A Practical, Lightweight, and Flexible Confinement Framework in eBPF

by

William P. Findlay

A thesis submitted to the Faculty of Graduate and Postdoctoral Affairs

in partial fulfillment of the requirements for the degree of

Master of Computer Science

August, 2021

Carleton University

Ottawa, Ontario

To my parents and grandparents, for believing in me

even when I didn't believe in myself.

Abstract

Confining operating system processes is essential for preserving least-privilege access to system resources, hardening the system against successful exploitation by malicious actors. Classically, confinement on Linux has been accomplished through a variety of disparate confinement primitives, each targeting a different aspect of process behaviour and each with its own set of policy semantics. This has led to difficulties in realizing practical confinement goals, due to the complexities, inter-dependence relationships, and semantic gaps that arise from recombining existing confinement primitives in unintended ways. Linux containers are a particularly poignant example of this phenomenon, with existing container security policies often being overly complex and overly permissive in practice.

To better isolate user processes and achieve practical confinement goals, we argue that novel confinement mechanisms are needed to bridge the semantic gap between security policy and enforcement. We hypothesize that a new Linux kernel technology, eBPF, enables the creation of precisely such a confinement mechanism. eBPF programs can be dynamically loaded into the kernel by a privileged process and are checked for safety before they run in kernelspace. This approach affords an opportunity to create an adoptable, container-specific confinement mechanism without tying the kernel down to one specific implementation. Further, an eBPF-based confinement solution can be loaded and unloaded at runtime, without updating or even restarting the operating system kernel; this prop-

erty enables rapid prototyping and debugging, similar in spirit to how we debug userspace applications in practice.

In this thesis, we present the design and implementation of two novel confinement solutions based on eBPF, BPFBox and its successor, BPFContain. We discuss issues in the Linux confinement space that motivated the creation of BPFBox and BPFContain, discuss policy examples, and present the results of a performance evaluation and informal security analysis. Results from this research indicate that BPFBox and BPFContain incur modest overhead despite their increased flexibility over existing Linux security solutions. We also find that there is significant opportunity to improve BPFBox and BPFContain, and to introduce future security mechanisms based on eBPF.

Prior Publication

A publication and pre-print have arisen as a direct result of the research in this thesis. While these works represent joint contributions of all authors, any sections reproduced in this thesis represent the sole work of the thesis author, with editorial and positioning contributions by co-authors. Each work is listed below.

Chapter 4 contains text and ideas from our paper "BPFBox: Simple Precise Process Confinement in eBPF (Extended BPF)" [59], co-authored with Anil Somayaji and David Barrera, and published at the Cloud Computer Security Workshop (CCSW) 2020 as part of the ACM CCS conference.

Chapter 5 contains some text and ideas from our paper "BPFContain: Fixing the Soft Underbelly of Container Security" [58], co-authored with David Barrera and Anil Somayaji. An early draft of this work is available on the ArXiV pre-print database, although it differs substantially from the version presented in this thesis.

Contents

List of Figures

List of Tables

List of Code Listings

Chapter 1.

Introduction

Virtualization is not confinement. Put simply, that which we can *see* is not the same thing as that which we can *do*. To security experts, this may be an obvious statement; however, not every user of an information technology system is a security expert, nor should they be. Unfortunately, these two disparate yet related concepts are often conflated, leading to dangerous security assumptions in practice. In particular, we tend to assume that a virtualized system is the same thing as a secure system, which may not necessarily be the case. Confinement is critically important to maintaining the *principle of least-privilege* [124], a quintessential property of a truly secure system [107]. Despite playing a critical role in systems security, the state of confinement on Linux is ill-suited to meet the practical needs end users.

Existing Linux confinement mechanisms are complex, and often target specific use cases beyond simple confinement. This results in a motley collection of isolation primitives being used to lock down basic application functionality. Namespaces

and cgroups virtualize system resources while Linux security modules, `seccomp(2)`, discretionary access controls, and more are used to restrict access. The need to combine these mechanisms begets unintuitive inter-dependence relationships that lead to additional complexity and security policies that are both painful to write and difficult to audit. In turn, these difficulties weaken the ultimate authority of the system administrator, shifting the burden of confinement onto distribution vendors or application authors.

Linux containers [50, 88, 94, 140] are a motivating example of this phenomenon. Intuitively, a user might be motivated to use a container to *contain things*. The reality of container confinement does not match this intuition. Containers are nothing more than a group of related processes (or even a single process) united by a shared view of system resources (based on Linux virtualization primitives). Despite their name, containers in general do a very poor job of actually containing anything. In particular, security defaults in container management engines like Docker [50] or LXC (Linux Containers) [94] rely on a myriad of unrelated confinement primitives, many of which were designed to holistically lock down entire systems. Misuse of these primitives results in a complex entanglement of related policies that ultimately must be simplified down to their basest elements. The result is containers that "just work", albeit operating under highly coarse-grained policies that provide little protection in practice [88, 140].

This thesis argues that the key to implementing lightweight confinement policies that work well in the context of containers lies in simplifying and unifying the underlying confinement framework, and bridging the semantic gap between confinement policies and the applications or containers they are designed to protect. In the past,

this may have been a difficult problem to solve. After all, existing confinement mechanisms are designed for general-purpose use cases, and the precise definition of what constitutes "container semantics" varies depending on the needs of the container deployment and the design of the container management engine. We posit that the key to designing a confinement mechanism that meets these goals is the ability to dynamically modify and extend the kernel's security monitor, building a security framework that is easy to deploy and simple to extend and modify. A new Linux technology, eBPF, now enables us to fill this technological gap.

Specifically, eBPF [65, 139] enables a privileged userspace process to safely and dynamically add simple hooking and filtering logic to key components of the kernel. By designing and deploying a specific set of eBPF programs, we can adjust the kernel's security semantics, without necessarily tying it down to a specific "one-size-fits-all" confinement solution. This enables us to build application- or container-specific policies that scale well and meet the needs of end users.

To improve the status quo of confinement on Linux, we present two research prototypes, BPFBox and its successor BPFContain. The former is a novel application sandboxing framework, and the latter extends that framework to work well in the context of container security. Both research systems are implemented using eBPF, the first such systems of their kind. In this thesis, we present a motivating re-framing of the confinement problem, examine the design and implementation of BPFBox and BPFContain, and show that they improve application and container security without a significant impact on system performance.

1.1. Research Questions

In this thesis, we consider the following research questions.

RQ1 What difficulties in the current state of Linux confinement lead to the semantic gaps between policies and the entities they are designed to lock down? How can we design a novel confinement mechanism to rectify these difficulties?

RQ2 Can eBPF be used to implement a full-featured confinement framework? What would such a framework look like and how could it be made to model container semantics?

RQ3 What levels of security and performance can we expect from a confinement mechanism designed around eBPF? What improvements to eBPF would be required for a complete solution?

1.2. Motivation

1.2.1. Contextualizing the Problem

Containers are *everywhere*. In the cloud, containers form the backbone of cloud-native computation. Kubernetes [80] clusters drive the microservices that power scalable web applications. In devops, Docker [50] containers often form the backbone of continuous integration workflows, providing reproducible environments for development, testing, and debugging. On the desktop, containerized package managers like Snap [132] and FlatPak [60] offer self-contained, isolated software bundles,

facilitating a smooth software installation process (mostly) free of dependency management concerns.

Despite a steadily increasing prevalence, containers face major adoptability challenges in deployments[1] where they are expected to outright replace virtual machines. Unlike virtual machines, which are abstracted away from the host and interact with a hypervisor, containers interact directly with the host operating system kernel. This means that, while much more lightweight than hypervisor-based virtualization, containers are inherently less isolated from each other and from the host system in general [22, 88, 101, 140]. In order to have truly secure containers, we must take great care to ensure that a container is properly *confined*. In practice, this means restricting the processes that run within the container from performing certain actions that can negatively impact or damage the system. As we have already discussed, virtualization primitives alone are not enough to achieve proper isolation. These primitives *must* be combined with confinement mechanisms and these confinement mechanisms *must* be applied properly. Otherwise, we risk overprivilege, resulting in potential violations of our security model.

Container security issues are widely studied in the literature [1, 22, 88, 101, 140]. Despite the fact that containers run directly on the host operating system and share a single kernel with other native processes, security is generally treated as an afterthought in the design of container management engines. If we truly want containers to be as secure as virtual machines, we must rethink the way we secure container deployments. Security must be prioritized from the ground up but must not get in the way of functionality. Existing container frameworks accomplish the second goal

[1] E.g., Cloud-Native deployments [19].

but not the first.

Docker, for instance, applies a default AppArmor policy revoking access to only the most sensitive kernel interfaces like procfs and sysfs and disabling the ability to mount new filesystems. Beyond these basic controls, the container has full permission to access all filesystem resources, has access to several POSIX capabilities, and may unmount any filesystem [49, 51]. Even worse, a kernel that does not support AppArmor or that is not properly configured is left totally bereft of this protection to begin with. Docker complements its default AppArmor profile with a set of sensible seccomp rules, revoking access to many privileged system calls. While such a policy *does* help to harden the container, it remains overly-generalized [140] and does not uniquely capture the needs of every container deployment. Users who wish to grant additional permissions to their container are left with the choice of either writing and auditing custom AppArmor and seccomp policies or outright disabling protections altogether with the `--privileged` flag.

Docker is but the most prominent example among many. In general, all existing container management frameworks rely on a patchwork of isolation mechanisms, each enforcing its own confinement policy and each with varying degrees of generalization. As a result, these policies are often difficult to reason about, and thus are difficult to effectively audit. A vulnerability in any individual mechanism or a misconfiguration in any individual policy opens the container or the host system itself up to attack. Blanket defaults are often ineffective for specific use cases and result in situations where the end user is forced to either abandon all hope of security or muddle through the configuration of multiple policy enforcement mechanisms.

1.2.2. Why Design a New Confinement Framework?

The process confinement problem dates back half a century [82]. Since the advent of multi-processing and multi-tenant systems in the 1960s and 1970s [36, 119, 150] with Multics and Unix, security experts have been concerned with designing systems in such a way that two running programs minimally interfere with one another. Since then, an abundance of tools and frameworks, some more practical than others, have been proposed to limit the damage that untrusted software can do to the system as a whole [128]. These are covered in more depth in Chapter 2. For now, we focus on why it might be prudent to design yet another confinement framework amidst this veritable ocean of prior work.

The Linux kernel already provides a number of confinement primitives. Seccomp allows for a process to confine itself by filtering the system calls it can make. Mandatory access control solutions based on LSM (Linux Security Modules) hooks can be configured to define and enforce powerful per-application policies, protecting system resources from unwanted access. Unix DAC (Discretionary Access Control) [76, 107, 119, 128] restricts access to system resources according to resource owners, groups, permission bits, and access control lists. When applied to container security, the common problem faced by these security mechanisms is that they are being applied to solve a problem for which they were not originally designed. To solve this problem, we seek to design a unifying security abstraction for containers and apply this abstraction to enforce per-container policy in kernelspace.

From the kernel's perspective, a containerized process is just like any other [140]. While it may be virtualized under one or more namespace and process control groups,

there is no precise definition of what exactly constitutes a *container*. This lack of a solid abstraction widens the semantic gap between traditional policy enforcement mechanisms and security policy designed to protect containers. In defining a new policy enforcement mechanism focused specifically on containers, we have an opportunity to narrow this semantic gap, simplify the resulting policies, and eliminate the need to combine several security mechanisms together to do a job that could be accomplished by just one. Since our proposed solution is based on eBPF, it requires no modification of the kernel and can be dynamically loaded at runtime. This means that we can provide such a unified abstraction without sacrificing forward or backward compatibility with alternative approaches.

1.2.3. Why eBPF?

An eBPF-based confinement mechanism provides several advantages over traditional confinement models. Firstly, eBPF is *lightweight*. eBPF programs can monitor many aspects of system behaviour, from userspace function calls to kernelspace function calls, system calls, security hooks, and the networking stack. Data from these events can be aggregated in real time in kernelspace, providing an extensible, performant, and flexible framework for modelling relationships and enforcing policy decisions based on these relationships. A single security mechanism based on eBPF can combine the advantages of several disparate mechanisms that would ordinarily need to be combined together to provide full protection. This notion is the antithesis of the way container security is currently done on Linux. Rather than combining namespaces, cgroups, seccomp, and mandatory access control together, eBPF provides the

opportunity to design a single framework providing the advantages of each.

A second advantage of eBPF for writing a security framework is that it is *dynamic*. eBPF programs can be loaded into the kernel dynamically and attached to multiple events. Instrumenting a system event with eBPF can be done at runtime, *without* the need to modify the kernel in any way. Similarly, eBPF maps, the canonical runtime data store for eBPF programs, can be loaded, unloaded, modified, and queried at runtime from both userspace and kernelspace, providing a rich substrate for a dynamic model of system behaviour. These properties culminate in the ability to design a flexible security mechanism without tying the kernel down to any one particular abstraction. In the context of container security, this is a particularly important goal, as containers are traditionally a *userspace* concept, glued together with various abstractions provided by the kernel.

Production safety is a third advantage provided by eBPF. All eBPF programs go through a verification process before they are loaded into the kernel. The eBPF verifier analyzes the program, asserting that it conforms to a number of safety requirements, such as program termination[2], memory safety, and read-only access to kernel data structures. While itself not formally verified, the eBPF verifier facilitates the adoption of new eBPF programs into production use cases, since an eBPF program is far less likely to adversely impact a production system than other methods of extending the kernel (e.g. kernel patches and loadable kernel modules). In fact, eBPF is already being used in production at large datacenters by Facebook, Netflix, Google, and others to monitor server workloads for security and performance regressions [65]. These factors make eBPF a promising choice for designing an *adoptable*

[2]This property is enforceable due to the fact that eBPF programs are not Turing complete [65].

security mechanism.

In summary, eBPF offers unique and promising advantages for developing novel security mechanisms. Its lightweight execution model coupled with the flexibility to monitor and aggregate events across userspace and kernelspace provide the ability to control and audit nearly any aspect of the running system. eBPF maps, shareable across programs and between userspace and the kernel offer a means of aggregating data from multiple sources at runtime and using it to inform policy decisions across domains. A security mechanism based on eBPF can be dynamically loaded into the kernel as needed, and eBPF's safety guarantees combined with its increasing adoption in production use cases provide strong adoptability advantages. This means that a security mechanism based on eBPF can be both adoptable and effective.

1.3. Contributions

This thesis offers several contributions to the fields of computer science, computer security, and confinement. These contributions are as follows.

- We present a novel framing of the confinement problem (Chapter 3) in the context of Linux, arguing that inherent complexity and misuse of existing primitives has led to semantic gaps in confinement. We argue that these gaps, in turn, impact security by encouraging the adoption of overly-generic polices that impact each other in unforeseen ways.

- We present the design and implementation of two eBPF-based confinement engines, BPFBox (Chapter 4) and BPFContain (Chapter 5). The former is a

prototype for eBPF-based confinement and the latter extends BPFBox, improving its security and introducing a model for container-specific confinement. BPFBox and BPFContain are the first high-level, eBPF-based confinement frameworks of their kind.

- We evaluate (Chapter 6) BPFBox and BPFContain in the context of their performance overhead and security. Specifically, we present results from benchmarks along with an informal security analysis. We also discuss how extensions on top of BPFBox and BPFContain could improve their performance and security in the future.

1.4. Outline

The rest of this thesis proceeds as follows. Chapter 2 presents detailed background information on virtualization, confinement, operating system security, and eBPF. Chapter 3 presents a novel framing of the confinement problem, outlining the motivation and design goals for BPFBox and BPFContain, and presenting a threat model for confinement. Chapter 4 describes the design and implementation of the initial BPFBox prototype and documents its original policy language. Chapter 5 describes the design and implementation of BPFContain, discusses how it has evolved from BPFBox, and highlights opportunities for future extensions that can make BPFContain more useful for confining containers.

Chapter 6 presents an evaluation of the BPFBox and BPFContain prototypes from the perspective of performance and security. We present benchmarking data

comparing both systems with AppArmor, a popular LSM (Linux Security Modules) implementation of mandatory access control. We also present a security analysis of BPFBox and BPFContain, highlighting areas of weakness and specific aspects of BPFBox upon which BPFContain improves. Chapter 7 presents policy examples for BPFBox and BPFContain and provides a detailed comparison, highlighting the strengths and weaknesses of each. Chapter 8 concludes with a high-level discussion on BPFBox and BPFContain, including limitations and opportunities for future work.

Chapter 2.

Background and Related Work

This chapter presents technical background information required to understand this thesis and discusses related work from the perspective of industry and academia. Section 2.1 presents technical background on historical models for process-level confinement, with a particular emphasis on Multics, Unix, and Unix derivatives. Section 2.2 focuses on subsequent extensions to the Unix security model and covers related work in the confinement space. Section 2.3 examines process-level virtualization technologies in Unix-like operating systems. Section 2.4 discusses the differences between hypervisor- and container-based virtualization, container security, and presents related work in the container security space. Finally, Section 2.5 presents a detailed history of eBPF, describes its architectural components and features, and discusses use cases in security and beyond.

2.1. Classic Unix Process Security Model

This section reviews foundational concepts in OS (Operating System) (particularly Unix-like OSs and Linux) security. In particular, we discuss the *reference monitor concept*, *virtual memory and user processes*, and *discretionary access control* and how these concepts interact to form the backbone of process-level security in Unix. The goal of this section is to help the reader build a mental model of how operating systems isolate and protect processes and system resources at the most fundamental level. Readers familiar with these OS security concepts can skip this section in favour of Section 2.2 which examines more advanced security mechanisms and discusses related work.

2.1.1. The Reference Monitor

The *reference monitor concept*, first introduced in the landmark 1972 Anderson Report [2], was among the earliest complete descriptions of a full access control mechanism and remains influential in operating system design to date. The reference monitor is an abstract model for a secure reference validation mechanism built into the operating system. The model partitions the system into *subjects* (users, processes, etc.) and *objects* (system resources). Subjects request access to objects and the reference monitor checks this access against a known list of allowed accesses, parameterized by the subject, object, and requested access. The software implementation of a reference monitor is known as the *security kernel*. Figure 2.1 depicts the reference monitor concept as it was first presented by Anderson [2].

While the majority of modern operating systems do not include a security kernel as

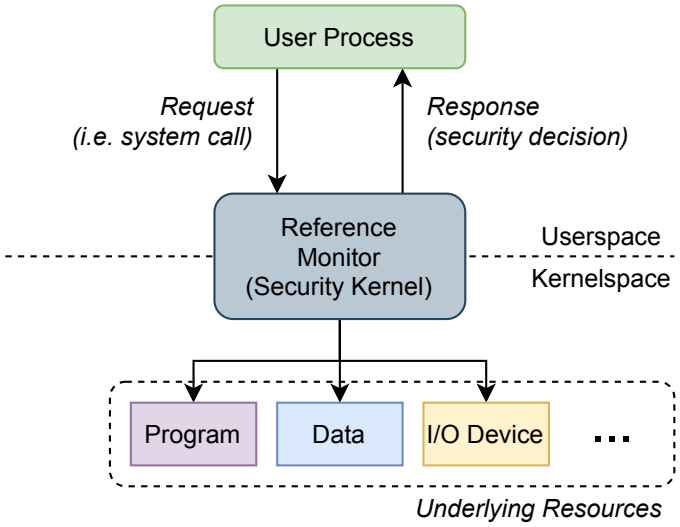

Figure 2.1: The reference monitor concept as outlined in the Anderson Report. Redrawn and adapted from Anderson [2]. User processes make requests (e.g. via system calls to the operating system). The OS kernel invokes the reference monitor, which is implemented in software as a security kernel. The reference monitor queries its security policy taking the subject, object, and other parameters as input. As output, it returns a security decision (i.e. whether the requested access should be *allowed* or *denied*).

described by Anderson, the reference monitor architecture has informed the design of modern access control mechanisms and models the reference validation process that occurs when the kernel is servicing userspace requests (i.e. system calls) [107]. In order for such a design to be considered valid, Anderson enumerates three key properties: (i) Tamper Resistance; (ii) Complete Mediation; and (iii) Verifiability. These properties facilitate reasoning about the security of modern access control mechanisms, even if they do not strictly adhere to the reference monitor model.

Tamper Resistance In order for the reference monitor to be considered *tamper resistant*, an unauthorized party must not be able to alter the reference monitor's code or modify any data (e.g. memory, persistent storage) that the reference monitor relies on to enforce correct reference validation [2]. This property follows from the fact that unauthorized tampering with the reference monitor totally invalidates any security guarantees.

Complete Mediation The property of *complete mediation* means that the reference monitor should be invoked on all security sensitive events. It should be impossible for an attacker to bypass the reference monitor in any way. Any software that is not subject to reference validation should be considered a part of the reference monitor [2].

Verifiability *Verifiability* refers to the ability to reason about or prove the correctness of the reference monitor (i.e. that the first two properties hold). Formal verification methods are the best way of achieving verifiability, although this may not necessarily be practical for highly complex systems. For this reason, it is rec-

ommended to design the reference monitor in such a way that verifiability is maximized [2].

2.1.2. Virtual Memory and Memory Protection

Virtual memory [45] is a mechanism for mapping *virtual* memory addresses to *physical* machine addresses. First introduced in the 1950s, the original goal of virtual memory was to make it easier for programmers to manipulate memory without worrying about the underlying details of storage configuration [45]. With the advent of multi-processing systems, virtual memory took on a new role — separation of memory resources between distinct user processes. This separation is a fundamental notion for secure multi-processing; two user processes should not be able to interfere with each other's memory, and a user process should not be able to interfere with the OS kernel, resident in ring 0 memory.

By partitioning memory into virtual address spaces, virtual memory forms the most fundamental isolation barrier between user processes. To accomplish this goal, a hardware mechanism, the MMU (Memory Management Unit), translates virtual addresses to physical addresses using a *page table* maintained by the operating system in main memory. To accelerate the translation of memory addresses, processors cache this mapping in a specialized cache area called the TLB (Translation Lookaside Buffer). In Unix, each user process gets its own virtual address space by default, maintained in a per-process page table. Where necessary, this isolation may be voluntarily broken using memory sharing mechanisms provided by the OS kernel (e.g. multi-threading or shared memory mappings). The kernel also gets its own

address space which maps the entirety of physical memory.

While virtual memory can help isolate user processes from each other and user process from the kernel, additional protection mechanisms are required to strengthen this isolation. To this end, the CPU (Central Processing Unit) ISA (Instruction Set Architecture) generally defines memory protection bits that can be applied to physical pages and enforced in the MMU. For instance, individual pages can be marked as readable, writable, and/or executable depending on how the memory is to be used. How these protections are used is generally up to the operating system; for instance, modern operating systems often enforce a policy where pages marked writable are not allowed to be marked executable and vice versa (this is often referred to as W$\oplus$X). This helps to prevent basic buffer overflow attacks. Another important protection mechanism employed by the operating system is ASLR (Address Space Layout Randomization), which slightly randomizes virtual address space mappings, making them unpredictable and thus more difficult for attackers to exploit consistently. A similar mechanism, KASLR (Kernel ASLR), protects the kernel. The `grsecruity` patch suite [68] offers additional memory protection for the Linux kernel, hardening the boundary between userspace and kernelspace and applying additional mitigations to prevent return oriented programming attacks.

Additional protections are afforded by *memory protection rings*, a concept first introduced by Multics [36, 150] in the mid-1960s. In the original design, 64 protection rings were defined in hardware, numbered from 0–63. A task running in a higher-numbered protection ring would be unable to access any memory marked with a lower-numbered protection ring, effectively isolating sensitive code and data from unprivileged tasks. Modern CPUs carry forward this notion of protection rings,

although a typical modern processor only defines far fewer protection rings in practice. For instance, x86 only implements four protection rings in total. Modern OSs, including Linux, generally only use *two* of these rings, ring 0 and ring 3 (on x86 processors) for kernelspace and userspace respectively. Figure 2.2 depicts this design. Lee *et al.* [84] have proposed using the remaining two rings for finer-grained isolation on x86.

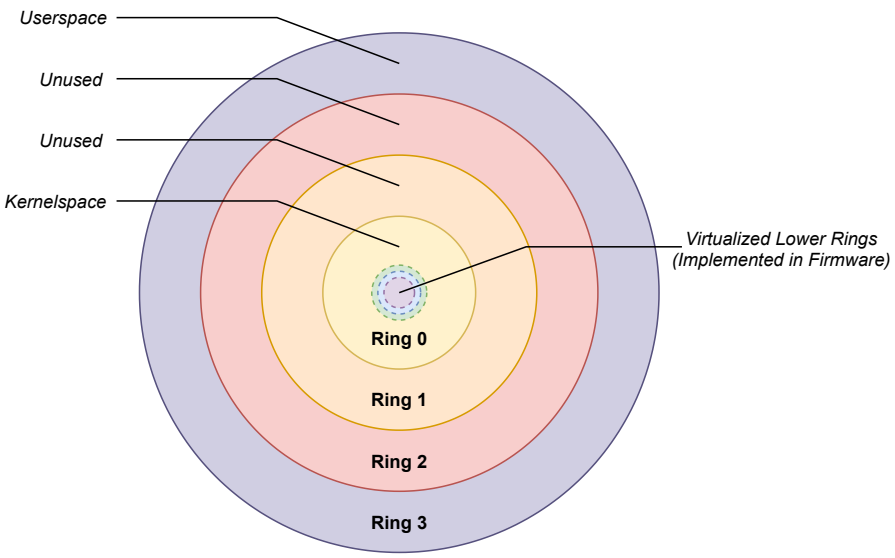

Figure 2.2: A visualization of protection rings on x86 CPUs. Ring 3 contains *userspace*, the address space of ordinary applications that run in user mode. Ring 0 contains *kernselspace*, the kernel's address space. Code running in ring 0 is said to run in *supervisor mode*. Rings 1–2 are generally unused by COTS (Commercial Off-The-Shelf) operating systems. CPUs often implement "lower" rings (-1, -2, etc.) by virtualizing ring 0 in firmware. These are typically reserved for the hypervisor and other hardware-backed features like Intel's System Management Mode [144].

2.1.3. Discretionary Access Control

DAC (Discretionary Access Control) comprises the most basic form of access control in many operating systems, including Linux, other Unix-like operating systems, and Microsoft Windows. First formalized in the 1983 US Department of Defense standard [149], a discretionary access control mechanism partitions and labels system objects (i.e. resources such as files) by the subjects (i.e. actors such as users and user processes) that *own* them. The corresponding resource owner then has full authority to decide which subjects have access to its owned objects. This notion of ultimate authority over a subject's owned objects constitutes the primary difference between discretionary access control and mandatory access control, which is covered in Section 2.2.2.

Classically, Unix-like systems have implemented discretionary access control in the form of *permission bits* and *access control lists*. Each process on the system runs under a specific user and group ID, which uniquely identify the user and group of the process respectively, where each group is a collection of one or more users. Permission bits and access control lists denote access permissions according to the user ID and group ID of the process requesting the resource. These permissions can in turn be overridden by the *superuser* or *root* [76, 107].

Permission Bits

Permission bits in Unix are special metadata associated with a file that determine coarse-grained access to the file according to a subject's UID (User ID) and GID (Group ID). Permission bits are divided into three sections: *User*, *Group*, and *Other*.

The *User* bits apply to subjects whose UID matches the resource owner's UID, while the *Group* bits consider the GID instead. In all other cases (i.e. when neither the UID nor the GID matches), the *Other* bits determine the allowed access. To determine which access should be allowed, permission bits encode a coarse-grained *access vector*, specifying read, write, and execute access on a file or directory (in the case of a directory, execute access implies the ability to `chdir(2)` into that directory).

While convenient, permission bits are generally insufficient to provide legitimate security guarantees to modern systems [76, 107]. In particular, permission bits encode coarse-grained permissions and apply these permissions in a coarse-grained, all-or-nothing, manner. For instance, consider the use case of granting read-only access to another user. Specifying such access as part of the *Other* bitmask implies granting access to any user on the system. Specifying access to a particular *Group* is slightly better, but the resource owner has no direct control over which other users belong to this group, now or in the future. Thus, we cannot say with certainty that we may specify such access without violating our security assumptions.

Access Control Lists

ACLs (Access Control Lists) offer a slightly more granular alternative to permission bits, at the expense of increased complexity [76, 107]. Unlike permission bits, which rely on three coarse-grained subject categories (*User*, *Group*, and *Other*), an access control list defines a set of subjects and their corresponding permissions for every object. It may be helpful to think of this as breaking up the *Other* category into distinct subjects rather than granting or revoking blanket access to all other users on the system.

Capability lists, complementary to access control lists, define a set of objects and allowed access patterns for every subject. A capability list for a given subject can be derived by taking the set of all access control lists for every object and vice versa [107]. Together, the set of all access control lists (or capability lists) forms an *access matrix*, describing the DAC policy over the entire system. Figure 2.3 depicts this relationship.

The Superuser and Setuid

To facilitate system administration, many DAC schemes incorporate the notion of a *superuser* or *administrator role* into their model. In Unix and Unix-like operating systems, the superuser or *root* user is denoted by the UID of zero. Any process running with the EUID (Effective User ID) of zero is said to be *root-privileged*. These root-privileged processes can then override the system's DAC policy, bypassing permission bits and access control entries on system objects.

In many cases, a program requires additional privileges in order to function. For instance, a `login` program would require the ability to read security-sensitive password entries in `/etc/shadow`. To achieve such functionality, Unix provides special `setuid` and `setgid` permission bits that implicitly set the effective user and group IDs of a process to those of the file owner. A sufficiently-privileged process may also change its own EUID or EGID (Effective Group ID) at runtime using the `setuid(2)` and `setgid(2)` family of system calls. The example login program, for instance, could use these system calls to drop its privileges to those of the user being logged in. While necessary under the Unix DAC model, setuid and setgid binaries have long been the target of exploitation, particularly for privilege escalation attacks [47, 76,

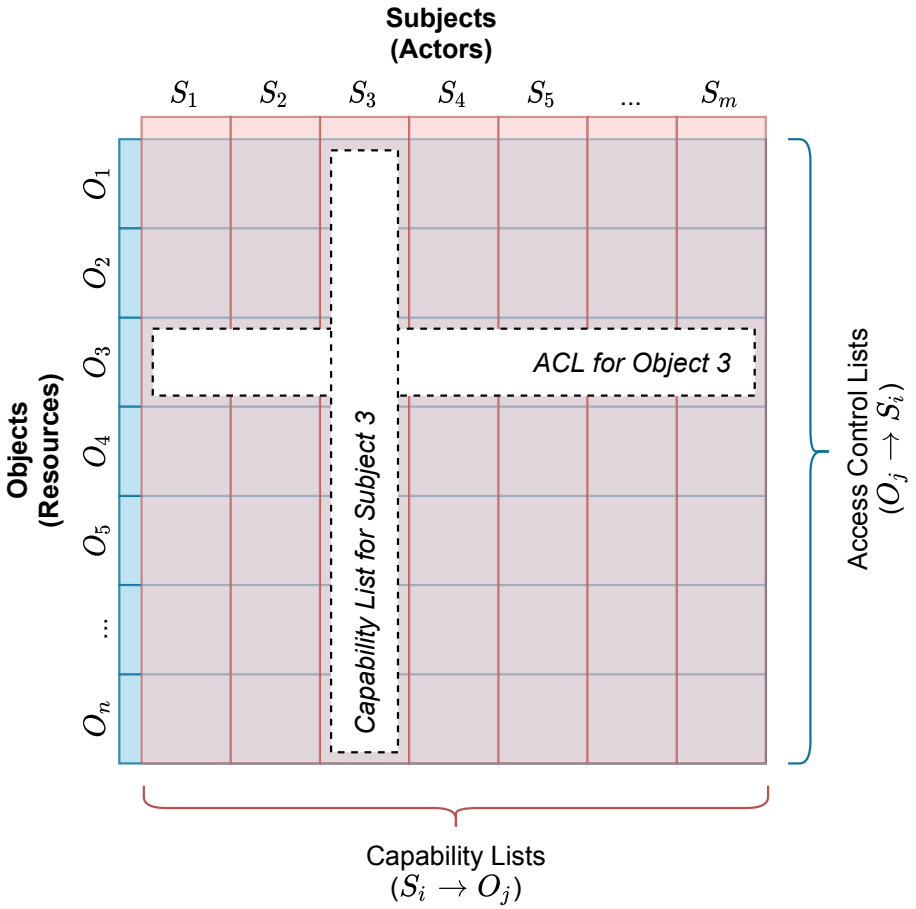

Figure 2.3: The access matrix and the relationship between ACLs and capability lists [2, 76, 107]. ACLs define the set of subjects that have specific access rights on a particular object. Capability lists conversely define the access rights that a specific subject has on a set of objects. Note that an ACL may be derived by taking the set of all capability lists and vice versa. Taken together, these form the access matrix.

107].

User and Group Assignment

To alleviate concerns with discretionary access control, systems often take the approach of assigning a unique user and/or group to a specific application. Such applications are typically security sensitive, such as a privileged daemon or network-facing service. This technique achieves a dual purpose: firstly, the application can lock down any resources it owns, simply by restricting any access to its own UID; secondly, the resulting process no longer needs to run under the same UID as its parent. This effectively limits the amount of outside resources that the application can access (so long as permission bits are correctly configured).

The Android operating system takes this model a step further, assigning a unique UID and GID to every application on the system, with optional UID sharing between applications that come from the same vendor. Under this model, no process' UID ever corresponds to a human user. While this arguably improves security, Barrera *et al.* [10] found weaknesses in Android's UID sharing model that can reduce its security to the trustworthiness of an app's signing key.

DAC Security Assumptions and Attacks

Although discretionary access control provides a convenient and intuitive user-centric model for object ownership and permissions, it makes some dangerous assumptions about security that can totally invalidate the model in practice [128]. In particular, DAC assumes that all processes are benign and contain no exploitable vulnerabilities. The mere existence off malware and exploitable vulnerabilities (e.g. memory safety

vulnerabilities) immediately invalidates this assumption. For instance, consider an honest but vulnerable piece of software running under a given UID X. An attacker exploiting a vulnerability in this application could perform arbitrary operations on any files owned by X. Similarly, a Trojan horse[1] [107, 128] can perform arbitrary malicious operations on X's files without needing to exploit any vulnerability. The fundamental issue with Unix DAC is that these files need not necessarily have *anything* to do with the program in question.

Another fundamental issue with Unix DAC lies in the ultimate authority of the root user. Any process running with EUID=0 is immediately part of the system's TCB (Trusted Computing Base)[2]. The same applies to any executable marked as setuid root. Processes that run with root privileges are prime targets for attacker exploit, since a successful attack can effectively compromise the entire system. For instance, confused deputy attacks [70, 128] can exploit privileged processes by tricking them into performing some undesired action. The coarse granularity of Unix DAC renders it particularly vulnerable against such attacks.

Proposals for Alternative Schemes

Both industry and academia have long recognized that weaknesses in the Unix discretionary access control model must be addressed. Many have turned to mandatory access control [42, 69, 71, 122, 123, 125, 129, 130, 137, 154] (c.f. Section 2.2.2) to solve the fundamental issues in DAC, while others have proposed improvements or

[1] A Trojan horse is a piece of ostensibly benign software that is designed to perform some malicious action or actions in addition to its ordinary functionality [107].

[2] The *trusted computing base* is the set of all hardware and software that must be trusted in order for the system to be considered trusted. Typically, this includes system hardware, the operating system itself, and a small subset of userspace programs [76].

alternative schemes for implementing discretionary access control [25, 47, 52, 96, 134, 146]. This subsection focuses specifically on the latter.

Mao *et al.* [96] proposed IFEDAC (Information Flow Enhanced DAC) as an alternative DAC model that is resistant to Trojan horse attacks. The insight behind their work was that DAC's primary weaknesses lie in the inability to distinguish requests involving multiple actors. Their mechanism proposes to track information flows between subjects and use these flows to infer a list of subjects that have influenced a request.

Under the traditional Unix DAC model, only the UID and GID of the process are considered when making access control decisions; under IFEDAC, the UID and GID of the owner of the underlying executable would also be considered, along with any other parties that may have influenced the state of the running process. This approach is similar in spirit to taint tracking mechanisms [90] (c.f. Section 2.2.4). To enable programs to function correctly, IFEDAC enables the user to define *exception policy* that specifies exceptions to IFEDAC enforcement. Mao *et al.* recommend that application authors and OS vendors should be responsible for distributing such policies [96].

Dranger, Solworth, and Sloan [52, 134] presented a three-layered model of DAC mechanisms. The *base layer* defines the general access control model, while the *parameterization layer* parameterizes it according to deployment needs. Finally, the *local initialization layer* comprises the set of subjects and objects along with their associated protections. The authors showed that their model was generalizable and that it could be used to implement any DAC mechanism.

Dittmer and Tripunitara [47] examined the implementation and common usage

patterns of the POSIX setuid and setgid API (Application Programming Interface) across multiple Unix-like operating systems. They identified weaknesses in systems that do not implement the latest POSIX standard revisions and suggested that mismatched semantics between various implementors can be a source of developer error. Finally, they presented an alternative API that partitions UID changes into permanent and temporary categories. Tsafrir *et al.* [146] and Chen *et al.* [25] identified the same fundamental issues and proposed the adoption of similar mechanisms.

2.2. Extensions to the Unix Security Model

Having examined the classical components of an operating system security framework, we now turn our attention to recent extensions on top of the Unix security model, with a particular emphasis on Linux and other free and open source Unix-like operating systems. This section presents a selection of key developments on top of the original Unix security model which have developed over time, with a particular emphasis on process-level confinement.

2.2.1. POSIX Capabilities

POSIX capabilities [24, 37, 38] are highly related to Unix DAC in the sense that they were originally designed to break up the multitude of privileges associated with the *root* user into more manageable pieces. In this sense, POSIX capabilities (when properly used) are more conducive to the principle of least-privilege. A process need not necessarily possess full root-level access to the system when only a small subset of those privileges are actually required.

Originally specified in the (now withdrawn) 1003.1e POSIX standard, POSIX capabilities were only ever (partially) implemented on Linux [3]. Other Unix-like operating systems prefer alternative methods of restricting privileges, many of which are discussed in Section 2.2.3. POSIX capabilities specify three *capability sets* for a given process: the **bounding set**, the **inheritable set**, and the **effective set**. The bounding set determines the set of all capabilities that a process is ever allowed to possess. The inheritable set determines the set of all capabilities that can be inherited across `execve` calls. Finally, the effective set determines the set of capabilities that a process can use (i.e. which capabilities a process currently possesses).

Linux exposes POSIX capabilities through extended filesystem attributes, much the same way that ACLs are implemented [38]. These file-based capabilities function in a similar manner to the setuid bit, implicitly setting the bounding, inheritable, and effective capability sets on execution. In addition so supporting capabilities as extended filesystem attributes, the kernel also supports dropping specific capabilities from each of the three sets through the `ptrctl(2)` system call. This enables a higher-privileged process (e.g. running as root) to drop elevated privileges while retaining those it needs to function. As of Linux 5.12, the kernel supports 41 capabilities in total, including the all-encompassing `CAP_SYS_ADMIN` [147].

It is worth mentioning that the term "POSIX capabilities" does *not* describe capabilities as they are broadly defined by operating system security researchers [3]. In particular, Dennis and Van Horn [46] first defined the notion of capabilities as a means of restricting access to *pointers*, guarding references to system objects. Unlike the capabilities defined by Dennis and Van Horn, POSIX capabilities are not associated with any given system object. Dennis and Van Horn's capabilities more closely

resemble that of the access matrix introduced by Anderson [2] and similar mechanisms have been implemented in other systems such as FreeBSD's Capsicum [152] and the CHERI architecture [43, 153]. These are discussed in more detail in Section 2.2.3.

2.2.2. Mandatory Access Control

In contrast with DAC, *MAC (Mandatory Access Control)* does not delegate permission assignment to the resource owner [76, 107, 137]. In the context of Unix, this means that MAC both overrides traditional discretionary access controls *and* applies access controls to all users on the system, including root. Historical implementations of MAC have focused primarily on MLS (Multi-Level Security), an access control scheme that revolves around the *secrecy* of objects and *access level* of subjects [12]. In a nutshell, MLS prevents a subject from reading data with a higher secrecy level or writing data with a lower secrecy level, preventing breaches in confidentiality and integrity [76].

Historical Approaches

Multics [36, 150] was the first operating system to pioneer the use of an MLS access control scheme. Memory in Multics was virtualized into *segments*, each with a *segment descriptor* that outlined protections that should be applied to that memory. To define an MLS policy, these protections included a secrecy level, enforced according to the subjects secrecy level. These MLS-style protections were complementary to discretionary ACLs defined for every segment along with memory protection rings

(the first of their kind), enforced in hardware [76].

While MLS is primarily applicable to military contexts, MAC has since evolved into mainstream use through the advent of alternative implementations. The Flask microkernel [137] introduced a practical architecture for MAC policy enforcement that was both scalable and effective. The non-discretionary components of the Flask security model were hugely influential in the design and implementation of subsequent MAC enforcement mechanisms, most notably SELinux [92, 130].

The basic notion behind flask is straightforward; the security architecture is divided into a security server, responsible for storing security policy, and a object manager that serves requests to userspace applications. When an application requests a resource, the object manager queries security policy from the security server and decides whether or not to serve the request based on the resulting enforcement decision. By designing Flask to be modular in this way, Spencer *et al.* achieved a separation of concerns between policy enforcement and policy decision-making, vital for scalability [92, 130, 137].

Linux Security Modules, SELinux, and AppArmor

The NSA first introduced SELinux (Security Enhanced Linux) [92, 130] as a Linux kernel patch, with the goal of providing an implementation of the Flask security architecture [137] for the Linux kernel. Reluctant to restrict users to just one security architecture, the kernel community eventually agreed that a generic security framework would provide more value than a single implementation, allowing for multiple upstream security implementations that could be selected based on the downstream use case. This effort culminated in the introduction of the *Linux Security Modules*

(LSM) [154] framework.

The LSM framework consists of a set of security hooks, placed in strategic locations throughout the kernel [154]. Hooks can roughly be divided into *enforcement* hooks and *bookkeeping* hooks. Enforcement hooks serve as checkpoints for security enforcement over specific access categories, while bookkeeping hooks enable a security module to maintain stateful information about subjects and objects on the system. LSM hooks are not considered to be static, and often change between kernel versions as new hooks are implemented and both new and existing hooks placed into various kernel functions [159]. The eventual goal of the LSM framework is to provide complete mediation over kernel security events, however this is an evolving process and no formal verification exists to prove the security of LSM hooks [61]. Figure 2.4 depicts the basic LSM architecture.

After the introduction of the LSM framework, SELinux was refactored into a Linux security module [130] and subsequently merged into the mainline kernel. SELinux supports three major types of mandatory access control: (i) role-based access controls; (ii) type enforcement; and (iii) an optional MLS policy. Fundamentally, SELinux policies are based on a notion of subject and object labelling. A security policy assigns a specific label to subjects and objects, and then specifies access patterns between these labels.

