LEN >> 1) - 1);
            sz = bpf_probe_read_str(
                &(pd->buf[off & ((MAX_PATH_LEN >> 1) - 1)]),
    len, (void *)d_name.name
            );
        } else break;

        if (sz > 1) {
            buf_off -= 1;
            bpf_probe_read(&(pd->buf[buf_off & (MAX_PATH_LEN -
     1)]), 1, &slash);
            buf_off -= sz - 1;
        } else break; // path should not be null or empty

        dentry = d_parent;
    }

```

```
    pd->truncated = dentry != d_parent;

    // prepend slash and null-terminate
    buf_off -= 1;
    bpf_probe_read(&(pd->buf[buf_off & (MAX_PATH_LEN - 1)]),
    1, &slash);
    bpf_probe_read(&(pd->buf[(MAX_PATH_LEN >> 1) - 1]), 1, &
    zero);

    pd->offset = buf_off;
    return buf_off;
}
```

## B.3 k6 Load Test Script

```
import http from 'k6/http';

import { check } from 'k6';
import exec from 'k6/execution';
import { randomString } from 'https://jslib.k6.io/k6-utils
    /1.2.0/index.js';

const serviceBaseUrl = 'http://blogapi-svc.workload:8089';

export const options = {
    vus: 3,
    stages:[
        {duration: '30s', target: 3},
        {duration: '29m', target: 3},
        {duration: '30s', target: 0},
    ],
};

function createUsers(nUsers) {
  const userData = [];
  for (var i = 0; i < nUsers; i++) {
    userData.push({
      username: 'user' + i,
      password: 'password' + i,
      fullname: 'User ' + i,
    });
  }

  return userData;
}

export function setup() {
  const users = 3;
  const userData = createUsers(users);

  for (const user of userData) {
```

```
36      const responseReg = http.post(serviceBaseUrl + '/user',
37        JSON.stringify(user),
38        { headers: { 'Content-Type': 'application/json' } }
39      );
40      check(responseReg, { 'status was 200 or 400': (r) => r.
        status == 200 || r.status == 400});
41    }
42
43    return userData;
44  }
45
46  export default function (data) {
47    const id = exec.vu.idInTest;
48    const user = data[id % data.length]
49
50    const responseAuthen = http.post(serviceBaseUrl + '/
        authenticate', JSON.stringify({
51        username: user.username,
52        password: user.password
53      }),
54      { headers: { 'Content-Type': 'application/json' } }
55    );
56    check(responseAuthen, { 'status was 200': (r) => r.status ==
        200 });
57
58    const authenToken = responseAuthen.json().access_token;
59    const params = {
60      headers: { 'Authorization': 'Bearer ' + authenToken },
61    };
62
63    const responseCreateNewBlog = http.post(serviceBaseUrl + '/
        blog', JSON.stringify({
64        title:"Blog title: " + randomString(10),
65        content: "Blog entry: " + randomString(100),
66        author: data.fullname
67    }), params);
68    check(responseCreateNewBlog, { 'status was 200': (r) => r.
        status == 200 });
69  }
```