In an effort to simplify the SELinux policy language, a *reference policy* was introduced by PeBenito in 2006 [110], providing a framework for creating and managing reusable policy templates which can be integrated into new and existing security policies. SELinux reference policies can be augmented with boolean options, called *tunables* that provide coarse-grained control over policy behaviour to system admin-

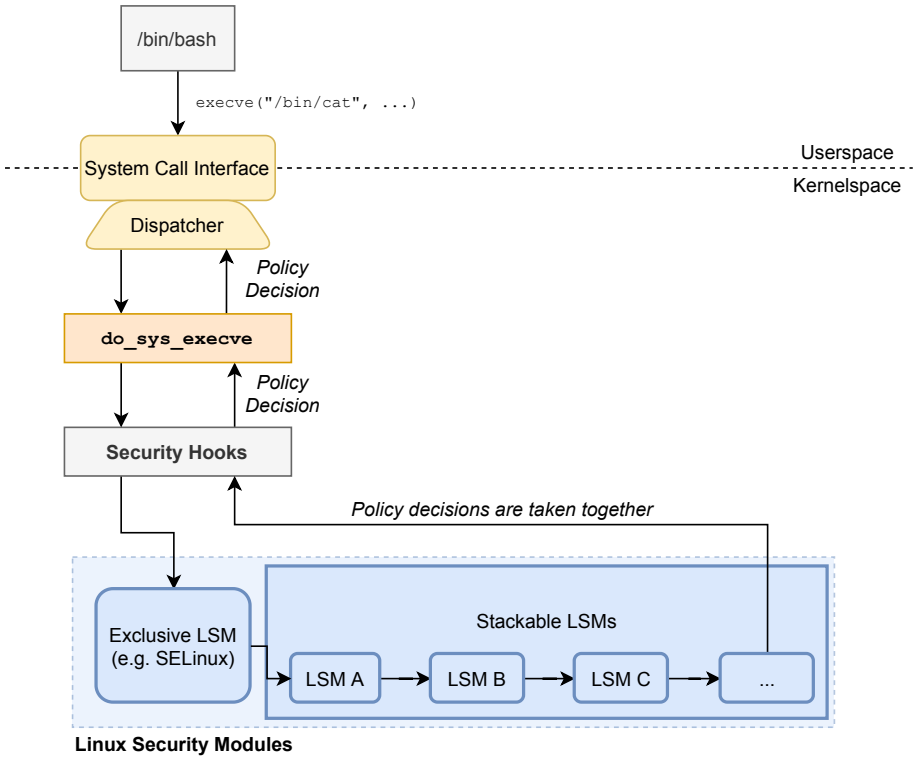

Figure 2.4: The LSM architecture. A single *exclusive* LSM may be loaded at a time, and complemented by zero or more *stackable* LSMs. When userspace requests a privileged operation (e.g. through a system call), this operation causes one or more security hooks to be invoked. These hooks in turn call into the respective hook implementations provided by each loaded LSM. Each hook returns a policy decision, and these decisions are then taken together to arrive at a final decision.

istrators. Sniffen [133] implemented a guided policy generation system that walks application authors through the process of writing SELinux policy. This framework was further augmented by MacMillan [95], culminating in the eventual introduction of the `audit2allow` [131] command line utility for automated policy generation. Despite these usability improvements, the SELinux policy language is generally considered to be quite arcane [125], rendering it difficult for non-expert users to write and audit security policy.

Since the introduction of SELinux, several alternative LSMs have been proposed, many of which have subsequently been merged into the mainline Linux kernel. AppArmor (originally called SubDomain) [42] takes an alternative approach to SELinux, enforcing security policy based on *pathnames* rather than security labels. AppArmor policies, called *profiles*, are assigned on a per-executable basis. Rather than being labelled, AppArmor policies identify system objects directly (e.g. through pathnames, IPC (Inter-Process Communication) categories, or network IP (Internet Protocol) addresses). Each system object is associated with a particular access pattern, which determines the privileges for a given AppArmor profile. Similar to SELinux, AppArmor offers a suite of userspace tooling for automated and semi-automated policy generation based on enforcement logs [4, 5, 6].

Alternative Linux Security Modules

Recognizing usability issues in the complexity of SELinux and AppArmor policies, Schaufler [123] introduced SMACK (Simplified Mandatory Access Control Kernel) to offer a simplified label-based enforcement scheme that focuses on expressing a minimal set of permissions. Like SELinux, SMACK relies on labelling filesystem objects

using extended filesystem attributes. Policies then attach simple access specifiers to these labels, based on canonical Unix permissions like `read`, `write`, and `execute`. Some default labels are provided, which grant or revoke blanket permissions for common tasks such as system daemons or the `init` process.

Schreuders *et al.* [125] designed FBAC-LSM (Functionality-Based Access Control LSM) with a similar goal of simplifying policy definition. Unlike SMACK, which is based on labelling, FBAC-LSM specifies policy in terms of desired *functionality*. Specifically, FBAC-LSM policies define a set of high-level functionalities that an application should exhibit. All other functionalities are prohibited. Schreuders *et al.* evaluated FBAC-LSM in terms of its usability and found that it compares favourably against AppArmor and SELinux [125].

Hu *et al.* [71] proposed FSF (File System Firewall), which applies firewall-like semantics to filesystem objects. Their goal was to create a more usable, file-specific access control mechanism. FSF policies are defined in policy files, comprised of simple rules that specify a file along with some combination of `read`, `write`, and `execute` access. *Redirection rules* are primary differentiating factor between FSF and conventional file-based access controls. Taking inspiration from similar functionality in network firewalls, a redirection rule can be used to convert one file access into another, transparently to the target application. Hu *et al.* conducted a user study comparing their prototype to Unix DAC and SELinux and found that FSF performed favourably in policy comprehensibility and accuracy.

The TOMOYO [69] LSM takes an alternative approach, emphasizing policy generation and building generation functionality into the LSM directly. Rather than involving userspace helpers, TOMOYO generates policy by inferring per-application

profiles through the LSM hooks they invoke. Programs can additionally discard their privileges through a TOMOYO-specific system call, similar to the notion of dropping privileges under the POSIX capabilities model. Like AppArmor, TOMOYO relies on pathnames to identify resources rather than assigning security labels. Users have the option to edit generated policies as required, but the intent is to require as little user involvement as possible [69].

The Landlock [121, 122] LSM was recently introduced into the mainline Linux kernel as a contemporary alternative to `seccomp(2)` (covered in Section 2.2.3). Landlock was originally intended to allow unprivileged userspace processes to load highly restricted eBPF programs into the kernel to define security filtering logic [121]. However, due to concerns related to the security of unprivileged eBPF, this functionality was later reworked into a set of simple access rules and no longer has any association with eBPF [122]. Under Landlock, a process creates a *ruleset* using the `landlock_create_ruleset(2)` system call, adds rules to that ruleset, then confines itself by committing to that ruleset. Developers may add this confinement logic directly into their software or may write specialized userspace wrappers to apply generic confinement to applications.

Singh [129] introduced the KRSI (Kernel Runtime Security Instrumentation) framework for attaching eBPF programs to LSM hooks with the goal of defining dynamic audit and policy enforcement filters. While similar in spirit to the original Landlock proposal [121], KRSI differs fundamentally in that it remains a *privileged* LSM; only root-privileged processes may load eBPF LSM programs into the kernel. BPFContain and BPFBox are both based on the KRSI framework. Section 2.5.2 examines the KRSI framework in more detail.

2.2.3. System Call Filtering and Capabilities

Since system calls define the canonical interface for communication between userspace processes and the operating system kernel [76], they are a natural fit for defining the protection interface of a confinement mechanism. In particular, *system call filtering* is a widely-used technique for application sandboxing and self-confinement [3]. Indeed, LSMs, covered in the previous section, can be thought of as a form of system call filtering, although security hooks are placed manually within system call implementations and do not necessarily conform to the same semantics as the underlying system calls [154].

Related to this notion of system call filtering are *capabilities*[3], which guard access to a particular reference, associating privileges with a handle to a given object on the system. For instance, the `open(2)` system call returns a *file descriptor*, which constitutes a reference to a particular filesystem object. A capability associated with this file descriptor would then allow specific operations on the file descriptor, applying a default-deny policy to all other operations.

System Call Tracing

In Unix, ptrace [108, 117] (short for process trace) is a mechanism provided by the kernel that allows one process (the tracer) to attach itself to another (the tracee), tracing and possibly manipulating nearly any aspect of its execution, system calls, memory, and registers. Originally designed as a debugging interface [108, 117], ptrace has also been applied to implement system call filtering [64, 77, 151]. However, ptrace

[3]Here, the term "capabilities" is a disparate term from "POSIX capabilities," covered in Section 2.1.3.

has fallen out of favour, particularly in production use cases, due to its immensely high overhead (on the order of several thousand percent [161]) and propensity to introduce undefined behaviour when tracing even moderately complex software [142]. Due to its invasive nature, a tracer process must either be the direct parent of a tracee or must have sufficient privileges to trace the child process — on Linux, this translates to either the `CAP_SYS_PTRACE` capability or the all-encompassing `CAP_SYS_ADMIN`.

Janus [64, 151] was an early exploration of how `ptrace` could be applied to confine applications by filtering system calls. The original Janus prototype was designed for Oracle Solaris using its ptrace interface, exposed through the `procfs` virtual filesystem [64]. A subsequent port of Janus was released for Linux [151], although it required invasive modifications to Linux's ptrace implementation, dubbed ptrace++ by the authors. Janus worked by attaching to the target process using ptrace, then tracing system calls made by the target process and categorizing them into groups based on functionality. A Janus policy could allow or deny specific categories of system call, confining the application in a coarse-grained manner. The tracer process, called the *supervisor process*, would then be able to kill the offending process or inject failure into the offending system call when it detected a policy violation. Jain and Sekar [77] implemented a similar system call monitor, adding the ability to modify system call arguments and using a different policy language design.

Provos' Systrace [115] uses ptrace to analyze per-process system calls and generate a system-call-level policy. Unlike Janus [64, 151] and Jain *et al.* [77], Systrace supports intrusion detection, policy generation, and audit logging, providing a mechanism to automatically analyze process behaviour. Systrace also supports one highly unconventional feature, which Provos calls *privilege elevation*. The notion be-

hind privilege elevation is to allow a program to escalate its privileges selectively for specific system call access patterns, preventing the need for coarse-grained privilege escalation such as setuid root.

Somayaji and Forrest [136] implement an intrusion detection system, pH (process Homeostasis), based on system call sequences, although it does not rely on ptrace and analyzes system call sequences instead of individual call patterns. Rather than as a confinement solution, pH was strictly designed as a behavioural anomaly detection system, although this approaches confinement as profile accuracy improves. Findlay [57] (the author of this thesis) later ported pH to use eBPF to analyze system call sequences.

OpenBSD Pledge and Unveil

OpenBSD's `pledge(2)` and `unveil(2)` system calls form the backbone of its built-in sandboxing framework. A pledge [112] consists of a list of *promises*, high-level descriptions of what behaviours a program expects to exhibit in the future, similar in spirit to Janus' high-level categories [64, 151]. The pledge system call takes two space-separated lists of promises, one to be applied immediately and another to be applied upon making an `execve(2)` call. To prevent privilege escalation, subsequent calls to `pledge(2)` take the union of existing promises and new promises, precluding a process from escaping its initial bounding set [112].

Promises vary in granularity; the most coarse-grained promise, `stdio`, allows a total of 69 distinct system calls, enabling the full suite of C standard library stdio-family calls. Others, like `chown`, are more conservative, enabling only one (albeit in this case very powerful) system call. In total, `pledge(2)` includes 33 distinct

promises, as of OpenBSD 6.9 [112]. Due to its coarse granularity and lack of concern for specific system objects, pledge has been criticized as being overly-permissive [3].

Unlike pledge, `unveil(2)` [39, 148] operates on specific filesystem paths, making a promise about the kinds of operations the process will perform on file descriptors associated with these paths. Specifically, unveil is concerned with four kinds of permissions: read, write, execute, and create/delete. Unveiling a directory unveils all files and directories underneath, recursively. Although this approach is finer-grained than pledge, it is file-specific and offers a trade-off between granularity and usability. The official manual page for unveil [148] recommends that developers use it at the granularity of directories, despite the fact that this may result in overpermission in practice. For instance, consider a hard link to the root of the filesystem placed by an attacker within some unveiled directory. The unveiling process would now have full access to the entire filesystem, constituting a sandbox escape.

Linux Seccomp and Seccomp-bpf

In Linux, the primary facility for direct system call filtering is `seccomp(2)` [3, 54, 116, 126]. Unlike OpenBSD's pledge and unveil [112, 148], seccomp filters directly over system calls, without any blanket categorization. Initially, seccomp was highly limited, restricting a process to only four system calls: `read(2)` and `write(2)` for reading and writing open file descriptors, `sigreturn(2)` for handling signals, and `exit(2)` to enable self-termination. Using any other system call would result in an immediate SIGKILL delivered from the kernel, forcefully ending the offending process.

Later, seccomp was extended to enable processes to define custom allowlists[4], denylists, and enforcement actions using classic BPF (Berkeley Packet Filter) filters [54]. This new incarnation was dubbed seccomp-bpf. While allowing for much finer-grained confinement policy than pledge and unveil, seccomp-bpf has its own limitations which can result in ineffective (and possibly dangerous) policies. In seccomp-bpf, filters are defined over system call numbers and (optionally) arguments. Unless the developer takes great care to correlate system call numbers with the specific target architecture, the resulting policy may allow and deny incorrect system calls, resulting in broken policies that break applications in the best case and expose security vulnerabilities in the worst case.

Another innate problem with seccomp arises due to its fine granularity. Paradoxically, avoiding system call categorization can expose vulnerabilities, due to system call equivalence classes. For instance, the `openat(2)` system call can perform the same functionality as the `open(2)` system call, with slightly different API semantics. A seccomp-bpf filter allowing one system call but denying the other is now totally broken and vulnerable to sandbox escape. Similarly, argument checking on pathnames or file descriptors can be vulnerable to TOCTTOU (Time of Check to Time of Use) race conditions in practice, rendering such policies ineffective [3].

A final consideration for seccomp-bpf is that the development of seccomp-bpf policies requires knowledge of the relatively arcane cBPF (Classic BPF) syntax. This problem is somewhat alleviated by the existence of library wrappers [87] around seccomp-bpf functionality, although the usability of these solutions remains some-

[4] Allowlist and denylist are the new politically correct terms (in addition to being more semantically meaningful) for whitelist and blacklist respectively.

what questionable, particularly given the many pitfalls of seccomp-bpf policy authorship [3].

FreeBSD Capsicum and CHERI Capabilities

Unlike the system call filters presented earlier in this section, FreeBSD's `capsicum(2)` [3, 152] is a true implementation of capabilities as they were originally described by Dennis and Van Horn [46]. Specifically, capsicum capabilities are an extension on top of Unix file descriptors, the canonical reference to files and file-like objects such as network sockets and character devices. Capsicum adds an unforgeable access token to each file descriptor, granting the corresponding process specific access rights over that file descriptor. Whereas alternatives like seccomp-bpf [126] and pledge [112] restrict access at the system-call-level, Capsicum restricts access at the resource-level and enforces this access within the system call layer.

To confine processes, Capsicum exposes a special `cap_enter(2)` system call which causes a process to enter *capability mode*. A process in capability mode no longer has access to global namespaces (e.g. the PID (Process ID) namespace) and may only make a subset of system calls which do not directly access these global namespaces. Other system calls are constrained so that they may only operate under the context of an open capability descriptor (a file descriptor which has been extended with capability information) [152]. The end-result is an expressive and fine-grained self-confinement framework for FreeBSD applications, which comes at a small usability cost compared with coarser-grained alternatives like `pledge(2)` [112].

In 2015, Watson *et al.* designed CHERI [153] as an extension to the MIPS ISA enabling the capability-based protection of memory pages. Under CHERI, the op-

erating system kernel and userspace runtime extend the traditional memory model with capabilities using the CHERI ISA. This enables generic capability-based protection at the level of memory pages. While powerful, this extension requires dedicated hardware support. Watson *et al.* implemented their prototype on an FPGA (Field-Programmable Gate Array) [153] and later extended the FreeBSD ABI (Application Binary Interface) to work with CHERI capabilities [43].

2.2.4. Taint Tracking

Taint tracking [90] describes the notion of tracking changes to memory containing application data as it is mutated, copied, and moved by the underlying application. Such data is considered *tainted* when it is modified by some external source in such a way as the data can no longer be trusted. For instance, a buffer might be populated by an external network connection or local user input. The security benefits of such a mechanism are obvious. An active attack requires some user input into a program in order to exploit a vulnerability; by tracking untrusted user input and treating it as untrusted, developers can avoid attacker exploitation of sensitive code paths. Beginning with Perl's *taint mode* [72], taint tracking has enjoyed a rich body of literature [13, 26, 27, 30, 33, 55, 90, 156, 157, 160] since its inception.

In Perl's taint mode [72], a special command line flag triggers the interpreter to flag untrusted user input and prevent it from being passed as input to functions explicitly marked as *unsafe*. To circumvent this restriction, a developer could perform a predetermined set of sanity checks on the data to *untaint* it. Rather than acting as an outright security mechanism, the goal was to encourage developers to take care

in processing untrusted data. Incorrect or insufficient sanity checks on the data or running the Perl interpreter without the taint flag would result in no additional security benefits whatsoever. After Perl, similar taint tracking mechanisms have been added to other interpreters and language runtimes, including Ruby, PHP, and Python [33].

Conti, Bello, and Russo [13, 33] implemented more advanced taint tracking functionality for the Python programming language as a library that developers could use directly. Their argument was that implementing such a taint mechanism at the language-level rather than at the interpreter-level could enrich the traditional taint-tracking approach with use-case-specific metadata and facilitate extensions to support complex data types.

With the goal of creating a generic and reusable taint tracking mechanism, several researchers have proposed application-transparent taint tracking. Many have turned to virtualization or emulation runtimes [55, 156, 157] such as QEMU, KVM, or Xen, using built-in introspection features to track the propagation of data within (and even between) running processes. Others have proposed the adoption of static analysis or library instrumentation [26, 30, 160] techniques to reduce overhead and eliminate the need to run applications under expensive virtualization monitors. Others have built taint tracking logic into existing language runtimes, such as the Java Virtual Machine [27].

2.3. Process-Level Virtualization

We now step away from *confinement* to focus on process-level *virtualization* primitives employed in Unix-like operating systems. Whereas confinement primitives have the goal of restricting a process' behaviour, virtualization primitives instead limit the process' ability to see the world around it. This property should not be confused with *isolation*. True isolation requires a mixture of both virtualization and confinement mechanisms to restrict access to system resources and prevent unwanted behaviour.

Chroots and Chroot Jails

To virtualize the filesystem, Unix has classically supported the `chroot(2)` system call [99], used to change the filesystem root ("`/`") to some directory, specified as an argument. From the process' point of view, this directory becomes its new filesystem root. However, chroot suffers from several issues that render it totally ineffective as a security mechanism. Chroot escapes, path traversals, spurious access to special filesystems and devices, and superuser privileges all totally invalidate chroot as an isolation mechanism [99].

For instance, consider a call to `chroot("/my/new/root")`. Without a follow-up call to `chdir(2)` to change the process' current working directory, a simple call to `chdir("..")` is enough to escape the chroot jail. Even with the aforementioned precautions, a process that has or is able to obtain superuser privileges can simply create a new directory, re-invoke chroot, and perform the same escape as before [99]. Without the proper confinement mechanisms and necessary precautions to prevent such escapes, chroot cannot be considered an effective isolation technique. In fact,

chroot escapes have been a source of many vulnerabilities with container management engines like Docker in the past [32]. McFearin [99] proposed updates to the POSIX standard that fix many of chroot's security flaws, but these have not been adopted.

FreeBSD Jails and Solaris Zones

Kamp *et al.* [79] presented FreeBSD's `jail(2)` as a more secure alternative to `chroot(2)` jails. In particular, the jail system call is a heavily extended wrapper around FreeBSD's chroot implementation. A call to jail begins by allocating and populating a `prison` data structure that maintains metadata related to the jailed process group, and finishes by simply invoking the standard chroot implementation. Unlike chroot, Jails take care to avoid the standard pitfalls that may result in a chroot escape and heavily limit the privileges of the root user within the jail. In this respect, the jail system call can be seen as a hybrid between a virtualization and confinement mechanism, approaching a full solution.

Jails take the approach of defining a clear security boundary around a collection of processes, filesystem resources, and network resources [79]. A process existing within this boundary (i.e. a member of the jail) enjoys the standard set of Unix permissions on resources within the jail. Access to resources outside of the jail is forbidden, including access by the root user. Visibility is similarly restricted by remapping namespaces in such a way as outside resources are effectively invisible. Defining a clear protection boundary enables the jail to enforce sensible security policy without burdening the administrator with the details of writing such a policy [79].

Solaris Zones [114] later arose as a commercial solution for process-level virtualization. The goal was to implement namespace remapping and security isolation for

commercial server deployments in (possibly multi-tenant) Solaris environments. The implementation details of Solaris' Zones are similar to those of FreeBSD's Jails; a per-task data structure manages the associate between tasks and Zones, allowing the kernel to perform the necessary remapping and security checks.

Linux Namespaces and Cgroups

Unlike FreeBSD [79] and Solaris [114], Linux takes a different approach to process-level virtualization. In particular, Linux's virtualization strategy consists of two separate mechanisms, *namespaces* and *cgroups* (short for process control groups). These mechanisms, in turn, can be further subdivided into specific types, targeting different kinds of system resources. Notably, namespaces and cgroups are *not* confinement mechanisms in and of themselves. This property is in stark contrast with Jails and Zones, which were designed to offer strong confinement guarantees over their respective security boundaries. As the canonical process-level virtualization building blocks in Linux, namespaces and cgroups form the backbone of Linux containers (c.f. Section 2.4.1).

Linux namespaces [14, 104] limit the visibility of system resource by providing a virtual remapping of global resource identifiers to a process or process group. Such identifiers include process IDs, user IDs, filesystem mounts, IPC objects, and network interfaces, among others. Linux supports namespace creation and entry via the `clone(2)` and `unshare(2)` system calls, for isolating child processes and existing processes respectively. As of version 5.13, the Linux kernel supports eight distinct namespaces, depicted in Table 2.1. Other namespaces have been proposed and/or planned for inclusion, including a security namespace [141] for virtualizing

Table 2.1: Linux namespaces (as of kernel version 5.13) and what they can be used to isolate [104].

Namespace	Isolates
PID	PIDs
Mount	Filesystem mountpoints
Net	Networking stack
UTS	Host and domain names
IPC	Inter-process communication objects
User	UIDs and GIDs
Time	System time
Cgroup	Cgroup membership

LSM hooks.

Complementary to namespaces, cgroups [34, 62] divide processes into hierarchical groups, performing resource accounting and restricting access to quantifiable resources such as memory, CPU clock cycles, block I/O, and device drivers. From a security perspective, such restrictions are useful to prevent resource starvation attacks against the host. However, Gao *et al.* [62] found that this protection is incomplete and often misleading, allowing up to 200× the allotted resource limits by exploiting out-of-band resource consumption techniques. To interact with the cgroup hierarchy, Linux exposes a virtual filesystem which encodes cgroup membership in its directory structure. Cgroup membership visibility can be virtualized using the cgroup namespace [34].

2.4. Containers and Virtual Machines

Classically, virtualization of a system has been accomplished by means of a hypervisor [53, 140]. Hypervisors implement an interface which overlays the underlying system hardware, enabling one or more *guest operating systems* to be installed on a single physical host. These guest systems are then called *virtual machines*. Due to the level of indirection provided by the hypervisor and the guest operating system kernel, virtual machines are generally considered to be quite *strongly isolated* from each other, but this isolation comes at the cost of much higher overhead, both in terms of storage and performance [53, 140]. Section 3.1 in Chapter 3 discusses the differences between virtual machines and containers in more detail and addresses some common misconceptions about the levels of isolation they provide.

Containers have emerged as a new unit of computation in recent years, providing a more lightweight alternative to full hypervisor-based virtualization [53, 140]. Unlike virtual machines, containers run directly on the host operating system, rather than directly atop a hypervisor, and share the host operating system kernel with each other and with ordinary host processes. In fact, a container is nothing more than a discrete collection of processes (sometimes only one process) that share some common isolation from the rest of the system by way of virtualization and confinement primitives. These primitives typically include namespaces and cgroups, along with optional confinement mechanisms like seccomp-BPF and Linux MAC policy. Since they directly share the host OS kernel and do not require a guest operating system, containers are significantly more lightweight and more performant than virtual machines, but at the cost of weaker base isolation. Figure 2.5 depicts the major

differences between container and hypervisor-based architectures.

2.4.1. Container Security

Due to the nature of containers as specialized process groups, any notion of container security is necessarily tightly coupled with the underlying security primitives exposed by the host operating system. These include the virtualization primitives discussed in Section 2.3, and the confinement primitives discussed in Sections 2.1 and 2.2. Whereas the primary attack surface of a virtual machine is comprised of the interface exposed on top of hardware, the attack surface of a container is comprised of the system calls that it uses to interact with the host operating system kernel. Every system call is an opportunity to exploit a kernel vulnerability; over-zealous resource-sharing is an opportunity to establish a covert channel, or perform a confused deputy attack [70] against a privileged application. The cost to mount such attacks rapidly decreases as isolation from the host operating system decreases.

Figure 2.6 depicts a high-level threat model for containers from the perspective of the host operating system. This model comprises a number of attack vectors, each targeting a different part of the system. For instance, a malicious container process can target the host kernel directly, providing input to a kernel function through a system call (Item A in Figure 2.6). This input can be crafted to trigger a specific vulnerability in the host kernel, resulting in privilege escalation, information disclosure, or denial of service.

A malicious container process can also target a process belonging to another container (Item B in Figure 2.6) or a (possibly privileged) host process (Item C in

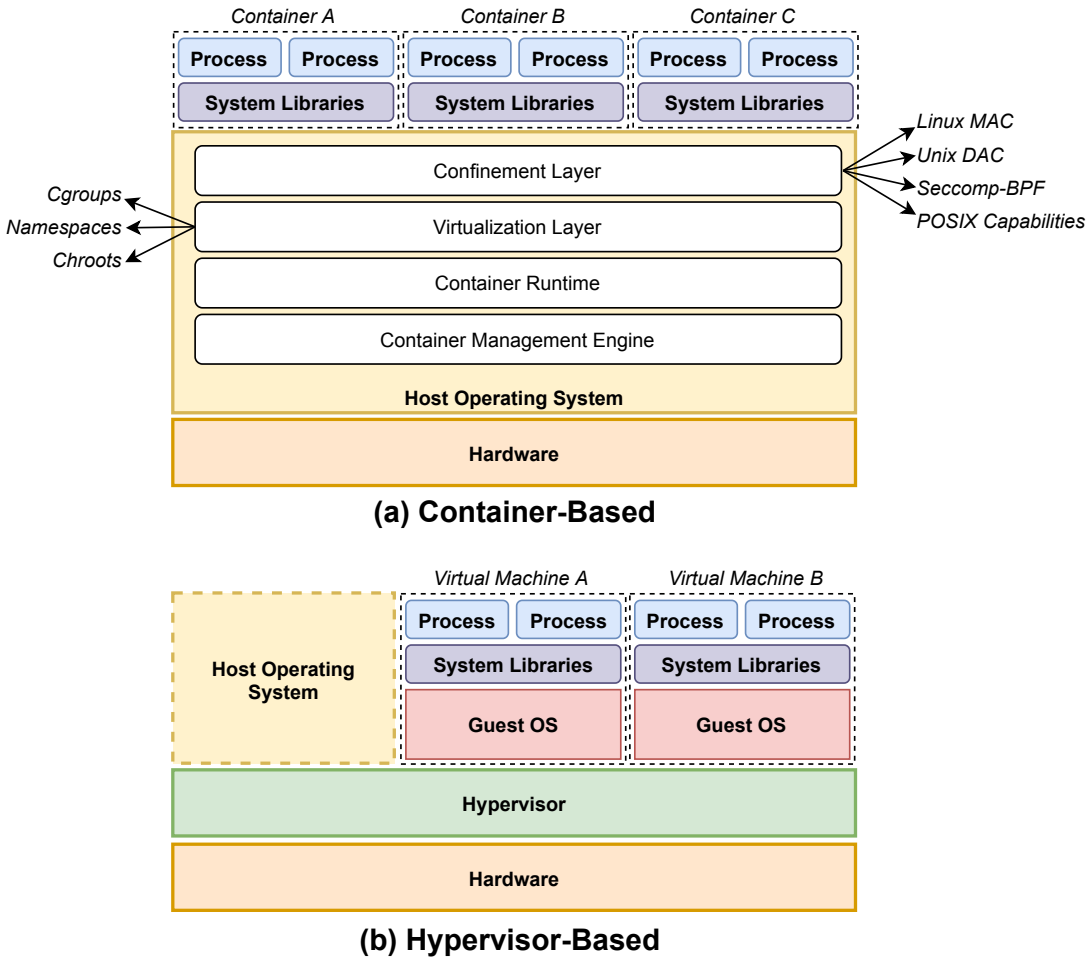

Figure 2.5: A comparison of virtual machine and container architectures [53, 140]. Containers **(a)** achieve virtualization using a thin layer provided by the host OS itself. They share the underlying operating system kernel and resources, requiring no guest OS. A hypervisor **(b)** virtualizes and controls the underlying hardware directly, but requires full guest operating systems on top of the virtualization layer.

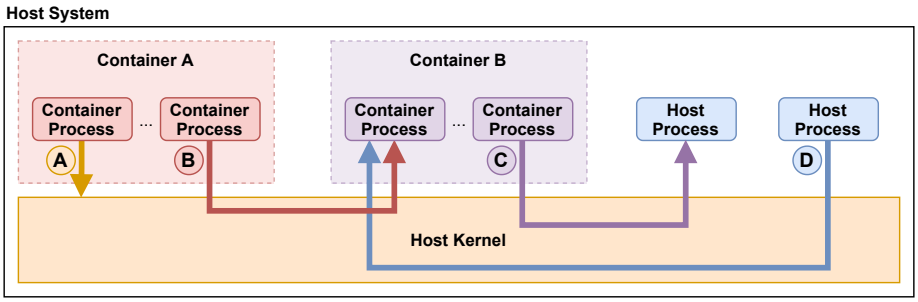

Figure 2.6: A sample threat model for container security. **(A)** A container process attacks the host kernel directly. **(B)** A container process attacks another container process, passing through the host kernel's reference monitor. **(C)** A container process attacks a (privileged) host process, passing through the host kernel's reference monitor. **(D)** A host process attacks a container, passing through the host kernel's reference monitor.

Figure 2.6). An attack could occur as a result of a misconfigured security policy or some system resource shared by both processes. The end goal might be tampering with the other process, or privilege escalation (for instance, via a Confused Deputy attack [70]). Finally, a host process can attack a container process (Item D in Figure 2.6). Goals in this case might include tampering with a container belonging to another user of a multi-tenant system.

Not pictured in Figure 2.6 but important nonetheless is the case where a process (containerized or otherwise) abuses a vulnerability in system hardware or firmware (e.g. a side channel attack on cache memory) to leak information about another process (containerized or otherwise). Such attacks can totally bypass the operating system's reference monitor and generally require trusted computing technologies or hardware/software co-design to mitigate. As such, they are out of scope for this thesis. Resource starvation attacks are also considered out of scope.

Container Security in Industry

Existing container security on Linux is supported by a number of fundamental virtualization and confinement mechanisms, covered in Section 2.1 and Section 2.2. Namespaces virtualize and limit the visibility of global resource identifiers and cgroups virtualize and limit the available quantities of system resources. Dropped POSIX capabilities allow the container to partition coarse-grained root privileges into finer components. Seccomp-bpf filters limit the availability of system calls to the container, while Linux MAC policies, enforced by an LSM restrict the container's access to system resources. Unix DAC, possibly accompanied by a new user namespace, can limit the container's access as well, when properly configured.

LXC [94] is a container runtime for Linux that directly exposes low-level virtualization and confinement primitives. LXC exposes namespaces and cgroups to virtualize system resources and seccomp-bpf to filter system calls, reducing the kernel's attack surface.

Docker [22, 32, 50], originally based on LXC, provides a high-level interface for creating, manipulating, and running container images. Docker places containers in a new cgroup with sensible defaults for resource virtualization. To isolate processes, the filesytem, and network interfaces, Docker containers run in a new PID, IPC, mount, and net namespace by default. Similarly, Docker uses the UTS (Unix Timesharing System) namespace for hostname virtualization and the cgroup namespace to limit cgroup membership visibility. To reduce the kernel's attack surface, Docker also includes a default seccomp-bpf profile that blocks 51 system calls. Finally, Docker supports integration with the AppArmor LSM when it is enabled on the host, using

a generic default profile that provides modest protection [49, 51].

Unfortunately, Docker does not run in a new user namespace by default, making privilege escalation from within a container significantly more likely [50]. However, it does offer the ability to opt-in to user namespace confinement with some additional setup. Docker also provides a `--privileged` flag which allows a user to totally ignore all security defaults, essentially granting the container the same access as a (often root-privileged) host process.

Container Security in Academia

Sultan *et al.* [140] published a comprehensive review of container security, including a threat model and comparison with full-virtualization solutions like type I and type II hypervisors. Their work outlined four distinct cases and presented a survey of existing security mechanisms targeting each case: (i) a containerized attacking the container; (ii) a container attacking other containers; (iii) a container attacking the host; and (iv) the host attacking a container. Their recommendations included an increased adoption of trusted computing technologies to solve case iv and that work towards a container-specific LSM would be necessary to harden against cases i–iii. BPFContain, one of the two research systems presented in this thesis, represents a step towards such a container-specific LSM.

Lin *et al.* [88] presented a measurement study on container security measures and attacks. They hand-crafted an exploit data set consisting of 233 exploits and used it to test the security defaults employed by Docker. Their findings indicated that inter-dependence and mutual-influence among several disparate kernel security mechanisms resulted in weaknesses in protection. Motivated by their findings, they

developed a simple kernel patch hardening the `commit_creds()` function against simple privilege escalation attacks mounted from containers.

Combe *et al.* [32] and Bui [22] presented informal security analyses of Docker's default security configurations and Docker security in general. Combe *et al.* [32] found that Docker configurations are weak to supply-chain attacks involving malicious images and configurations on Docker Hub. Additionally, they found that, while Docker's default configuration is relatively secure, container misconfiguration, or the absence of security mechanisms such as AppArmor on the host leaves the host vulnerable to attack. They also found that the default mandatory access control policies employed by Docker were overly-permissive and far too generalized to provide practical protection.

Bui [22] found that, while Docker does offer inadequate protection against many sophisticated attacks, it still yielded security benefits over running applications natively on the host. Bui recommends that containers be run under virtual machines to add an additional layer of isolation from the host system. These findings, however, demonstrate a lax attitude toward container security, opting to rely on additional layers of indirection to provide real security guarantees and positing that at least some protection is better than none at all. Eder [53] compared hypervisor- and container-based virtualization and found that hypervisors are naturally more secure due to increased levels of independence and isolation.

Babar *et al.* [9], and Mullinix *et al.* [101] studied the container security mechanisms underlying the Linux container infrastructure. Their findings separately indicate that existing security mechanisms provided by the kernel are insufficient to offer full protection from container vulnerabilities, particularly given the unique nature of

the attack surface exposed by the container running directly on the host operating system.

To address limitations imposed by container security defaults and alleviate concerns about poor security practices in default configurations, many researchers have turned to automatic policy generation [63, 85, 93]. Dockersec [93] uses a combination of static analysis techniques on existing security profiles and a dynamic training process to automatically infer AppArmor profiles for containers. These inferred profiles provide greater protection than the generic default profile since they are finer-grained and tailored to the container's access patterns. In addition to generating MAC policy, others have focused on generating seccomp-bpf policy to reduce the kernel's attack surface from within a container. Confine [63] uses static binary analysis and library call instrumentation to generate seccomp-bpf policy for container images. Their results showed that they were able to significantly reduce the attack surface for kernel exploitation in many of the most popular Docker images. SPEAKER [85] partitions application containers into two distinct phases — the setup and execution phase — and generates a unique seccomp-bpf policy for each phase, enabling a tighter bound on confinement for each phase.

Others have focused on promoting self-confinement for containerized applications. Sun *et al.* [141] proposed the inclusion of a security namespace into the kernel, allowing individual containers to load their own independent MAC policy. This approach enables a clear separation of concerns between host policy and container policy, and provides a clear path toward unprivileged self-confinement. The approach is also generic enough to enable the use of alternative LSM-based confinement solutions on a per-container basis. BPFContain, for instance, might work cooperatively with

security namespaces for more efficient per-container confinement.

Vulnerability analysis of container images [19, 81, 127] can be an effective technique for identifying weaknesses in container deployments. Unlike policy generation, vulnerability analysis is a strictly informative tool, allowing security experts to identify weaknesses in production deployments and fix them. Shu *et al.* [127] presented DIVA, a framework for analyzing vulnerabilities in images deployed from Docker Hub. They aggregated data from over 350,000 container images and found that images contained an average of 180 security vulnerabilities. Kwon and Lee [81] proposed DIVDS, which extends prior work by providing an interface to compare and allow specific image vulnerabilities. Brady *et al.* [19] applied similar vulnerability scanning techniques to a continuous integration pipeline, flagging and fixing image vulnerabilities during development.

eBPF is seeing increasing prominence within the container security space. Besides BPFBox (c.f. Chapter 4) and BPFContain (c.f. Chapter 5), other projects have arisen over the past few years, albeit with a general focus on observability rather than policy enforcement. Tracee [8] is a container observability tool developed by Aqua Security that can watch system calls made by a container, along with other security-sensitive events, and generate audit logs for further analysis. Cilium [28] is a popular security daemon for the Kubernetes container orchestration framework, with a focus on network security for distributed container deployments. Cilium provides observability metrics through a configurable audit framework and allows the end user to define network policy for telemetry, performance optimization, and security.

2.5. Extended BPF

eBPF stands for "Extended BPF", though in reality it has very little to do with Berkeley, packets, or filtering in its current form [65]. In a nutshell, eBPF is a Linux kernel technology that supports dynamic system monitoring through the attachment of special "hooks" called BPF programs to specific kernel interfaces and userspace functions. In recent years, eBPF's role has expanded, providing an interface to make extensions to the kernel as well as the classic monitoring use case. In this section, we discuss the origins of eBPF, its components and how they work, its applications under the Linux kernel, and how it has evolved over time.

2.5.1. Origins of BPF: Efficient Packet Filtering and Beyond

The original Berkeley Packet Filter, hereafter referred to as cBPF[5], arose out of a need to implement a more efficient packet filtering mechanism for BSD Unix. McCanne and Jacobson [98] published their work on cBPF in 1993, marking an improvement over existing mechanisms in a number of ways. Many of the reasons why classic BPF was such an improvement over the status quo are still relevant when discussing *eBPF*, and so we will briefly cover them here as well.

In essence, classic BPF is a *register virtual machine* designed to take packets as input and produce *filtering decisions* as output. These filtering decisions could then used to make decisions about whether a packet should be passed down to a more complex pipeline for further analysis. The key insight behind cBPF is that these

[5]Throughout the rest of this thesis, we refer to extended BPF using the terms "eBPF" and "BPF" interchangeably. This is a matter of established convention within the eBPF community. Classic BPF will be explicitly referred to by its full name or the cBPF acronym.

filtering decisions could be made more efficiently in *kernelspace*, the part of the operating system that runs in protection ring 0[6] and which is most commonly associated with any parts of the operating system that do not run in *userland* (i.e. the context of an ordinary user process). This provides a considerable performance advantage over conventional approaches to network monitoring. A typical network monitor runs in *userspace*, meaning that packets need to be copied over from kernelspace before they can be properly analyzed. This is an expensive operation, requiring several context switches and potentially sleeping in the event of a page fault [98]. By applying filtering logic in the kernel, this expensive copying could be skipped for packets that would be discarded or ignored by the network monitor anyway.

Classic BPF can be divided into two major components: a *tap* mechanism and a set of one or more *filter* programs. The cBPF architecture is depicted in Figure 2.7. cBPF programs are expressed as a control-flow graph (CFG) over a set of abstract registers, backed by physical registers on the CPU. The tap mechanism hooks into packets as they enter the networking stack, copying and forwarding them to the filters. At runtime, the filter programs walk their control-flow graph, taking the forwarded packets as input. As output, they return a filtering decision which controls whether or not the packet should be forwarded to userspace [98].

Since its original introduction in 1993, classic BPF has since been ported to a number of Unix-like operating systems, including Linux [16], OpenBSD [17], and FreeBSD [18]. Classic BPF forms the backbone of widely used traffic monitoring tools, most notably tcpdump [98, 143]. In Linux, the `seccomp(2)` system call [3] was

[6] Code that runs in ring 0 is said to run with *supervisor privileges* and is able to access all system memory. Ring 0 is the highest level of memory protection provided by the CPU [76].

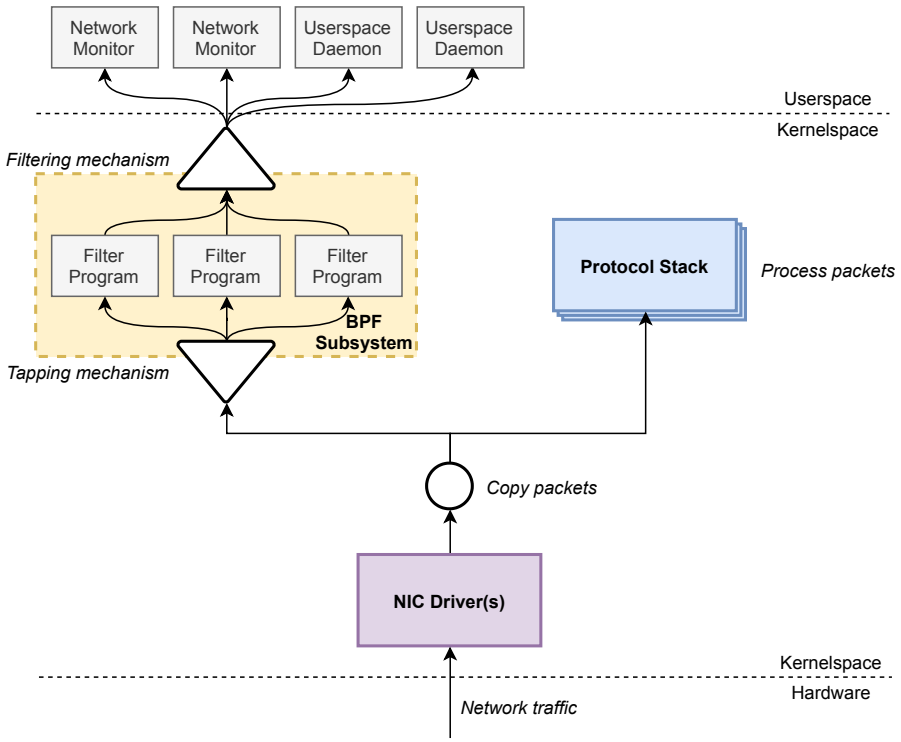

Figure 2.7: The classic BPF architecture. Adapted from McCanne and Jacobson [98].

enhanced to include classic BPF filters, allowing a user process to use classic BPF programs to define allowlists and denylists of system calls (c.f. Section 2.2.3).

In 2014, Alexei Starovoitov and Daniel Borkmann [139] first proposed a total overhaul of the Linux BPF engine. Their proposal, dubbed eBPF, expanded the classic BPF execution model into a full-fledged virtual instruction set. In particular, the extensions included a 512 byte stack, 11 registers (10 of which are general-purpose), the ability to call a set of allowlisted kernel helper functions, the ability to attach programs to a variety of system events, specialized data structures (called BPF maps) to store and share data at runtime, and an in-kernel verification engine to check for program safety. At runtime, programs can be dynamically attached to system events and are just-in-time compiled into the native instruction set. Figure 2.8 depicts the eBPF architecture in detail. The reader is encouraged to compare this with the classic BPF architecture, depicted in Figure 2.7.

Dtrace [23, 67] served as an early inspiration for the design of eBPF and related tooling. The original Dtrace model was to make low-level systems tracing available to end users by exporting a simple tracing language (called D) and supporting upstream hooking of kernel functions using this language. eBPF implements a superset of Dtrace's original functionality, enabling userspace applications to hook into the kernel and other userspace applications, attaching bytecode programs to be run as callbacks. Userspace eBPF tooling then simplifies the process of writing eBPF programs by introducing increasingly high-level layers of abstraction.

While modern eBPF has very little to do with the execution model of its older cousin, some of the properties that made classic BPF so performant still hold true today. In particular, to notion of aggregating and processing data in kernelspace

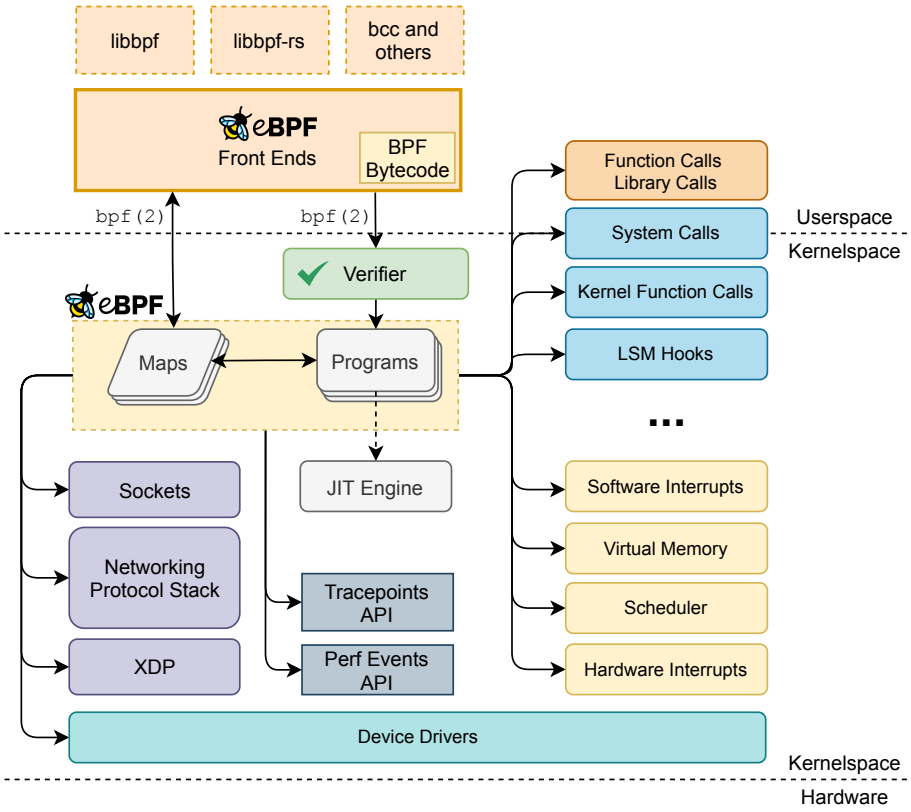

Figure 2.8: The extended BPF architecture. Unlike classic BPF, eBPF programs are JIT (Just-In-Time) compiled to the native instruction set, share data using specialized map data structures, and can be attached to many different kinds of system events. Programs can share data with each other and with the controlling userspace process using specialized map data structures. All eBPF bytecode goes through a verification step before it can be loaded into the kernel.

before (optionally) handing it off to userspace is a key aspect of classic BPF that has carried over to eBPF. What this means in practice is that eBPF programs can be used to implement very efficient monitoring software, harnessing the performance benefits of a pure kernelspace implementation while maintaining the flexibility of a userspace implementation.

2.5.2. eBPF Programs

eBPF programs are expressed in a virtual RISC machine language called BPF bytecode. While it is technically possible to write BPF bytecode by hand, programs are most often compiled from a restricted subset of the C programming language[7] using the LLVM toolchain. Programs can be loaded and attached to system events using the `bpf(2)` system call, at which point control passes to the eBPF verifier, which checks the programs to make sure they satisfy a set of safety constraints [65, 139].

In particular, eBPF programs must consist of fewer than 1 million BPF instructions and must not call into any kernel functions outside of the allowlisted helpers. The program is also constrained to a 512 byte stack size; any additional memory required by the program must come from an eBPF map (c.f. Section 2.5.3). For safety, memory accesses into allocated buffers must be properly bounds checked, pointers must be null checked before dereferencing, and any access to external memory (e.g. belonging to userspace programs or to the kernel itself) must be read only. Since eBPF programs must provably terminate, no back-edges are permitted in their control flow and all

[7] Other languages may eventually be used to write eBPF programs as well. For instance, an experimental Rust eBPF target has recently been merged into the Rust compiler [44]. The important distinction here is that the set of all possible eBPF programs is a strict subset of the set of all possible programs.

loops must be bounded by some fixed constant i iterations.

To guard against data races, eBPF programs always hold the kernel's RCU (read-copy-update) lock while executing, gated by the `bpf_prog_enter` and `bpf_prog_exit` functions in the kernel. In simple terms, the RCU lock allows concurrent reads, except in the presence of updates, optimizing for read-mostly workloads (i.e. precisely the sort of workload eBPF is designed for) [100]. This implicitly enables BPF programs to read from many common kernel data structures without fear of data races and simultaneously protects reads and updates to eBPF maps, at a slight (albeit reasonable) performance penalty [100]. In addition to holding the RCU lock, eBPF programs are not considered *preemptable* by default. In practice, this means that eBPF programs cannot sleep and must run to termination on their assigned core. This property, while useful in many circumstances, enforces undesirable limitations on eBPF helpers, since it precludes any functionality that may cause the program to sleep (e.g. a page fault). To account for use cases where sleeping is unavoidable, Linux 5.10 introduced sleepable versions of some eBPF program types [138].

Once loaded into the kernel, eBPF programs are represented as BPF objects, each with its own reference count. Loading a BPF program and attaching it to a system event increments the reference count, while detaching and unloading the program decrements the reference count. The kernel also exposes a special filesystem, *bpffs*, which allows BPF programs to be pinned. This also increments the reference count, allowing an attached program to outlive its controlling process (i.e. the process that loaded and attached it) [65].

Working with the Verifier

In practice, the restrictions imposed by the verifier mean that eBPF programs are not *Turing complete* [65]. This property is required, given that the halting problem (i.e. the decidability of program termination) is known to be unsolvable for Turing-complete programs. This notion of Turing-incompleteness means that the set of all possible eBPF programs is a strict subset of the set of all possible C programs. While these limitations help to ensure program safety, they also naturally restrict some operations which *may* be safe but are not strictly verifiable. To overcome the limitations imposed by the verifier and achieve this safe-yet-unverifiable behaviour, eBPF programmers have a few tools in their arsenal. For instance, a specific set of allowlisted kernel helpers offers the ability to call into specific kernel functions, bypassing the limitations imposed by the eBPF verifier. As a simple example, the `bpf_probe_write_user()` helper allows an eBPF program to write to a userspace memory address, bypassing the read-only restrictions imposed by the verifier. While these allowlisted helpers operate in a *mostly* unrestricted context, their usage *is* restricted at the function call boundary, ensuring that the eBPF program obeys the safety contract specified by the helper function. Another common design pattern is using a dummy eBPF map as a scratch buffer to reserve a larger amount of memory for the eBPF program. Since eBPF programs cannot sleep [65], dynamic memory allocation within the BPF context is impossible. These dummy maps offer a way to access additional memory from a pool reserved at the time the map was loaded into the kernel.

eBPF Program Types and Use Cases

Each eBPF program has a specific *program type*, which determines both the set of system events to which the program can attach and the set of allowed kernel helpers that can be called from within the program context. Each program type roughly corresponds with a distinct eBPF use case. For the purposes of this thesis, we will primarily be dealing with *LSM probes*, *raw tracepoints*, *uprobes/uretprobes*, *kprobes/kretprobes*, *fentry/fexit probes*, and *USDT (User Statically Defined Tracepoints) probes* as they form the basis of BPFBox and BPFContain's kernelspace implementations. Table 2.2 summarizes the relevant program types and their properties.

Table 2.2: A selection of relevant eBPF program types for BPFBox and BPFContain.

Program Type	Description
LSM Probes	LSM probes [129] attach to the kernel's LSM hooks and can be used to audit security events and make policy decisions.
Raw Tracepoints	Raw tracepoint programs attach to a stable tracing interface exposed by the Linux kernel. Tracepoints are considered a stable API but are more limiting than alternatives such as Kprobes or Fentry probes.
Kprobes/Kretprobes	Kprobe programs can attach to any kernel function, by replacing the function with a trap into the BPF program. The BPF program has read-only access to the function arguments. Kretprobes work in the same way, but handle function returns instead of function calls.

Fentry/Fexit Probes	A more efficient version of Kprobes and Kretprobes that directly trampolines into the BPF program instead of trapping. These programs can also be used to modify the return value of specifically allowlisted kernel functions (e.g. system call implementations).
Uprobes/Uretprobes	The userspace equivalent of Kprobes and Kretprobes.
USDT Probes	A statically defined version of uprobes and uretprobes. Application developers may place these at strategic points within an application in order to add explicit support for userspace tracing at compile-time.

LSM Probes: Making Security Decisions with eBPF

It is worth spending more time focusing specifically on LSM probes, as these are used extensively in BPFBox and BPFContain to enforce policy over security-sensitive events. Introduced by KP Singh in his KRSI (Kernel Runtime Security Instrumentation) patch [129], LSM probes define a canonical framework for attaching eBPF programs to the Linux kernel's LSM security hooks (c.f. Section 2.2.2). Unlike traditional LSMs which are implemented as static kernel modules, LSM probes are *dynamically attachable*, meaning that access control and audit policy can be adjusted at runtime, simply by loading a new eBPF program. Figure 2.9 depicts how LSM probes integrate with the LSM framework.

Each LSM probe can be attached to one or more LSM hooks defined in the kernel. When the hook fires (i.e. when a task requests a privileged operation form the kernel), every attached probe fires as part of the normal LSM pipeline. The body of the BPF program defines filtering and audit logic, optionally accessing maps to store and query persistent state. The BPF program then returns a security decision about whether

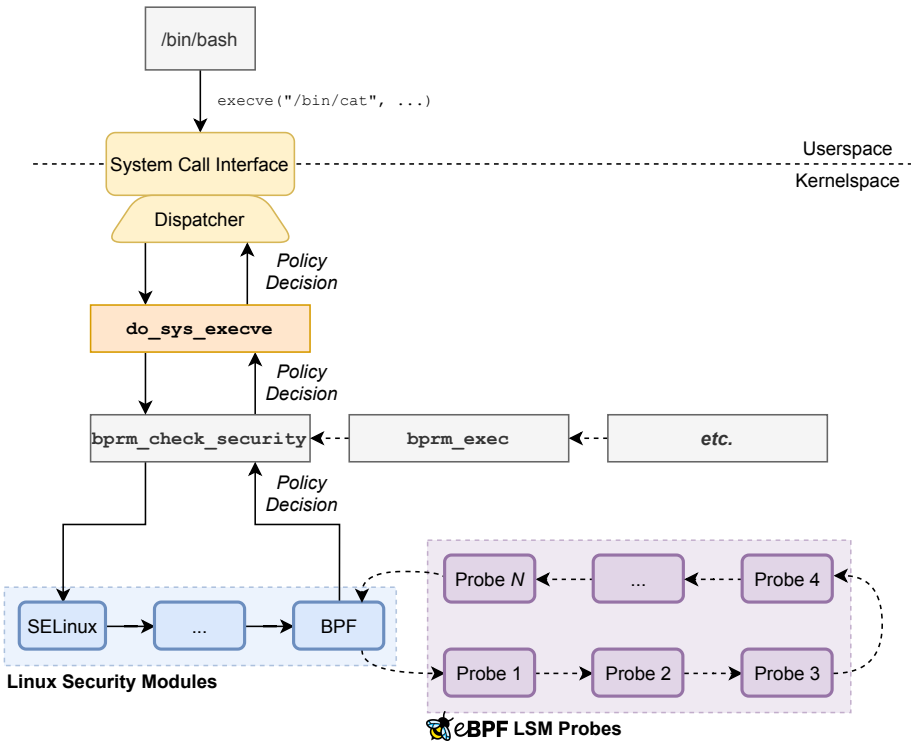

Figure 2.9: A simplified example of how eBPF LSM probes make policy decisions. Privileged userspace processes can attach one or more LSM probes to a given hook. When a userspace process requests a privileged operation, the kernel implicitly calls into the corresponding LSM hooks, which in turn invoke the logic associated with each LSM. A shim LSM is responsible for invoking each LSM probe, and any resulting policy decisions are taken together to arrive at a final decision. As with ordinary LSMs, the final decision is consensus-based. That is, if *any LSMs* or *any BPF LSM probes* disagree on a policy decision, the privileged operation is denied.

the requested operation should be allowed or denied. In order for an operation to be allowed, *all* other LSMs and LSM probes must agree on the policy decision and ordinary security checks performed by the operating system must also succeed. In other words, it is not possible to grant additional privileges using an LSM probe.

Owing to the properties discussed earlier in this section, eBPF confers a natural flexibility to LSM probes quite unlike that of traditional LSM-based security frameworks. In particular, LSM probes can be attached at runtime and can cooperate with other eBPF program types using eBPF maps (c.f. Section 2.5.3). This notion of cooperating programs presents an opportunity to design modular policy enforcement mechanisms that operate beyond the scope of the LSM hooks framework itself. Another key advantage of LSM probes over traditional LSMs lies in their adoptability. While industry actors may be understandably reluctant to adopt "yet another out-of-tree LSM", a security mechanism based on eBPF does not carry the same technical baggage. eBPF programs are safe to use in production and can be deployed at runtime on an unmodified kernel. This makes eBPF a particularly attractive target for developing new security solutions.

2.5.3. eBPF Maps

eBPF maps serve as both a runtime data store for eBPF programs and the canonical method of communication between different eBPF programs and userspace applications. Like eBPF programs, maps can be pinned to *bpffs* to increment their reference count in the kernel. Concurrent access to eBPF maps from within kernelspace is protected by an implicit RCU lock, and a spinlock concurrency primitive is exposed via a

helper function to guard map accesses between kernelspace and userspace. From the eBPF side, maps can be accessed using a set of provided helper functions. Userspace applications can access maps using the `bpf(2)` system call or through direct memory mapping (only available for arrays) via `mmap(2)` [65]. While many eBPF maps are designed to be generic, others are highly specialized for specific use cases. BPFContain and BPFBox make use of several eBPF map types (see Table 2.3.

Table 2.3: A selection of relevant eBPF map types for BPFBox and BPFContain.

Map Type	Description
BPF Hashmap	A key-value hashmap. Keys and values can be arbitrary data structures.
BPF Array	A fixed-size array with integer indices. Values can be arbitrary data structures.
BPF Array/Map of Maps	A BPF array or map that stores handles into *other maps*.
BPF Per-CPU Array/Map	Like a BPF hashmap or BPF array but with a separate copy per logical CPU. This enables concurrent access across CPUs, but without synchronization.
BPF Local Storage	A dummy BPF map that provides a handle into local storage for a given kernel data structure. For instance, task local storage provides storage per task struct. Values can be arbitrary data structures.
BPF Ringbuf	A concurrent circular buffer that passes event handles from kernelspace to userspace. To communicate, eBPF programs submit events and userspace applications poll events.

2.5.4. Userspace Front Ends

Although eBPF programs and maps can exist on their own after being pinned to *bpffs*, the more common approach is to manage their lifetime using a controlling process. Userspace applications implementing such a controlling process typically use an eBPF

front-end framework to facilitate loading and interacting with programs and maps. A number of such front ends exist [7, 29, 74, 75, 86, 118, 155], some more practical than others. *bcc* [74] was the first eBPF framework to offer high-level userspace tooling around eBPF, providing an LLVM backend for compiling eBPF programs and a Python library for loading and interacting with them. *libbpf* [86] offers a pure C alternative to bcc and has since been upstreamed into the Linux kernel. *libbpf-rs* [155] and *libbpfgo* [7] offer Rust and Golang bindings for libbpf respectively. Other tooling [29, 118] bypasses libbpf entirely, providing fully native eBPF bindings for Rust and Golang.

Libbpf and BPF CO-RE

Among the myriad of userspace front ends available for eBPF, libbpf stands out as the only one with official upstream support from the Linux kernel. Recent improvements to libbpf have solidified its position as the dominant framework. In particular, libbpf supports a new way of compiling and loading BPF programs into the kernel, BPF CO-RE (Compile Once, Run Everywhere) [66, 102]. BPF CO-RE uses BTF (BPF Type Format) debugging information exposed by the kernel, along with load-time relocation logic to support loading the same compiled eBPF bytecode across multiple target kernels.

With libbpf and CO-RE, eBPF programs can now be compiled once and run on any target kernel that supports the required BPF features. This provides a powerful advantage over other eBPF frameworks and even alternatives to eBPF, such as loadable kernel modules. A CO-RE program that runs on one kernel will be guaranteed to run on another of the same version or higher, barring any API

incompatibilities like changes in a hooked function signature. Such incompatibilities can be resolved with the use of built-in kernel configuration checks.

BPFContain (c.f. Chapter 5) leverages libbpf and CO-RE through libbpf-rs, the canonical Rust bindings for libbpf, providing adoptability advantages over the original BPFBox prototype (c.f. Chapter 4), which uses bcc.

2.5.5. Comparing eBPF with Loadable Kernel Modules

Before eBPF, the primary means of modifying the Linux kernel at runtime was through the use of *loadable kernel modules* [41]. A kernel module can be thought of as a discrete bundle of code that can be loaded into the kernel (or compiled into its binary image). Like other kernel code, including eBPF, modules are event-based and run in ring 0, responding to and handling system events as they occur. Since kernel modules and eBPF can serve similar (but not strictly equivalent) purposes, comparing the two can offer some insight about how they differ and which technology is better fit for a specific purpose.

At a first approximation, eBPF differs from kernel modules in the following meaningful ways [65]:

1. eBPF programs **must pass verification checks** before they can be loaded into the kernel. This verification step provides assurances about program safety. For instance, eBPF programs are guaranteed not to deadlock the kernel, and are far less likely to suffer from memory safety issues. In contrast, misuse of kernel APIs in a kernel module can have dangerous implications for system safety and security.

2. An implicit advantage provided by eBPF is that BPF programs can be **easier to reason about** than other kernel code. eBPF abstracts away much of the complex functionality required to make kernel code operate correctly by providing implicit guarantees about program execution. Even helper functions, which offer functionality beyond the scope of verifiability, must obey a predetermined contract with the verifier in order to be considered safe. Thus, when an eBPF program passes verification, there is a much higher likelihood that it will "just work."

3. eBPF **exposes map-like data structures** to facilitate runtime data storage, communication between eBPF programs, and communication with userspace applications. In the case of kernel modules, data structures often must be implemented by hand, taking great care not to introduce potential bugs or security vulnerabilities, particularly in the case of memory management. Communication with userspace from a kernel module might be done via netlink sockets, file operations, or similar means [41]. These modes of communication are often less streamlined and, in the case of file operations, must be implemented by hand, increasing the likelihood of programmer error.

4. eBPF programs are **not Turing-complete**. Intuitively, this means that the set of operations a kernel module can perform is a strict superset of eBPF. While this may appear to be a hugely limiting factor, in practice eBPF programs are often sufficient to implement sophisticated tracing, filtering, and policy enforcement logic. Where the verifier gets in the way, the programmer can reach for a number of helper functions provided by the kernel to achieve more complex behaviour.

5. eBPF is **not *generally* suitable for implementing device drivers** or other complex functionality that requires ad-hoc access to various kernel facilities and write access to arbitrary memory locations. Where necessary, eBPF helpers can be added to the kernel to perform more complex functionality from within a BPF program. However, these helpers must be upstreamed in the kernel in order to be used, are limited to specific program types, and must obey a safety contract with the verifier.

Table 2.4: A high-level comparison between eBPF, loadable kernel modules, and kernel patches. ● = property satisfied; ◐ = property somewhat satisfied; ○ = property not satisfied.

	Loadable at Runtime	Verified Safety	Easy Abstractions	Cross-Boundary Data Structures	Turing Complete	Complex Functionality
eBPF	●	●	●	●	○	◐
Kernel Modules	◐	○	○	○	●	●
Kernel Patches	○	○	○	○	●	●

Table 2.4 presents a summary comparison of eBPF, kernel modules, and kernel patches as a means of running custom code in the kernel. In summary, eBPF is useful for observability use cases, or cases in which the functional requirements of the kernel code are not expected to be complex or might be expected to change frequently. eBPF

programs and maps are particularly good at separation of concerns, composability, and modularity. eBPF maps facilitate easy communication between kernelspace and userspace, and provide the ability to build relationships between data from different program types. Kernel modules should be preferred for the implementation of more complex kernel functionality, such as device drivers.

Chapter 3.

The Confinement Problem

Researchers have been studying confinement for decades [82], and have been designing and applying confinement primitives since the early days of time-sharing computers and multi-tenant systems [128]. While many developments have been made in the mean time, the current confinement landscape (particularly within the Linux ecosystem) suffers from fundamental flaws that culminate in poor container security practices; this does not need to be so. This chapter presents a critique of the current state of confinement on Linux, examines how confinement primitives are applied to containers, and proposes a fundamental re-framing of the problem to focus on complexity, adoptability, and suitability for container-specific applications. In light of this re-framing, we consider design goals for BPFBox and BPFContain and present the threat model for confinement under these research systems.

3.1. Rethinking the Virtualization Narrative

Hypervisor-backed virtualization is commonly considered more secure than container-based virtualization [53, 140] (see Figure 3.1). Intuitively, this makes sense. Containers run directly on the host operating system, whereas a virtual machine runs on top of a hypervisor, separated by at least one layer of indirection from the host system. However, this intuition does not strictly stand up to scrutiny. A virtual machine running on top of a hypervisor makes requests to the hypervisor's API (via hypercalls), in much the same way as a container running on a host operating system makes requests to the kernel's API (via system calls). Figure 3.2 illustrates this parity. In both cases, a vulnerable interface into a more privileged component of the system is directly exposed to the attacker.

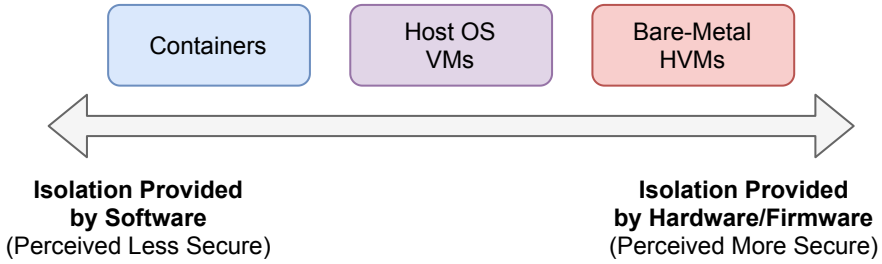

Figure 3.1: Comparing the isolation and perceived security of containers and VMs. A boundary defined in hardware is often considered more rigid, whereas a software defined boundary is generally considered more malleable, and thus weaker. However, this need not be the case. A goal of this thesis is to reposition containers to be as secure, if not more secure, than hypervisor-based virtualization.

In the case of virtual machines, security is an emergent phenomenon. The implicit isolation provided by a virtual machine is purely a function of the semantic gap between guest operating system, virtualized hardware, and physical hardware. Here,

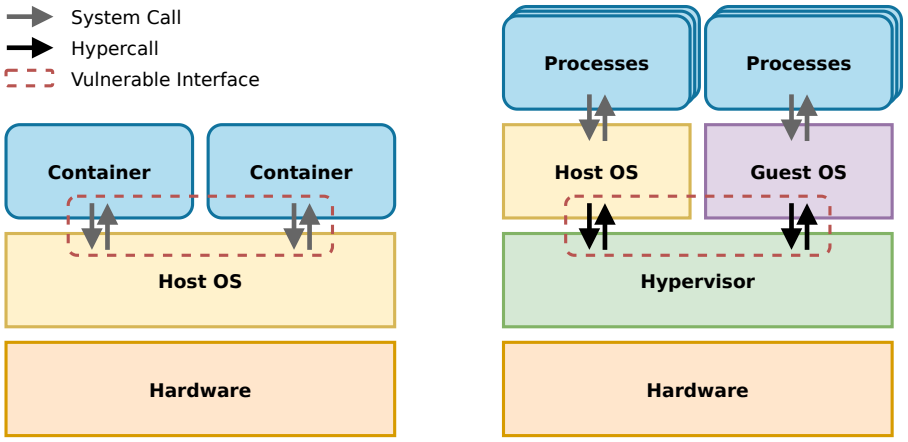

Figure 3.2: System calls and hypercalls as vulnerable interfaces. Containers (which are really just process groups running on the host OS) make system calls to the more-privileged host OS kernel. Similarly, a guest operating system makes hypercalls to the more-privileged hypervisor. Both of these interfaces are ripe targets for a myriad of attacks, including privilege escalation and tampering with sensitive resources. This parity becomes particularly evident when we assume that an attacker has or is able to obtain control over the guest OS.

security is intrinsically tied to functionality. Administrators poke holes in this isolation all the time through shared filesystems, virtual network interfaces, and other virtual device drivers. From there, it is up to the administrator to use conventional OS security mechanisms (c.f. Sections 2.1 and 2.2 in Chapter 2) such as filesystem access controls and network firewalls to lock down the exposed interfaces. A clear pattern emerges: first provision the resource, then secure it, just as in an ordinary operating system (after all, a guest operating system *is* just an operating system). The end result of this process is a combination of implicit isolation and additional OS-level security mechanisms: a form of *policy through mechanism*.

The central argument of this thesis is that there is no fundamental reason why containers cannot be as — if not more — secure than virtual machines. While it is true that a containerized process is nothing more than a host process running under one or more OS-level virtualization primitives, an operating system that provides and enforces the right set of confinement primitives should be able to lock a container down in much the same way that a hypervisor implicitly enforces isolation. Unlike hypervisor-based isolation, such a policy enforced by the operating system has the potential to be more minimal, centralized, and accessible to end users. Further, we can attain a high level of flexibility through a software-based enforcement mechanism, unencumbered by the restrictions of a hardware-level interface.

The key to developing such an enforcement mechanism lies in defining a clear protection boundary within the kernel and enforcing access across this boundary. Rather than *policy through mechanism*, this would be *policy through exception*. Under this model, the operating system enforces a clear protection boundary around the container, and security policy then defines exceptions in this boundary, exposing a

minimal interface to the outside world. Such an interface has the potential to be *far more minimal* than that of a virtual machine, by taking advantage of the fine-grained semantics exposed by the operating system.

However, the above depiction of container security does not match the current reality. Existing container runtimes simply reuse confinement primitives exposed by the operating system, combining multiple OS-level security mechanisms to secure specific system resources. This approach is neither simple nor unified, and results in increasingly complex and verbose policies, inundated with the technical baggage associated with locking down an entire operating system. Yet it is hard to fault the designers of container runtimes for this design choice; a container runtime's job is to run containers, not to implement new security mechanisms. Intuitively, it makes sense to reach for primitives that already exist, regardless of the fact that these primitives may have been designed for system-wide use cases, beyond individual processes or even process groups. Likewise, the goal of this thesis is not to design a new container runtime — rather, it is to implement the missing confinement mechanism that will enable our vision for container security to become a reality.

Before we can address concrete goals for such a confinement mechanism (c.f. Sections 3.4 and 3.6), it is important to understand the fundamental issues underlying the current state-of-the-art in Linux confinement and container security. To that end, Section 3.2 outlines three main issues with current Linux confinement primitives and Section 3.3 examines and critiques how container runtimes apply these primitives in practice.

3.2. Fundamental Issues with Linux Confinement

Here we identify three fundamental issues with the current state of Linux confinement and contextualize them with examples from existing container and confinement frameworks. This section serves to provide additional context for the container-specific issues discussed in Section 3.3, and the design goals for BPFBox and BPF-Contain, discussed in Section 3.4.

1. **Complexity, Interdependence, and Inflexibility.** Existing confinement primitives are overly complex and designed for use cases beyond simple process confinement. To achieve simple confinement, frameworks must abuse and recombine a number of existing primitives, each designed for a different use case. Namespaces [14, 104] were designed to virtualize resources; they do not provide confinement by themselves. To truly confine a process using namespaces, we need a way to account for namespace escapes. Cgroups [34] similarly, were designed to virtualize the availability of quantifiable resources, not to directly confine. Unix DAC [76, 107] is far too coarse-grained and easy to bypass to be directly useful for confinement. POSIX capabilities [24] can be used to reduce overprivilege by partitioning root privileges, but these do not implement confinement by themselves.

   Seccomp-bpf [54, 126] works well to reduce the attack surface exposed by system calls, but writing classic BPF filters is a complex and error-prone process. Fine-grained filtering quickly becomes untenable, particularly when considering race conditions when checking system call arguments and system call equivalence classes. Linux MAC can be used to implement true confinement, but a typi-

cal LSM like AppArmor [42] or SELinux [130] is designed for use cases beyond simple sandboxing. These mechanisms are designed to implement and enforce system-wide MAC policy, not simple process-level confinement [11]. Further, major LSMs are statically loaded and unstackable, meaning that end users must generally choose one major solution with little room to adjust enforcement at runtime.

To implement confinement, sandboxing and containerization frameworks generally mix and match the aforementioned solutions — some of which were designed for confinement, some of which were not, and none of which were designed for *simple, process-level* confinement. LXC, Docker, Snap, and others all combine namespaces, cgroups, capabilities, seccomp-bpf, and AppArmor/SELinux policy to achieve confinement. In the case of Snap, high-level abstractions in policy definition can simplify the process of policy authorship to a certain extent, but simple policies are still compiled down into thousands of lines of policy soup, spanning multiple confinement mechanisms. FreeBSD Jails and Solaris Zones take an alternate approach, enforcing a rigid protection boundary around the container, but with little flexibility for policy customization.

2. **Unsuitability for Containers.** Existing Linux DAC and MAC is unsuitable for containers. LSM-based MAC implementations like AppArmor and SELinux are designed to implement global, system-wide confinement policy; they are not designed for ad-hoc, process-level confinement [11]. Additionally, these LSM implementations were not designed with containers in mind and thus do not consider container semantics in policy definition and enforcement. This lack of semantic

awareness further complicates policy authorship for containers and forces the end user to make compromises between security and functionality.

Sultan *et al.* [140] suggest that the container security community should move towards a container-specific LSM implementation. Security namespaces, proposed by Sun *et al.* [141] can be seen as a partial step toward solving this problem. Under security namespaces, each container can load and use its own LSM of choice, but these LSMs would still be subject to many of the same aforementioned restrictions. That is, existing LSMs are not designed with container semantics in mind. A truly container-specific LSM could incorporate container semantics into policy enforcement for cleaner and more effective policies.

UID remapping under a new user namespace does help with the DAC case by remapping root to a non-root UID, but this is only really helpful for limiting the power of root. Other limitations of DAC still apply. For instance, a world-readable file could still be used for information disclosure or a world-writable file could still be the target of data corruption. For such reasons, DAC alone appears to be fundamentally insufficient for true isolation between host and container. Thus, to achieve proper process-level confinement for containers, we need an LSM-based solution that is aware of container semantics.

3. **Difficulty Adopting New Solutions.** Motivated by the inherent difficulties associated with the existing confinement space, academics are often tempted to propose new confinement solutions. Many try to solve the problem by simply recombining and reusing existing primitives in new and innovative ways. However, these types of solutions generally are not really a step forward with respect to

addressing the issues in items 1 and 2, since these are emergent properties inherent to the underlying confinement primitives themselves.

In order to truly solve these fundamental issues, we need kernel support for new primitives. Unfortunately, this begets yet another fundamental issue: adding new solutions directly to the kernel is difficult, particularly from an adoptability standpoint. New kernel code can introduce bugs and security vulnerabilities. It needs to be thoroughly tested before it can be considered production ready. Paradoxically, the potential to introduce new security vulnerabilities can make the use of such novel primitives *less* secure. Similarly, kernel bugs can introduce availability concerns in production systems, even when such bugs are not security critical. For these reasons, industry managers may be reluctant to adopt new, out-of-tree solutions based on loadable kernel modules, for example [65].

[TODO Anil's comment: What changes? Are Solaris Zones not enough?] Another adoptability concern arises when we consider *container-specific* confinement [140, 141] as an end-goal. To date, the definition of precisely what a container is has been more or less in flux. The requirements and precise specifications of what constitutes a container tend to change as container frameworks evolve and new use cases crop up. If not everyone can agree on what a container even is, how can we expect to reach agreement on which underlying container security abstraction should be merged into the mainline kernel? To solve this problem, we need a way to add abstractions into the kernel in such a way that is neither binding nor limited by the lack of adoptability associated with traditional kernel-based solutions. These requirements motivate the use of eBPF

for designing a container-specific security solution, and thus motivate the design of BPFBox and BPFContain.

3.3. How Containers Apply Confinement Primitives

This section examines and critiques the way Linux container technologies apply confinement primitives to lock down container deployments. We focus primarily on Docker as a case study; however, these principles in general apply to the majority of container management frameworks.

In general, Linux containers have three broad goals. However, these goals are neither equally met nor equally prioritized by existing container management frameworks. In order of decreasing prioritization, they are:

1. **Dependency Management / Reproducibility.** Containers should provide an easy and robust framework for creating reproducible development environments. Dependencies should be maximally self-contained such that a containerized environment "just works" to the maximum possible extent. We can see examples of this property in Docker, the predominant container framework at the time of writing. Docker Hub [48] allows container images to be pulled from the Internet, recombined, and used to create further images. The end result is a flexible framework for creating and distributing reproducible development environments.

2. **Virtualization.** Containers should virtualize system resources, creating the illu-

sion of running on a separate physical machine. Where possible, resources should be transparently reused between multiple containers (e.g. sharing a single base copy of the same shared library between two container images).

To achieve virtualization, containers generally rely on the namespaces and cgroups primitives provided by the Linux kernel. Overlay filesystems [21] combined with the mount namespace allow containers to perform one-way sharing of filesystem resources. The PID namespace allows each container to have its own *init* process and virtual process tree. The network namespace allows the container to virtualize its network devices while the UTS (Unix Timesharing System) namespace virtualizes host and domain names. Control groups virtualize other resources such as the CPU, main memory, and device drivers.

3. **Confinement.** Containerized processes should be confined by default. That is, a containerized process should have access to the minimal set of privileges required for it to operate normally. Container runtimes leverage existing confinement primitives provided by the operating system, when available, to confine themselves. However, the extent to which this property is achieved varies greatly, both by the specific container runtime and by the characteristics of the deployment environment [22, 88, 140]. In general, proper confinement is not a high priority of container runtimes, and this tends to result in sacrificing security for ease of deployment.

The aforementioned goals are not only ordered by their decreasing prioritization in extant container management frameworks; they are also ordered by increasing relevance to container security. That is to say, existing frameworks generally prioritize

goals unrelated to security and leave security as an afterthought. Since containers are really just process groups running directly on the host operating system, an unconfined container therefore exposes the same attack surface as an ordinary host process. Thus, one might expect container security to be of paramount importance. Unfortunately, this is not the case. These difficulties in confinement motivate the need to revisit container security and approach it from a confinement-first perspective. To understand how these confinement issues impact containers, we briefly review how container management systems apply confinement primitives in practice.

To achieve confinement in the first place, container frameworks cobble together existing confinement technologies and apply them ways that are often simultaneously confusing and difficult to audit. The result is a complex policy soup with little room for customization or auditability. Item 1 in Section 3.2 outlines some examples of the inherent complexity that arises from mixing and matching confinement primitives in this way. To deal with this complexity, some container runtimes elect to use a high-level policy language that compiles down to thousands of lines of policy under the hood. Snap [132] is one such mechanism. Docker [49, 50, 51] instead elects to use an overly-permissive, generic policy template to avoid the potential issues associated with fine-grained policy defaults.

Part of the problem in confining containers is that, in general, they are designed to "just work". Overly fine-grained security policies may get in the way of this, particularly as end user requirements vary and evolve across deployments. Docker [50], for instance, provisions an overly-permissive default AppArmor policy [51] designed to enforce basic protections against interacting with sensitive kernel parameters without impacting the functionality of the container.

Even worse, many container management systems operate under a fail-open approach when the necessary security mechanisms are not supported. This results in low-security deployments, often without even notifying the user that there may be such a configuration. Since the end user generally doesn't even participate in the policy authorship process, they may not even be aware of the level of protection that is being applied, resulting in a dangerous false sense of security. Docker's AppArmor policy [49, 51], for instance, is not applied when the deployment environment doesn't support AppArmor or AppArmor is disabled. Snap [132] and others that rely on the AppArmor or SELinux LSMs for confinement suffer from similar failings.

Other aspects of confinement policy may be ignored entirely or even worse, overridden by a more permissive policy, possibly without the user's knowledge. Docker [50] applies a dangerously permissive `iptables` policy that can transparently expose a container to an external network, even overriding existing deny rules. This overly-permissive network policy was the direct cause of a recent data breach at NewsBlur [31], a news aggregation website.

3.4. Design Goals

To rectify the issues discussed in Section 3.2 and Section 3.3, this thesis introduces two novel confinement mechanisms, BPFBox and BPFContain, implemented using eBPF. BPFBox (c.f. Chapter 4) is a sandboxing framework that enables the definition of simple yet precise per-application policies that can be dynamically loaded and enforced at runtime. Leveraging eBPF's system introspection capabilities, BPFBox policies can specify rules that span userspace and kernelspace, targeting be-

haviours at the per-function-call level and enforcing policy through LSM hooks.

BPFContain (c.f. Chapter 5) extends BPFBox to model container semantics, enabling it to clearly define a hard boundary around containerized processes. BPFContain policies then define explicit exceptions to the default protection boundary, offering fine-grained control over the interface that a container exposes to the outside world.

The ultimate goal of BPFBox and BPFContain is to expose centralized, flexible policies that are simple enough for an end user to perform ad-hoc confinement. In the case of BPFContain, this goal is further extended to promote the adoption of container-specific policies that isolate by default and can be extended to support inter-container communication and resource sharing. To guide BPFBox and BPFContain toward this goal, we consider three primary design goals, derived from the fundamental issues identified in Section 3.2. They are enumerated as follows.

1. **Simple and Flexible Policies.** Policies should be simple and flexible, without sacrificing expressiveness. It should be possible to use our solution for ad-hoc confinement of individual applications and containers, without worrying about the underlying details of enforcement. At the same time, the policy language and enforcement engine should be flexible enough to support expressive and fine-grained policies that target specific system resources where required. That is, the barrier to entry for writing an effective security policy should be low, yet it should still be possible to write a sophisticated security policy where needed. Further, the policy enforcement engine underlying our confinement solution should be readily extensible, such that new kernel interfaces and policy rules can be easily supported

as required.

2. **Suitable for Containers.** Our confinement solution should be suitable for containers. To support this goal, the policy language should encourage the authorship of lightweight, localized policies, tailored to specific use cases rather than a heavyweight, system-wide MAC policy. An ideal policy language for this purpose should be designed with container semantics in mind, enforcing a strong boundary around a container and related resources. To support inter-container communication and resource sharing, such a policy language should support the ability to selectively define exceptions to this boundary, as required.

3. **High Adoptability.** Our confinement solution should be readily adoptable, even in production environments. All privileged code should be verifiably production-safe and should not negatively impact the rest of the system when loaded into the kernel. Performance overhead should at least be in line with alternatives like SELinux and AppArmor and our solution should work out of the box on a vanilla Linux kernel, without requiring any out-of-tree kernel patches or modules.

The key insight behind this work is that novel *kernel-level mechanisms* are required to realize the aforementioned design goals. eBPF provides precisely the right framework for developing such mechanisms. Its safety, compatibility with vanilla Linux kernels, and the ability to dynamically load and unload programs all contribute to strong adoptability guarantees. The ability for distinct program types to use eBPF maps to communicate and share state enables the development of powerful confinement solutions that can unify policy across disparate interfaces. Using eBPF, we can

trace individual userspace and kernel function calls, along with the entire lifecycle of a process or container. This property enables the creation of a fine-grained policy enforcement mechanism that can easily be adapted and extended to support new semantics and kernel interfaces as required.

3.5. Why Two Implementations?

In light of the design goals outlined in Section 3.4, we now examine why this thesis presents *two* confinement implementations, rather than just one. The simple explanation for this is that BPFBox can be seen as a rough, first cut at solving the confinement problem described in this chapter. Rather than a completely new system, BPFContain should be seen as an *iteration* on the original BPFBox design. In particular, BPFBox satisfies each of the design goals enumerated in Section 3.4 to varying degrees; BPFContain improves upon this by further simplifying the policy language, introducing container-level policy semantics, and improving adoptability by leveraging BPF CO-RE (c.f. Section 2.5.4 in Chapter 2).

Thus, BPFBox and BPFContain should not be seen as competing or even complementary solutions to the confinement problem. Rather, the delta from BPFBox to BPFContain is representative of the intellectual journey from a first approximation to a far more refined approach, more conducive to the insights outlined in this chapter. Chapter 4 and Chapter 5 outline this journey in more detail, focusing on specific implementation differences between the two systems and how they arise from an evolution in understanding the confinement problem from a container-centric perspective.

3.6. The BPFBox and BPFContain Threat Model

This section outlines the threat model for BPFBox and BPFContain. In particular, we provide a scoping definition of confinement policy and what it means for a policy enforcement mechanism to confine a subject using that policy. We also discuss the adversary's capabilities, goals, and potential attack vectors in a commodity Linux-based operating system. In general, both BPFBox and BPFContain have a very similar threat model, with subtle and specific differences arising in a few key areas. Where discrepancies arise, they will be noted accordingly.

3.6.1. Confinement Policy and Enforcement Engine

We define *confinement* as the restriction of *subject* (system actors such as processes) behaviours to a set of desired behaviours, as they pertain to *objects* (system resources such as files, network sockets, devices, and IPC handles onto other subjects). Consider the set of all subjects $\mathcal{S}$ and the set of all objects $\mathcal{O}$. We define $\mathcal{S} \subseteq \mathcal{O}$ to account for IPC objects. The goal is to create a confinement policy $\mathcal{P}_i$ for a subject $\mathcal{S}_i$ that maps $(\mathcal{S}_i, \mathcal{O}_j, Op)$ tuples to *policy decisions* where Op is an operation $\mathcal{S}_i \xrightarrow{Op} \mathcal{O}_j$. The policy is written in some abstract policy language that encodes such tuples in a human-readable format.

An *enforcement engine* operates between the subject and system objects. It intercepts requests to access these objects and makes an access control decision according to the corresponding confinement policy. The end result is a behavioural restriction on the subject to some subset of all possible behaviours. The goal of an effective confinement policy and enforcement engine is to achieve the minimal possible subset

for normal operation, thus achieving the principle of least privilege and minimizing the attack surface for potential exploitation.

Under BPFBox, the enforcement engine targets access at the process level, interposing on individual system calls using eBPF programs attached to LSM hooks and enforcing access based on per-executable policies. Under BPFContain, this enforcement is expanded to operate at the container level. Like BPFBox, BPFContain interposes on system calls using eBPF programs attached to LSM hooks, but these programs account for the state of an individual container, including properties like a process' container membership and whether a given resource is part of a container-local namespace. Whereas BPFBox defines its protection boundary around the process, BPFContain defines a protection boundary around the container itself; access to resources *within* the container is considered default-allow, whereas resources *outside* of the container or operations that can affect global system state are considered default-deny.

3.6.2. The Adversary and Attack Vectors

We consider a privileged remote adversary with root-level access under conventional Unix discretionary access controls. Further, we assume that the adversary has already achieved local code execution at the process level. This means that the adversary is capable of running arbitrary code in the context of a given process or container and can perform arbitrary interactions with the kernel's reference monitor; these interactions may be allowed or denied at the discretion of the reference monitor, and may or may not result in the subsequent execution of kernel code, such

as system call implementations. Without any confinement in place, the adversary is capable of reading or writing any file, accessing any device, loading kernel code, and performing any other privileged operation.

Our goal is to confine the adversary such that they are unable to access security-sensitive resources, interfere with external processes, make changes to global system state, or perform any other operation in violation of our security model. The adversary's goal is simple: to escape confinement. This goal of escaping confinement (tantamount to privilege escalation) can be a subgoal used to achieve some other purpose, such as spoofing, tampering, information disclosure, or persistence.

3.7. Summary

This chapter has presented a novel framing of the confinement problem as it pertains to Linux and Linux containers. In particular, we reexamine the differences between virtual machines and containers and argue that the former need not be more secure than the latter, we identify three distinct problems with modern Linux confinement, and we examine how containers apply existing confinement primitives. This re-framing of the confinement problem both serves as a motivation for the creation of BPFBox and BPFContain and informs the design goals behind these two research systems. Finally, we present a high-level threat model for BPFBox and BPFContain, providing a scoping definition for what it means to confine an adversary. The next two chapters, Chapter 4 and Chapter 5, present the design and implementation of BPFBox an BPFContain in detail and outline the intellectual journey from a prototype sandboxing mechanism to a container-specific confinement

solution.

Chapter 4.

BPFBox: A Prototype Process Confinement Mechanism

This chapter presents the design and implementation of BPFBox, an initial research prototype of an eBPF-based confinement framework. BPFBox is the first full-fledged confinement framework to leverage KRSI's LSM programs to enforce high-level policy. Using eBPF, it combines various behavioural aspects of the sandboxed application from both userspace and kernelspace to enforce a simple, yet fine-grained policy defined in a domain-specific policy language. Portions of this chapter were part of a previously published paper at CCSW'2020, co-authored with Anil Somayaji and David Barrera [59].

4.1. BPFBox Overview

At a high level, BPFBox is a confinement mechanism based on eBPF. As outlined in Section 3.4 of Chapter 3, our primary design goals are simplicity, flexibility, and suitability for containerized applications[1]. With this in mind, BPFBox attempts to be as lightweight as possible, with a simple policy language that supports optional granularity. Perhaps the most important goal of BPFBox may be derived from the aforementioned goals: to make per-application security policy accessible to end users. To achieve these goals, we leverage eBPF for BPFBox's kernelspace implementation and rely on a number of its intrinsic properties.

In particular, we take advantage of multiple program and map types (outlined in Section 4.2.1). This design enables us to trace multiple aspects of system behaviour, including userspace and kernel function calls, and combine these with LSM-layer enforcement, thanks to the KRSI extension that enables BPF programs to be attached to LSM hooks. By sharing data across program types in this way, we enable BPFBox to define extremely fine-grained LSM policy at the per-function-call level, something which no existing process confinement mechanism can do.

Since eBPF programs may be loaded dynamically into a vanilla kernel and provide implicit safety guarantees thanks to the verifier, we ensure that BPFBox is both lightweight and more adoptable than conventional solutions based in static LSMs like SELinux or AppArmor. Since all of BPFBox's kernelspace code is pre-verified, it is also significantly less likely to adversely affect a production kernel than an alternative solution implemented as a kernel patch or kernel module.

[1] While BPFBox marks a step toward achieving this goal, BPFContain (Chapter 5) is far better suited to container-level confinement.

Whereas the kernelspace components are implemented using eBPF programs written in C, BPFBox's userspace components are implemented in Python3. In particular, this consists of a privileged daemon loads BPFBox's eBPF programs and maps into the kernel, manages their lifecycle, and logs policy enforcement actions for later examination.

4.1.1. Policy Enforcement at a High Level

To confine an application, a user first authors a high-level policy written in BPFBox's domain-specific policy language[2] (outlined in Section 4.3). The policy language is designed in such a way as to permit the authorship of simple confinement policies while offering the ability to augment them with specific context. Thus, the user has full control over the balance between policy expressiveness and policy simplicity. We expect that application authors may wish to take advantage of BPFBox's full expressiveness, whereas end users may wish to overlook advanced features in favour of simple, lightweight confinement policy.

Once a policy has been written, the user places it in a pre-determined policy directory and loads `bpfboxd`, the BPFBox daemon. The daemon compiles and loads its BPF programs and maps into the kernel, then parses the user-supplied policy and encodes it into policy maps. When a user launches the target application, BPFBox begins tracing the lifecycle of the corresponding processes and associates them with the correct policy. As the application runs, BPFBox continually updates

[2]Subsequent iterations on BPFBox experimented with a TOML-based policy language, before the transition to BPFContain. We document the original domain-specific language here, and leave alternative policy languages to BPFContain (c.f. Chapter 5).

security-critical aspects of its state, stored in intermediary maps. As the application makes requests to sensitive resources, BPFBox queries policy maps and state maps and uses this information to come to an enforcement decision. Figure 4.1 outlines this process in full.

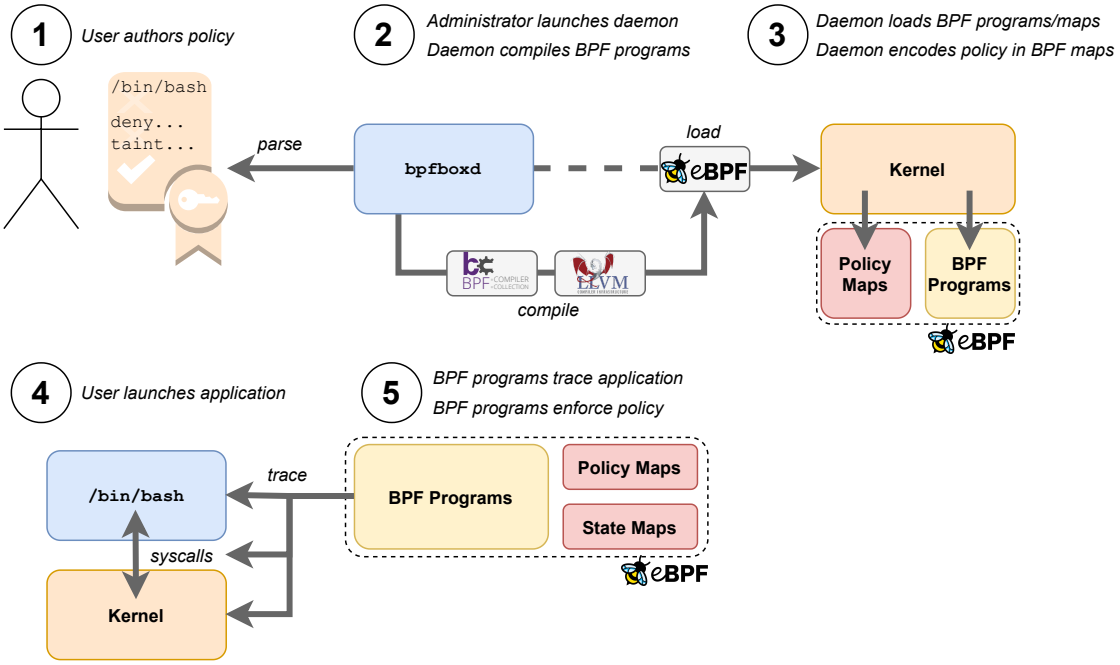

Figure 4.1: A high-level overview of how BPFBox confines applications. Users write policy files which the daemon encodes into eBPF maps. Using the bcc toolchain for Python, the daemon compiles and loads its eBPF programs and maps into the kernel. At runtime, BPFBox's BPF programs trace the application and enforce security policy by querying the state stored in its maps.

4.2. BPFBox Implementation

This section presents the implementation details and full architecture of BPFBox. In particular, we provide an architectural overview and discuss BPFBox's policy enforcement implementation, along with how it tracks and manages the state and lifecycle of sandboxed processes. We focus specifically on implementation details here, leaving policy language design and documentation to Section 4.3.

4.2.1. Architectural Overview

In userspace, BPFBox runs as a privileged daemon (`bpfboxd`), implemented in Python3 using the bcc [74] userspace library for eBPF. The daemon uses the LLVM toolchain [91] to compile eBPF programs which are then loaded into the kernel using bcc-provided wrappers around the `bpf(2)` system call. The daemon provides a userspace front-end to BPFBox, managing the lifecycle of its BPF programs and maps and logging security-sensitive events as they occur. To load policy into the kernel, `bpfboxd` implements a full parser and lexer for BPFBox's custom policy language. After parsing policy, `bpfboxd` encodes it into a format that can be subsequently loaded into kernelspace through BPF maps.

BPFBox's kernelspace components are implemented in eBPF and based on several BPF program and map types, summarized as follows. Maps are outlined in **green** and programs are outlined in **purple**. The reader is encouraged to revisit Sections 2.5.2 and 2.5.3 in Chapter 2, where necessary, for clarification on specific program and map types.

Maps:

- BPFBox uses **Hashmaps** to store runtime state, share state between its BPF programs, and communicate with the userspace daemon. In particular, BPFBox maintains a set of hashmaps to store per-process state and a set of hashmaps to store policy. We call these *state maps* and *policy maps* respectively. At runtime, BPFBox's LSM programs query these state maps and policy maps to make enforcement decisions.

- **Ringbufs** provide BPFBox's BPF programs with a canonical data store to push per-event audit data to userspace. In the kernel, the ringbuf map is implemented as a circular buffer that is efficiently shared across all CPUs. In userspace, the BPFBox daemon maps the ring buffer into memory and continually polls for new events over a fixed interval.

Programs:

- **Tracepoints** enable BPFBox to track the state of a process from the point where it forks or executes a new binary to when it exits. BPFBox stores per-process state from its tracepoints in *state maps* for later use.

- **LSM Probes** enforce policy by attaching to LSM hooks in the kernel. These hooks are called by kernel functions such as system call implementations and trigger the corresponding BPF program, which then enforces a policy decision on the target application. To enforce policy, BPFBox's LSM probes query *policy maps* and *state maps*.

- **Kprobes and Uprobes** are used to enforce *stateful policy*, according to what function calls a process has made, in kernelspace and userspace respectively. A BPFBox policy file may outline that certain rules should only apply within the context of a specific function call; when a process runs some code that results in such a function call, the corresponding kprobe or uprobe will make an update to the process' *state map* to indicate this. BPFBox then considers this state when making later enforcement decisions.

- **USDT Probes** form the backbone of *libbpfbox*, providing a kernel-side implementation for various "commands". Commands are implemented in userspace as stub USDT functions that trap to a kernelspace USDT program. These are used to load policy into the kernel and perform various other interactions between the daemon and its BPF programs and maps.

4.2.2. BPFBox Policy Enforcement

BPFBox policies are written using a custom policy language. BPFBox's policy language supports three distinct policy decisions for a given rule; the operation may be allowed, audited (logged), and/or tainted. Any unspecified operations are denied by default. Tainting is similar in spirit to Perl's classic taint mode [72], however, rather than marking data, it marks the entire process. Tainting allows for more restrictive policies to be enforced once a process has engaged in specific unsafe operations, say by reading from a network socket. We present the design and syntax of the BPFBox policy language in Section 4.3; here we discuss the functionality it provides and how it is implemented.

BPFBox policies are per-executable and are stored in an exclusively root-controlled directory (by default, `/var/lib/bpfbox/`), written in BPFBox's policy language (c.f. Section 4.3). When an executable is loaded, BPFBox loads the corresponding policy file (if it exists) and translates it into a series of function calls to USDT stub functions. These function calls trigger the corresponding eBPF code, thus recording the policy in the policy maps as a set of policy structures. A policy structure consists of three distinct access vectors: one to define tainting operations, one to define allowed operations, and one to define audited operations.

In order to enforce policy, BPFBox leverages the KRSI patch by KP Singh [40, 129] which was upstreamed in Linux 5.7. This patch provides the necessary tools to implement MAC policies in eBPF by instrumenting probes on LSM hooks provided by the kernel. The eBPF program can then audit the event and optionally enforce policy by returning a negative error value. BPFBox instruments several LSM probes covering filesystem access, IPC, network sockets, `ptrace(2)`, and even `bpf(2)` itself. When these hooks are called in the kernel, they trigger the execution of the associated eBPF program which is, in general, composed of the following six steps:

1. Look up the current process state. If no state is found, the process is not being traced, so **grant access**.

2. Determine the *policy key* by taking the executable's inode number and filesystem device number together as a struct.

3. Look up the policy corresponding to the *policy key* calculated in step (2). If the process is *tainted* and no such policy exists, **deny access**.

4. If the process is *not tainted* and the current access corresponds to a **taint rule**, **taint** the process and **grant access**.

5. If the current access matches an **allow rule**, **grant access**. Otherwise **deny access**.

6. If the current access matches an **audit rule** or access is **denied**, submit an *audit event* to userspace.

When a sandboxed application requests access, a corresponding LSM hook is called which in turn traps to one or more of BPFBox's LSM probes. The probe queries the state of the currently running process along with the policy corresponding to the requested access and takes these factors together to come to a policy decision.

The ability to combine various aspects of system behaviour, both in kernelspace and in userspace, is a key advantage of an eBPF-based solution over traditional techniques. BPFBox uses this capability to optionally augment the information provided by the LSM hooks themselves with additional context obtained by instrumenting other aspects of process behaviour. For instance, profiles may optionally define *function contexts* which determine the validity of specified rules; a rule could specify that a certain filesystem access must occur within a call to the function `foo()` or that it must be audited within a call to the function `bar()`. This allows for the creation of extremely fine-grained policies at the discretion of the policy author. The mechanisms by which this is accomplished are discussed further in Section 4.2.3.

Due to BPFBox's strict resolution of filesystem objects at policy load time, a problem arises when dealing with applications that read or write temporary files on

disk or create new files at runtime. In order to circumvent this issue, BPFBox treats the creation of new files as a special case. In order for a new file to be created, the process must have write access to the directory in which the files will be created. Supposing, for instance, the temporary file would be written to `/tmp`, this means that, at a minimum, the policy in question must specify that `/tmp` is writable. When the sandboxed application creates a new child inode of `/tmp`, BPFBox dynamically creates a temporary rule that grants the application full read, write, link, and unlink capabilities on the created file. This rule is keyed using a combination of the standard filesystem policy key and the PID (process ID) of the sandboxed process. This rule is then automatically cleaned up when the process exits or transitions to a new profile.

Another important detail to consider is the possibility of other applications using the `bpf(2)` system call to interfere with BPFBox's mediation of sandboxed applications. For instance, another application might attempt to unload an LSM probe program or make changes to the policy or process state maps. To prevent this, BPFBox instruments an additional LSM probe to mediate access to `bpf`. It uses this probe to deny all calls to `bpf` that attempt to modify BPFBox's programs or maps that do not directly come from BPFBox itself. Further, all sandboxed applications are strictly prohibited from making *any* calls to `bpf`—a sandboxed application has *no business* performing the kind of powerful system introspection that eBPF provides.

Similarly to mandatory access control systems like SELinux [130] and AppArmor [42], BPFBox supports the ability to run in either a *permissive mode* or *enforcing mode*. When running in permissive mode, BPFBox continues to audit denied operations, but allows them to continue unobstructed. This enables the user to debug policies before putting them into effect and also introduces the possibility

of creating new policy based on the generated audit logs.

4.2.3. Managing Process State

In order for BPFBox to know what policy to apply to a given process, it must track the lifecycle of processes through the instrumentation of key events within the kernel. For this, BPFBox uses three tracepoints exposed by the scheduler: `sched:process_fork`, `sched:process_exec`, and `sched:process_exit`. Figure 4.2 shows the events that BPFBox instruments in order to track process state, along with their corresponding probe types. These tracepoints are used to create, update, and delete per-task entries in a global hashmap of *process states*. Each entry in the map is keyed by TID (Thread ID). The entries themselves consist of a data structure that tracks policy key association and a 64-bit vector representing the *state* of the running process. This state vector is used to track whether the process is currently tainted and what important function calls are currently in progress.

Instrumenting a tracepoint on `sched:process_fork` allows BPFBox to detect when a new task is created via the `fork(2)`, `vfork(2)`, or `clone(2)` system calls. This tracepoint creates an entry in the *process states* hashmap and initializes it according to the state of the parent process; if the parent process is associated with a BPFBox profile, its key is copied to the child until such time as the child makes an `execve(2)` call.

The `sched:process_exec` tracepoint is triggered whenever a task calls `execve` to load a new program. BPFBox uses this tracepoint to manage the association of *policy keys* to a particular *process state*. BPFBox policy may optionally specify

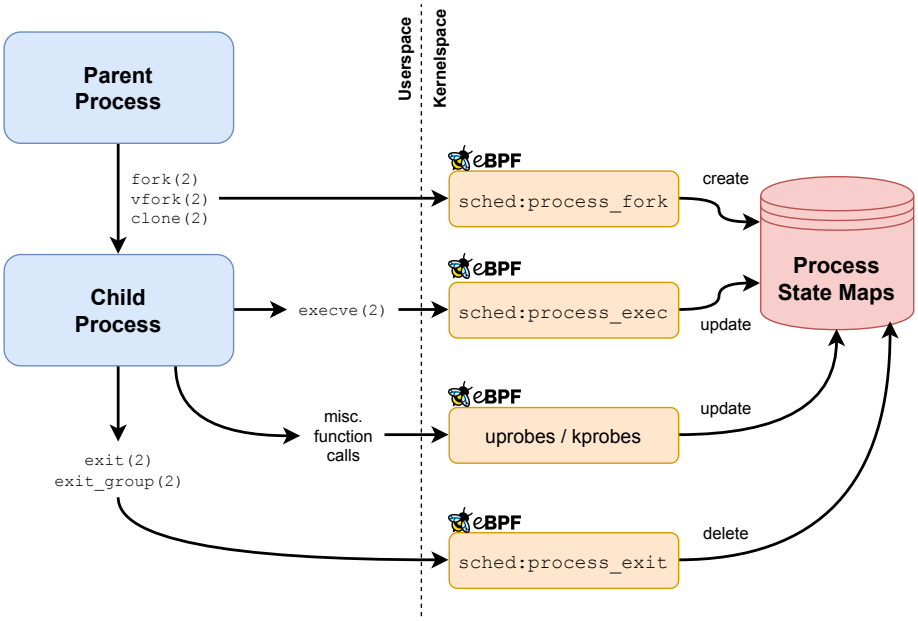

Figure 4.2: The various mechanisms that BPFBox uses to manage process state. Probes marked `sched:*` are tracepoints instrumenting scheduler events in the kernel. Uprobes and kprobes instrument userspace and kernelspace function calls respectively.

whether a transition from one profile to another may occur in a given call to `execve`; this transition is disallowed by default.

Finally, the `sched:process_exit` tracepoint allows BPFBox to detect when a task exits. This tracepoint deletes the corresponding entry in the *process states* map.

4.2.4. Context-Aware Policy

If the policy for a given executable defines specific function call contexts for particular rules, BPFBox instruments these function calls using uprobes (for userspace functions) and kprobes (for kernelspace functions). Each instrumented function call is associated with a unique bit in the process' *state* bitmask. A probe is triggered on entry that causes BPFBox to flip the corresponding bit to a `1`, and again on return, flipping the corresponding bit back to a `0`. Figure 4.3 depicts how BPFBox instruments userspace and kernelspace function calls for policy enforcement.

This approach is subject to a few inherent limitations. Firstly, compile-time optimizations such as function inlining can invalidate the probe by removing the corresponding symbol in the target object file. Secondly, a recursive call that is not tail-optimized will break enforcement by prematurely signalling to BPFBox that a process has exited a given function context. The first limitation may be trivially worked around by hinting to the compiler that a given function should not be inlined; although this sacrifices some application transparency and incurs a slight performance penalty, the potential security benefits from such a fine-grained policy are arguably worth the trade-off. The second limitation *could* be worked around

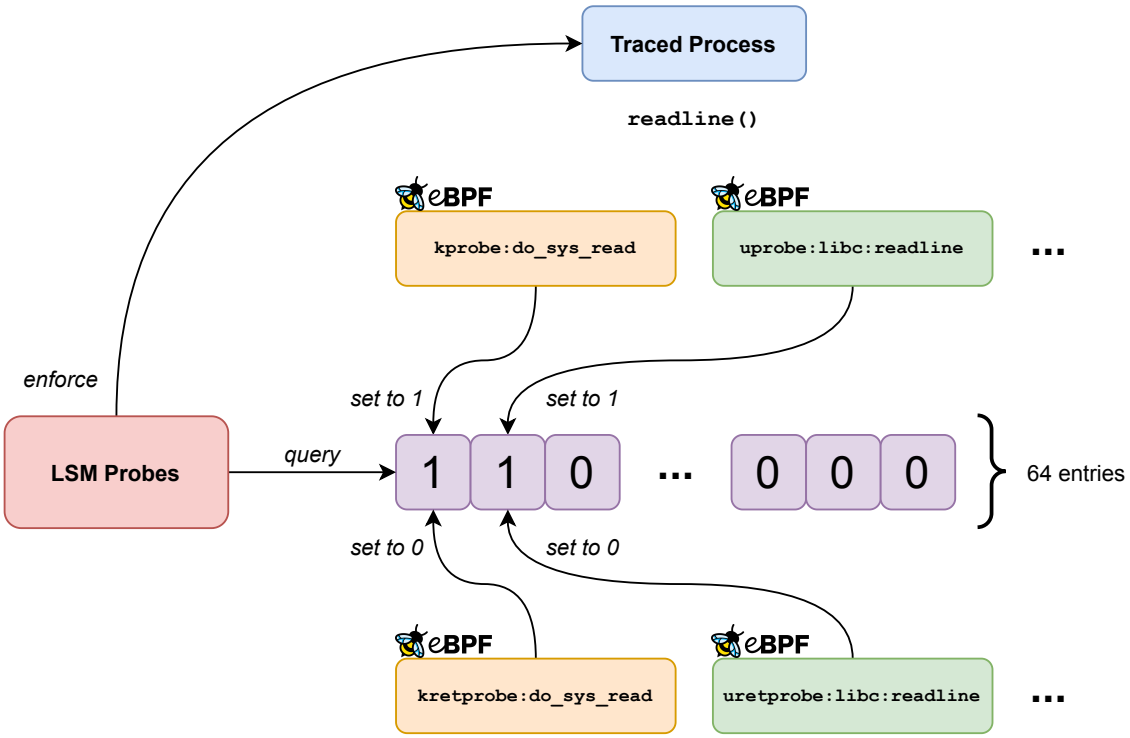

Figure 4.3: How BPFBox uses kprobes and uprobes to track function calls. If a policy identifies that a rule should apply within the context of a certain function call, BPFBox instruments a probe on function entry and return. These probes flip the corresponding bit in the process' state vector, indicating to policy enforcement probes whether or not the process is in the middle of making the function call.

by maintaining a reference counter for each function call rather than a flat vector. BPFBox currently does not do this, since it would incur a larger memory overhead for each active process, but it would be possible to extend a future version of BPFBox with this workaround. In case working around these limitations is impractical, the policy author would simply fall back to specifying ordinary rules rather than context-specific ones.

4.2.5. Collecting and Logging Audit Data

When an operation is denied or matches with an audit rule, BPFBox submits an event to userspace for logging. To accomplish this, we leverage the new ringbuf map type added in Linux 5.8. The eBPF ringbuf map implements an efficient ring buffer that is shared across all CPUs. This new map type comes with a number of optimizations for fast reads and writes and in-order guarantees for asynchronous events across multiple CPUs, allowing per-event data to be efficiently shared with userspace in near real time.

In userspace, the BPFBox daemon uses `mmap(2)` to map the corresponding memory region and polls for new data at regular intervals. As events are consumed in userspace they are removed from the ring buffer to make room for new events. Since the ringbuf map provides strong order guarantees and high performance under contention, we can ensure that BPFBox always provides highly reliable and performant per-event auditing.

4.3. BPFBox Policy Language

BPFBox policies are a series of rules and accompanying decorators. A decorator may annotate either individual rules or blocks of rules denoted by braces and is used to specify additional context or policy actions. The first line in a BPFBox policy is always a special "profile decorator", written as `#![profile "/path/to/exe"]`, which marks the executable to which the policy should be associated. Other than the profile decorator, all others take the form of `#[decorator] { rule() }`. Multiple decorators may be specified before a set of rules, meaning that all decorators apply to each rule.

Profile assignment occurs when a process makes an `execve(2)` call that results in loading the specified executable. Once a process has been assigned a profile, this profile cannot change again, unless an `execve(2)` occurs which has been explicitly marked with the `#[transition]` decorator. This ensures that policy transitions only occur when expected and prevents malicious `execve(2)` calls from changing BPFBox's treatment of a process.

The sections that follow describe the rule categories supported by BPFBox (Sections 4.3.1 to 4.3.4) and the decorators that may optionally be used to augment them (Sections 4.3.5 to 4.3.6). Listing 4.1 depicts a simple example BPFBox policy.

4.3.1. Filesystem Rules

Filesystem rules in BPFBox govern what operations a process may perform on filesystem objects such as files and directories. They are written as `fs("pathname", access)` where `"pathname"` is a string containing the pathname of the file and

Listing 4.1: An example BPFBox policy for a simple remote login program. This example offers a fairly complete idea of the BPFBox policy language's various features.

```
/* This policy applies to the /usr/bin/mylogin
 * executable */
#![profile "/usr/bin/mylogin"]

/* Taint process state upon binding to
 * any IPv4/IPv6 network socket */
#[taint] {
  net(inet,  bind)
  net(inet6, bind)
}

/* Allow network connections/operations */
#[allow] {
  net(inet,  accept|listen|send|recv)
  net(inet6, accept|listen|send|recv)
}

/* Allow the check_login function to read
 * /etc/passwd and /etc/shadow */
#[func "check_login"] {
  fs("/etc/passwd", read)
  fs("/etc/shadow", read)
}

/* Allow the add_user function to read
 * and append to /etc/passwd, but log such
 * events to the audit logs */
#[func "add_user"]
#[audit] {
  fs("/etc/passwd", read|append)
}

/* Read and append to any immediate child
 * of the /var/log/mylogin/ directory */
fs("/var/log/mylogin/*", read|append)

/* Allow the execution of /bin/bash, transitioning
 * profiles to bash's profile after the execve(2)
 * and untainting the process */
#[transition]
#[untaint] {
  fs("/bin/bash", read|exec)
}
```

`access` is a list of one or more file access permissions joined by the vertical bar symbol. For instance, to represent read and append permissions on `/var/log/my_log`, the corresponding BPFBox rule would be `fs("/var/log/my_log", read|append)`. In total, BPFBox supports nine distinct filesystem access flags as shown in Table 4.1.

Table 4.1: The filesystem access flags supported in BPFBox.

Flag	Meaning
`read`	The subject may read the object.
`write`	The subject may write to the object.
`append`	The subject may append to the object.
`exec`	The subject may execute the object.
`setattr`	The subject may change the object's filesystem attributes.
`getattr`	The subject may read the object's filesystem attributes.
`rm`	The subject may remove the object's inode.
`link`	The subject may create a link to the object's inode.
`ioctl`	The subject may perform an ioctl call on the object.

BPFBox supports a limited globbing syntax when defining pathnames, allowing multiple rules matching similar files to be combined into one. Although filesystem rules are specified using pathnames, BPFBox internally uses inode and device numbers rather than the pathnames themselves. When loading policies, BPFBox automatically resolves the provided pathnames into their respective inode-device number pairs. This information is then used to look up the correct policy whenever a sandboxed application attempts to access an inode. Since BPFBox does not check the

pathnames themselves when referring to files, it is able to defeat TOCTTOU (Time of Check to Time of Use) attacks, where an attacker quickly swaps out one file with a link to another in an attempt to circumvent access control restrictions in a privileged (most often setuid) binary [15]. In such a situation, BPFBox would simply see a different inode and deny access.

In addition to regular filesystem rules, BPFBox provides a special rule type for `/proc/pid` entries in the `procfs` virtual filesystem. `procfs` rules, written as `proc("exe", access)` where `"exe"` is a string containing the pathname of another executable and `access` is the desired access. For example, read-only access to the `procfs` entries of executables running `/usr/bin/ls` may be specified with `proc("/usr/bin/ls", read)`. Access to any `procfs` entry may be specified using the special keyword `any`.

4.3.2. Network Rules

BPFBox implements networking policy at the socket level, covering both Internet sockets as well as Unix domain sockets. Networking rules are specified using `net(protocol, access)`, where `protocol` is a networking protocol like `inet`, `inet6`, or `unix` and `access` is a list of socket operations (Table 4.2) separated by vertical bars. For example, a rule targeting `bind`, `accept`, and `connect` operations on an `inet6` socket would look like `net(inet6, bind|connect|accept)`, while a rule targeting `create` operations on a `unix` socket would look like `net(unix, create)`.

Table 4.2: The socket operation flags supported in BPFBox.

Flag	Meaning
`connect`	Subject may connect a socket to a remote address.
`bind`	Subject may bind a socket to a local address.
`accept`	Subject may accept an incoming socket connection.
`listen`	Subject may listen for incoming socket connections.
`send`	Subject may send messages over a socket.
`recv`	Subject may receive messages over a socket.
`create`	Subject may create new sockets.
`shutdown`	Subject may shut down a socket connection.

4.3.3. Signal Rules

Specifying signal behaviour in BPFBox is done using the `signal("exe", access)` where `"exe"` is the pathname of another executable and `access` is a list of signals allowed to be sent, separated by vertical bars. Normally, only processes running the executable `"exe"` are allowed to be signalled, but the special keyword `any` may be used instead to specify the ability to signal *any* process on the system. Two additional keywords, `parent` and `child`, allow parent and child processes to be signalled instead. The `access` argument supports any Linux signal, in addition to a few helper keywords that can be used to specify broad categories, such as `fatal` for fatal signals and `nohandle` for signals that cannot be handled. For example, to specify the ability to send fatal signals to any process running `/usr/bin/ls`, the corresponding BPFBox rule would be `signal("/usr/bin/ls", fatal)`. To narrow permissions such that only `SIGTERM` and `SIGINT` are allowed, `signal("/usr/bin/ls",`

`sigterm|sigint)` could be used instead.

4.3.4. Ptrace Rules

Just like with signals, `ptrace` access is specified as `ptrace("exe", access)`, where `access` is a list of allowed `ptrace` modes separated by vertical bars. The `child` keyword is also available for `ptrace` rules to allow tracing of any child process, regardless of the child's current profile. For instance, a rule that allows a process to read and attach to a child process would be written as `ptrace(child, read|attach)`, while a rule that allows only read access to processes running `/usr/bin/ls` would be written as `ptrace("/usr/bin/ls", read)`. Note that currently `ptrace` rules do not override other `ptrace` restrictions, such as those imposed by Yama [35].

4.3.5. Allow, Taint, and Audit Decorators

BPFBox supports three distinct decorators for defining *actions* that should be taken when a given access matches a rule. The `#[allow]` decorator causes BPFBox to allow the access; however, it is not typically necessary to explicitly specify this as undecorated rules are assumed to be allowed by default. Regardless, it may be desirable to decorate such rules with `#[allow]` to improve the clarity of the policy. `#[taint]` is used to mark a rule as a *taint rule*, which causes the process to enter a tainted state when matched. These rules can be thought of as gateways into the rest of the policy. Once a process is tainted, this cannot be reversed unless it makes an `execve(2)` call explicitly marked with `#[untaint]`. Finally, `#[audit]` may be combined with `#[allow]` to cause BPFBox to log the matching operation to its

audit logs. This can be useful for marking rare behaviour that should be investigated or for determining how often a given rule is matched in practice.

4.3.6. Func and Kfunc Decorators

One of the key features of BPFBox is the ability to specify specific application-level and kernel-level context for rules. In the policy language, this is done by decorating rules with the `#[func "fn_name" ("filename")]` and `#[kfunc "fn_name"]` decorators for userspace and kernelspace instrumentation respectively. Here, `"fn_name"` refers to the name of the function to be instrumented and `"filename"` refers to the filename where the function symbol should be looked up — this parameter is optional and allows for the instrumentation of shared libraries. These decorators provide powerful tools for defining extremely fine-grained, sub-application level policy. For instance, to declare that read access to the file `/etc/shadow` should only occur during a call to the function `check_password()`, the corresponding BPFBox rule would look like:

```
#[func "check_password"]
fs("/etc/shadow", read)
```

A process that is sandboxed using this policy would be unable to access `/etc/shadow` except within a call to the specified function.

4.4. State of the BPFBox Implementation

This chapter has presented BPFBox as it was originally designed. However, some aspects of the BPFBox policy language were never fully realized, despite being implemented on the eBPF side. In particular, decorators to transition profiles and untaint a process when making an execve call are currently unimplemented. Other aspects of the policy language, such as the formulation of ptrace rules, also differ from the current implementation. Before a full implementation could be completed, the transition toward BPFContain (c.f. Chapter 5) had already begun; in our view, BPFContain is a successor to BPFBox, and thus deprecates the original BPFBox implementation. As a result, many of the unimplemented aspects of BPFBox's policy language are reflected in the BPFContain implementation, either through its policy language schema (Section 5.4), or through its default policy enforcement (Section 5.3.4).

4.5. Summary

This chapter has presented the design and implementation of BPFBox, a prototype process confinement mechanism leveraging eBPF for dynamically loadable policies that balance simplicity and flexibility. In particular, we outline BPFBox's architecture, the implementation details of its eBPF-based policy enforcement mechanism, and the design of its custom policy language. Through a combination of eBPF-based enforcement and a lightweight yet fine-grained policy language, BPFBox represents a step towards the container-specific design outlined in Section 3.4. In the next chap-

ter, we outline BPFContain, an iteration of BPFBox that addresses a few of its fundamental limitations and makes a transition toward container-specific policies.

Chapter 5.

BPFContain: Extending BPFBox to Model Containers

In this chapter, we present BPFContain, an iteration on the original BPFBox system presented in Chapter 4. BPFContain is a superset of BPFBox. In particular, it is a streamlined re-implementation that focuses on container-specific confinement policy, low dependency overhead, and maximizing adoptability. Portions of this chapter are taken from an upcoming paper, co-authored with David Barrera and Anil Somayaji, and planned for submission at USENIX Security 2022. A draft of this paper is currently available [58], although significant portions of this chapter differ from the publicly available archive due to subsequent updates to BPFContain.

5.1. BPFBox's Limitations and the Transition Toward BPFContain

The previous chapter presented BPFBox, a prototype process confinement mechanism and precursor to BPFContain. While BPFBox certainly offers a new perspective on confinement and improves the status quo, the extent to which it achieves the design goals outlined in Section 3.4 of Chapter 3 is arguably hampered by a few inherent limitations. We enumerate and describe these limitations as follows. The goal is to examine these limitations as an early motivating factor for the development of BPFContain, which will inform later comparisons between these two systems (c.f. Section 5.5).

1. **Dependency Overhead and Runtime Overhead.** Due to its userspace implementation using bcc, BPFBox has a high dependency overhead. This overhead is the combined result of a number of requirements imposed on the host system by the bcc toolchain. On the userspace side, bcc depends on Python as well as the entire LLVM toolchain for program compilation. Both of these are rather hefty requirements on their own. Python requires an entire language runtime, and a full LLVM toolchain can introduce hundreds of megabytes of additional code (approximately 587MiB when installing LLVM version 12.0 on Arch Linux).

   Furthermore, Python and bcc incur significant runtime overhead. Python is an interpreted language with a much heavier runtime than compiled systems languages like C or Rust. This runtime incurs additional performance disadvantages due to safety features like the global object lock, which impede concurrency. Since

bcc compiles eBPF programs at runtime, we incur additional compilation overhead for each program, sometimes resulting in significant startup delays depending on the complexity of the application. The runtime compilation of eBPF programs also necessitates the availability of kernel headers as a compilation dependency in the target environment, adding further storage overhead (approximately 129MiB for Linux 5.12 on a stock Arch kernel).

2. **Lack of Container Semantics.** Although BPFBox exposes a lightweight policy language with high-level semantics to the user, it fails to consider container semantics, as outlined in design goal 2 in Section 3.4. While this marks an improvement over existing confinement solutions by offering a terse yet expressive policy language, it fails to fully address the container-specific use case; in other words, BPFBox is more suitable to generic, ad-hoc application sandboxing than to container-specific applications. In improving how the BPFBox model handles containers, we can simultaneously simplify policies and improve security by defining a clear protection boundary around a container.

3. **Policy Language Improvements.** In addition to adding container semantics, other aspects of the BPFBox policy language can also be improved and simplified. For instance, BPFBox implements policy in a domain-specific policy language, designed specifically with BPFBox's enforcement engine in mind. While effective, this approach is tightly-coupled with policy enforcement and introduces additional cognitive overhead when making extensions to or modifying the policy language design. Furthermore, learning the syntax of a custom policy language

can introduce an additional barrier-to-entry for new policy authors, to the detriment of BPFBox's original goal of making policy authorship available to end users.

5.1.1. Motivating BPFContain

The key insight behind BPFContain is that BPFBox approached the confinement problem (c.f. Chapter 3) from a *per-process* perspective. When dealing with containers, we should instead approach the confinement problem from a *per-process-group* perspective. That is, we expand the unit of confinement from an individual process to a collection of related processes. Specifically, our goal is to define a clear boundary between the container and the outside world, while minimizing the friction between two subjects that operate within this boundary.

Under BPFBox, policies applied to individual applications, inheriting policy across forks, and selectively transitioning across execves. While this model is effective for application-level confinement, it fails to meet the needs of a container-specific deployment. Conversely, BPFContain expands this model to incorporate container semantics, grouping processes into a container and enforcing a protection boundary around the container. Implementing confinement in this way requires some fundamental changes to both the underlying policy language and policy enforcement mechanism. In particular, we alter the policy language to work with higher level semantics that support container-level confinement. Moreover, BPFContain's enforcement engine employs a more nuanced default policy that considers the relationship between processes and resources that exist within the context of a container.

These changes from a policy and enforcement perspective enable BPFContain to enforce simple container-level policies while reusing the initial ideas from BPFBox: namely, dynamic, lightweight enforcement based on eBPF.

While implementing BPFContain, opportunities arose to improve how it handles dependencies and manages the lifecycle of its BPF programs and maps. Specifically, we architect BPFContain based on Rust, libbpf-rs [155], and BPF CO-RE [102]. These changes totally eliminate the runtime overhead introduced by BPF program compilation and the dependency overhead from LLVM and kernel headers. Further, CO-RE enables BPFContain to work seamlessly across multiple kernel versions and configurations. These changes improve the adoptability of BPFContain, particularly in containerized environments, wherein heavyweight dependencies can critically impact deployments.

Table 5.1 offers a high-level comparison between the properties and features of BPFBox and BPFContain. This comparison provides a high-level overview of the major differences and motivating properties in each design. The sections that follow will discuss each of these items in detail.

Table 5.1: Comparing BPFBox and BPFContain by their properties and how well each satisfies the design goals outlined in Section 3.4.

Property/Goal	BPFBox	BPFContain
Userspace Implementation	Python + bcc	Rust + libbpf-rs
Kernelspace Implementation	bcc + LLVM	BPF CO-RE
Dependencies	$>$ 800MiB	$<$ 5MiB
State Management	Process-Level	Container-Level

Default Policy	Default-Deny	Container Boundary[†]
Policy Language	Custom DSL	YAML/JSON/TOML
Simple/Flexible Policies?	Yes	Yes
Suitable for Containers?	No	Yes
High Adoptability?	Somewhat[‡]	Yes

[†]BPFContain's policy defaults are more nuanced than BPFBox. We enforce a default-allow policy on resources *within* the container, and a default-deny policy on resources *outside* of the container. See Section 5.3.4 for details.

[‡]While BPFBox has a more adoptability than an out-of-tree LSM (due to eBPF), its practical adoption is hampered by heavy runtime dependencies and incompatibility with the container model.

5.2. BPFContain Overview

BPFContain is a container security daemon for Linux with an emphasis on simple, high-level confinement policies for container deployments. Although it is expressly designed to work with container semantics in mind, BPFContain implements a superset of BPFBox's capabilities (c.f. Chapter 4) and works for confining ordinary applications as well as containers. To achieve confinement, BPFContain leverages eBPF programs attached to LSM hooks in the kernel for security enforcement.

As a container-specific confinement solution, BPFContain has a number of important design goals, some of which are shared with BPFBox. First, we seek to design a simple yet flexible policy language that supports ad-hoc confinement use cases, enabling an end user to write a custom confinement policy to suit their needs. We extend this goal by seeking to make the policy language and enforcement engine *container specific*. This means that BPFContain policies should conform to

container semantics and work in tandem with container virtualization primitives to improve policy enforcement and further simplify the underlying policies. An implicit sub-goal of container-specific policy is to improve the level of confinement afforded to container deployments, bringing security more in line with that of alternative isolation techniques, such as virtual machines. Finally, BPFContain should be readily adoptable in production use cases and should be useful for confining existing container workloads.

At the surface level, BPFContain's architecture is similar to that of BPFBox, albeit with significant differences in implementation and design details. In a nutshell, BPFContain is implemented as a privileged userspace daemon that loads eBPF programs and maps into the kernel, which then enforce policy. Section 5.2.1 provides a high-level overview of how BPFContain enforces policy, whereas Section 5.3 covers BPFContain's architecture and implementation details in full.

5.2.1. Policy Enforcement at a High Level

BPFContain enforces confinement policy using eBPF programs attached to LSM hooks in the kernel. Like BPFBox, BPFContain leverages the KRSI [129] LSM programs introduced in Linux 5.7 for this purpose. Unlike BPFBox, BPFContain generates BPF bytecode at *compile time* rather than at runtime. The bytecode is then embedded directly in BPFContain's binary object file, where it can subsequently be loaded into the kernel at runtime. This has the benefit of eliminating initial compile-time overhead and supporting cross-architecture deployments using BPF CO-RE.

To confine a container, an administrator authors a high-level confinement policy that specifies which operating system interfaces and resources should be exposed to the container. Thanks to a modular approach to encoding and decoding policies, BPFContain policies may be written in a number of user-facing data serialization languages, including YAML [56], TOML [113], and JSON [20]. (For the rest of this thesis, we assume the YAML format for simplicity.) At a minimum, BPFContain policies include a policy name, along with a few tunable parameters. From there, the administrator may specify zero or more policy rules over three distinct categories: allow, deny, and taint. We cover BPFContain policy in more detail in Section 5.4.

The BPFContain daemon runs as a privileged process, parsing and loading user policy by encoding it into a series of BPF maps. The user then launches their container using an unprivileged wrapper program, `bpfcontain-run`. The sole task of this wrapper application is to invoke a stub function which does nothing more than pass the desired policy ID as an argument. BPFContain traces this function call and uses it to confine the container with the correct policy. Unlike BPFBox, this technique enables the user to associate any container with any policy, rather than a fixed one-to-one mapping.

At runtime, BPFContain's BPF programs trace the behaviour of processes running under the container and confine it according to a mixture of default policy and policy rules specified by the user. Like BPFBox, enforcement is accomplished primarily through BPF programs attached to LSM hooks in the kernel. The precise implementation details of these programs vary significantly, and are covered in detail in Section 5.3. Figure 5.1 illustrates a high-level overview of the policy enforcement process described here.

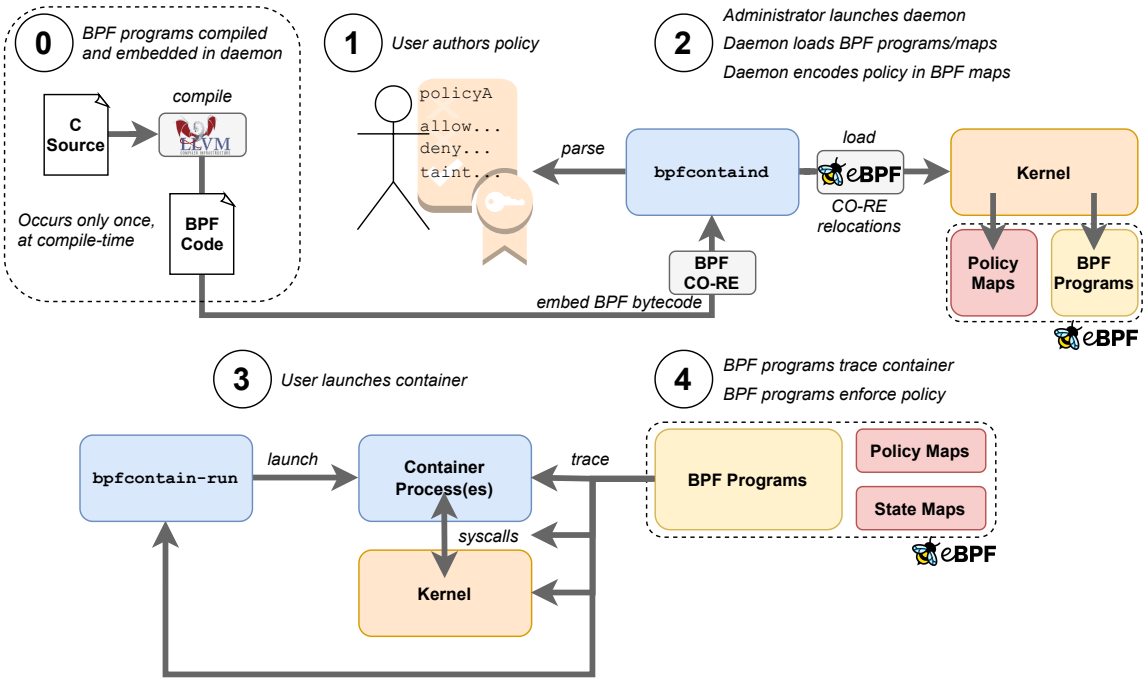

Figure 5.1: A high-level overview of how BPFContain confines containers. At compile-time, the object code for BPFContain's BPF programs is embedded directly into the resulting binary. To confine a container, the user first authors a high-level policy in their chosen data serialization language. The daemon parses this policy and loads it into the kernel by encoding it into eBPF maps. The user then launches the container using the `bpfcontain-run` wrapper, at which point BPFContain begins tracing it and enforcing policy. Note the subtle differences between this figure and Figure 4.1 in Chapter 4.

5.3. BPFContain Implementation

This section presents the implementation details and architecture of BPFContain's policy enforcement mechanism. Specifically, we provide an initial overview of BPFContain's userspace and kernelspace components, then examine how BPFContain enforces policy in the kernel using eBPF. Whereas this section focuses specifically on policy enforcement, Section 5.4 outlines and documents the details of BPFContain's policy language.

5.3.1. Architectural Overview

Like BPFBox, BPFContain is implemented as privileged daemon that runs in userspace and loads eBPF code into the kernel for policy enforcement. However, the precise architecture and implementation details of this daemon are quite different. In particular, the daemon is implemented in Rust and leverages the libbpf-rs crate[1] [155] to load its eBPF programs and maps into the kernel. This results in a number of advantages, which we discuss in more detail in the following section.

BPFContain's kernelspace eBPF programs trace container lifecycle and enforce policy, while eBPF maps store policy and pass intermediary state between program invocations. This architecture is similar in spirit to the design of BPFBox, but with a few fundamental differences. Rather than using the LLVM toolchain to compile programs at runtime, BPFContain pre-compiles and embeds the BPF object code into its binary object file. Using BPF CO-RE [102], these programs can then be

[1] A crate is a Rust package that can be added as a dependency to a project. For the purposes of this thesis, we can consider the terms "crate" and "library" to be equivalent.

dynamically loaded into any supported kernel, regardless of the underlying configuration or architectural details.

BPFContain leverages several BPF program and map types to implement container tracing and confinement. While many of the major program types are shared with BPFBox, there are a few distinct differences (c.f. Section 4.2.1). We enumerate these differences as follows. Map types are outlined in **green** and program types are outlined in **purple**.

Maps:

- BPFContain replaces many of BPFBox's **Hash Maps**, particularly those used to track process state, with equivalent **Local Storage Maps**. Local storage is a new eBPF map type supported in the latest kernels (Linux 5.11 and onwards). Local storage maps tether the underlying value to a kernel data structure, such as a task or inode, used as a key into the map. The result is a dynamically-allocated and garbage-collected per-structure storage blob. BPFContain leverages these for more memory-efficient storage of per-task and per-inode state.

Programs:

- BPFContain replaces the use of scheduler **Tracepoints** with equivalent **LSM Probes** that expose the same information. This reduces potential overhead from multiple BPF program invocations on the same code path, most notably over `fork(2)` and `clone(2)` family system calls.

- BPFContain uses **Fentry and Fexit** probes in place of **Kprobes**. These use a more efficient trampoline technique for program entry and use BTF information exposed by the kernel for direct memory access, making them far more efficient than kprobes[2].

Aside from the aforementioned differences, BPFContain uses the same BPF program and map types as BPFBox. However, the underlying implementation details of each BPF program will be quite different from BPFBox, as BPFContain is dealing with container semantics, new policy rules, and more nuanced policy defaults. We examine the most important implementation details in the subsections that follow.

5.3.2. Policy Deserialization and Loading

When designing the BPFContain policy language, we made a conscious design decision to avoid constraining the user to one specific language syntax. In particular, we wanted to avoid another domain-specific language, as learning the policy language could be a barrier to entry for new users. A domain-specific policy language also presents issues when making changes to or adding new features to the policy language specification, as the parser and lexer must both be modified, along with underlying rule representation and enforcement engine. Instead, we elected to decouple the policy language from the policy specification, using Serde [145], a data serialization and deserialization crate for Rust.

[2]However, the majority of BPFBox and BPFContain's eBPF programs are LSM probes rather than kprobes or fentry probes. As a result, this design change has little consequence on overall performance overhead.

Serde leverages Rust's powerful type system and procedural macros to derive serialization and deserialization logic for vanilla Rust structs and enums. Rust crates that consume Serde's API can then use the automatically generated logic for serialization and deserialization. This design enables a plug-and-play relationship between a data schema, defined as a Rust data structure, and any data serialization language supported through the Rust crates ecosystem. BPFContain uses Serde to automatically generate the accompanying deserialization logic for a `Policy` struct and several `Rule` structs, one for each supported rule type. Listing 5.1 depicts a simplified example of how this works.

Listing 5.1: A simplified example of BPFContain's policy deserialization logic. Policy rules are specified declaratively using Rust structs and the corresponding deserialization logic is automatically generated by the Serde crate using a simple decorator macro.

```rust
use serde::Deserialize;

/// The policy data structure
#[derive(Deserialize)]
pub struct Policy {
  name: String,
  /* Other policy metadata would go here... */
  allow: Vec<Rule>,
  deny: Vec<Rule>,
  taint: Vec<Rule>,
}

/// An enum encompassing all rule types
#[derive(Deserialize)]
pub enum Rule {
  FileRule(FileRule),
  /* Other rule types would go here... */
}

/// A "file access" rule
#[derive(Deserialize)]
```

```
22 pub struct FileRule {
23   pathname: String,
24   access: String,
25 }
26
27 /* Other rule types would go here... */
```

To enable the daemon to encode policy as an eBPF map, each rule type implements the `LoadableRule` trait. The daemon uses this logic to convert a policy rule into a canonical format that can be represented in the kernel and thus used to enforce security policy. Implementing this trait is as simple as writing a `load()` function that makes a series of map updates to load the rule into the kernel; we leverage libbpf-rs [155] for this purpose. When loading a policy into the kernel, the daemon simply invokes this `load()` function for each policy rule.

Implementing policy deserialization and loading logic in this way has a number of advantages. Since the policy schema is simply encoded declaratively in vanilla Rust, it is easy for a developer (even a new contributor to BPFContain) to implement a new rule type and add it to BPFContain. Adding a new rule type is as simple as defining a new Rust data type to represent the rule and implementing the `LoadableRule` trait, enabling the daemon to encode the rule as an eBPF map. Due to Serde's modular design, supporting a new serialization language for BPFContain policies is trivial; we simply pull in the corresponding consuming crate as a dependency. Currently, BPFContain supports YAML, JSON, and TOML as policy language encodings, but this can easily be extended in future versions.

While these conveniences may add some modest performance overhead, this overhead is incurred at policy load time and has no impact on any of BPFContain's

kernelspace code paths. Therefore, we expect the overall impact of this design choice on system performance to be minimal.

5.3.3. Policy Enforcement

Policy enforcement under BPFContain can be thought of as a combination of *explicit policy* (the rules defined in the policy file) and a nuanced *default policy* (the set of sensible defaults that BPFContain enforces to define a boundary around the container). In particular, default access to resources is determined based on whether that resource exists within the context of a container. Resources within the container, such as IPC handles into container processes or filesystems belonging to the container's user namespace mount are considered default allow. Conversely, resources outside of the container, such as external files or processes, are considered default deny. Similarly, access is also denied to any operating system interfaces that could affect global system state, such as character devices, kernel modules, eBPF, and some special filesystems. Exceptions to these defaults may be explicitly defined in the policy file as required. Section 5.3.4 examines the implementation of BPFContain's default policy in more detail.

Like BPFBox, BPFContain maintains its policy files in a root-controlled directory (`/var/lib/bpfcontain/policy` by default). These policy files may be written in any policy language supported by BPFContain's policy deserializer, as documented in Section 5.3.2. The BPFContain daemon watches the policy directory for updates to policy files and triggers a reload of the corresponding policy when a file changes. To load a policy, the daemon deserializes the policy file into a Rust

data structure consisting of a series of policy rules and accompanying metadata. It then encodes this policy structure into a series of resource IDs and access vectors and loads these into the correct policy maps in the kernel using the `bpf(2)` system call. Once a policy has been loaded into the kernel, BPFCONTAIN's eBPF programs can begin enforcing it.

To start confinement, a user invokes an unprivileged application, `bpfcontain-run`, which wraps the target executable. This wrapper's only purpose is to enable BPFCONTAIN's eBPF programs to associate its process group with the correct policy in the kernel. This is done by invoking a stub function, traced by a USDT probe. When the probe fires, it reads the policy ID, passed as an argument to the stub, along with other information such as the task's PID and namespace membership taken from its task struct in the kernel. The probe then updates a global process state map with this information. Subsequent eBPF programs use this information when making enforcement decisions and when managing the container's state. Section 5.3.5 describes state management in more detail.

BPFCONTAIN enforces most policy rules using KRSI [129], which enables it to attach eBPF programs to LSM hooks in the kernel. In cases where LSM hooks alone are insufficient or no LSM hook is exposed to guard the target operation, BPFCONTAIN falls back to an fentry probe, hooking the underlying kernel functions directly. In total, BPFCONTAIN instruments 46 LSM probes, covering filesystem, network socket, IPC, and capability-level access, in addition to miscellaneous privileged operations like loading a kernel module or updating an eBPF program or map. One fentry probe is used to prevent a container from modifying its namespace membership after starting confinement.

BPFContain supports four distinct policy decisions for security-sensitive operations: `allow`, `deny`, `taint`, or `forcequit`. A decision of `allow` causes the access to be allowed, as normal. A decision of `deny` results in the access being denied, and the corresponding system call returning `-EACCES` to the user. A decision of `taint` causes BPFContain to taint the container, a process that is similar in spirit to tainting under BPFBox (c.f. Section 4.2.2 of Chapter 4). When a container is tainted, it transitions into a stricter default-deny policy. Finally, a decision of `forcequit` causes the kernel to terminate the offending process by delivering an uncatchable `SIGKILL`[3]. This decision is reserved for aggressive violations of BPFContain's default policy, such as a process attempting to load code into the kernel (c.f. Section 5.3.4).

When enforcing policy, BPFContain employs a simple heuristic to judge what the resulting policy decision should be. If any rule matches a `deny` decision, the operation is denied. If any rule matches a `taint` decision, the container is tainted. Finally, if no rule matches a `deny` decision and any rule matches an `allow` decision, the operation is allowed. In the case where no rule matches are found, BPFContain falls through to its default policy (Section 5.3.4). This process is depicted in full in Figure 5.2 on page 137.

5.3.4. Default Policy

Since BPFContain is designed to confine containers, it is able to achieve some nuanced policy defaults by leveraging container-level semantics. This marks a significant improvement over both conventional LSMs such as SELinux and AppArmor,

[3] `SIGKILL` is a POSIX signal that causes a process to immediately force quit. Unlike most signals, this signal cannot be ignored or handled by the process.

and the original BPFBox system, which each require the user to either explicitly mark each desired access or to over-generalize access in favour of simpler policies. By taking container semantics into account, BPFContain policies can be simultaneously expressive and secure yet offer a simple path to achieving strong protection defaults. Figure 5.2 depicts BPFContain's default enforcement strategy.

BPFContain's default policy depends largely on the type of access that a container is requesting. If the requested access is to a regular file, BPFContain checks to see if this file exists under the container's user namespace, provided that this namespace is non-global. This covers, for example, a temporary or overlay filesystem mounted within a non-global user namespace. Similarly, default IPC access is gated by whether or not the processes on either end of the IPC handle belong to the same container. If they do, access is granted; otherwise, access is denied. Ptrace is similarly restricted; a process may only ptrace another if both processes exist in the same container. In this way, we preserve the semantics of the container: resources that exist *within* a container are accessible by processes within the same container, but resources that exist *without* are not be accessible by default.

Special files such as character or block devices are treated separately from regular files. Since these provide direct interfaces into the kernel, it does not make sense to treat them with the same semantics. Instead, access to special files is *always* denied, unless they are covered by an explicit `allow` rule. Similar protections are applied to security-sensitive special filesystems such as `procfs`, `sysfs`, `securityfs`, and others. These filesystems are responsible for exposing direct interfaces into the kernel, often with the ability to manipulate behavioural parameters. For these special filesystems, BPFContain also always assumes a default-deny policy, except in cases that are

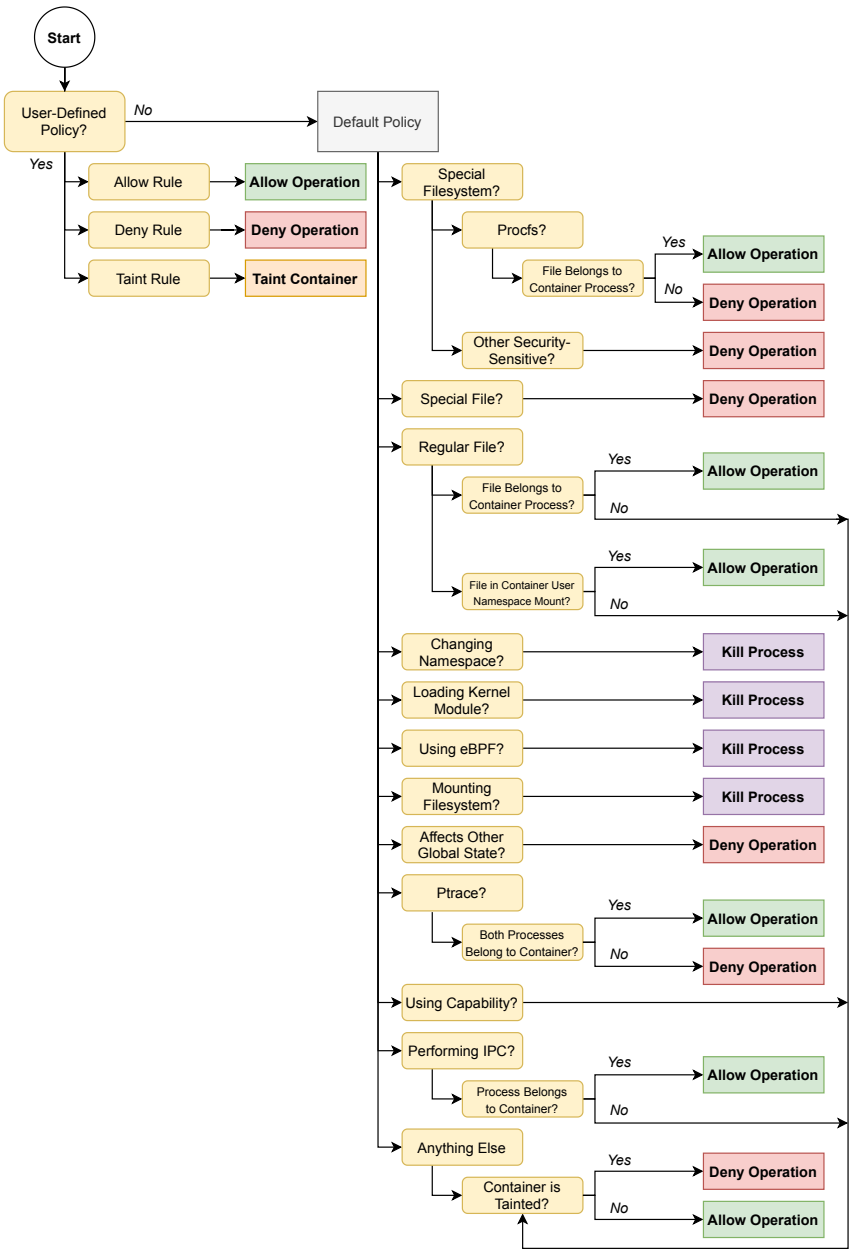

Figure 5.2: The policy enforcement strategy under BPFCONTAIN, expressed as a flowchart. Using container semantics, BPFCONTAIN achieves rich policy defaults, denying access to global resources and operations which can affect global system state while preserving intra-container access. This greatly simplifies the resulting confinement policy and enables the user to focus on specific exceptions to default protection.

explicitly covered by an `allow` rule. The only major exception to this policy is in the case of per-process entries exposed by `procfs`; in this case, BPFCONTAIN assumes default allow provided that the corresponding process is a member of the container.

An implicit assumption underlying BPFCONTAIN's default enforcement strategy is that a container is unable to mutate its namespace membership or manipulate its view of the filesystem in any way. To ensure this, BPFCONTAIN strictly prohibits a container from altering its namespace membership using an Fentry probe on the kernel's `switch_task_namespaces()` function. Similarly, BPFCONTAIN prohibits a container from ever mounting a new filesystem after starting confinement. In both cases, violating this policy results in immediate delivery of `SIGKILL` to the offending process.

Another underlying assumption is that a container cannot interfere with BPFCONTAIN's normal operation. Without this assumption, BPFCONTAIN would be unable to enforce any security guarantees whatsoever, as an attacker could trivially bypass or disable it. To enforce this, BPFCONTAIN prohibits a container from ever loading code into the kernel or interacting with eBPF. This design choice makes practical sense from a container security perspective, as a confined container should not be able to load code into the kernel to begin with — otherwise, escaping confinement would be trivial, as an adversary could simply bypass the protection mechanism enforced by the kernel. As with namespace and mount policy, violating these restrictions results in the immediate delivery of an uncatchable `SIGKILL`.

Some eBPF-based monitoring suites (e.g. Cilium [28] and Tracee [8]) are delivered as containers. Since BPFCONTAIN currently prohibits eBPF usage by a container, it would be a poor choice for confining these suites. Future iterations on BPFCON-

tain may support confining containers with access to eBPF, although this would require BPFContain to protect its own programs and maps using finer-grained policy defaults for the `bpf(2)` code path.

Aside from the aforementioned defaults, BPFContain also denies other miscellaneous operations that can affect global state, including rebooting the system, attempting to modify global system time, access to the kernel keyring, and access to the kernel's perf events subsystem. Using a capability that has not been expressly marked with an `allow` rule is considered default-deny. Any other access is considered default-deny, if the container has been tainted.

Future Improvements to Default Policy

Presently, a major limitation of BPFContain's approach to default filesystem policy is that it relies on a container being in its own user namespace in order for default-allow access to be considered. If a filesystem exists in the container's mount namespace but does not belong to its user namespace, BPFContain must assume default-deny. However, most container management systems, including Docker [50], do not run containers under a new user namespace by default, meaning that BPFContain would be left unable to use its default filesystem policy to grant access. While it is possible to force Docker to run a container in a new user namespace, it would be beneficial if BPFContain's default policy would work regardless of user namespace membership.

There are a few technical challenges surrounding this idea, but it should be possible to achieve in the future once BPFContain has been more integrated with Docker (c.f. Section 8.3.2 in Chapter 8). In particular, we can use eBPF uprobes and kfunc

probes to trace Docker's `containerd` shim as it switches namespaces and mounts the appropriate filesystems. We could then incorporate these filesystems directly into BPFCONTAIN's default policy without relying on user namespace information provided by the kernel. Exploring this option is left as future work.

Another major technical issue underlying BPFCONTAIN's default filesystem policy is that the Linux overlay filesystem currently performs permission checks on the underlying inode (from the original filesystem), rather than the overlayfs inode. This currently makes it impossible to enforce BPFCONTAIN's default filesystem policy on overlay filesystems, which are generally used to implement the majority of a container's filesystem layout. To rectify this, we can leverage more eBPF programs to trace the underlying overlay filesystem operations and temporarily manipulate BPFCONTAIN's state model to account for this. This may add modest overhead to overlayfs operations. Like Docker integration, we leave this for future work.

5.3.5. Managing Container State

Like BPFBOX, BPFCONTAIN tracks the association of processes with policy profiles, along with other state. However, under BPFCONTAIN, this state-tracking happens at the level of individual containers rather than individual processes — we are concerned with *groups of processes*, associated by a shared sense of virtualization and confinement. Tracking state at the level of containers rather than individual processes is a major enabling factor behind BPFCONTAIN's nuanced policy defaults, as described in Section 5.3.4.

To start tracing a container, BPFCONTAIN relies on the `bpfcontain-run` shim

to provide its kernelspace programs with basic information about which confinement policy to associate with the container. Specifically, we care about the desired policy ID, a unique 64-bit integer associated with each BPFCONTAIN policy. In addition to a policy ID, BPFCONTAIN generates a unique container ID for the container, a combination of a 32-bit random integer and the 32-bit process ID of the container's *init* process. BPFCONTAIN maintains a hash map, mapping a container ID to a policy ID, to track the association of a container with a confinement policy.

In addition to policy association, BPFCONTAIN tracks other metadata about the container, including namespace membership, a reference count of how many processes are running under the container, whether the container has been *tainted*, and whether the container is running in *complaining mode*[4]. These metadata are associated with using an eBPF hash map with a simple data structure as a map value and the container ID as the key. Namespace membership is determined at runtime by taking the namespace IDs associated with the container's *init* process' task struct.

To manage container membership, BPFCONTAIN maintains a security blob in each containerized task[5] using a task local storage map. Each task is associated with a given container ID. When a task makes a request to a sensitive resource, BPFCONTAIN makes a chain of map lookups, querying the task's container ID from the local storage map, then using this container ID to query the associated policy ID. When a task forks itself, BPFCONTAIN looks up its container membership and applies the same membership to the child task, incrementing the container's reference

[4]Recall that complaining mode causes BPFCONTAIN to log would-be denials without actually denying the operation.

[5]A task is a Linux kernel data structure that represents a unit of scheduling (i.e. a process or a thread). BPFCONTAIN tracks processes and threads individually at the per-task level.

count. When a task exits, BPFContain simply decrements the container's reference count, cleaning up the container when its reference count reaches zero. Any task-specific metadata is automatically cleaned up by the local storage map.

5.3.6. Collecting and Logging Audit Data

While BPFContain does not expressly define audit rules, it still uses logging to record any policy decisions to a file for subsequent analysis. To achieve this, BPFContain uses the same strategy as BPFBox, relying on a ring buffer map to pass events to userspace for further treatment. Using this ring buffer, BPFContain achieves efficient, in-order event logging across all CPUs. Like BPFBox, BPFContain supports placing a container into a *complaining mode*, enabling it so log denials that would have happened while still granting access. This enables a policy author to test their policy before running it in production and may be used to accommodate log-based policy generation in the future.

5.4. BPFContain Policy Language

This section presents the BPFContain policy language in detail. In particular, we document the policy language schema and offer some insight into how rules can be used to define exceptions to BPFContain's default enforcement. Due to the modular design of BPFContain's policy deserializer, it supports a number of different serialization formats to encode policy. In particular, YAML, TOML, and JSON are currently supported, with the possibility to add others in the future. For the purposes of this section, we assume the YAML format for consistency and readability.

BPFContain policies are stored in a central, root-controlled directory. At runtime, the daemon watches policy files for changes and parses and loads the policy into the kernel when updates occur. At a minimum, each BPFContain policy contains some metadata, including the policy name and a few tunable parameters. Tunables include the ability to mark a container as pre-tainted and the ability to specify a command to use as the default entry point for `bpfcontain-run`. A pre-tainted container spawns tainted rather than untainted, falling back to stricter defaults when no rule matches the requested access (c.f. Figure 5.2 on page 137).

Aside from policy metadata, the policy is divided into three sections: allow, deny, and taint. Each of these sections specifies the corresponding policy decision for any rules declared within. When the BPFContain enforcement engine matches on a rule, it takes the rule's policy decision as an enforcement action. In turn, policy rules are divided into several categories based on the type of access that they specify. The subsections that follow examine and document each supported rule category.

Note that, unlike BPFBox, BPFContain does not currently support the ability to define function-level policy. The rationale for this design choice is that hyper-fine-grained policy makes little sense in the context of a container, particularly considering BPFContain's highly-nuanced policy defaults. Future work may involve examining this design choice, along with other aspects of the BPFContain policy language design, in the context of a user study (see the discussion in Chapter 8). Listing 5.2 depicts an example BPFContain policy for a simple remote login program.

Listing 5.2: An example BPFContain policy for a simple remote login program, written in YAML. This example offers a fairly complete idea of the BPFContain policy language's various features. The reader is encouraged to compare this policy with the policy depicted in Listing 4.1 on page 111.

```yaml
# Name of the policy
name: mylogin
# Container entrypoint
cmd: /usr/bin/mylogin
# Spawn container untainted
defaultTaint: false

allow:
  # Perform send/recv operations as a client
  - net: [client, send, recv]
  # Grant read and append access to /etc/passwd
  - file: {pathname: /etc/passwd, access: ra}
  # Grant read-only access to /etc/shadow
  - file: {pathname: /etc/shadow, access: r}
  # Grant read and append access to any immediate child of
    /var/log/mylogin
  - file: {pathname: /var/log/mylogin/*, access: ra}
  # Grant read and execute access to bash
  - file: {pathname: /bin/bash, access: rx}
  # Grant read/write access to the TTY
  - dev: terminal

taint:
  # Taint after performing any network operation
  - net: any
```

5.4.1. File and Filesystem Rules

For specifying access to regular files, BPFCONTAIN supports two major rule types. *File rules* specify access at the granularity of individual files while *filesystem rules* specify access at the granularity of a filesystem. These may be combined to grant or restrict coarse-grained access to entire filesystems and define fine-grained exceptions to this coarse-grained access for specific files. These rules are necessary since not every BPFCONTAIN policy targets an application running in a container, and containers often access files directly from the host filesystem (e.g. through a Docker volume mount). Each file and filesystem rule consists of a pathname and an access pattern. In the case of filesystem rules, the given pathname must be the mountpoint of the filesystem. BPFCONTAIN supports several access flags for fine-grained control over file access. Table 5.2 describes each flag and its corresponding effect.

Table 5.2: File access flags in BPFCONTAIN.

Pattern	Access
`r`	Read (`read(2)`, `getattr(2)`, etc.)
`w`	Write (`write(2)`, `setattr(2)`, etc.)
`a`	Append (`write(2)` with append-only flag set)
`x`	Execute (`execve(2)`)
`m`	Map executable memory (`mmap(2)`)
`c`	Modify Unix DAC (`chmod(2)`/`chown(2)`)
`d`	Unlink/delete a file
`l`	Create a hard link to a file
`i`	Make an `ioctl(2)` call on a device

When loading a file rule into the kernel, BPFCONTAIN translates the pathname into a list of tuples uniquely describing the file. Each tuple contains the file's inode number along with the unique device ID associated with the filesystem on which the inode resides. These two numbers taken together can uniquely identify any file on the system. BPFCONTAIN's file rules similarly take the device ID of the filesystem root, ignoring the inode. An implicit side effect of this technique is that files are immutably resolved at policy load time, meaning that BPFCONTAIN can achieve pathname resolution without becoming vulnerable to TOCTTOU attacks. Like BPFBOX, BPFCONTAIN deals with newly-created files by associating them with the task that created them, granting default access to these files for the owning task — this resolves the use case where a task requires access to a file created *after* its policy has already been loaded.

5.4.2. Device Rules

Unlike BPFBOX, BPFCONTAIN takes great care to avoid conflating regular files and special files. The key insight underlying this design choice is that the semantics of regular files and special files are quite different, despite supporting fundamentally the same operations. Provisioning over-permissive access to the wrong special file (e.g. `/dev/mem`, which provides access to the system memory map) can have devastating security consequences. For this reason, access to a special file must be specified via a *device rule* using the **`dev`** keyword, rather than the **`file`** or **`filesystem`** keywords.

BPFCONTAIN supports several major classes of character device, each with a default access pattern according to the device's semantics. For instance **`dev:`** `terminal`

enables read and write access on `/dev/tty` to support standard input and output to the terminal. Likewise, `dev: random` grants read only access to `/dev/random` and `/dev/urandom`. When loading a device rule into the kernel, BPFContain resolves the device's major and minor number pair and maps it to the corresponding access pattern.

More nuanced device access patterns may be specified using a numbered device rule, specified as `numberedDev: {major: major, minor: minor, access: access}` where *major* and *minor* are the device's major and minor number, and *access* is an access flag pattern. This access pattern uses the same file access flags as outlined in Table 5.2. The minor number is optional and may be omitted to match *any* device of the specified major number. Note that modern kernels dynamically allocate their major and minor number, meaning that it is possible for BPFContain to lose track of the association between these numbers and the underlying device driver. We acknowledge this limitation in Section 6.2 and describe how the BPFContain prototype can be trivially modified to address it.

5.4.3. Network Rules

BPFContain simplifies BPFBox's network policy by categorizing network accesses into high-level use cases rather than the underlying socket operations themselves. This approach is informed by the insight that specific applications tend use specific sets of socket operations, depending on if the application is designed as a client, a server, or some combination of the two (e.g. a peer-to-peer model). Specifically, a server would require the ability to create sockets, bind them to an IP address,

listen for and accept incoming connections, and shut down existing connections. Conversely, a client generally needs to connect to an existing bound socket. We further partition access by provisioning send and receive access separately. Table 5.3 provides an overview of these access categories.

Table 5.3: Network access categories in BPFContain. By combining the `client` or `server` keywords with the `send` and `recv` keywords, a policy can specify the correct level of access to required TCP socket operations.

Category	Access
`server`	Create, bind, listen, accept, and shut down socket connections
`client`	Connect to a bound socket
`send`	Send data over the socket
`recv`	Receive data over the socket

BPFContain's network policy covers IPv4 and IPv6 sockets. Netlink and raw packet sockets are prohibited by default, and Unix domain sockets are relegated to IPC rules rather than network rules (c.f. Section 5.4.4).

5.4.4. IPC Rules

In general, container IPC policy is handled by BPFContain's default enforcement, which permits IPC between two processes provided that they belong to the same container. All other instances of IPC are denied by default. In cases where inter-container IPC is required, BPFContain provisions IPC rules which are defined as `ipc: name` where *name* is the name of another BPFContain policy. In order for inter-container IPC to be allowed, both policies must mutually grant each other

IPC access. This ensures that any communication between containers is mutually authorized, preventing attackers from bypassing the security assumptions of a policy. BPFContain's IPC rules cover all canonical forms[6] of IPC available on the system, including signals, System V IPC objects, and Unix domain sockets.

5.4.5. Capability Rules

Since container execution models can be (and often are) privileged by default, BPFContain takes great care to be distrustful of any POSIX capabilities assigned to the container. Specifically, BPFContain denies the use of *any* POSIX capabilities as part of its default policy. While this is a simple and highly effective strategy for limiting the privileges of containers running under root's UID, some container use cases require additional privileges to correctly function. To accommodate these use cases, BPFContain provisions a *capability rule* which can be used to specify allowed capabilities. The capability rule is specified using `capability: [capabilities...]` where *capabilities* is a list of POSIX capabilities.

Note that capability rules *do not grant* additional capabilities to a container; rather they are a mask over the set of all capabilities that a container can ever possess. In particular, this means that a container must already have the corresponding capability under the traditional POSIX capabilities model in order to be able to use it. Thus, capability rules merely add an extra level of protection on top of the existing model, preventing overprivilege by restricting the bounding capability set.

[6] However, BPFContain does not currently support fine-grained access control over TCP/IP sockets. This is left as future work (see Section 8.3).

5.5. Improvements Over BPFBox

As a successor to BPFBox, BPFContain makes several fundamental improvements in terms of dependency overhead, policy language simplification, and container-specific extensions. This section summarizes some of these improvements in light of the implementation details discussed earlier in this chapter.

5.5.1. Minimizing Runtime Dependencies

BPFContain solves BPFBox's dependency and runtime overhead issues by leveraging Rust and libbpf CO-RE [102] rather than Python and bcc. Unlike bcc, libbpf CO-RE enables BPF programs to be compiled once and run anywhere, thanks to BTF information provided by the kernel and load-time symbol relocation. Program bytecode can then be embedded directly into the compiled object file, meaning the single pre-compiled BPFContain binary can be deployed on any target kernel that meets a minimal set of requirements. As a side effect, BPFContain requires neither a full LLVM toolchain nor kernel headers to be available in the target deployment.

Moreover, implementing the BPFContain daemon in Rust allows BPFContain to take advantage of a myriad of benefits offered by the Rust language. In particular, Rust enables BPFContain's userspace components to be safe, secure, and fast. Thread and memory safety guarantees provided by Rust ownership model eliminate many common security bugs including memory corruption vulnerabilities and race conditions between threads. These safety guarantees provide critical security advantages, particularly given the fact that the BPFContain daemon is a long-running, privileged process — a ripe target for attacker exploitation. Thanks to an emphasis

on speed and zero-cost abstractions, Rust can provide these benefits at virtually zero overhead, in line with traditional systems programming languages like C and with significantly smaller overhead than interpreted languages such as Python.

5.5.2. Improved Policy Language

BPFContain greatly simplifies the original BPFBox policy language. Rather than defining a specific policy language syntax, BPFContain defines a schema that can be encoded in multiple different data serialization languages. This simultaneously enables BPFContain to support a policy language that users are already familiar with (e.g. YAML) and provides a clear path for extending BPFContain to support additional policy languages in the future. Further, this approach presents an opportunity for integrating BPFContain policy with existing specifications, such as the OCI (Open Container Initiative) specification or the rego [109] policy framework, both of which are encoded in JSON. Integrating with the OCI specification will enable BPFContain policies to be specified directly within container manifests. Integrating with rego would enable BPFContain policies to interact with the Open Policy Agent, widely used to implement policy in the Kubernetes container orchestration framework.

Further simplifications to the BPFContain policy language are afforded by its goal of container-specific confinement. By focusing on container-specific use cases, BPFContain's default policy enforcement can be far more nuanced than a traditional sandboxing framework. This property enables the user to focus on defining specific exceptions to a well-defined security boundary rather than enumerating every

single possible resource that a container can access. Along with the simplifications afforded by BPFContain's sensible policy defaults, we make additional changes to the policy language that help decouple it from the underlying details of the operating system, such as higher-level network policy and semantically-guided device driver access defaults.

Moreover, BPFContain improves upon the original BPFBox policy language design by introducing new rule types to cover weaknesses in the original design. It also fully implements many aspects of BPFBox that were left unfinished in the current implementation, providing a more fully realized prototype. For these reasons, BPFContain deprecates BPFBox by implementing a superset of its original functionality and improving upon flaws in the original BPFBox design.

5.5.3. Container-Specific Extensions

Perhaps the most significant improvement over the original BPFBox design is that BPFContain implements container-specific confinement. Whereas BPFBox is well suited to fine grained, process level confinement, BPFContain extends this design to model containers. In particular, BPFContain tracks namespace membership as well as the association between processes and containers, enabling access to resources within a container and restricting access to the outside world. This approach is similar in spirit to FreeBSD Jails [79]. Unlike Jails, however, the BPFContain implementation applies such container specific defaults without any changes to the upstream kernel.

BPFContain's container-specific extensions enable BPFContain to enforce

container-level policy with a well-defined security boundary around the container, simultaneously improving security and greatly simplifying the resulting policies. Rather than focusing on each and every resource associated with the container, security policies can instead focus on defining exceptions in BPFCONTAIN's security boundary, resulting in policies that closely mirror the exposed interface to the outside world.

5.6. Summary

This chapter has presented the design and implementation of BPFCONTAIN, an extension on top of the original BPFBOX design that promotes container-specific confinement and uses container semantics to simplify policies while providing strong security guarantees. Using eBPF, BPFCONTAIN supports container-level semantics in a kernel-level enforcement engine without sacrificing adoptability or tying the kernel down to a specific definition of a container. BPFCONTAIN uses these container-level semantics to define a clear protection boundary around containers and provides a simple policy language for defining exceptions to this protection boundary.

Chapter 6.

Evaluation

This chapter presents an evaluation of BPFBox and BPFContain in terms of their performance and security. Section 6.1 presents the methodology and results of a performance evaluation involving micro- and macro-benchmarking of BPFBox and BPFContain. Results are compared with AppArmor [42], a popular LSM framework for MAC security policy. Finally, Section 6.2 presents a security analysis of BPFBox and BPFContain under the threat model outlined in Section 3.6.

6.1. Performance Evaluation

This section presents a performance evaluation of BPFBox and BPFContain, measuring their performance overhead using a variety of micro- and macro-benchmarking tests. In particular, we use the Phoronix Test Suite [83] to measure overhead across a variety of computational tasks, workloads, and kernel interfaces. Each of these benchmarks exercises a different subset of BPFBox and BPFContain's enforce-

ment engine, providing an approximation of their impact on the overall system. We also measure the performance of the base system as a control and the performance overhead of AppArmor as a basis for direct comparison. The subsections that follow provide an overview of our testing methodology and present the benchmark results.

6.1.1. Methodology

As a test environment, we utilize a bare-metal system running Arch Linux with a stock 5.12.14-arch-1-1 kernel. The choice of a bare-metal system (rather than a virtual machine, for instance) reduces the risk of introducing additional sources of variance into the benchmarks. Table 6.1 provides a detailed account of the test system configuration.

Table 6.1: System configuration for benchmarking tests.

Item	Description / Configuration
CPU	Intel i7-10875H; 8 cores, 16 threads at 2.3GHz; 16MB cache
GPU	Nvidia RTX 2060 with 6GB GDDR6 VRAM
RAM	2×16GB DDR4 at 3.2GHz
Disk	1TiB Samsung NVME M.2 SSD
Motherboard	System76 oryp6
OS	Arch Linux (Rolling)
Kernel	Linux v5.12.14-arch-1-1
Libc	glibc v2.33-5
Phoronix	v10.4.0-1

To simulate the Docker container use case, we run all tests in a privileged Docker container, using Docker volumes to mount the host filesystem in the benchmarking

directory. To improve benchmarking accuracy, we also perform the following setup before each test. (1) We disable SMT hyperthreading by turning off each logical CPU core pair, leaving only the physical cores active; (2) We disable turbo boost, capping the CPU at its stock speed of 2.3GHz; (3) We set the CPU frequency scaling governor to "performance" to limit the impact of thermal throttling and power saving features; and (4) We globally disable ASLR by setting the appropriate kernel parameter. These settings, consistent with best practices, improve benchmark accuracy by making the environment more consistent and eliminating as many external factors as possible.

Table 6.2: A list of the benchmarking suites used to test performance overhead, and what each measures.

Test Suite	Test	Measures
OSBench	Create Files	Time to create and delete files
	Create Threads	Time to create new threads
	Launch Programs	Time to fork + execve
	Create Processes	Time to create new processes
	Memory Allocations	Memory allocation throughput
Kernel Compilation	—	Time to compile Linux Kernel
Apache Web Server	—	Apache HTTP request throughput

To measure the performance overhead of BPFBox and BPFContain (compared with the base system and with AppArmor) we use the Phoronix Test Suite [83], a popular cross-platform benchmarking framework that has seen wide use for measuring system performance. The Phoronix framework comprises a number of open source test suites, each targeting a different aspect of system behaviour. For the purposes of this thesis, we select three separate test suites, measuring a variety of

OS-level functionality and exercising multiple LSM hooks. In particular, we select the OSBench suite, the Kernel Compilation suite, and the Apache suite. Table 6.2 describes each test suite and what it measures.

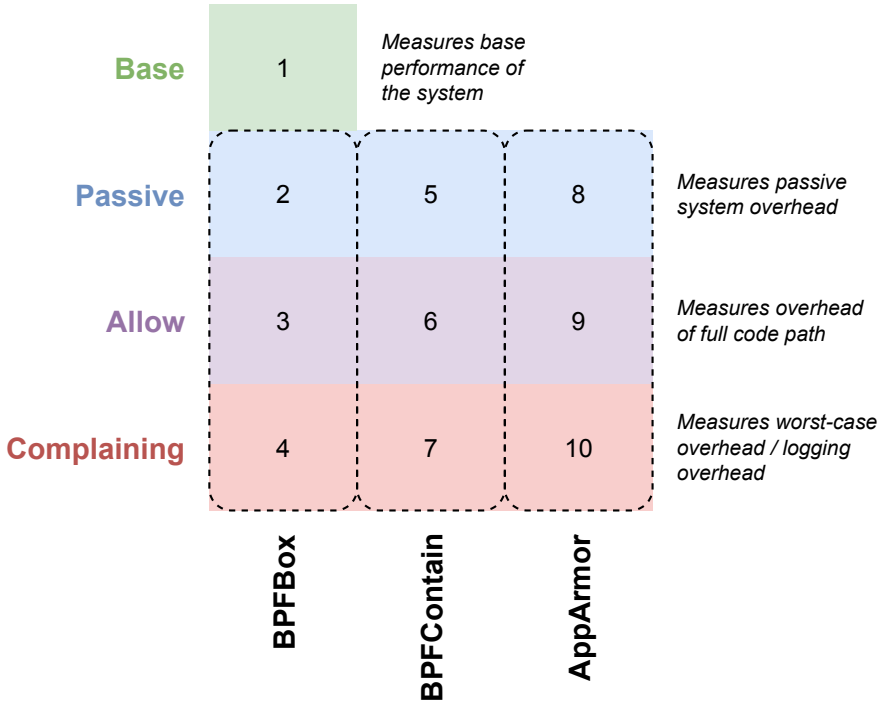

Figure 6.1: The various system configurations used in the benchmarking tests. Each numbered cell constitutes one configuration, for a total of ten.

We consider ten system configurations in total (Figure 6.1). The **Base** configuration is the base system without any LSMs or other confinement primitives active or loaded in the kernel. The **BPFBox**, **BPFContain**, and **AppArmor** configurations measure the performance overhead of BPFBox, BPFContain, and AppArmor, respectively. We then divide each of these three configurations into three distinct test cases each. The **Passive** case measures global system overhead without

any active enforcement. The **Allow** case measures active enforcement, allowing all security-sensitive operations. Finally, the **Complaining** case measures the worst-case overhead for each system, exercising the full code path of each LSM hook and logging every attempted access.

To calculate percent overhead for each test configuration, we take the mean of all test results for a given configuration and calculate the percent change from the base configuration. This is done using the following formula:

$$\text{Percent Overhead} = \frac{(\bar{x}_{test} - \bar{x}_{base})}{|\bar{x}_{base}|} \times 100$$

To ensure statistically valid results, we run each test at least eleven times, until a standard deviation of at most 2% of the mean is achieved. The standard deviation bound is a sensible default enforced by the Phoronix Test Suite to ensure statistically valid results. We also discard the first run of each test to control for initial I/O transients. In the end, no additional trials were necessary, as we were able to achieve under 2% standard deviation for all test configurations. For reproducibility, we make the benchmarking repository publicly available[1], including all results and related scripts.

6.1.2. Results

This section presents the results of the OSBench micro-benchmarks (Figure 6.2 and Tables 6.3 to 6.7), the kernel compilation (Figure 6.3 and Table 6.8) and Apache web server (Figure 6.4 and Table 6.9) macro-benchmarks. We find that BPFBox and

[1]Benchmarking tests are available: https://github.com/willfindlay/bpfcontain-benchmarks

BPFContain incur modest overhead in many common use cases cases, with BPFContain experiencing performance degradations in some cases. Also, we discuss how future optimizations to BPFContain and the KRSI framework could greatly improve its performance overhead in practice.

OSBench File Creation

The file creation benchmark (Table 6.3 and Figure 6.2) indicates that BPFBox and BPFContain have significantly higher overhead than AppArmor in the **Passive** and **Allow** cases. BPFContain, in particular, performs the worst out of the three systems in these two cases. It is perhaps unsurprising that BPFContain performed worse on this test, since it performs complex analysis on filesystem operations to come to a policy decision. We made no deliberate optimization attempts in this research prototype, but expect that future performance improvements are possible. Conversely, AppArmor is a well-established security mechanism which has undergone significant performance optimizations over time. Future optimizations on BPFContain could likely improve its performance overhead in practice. Despite a seemingly high performance overhead in the **Passive** and **Allow** cases, BPFBox and BPFContain incur a performance penalty of under 12μs each. The kernel compilation macro-benchmarks, presented later in this section (c.f. Figure 6.3 and Table 6.8) indicate that this slowdown has very little effect on even moderately complex workloads. Moreover, in the **Complaining** case, BPFBox and BPFContain significantly outperform AppArmor. This result can be attributed to implementation differences in their event-logging mechanisms.

In the **Passive** case, BPFBox and BPFContain's high performance overhead

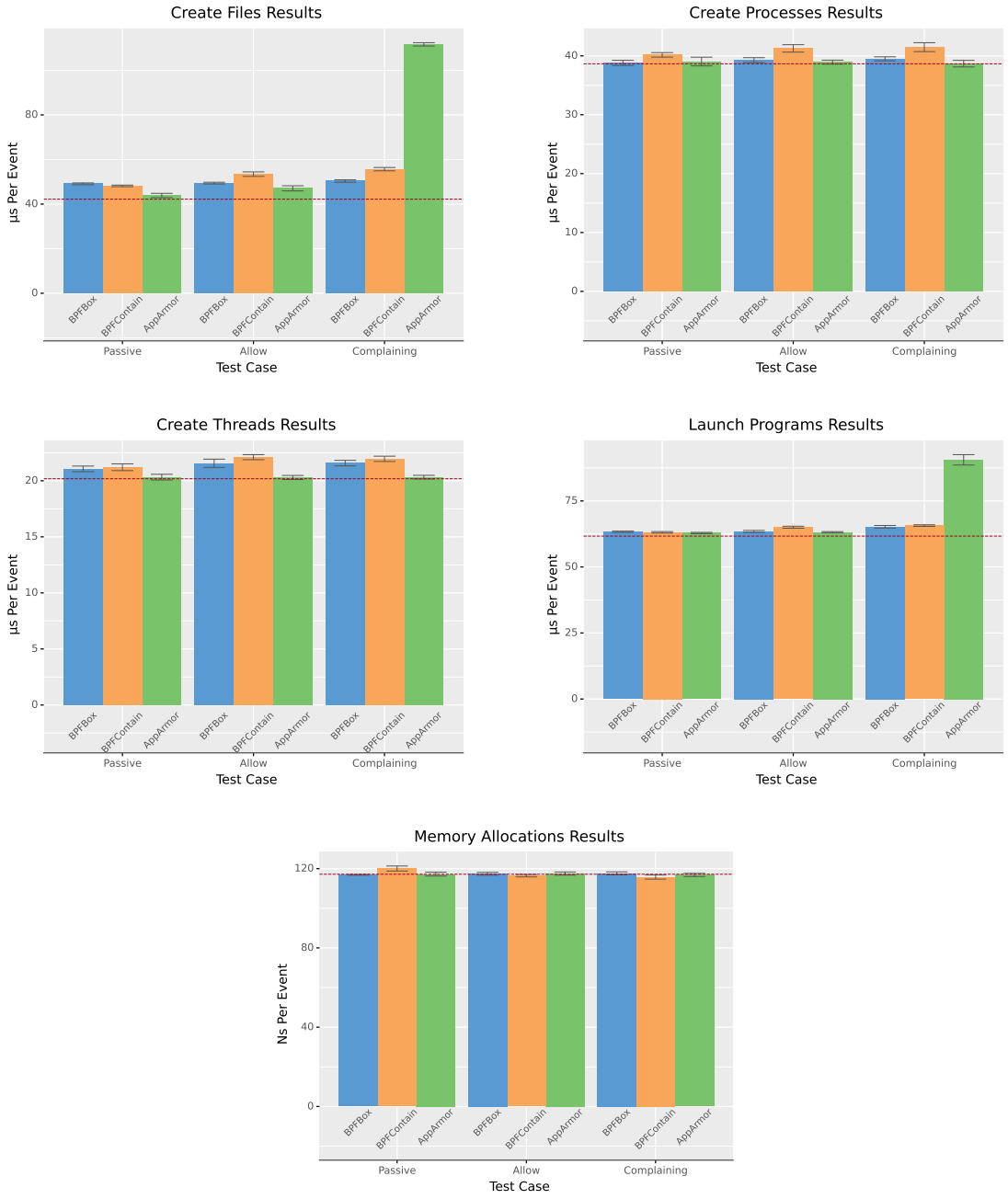

Figure 6.2: The results of the OSBench micro-benchmarks. The error bars show standard deviation and the red lines show base measurements for each test. Lower times are better.

Table 6.3: Results of the Create Files benchmark. Units are μs per event. Lower is better. Percent overhead is compared to the baseline result.

Test Case	System	Mean	Std	Overhead
Base	—	42.21	1.22	—
Passive	BPFBox	49.09	0.40	16.31%
	BPFContain	48.16	0.36	14.11%
	AppArmor	43.87	0.95	3.93%
Allow	BPFBox	49.41	0.39	17.08%
	BPFContain	53.43	1.01	26.60%
	AppArmor	47.07	1.15	11.52%
Complaining	BPFBox	50.34	0.54	19.27%
	BPFContain	55.67	0.75	31.89%
	AppArmor	111.66	0.75	164.55%

can be attributed to the fact that they each invoke multiple BPF programs over multiple LSM hooks on the `open(2)`, `write(2)`, and `unlink(2)` code paths. Unlike AppArmor, BPFBox and BPFContain invoke a new eBPF program on every LSM hook along this code path and then perform a map lookup to determine whether the process is being actively traced. The overhead associated with this many BPF program invocations is non-trivial compared with the overhead of simply calling into an LSM hook. Future improvements to the KRSI framework may also be able to reduce the performance overhead of BPF LSM programs.

In the **Allow** case, BPFBox is more in line with AppArmor, while BPFContain is shown to exhibit a slightly higher overhead. We can attribute the additional overhead shown by BPFContain to the nuances associated with its code path for

file and filesystem policies. For each file operation, BPFContain performs multiple map queries and reads information from multiple kernel data structures to enforce its default policy. Future iterations of BPFContain may improve this overhead by caching policy decisions for filesystem objects and/or resolving inefficiencies in how BPFContain reads information from kernel data structures.

In the **Complaining** case, BPFBox and BPFContain significantly outperform AppArmor, a fact which can be attributed to inefficiencies in AppArmor's logging mechanism, which relies on the kernel's audit framework. The ring buffer maps used by BPFBox and BPFContain are known to exhibit comparatively less overhead [103, 158, 159]. Additional overhead may also arise due to differences in how AppArmor translates files and access patterns to log messages.

OSBench Process Creation

The results of the process creation benchmark (Table 6.4 and Figure 6.2) indicate that BPFBox and BPFContain introduce modest overhead on top of the `fork(2)` system call. Comparatively, AppArmor introduces very little overhead, well within the margin of error for measurements. The additional overhead introduced by BPFBox and BPFContain can be explained by the additional per-process and per-thread accounting performed by each system. BPFContain, in particular, handles a significant amount of per-process and per-thread metadata, which must be populated each time a `fork(2)` or `clone(2)` occurs and cleaned up each time a process exits. However, it should be noted that both BPFBox and BPFContain introduce less than 10% overhead along this code path (within 1–2μs), which should be imperceptible in practice.

Table 6.4: Results of the Create Processes benchmark. Units are μs per event. Lower is better. Percent overhead is compared to the baseline result.

Test Case	System	Mean	Std	Overhead
Base	—	38.65	0.36	—
Passive	BPFBox	38.81	0.44	0.40%
	BPFContain	40.17	0.39	3.93%
	AppArmor	39.04	0.74	1.01%
Allow	BPFBox	39.28	0.41	1.63%
	BPFContain	41.27	0.63	6.77%
	AppArmor	38.94	0.34	0.74%
Complaining	BPFBox	39.51	0.33	2.22%
	BPFContain	41.49	0.76	7.33%
	AppArmor	38.68	0.56	0.07%

OSBench Thread Creation

The thread creation results (Table 6.5 and Figure 6.2) are directly related to the process creation results, insofar as both operations exercise the same BPF programs in BPFBox and BPFContain. Since thread creation is faster then process creation, the percentage overhead of BPFBox and BPFContain appear comparatively higher, but the underlying delta is the same, at roughly 1–2μs per event. Despite these differences in thread and process creation speeds, the resulting percentage overhead of BPFBox and BPFContain is still under 10%.

Table 6.5: Results of the Create Threads benchmark. Units are μs per event. Lower is better. Percent overhead is compared to the baseline result.

Test Case	System	Mean	Std	Overhead
Base	—	20.18	0.19	—
Passive	BPFBox	21.06	0.25	4.37%
	BPFContain	21.21	0.30	5.08%
	AppArmor	20.32	0.25	0.71%
Allow	BPFBox	21.56	0.37	6.84%
	BPFContain	22.11	0.22	9.53%
	AppArmor	20.29	0.18	0.54%
Complaining	BPFBox	21.57	0.25	6.90%
	BPFContain	21.96	0.24	8.80%
	AppArmor	20.32	0.16	0.70%

OSBench Program Launching

The launch programs benchmark (Table 6.6 and Figure 6.2) is essentially the same as the process creation benchmark (c.f. Table 6.4), with one major difference: the addition of an `execve(2)` call after the `clone(2)` system call. This `execve(2)` call adds a constant overhead of about 20μs on top of the original process creation results, as well as additional LSM hook invocations along the `execve(2)` code path. These factors contribute to BPFBox and BPFContain performing slightly worse than AppArmor in the **Passive** and **Allow** cases and significantly better than AppArmor in the **Complaining** case.

The additional LSM hook invocations caused by the `execve(2)` severely impact AppArmor's performance in the **Complaining** case for the same reasons as dis-

cussed in the file creation results (c.f. Table 6.3). BPFBox and BPFContain exhibit comparatively little overhead despite the `execve(2)` call. This result can be explained by the fact that `execve(2)`'s code path invokes significantly fewer LSM hooks than the file creation and deletion code paths we examined earlier. In all test cases, BPFBox and BPFContain are able to achieve under 7% overhead in the worst case and under 3% in the majority of cases.

Table 6.6: Results of the Launch Programs benchmark. Units are μs per event. Lower is better. Percent overhead is compared to the baseline result.

Test Case	System	Mean	Std	Overhead
Base	—	61.67	0.20	—
Passive	BPFBox	63.30	0.28	2.64%
	BPFContain	63.12	0.28	2.34%
	AppArmor	62.84	0.25	1.89%
Allow	BPFBox	63.44	0.40	2.86%
	BPFContain	65.05	0.38	5.47%
	AppArmor	63.17	0.21	2.43%
Complaining	BPFBox	65.22	0.48	5.75%
	BPFContain	65.66	0.30	6.47%
	AppArmor	90.56	1.97	46.83%

OSBench Memory Allocations

The memory allocation benchmark (Table 6.7 and Figure 6.2) indicates that none of the systems had any significant affect on memory allocation. In some cases, percent overhead falsely appears to indicate a performance *improvement*, which we

attribute to measurement error rather than any indication of increased performance; all results from this trial were well within the margin of error. This result is consistent with expectations since memory allocation does not directly interact with any LSM hooks in the kernel, and neither BPFBox nor BPFContain instruments any BPF programs on the page allocation code path.

Table 6.7: Results of the Memory Allocations benchmark. Units are ns per event. Lower is better. Percent overhead is compared to the baseline result.

Test Case	System	Mean	Std	Overhead
Base	—	117.17	0.67	—
Passive	BPFBox	116.87	0.18	-0.26%
	BPFContain	120.05	1.27	2.46%
	AppArmor	117.24	0.97	0.06%
Allow	BPFBox	117.41	0.71	0.21%
	BPFContain	116.42	0.62	-0.64%
	AppArmor	117.49	0.81	0.28%
Complaining	BPFBox	117.62	0.75	0.38%
	BPFContain	115.73	1.04	-1.22%
	AppArmor	116.81	0.80	-0.31%

Kernel Compilation Results

The kernel compilation benchmark (Table 6.8 and Figure 6.3) provides a representative depiction of overhead for a computationally-heavy task that involves multiple processes and significant amounts of file I/O. The results of this benchmark indicate that BPFBox and BPFContain exhibit performance overhead that is roughly con-

sistent with AppArmor in the average case. The **Passive** and **Allow** results indicate that all three systems exhibit an acceptable performance overhead of under about 3%. The **Complaining** results indicate that BPFContain performs significantly better than both BPFBox and AppArmor under a large event logging volume. This result can be attributed to minor implementation details, including improvements in how BPFContain handles event logging from multiple distinct sources.

Table 6.8: Results of the Kernel Compilation benchmark. Units are seconds. Lower is better. Percent overhead is compared to the baseline result.

Test Case	System	Mean	Std	Overhead
Base	—	235.32	1.96	—
Passive	BPFBox	237.95	1.88	1.12%
	BPFContain	237.63	2.08	0.98%
	AppArmor	236.45	1.92	0.48%
Allow	BPFBox	238.23	2.19	1.24%
	BPFContain	243.09	2.19	3.30%
	AppArmor	237.59	2.04	0.97%
Complaining	BPFBox	269.64	1.98	14.59%
	BPFContain	244.80	2.04	4.03%
	AppArmor	288.54	2.11	22.62%

Apache Web Server Results

The Apache web server benchmark (Table 6.9 and Figure 6.4) indicates that, while BPFBox and BPFContain do exhibit a higher performance overhead than AppArmor, this overhead is still within an acceptable range at around 11% in the worst

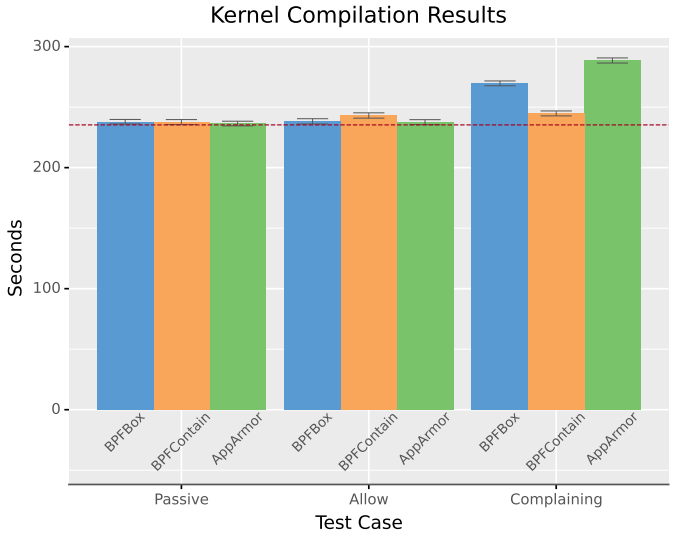

Figure 6.3: Results of the kernel compilation benchmark. The error bars show standard deviation and the red line shows the base measurement. Lower times are better.

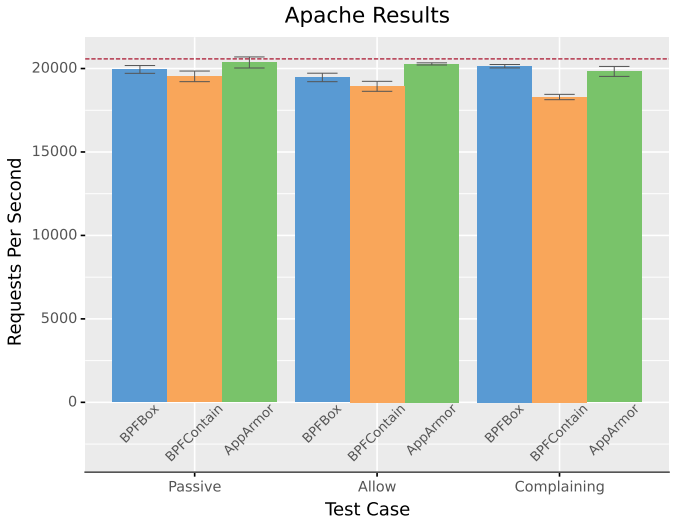

Figure 6.4: Results of the Apache web server benchmark. The error bars show standard deviation and the red line shows the base measurement. Higher requests per second are better.

Table 6.9: Results of the Apache benchmark. Units are requests per second. Higher is better. Percent overhead is compared to the baseline result.

Test Case	System	Mean	Std	Overhead
Base	—	20576.49	281.94	—
Passive	BPFBox	19946.04	233.62	3.06%
	BPFContain	19530.92	317.95	5.08%
	AppArmor	20363.42	331.64	1.04%
Allow	BPFBox	19465.86	253.81	5.40%
	BPFContain	18934.55	299.23	7.98%
	AppArmor	20276.95	64.30	1.46%
Complaining	BPFBox	20139.10	101.59	2.13%
	BPFContain	18293.09	160.00	11.10%
	AppArmor	19827.05	298.27	3.64%

case for BPFContain. This overhead should still be quite acceptable in practice, and can be improved through further optimizations in BPFContain's enforcement engine, which is still in the prototype phase. The results from the **Complaining** case appear to indicate a slight performance improvement for BPFBox over AppArmor; this is likely due to variance in the measurements rather than a true performance improvement, as the difference between the two systems falls within the margin of error.

It is worth noting that, due to an intentional ABI breakage in the upstream AppArmor module, AppArmor currently does not enforce any network policy on a stock Linux kernel [78, 120]. These results were confirmed experimentally when inspecting the AppArmor logs for the Apache **Complaining** test case. This means that

the performance results for AppArmor in the Apache test were at least significantly biased in favour of AppArmor. Future experimentation is required with a patched version of the Linux kernel to determine AppArmor's real overhead in this test case. We hypothesize that AppArmor's true overhead will more closely match BPFContain's.

6.1.3. Discussion of Performance Results

The results of the benchmarking tests show that both BPFBox and BPFContain incur acceptable performance overhead in practice. In many cases, overhead is competitive with AppArmor. In other cases, the performance overhead of BPFBox and BPFContain is higher than that of AppArmor, but still within a modest range, such that the slowdown should be either imperceptible or acceptable in many use cases. As BPFBox and BPFContain are both research prototypes, they have not yet been optimized to the extent that AppArmor has. This lack of optimization is particularly evident in the results for BPFContain, and may account for significant differences in performance in the file I/O and Apache web server tests.

While BPFBox and BPFContain are not as efficient as AppArmor in the **Passive** and **Allow** test cases, the additional performance overhead comes with an increase in flexibility and system observability afforded by an eBPF implementation as opposed to a traditional Linux Security Module. Moreover, BPFContain extends BPFBox's original enforcement model, adding new rule categories and enforcement defaults. While these enhancements may be the cause of some additional performance overhead, we hypothesize that the majority of BPFContain's perfor-

Table 6.10: Geometric means of Phoronix benchmarking results, as provided by the Phoronix Test Suite. These are indicative of overall performance across all tests. For test case, percent change from the base results are also given. Higher values are better.

Test Case	System	Geom. Mean	Overhead (%)
Base		6.238	—
Passive	BPFBox	6.007	3.70%
	BPFContain	5.951	4.60%
	AppArmor	6.158	1.28%
Allow	BPFBox	5.944	4.71%
	BPFContain	5.763	7.61%
	AppArmor	6.086	2.35%
Complaining	BPFBox	5.823	6.65%
	BPFContain	5.693	8.74%
	AppArmor	4.962	20.46%

mance overhead is due to sub-optimal memory allocations and data structure access patterns, which can be optimized in the future.

Despite under-performing in the **Passive** and **Allow** cases, both BPFBox and BPFContain significantly outperform AppArmor in the **Complaining** test case, due to a more efficient event logging mechanism. Table 6.10 shows the geometric mean of all tests for each test case, indicating that BPFBox and BPFContain exhibit an average performance penalty of under 9% in practice, while AppArmor can exhibit overheads of up to 20% in the worst case.

Although the results presented in this section indicate a comparative performance with AppArmor, many other widely-adopted LSMs can perform significantly worse

than AppArmor in some cases. For instance, Zhang *et al.* [159] found that SELinux, perhaps the most widely-used Linux MAC implementation, exhibits significant performance overhead, many times worse than AppArmor in some cases. These results could indicate that BPFBox and BPFContain might perform favourably compared to alternative LSMs like SELinux, although further investigation is needed in order to establish a direct comparison.

6.2. Security Analysis

We now turn our attention to the security of BPFBox and BPFContain. Specifically, we conduct an informal security analysis on both systems, evaluating how well they are able to confine an attacker under the threat model presented in Section 3.6. In particular, we examine the various policy rule categories provided by both BPFBox and BPFContain as well as how their respective enforcement engines enforce policy at runtime. We characterize an adversary's ability to escape confinement based on whether the adversary is able to violate the security assumptions of BPFBox and BPFContain under a policy designed to prevent such violations.

6.2.1. Threat Model Revisited

Recall the threat model presented in Section 3.6. We assume a remote adversary who is confined by some policy $\mathcal{P}$. The adversary's goal is to escape confinement by circumventing $\mathcal{P}$, enabling them to access sensitive resources, interfere or tamper with the system, or perform other unauthorized actions. Our goal is to confine the adversary, limiting the set of all actions they can perform to some subset of

allowed actions. We express such confinement using the policy $\mathcal{P}$ which defines rules governing the set of operations some subject $\mathcal{S}_i$ may perform on system objects $\mathcal{O}_1 \ldots \mathcal{O}_n$. To enforce our confinement policy, we rely on a *confinement engine* which has been loaded into the kernel's reference monitor.

Under our threat model, we assign significant capabilities to the adversary. Aside from the restrictions imposed by our confinement policy, we assume that they have root-level access to the system, including the ability to load code into the kernel, bypass discretionary access controls, read or modify any persistent resource, and establish persistent access to the system. Thus, an attacker that is able to escape or bypass confinement has effectively compromised the entire system. Further, our enforcement engine must take steps protect itself, preventing the attacker from simply loading or modifying code in the kernel, which would result in the ability to tamper with or bypass the enforcement engine.

In the subsections that follow, we consider four broad access categories and describe how BPFBox and BPFContain's confinement policy and enforcement engine prevent attacks related to these access categories. In some cases, the original BPFBox policy specification (as presented in this thesis) provided insufficient protection against specific access patterns. In these instances, we describe how BPFContain improves upon the original BPFBox design.

6.2.2. Files, Filesystems, and Kernel Interfaces

Correct mediation of file and filesystem accesses is critical to ensure that an adversary is confined. Files define the canonical persistent data store in modern COTS (Com-

mercial Off-The-Shelf) operating systems, including Linux. Depending on the nature of the file, the information stored within may be confidential, security sensitive, or otherwise critical to normal system operation. Attacker modification of persistent files is often the first step in mounting a Confused Deputy attack [70] for privilege escalation, along with other classes of attack such as data corruption, information disclosure, and memory safety attacks.

Under the Unix model, special files and filesystems define an entry point into kernel interfaces, many of which are security sensitive. For example, character devices expose an interface into device drivers, while special filesystems like `sysfs` and `securityfs` expose behavioural parameters and export sensitive information such as the system memory map. Limiting access to these files is of paramount importance, since unrestricted access could enable an attacker to change the behaviour of the kernel, read sensitive information, modify global system parameters, or interfere with arbitrary user processes.

In order to restrict access to files (and special files), our enforcement engine must have complete mediation over the set of all file operations exposed by the OS kernel and our policy language must be expressive enough such that it can express the set of all allowed operations on a specific filesystem object (e.g. an inode).

BPFBox

To confine a process's access to the filesystem, BPFBox supports "file" rules (c.f. Section 4.3.1), which take a pathname and corresponding access vector. This access vector encodes the specific file operations that a process can perform on the file. Since BPFBox policies are default deny, an adversary running under a BPFBox

confinement policy should be unable to perform an operation Op_i on any file F_j unless this operation is explicitly covered under a file rule. For the purposes of confinement, BPFBox treats all files equally, regardless of whether the file is a special file or belongs to a special filesystem. Thus, any access to any file on the system is governed by the same set of file rules. The only exception to this is a special "proc" rule which enables a policy to define access to per-pid `procfs` entries belonging to another process.

To enforce its file policy, the BPFBox enforcement engine instruments eBPF programs on several LSM hooks, including inode-based, file-based, and path-based hooks. Taken together, these hooks provide mediation[2] over the set of all file operations, by instrumenting access at the VFS (Virtual Filesystem) layer. BPFBox encodes its file policy using an eBPF map, taking the inode and device numbers associated with a file and its filesystem as a key. When a process requests access to a file, BPFBox computes a key for that file and makes a query against its policy map. This technique has a natural side effect of eliminating TOCTTOU race conditions on a file's pathname, since inodes are resolved at policy load time.

Despite being able to uniquely identify a file, BPFBox's inode resolution strategy is subject to a few fundamental limitations. Since inode's are resolved at policy load time, BPFBox traces the confined process, granting implicit access to any files that the process creates during its lifecycle. An attack against this model would consist of unlinking and re-creating a file after a policy has been loaded, causing BPFBox to see it as a different file when enforcing access control. In the worst case, this is

[2]This mediation is complete insofar as the LSM hook placement is correct. While this has not been formally verified, LSM hook placement is the basis for security guarantees of all LSMs.

effectively a denial of service against the confined process, since BPFBox enforces a default-deny policy on unrecognized files. However, this attack would also require the adversary to be unconfined or to be operating under a confinement policy that permitted the necessary operations on the file.

BPFContain

Like BPFBox, BPFContain supports file rules that specify a target pathname and corresponding access vector. In addition to file rules, BPFContain also supports filesystem rules for defining per-filesystem policy. A filesystem rule can be thought of as a coarser-grained version of a file rule, specifying access at the per-filesystem rather than the per-file level. In order to ensure mediation over explicit denials, BPFContain always prioritizes fine-grained file rules over coarse-grained filesystem rules. This prevents a policy from inadvertently obviating its own file rules with a careless filesystem rule.

Due to current limitations of eBPF, BPFContain currently uses the same inode resolution strategy as BPFBox. The strengths and weaknesses of this technique are the same as in BPFBox, with no major differences. However, future versions of BPFContain may move to runtime path-based resolution, once the kernel offers better support for string helpers and unbounded loops.

Unlike BPFBox, BPFContain treats regular files differently from special files and special filesystems. This improves security, as not only do special files and filesystems have different semantics from regular files, but they also directly expose interfaces into (potentially untrusted) kernel code. Rather than resolving special files by inode and filesystem number, BPFContain looks at their major and minor

number pair. Taken together, these two numbers always uniquely identify a special file, regardless of when it was created or whether it exists in multiple places at once. This resolution strategy is not vulnerable to the same attack as the inode resolution strategy, since the adversary cannot control a device's major and minor number. However, since modern systems dynamically allocate device numbers, this could cause inconsistencies between the policy and the system, potentially resulting in spurious policy decisions on device access. To fix this problem, a future version of BPFContain will resolve device numbers using a pathname at runtime.

Rather than being outright default deny, BPFContain uses heuristics to determine the appropriate default policy action on system objects. In the case of regular files, BPFContain checks to see whether the filesystem superblock is a part of the container's user namespace (provided that namespace is not the global namespace). If it is, the filesystem was mounted within the context of the container, and so it is safe to allow access. Otherwise, access is denied. Since special files and filesystems can be significantly more dangerous, BPFContain assumes a default-deny policy instead. This strategy allows BPFContain to (under certain conditions) relax requirements on policy authors without sacrificing security.

6.2.3. POSIX Capabilities and Privileged System Calls

Under Linux, POSIX capabilities [24] define a process' ability to override discretionary access controls and access certain privileged kernel interfaces. Although these cannot be used to override mandatory access controls (such as those enforced by BPFBox and BPFContain), they still provide a means of limiting the power of

the root user. This becomes particularly evident in the case of privileged kernel interfaces which can affect global system state or enable an attacker to load untrusted code into the kernel.

BPFBox

The BPFBox prototype presented in this thesis does not directly interact with POSIX capabilities in any way. Instead, BPFBox denies access to system resources using its LSM hooks directly. To prevent the adversary from interfering with BPFBox's eBPF programs or maps, BPFBox always denies the `bpf(2)` system call from a confined process, as well as any system calls that could load code into the kernel (e.g. load a kernel module).

The current version of BPFBox does not perform any access validation on mounting filesystems, which could potentially allow an attacker to modify the global filesystem hierarchy or interfere with the operation of other processes. This weakness was fixed in BPFContain with the additional instrumentation of the `sb_mount` LSM hook.

BPFContain

Unlike BPFBox, BPFContain provisions policy authors with a way to limit the POSIX capabilities that a container can access. This is done using a "capability" rule, which takes a list of capabilities. When a containerized process wishes to use a capability, it must already possess the capability, pass existing checks in the kernel, and pass BPFContain's capability rule checks. This improves security by introducing an additional layer of control over a process' bounding capability set.

Like BPFBox, BPFContain places restrictions on the `bpf(2)` system call, ensuring that a confined process can never interfere with its eBPF programs or maps. This ensures that an adversary cannot trivially escape confinement by passing the enforcement engine. Further, BPFContain instruments the `locked_down` LSM hook which mediates kernel features that could potentially enable arbitrary code execution in kernelspace. We explicitly deny any such action, ensuring that a confined process cannot directly interfere with BPFContain or other aspects of the kernel.

In addition to limiting `bpf(2)` and kernel modules, BPFContain places similar restrictions on the perf events subsystem (a debugging feature supported by the kernel that enables tracing the CPU and system calls). BPFContain also blocks operations that could impact the global system state, such as accessing the kernel keyring, rebooting the system, modifying system network policy, or changing the system time.

To prevent a confined process from manipulating the layout of the filesystem or subverting BPFContain's default policy at runtime, BPFContain prohibits a confined process from mounting or unmounting any filesystems using the `sb_mount` LSM hook. BPFContain also prevents a confined process from changing its namespace by instrumenting an fentry program on the `site_task_namespaces` kernel function. This provides improved security over traditional LSM-based approaches, as no LSM hook currently guards changing namespaces.

6.2.4. Networking

Access to the kernel's networking stack enables an adversary to connect to remote hosts, potentially enabling the exfiltration of sensitive data or providing an entry point for additional attacks. A network socket could also be used by an attacker to bypass inter-process communication checks (i.e. two processes could communicate over the network instead of using canonical IPC mechanisms provided by the kernel). Thus, securing the kernel's networking stack is critically important for ensuring confinement.

BPFBox

BPFBox secures the network stack at the socket layer, defining "network" rules that map address families to a set of allowed socket operations. Since BPFBox is default deny, a policy without any network rules will implicitly cause any socket operations to be denied by default, thus preventing the process from accessing the network stack. Using a network operation as a taint rule enables BPFBox to start enforcing a stricter policy once a process has started communicating with the outside world, improving security in the case where a remote adversary wants to interact with a process or exfiltrate sensitive information.

The BPFBox prototype presented in this thesis does not support defining more advanced network policy at the protocol level (e.g. filtering network traffic by IP addresses and port numbers). This means that BPFBox cannot discriminate between network traffic according to its source and destination, potentially enabling network-based attacks for software that requires a network connection to function.

BPFContain

Like BPFBox, BPFContain defines network policy at the socket layer, through its own "network" rules. However, BPFContain greatly simplifies the BPFBox model by restricting network rules to the IPv4 and IPv6 address families. Other families are either covered under IPC rules (c.f. the next subsection) or are outright denied by default. Future versions of BPFContain may define additional rules for interacting with other address families like netlink.

As with BPFBox, BPFContain does not currently support defining network policy at the protocol-level. This opens a BPFContain container up to the same class of network-based attacks as BPFBox, so long as the container already has access to the networking stack through its BPFContain policy. Network firewalls like `iptables` can be used to provide additional network security, but future versions of BPFContain will support such policy natively, obviating the need to use `iptables`.

6.2.5. IPC

Inter-process communication mechanisms enable processes to communicate with each other, sending data back and forth, sending signals to each other, or sharing resources such as open file descriptors. Without a secure IPC policy, an adversary may be able to trivially escape confinement by establishing a communication channel with an unconfined process or sharing information or resources between two processes running under a different confinement policy. Thus, securing IPC communication is essential to ensuring that confinement guarantees hold.

BPFBox

BPFBox provisions three rules for IPC access: "ptrace" rules, "signal" rules, and "network" rules using the `AF_UNIX` address family. These rules cover ptrace, signal, and Unix socket access respectively. Ptrace rules can be used to control whether a confined process can ptrace other processes and whether another process can ptrace a confined process, providing two-way protection. Signal rules control the ability for a confined process to send specific categories of signals to another process. Signals are categorized based on their implications, with fatal and uncatchable signals belonging to their own category due to their increased severity. A network rule with the address family of Unix controls a process' ability to interact with Unix domain sockets. Named pipes are covered under BPFBox's file policy.

The BPFBox prototype presented in this thesis does not provide a way to control which process is at the other end of a Unix socket, resulting in potential security vulnerabilities when a confined process is able to establish a socket connection with an unconfined process. Another weakness in BPFBox's current IPC policy is that it does not provision rules for restricting access to System V IPC objects. BPFContain addresses these shortcomings as explained below.

BPFContain

Rather than defining multiple rule categories for IPC, BPFContain defines a single "IPC" rule, which takes as an argument the name of another BPFContain policy. This rule can be used to grant mutual IPC access between two containers running under a different policy. If either policy does not include an IPC rule granting

access to the other, access is denied. This ensures that both policies mutually agree that they should be able to communicate, eliminating the problem of an adversary stealthily infiltrating or exfiltrating data into or out of a confined container.

IPC rules under BPFContain cover all categories of IPC under Linux, including Unix sockets, System V IPC, pipes, and signals. The justification for this is that one IPC technique is more or less equivalent to another, and thus allowing one category of IPC but not another does not provide any additional security. Due to BPFContain's container-specific default policy, processes within the same container are allowed to communicate with each other without defining any IPC rules. This is acceptable since we treat the container as a unit of security.

Ptrace access is also covered under BPFContain's default policy. A process may ptrace another if and only if the two processes exist in the same container. Otherwise, access is denied. Since ptrace is an extremely powerful interface (effectively giving the tracer total control over the tracee), BPFContain defines no policy rules to make exceptions to its default ptrace enforcement.

6.3. Summary

In this chapter, we have evaluated the performance and security of the BPFBox and BPFContain research prototypes. We conduct a series of benchmarking tests to evaluate performance in comparison to the base system and AppArmor, a widely-used LSM that has been upstreamed in the Linux kernel. We find that BPFBox and BPFContain exhibit modest percent overhead over AppArmor in the majority of tests, and that they can outperform AppArmor under specific circumstances. While

BPFContain introduces additional performance overhead on top of the original BPFBox design, this comes with additional flexibility, improved security, and a more nuanced default policy. Since BPFBox and BPFContain are kernel agnostic, we can update them without updating the host kernel, enabling bugs to be fixed like userspace programs. Further, we argue that BPFContain can be significantly optimized in the future, enabling future iterations on the design to perform more competitively with existing LSMs.

An informal security analysis reveals that BPFBox and BPFContain found limitations in BPFBox that were later addressed in BPFContain, such as a lack of support for capability-level policy and limited support for restricting access to IPC objects. BPFContain resolves the majority of BPFBox's limitations while simultaneously simplifying the policy language. While BPFContain also has limitations with respect to device-level policy and coarse-grained network policy, these can be addressed in future work (see Chapter 8).

Chapter 7.

Case Studies

In this chapter, we examine specific case studies, where we apply BPFBox and BPFContain policies to solve realistic problems. We examine how BPFBox and BPFContain can be used to confine a webserver and database and how they can implement the default Docker confinement policy. We also discuss how future extensions to BPFContain could enable it to confine an untrusted container with a minimal policy file. To offer a basis for comparison, we contrast presented policies with some available equivalents and discuss how the semantics of the policy language and enforcement engine can impact the resulting policy file.

7.1. Confining a Web Server and Database

We first examine a practical use case: confining a simple web server and database deployment. In particular, we focus on the Apache httpd web server and the MySQL database management system. These two pieces of software are often used together

(e.g. as part of the LAMP stack [73]) and provide a good example of how we can define specific policy exceptions to allow two processes or containers to communicate with each other and with the outside world. In this example, we assume that httpd and mysqld communicate with each other using a shared Unix domain socket created by mysqld. We synthesize policies through a combination of running the programs under `strace`, finding library dependencies with `ldd`, and running the programs under BPFContain's complaining mode. We use the output of these tools along with BPFContain's log output to write the security policies by hand.

BPFBox

In BPFBox, we define two profiles: one for httpd, and one for mysqld. Listing 7.1 depicts the httpd policy while Listing 7.2 depicts the mysqld policy. These policies are simplified examples but provide a representative idea of what it's like to confine a complex application down to its basest functionality.

The httpd policy (Listing 7.1) defines allow and taint rules for three categories of socket access: `inet`, `inet6`, and `unix`. These categories cover IPv4 and IPv6 network access as well as IPC over a Unix domain socket. This Unix domain socket is what httpd will use to communicate with the database. By defining these network rules as both allow and taint, we indicate that default-deny enforcement should begin only *after* the Apache daemon has begun interacting with the outside world. Using this technique, the BPFBox policy may be greatly simplified by eliminating the need to define any policy corresponding to the setup phase of httpd.

The bulk of the BPFBox policy is made up of filesystem rules, enabling access to a variety of configuration files that httpd needs to read at runtime, a few informational

files exposed by the kernel under procfs, shared libraries that may be loaded at runtime, the mysqld Unix socket, and the directory that httpd will use to serve web content. We also define a rule allowing httpd to run suexec, a helper application used to launch CGI (Common Gateway Interface) scripts. We indicate that launching suexec should untaint the process and transition to a suexec profile. This profile would then define the access control policy for suexec (e.g. which scripts it is allowed to run).

Listing 7.1: A BPFBox policy for Apache httpd.

```
#![profile "/bin/httpd"]

/* Allow IP and Unix socket operations, and taint when
 * sending or receiving */
#[allow] {
  net(inet, any)
  net(inet6, any)
  net(unix, any)
}
#[taint] {
  net(inet, send|recv)
  net(inet6, send|recv)
  net(unix, send|recv)
}

#[allow] {
  /* Allows kill(2) to check for process existence
   * and to send fatal signals */
  signal("/bin/httpd", check|fatal)

  /* Write to logs */
  fs("/var/log/httpd/*log", getattr|read|append)
  fs("/var/log/httpd", getattr|read|write)

  /* Create PID file */
  fs("/run/httpd/", write)
  /* Delete or modify an existing PID file if necessary */
  fs("/run/httpd/httpd.pid", getattr|rm|write)

  /* Serve files from /srv/html/ and all subdirectories */
```

```
31    fs("/srv", read|getattr)
32    fs("/srv/html", read|getattr)
33    fs("/srv/html/**", read|getattr)
34
35    /* Access to mysqld socket */
36    fs("/run/mysqld", getattr|read)
37    fs("/run/mysqld/mysqld.sock", getattr|read|write)
38
39    /* Read configuration */
40    fs("/usr/share/httpd/**", read|getattr)
41    fs("/etc/httpd/", getattr)
42    fs("/etc/httpd/conf/**", read|getattr)
43    fs("/usr/share/zoneinfo/**", read|getattr)
44
45    /* Read hostname information */
46    fs("/etc/resolv.conf", read|getattr)
47    fs("/etc/host*", read|getattr)
48
49    /* Read-only access to required kernel info */
50    fs("/proc/sys/kernel/random/boot_id", read)
51    fs("/proc/sys/kernel/ngroups_max", read)
52
53    /* Shared libraries loaded at runtime */
54    fs("/usr/lib/httpd/modules/*.so", getattr|read|exec)
55    fs("/usr/lib/libnss*.so.*", getattr|read|exec)
56    fs("/usr/lib/libgcc_s.so.*", getattr|read|exec)
57  }
58
59  /* Transition to a separate suexec policy */
60  #[transition] {
61    fs("/usr/bin/suexec", getattr|read|exec)
62  }
```

The BPFBox policy for mysqld works in much the same way as the policy for httpd. Major differences include disabling all socket access except to Unix domain sockets. This ensures that the database is not exposed to the outside world but still enables is to communicate with httpd over its Unix socket. Like with httpd, we define specific rules enabling mysqld to read important configuration files, log events, create and modify its PID file and Unix socket, and load some shared libraries at

runtime. The entire setup phase for mysqld occurs while the process is untainted, thus allowing us to eliminate rules for any shared libraries loaded before the process becomes tainted.

Listing 7.2: A BPFBox policy for MySQL.

```
#![profile "/bin/mysqld"]

/* Allow Unix socket operations, and taint when
 * sending or receiving. Also allow creating and
 * binding inet and inet6 sockets (necessary to
 * pass assertion checks) */
#[allow] {
  net(unix, any)
  net(inet, create|bind)
  net(inet6, create|bind)
}
#[taint]
net(unix, send|recv)

#[allow] {
  /* Allows kill(2) to check for process existence
   * and to send fatal signals */
  signal("/bin/mysqld", check|fatal)

  /* /dev/null and /dev/urandom */
  fs("/dev/null", getattr|read|write)
  fs("/dev/urandom", getattr|read)

  /* Write to logs */
  fs("/var/log/mysqld/*log", getattr|read|append)
  fs("/var/log/mysqld", getattr|read|write)

  /* Access to /var/lib/mysql */
  fs("/var/lib/mysql", read|write|getattr)
  fs("/var/lib/mysql/**", read|write|getattr|rm)

  /* Create PID file and socket */
  fs("/run/mysqld", getattr|read|write)
  fs("/run/mysqld/**", getattr|read|write|rm)

  /* Read configuration */
  fs("/etc/mysql", read|getattr)
  fs("/etc/mysql/**", read|getattr)
```

```
39    fs("/usr/share/zoneinfo", read|getattr)
40    fs("/usr/share/zoneinfo/**", read|getattr)
41
42    /* Shared libraries loaded at runtime */
43    fs("/var/lib/mysql/plugin/*.so", getattr|read|exec)
44    fs("/usr/lib/libnss_files-*.*.so", getattr|read|exec)
45  }
```

BPFContain

In BPFContain we once again define a policy for httpd and a policy for mysqld. These policies are depicted in Listing 7.3 and Listing 7.4 respectively. These policies are largely similar to the BPFBox policies, with a few minor differences that can be attributed to BPFContain's nuanced policy defaults and its updated policy language.

Like BPFBox, the majority of the BPFContain httpd policy (Listing 7.3) focuses on specifying filesystem access for httpd. Since BPFContain also supports tainting semantics, we leverage these to eliminate the need to define rules for operations preceding the taint. Specifically, we taint container once it has performed any networking operations or any IPC with mysqld. Unlike BPFBox, however, BPFContain does not support untainting a process. This is a natural extension of the fact that BPFContain deals in container semantics rather than process-level confinement — it does not make sense to untaint and transition profiles in the context of a container, since a single security policy applies to the entire container. This means that we must still specify access to shared libraries that will be loaded by suexec (along with whatever applications suexec will run, such as Python).

Aside from the aforementioned differences, the per-file policy is more or less the

same as BPFBox. To enable IPC between the httpd and mysql, we define an IPC allow rule that lists the mysqld policy. We also enable socket networking using a network allow rule and enable signalling of existing instances of httpd with a signal rule. Finally, we use a capability rule to grant access to the `CAP_NET_BIND_SERVICE` capability, allowing httpd to bind to privileged ports. Since BPFBox does not support capability rules, there is no equivalent to this rule in the BPFBox policy, meaning that POSIX capabilities would effectively be unrestricted.

Listing 7.3: A BPFContain policy for Apache httpd.

```
name: httpd
defaultTaint: false

allow:
  # /dev/urandom, /dev/random, /dev/null
  - dev: random
  - dev: null

  # Access to log files
  - file: {pathname: /var/log/httpd, access: rw}
  - file: {pathname: /var/log/httpd/*log, access: ra}

  # Create pidfile, delete or modify an existing pid file if necessary
  - file: {pathname: /run/httpd, access: rw}
  - file: {pathname: /run/httpd/**/*, access: rwd}

  # Read configuration
  - file: {pathname: /usr/share/httpd/**/*, access: r}
  - file: {pathname: /etc/httpd, access: r}
  - file: {pathname: /etc/httpd/conf/**/*, access: r}
  - file: {pathname: /usr/share/zoneinfo/**/*, access: r}

  # Serve files
  - file: {pathname: /srv, access: r}
  - file: {pathname: /srv/html, access: r}
  - file: {pathname: /srv/html/**, access: r}

  # Read hostname information
  - file: {pathname: /etc/resolv.conf, access: r}
  - file: {pathname: /etc/host*, access: r}
```

```
31
32    # Shared libraries loaded at runtime
33    - file: {pathname: /usr/lib/httpd/modules/*.so, access: mr}
34    - file: {pathname: /usr/lib/libnss*.so.*, access: mr}
35    - file: {pathname: /usr/lib/libgcc_s.so.*, access: mr}
36
37    # Execute suexec and python
38    - file: {pathname: /usr/bin/suexec, access: rx}
39    - file: {pathname: /usr/bin/python, access: rx}
40
41    # Shared libraries required for suexec and python
42    # This is unfortunately required since BPFContain currently
43    # has no notion of untainting like BPFBox
44    - file: {pathname: /usr/lib/libpython*.so.*, access: mr}
45    - file: {pathname: /usr/lib/libc.so.*, access: mr}
46    - file: {pathname: /usr/lib/libpthread.so.*, access: mr}
47    - file: {pathname: /usr/lib/libdl.so.*, access: mr}
48    - file: {pathname: /usr/lib/libutil.so.*, access: mr}
49    - file: {pathname: /usr/lib/libm.so.*, access: mr}
50    - file: {pathname: /usr/lib64/ld-linux-x86-64.so.*, access: mr}
51
52    # Allow ipc with mysql
53    - ipc: mysqld
54
55    # Allow sending signals to existing httpd instances
56    - ipc: httpd
57
58    # Bind to privileged ports, change uid and gid
59    - capability: [netbindservice, setuid, setgid]
60
61    # Use networking
62    - net: [server, send, recv]
63
64  taint:
65    # Taint when performing any ipc or networking
66    - net: [send, recv]
67    - ipc: mysqld
```

The mysqld policy for BPFContain (Listing 7.4) also shares many similarities with the BPFBox version. In particular, we define equivalent file access rules to enable the mysqld to access all of the files it requires for normal operation. We define an IPC rule, granting mutual IPC access to the httpd policy, and enabling the two

to communicate with each other. We taint the container once it has performed any IPC with httpd. We enable the creation of new network sockets in order to pass assertion checks, but otherwise explicitly deny any network access.

Listing 7.4: A BPFContain policy for MySQL.

```
name: mysqld
defaultTaint: false

allow:
  # Access to log files
  - file: {pathname: /var/log/mysqld, access: rw}
  - file: {pathname: /var/log/mysqld/*log, access: ra}

  # Access to /var/lib/mysql
  - file: {pathname: /var/lib/mysql, access: rw}
  - file: {pathname: /var/lib/mysql/**/*, access: rwd}

  # Create pidfile and socket
  - file: {pathname: /run/mysqld, access: rw}
  - file: {pathname: /run/mysqld/**/*, access: rwd}

  # Read configuration
  - file: {pathname: /etc/mysql, access: r}
  - file: {pathname: /etc/mysql/**/*, access: r}
  - file: {pathname: /usr/share/zoneinfo, access: r}
  - file: {pathname: /usr/share/zoneinfo/**/*, access: r}

  # Shared libraries loaded at runtime
  - file: {pathname: /var/lib/mysql/plugin/*.so, access: mr}
  - file: {pathname: /usr/lib/libnss_files-*.*.so, access: mr}

  # Allow ipc with httpd
  - ipc: httpd

  # Allow sending signals to existing mysqld instances
  - ipc: mysqld

  # Allow mysqld to create a new network socket
  # (Necessary to pass assertions)
  - net: server

taint:
  # Taint when performing ipc with httpd
```

```
39    - ipc: httpd
40    # Taint when sending or receiving any network traffic
41    - net: [send, recv]
42
43  deny:
44    # Explicitly deny sending or receiving any network traffic
45    - net: [send, recv]
```

Simplifying the BPFContain Example

Once BPFContain has been fully integrated with Docker support, it may become possible to greatly simplify the above policy examples, leveraging default filesystem policy to grant access to all of the required files, without the need to explicitly specify rules for each file. We leverage a shared `/tmp` filesystem, mounted on the host, to allow both mysqld and httpd to access the same Unix socket. The result is an extremely simple policy that can express all the required interfaces in just a few lines. Listing 7.5 and Listing 7.6 give example policies for httpd and mysqld respectively.

Listing 7.5: A simplified BPFContain policy for Apache httpd running in a Docker container, leveraging future support for automatic filesystem policy.

```
1  name: httpd-container
2  defaultTaint: true
3
4  allow:
5    # Grant access to global /tmp filesystem, mounted as a Docker volume
6    # This is where the mysqld Unix socket will go
7    - fs: {pathname: /tmp, access: r}
8    # /dev/urandom, /dev/random, /dev/null
9    - dev: random
10   - dev: null
11   # Allow network access
12   - net: [server, send, recv]
13   # Allow ipc access with mysqld
```

```
14    - ipc: mysqld-container
15    # Bind to privileged ports, change uid and gid
16    - capability: [netbindservice, setuid, setgid]
```

Listing 7.6: A simplified BPFContain policy for MySQL running in a Docker container, leveraging future support for automatic filesystem policy.

```
1  name: mysqld-container
2  defaultTaint: true
3
4  allow:
5    # Grant access to global /tmp filesystem, mounted as a Docker volume
6    # This is where the mysqld Unix socket will go
7    - fs: {pathname: /tmp, access: rw}
8    # Allow mysqld to create a new network socket
9    # (Necessary to pass assertions)
10   - net: server
11   # Allow ipc access with httpd
12   - ipc: httpd-container
13 deny:
14   # Explicitly deny sending or receiving any network traffic
15   - net: [send, recv]
```

7.2. The Default Docker Policy

Docker [50] applies a coarse-grained default confinement policy to all containers using a combination of Linux confinement primitives. On supported systems[1], this includes a default AppArmor policy template [49, 51], a default Seccomp-bpf profile, and a set of POSIX capabilities which are dropped at runtime [50].

Docker's policy defaults are highly coarse grained, with an emphasis on practical security while ensuring that the vast majority of container configurations will "just

[1]Recall that not all Linux distributions support AppArmor or Seccomp-bpf to begin with. In such cases, Docker simply discards its default confinement policy altogether.

work," out of the box. This affords a practical opportunity to examine how BPFBox and BPFContain policies compare with the default Docker policy. Table 7.1 summarizes the key aspects of Docker's confinement policy, highlighting default access levels enforced by various Linux confinement primitives. Listing 7.7 depicts Docker's default AppArmor template, taken directly from the Docker sources on GitHub [51].

Table 7.1: A summary of Docker's default confinement policy [49, 50, 51]. Policy is enforced using a number of Linux confinement primitives, including AppArmor, Seccomp-bpf, and dropped POSIX capabilities at runtime. Docker generates and loads AppArmor policy at container runtime using a pre-determined, coarse-grained AppArmor template file (c.f. Listing 7.7).

Access Category	Default	Docker Implementation
Files	Allow access to all files except specific procfs and sysfs entries.	AppArmor Template
Filesystem Mounts	Deny all filesystem mounts.	AppArmor Template
POSIX Capabilities	All capabilities enabled in AppArmor. Drop specific capabilities at runtime.	AppArmor Template and Dropped Capabilities
Ptrace	Allowed within container.	AppArmor Template
Signals	Allowed within container.	AppArmor Template
Network	Allow all network access.	AppArmor Template
IPC	Allow all IPC access.	AppArmor Template
System Calls	Deny about 60 obsolete/dangerous system calls.	Seccomp-bpf

BPFBox

We begin by examining a mostly equivalent policy in BPFBox, given in Listing 7.8. Re-implementing Docker's default confinement policy in BPFBox is surprisingly

Listing 7.7: Docker's default AppArmor template [51], at the time of writing this thesis. Docker uses Go's string templating syntax to modify the AppArmor profile according to the current Docker version and container metadata.

```
{{range $value := .Imports}}
  {{$value}}
{{end}}
profile {{.Name}} flags=(attach_disconnected,mediate_deleted) {
{{range $value := .InnerImports}}
  {{$value}}
{{end}}
  network,
  capability,
  file,
  umount,
{{if ge .Version 208096}}
  # Host (privileged) processes may send signals to container processes.
  signal (receive) peer=unconfined,
  # dockerd may send signals to container processes (for "docker kill").
  signal (receive) peer={{.DaemonProfile}},
  # Container processes may send signals amongst themselves.
  signal (send,receive) peer={{.Name}},
{{end}}
 # deny write for all files directly in /proc (not in a subdir)
  deny @{PROC}/* w,
  # deny write to files not in /proc/<number>/** or /proc/sys/**
  deny @{PROC}/{[^1-9],[^1-9][^0-9],
    [^1-9s][^0-9y][^0-9s],[^1-9][^0-9][^0-9][^0-9]*}/** w,
  # deny /proc/sys except /proc/sys/k* (effectively /proc/sys/kernel)
  deny @{PROC}/sys/[^k]** w,
  # deny everything except shm* in /proc/sys/kernel/
  deny @{PROC}/sys/kernel/{?,??,[^s][^h][^m]**} w,
  deny @{PROC}/sysrq-trigger rwklx,
  deny @{PROC}/kcore rwklx,
  deny mount,
  deny /sys/[^f]*/** wklx,
  deny /sys/f[^s]*/** wklx,
  deny /sys/fs/[^c]*/** wklx,
  deny /sys/fs/c[^g]*/** wklx,
  deny /sys/fs/cg[^r]*/** wklx,
  deny /sys/firmware/** rwklx,
  deny /sys/kernel/security/** rwklx,
{{if ge .Version 208095}}
  # suppress ptrace denials when using 'docker ps' or using 'ps' inside a
    container
  ptrace (trace,read,tracedby,readby) peer={{.Name}},
{{end}}
}
```

challenging. BPFBox is not designed to implement coarse-grained confinement policy, and so specifying things like global access to all files is impossible. We compromise by granting recursive access to all files within a given filesystem, repeating the process for each filesystem as required. This is *not* the intended use case for BPFBox file rules, but it is required to match the over-permissive filesystem access provisioned by Docker. Aside from filesystem-specific policy, most of Docker's default policy can be implemented relatively easily and cleanly in BPFBox's policy language.

Listing 7.8: Implementing the default Docker policy in BPFBox.

```
#![profile "/path/to/init/program"]

#[allow] {
  /* Allow essentially global access to a filesystem */
  fs("/path/to/filesystem/**", read|write|setattr|getattr|rm|link|ioctl)
  /* Repeat for others... */

  /* Allow access to /proc/sys/kernel/shm* */
  fs("/proc/sys/kernel/shm*", read|write|setattr|getattr)

  /* Sensible default access for procfs per-pid entries */
  proc(self, read|write)
  proc(child, read|write)
}

#[allow]
#[taint]
{
  /* Access to network families */
  net(inet, any)
  net(inet6, any)
  net(unix, any)

  /* Ptrace child processes */
  ptrace(child, read|write|attach)

  /* Send sigchld up to parent processes, any signal to children */
  signal(parent, sigchld)
```

```
29    signal(child, any)
30  }
31
32  #[transition]
33  #[untaint]
34  {
35    /* Allow execve calls to allowed executables,
36     * tainting and transitioning profiles when doing so */
37    fs("/path/to/allowed/executable", read|exec)
38    /* Repeat for others... */
39  }
```

Like Docker's AppArmor policy, our BPFBox policy enables access to per-pid entries in procfs and uses BPFBox's default-deny enforcement to restrict all others. Similar logic applies to the `/proc/sys/kernel/shm*` entries under procfs. We also grant full networking stack access, ptrace access for child processes, and full signal access for child processes running under the container. Since these operations have the potential to introduce vulnerabilities from outside sources, we mark them as tainting the corresponding process. Leveraging taintedness, the BPFBox policy eliminates the need to specify access to shared library dependencies and other artifacts of the C runtime.

For more complex container deployments that include more than a single binary, the BPFBox policy may need to specify access to alternative executables under the container. We do so using an individual file rule for each executable, optionally specifying that the process should untaint itself and/or transition to a new profile. Notably, the version of BPFBox presented in this thesis does *not* include capability-level policy, and so it is not included here[2]. However, the default Docker confinement policy does not implement capability-level filtering anyway, instead relying on

[2]BPFContain later rectified this gap in BPFBox's policy language.

dropped capabilities at runtime.

Although the BPFBox policy depicted in Listing 7.8 does not fully map to the precise Docker default policy, it gets very close in most respects, aside from filesystem policy. Under BPFBox, filesystem policy is necessarily finer-grained, as it does not support the ability to specify coarse-grained access to all files on the system. Despite these challenges, the end result is a functional (and, in some aspects, more secure) alternative to the default Docker policy.

BPFContain

Having examined how BPFBox can be used to implement an approximate version of Docker's default confinement policy, we now turn our attention to BPFContain. Listing 7.9 shows the full BPFContain policy. Note that many aspects of Docker's default policy are covered by BPFContain's default container-boundary enforcement. Using this to its advantage, the BPFContain policy is significantly simpler than both the AppArmor and BPFBox versions while maintaining the same level of expressiveness.

Listing 7.9: Implementing the default Docker policy in BPFContain.

```
name: default-docker
defaultTaint: true

allow:
  # Grant access to the entire root filesystem
  - fs: {pathname: /, access: any}
  # Grant access to tempfs
  - fs: {pathname: /tmp, access: any}
  # Grant read/write access to /proc/sys/kernel/shm*
  - file: {pathname: /proc/sys/kernel/shm*, access: rw}

  # Grant access to the terminal, /dev/null, /dev/random, and /dev/urandom
```

```
13    - device: terminal
14    - device: null
15    - device: random
16
17    # Grant access to the entire networking stack
18    - net: any
19
20    # Enable Docker default capabilities
21    # All other capabilities are denied
22    - capability:
23      - chown
24      - dacoverride
25      - fsetid
26      - fowner
27      - mknod
28      - netraw
29      - setgid
30      - setuid
31      - setfcap
32      - setpcap
33      - netbindservice
34      - syschroot
35      - kill
36      - auditwrite
```

Compared with BPFBox, the BPFContain version of Docker's default policy is significantly simpler and fits more cleanly with Docker's AppArmor policy. This improvement is a direct result of a number of critical differences between BPFBox and BPFContain. Whereas BPFBox was designed for fine grained process-level confinement, BPFContain was directly designed with containers in mind. Since BPFContain policies are designed to be container specific, they are far more appropriate for a use case centered around the confinement of containers. In particular, BPFContain incorporates container semantics into its default policy enforcement, greatly simplifying the resulting policy. Further, changes to BPFContain's policy language, including the introduction of a coarser-grained filesystem rule and capa-

bility rules enables the resulting policy to more closely match the original Docker AppArmor policy.

To match Docker's default allow policy on filesystem access, the BPFContain policy includes a rule to enable any file operation on files within the root filesystem and tempfs. As with BPFBox, the point here is to match Docker's default policy without considering the security implications of granting full access to the entire root filesystem. We include another rule to enable similar access on the temporary filesystem. Despite the coarse granularity of these filesystem rules, BPFContain maintains a critical advantage over BPFBox and the original Docker policy. Due to its container-specific policy defaults, we can achieve Docker's fine-grained protection over procfs and sysfs without the need to specify it in the security policy.

As with the procfs and sysfs policy, BPFContain also includes sensible defaults for IPC and ptrace access. In particular, processes running within the same container are free to perform IPC with one another and ptrace one another, so long as the basic Unix access rights are satisfied (e.g. the process possesses CAP_PTRACE or is the direct ancestor of the tracee). In the case of signals and ptrace, these defaults directly match the Docker policy (c.f. Table 7.1). In other cases, these defaults are more secure than the Docker policy, while permits all other forms of IPC regardless of container membership.

To prevent a container from escaping confinement or interfering with the host, BPFContain prohibits the container from mounting filesystems, loading kernel modules, using eBPF, changing the system time, rebooting the system, or performing a number of other privileged operations. These defaults also match or exceed Docker's default policy, and thus may also be omitted from the BPFContain policy.

While many aspects of BPFContain's default enforcement closely match the default Docker policy, BPFContain's defaults remain strictly less permissive. For instance, the default Docker policy mandates that `/proc/sys/kernel/shm*` be accessible to containers, but BPFContain denies access to all procfs entries that do not belong to a container process. We define an exception to BPFContain's default procfs policy by adding an explicit allow rule on this pathname. Similarly, BPFContain's default policy forbids network access by default, and so we must explicitly grant the container permission to use the networking stack. Unlike Docker, BPFContain prohibits the use of any POSIX capability that is not directly specified in the policy file. Thus, we include an additional allow rule that mirrors the set of capabilities dropped by Docker at runtime.

The resulting BPFContain policy implements a strict superset of Docker's default confinement policy, despite being significantly simpler, and more centralized. Since BPFContain directly models the relationship between containerized processes and their resources, we can achieve significant portions of Docker's default policy for free. In many cases, this default enforcement is actually finer grained than the Docker defaults. In order to achieve the same coarse granularity as the Docker policy, we adjust the BPFContain policy by incorporating a few additional allow rules, granting access to specific filesystems, the networking stack, and POSIX capabilities.

7.3. Confining an Untrusted Container

We now examine perhaps the most obvious and practical use case for BPFContain: confining an untrusted container. For instance, consider a new container image,

freshly downloaded from Docker Hub, to be used during application development or testing. We assume that the system administrator does not trust this container image and wishes to confine the resulting container, preventing it from damaging the rest of the system, leaking information, or performing other potentially unwanted actions. For this purpose, we leverage an extremely simple BPFContain policy (essentially the canonical "Hello World" example) and demonstrate how it can be customized to match the container's specific needs. Listing 7.10 depicts the BPFContain policy.

Here, we assume that this policy is applied to a future version of BPFContain *with* full Docker integration and automatic filesystem policy. (Section 5.3.4 of Chapter 5 explains how this will be done.) Without these extensions, the policy author would need to manually grant access to files underlying the container's overlay filesystem, either using a filesystem allow rule or per-file allow rules. While this shouldn't add much additional complexity to the policy, the specific file rules would largely depend on the container's configuration, and so we omit the details of such a policy here. For a similar reason, we omit the corresponding BPFBox policy as well, since BPFBox does not deal in container-level semantics as BPFContain does.

Listing 7.10: Confining an untrusted container with BPFContain. Note that this policy requires some extensions on top of the existing BPFContain model, such as instrumenting the Docker container runtime.

```
name: untrusted-container
defaultTaint: true

allow:
  # Specify full path and access for volume mounts from the host
  - file: {pathname: /path/to/volume/mount, access: rw}
  # Repeat for other volume mounts as required...

  # If the container requires networking
```

```
10  # We could also define finer-grained access by replacing the "any"
11  # keyword with specific socket operations
12  - net: any
13
14  # If the container requires any capabilities
15  - capability: [dacoverride, dacreadsearch, netbindservice] # etc.
16
17  # Further extensions to the policy as required...
```

We define a default-tainted BPFContain policy called "untrusted-container". Just this policy alone should be enough to confine a simple container. BPFContain's default policy would prohibit network access, the use of any POSIX capabilities, access to any files outside of the container's overlay filesystem, and any operations that can impact the system as a whole. This default policy prevents entire classes of attacks, prohibiting the container from leaking outside information, changing global system parameters, loading code into the kernel, or forming unauthorized network connections. The reader is encouraged to revisit Figure 5.2 on page 137 of Chapter 5 for a depiction of how BPFContain's default enforcement works.

While the BPFContain default policy should be sufficient for simple use cases, more advanced container images may require some slight modification, introducing a few allow rules to define exceptions in BPFContain's protection boundary. For instance, suppose the container image requires a docker volume to be mounted at runtime. To support this use case, we define a file rule specifying the system path to the volume mount and the corresponding access pattern, such as `rw` for read and write access. All other accesses to the host filesystem remain denied. If the container requires access to the networking stack, we similarly define a net rule. Capability rules can be used to allow the container to use a selected subset of POSIX capa-

bilities, assuming it already possesses these capabilities at runtime. For example, we may wish to grant the container the `DAC_OVERRIDE` and `DAC_READ_SEARCH` capabilities to allow it to interact with the Docker volume we specified earlier, or the `NET_BIND_SERVICE` capability to allow it to bind to privileged ports.

7.4. Summary

This chapter has presented and compared BPFBox and BPFContain policies for various use cases. In particular, we examine how BPFBox and BPFContain can confine a simple web server and database deployment, how each system can be used to implement a policy resembling the Docker default policy, and how a future version of BPFContain can be used to confine an untrusted container with minimal effort. We find that each confinement mechanism has its respective strengths and weaknesses. BPFContain supports more access categories and combines semantically related accesses into the same rule types, simplifying policies and providing increased expressiveness. However, BPFBox's tainting and untainting semantics prove advantageous for complex deployments on the host system. Future iterations on BPFContain's default policy can greatly simplify existing container-specific policy semantics, shortening long and complex policies down to just a few lines.

Chapter 8.

Discussion and Concluding Remarks

This chapter discusses the relevance of BPFBox and BPFContain, positioning them as novel extensions to the existing confinement literature. We also examine limitations of both research systems and present opportunities for future work. Namely, we propose ways to address current limitations, improve the eBPF ecosystem for confinement use cases, add features to BPFBox and BPFContain, and conduct further research on the usability of both systems.

8.1. Research Questions Revisited

In Section 1.1, we proposed three research questions for this thesis. In this section, we revisit these research questions and discuss how the various chapters in this document answer them.

8.1.1. Answering RQ1

Research Question **RQ1** asks what difficulties exist in the current state of Linux confinement that might give rise to semantic gaps between security policies and the entities they are designed to lock down. It also asks what design goals a novel confinement mechanism would need to satisfy in order to rectify these difficulties.

In Chapter 3, we present a novel framing of the confinement problem, illustrating that semantic gaps arise due to inherent complexities and inter-dependence relationships that form between disparate confinement primitives. In light of this characterization, we identify a particular gap in container-level confinement that could be filled by a novel confinement mechanism that focuses on container semantics. We use the insights outlined in this chapter to inform a threat model and a set of design goals for a novel confinement mechanism.

8.1.2. Answering RQ2

Research Question **RQ2** asks how eBPF might be used to implement a novel confinement framework, what such a confinement framework would like, and how it might be made to model container semantics.

In Chapter 4 and Chapter 5, we answer this research question by presenting the design and implementation of two confinement mechanisms based on eBPF, BPF-Box and BPFContain. The former is designed to sandbox user processes while the latter is designed to confine containers by accounting for container semantics in its policy enforcement engine. We present a design pattern for encoding a security policy into eBPF maps and show that this policy can be augmented with information

about system state gathered by eBPF programs at runtime.

We find that eBPF offers numerous advantages when designing a novel confinement mechanism. In particular, eBPF programs used for enforcement can be safely and dynamically loaded into a running kernel, enabling us to iterate on and debug BPFBox and BPFContain as we would a userspace application. eBPF's safety guarantees and wide adoption in industry position BPFBox and BPFContain as adoptable alternatives to traditional LSMs. Using multiple eBPF program and map types, we can develop a rich confinement model that incorporates process- and container-level semantics into its policy decisions.

8.1.3. Answering RQ3

Research Question **RQ3** asks how an eBPF-based confinement solution might compare with existing confinement mechanisms, in terms of its performance and security. It also asks what improvements to the eBPF ecosystem might be needed to improve upon a proposed solution based on eBPF.

In Chapter 6, we present the results of several micro- and macro-benchmarks that compare the performance overhead of BPFBox and BPFContain with AppArmor. We find that, while BPFBox and BPFContain perform comparatively worse in some micro-benchmarks, their worst-case overhead is significantly lower than that of AppArmor. Moreover, the results of the macro-benchmarks indicate that this additional overhead is far more modest for computationally complex workloads. Furthermore, we hypothesize that it may be possible to optimize BPFContain in the future.

Chapter 6 also presents an informal security analysis which indicates that BPFBox and BPFContain provide practical security guarantees in the context of process- and container-level confinement, although there are opportunities to further develop their respective policy languages and enforcement engines. Chapter 7 presents example confinement policies for three distinct use cases, examining how BPFBox and BPFContain policies could be used to achieve realistic confinement goals.

Finally, we identify some key limitations of eBPF that could be addressed to further strengthen an eBPF-based confinement mechanism. Namely, we highlight the need for new eBPF helper functions to deal with pathname semantics in security policies. We also address the need for a formal evaluation of BPFBox, BPFContain, and eBPF itself in terms of their security and tamper resistance. We leave such an evaluation as a promising topic for future work (c.f. Section 8.2.4).

8.2. Limitations

In this section, we discuss some limitations of the BPFBox and BPFContain prototypes. While some limitations arise due to a lack of support for the correct primitives in the current eBPF ecosystem, others arise due to the prototypical nature of BPFBox and BPFContain as research systems. In both cases, we discuss how future iterations of BPFBox and BPFContain could address these limitations, either through extensions to the policy enforcement mechanism or future improvements to the eBPF ecosystem.

8.2.1. Semantic Issues in the Policy Language

It is challenging to refer to files from eBPF. In the kernel, files are generally uniquely described by an *inode* structure, which in turn maps to one or more pathnames via a *file* structure. Each inode belongs to a distinct filesystem and is uniquely enumerated within that filesystem by an inode number. In BPFBox and BPFContain, we uniquely identify inodes using a combination of their inode number and the unique device identifier of the filesystem on which the inode resides. While this is an effective technique for runtime monitoring, things begin to fall apart when dealing with a *user-facing* data store, such as a policy map.

While the kernel refers to files by their inodes within a filesystem, users do not. For the most part, userspace does not deal in inode-level semantics — instead, we deal in *pathnames*, a string that describes the path required to move from the filesystem root to a given file. Indeed, the BPFBox and BPFContain policy languages use pathnames rather than inodes to refer to files. Unfortunately, this creates an undesired dichotomy between the user-facing components of BPFBox and BPFContain and the kernelspace implementation. To resolve this dichotomy, we translate the pathnames into inode and device pairs at policy load-time. This is a workaround and is subject to several fundamental limitations. In particular, referring to a pathname that doesn't yet exist becomes difficult, as inode numbers do not yet exist; inodes that are deleted or freshly created at runtime must be treated as special cases, dynamically updating the policy as required; finally, globbing pathnames can result in an explosion in the size of maps storing file rules, as each globbed file is translated into a unique inode-device pair.

To resolve these issues, it would be ideal if we could refer to pathnames directly from BPF programs. In particular, a design using this capability might resolve pathnames within eBPF programs and define a finite state machine to match globbing rules over the pathname. Unfortunately, current support for pathname resolution in eBPF is primitive. Difficulties arise due to a few fundamental limitations imposed by the verifier and the eBPF runtime:

1. The verifier imposes a hard limit of 512 bytes of stack space for each BPF program. This makes it unrealistic to store strings on the stack, instead requiring that a buffer be allocated in the heap. In the context of BPF, this can only be done using a dummy map as a scratch buffer.

2. The verifier also imposes restrictions on how eBPF programs can loop and how these loops can access map data. Specifically, loops must provably terminate and any array access within a loop must be appropriately bounded by a fixed constant (to ensure no buffer overflows or similar issues). In practice, enforcing these restrictions is difficult, and the verifier errs on the side caution when reasoning about a loop is unclear. This can result in safe programs that manipulate long strings being erroneously rejected.

3. While helper functions can get around such restrictions, the current ecosystem for string manipulation helpers in eBPF is immature. For instance, Linux 5.10 added a `bpf_d_path()` helper [106] to extract pathnames from a kernel directory entry. However, this helper is only available for sleepable BPF programs, since allocating a buffer for the string can result in a page fault. Support for sleepable BPF is

very new and has not had a chance to mature; currently, sleepable programs are restricted to a small subset of LSM programs. Aside from pathname resolution, no other string helpers currently exist, although they have been on the radar of eBPF developers for some time.

Although the current state of the eBPF ecosystem makes it impossible for BPFBox and BPFContain to directly use pathname-based enforcement in the kernel, this will not necessarily be the case in the future. eBPF is in active development, and each subsequent kernel version adds new features and capabilities. For instance, the kernel community is currently working on a generic solution for sleepable BPF that will greatly expand the number of programs that can handle page faults. When this support arrives, it is likely that working with strings will become much easier. Thus, future versions of BPFBox and BPFContain will likely be able to incorporate pathname-based policy enforcement in their kernelspace implementations.

8.2.2. Fixed-Size Policy Maps

Currently, policy maps under BPFBox and BPFContain are of a fixed size, determined at policy load time. This is due to a fundamental limitation of eBPF: maps must be allocated at a fixed size and may not be directly resized at runtime. To get around this limitation, BPFBox and BPFContain tailor policy map size to the number of rules that will be loaded into the kernel. While this approach is effective, it would be better to have the ability to dynamically grow policy maps at runtime. This would enable BPFBox and BPFContain to support loading arbitrary length policies at runtime, and even runtime policy generation, without excessively large

map sizes.

As with string helpers in the previous section, the primary bottleneck preventing dynamically-sized maps is the adoption of sleepable BPF. Alexei Starovoitov expects that subsequent versions of the kernel will support dynamically allocated maps as the number of sleepable BPF programs increases. When support for this lands, BPFBox and BPFContain will be able to leverage this new functionality to greatly improve the memory efficiency of policy maps and support runtime policy generation from the eBPF side.

8.2.3. Performance Overhead

The performance results presented in Chapter 6 indicate that BPFBox and BPFContain have higher overhead than AppArmor in the OSBench micro-benchmarks. However, this is not indicative of their performance in the general case. In fact, BPFBox and BPFContain perform competitively with AppArmor in the kernel compilation macro-benchmarks, with similar overhead in the **Passive** and **Allow** test cases and significantly better overhead in the **Complaining case**. This suggests that the overhead of individual system calls is outweighed by the computational complexity of an instrumented application and the synchronous delays introduced by blocking system calls on I/O. These findings are further reflected when comparing the geometric means of each test configuration. While AppArmor performs better in the Apache web server benchmark, this is likely due to the fact that it does not properly enforce its network policy on modern Linux kernels (see Section 6.1). Moreover, neither BPFBox nor BPFContain has been deliberately optimized, suggesting

that there may be opportunities to improve performance in the future.

In addition to considering potential improvements to their base overhead, we must also consider the additional flexibility provided by an eBPF-based confinement mechanism. Using BPF CO-RE, BPFContain can be run on any supported kernel without the need to recompile or patch either the kernel of BPFContain itself. The dynamic nature of eBPF means that software bugs or vulnerabilities in the enforcement engine can be fixed without the need to update the kernel or even reboot the system. Moreover, eBPF programs are verified for safety, meaning that they are far less likely to expose additional attack surfaces in the kernel than a kernel patch or loadable kernel module.

8.2.4. Security Guarantees

In Chapter 6, we presented a detailed security analysis of the BPFBox and BPFContain designs. However, this informal analysis is weak evidence for the actual security guarantees of these research prototypes. In the future, we could conduct a formal security evaluation of BPFContain, measuring its ability to confine a container using a dedicated security test suite. Such a test suite would involve a combination of existing (vulnerable) container images and accompanying CVEs. We would then attach a BPFContain policy to the resulting container and determine its ability to (1) accommodate the basic functionality of the container; and (2) prevent the exploitation of any related CVEs. These tests would then enable us to establish BPFContain as a useful security mechanism for confining containers.

While others [8, 28] have examined eBPF's utility for security use cases, this thesis

presents the earliest research into defining security policies using eBPF maps. Currently, BPFBox and BPFContain protect themselves from a confined adversary by instrumenting eBPF programs on the `bpf(2)` system call code path. However, in order to present a rigorous security argument, we must also formally verify the security of BPFContain's eBPF programs and maps to show that a confined adversary cannot tamper with them.

Another important aspect underpinning the security of solutions like BPFBox and BPFContain is the security of eBPF itself as well as the KRSI LSM programs and the underlying LSM hooks themselves. To date, no formal analysis exists to verify any of these components. It may not be possible to formally verify that LSM hooks provide complete mediation due to the complexity of the Linux kernel [76]. However, due to the simplicity of eBPF and KRSI, it could be feasible to formally verify their security guarantees within the context of an unverified kernel. We leave such formal analysis to future work.

8.3. Future Work and Research Directions

This section discusses opportunities for future work and potential research directions. We specifically examine opportunities to further evaluate the effectiveness of BPFBox and BPFContain as well as potential extensions that can improve BPFContain's container-specific policies. We also consider potential avenues for implementing automatic policy generation in BPFContain.

8.3.1. The Need for a User Study

While the BPFBox and BPFContain policy languages are designed to be simple, the usability argument behind their design is as of yet untested. In particular, both prototypes could benefit from a user study. A user study could offer numerous insights into how users interact with the policy languages, whether the resulting enforcement actions meet expectations, and how each system compares with equivalents like SELinux or AppArmor from a usability perspective. Such insights could be used to inform design directions for future iterations of the policy language or to establish a stronger basis for comparison between BPFBox, BPFContain, and alternative confinement mechanisms.

User studies have proven useful for gleaning insights about prior work in the confinement space. Schreuders *et al.* [125] conducted a user study examining the usability of their FBAC-LSM security module, AppArmor, and SELinux, to establish a basis for comparison between alternative policy schemes. Their work revealed insights into how users manage expectations when writing security policy and the semantic gap between existing policy languages and user expectations. As systems that incorporate their own policy language design, BPFBox and BPFContain could greatly benefit from a similar user study.

8.3.2. OCI Specification and Docker Integration

As a container-specific confinement solution, integration with the OCI specification and Docker container runtime are a natural path forward for BPFContain. The OCI specification is an open, platform-agnostic standard for defining a container

image using JSON [89]. Container runtimes like Docker generate and parse an OCI specification for a container image before running it. Integrating BPFContain with the OCI specification would enable BPFContain policies to be expressed directly within an image's OCI representation and enable BPFContain to infer certain aspects of its policy from existing OCI data.

Another potential avenue for extending BPFContain is to use eBPF programs to trace the container runtime (e.g. Docker's `containerd` and `moby`), mediating the setup phase to enforce policy on specific aspects of container construction. This approach could also be used to infer policy, for example by interposing on the filesystem mounts created by the container runtime to infer filesystem policy or observing the creation of the Docker network interface to inform network policy. Using uprobes, BPFContain could also associate a Docker container with a given container image, enabling it to automatically apply policy when a specific container image starts.

8.3.3. Fine-Grained Network Policy

Currently, BPFBox and BPFContain both expose a highly coarse-grained network policy. In particular, network policy only operates at the socket level, and does not consider more nuanced access controls, such as filtering by IP address. This means that specifying access to a network resource essentially gives a process of container global access to network resources, so long as any requested access conforms to a subset of specified operations. While this is not strictly problematic for applications that do not require network access, it quickly becomes an overprivilege issue for applications that do.

Under the current implementation, provisioning a fine-grained network policy would require a netfilter-based firewall such as iptables. While this is not strictly an issue, one of the main goals of BPFBox and BPFContain is to eliminate the need to recombine existing policy mechanisms. Therefore, it would be best if we could incorporate finer-grained network policy into future iterations. Such a network policy is possible to enforce using eBPF; rather than LSM probes, we would leverage another program type, `TC_CLSACT`, that enables finer-grained network filtering on individual packets.

The `TC_CLSACT` (short for Traffic Control, Classifier and Action) program type is a socket filter that can discriminate between network traffic at the protocol-level by parsing packet headers. In addition to classifying traffic, this program type can make filtering decisions, deciding whether a packet should move on in the kernel's networking stack or be dropped. Using this program type to enforce network policy would enable BPFContain to discriminate between source and destination addresses and ports when making network policy decisions on ingress and egress traffic. The policy language could then be extended to specify specific address patterns that should be matched in order for a network connection to be allowed. Extending the BPFContain policy in this way would mark a significant improvement in BPFContain's network security guarantees.

8.3.4. BPFContain Policy Generation

The version of BPFContain presented in this thesis requires a policy author to write security policies by hand. Future iterations of BPFContain should support

automatic or at least semi-automatic policy generation to ease the burden on users and facilitate security policy authorship. This can be accomplished in one of two ways. We discuss each in turn.

1. We could extend BPFContain's eBPF programs with the ability to automatically generate policy in real time. This approach is similar to anomaly detection, establishing a normal behavioural profile for a container during a training phase and enforcing security policy once a normal profile has been established. Implementing such profile generation would require minimal changes to BPFContain's existing enforcement mechanisms, although further investigation is required to determine the correct approach for minimizing false positives and false negatives in policy generation.

2. We could implement policy generation from audit logs, similar in spirit to SELinux's `audit2allow` [131] or AppArmor's `aa-logprof` [6]. Accomplishing this would require a few changes to BPFContain's log generation. In particular, we would need a way to map inode and filesystem number pairs back to the underlying host pathnames so that they can be correctly logged in userspace. The alternative is translating or caching these pairs in userspace, which may prove to be too expensive in practice. In order to support pathname translation in BPFContain, additional upstream work in the eBPF subsystem is required to enable pathname resolution in more LSM hooks, as outlined in Section 8.2.1.

8.4. Improving the Status Quo

In this section, we discuss how BPFBox and BPFContain improve upon the status quo in confinement, with a particular emphasis on how their unique properties encourage application- and container-specific confinement, and promote local policy variation. We also examine how their implementation as eBPF-based security solutions positions them as highly adoptable alternatives to traditional kernel security mechanisms with the potential to drive further innovation going forward.

8.4.1. Application-Specific and Container-Specific Policies

Due to their simplicity and flexibility, BPFBox and BPFContain encourage a different kind of confinement compared with existing LSM-based solutions. Rather than focusing on global, system-wide MAC policy enforcement like SELinux [130] or AppArmor [42], BPFBox and BPFContain focus on application-specific and container-specific confinement respectively. The former enables lightweight, ad-hoc confinement of individual Linux processes, while the latter extends this model to work with container semantics. This represents a stark contrast over existing work in the confinement space which generally focuses on reusing existing primitives designed for global enforcement. Rather than outright simple policies, these confinement frameworks compile down into hundreds or even thousands of lines of policy for the underlying confinement primitive.

The *application-specific* approach of BPFBox takes a different path — a BPFBox policy directly applies to the BPFBox enforcement engine, without relying on existing primitives like SELinux, AppArmor, or Seccomp-bpf. The result is policies

that are far simpler without being too coarse-grained or abstracted beyond the point of auditability. Further, application-specific policies afford a great deal of flexibility to the end user; rather than authoring complex policies that cover the entire system, instead they may focus on the specific behaviours that they want their applications to exhibit. In turn, this flexibility means that different types of users can leverage BPFBox in different ways. Application developers can write fine-grained policies that enforce behaviours at the function-call-level. Conversely, end users can deploy custom BPFBox policies to restrict specific behaviours, such as access to the home directory.

BPFContain extends the BPFBox model to be *container specific*. Like BPFBox, much of the implicit strength inherent in this model is that it does not rely on existing primitives and does not attempt to enforce policy over the entire system. Instead, we focus on individual containers, identifying the specific OS interfaces needed for the container to function. This approach introduces additional advantages specific to the container use case, on top of the advantages already present in the BPFBox design. By including container semantics as part of its internal model of the system, BPFContain can enforce highly nuanced policy defaults, defining an enforcement boundary around the container. This boundary, in turn, simplifies resulting policies by allowing the policy author to focus on which external interfaces the container needs, without worrying about specifying access to internal resources. In this way, BPFContain policies mirror the process of provisioning resources in a virtual machine.

8.4.2. Encouraging Local Policy Variation

An implicit advantage of simple yet flexible confinement policies is that they encourage *policy variation*. In computer security, *diversity* is an area that has seen some exploration in the past [97, 105, 111, 135]. The idea is deceptively simple, inspired by biological sources of diversity found in nature. Attackers rely on similarities between system for widespread exploitation; for instance, consider a set of deployed systems all running the same vulnerable version of a piece of a software with similar configurations. Since each of these systems have the same vulnerability, exploiting one system is much like exploiting the others; thus, the knowledge and effort required to exploit each system will be roughly equivalent to exploiting just one. The point of introducing diversity into software deployments and configurations is to confound this basic assumption. On a macroscopic scale, increased diversity improves the security of the entire population.

Classically, computer diversity has primarily been explored through source- or binary-level variations [105, 111, 135]. Solutions recombine existing software in new ways, such that each deployment has a unique code footprint. However, a primary limiting factor to the adoption of such security solutions is that they fly in the face of the traditional approach to computing. Namely, we want our software to be as predictable, reliable, and effective as possible. The same input into a piece of software should produce the same (or at least predictable) output, regardless of the underlying system configuration. Source- or binary-level software diversity invalidates this assumption, by introducing potential variations in common code paths.

Unlike traditional sources of diversity in computing, *policy-level diversity* has the potential to be quite effective while maintaining the underlying assumption that software should "just work". Security policies may be tailored to some locally-desired use case, only invalidating code paths that would never be taken to begin with. For example, consider an Apache web server configuration. In one deployment, it may be necessary to support the execution of CGI helper scripts, whereas another may only need to serve static webpages. Each configuration could use a different security policy, based on the needs of the local deployment. A natural diversity arises as an emergent property of this model when scaled up over thousands of deployments across thousands of applications. Since BPFBox and BPFContain policies are so simple yet so flexible, they naturally encourage precisely the sort of local policy variations required to achieve policy-level diversity at scale.

8.4.3. eBPF, Adoptability, and Future Innovation

A third improvement over the status quo in confinement lies in BPFBox and BPFContain's novelty as eBPF-based confinement implementations. Whereas existing kernel security mechanisms are based on kernel patches or loadable kernel modules, BPFBox and BPFContain leverage eBPF for dynamic runtime security monitoring, verified for safety and correctness before being loaded into the kernel. An eBPF-based implementation affords BPFBox and BPFContain a number of advantages over traditional security solutions.

1. **eBPF is already widely used in production Linux environments.** At the time of writing this thesis, major software companies like Facebook, Netflix, and

Google [65] already use eBPF in production servers for performance and security monitoring, the implementation of L4 routing and load balancing algorithms, and other various use cases. In fact, the KRSI patch that provides the LSM probes used by BPFBox and BPFContain was initially developed at Google for dynamic security monitoring use cases [129]. As time goes on, eBPF-based kernel code is seeing increasing deployment across multidisciplinary areas of industry. This widespread deployment is a key incentive toward the adoption of security mechanisms like BPFBox and BPFContain. Not only is this widespread adoption an emergent phenomenon from eBPF's safety and flexibility properties, but it is also a valuable case study in how eBPF is reshaping the way we think about kernel observability in practice.

2. **The barrier to entry for running new eBPF code on a production system is much lower than a new kernel module or kernel patch.** Since eBPF programs and maps can be dynamically loaded into the kernel and are checked for safety and correctness before they are allowed to run, the barrier to entry for running a new eBPF-based implementation in production is far lower than that of a kernel module or kernel patch. eBPF programs are far less likely to contain memory safety errors or other software bugs that plague kernel code in practice. Due to new technologies like BPF CO-RE [102], eBPF-based solutions are also far more portable across different kernel versions and configurations. These factors combined make BPFBox and particularly BPFContain far more adoptable as novel kernel security mechanisms. Future security implementations based on eBPF can also enjoy these advantages.

3. **eBPF enables rapid prototyping and deployment of kernel security mechanisms.** Due to its dynamic nature, safety, and portability across kernels, it is far easier to rapidly prototype, test, and deploy novel security solutions based on eBPF. In the case of BPFBox and BPFContain, this enables rapid prototyping of the policy language and enforcement engine, and makes it easy to incorporate novel extensions on top of these research systems, redefining key aspects of policy enforcement and adding new data sources from the kernel and userspace programs. In the future, we will likely see eBPF positioned as a key enabling factor behind the rapid development of novel kernel security extensions, further combining system observability with dynamic policy enforcement.

8.5. Conclusion

This thesis has presented the design, implementation, and motivation behind BPFBox and BPFContain, two novel confinement mechanisms for the Linux kernel, with an emphasis on simple yet precise policies, application- and container-specific confinement, and high adoptability. We analyze the performance of these research systems and find that they are competitive with existing confinement implementations while providing superior flexibility for local confinement. While undoubtedly useful on their own, perhaps the greatest value provided by these research systems is as a proof of concept, demonstrating the value of eBPF-based confinement solutions.

A myriad of potential extensions on top of BPFBox and BPFContain can provide increased security and flexibility. Future optimizations in the BPFBox and BPFContain enforcement engines can further improve their performance, making

them more competitive with existing confinement solutions. Extensions on top of the existing BPF ecosystem can help to position it as the dominant framework for implementing future kernel-based security solutions. Such extensions can help to improve BPFBox and BPFContain as well as promote the adoption of other security mechanisms based on eBPF.

In the future, a security solution based on eBPF may comprise a generic framework, capable of loading and managing multiple BPF program types to hook into any aspect of system behaviour. Such a solution would encompass the capabilities of BPFBox and BPFContain and potentially expand them to include intrusion detection, network filtering, software hot patches, and beyond. In the short term, improvements on top of the eBPF ecosystem as well as BPFBox and BPFContain can make incremental progress towards realizing this goal.

Bibliography

[1] Amith Raj MP, A. Kumar, *et al.*, "Enhancing Security of Docker Using Linux Hardening Techniques," in *2016 2nd International Conference on Applied and Theoretical Computing and Communication Technology (iCATccT)*, 2016, pp. 94–99. DOI: 10.1109/ICATCCT.2016.7911971.

[2] J. P. Anderson, "Computer Security Technology Planning Study (Vol. I and II, "Anderson Report")," Box 42, Fort Washington, PA, 19034 USA, Tech. Rep., 1972. [Online]. Available: https://apps.dtic.mil/sti/citations/AD0758206.

[3] Jonathan Anderson, "A Comparison of Unix Sandboxing Techniques," *FreeBSD Journal*, 2017. [Online]. Available: http://www.engr.mun.ca/~anderson/publications/2017/sandbox-comparison.pdf.

[4] AppArmor, *aa-easyprof(8)*, Linux user's manual. [Online]. Available: https://manpages.ubuntu.com/manpages/precise/man8/aa-easyprof.8.html.

[5] AppArmor, *aa-genprof(8)*, Linux user's manual. [Online]. Available: https://manpages.ubuntu.com/manpages/precise/man8/aa-genprof.8.html.

[6] AppArmor, *aa-logprof(8)*, Linux user's manual. [Online]. Available: https://manpages.ubuntu.com/manpages/precise/man8/aa-logprof.8.html.

[7] Aqua Security, *libbpfgo*, GitHub repository. [Online]. Available: https://github.com/aquasecurity/libbpfgo (visited on 06/07/2021).

[8] Aqua Security, *Tracee*, Github repository. [Online]. Available: https://github.com/aquasecurity/tracee (visited on 06/29/2021).

[9] M Ali Babar and Ben Ramsey, "Understanding Container Isolation Mechanisms for Building Security-Sensitive Private Cloud," CREST, The University of Adelaide, Australia, Tech. Rep., 2017. [Online]. Available: https://malibabar.files.wordpress.com/2017/04/container_isolation_secure_cloud_2017.pdf.

[10] David Barrera, Jeremy Clark, *et al.*, "Understanding and Improving App Installation Security Mechanisms Through Empirical Analysis of Android," in *SPSM'12, Proceedings of the Workshop on Security and Privacy in Smartphones and Mobile Devices, Co-located with CCS 2012, October 19, 2012, Raleigh, NC, USA*, Ting Yu, William Enck, and Xuxian Jiang, Eds., ACM, 2012, pp. 81–92. DOI: 10.1145/2381934.2381949.

[11] Maxime Bélair, Sylvie Laniepce, and Jean-Marc Menaud, "Leveraging Kernel Security Mechanisms to Improve Container Security: a Survey," in *Proceedings of the 14th International Conference on Availability, Reliability and Security, ARES 2019, Canterbury, UK, August 26-29, 2019*, ACM, 2019, 76:1–76:6. DOI: 10.1145/3339252.3340502.

[12] David Elliott Bell, "Looking Back at the Bell-La Padula Model," in *21st Annual Computer Security Applications Conference (ACSAC 2005), 5-9 December 2005, Tucson, AZ, USA*, IEEE Computer Society, 2005, pp. 337–351. DOI: 10.1109/CSAC.2005.37.

[13] Luciano Bello and Alejandro Russo, "Towards a Taint Mode for Cloud Computing Web Applications," in *Proceedings of the 2012 Workshop on Programming Languages and Analysis for Security, PLAS 2012, Beijing, China, 15 June, 2012*, Sergio Maffeis and Tamara Rezk, Eds., ACM, 2012, p. 7. DOI: 10.1145/2336717.2336724.

[14] Eric W Biederman, "Multiple Instances of the Global Linux Namespaces," in *Proceedings of the Linux Symposium*, vol. 1, Linux Symposium, 2006, pp. 101–112. [Online]. Available: https://www.kernel.org/doc/ols/2006/ols2006v1-pages-101-112.pdf.

[15] Matt Bishop and Michael Dilger, "Checking for Race Conditions in File Accesses," *Computing Systems*, vol. 9, no. 2, pp. 131–152, 1996. [Online]. Available: https://static.usenix.org/publications/compsystems/1996/spr_bishop.pdf.

[16] *bpf(2)*, Linux Programmer's Manual. [Online]. Available: https://www.man7.org/linux/man-pages/man2/bpf.2.html (visited on 06/08/2021).

[17] *bpf(4)*, Device Drivers Manual. [Online]. Available: https://man.openbsd.org/bpf (visited on 06/08/2021).

[18] *bpf(4)*, BSD Kernel Interfaces Manual. [Online]. Available: https://www.unix.com/man-page/FreeBSD/4/bpf (visited on 06/08/2021).

[19] Kelly Brady, Seung Moon, *et al.*, "Docker Container Security in Cloud Computing," in *10th Annual Computing and Communication Workshop and Conference*, IEEE, 2020, pp. 975–980. DOI: 10.1109/CCWC47524.2020.9031195.

[20] T. Bray, Ed., *The JavaScript Object Notation (JSON) Data Interchange Format*, RFC, 2017. [Online]. Available: https://datatracker.ietf.org/doc/html/rfc7159 (visited on 07/16/2021).

[21] Neil Brown, *Overlay Filesystem*. [Online]. Available: https://www.kernel.org/doc/html/latest/filesystems/overlayfs.html (visited on 06/29/2021).

[22] Thanh Bui, "Analysis of Docker Security," *CoRR*, vol. abs/1501.02967, 2015. arXiv: 1501.02967. [Online]. Available: http://arxiv.org/abs/1501.02967.

[23] Bryan M. Cantrill, Michael W. Shapiro, and Adam H. Leventhal, "Dynamic Instrumentation of Production Systems," in *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ser. ATEC '04, Boston, MA: USENIX Association, 2004, pp. 2–2. [Online]. Available: https://www.usenix.org/legacy/publications/library/proceedings/usenix04/tech/general/full_papers/cantrill/cantrill.pdf.

[24] *capabilities(7)*, Linux user's manual. [Online]. Available: https://linux.die.net/man/7/capabilities.

[25] Hao Chen, David A. Wagner, and Drew Dean, "Setuid Demystified," in *Proceedings of the 11th USENIX Security Symposium, San Francisco, CA, USA, August 5-9, 2002*, Dan Boneh, Ed., USENIX, 2002, pp. 171–190. [Online]. Available: http://www.usenix.org/publications/library/proceedings/sec02/chen.html.

[26] Winnie Cheng, Qin Zhao, *et al.*, "TaintTrace: Efficient Flow Tracing with Dynamic Binary Rewriting," in *Proceedings of the 11th IEEE Symposium on Computers and Communications (ISCC 2006), 26-29 June 2006, Cagliari, Sardinia, Italy*, Paolo Bellavista, Chi-Ming Chen, *et al.*, Eds., IEEE Computer Society, 2006, pp. 749–754. DOI: 10.1109/ISCC.2006.158.

[27] Erika Chin and David A. Wagner, "Efficient Character-Level Taint Tracking for Java," in *Proceedings of the 6th ACM Workshop On Secure Web Services, SWS 2009, Chicago, Illinois, USA, November 13, 2009*, Ernesto Damiani, Seth Proctor, and Anoop Singhal, Eds., ACM, 2009, pp. 3–12. DOI: 10.1145/1655121.1655125.

[28] Cilium, *Cilium*, Github repository. [Online]. Available: https://github.com/cilium/cilium (visited on 06/29/2021).

[29] Cilium and Cloudflare, *ebpf*, GitHub repository. [Online]. Available: https://github.com/cilium/ebpf (visited on 06/07/2021).

[30] James A. Clause, Wanchun Li, and Alessandro Orso, "Dytan: A Generic Dynamic Taint Analysis Framework," in *Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2007, London, UK, July 9-12, 2007*, David S. Rosenblum and Sebastian G. Elbaum, Eds., ACM, 2007, pp. 196–206. DOI: 10.1145/1273463.1273490.

[31] Samuel Clay, *How a Docker Footgun Led to a Vandal Deleting NewsBlur's MongoDB Database*, NewsBlur company blog, 2021. [Online]. Available: https://blog.newsblur.com/2021/06/28/story-of-a-hacking/ (visited on 07/04/2021).

[32] Théo Combe, Antony Martin, and Roberto Di Pietro, "To Docker or Not to Docker: A Security Perspective," *IEEE Cloud Comput.*, vol. 3, no. 5, pp. 54–62, 2016. DOI: 10.1109/MCC.2016.100.

[33] Juan José Conti and Alejandro Russo, "A Taint Mode for Python via a Library," in *Information Security Technology for Applications - 15th Nordic Conference on Secure IT Systems, NordSec 2010, Espoo, Finland, October 27-29, 2010, Revised Selected Papers*, Tuomas Aura, Kimmo Järvinen, and Kaisa Nyberg, Eds., ser. Lecture Notes in Computer Science, vol. 7127, Springer, 2010, pp. 210–222. DOI: 10.1007/978-3-642-27937-9_15.

[34] *Control Group v2*, Linux Kernel Documentation. [Online]. Available: https://www.kernel.org/doc/html/latest/admin-guide/cgroup-v2.html (visited on 06/26/2021).

[35] Kees Cook, *[PATCH] security: Yama LSM*, Jun. 2010. [Online]. Available: https://lkml.org/lkml/2010/6/21/407 (visited on 07/12/2021).

[36] Fernando J. Corbató and Victor A. Vyssotsky, "Introduction and Overview of the Multics System," in *Proceedings of the 1965 fall joint computer conference, part I, AFIPS 1965 (Fall, part I), Las Vegas, Nevada, USA, November 30 - December 1, 1965*, Robert W. Rector, Ed., ACM, 1965, pp. 185–196. DOI: 10.1145/1463891.1463912.

[37] Jonathan Corbet, "A bid to resurrect Linux capabilities," *LWN*, 2006. [Online]. Available: https://lwn.net/Articles/199004/.

[38] Jonathan Corbet, "File-based capabilities," *LWN*, 2006. [Online]. Available: https://lwn.net/Articles/211883/.

[39] Jonathan Corbet, "OpenBSD's unveil()," *LWN.net*, Dec. 2018. [Online]. Available: https://lwn.net/Articles/767137/.

[40] Jonathan Corbet, "KRSI — The Other BPF Security Module," *LWN.net*, Dec. 2019. [Online]. Available: https://lwn.net/Articles/808048/.

[41] Jonathan Corbet, Alessandro Rubini, and Greg Kroah-Hartman, *Linux Device Drivers*, 3rd. O'Reilly, 1998. [Online]. Available: https://lwn.net/Kernel/LDD3/.

[42] Crispin Cowan, Steve Beattie, *et al.*, "SubDomain: Parsimonious Server Security," in *Proceedings of the 14st Large Installation Systems Administration Conference (LISA)*, New Orleans, LA, United States: USENIX Association, 2000. [Online]. Available: https://www.usenix.org/legacy/event/lisa2000/full_papers/cowan/cowan.pdf.

[43] Brooks Davis, Robert N. M. Watson, *et al.*, "CheriABI: Enforcing Valid Pointer Provenance and Minimizing Pointer Privilege in the POSIX C Runtime Environment," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2019, Providence, RI, USA, April 13-17, 2019*, Iris Bahar, Maurice Herlihy, *et al.*, Eds., ACM, 2019, pp. 379–393. DOI: 10.1145/3297858.3304042.

[44] Alessandro Decina, "BPF target support," Patch, Jun. 2021. [Online]. Available: https://github.com/rust-lang/rust/pull/79608.

[45] Peter J. Denning, "Virtual Memory," *ACM Comput. Surv.*, vol. 2, no. 3, pp. 153–189, 1970. DOI: 10.1145/356571.356573.

[46] Jack B. Dennis and Earl C. Van Horn, "Programming Semantics for Multiprogrammed Computations," *Commun. ACM*, vol. 9, no. 3, pp. 143–155, 1966. DOI: 10.1145/365230.365252.

[47] Mark S. Dittmer and Mahesh V. Tripunitara, "The UNIX Process Identity Crisis: A Standards-Driven Approach to Setuid," in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, Scottsdale, AZ, USA, November 3-7, 2014*, Gail-Joon Ahn, Moti Yung, and Ninghui Li, Eds., ACM, 2014, pp. 1391–1402. DOI: 10.1145/2660267.2660333.

[48] Docker, *Docker Hub*. [Online]. Available: https://hub.docker.com/ (visited on 06/29/2021).

[49] Docker Developers, *AppArmor security profiles for Docker*, Developer documenation. [Online]. Available: https://docs.docker.com/engine/security/apparmor/ (visited on 05/29/2021).

[50] Docker Developers, *Docker Security*, Developer documenation. [Online]. Available: https://docs.docker.com/engine/security/ (visited on 06/29/2021).

[51] Docker Developers, *profiles/apparmor/template.go*, Github repository. [Online]. Available: https://github.com/moby/moby/blob/master/profiles/apparmor/template.go (visited on 05/29/2021).

[52] Stephen Dranger, Robert H. Sloan, and Jon A. Solworth, "The Complexity of Discretionary Access Control," in *Advances in Information and Computer Security, First International Workshop on Security, IWSEC 2006, Kyoto, Japan, October 23-24, 2006, Proceedings*, Hiroshi Yoshiura, Kouichi Sakurai, *et al.*, Eds., ser. Lecture Notes in Computer Science, vol. 4266, Springer, 2006, pp. 405–420. DOI: 10.1007/11908739_29.

[53] Michael Eder, "Hypervisor- vs. Container-Based Virtualization," *Future Internet (FI) and Innovative Internet Technologies and Mobile Communications (IITM)*, vol. 1, 2016. DOI: 10.2313/NET-2016-07-1_01.

[54] Jake Edge, "A Seccomp Overview," *LWN.net*, 2015. [Online]. Available: https://lwn.net/Articles/656307/.

[55] Andrey Ermolinskiy, Sachin Katti, *et al.*, "Towards Practical Taint Tracking," UC Berkeley, Tech. Rep. UCB/EECS-2010-92, 2010. [Online]. Available: http://www.eecs.berkeley.edu/Pubs/TechRpts/2010/EECS-2010-92.html.

[56] Clark Evans, Ingy döt Net, and Oren Ben-Kiki, *YAML: YAML Ain't Markup Language*, version 1.2, Official specification, 2021. [Online]. Available: http://yaml.org/spec/1.2/spec.html (visited on 07/16/2021).

[57] William Findlay, "Host-Based Anomaly Detection with Extended BPF," Honours Thesis, Carleton University, 2020. [Online]. Available: https://www.cisl.carleton.ca/~will/written/coursework/undergrad-ebpH-thesis.pdf.

[58] William Findlay, David Barrera, and Anil Somayaji, "BPFContain: Fixing the Soft Underbelly of Container Security," *CoRR*, vol. abs/2102.06972, 2021. arXiv: 2102.06972. [Online]. Available: https://arxiv.org/abs/2102.06972.

[59] William Findlay, Anil Somayaji, and David Barrera, "BPFBox: Simple Precise Process Confinement with eBPF," in *Proceedings of the 2020 ACM SIGSAC Conference on Cloud Computing Security Workshop*, ser. CCSW'20, Virtual Event, USA: Association for Computing Machinery, 2020, pp. 91–103. DOI: 10.1145/3411495.3421358.

[60] Flatpak, *Sandbox Permissions*, 2020. [Online]. Available: https://docs.flatpak.org/en/latest/sandbox-permissions.html (visited on 07/02/2021).

[61] Vinod Ganapathy, Trent Jaeger, and Somesh Jha, "Automatic Placement of Authorization Hooks in the Linux Security Modules Framework," in *Proceedings of the 12th ACM Conference on Computer and Communications Security, CCS 2005, Alexandria, VA, USA, November 7-11, 2005*, Vijay Atluri, Catherine A. Meadows, and Ari Juels, Eds., ACM, 2005, pp. 330–339. DOI: 10.1145/1102120.1102164.

[62] Xing Gao, Zhongshu Gu, *et al.*, "Houdini's Escape: Breaking the Resource Rein of Linux Control Groups," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, CCS 2019, London, UK, November 11-15, 2019*, Lorenzo Cavallaro, Johannes Kinder, *et al.*, Eds., ACM, 2019, pp. 1073–1086. DOI: 10.1145/3319535.3354227.

[63] Seyedhamed Ghavamnia, Tapti Palit, *et al.*, "Confine: Automated System Call Policy Generation for Container Attack Surface Reduction," in *23rd International Symposium on Research in Attacks, Intrusions and Defenses, RAID 2020, San Sebastian, Spain, October 14-15, 2020*, Manuel Egele and Leyla Bilge, Eds., USENIX Association, 2020, pp. 443–458. [Online]. Available: https://www.usenix.org/conference/raid2020/presentation/ghavanmnia.

[64] Ian Goldberg, David Wagner, *et al.*, "A Secure Environment for Untrusted Helper Applications (Confining the Wily Hacker)," in *Proceedings of the Sixth USENIX UNIX Security Symposium*, The USENIX Association, 1996. [Online]. Available: https://www.usenix.org/legacy/publications/library/proceedings/sec96/full_papers/goldberg/goldberg.pdf.

[65] Brendan Gregg, *BPF Performance Tools*. Addison-Wesley Professional, 2019, ISBN: 0-13-655482-2.

[66] Brendan Gregg, *BPF binaries: BTF, CO-RE, and the future of BPF perf tools*, Blog post, Nov. 2020. [Online]. Available: http://www.brendangregg.com/blog/2020-11-04/bpf-co-re-btf-libbpf.html (visited on 06/07/2021).

[67] Brendan Gregg and Jim Mauro, *DTrace: Dynamic Tracing in Oracle Solaris, Mac OS X, and FreeBSD*, 1st. Prentice Hall, 2011.

[68] Grsecurity, *Memory Corruption Defenses*, Official website, 2021. [Online]. Available: https://grsecurity.net/featureset/memory_corruption (visited on 07/08/2021).

[69] Toshiharu Harada, Takashi Horie, and Kazuo Tanaka, "Task Oriented Management Obviates Your Onus on Linux," Jan. 2004.

[70] Norman Hardy, "The Confused Deputy (or why capabilities might have been invented)," *ACM SIGOPS Oper. Syst. Rev.*, vol. 22, no. 4, pp. 36–38, 1988. DOI: 10.1145/54289.871709.

[71] Lihui Hu, Jean Mayo, and Charles Wallace, "An Empirical Study of Three Access Control Systems," in *The 6th International Conference on Security of Information and Networks, SIN '13, Aksaray, Turkey, November 26-28, 2013*, Atilla Elçi, Manoj Singh Gaur, *et al.*, Eds., ACM, 2013, pp. 287–291. DOI: 10.1145/2523514.2523550.

[72] Andrew Hurst, *Analysis of Perl's Taint Mode*, 2004. [Online]. Available: http://hurstdog.org/papers/hurst04taint.pdf.

[73] IBM Cloud Education, *LAMP Stack Explained*, IBM Cloud Education article, 2019. [Online]. Available: https://www.ibm.com/cloud/learn/lamp-stack-explained (visited on 07/31/2021).

[74] IOVisor, *bcc*, GitHub repository. [Online]. Available: https://github.com/iovisor/bcc (visited on 06/07/2021).

[75] IOVisor, *gobpf*, GitHub repository. [Online]. Available: https://github.com/iovisor/gobpf (visited on 06/07/2021).

[76] Trent Jaeger, *Operating System Security*, Ravi Sandhu, Ed. Morgan & Claypool Publishers, 2008, ISBN: 978-1598292121.

[77] K. Jain and R. Sekar, "User-Level Infrastructure for System Call Interposition: A Platform for Intrusion Detection and Confinement," in *Proceedings of the Network and Distributed System Security Symposium, NDSS 2000, San Diego, California, USA*, The Internet Society, 2000. [Online]. Available: https://www.ndss-symposium.org/ndss2000/user-level-infrastructure-system-call-interposition-platform-intrusion-detection-and-confinement/.

[78] John Johansen, "[PATCH] AppArmor: Patch to Provide Compatibility with v2.x Net Rules," Kernel patch, Jun. 2018. [Online]. Available: https://raw.githubusercontent.com/openSUSE/kernel-source/master/patches.suse/apparmor-compatibility-with-v2.x-net.patch.

[79] Poul-Henning Kamp and Robert NM Watson, "Jails: Confining the Omnipotent Root," in *Proceedings of the 2nd International SANE Conference*, vol. 43, 2000, p. 116. [Online]. Available: https://ivanlef0u.fr/repo/madchat/sysadm/bsd/kamp.pdf.

[80] Kubernetes, *Kubernetes*, Official website, 2021. [Online]. Available: https://kubernetes.io/ (visited on 07/30/2021).

[81] Soonhong Kwon and Jong-Hyouk Lee, "DIVDS: docker image vulnerability diagnostic system," *IEEE Access*, vol. 8, pp. 42 666–42 673, 2020. DOI: 10.1109/ACCESS.2020.2976874.

[82] Butler W. Lampson, "A Note on the Confinement Problem," *Communications of the ACM*, vol. 16, no. 10, pp. 613–615, 1973, ISSN: 0001-0782. DOI: 10.1145/362375.362389.

[83] Michael Larabel and Matthew Tippett, *Phoronix Test Suite*, 2011. [Online]. Available: http://www.phoronix-test-suite.com (visited on 12/23/2020).

[84] Hojoon Lee, Chihyun Song, and Brent ByungHoon Kang, "Lord of the x86 Rings: A Portable User Mode Privilege Separation Architecture on x86," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, October 15-19, 2018*, David Lie, Mohammad Mannan, *et al.*, Eds., ACM, 2018, pp. 1441–1454. DOI: 10.1145/3243734.3243748.

[85] Lingguang Lei, Jianhua Sun, *et al.*, "SPEAKER: Split-Phase Execution of Application Containers," in *Detection of Intrusions and Malware, and Vulnerability Assessment - 14th International Conference*, ser. Lecture Notes in Computer Science, vol. 10327, Springer, 2017, pp. 230–251. DOI: 10.1007/978-3-319-60876-1_11.

[86] *libbpf*, GitHub repository. [Online]. Available: https://github.com/libbpf/libbpf (visited on 12/13/2020).

[87] *libseccomp*. [Online]. Available: https://github.com/seccomp/libseccomp (visited on 06/22/2021).

[88] Xin Lin, Lingguang Lei, *et al.*, "A Measurement Study on Linux Container Security: Attacks and Countermeasures," in *Proceedings of the 34th Annual Computer Security Applications Conference*, ser. ACSAC '18, San Juan, PR, USA: Association for Computing Machinery, 2018, pp. 418–429, ISBN: 9781450365697. DOI: 10.1145/3274694.3274720.

[89] Linux Foundation, *Open Container Initiative*, 2020. [Online]. Available: https://opencontainers.org (visited on 12/20/2020).

[90] Benjamin Livshits, "Dynamic Taint Tracking in Managed Runtimes," Microsoft Research, Tech. Rep. MSR-TR-2012-114, 2012. [Online]. Available: https://www.microsoft.com/en-us/research/wp-content/uploads/2016/02/tr-6.pdf.

[91] LLVM, *BPF Directory Reference*, Developer documenation. [Online]. Available: https://llvm.org/doxygen/dir_b9f4b12c13768d2acd91c9fc79be9cbf.html (visited on 12/13/2020).

[92] Peter Loscocco and Stephen Smalley, "Integrating Flexible Support for Security Policies into the Linux Operating System," in *Proceedings of the FREENIX Track: 2001 USENIX Annual Technical Conference, June 25-30, 2001, Boston, Massachusetts, USA*, Clem Cole, Ed., USENIX, 2001, pp. 29–42. [Online]. Available: http://www.usenix.org/publications/library/proceedings/usenix01/freenix01/loscocco.html.

[93] Fotis Loukidis-Andreou, Ioannis Giannakopoulos, *et al.*, "Docker-Sec: A Fully Automated Container Security Enhancement Mechanism," in *38th IEEE International Conference on Distributed Computing Systems*, IEEE Computer Society, 2018, pp. 1561–1564. DOI: 10.1109/ICDCS.2018.00169.

[94] LXC Developers, *LXC Security*, Developer documenation. [Online]. Available: https://linuxcontainers.org/lxc/security/ (visited on 07/02/2021).

[95] Karl MacMillan, "Madison: A New Approach to Policy Generation," in *SELinux Symposium*, 2007. [Online]. Available: http://selinuxsymposium.org/2007/papers/08-polgen.pdf.

[96] Ziqing Mao, Ninghui Li, *et al.*, "Trojan Horse Resistant Discretionary Access Control," in *14th ACM Symposium on Access Control Models and Technologies, SACMAT 2009, Stresa, Italy, June 3-5, 2009, Proceedings*, Barbara Carminati and James Joshi, Eds., ACM, 2009, pp. 237–246. DOI: 10.1145/1542207.1542244.

[97] Ashraf Matrawy, Paul C Van Oorschot, and Anil Somayaji, "Mitigating Network Denial-of-Service Through Diversity-Based Traffic Management," in *International Conference on Applied Cryptography and Network Security*, Springer, 2005, pp. 104–121.

[98] Steven McCanne and Van Jacobson, "The BSD Packet Filter: A New Architecture for User-level Packet Capture," *USENIX Winter*, vol. 93, 1993. [Online]. Available: https://www.tcpdump.org/papers/bpf-usenix93.pdf.

[99] Lee D. McFearin, "Chroot Jail," in *Encyclopedia of Cryptography and Security, 2nd Ed*, Henk C. A. van Tilborg and Sushil Jajodia, Eds., Springer, 2011, pp. 206–207. DOI: 10.1007/978-1-4419-5906-5_778.

[100] Pual McKenney, "What is RCU, Fundamentally?" *LWN.net*, Dec. 2007. [Online]. Available: https://lwn.net/Articles/262464/.

[101] Samuel P. Mullinix, Erikton Konomi, *et al.*, "On Security Measures for Containerized Applications Imaged with Docker," *CoRR*, vol. abs/2008.04814, 2020. [Online]. Available: https://arxiv.org/abs/2008.04814.

[102] Andrii Nakryiko, *BPF Portability and CO-RE*, Blog post, Feb. 2020. [Online]. Available: https://facebookmicrosites.github.io/bpf/blog/2020/02/19/bpf-portability-and-co-re.html (visited on 06/07/2021).

[103] Andrii Nakryiko, "bpf: add BPF ringbuf and perf buffer benchmarks," Kernel patch, May 2020. [Online]. Available: https://patchwork.ozlabs.org/project/netdev/patch/20200529075424.3139988-5-andriin%5C@fb.com/.

[104] *namespaces(7)*, Linux Programmer's Manual. [Online]. Available: https://man7.org/linux/man-pages/man7/namespaces.7.html (visited on 06/26/2021).

[105] Saran Neti, Anil Somayaji, and Michael E Locasto, "Software Diversity: Security, Entropy and Game Theory," in *HotSec*, 2012.

[106] Jiri Olsa, "bpf: Add d_path helper," Kernel patch, Aug. 2020. [Online]. Available: https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=6e22ab9da79343532cd3cde39df25e5a5478c692.

[107] Paul C. van Oorschot, *Computer Security and the Internet - Tools and Jewels*, ser. Information Security and Cryptography. Springer, 2020, ISBN: 978-3-030-33648-6. DOI: 10.1007/978-3-030-33649-3.

[108] Pradeep Padala, "Playing with ptrace, Part I," *Linux Journal*, vol. 2002, no. 103, p. 5, 2002. [Online]. Available: https://www.linuxjournal.com/article/6100.

[109] Daniel Paulus, *The Rego Language*, Blog post, 2020. [Online]. Available: https://danielpaulus.com/the-rego-language/ (visited on 07/19/2021).

[110] Christopher J. PeBenito, Frank Mayer, and Karl MacMillan, "Reference Policy for Security Enhanced Linux," in *SELinux Symposium*, 2006. [Online]. Available: http://selinuxsymposium.org/2006/papers/05-refpol.pdf.

[111] Bheesham Persaud, Borke Obada-Obieh, *et al.*, "Frankenssl: Recombining Cryptographic Libraries for Software Diversity," in *Proceedings of the 11th Annual Symposium On Information Assurance. NYS Cyber Security Conference*, 2016, pp. 19–25.

[112] *pledge(2)*, OpenBSD system calls manual, Jul. 2020. [Online]. Available: https://man.openbsd.org/pledge (visited on 06/21/2021).

[113] Tom Preston-Werner, *TOML: Tom's Obvious Minimal Language*, version 1.0.0, Official specification, 2021. [Online]. Available: https://toml.io/en/v1.0.0 (visited on 07/16/2021).

[114] Daniel Price and Andrew Tucker, "Solaris Zones: Operating System Support for Consolidating Commercial Workloads," in *Proceedings of the 18th Conference on Systems Administration (LISA 2004), Atlanta, USA, November 14-19, 2004*, Lee Damon, Ed., USENIX, 2004, pp. 241–254. [Online]. Available: http://www.usenix.org/publications/library/proceedings/lisa04/tech/price.html.

[115] Niels Provos, "Improving Host Security with System Call Policies," in *Proceedings of the 12th USENIX Security Symposium, Washington, D.C., USA, August 4-8, 2003*, USENIX Association, 2003. [Online]. Available: https://www.usenix.org/conference/12th-usenix-security-symposium/improving-host-security-system-call-policies.

[116] *ptctl(2)*, Linux user's manual. [Online]. Available: https://linux.die.net/man/2/prctl.

[117] *ptrace(2)*, Linux user's manual. [Online]. Available: https://linux.die.net/man/2/ptrace.

[118] Redbpf Authors, *redbpf*, GitHub repository. [Online]. Available: https://github.com/foniod/redbpf (visited on 06/07/2021).

[119] Dennis Ritchie and Ken Thompson, "The UNIX Time-Sharing System," *Commun. ACM*, vol. 17, no. 7, pp. 365–375, 1974. DOI: 10.1145/361011.361061.

[120] Goldwyn Rodrigues, "[PATCH] AppArmor: Fix Unnecessary Creation of Net-Compat," Kernel patch, Jun. 2018. [Online]. Available: https://raw.githubusercontent.com/openSUSE/kernel-source/master/patches.suse/0001-apparmor-fix-unnecessary-creation-of-net-compat.patch.

[121] Mickael Salaun, "Landlock LSM: Toward Unprivileged Sandboxing," Kernel patch RFC, 2017. [Online]. Available: https://lkml.org/lkml/2017/8/20/192 (visited on 12/17/2020).

[122] Mickael Salaun, *landlock.io*, 2020. [Online]. Available: https://landlock.io (visited on 12/17/2020).

[123] Casey Schaufler, "The Simplified Mandatory Access Control Kernel, Smack White Paper," Whitepaper. [Online]. Available: http://schaufler-ca.com/yahoo_site_admin/assets/docs/SmackWhitePaper.257153003.pdf (visited on 06/18/2021).

[124] Fred B. Schneider, "Least Privilege and More," *IEEE Secur. Priv.*, vol. 1, no. 5, pp. 55–59, 2003. DOI: 10.1109/MSECP.2003.1236236.

[125] Z. Cliffe Schreuders, Tanya Jane McGill, and Christian Payne, "Towards Usable Application-Oriented Access Controls," in *International Journal of Information Security and Privacy*, vol. 6, 2012, pp. 57–76. DOI: 10.4018/jisp.2012010104.

[126] *seccomp(2)*, Linux user's manual. [Online]. Available: https://man7.org/linux/man-pages/man2/seccomp.2.html.

[127] Rui Shu, Xiaohui Gu, and William Enck, "A Study of Security Vulnerabilities on Docker Hub," in *Proceedings of the Seventh ACM Conference on Data and Application Security and Privacy*, ACM, 2017, pp. 269–280. DOI: 10.1145/3029806.3029832.

[128] Rui Shu, Peipei Wang, *et al.*, "A Study of Security Isolation Techniques," *ACM Computing Surveys*, vol. 49, no. 3, pp. 1–37, 2016. DOI: 10.1145/2988545. (visited on 07/27/2020).

[129] KP Singh, "MAC and Audit Policy Using eBPF (KRSI)," Kernel patch, 2019. [Online]. Available: https://lwn.net/ml/linux-kernel/20191220154208.15895-1-kpsingh@chromium.org/.

[130] Stephen Smalley, Chris Vance, and Wayne Salamon, "Implementing SELinux as a Linux security module," 43, vol. 1, 2001, p. 139. [Online]. Available: https://www.cs.unibo.it/~sacerdot/doc/so/slm/selinux-module.pdf.

[131] Justin R. Smith, Yuichi Nakamura, and Dan Walsh, *audit2allow(1)*, Linux user's manual. [Online]. Available: http://linux.die.net/man/1/audit2allow.

[132] Snapcraft, *Security Policy and Sandboxing*, 2020. [Online]. Available: https://snapcraft.io/docs/security-sandboxing (visited on 07/02/2021).

[133] Brian T. Sniffen, David R. Harris, and John D. Ramsdell, "Guided Policy Generation for Application Authors," in *SELinux Symposium*, 2006. [Online]. Available: http://gelit.ch/td/SELinux/Publications/Mitre_Tools.pdf.

[134] Jon A. Solworth and Robert H. Sloan, "A Layered Design of Discretionary Access Controls with Decidable Safety Properties," in *2004 IEEE Symposium on Security and Privacy (S&P 2004), 9-12 May 2004, Berkeley, CA, USA*, IEEE Computer Society, 2004, p. 56. DOI: 10.1109/SECPRI.2004.1301315.

[135] Anil Somayaji, "Immunology, Diversity, and Homeostasis: The Past and Future of Biologically Inspired Computer Defenses," *Information Security Technical Report*, vol. 12, no. 4, pp. 228–234, 2007.

[136] Anil Somayaji and Stephanie Forrest, "Automated Response Using System-Call Delay," in *9th USENIX Security Symposium, Denver, Colorado, USA, August 14-17, 2000*, Steven M. Bellovin and Greg Rose, Eds., USENIX Association, 2000. [Online]. Available: https://www.usenix.org/conference/9th-usenix-security-symposium/automated-response-using-system-call-delay.

[137] Ray Spencer, Stephen Smalley, *et al.*, "The Flask Security Architecture: System Support for Diverse Security Policies," in *Proceedings of the 8th USENIX Security Symposium, Washington, DC, USA, August 23-26, 1999*, G. Winfield Treese, Ed., USENIX Association, 1999. [Online]. Available: https://www.usenix.org/conference/8th-usenix-security-symposium/flask-security-architecture-system-support-diverse-security.

[138] Alexei Starovoitov, "Merge branch 'bpf-sleepable'," Kernel patch, Aug. 2020. [Online]. Available: https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=10496f261ed30592c6a7f8315f6b5ec055db624a.

[139] Alexei Starovoitov and Daniel Borkmann, "Rework/optimize internal BPF interpreter's instruction set," Kernel patch, Mar. 2014. [Online]. Available: https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=bd4cf0ed331a275e9bf5a49e6d0fd55dffc551b8.

[140] Sari Sultan, Imtiaz Ahmad, and Tassos Dimitriou, "Container Security: Issues, Challenges, and the Road Ahead," *IEEE Access*, vol. 7, pp. 52 976–52 996, 2019. DOI: 10.1109/ACCESS.2019.2911732.

[141] Yuqiong Sun, David Safford, *et al.*, "Security Namespace: Making Linux Security Frameworks Available to Containers," in *27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15-17, 2018*, William Enck and Adrienne Porter Felt, Eds., USENIX Association, 2018, pp. 1423–1439. [Online]. Available: https://www.usenix.org/conference/usenixsecurity18/presentation/sun.

[142] Robert Swiecki, *Promises and Pitfalls of Sandboxes*, Presentation slides, 2017. [Online]. Available: http://www.swiecki.net/resources/Promises%20and%20pitfalls%20of%20sandboxes.pdf (visited on 06/22/2021).

[143] Tcpdump Authors, *Tcpdump*. [Online]. Available: https://www.tcpdump.org/ (visited on 06/07/2020).

[144] Alexander Tereshkin and Rafal Wojtczuk, "Introducing Ring-3 Rootkits," in *Black Hat USA*, Presentation slides, 2009. [Online]. Available: https://www.blackhat.com/presentations/bh-usa-09/TERESHKIN/BHUSA09-Tereshkin-Ring3Rootkit-SLIDES.pdf (visited on 07/08/2021).

[145] Erick Tryzelaar, David Tolnay, and Serde Contributors, *Serde*, Official documentation, 2021. [Online]. Available: https://serde.rs/ (visited on 07/16/2021).

[146] Dan Tsafrir, Dilma Da Silva, and David A. Wagner, "The murky issue of changing process identity: Revising "setuid demystified"," *login Usenix Mag.*, vol. 33, no. 3, 2008. [Online]. Available: https://www.usenix.org/publications/login/june-2008-volume-33-number-3/murky-issue-changing-process-identity-revising.

[147] *uapi/linux/capability.h*, Linux Kernel sources. [Online]. Available: https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/include/uapi/linux/capability.h?h=v5.12 (visited on 06/15/2021).

[148] *unveil(2)*, OpenBSD system calls manual, Apr. 2020. [Online]. Available: https://man.openbsd.org/unveil (visited on 06/21/2021).

[149] US Department of Defense, "Trusted Computer System Evaluation Criteria," DOD Standard DOD 5200.58-STD, 1983.

[150] Victor A. Vyssotsky, Fernando J. Corbató, and Robert M. Graham, "Structure of the Multics Supervisor," in *Proceedings of the 1965 fall joint computer conference, part I, AFIPS 1965 (Fall, part I), Las Vegas, Nevada, USA, November 30 - December 1, 1965*, Robert W. Rector, Ed., ACM, 1965, pp. 203–212. DOI: 10.1145/1463891.1463914.

[151] David A. Wagner, "Janus: An Approach for Confinement of Untrusted Applications," M.S. thesis, University of California, Berkeley, 1999. [Online]. Available: https://www2.eecs.berkeley.edu/Pubs/TechRpts/1999/CSD-99-1056.pdf.

[152] Robert N. M. Watson, Jonathan Anderson, *et al.*, "Capsicum: Practical Capabilities for UNIX," in *19th USENIX Security Symposium, Washington, DC, USA, August 11-13, 2010, Proceedings*, USENIX Association, 2010, pp. 29–46. [Online]. Available: https://www.usenix.org/legacy/event/sec10/tech/full_papers/Watson.pdf.

[153] Robert N. M. Watson, Jonathan Woodruff, *et al.*, "CHERI: A Hybrid Capability-System Architecture for Scalable Software Compartmentalization," in *2015 IEEE Symposium on Security and Privacy, SP 2015, San Jose, CA, USA, May 17-21, 2015*, IEEE Computer Society, 2015, pp. 20–37. DOI: 10.1109/SP.2015.9.

[154] Chris Wright, Crispin Cowan, *et al.*, "Linux Security Modules: General Security Support for the Linux Kernel," in *Proceedings of the 11th USENIX Security Symposium, San Francisco, CA, USA, August 5-9, 2002*, Dan Boneh, Ed., USENIX, 2002, pp. 17–31. [Online]. Available: http://www.usenix.org/publications/library/proceedings/sec02/wright.html.

[155] Daniel Xu, *libbpf-rs*, GitHub repository. [Online]. Available: https://github.com/libbpf/libbpf-rs (visited on 05/28/2021).

[156] Heng Yin, Dawn Xiaodong Song, *et al.*, "Panorama: Capturing System-Wide Information Flow for Malware Detection and Analysis," in *Proceedings of the 2007 ACM Conference on Computer and Communications Security, CCS 2007, Alexandria, Virginia, USA, October 28-31, 2007*, Peng Ning, Sabrina De Capitani di Vimercati, and Paul F. Syverson, Eds., ACM, 2007, pp. 116–127. DOI: 10.1145/1315245.1315261.

[157] Angeliki Zavou, Georgios Portokalidis, and Angelos D. Keromytis, "Taint-Exchange: A Generic System for Cross-Process and Cross-Host Taint Tracking," in *Advances in Information and Computer Security - 6th International Workshop, IWSEC 2011, Tokyo, Japan, November 8-10, 2011. Proceedings*, Tetsu Iwata and Masakatsu Nishigaki, Eds., ser. Lecture Notes in Computer Science, vol. 7038, Springer, 2011, pp. 113–128. DOI: 10.1007/978-3-642-25141-2_8.

[158] Lei Zeng, Yang Xiao, and Hui Chen, "Auditing Overhead, Auditing Adaptation, and Benchmark Evaluation in Linux," *Secur. Commun. Networks*, vol. 8, no. 18, pp. 3523–3534, 2015. DOI: 10.1002/sec.1277.

[159] Wenhui Zhang, Peng Liu, and Trent Jaeger, "Analyzing the Overhead of File Protection by Linux Security Modules," in *ASIA CCS '21: ACM Asia Conference on Computer and Communications Security, Virtual Event, Hong Kong, June 7-11, 2021*, Jiannong Cao, Man Ho Au, *et al.*, Eds., ACM, 2021, pp. 393–406. DOI: 10.1145/3433210.3453078.

[160] David (Yu) Zhu, Jaeyeon Jung, *et al.*, "TaintEraser: Protecting Sensitive Data Leaks Using Application-Level Taint Tracking," *ACM SIGOPS Oper. Syst. Rev.*, vol. 45, no. 1, pp. 142–154, 2011. DOI: 10.1145/1945023.1945039.

[161] Jörg Zinke, "System Call Tracing Overhead," in *The International Linux System Technology Conference (Linux Kongress)*, Presentation slides, 2009. [Online]. Available: http://www.linux-kongress.org/2009/slides/system_call_tracing_overhead_joerg_zinke.pdf (visited on 06/22/2021).

Appendix A.

List of Acronyms

ABI Application Binary Interface. 42, 169

ACL Access Control List. 21, 23, 28, 29

API Application Programming Interface. 27, 40, 65, 70, 71, 76, 131

ASLR Address Space Layout Randomization. 18, 156

BPF Berkeley Packet Filter. 40, 48, 57–67, 69, 70, 72, 73, 80, 90, 96–101, 123, 125–130, 150, 161, 163, 166, 212–215, 225, 227

BTF BPF Type Format. 70, 130, 150

cBPF Classic BPF. 40, 57, 58

CGI Common Gateway Interface. 187, 224

CO-RE Compile Once, Run Everywhere. 70, 71, 90, 123, 125, 128, 150, 215, 225

COTS Commercial Off-The-Shelf. 173

CPU Central Processing Unit. 18, 47, 58, 85, 100, 109, 142, 179

DAC Discretionary Access Control. 7, 22, 24–27, 29, 34, 52, 80–82, 145

eBPF Extended BPF. iv, 3, 4, 8–11, 35, 38, 56, 57, 60–74, 83, 87, 89, 92, 95–99, 102–104, 109, 117, 121, 123–125, 127–130, 132–134, 138–141, 153, 161, 170, 175, 176, 178, 179, 202, 207–216, 218–221, 224–227

EGID Effective Group ID. 22

EUID Effective User ID. 22, 25

FPGA Field-Programmable Gate Array. 42

GID Group ID. 20, 21, 24, 26, 47

IP Internet Protocol. 33, 147, 148, 180, 218

IPC Inter-Process Communication. 33, 46, 52, 91, 102, 133, 134, 136, 148, 149, 180–184, 186, 190–193, 196, 202

ISA Instruction Set Architecture. 18, 41, 42

JIT Just-In-Time. 61

KASLR Kernel ASLR. 18

KRSI Kernel Runtime Security Instrumentation. 66, 95, 96, 125, 134, 159, 161, 216

LSM Linux Security Modules. 7, 31–36, 47, 52, 53, 55, 65–68, 81, 82, 87, 88, 92, 95, 96, 100, 102–104, 124–126, 129, 130, 134, 135, 154, 157, 158, 161, 164–166, 171, 172, 175, 178, 179, 183, 184, 209, 213, 216, 219–221, 225

LXC Linux Containers. 2, 52

MAC Mandatory Access Control. 29, 30, 48, 52, 55, 80, 81, 89, 154, 172, 221

MLS Multi-Level Security. 29–31

MMU Memory Management Unit. 17, 18

OCI Open Container Initiative. 151, 217, 218

OS Operating System. 14, 15, 17, 19, 48, 50, 77–79, 155, 157, 174, 222

PID Process ID. 41, 47, 52, 85, 134, 188

TCB Trusted Computing Base. 25

TID Thread ID. 105

TLB Translation Lookaside Buffer. 17

TOCTTOU Time of Check to Time of Use. 40, 146, 175

UID User ID. 20–22, 24–27, 47, 82, 149

USDT User Statically Defined Tracepoints. 65, 101, 102, 134

UTS Unix Timesharing System. 52

VFS Virtual Filesystem. 175